Architectural Modeling and Benchmarking for Digital DRAM PIM

Farzana Ahmed Siddique, Deyuan Guo, Zhenxing Fan, Mohammadhosein Gholamrezaei, Morteza Baradaran, Alif Ahmed, Hugo Abbot, Kyle Durrer, Kumaresh Nandagopal, Ethan Ermovick, Khyati Kiyawat, Beenish Gul, Abdullah Mughrabi, Ashish Venkat, Kevin Skadron
Department of Computer Science, University of Virginia
{farzana, dg7vp, fjy3ws, uab9qt, rgq5aw, alifahmed, drt3sm, kmd2zjw, tsx4uc, keg9ve, vyn9mp, bg9qq, cmv6ru, venkat, skadron}@virginia.edu

2024 IEEE International Symposium on Workload Characterization (IISWC) | 979-8-3503-5603-8/24/$31.00 ©2024 IEEE | DOI: 10.1109/IISWC63097.2024.00030

***Abstract*—Processing In Memory (PIM) integrates computational logic units directly into the memory architecture, offering significant performance improvements for memory-bound applications such as matrix operations, vector operations, and database applications compared to general-purpose CPUs or GPUs. However, the lack of a standardized benchmark suite and simulation framework poses a challenge in exploring, evaluating, and designing different PIM architectures. This paper addresses this gap by introducing a comprehensive benchmark suite, *PIMbench*, along with a performance and energy modeling framework, *PIMeval*, both designed to support a wide range of PIM architectures for DRAM. This paper also proposes a set of PIM APIs for writing PIM programs, enabling benchmarks to be executed across different PIM architectures, and allowing for comparing the performance of bit-serial and bit-parallel subarray-level PIM and bank-level PIM. PIMbench and PIMeval have been open-sourced and can be accessed at: https://github.com/UVA-LavaLab/PIMeval-PIMbench**

***Index Terms*—Processing in memory, benchmarks**

I. Introduction

DRAM [29] is a widely used memory technology consisting of multiple banks, each with multiple subarrays, each with a wide local row buffer. All of these locations can host some processing capability; however, due to pinout and signaling constraints, the channel's I/O interface is quite narrow, hiding this massive parallelism from the host CPU or GPU. Processing in Memory (PIM) integrates computation within memory, eliminating unnecessary data movement overhead between memory and the host, and also enabling high degrees of parallel processing by leveraging the inherent parallelism of the memory architecture. The concept of PIM has existed for decades [67], but recent developments, particularly the slowing of Moore's Law [46], have renewed interest in PIM. These architectures have demonstrated significant potential in enhancing performance and energy efficiency across various computing domains, leading to numerous proposed PIM architectures [9], [16], [26], [36], [37], [59], [62], [72]. However, the absence of a PIM-specific, generalized benchmark suite and modeling framework that are portable across different PIM architectures makes comparisons challenging and inhibits innovations in this promising architectural design space. This paper addresses these challenges by proposing a flexible PIM modeling framework, *PIMeval*, accompanied by a PIM benchmark suite, *PIMbench*, and a programming framework, the *PIM API*.

PIMeval. Evaluating PIM architectures is difficult due to the lack of flexible modeling tools. Prior PIM studies introduce their own custom simulation or modeling techniques. These include: i) coarse-grained analytical models, [64] ii) trace-driven simulations, e.g., [14] and iii) extending full-system simulators such as gem5 [4] to model-specific PIM architectures [56]. While these methods are effective for the particular architecture they target, they are not designed for portability. This paper introduces *PIMeval*, which supports modeling both performance and energy for a diverse range of PIM architectures. This paper demonstrates PIMeval's versatility by modeling three potential placement options at different levels of the DRAM hierarchy, as shown in Figure 2: at the bank level, a single processing element at the side of each subarray, as in Fulcrum [37]; and digital bit-serial processing capability associated with each sense amplifier across the width of each row buffer, similar to DRISA [38] and Micron's digital In-Memory Intelligence (IMI) [17]. Although prior work has explored the capabilities of analog bit-serial [26], [62], digital bit-serial techniques are less vulnerable to variations due to manufacturing and aging. Therefore, for modeling bit-serial PIM, we explore a digital technique that supports Boolean operations for general computation, and also incorporates native support for comparisons, similar to DRAM-CAM [73], an architecture we call DRAM-AP.

PIMbench. This paper also introduces a new benchmark suite for PIM, which we call *PIMbench*. It incorporates a diverse set of applications (Figure 1), each implemented using a high-level *PIM API* designed to be portable across varying PIM architectures. While prior PIM benchmark suites, such as PrIM [24] and InSituBench [37] exist, they are specifically built for a single architecture and cannot be easily ported to evaluate other PIM architectures. Our proposed PIMbench suite, shown in Table I, adapts some benchmarks from these previous suites but also introduces a wider variety of applications and domains. This includes benchmarks with

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.

2835-2238/24/$31.00 ©2024 IEEE
DOI 10.1109/IISWC63097.2024.00030

TABLE I: PIMbench Suite.

Domain	Application Name	Memory Access Pattern		Execution Type	Input
		Sequential	Random		
Linear Algebra	Vector Addition	✓		PIM	2,035,544,320 32-bit INT
	AXPY	✓		PIM	16,777,216 32-bit INT
	Matrix-Vector Mult. (GEMV)	✓		PIM	$2,352,160 \times 8,192$ 32-bit INT
	Matrix-Matrix Mult. (GEMM)	✓		PIM	$23,521 \times 4,096$ and $4,096 \times 512$ 32-bit INT
Sort	Radix Sort	✓	✓	PIM + Host	67,108,864 32-bit INT
Cryptography	AES-Encryption	✓	✓	PIM	1,035,544,320 Bytes
	AES-Decryption	✓	✓	PIM	1,035,544,320 Bytes
Graph	Triangle Count	✓	✓	PIM	227,320 nodes and 1,628,268 edges
Database	Filter-By-Key	✓		PIM + Host	1,073,741,824 key-value pairs
Image Processing	Histogram	✓		PIM	1.4×10^9 24-bit .bmp
	Brightness	✓		PIM	
	Image Down Sampling	✓		PIM	
Supervised Learning	K-nearest neighbors (KNN)	✓	✓	PIM + Host	6,710,886 2D data points
	Linear Regression	✓		PIM	1,500,000,000 2D points
Unsupervised Learning	K-means	✓	✓	PIM	67,108,864 2D data, $k = 20$
Neural Network	VGG-13	✓		PIM + Host	64, $224X224X3$ image matrix and 3X3X64 weight matrix
	VGG-16	✓		PIM + Host	
	VGG-19	✓		PIM + Host	

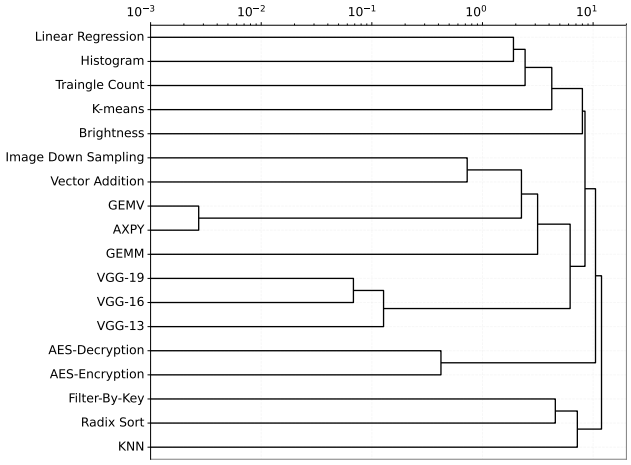


Fig. 1: Dendogram showing similarity between benchmarks in PIMbench. X-axis shows linkage distance in log scale.

more complex execution patterns, such as AES [51], and those involving PIM-host communication, such as radix sort and VGG [65]. Importantly, our benchmark suite does not focus on highlighting a single architecture, but rather showcases the strengths and weaknesses of each architecture explored.

In summary, this paper proposes an evaluation framework and benchmark suite that provide a common ground for evaluating and comparing PIM architectures, with the goal of making it easier for the architecture research community to explore the PIM design space.

II. Related Work

PIM Modeling and Simulation. DRAMsim3 [39] and Ramulator [33] are cycle-accurate DRAM simulators that focus on DRAM protocols and do not support PIM-specific simulation, limiting their use for PIM architecture evaluations—although both can be used to help model the timing of the DRAM read/write operations involved in PIM [14], [27]. In contrast, PIMSim [74] allows a program to be annotated with PIM instructions, which invoke a PIM performance model. The PIM performance models are based on the processor types modeled in gem5 [4], such as CPU and GPU cores. However, such heavyweight cores likely only make sense at the bank interface or outside the DRAM chip in a *near-memory* configuration, thus rendering them unsuitable for modeling subarray-level PIM architectures. gem5 is a versatile and widely used simulation infrastructure for CPU and GPU modeling, and provides a detailed model of the memory system for conventional reads and writes. While some PIM research, such as MIMDRAM [56], utilizes gem5, their approach lacks flexibility and extensibility for diverse PIM architectures. Incorporating PIMeval into gem5 is a valuable direction for future research.

PiMulator [47] is an FPGA-based platform for prototyping and evaluating PIM architectures. It offers detailed memory configuration but does not support diverse PIM functionality, due to which mapping a new PIM architecture to PiMulator is often a Herculean effort requiring extensive modifications within the FPGA framework.

MultiPIM [76] is intended to be a flexible PIM simulation framework, but assumes the same instruction set architecture (ISA) for both PIM and host cores, which limits its ability to explore PIM architectures with a different instruction set.

In all the existing frameworks, adding a new application requires coding the new application at an assembly level or similar low-level representation. In contrast, our PIM API allows applications to be written in a high-level language, abstracting away low-level hardware details and ensuring portability across different PIM architectures. Targeting this API as an intermediate representation with a compiler would be an interesting direction for future work.

Several DRAM power models have been proposed. PIMeval's approach is primarily based on the Micron power model [45] because it is based on vendor data and is straightforward to incorporate into PIMeval and derive energy for each PIM operation. Other open-source models include CACTI [2], DRAMPower [31], VAMPIRE [22], and the RAMBUS power model [68].

Benchmark Suites for PIM. PrIM [24] and InSituBench [37] are two open-source benchmark suites for PIM architectures. PrIM is specifically designed for UPMEM [24], [34], and provides mostly building-block benchmarks, while

InSituBench provides some mixture in terms of benchmark complexity, but primarily targets kernels for subarray-level PIM [35]–[37]. Neither suite is easily modified to study diverse PIM architectures. A previous generation of PIM research named IRAM project, proposed a small suite of five benchmarks [20] with diverse characteristics, but it does not appear to have been open-sourced. A version of this suite's histogram benchmark was also included in the Phoenix benchmark suite for map-reduce processing [75]. BLIMP [14] introduces compiler analysis to map the Phoenix and SPECcpu benchmarks to a bank-level PIM architecture 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.

PIMbench offers a broader range of applications, as illustrated in Figure 1, 8, including some adapted from PrIM, InSituBench, and Phoenix, as well as applications used in the evaluation of the SIMDRAM analog bit-serial [26] architecture and the DRISA [38] hybrid analog/digital bit-serial architecture. The benchmarks currently included in PIMbench, as listed in Table I, range from simple building-block benchmarks, like vector addition and AXPY, to benchmarks comprising multiple building blocks that require data re-layout between each kernel execution and PIM-host communication. Some of PIMbench's benchmarks were included in multiple prior suites, such as GEMV (PrIM and InSituBench), GEMM (InSituBench, Phoneix), filter-by-key (PrIM, InSituBench, and related to bitweaving in SIMDRAM), histogram (PrIM, IRAM, Phoneix), KNN (InSituBench, SIMDRAM), and VGG (SIMDRAM, DRISA). Others only appear in one suite, such as vector addition (PrIM), AXPY (InSituBench), radix sort (from follow-on work to InSituBench [35]), brightness (SIMDRAM), linear regression (Phoenix), K-means (Phoenix); and others have not been evaluated in prior PIM work to the best of our knowledge but we felt added significant value, namely triangle counting, image downsampling, and AES encryption/decryption.

Overall, PIMbench provides a greater range of building blocks used in modern applications, and is therefore better suited to evaluate how PIM may handle more realistic workloads, and helps illustrate pros and cons of diverse PIM architectures on different application behaviors. Lastly, our benchmarks are implemented using a common API, which allows them to be portable to future PIM architectures.

We are continuing to extend PIMbench with additional kernels, such as prefix sum (related to scan from PrIM and InSituBench), transitive closure from the IRAM suite, Principal Component Analysis (PCA) [42] and string match from Phoenix, apriori from DRAM-CAM [73], additional machine-learning algorithms, sparse algorithms such as sparse matrix-vector multiply (not easily supported in bit-serial PIM) and graph algorithms.

III. DRAM Background

DDR (Double Data Rate) DRAM [29] is the dominant, commodity memory technology in modern computing. In this paper, we focus on PIM for DDR; our modeling approach and benchmarks should be easily extensible to High Bandwidth Memory (HBM) [30], which is left for future work, although conclusions about which PIM architecture is best might change with HBM.

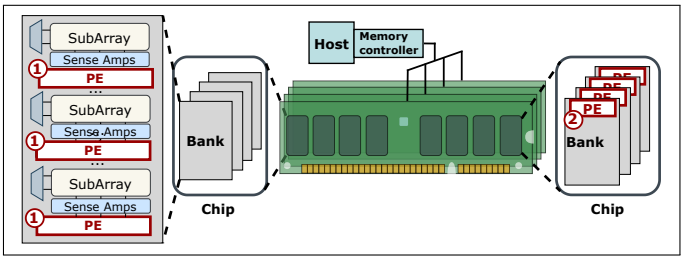


Fig. 2: DDR DRAM organization showing potential locations for PIM processing elements.

DRAM is organized hierarchically, with individual memory bit-cells grouped into successively larger units, up to the memory modules placed into the motherboard, called Dual In-line Memory Modules (DIMMs), as shown in Figure 2. Memory access occurs through multiple channels, which operate concurrently, and each channel has its own address, data, and command bus, allowing channels to operate completely independently. Multiple DIMMs can be connected to a single channel, and a DIMM consists of one or more ranks, where each rank comprises a set of DRAM chips that each contribute a subset of the bits needed for a single memory fetch from the memory controller.

Within each chip, memory is organized into banks, typically 16 or more per chip. Logically, a specific bank position in each chip in a rank will be grouped together to form a logical bank, so when the memory controller reads or writes to a given rank, it specifies one of these logical banks (e.g., a cache line fetch from bank 0 will fetch some bits from the same position in bank 0 of chip 0, bank 0 of chip 1, etc.). DRAM operations consist of read/write commands, which actually move data, and other operations, such as precharging the bitlines, or activating a row. Memory commands are sent to the appropriate bank, and operations to these logical banks can be interleaved, so that for example one bank can be precharging while another is providing data. Banks in turn are divided into subarrays, which are smaller groups of memory cells, to reduce electrical loads and noise. Each subarray within a bank has its own row decoder and local sense amplifiers, and the set of local sense amplifiers forms the subarray's local row buffer. Subarrays are typically 512-2K rows tall, and 4K-16K columns wide.

A subarray row activation reads an entire row and latches these bits into the row buffer. Successive accesses to the same row can therefore fetch data from the row buffer without the much higher latency and energy cost of closing a row (which requires writing back its values; row activation is destructive) and activating a new row. These successive accesses to the same row are called row-buffer or page hits. Reads fetch 64-256 bits from the open local row buffer over the *global data lines* (GDL) into the bank's global row buffer, which are then bursted out over the I/O pins; writes operate in the

opposite direction. For DDR, the GDL is likely 64–128 bits, while for HBM it is wider. Note that this narrow GDL limits the potential PIM bandwidth at the bank interface.

Subarrays in turn consist of MATs, which are roughly square, e.g. 512x512; and each MAT contributes one or more bits to the read. This means logically adjacent bits in a word of data (assuming traditional horizontal, row-oriented data layout) are not adjacent; they are scattered across MATs and only reunited in the global row buffer at the bank interface. Each MAT has its own subwordline driver, to boost the wordline signal, so that the overall subararay wordline does not need a single, large driver. MATs also allow the logic for column selection for reads and writes to be distributed. At this point in time, and for purposes of this paper, PIMeval only models subarrays as monolithic arrays, without the added detail of MATs, which is left for future work. However, as shown in the MIMDRAM [56] project, MATs can play an important role in PIM architecture. The Fulcrum [37] project also noted that the non-adjacency of a word's individual bits in the row buffer was a motivation for its architecture.

For a more in-depth overview of DRAM architecture, see Vogelsang [68], Zhang [77], Seshadri [63] or Marazzi [44]. Jacob [29] and the DRAMsim3 memory model [39] are also excellent resources, although they do not discuss subarrays.

For this study, we assume each rank consists of 8 chips with an 8-bit interface ("x8"). Each chip has 16 banks, and each bank is divided into 32 subarrays per chip, for a total of 4K subarrays per rank. Within one chip, each subarray consists of a matrix of memory cells organized into 1,024 rows and 8,192 columns. We also assume that DRAM used for PIM operation is dedicated for that purpose and physically separate, so that PIM does not interfere with regular memory read and write in the main memory, and simplifies memory management. This PIM module can be assigned to a specific memory controller or accessible over CXL system interconnect, similar to the way GPUs are attached via PCI Express. In this paper, we assume a DDR interface with 25.6 GB/s rank bandwidth.

We chose to separate the PIM memory module(s) because many PIM architectures require data to be laid out differently than in conventional memory. For example, bit-serial PIM architectures require data to be laid out vertically [20], instead of the usual horizontal layout, where all the bits are in the same row. Even PIM architectures that use a more conventional horizontal data orientation require all the bytes of each word to be together, and may need data placed carefully to spread it across all banks and subarrays to maximize parallelism. This may necessitate a different layout than the typical address interleaving. Therefore, many prior PIM architectures require some data movement to place the data into the proper layout for PIM operations. Using a separate memory module provides a location to place the data in the desired layout and to work around the memory system's address interleaving.

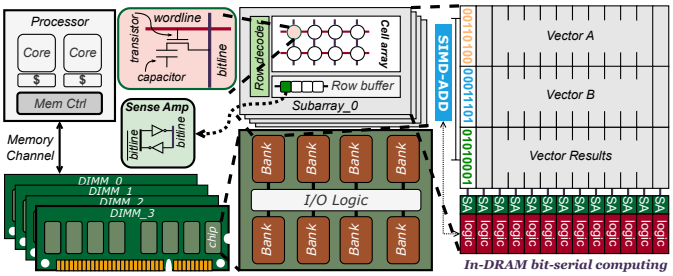


Fig. 3: Generic bit-serial PIM architecture.

IV. 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 DIMM module (e.g., AX-DIMM [66]) 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 2.

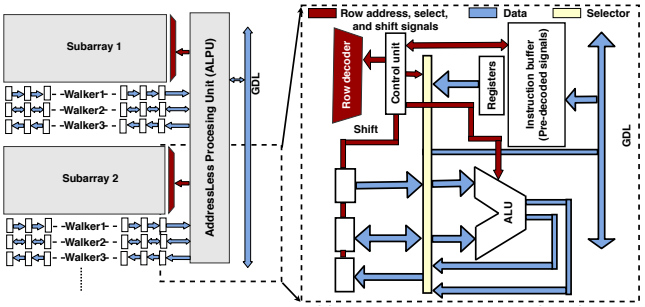


Fig. 4: Bit-parallel subarray-level PIM, based on Fulcrum. [37]

Subarray-level Bit-Serial. The first option, shown as (1) in Figure 2, 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 3. This configuration offers subarray parallelism as well as row-wide bit-slice parallelism, and is sometimes also referred to as Processing Using Memory (PUM) [49]. 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 digital subarray-level bit-serial PIM architecture inspired by previous works [17], [38], [73]. Previous bit-serial PIM proposals, such as Ambit and SIMDRAM [26], [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 [44]. 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 [44], DRAM vendors have expressed a preference for digital PIM approaches. DRISA [38] introduces a hybrid analog-digital model, extending the analog approach by adding digital gates to the sense amplifiers. This provides a richer set of Boolean operations, allowing arithmetic operations to be composed more efficiently. In this paper, we model a purely digital bit-serial PIM architecture (shown in Figure 3), modeled after Micron's digital IMI architecture [17], with a few additional logic gates to implement associative (conditional match-update) processing, inspired by prior work on associative processing [10], [73]. 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.

For high-level operations such as integer addition, bit-serial PIM runs a microprogram, i.e., a sequence of bit-serial operations, to achieve the desired computation. We assume the microprogram is executed by the memory controller, broadcasting each operation to all banks and subarrays. The complexity of the microprograms ranges from linear in terms of operand bitlength, e.g., for integer addition/subtraction, to quadratic, e.g., for integer multiplication and floating point.

Subarray-level Bit Parallel. The second option, which is another variation of subarray-level PIM, not shown in Figure 2, was introduced in the Fulcrum [37] work, and involves placing a more conventional bit-parallel, scalar ALU at the edge of the row buffer This architecture is shown in Figure 4. While Fulcrum was originally designed for 3D stacked Hybrid Memory Cube (HMC) [28], we have adapted its design for DDR to fit within the scope of our paper. 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, our DDR version offloads computation to the CPU if it cannot be performed locally at the subarray. The essence of this architecture is a 32-bit, 164MHz ALU shared between two consecutive subarrays, providing parallel processing capabilities. Figure 4 presents the overall architecture of Fulcrum, consisting of two primary components: (i) Walkers and (ii) the AddressLess Processing Unit (ALPU). The Walkers feature three rows of latches. These Walkers either capture input operands read from the subarray or store target variables before writing them back to the subarray. The read/write operations are carried out sequentially, using a one-hot-encoded value to determine the selected column for bus placement. The ALPU, integral to the architecture, includes four components: (i) a controller, (ii) three temporary registers, (iii) an ALU, and (iv) an instruction buffer. Unlike bit-serial architecture described in section IV, Fulcrum assumes that two consecutive subarrays can communicate with each other using the LISA technique [11], but we do not use that feature in these benchmarks—that is left for future work. The ALPU can be 32 or 64 bits wide; in this study, we model 32-bit ALPUs, able to perform SIMD operations if needed, so that the entire 32 bits can be processed in one step.

Bank-level PIM. The third option, shown as (2) in Figure 2, is to place a processing element at the bank interface, enabling bank parallelism, inspired by the BLIMP project [14]. BLIMP integrates a simple, in-order-issue 200 MHz RISC-V RV64GC processor without caches directly into each memory bank. BLIMP is designed so that banks operate independently, with each core associated with a bank only able to access data within that bank, and can operate fully in parallel, i.e. all-bank mode. The PIMeval bank-level model also assumes banks operate independently and support all-bank computation, but so far model only a simplified bank-level processing element, similar to the Fulcrum ALPU, featuring three Walkers, and a 128-bit processing unit attached to each bank, with datatypes smaller than 128 bits processed in SIMD fashion. We assume a 128-bit GDL here to be generous to bank-level PIM. The walkers are as wide as the subarray local row buffer, which allows some pipelining for fetching data into the walkers and computation. Since banks cannot communicate with each other, any computation requiring inter-bank communication is offloaded to the host CPU.

V. PIMeval

PIMeval is a versatile and highly extensible simulator designed to model various PIM architectures, including those with vertical or horizontal data layouts [21]. It includes built-in performance and energy models for three different PIM architectures (Section IV). Moreover, benchmarks can be written using high-level PIM APIs that will work on all supported PIM architectures without any code modification. Once a simulation target (i.e. PIM device) is specified, the PIM simulator automatically determines the data layout and provides comprehensive performance and energy statistics for data movement and PIM operations. This section demonstrates the design of PIMeval as well as how the functionality, performance, and energy is modeled.

A. Architecture

PIMeval has been developed as a C++ library to facilitate the development of benchmarks for PIM. The library can be linked with benchmark programs to generate executable capable of performing PIM simulations. It offers a comprehensive set of APIs (Section V-B), supporting the creation of PIM devices, resource management, data transfer, and a wide array of high- and low-level PIM operations. ThePIMeval framework is illustrated in Figure 5.

We abstract the basic processing unit, depending on the computational capability of each PIM architecture, as a *PIM core*. This PIM core can represent either a processing unit

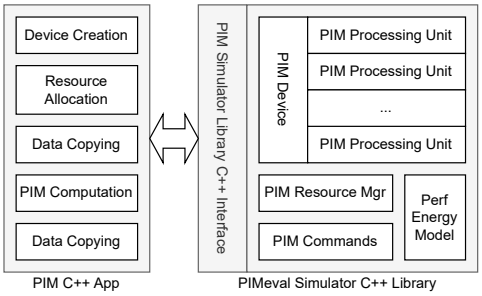


Fig. 5: PIMeval framework.

attached to subarray or bank (other options, such as module-level near-memory processing, are future work). Various high- and low-level PIM operations are then abstracted as PIM commands that can be executed on a PIM core. Benchmarks can utilize the same high-level operations for performance and energy measurement across different PIM architectures. PIMeval currently only supports digital techniques, but support for analog bit-serial techniques would be straightforward to add, and this is left for future work.

A PIM resource manager is implemented to handle data object allocation, deallocation, and tracking. A PIM data object can span multiple 2D memory regions across multiple PIM cores, leveraging subarray-level or bank-level parallelism. Depending on the PIM architecture, the data may be laid out horizontally or vertically within a memory subarray. PIM operations use object IDs as operands and can retrieve object information from the resource manager.

B. PIM API

PIMeval presents a high-level API set that supports functional behavior, performance and energy modeling across multiple PIM architectures, ensuring that the same benchmark implementation can be applied to different architectures. The API set includes common SIMD logical operations, integer arithmetic, comparisons, as well as broadcasting, popcount, and reduction sum operations. These APIs utilize PIM data object IDs as operands, with the assumption that a software runtime manager handles resource allocation and address mapping. The simulator's API set is easily extendable to accommodate additional PIM operations and data types. An example program, *AXPY*, implemented using these APIs, is shown in Listing 1.

```
void axpy(uint64_t vectorLength, const vector<int> &X, const vector<int> &Y, int A)
    {
  unsigned bitsPerElement = sizeof(int) * 8;
  // Allocate device memory
  PimObjId objX = pimAlloc(PIM_ALLOC_AUTO, vectorLength, bitsPerElement, PIM_INT32)
      ;
  PimObjId objY = pimAllocAssociated(bitsPerElement, objX, PIM_INT32);
  assert((objX != -1) && (objY != -1));
  // Copy inputs, perform operations, copy back results
  pimCopyHostToDevice(X.data(), objX);
  pimCopyHostToDevice(Y.data(), objY);
  pimScaledAdd(objX, objY, objY, A)
  pimCopyDeviceToHost(objY, Y.data());
  // Free allocated memory
  pimFree(objX);
  pimFree(objY);
}
```

Listing 1: AXPY implementation using PIMeval API set.

C. Performance Modeling

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

i. Data Movement Latency: This is modeled based on the number of bytes transferred and the available memory bandwidth. For more precise modeling, integration with DRAMsim3 [39] has been left as future work. PIMeval currently does not differentiate between channels and ranks; this distinction will be rectified via integration wtih DRAMsim3. This means that all ranks are treated as independent channels, which amplifies data transfer bandwidth. Overhead of large data transfers will increase once modeling accounts for multiple ranks sharing a channel.

ii. Kernel Execution: Each PIM benchmark consists of multiple PIM APIs, with some benchmarks including kernels executed on the host either due to random access or expensive inter-bank communication. The performance of the host portion is measured using C++ STL's high-resolution clock. PIMeval models the performance of each PIM API based on the characteristics of the PIM device and DRAM parameters obtained from DDR data sheet.

Performance of subarray-level bit-serial PIM is determined by factors such as the command type (e.g., arithmetic, logical) and data type (e.g., int32, int8). These operations often require a bit-serial microprogram using simpler, Boolean micro-ops, with a vertical data layout, to execute SIMD operations on bit-slices of input vectors. To accurately model the performance of targeted bit-serial PIM, all high-level PIM APIs are mapped to low-level bit-serial microprograms.

The performance 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 global row buffer (through narrow GDL width) and the latency to execute PIM command by the processing unit.

Besides SIMD PIM operations, some non-SIMD operations require special handling when modeling performance:

(1) Reduction: This operation is essential for a few kernels of our benchmark. 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 appropriate hardware support is available.

(2) Broadcasting: This involves copying the same scaler value or a pattern to all elements of PIM objects. Bit-serial PIM can efficiently broadcast a bit value to entire memory row, while bit-parallel PIM can propagate the value to a walker/register per subarray/bank.

D. 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 [45] because it is based on vendor data and is straightforward to incorporate into PIMeval.

i. Data Transfer Energy. The energy for data transfers between the PIM and host, in both directions, is calculated using the Micron power model. For instance, the read power is obtained using Equation 1 which is then multiplied by the time spent on memory read to calculate energy.

$$\text{Read Power} = V_{DD} \times (I_{DD4R} - I_{DD3N}) \quad (1)$$

ii. Application Execution Energy. Application execution energy is modeled at the granularity of PIM API calls. We aggregate the energy consumed by all the PIM APIs used in the application, depending on the specific PIM device executing 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 2. The GDL energy for transferring data within a bank, is scaled based on data from LISA [11]. 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.

$$\text{AP} = V_{DD} \times (I_{DD0} \times (t_{RAS} + t_{RP}) - (I_{DD3N} \times t_{RAS} + I_{DD2N} \times t_{RP})) \quad (2)$$

Energy for applications that has host execution, is modeled by multiplying the host execution runtime with the TDP listed in table II. 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 of DDR 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, observed 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-19, the CPU idle energy increases correspondingly. For the same input size, VGG-19 consumes 45k mJ of PIM execution energy and 22k mJ of CPU idle energy, leading to a 48% increase in total energy consumption.

TABLE II: Configuration of the Evaluated Architectures.

Architecture	Parameters
CPU	AMD EPYC 9124 16-core @ 3.71GHz, 200W TDP, 768GB DDR5, 12 memory channels, peak memory BW 460.8GB/s.
GPU	NVIDIA A100, 80GB HBM, 300W TDP, peak memory BW 1,935GB/s, peak compute rate for 32-bit FP is 19.5 TFLOPs.
Bit-serial	32GB DDR4, 32 ranks, 128 banks per rank, 32 subarrays per bank, 8192-bit local row buffers. A bit-serial processing unit is attached to each sense amplifier in local row buffer, with 4-bit registers and move/set/and/xnor/mux operations.
Fulcrum	32GB DDR4, 32 ranks, 128 banks per rank, 32 subarrays per bank, 8192-bit local row buffers. 32-bit 167 MHz integer ALU and three 8192-bit walkers are shared between every two subarrays.
Bank-level PIM	32GB DDR4, 32 ranks, 128 banks per rank, 32 subarrays per bank, 8192-bit local row buffers. 128-bit GDL, global row buffers, a 64-bit Fulcrum-style ALPU, and three 8192-bit walkers for each bank.

E. Verification

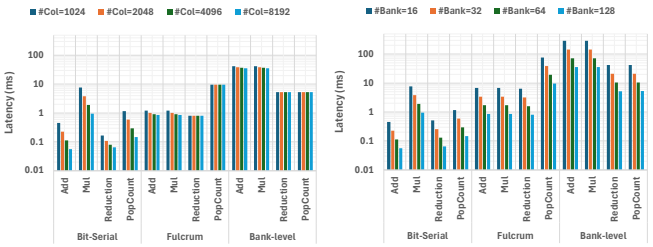
i. Functional Verification. The output of each PIM application was compared against the original CPU execution to ensure functional correctness.

ii. Performance Modeling Validation. We validated the performance model of PIMeval by comparing its results for Fulcrum with those produced by the original Fulcrum simulator across four benchmarks: Vector Add, AXPY, GEMV, and GEMM. The simulator achieved identical performance for Vector Add and AXPY compared to the Fulcrum simulator. However, for GEMV and GEMM, PIMeval's results were approximately 10% slower than those of the original Fulcrum simulator, which we attribute to the overhead associated with the data allocation mechanism within PIMeval. We further performed a performance verification for Vector Add and GEMV using UPMEM [24], and observed a 23% and 35% slowdown in our toy UPMEM model compared to the UPMEM hardware. We attribute this slowdown, in part, to PIMeval's inability to accurately model the tasklets utilized by UPMEM. Moreover, PIMeval exhibits limitations in its data allocation strategy, particularly for horizontal data layouts, as it assumes that the entire DRAM row is filled with valid data and computes the latency accordingly, even if the data are not large enough to fill the row. These limitations will be addressed in future versions of PIMeval.

VI. Methodology

Each PIM variant employs a DRAM configuration consisting of one or more ranks of DIMM, with each rank comprising 8 chips. Each chip contains 16 banks, and each bank is subdivided into 32 subarrays. These subarrays are organized as 1024x8192 matrices of memory cells. For our CPU and GPU baselines, we use the AMD EPYC 9124 16-Core Processor [1] and the NVIDIA A100 GPU [55], respectively. Table II summarizes the configurations for each architecture.

The benchmarks listed in Table I are implemented for PIM using our high-level PIM APIs, as detailed in Section V-B. For GPU implementations, we primarily rely on libraries such as cuBLAS [53], Thrust [54], and CUB [52]. Triangle counting on the GPU uses Gunrock [70]. For the CPU, implementations



(a) Varying #Columns (b) Varying #Banks

Fig. 6: Sensitivity Analysis of PIM devices based on different #columns and #banks for 256M 32-bit INTs.

are based on OpenMP [12] and pthreads [50], along with optimized libraries like OpenBLAS [71] where applicable. For triangle counting on the CPU, we use GAPBS [3]. For the GPU and CPU implementations of VGGs, we use PyTorch [58]. Also, we implemented the AES CPU baseline using the OpenSSL library [57], which utilizes Intel's AES-NI instruction set extensions [23] for optimized cryptographic performance.

We assume that the PIM and GPU are both attached via PCI Express or CXL (which is based on PCIe), so data transfer bandwidth would be the same for both. We therefore factor this out in the PIM vs. GPU comparisons. Similarly, we factor out CPU idle energy in PIM vs. GPU comparisons.

VII. Sensitivity Analysis Of PIM Variants

We perform a sensitivity analysis (eg. varying #columns, #banks) of bit-serial, Fulcrum, and bank-level PIM architectures using four primitive PIM operations: addition, multiplication, reduction, and popcount. These operations represent common logical and arithmetic tasks, focusing on 32-bit integers. The evaluation loads n-size vectors and performs a single operation, excluding data movement latency between the host and PIM. Figure 6 shows latency scaling of the operations for varying #columns and #banks. Bit-serial is most sensitive to these parameters. Fulcrum and bank-level, both being bit-parallel, show sensitivity to bank-level parallelism.

Addition. As data is laid out vertically in bit-serial PIM (section IV), it opens $3n$ DRAM rows to perform any two-input/one output n-bit simple operations, working one bit at a time. However, each DRAM row is processed in parallel, allowing the same operation to be performed on the entire bit-slice in one cycle. In contrast, Fulcrum only exploits subarray-level parallelism but not row-wide parallelism, and bank-level only exploits bank parallelism. Hence, bit-serial PIM achieves highest performance for operations akin to addition, with low complexity per bit.

Multiplication. Multiplication in bit-serial PIM is costly due to its quadratic relationship with element bit-width [32]. In contrast, Fulcrum and bank-level, being bit-parallel, can perform one full scalar multiplication per ALU cycle. Figure 6 shows that bit-serial still outperforms bank-level due to narrow GDL width and limited parallelism across banks. Fulcrum demonstrates the best performance for multiplication.

Reduction. Bit-serial PIM uses a popcount-based integer reduction sum [73]. Both Fulcrum and bank-level PIM perform reduction similar to addition. As bit-serial PIM has the advantage of higher parallelism over both Fulcrum and bank-level PIM, it shows the best performance (Figure 6).

Popcount. Fulcrum implements popcount using SWAR [13], requiring 12 ALU cycle per popcount. On the other hand, bit-serial popcount is log-linear with respect to element bit-width. The bank-level PIM can perform popcount in one CPU cycle [18], [19]. As a result, both bank-level and bit-serial PIM outperform Fulcrum. (Figure 6).

VIII. PIMbench: Analysis & Findings

This section describes the PIMbench suite, along with details on the adaptations made for PIM, and presents an analysis of the performance and energy consumption of each benchmark compared to baseline CPU and GPU architectures. Table I presents the list of the benchmarks, chosen to encompass a diverse range of applications, along with their input sizes, memory access patterns, and execution types. An execution type with the value *PIM + Host* denotes benchmarks that include constituent kernels running on the host due to either random access pattern or the need to support inter-bank communication. Figure 1 presents a dendrogram to quantify the diversity, based on the instruction mix, memory access pattern, execution type, and arithmetic intensity of each application. These parameters are then refined using a combination of PCA and hierarchical clustering [48] to produce the dendrogram.

A detailed breakdown of the PIM operation mix of PIMbench is shown in Figure 8. Figure 7 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.

Figure 9 shows the speedup achieved by each PIM variant, using 32 ranks, for different benchmarks compared to the CPU, while Figure 10a shows the speedup over the GPU. Note that some benchmarks are not able to benefit from so many ranks, due to data movement overhead, host interaction, or simply because the problem sizes we chose are not large enough. Rank scaling will be discussed in Section IX. Figure 9 shows speedup considering both data movement and kernel execution time together, as well as speedup based solely on kernel execution time. For Figure 10a we neither include *cudaMemCopy* cost for GPU, nor *PimCopyHostToDevice* for PIM, to ensure fair comparison, as both can use PCIe/CXL. Figures 11 and 10b demonstrate the energy reductions relative to the CPU and GPU, respectively. The GPU comparison factors out CPU idle energy during GPU/PIM execution.

Vector Addition. Vector addition [5] is an element-wise operation on two vectors and serves as an ideal candidate for PIM, particularly for bit-serial PIM, because addition is efficient even wtih bit-serial, as discussed in Section VII. Consequently, bit-serial PIM demonstrates the highest speedup

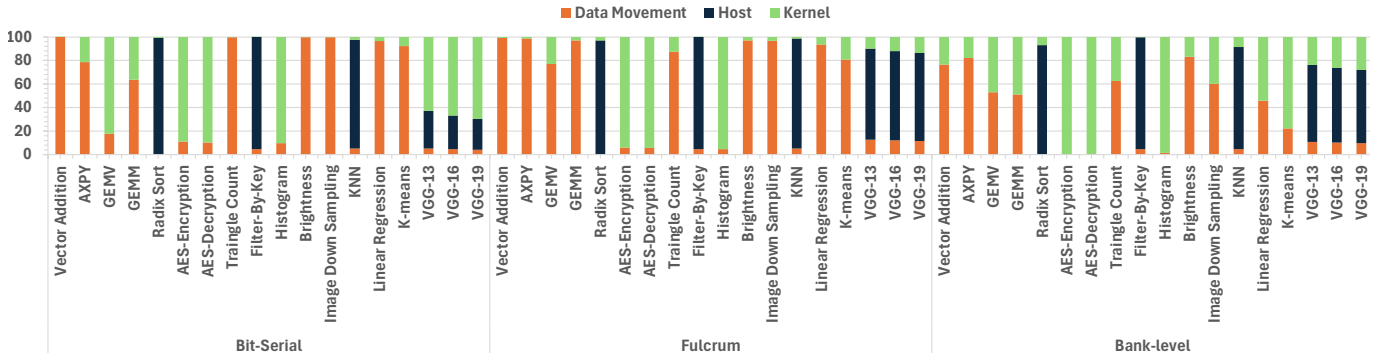


Fig. 7: Performance Breakdown in percentage for Rank 32.

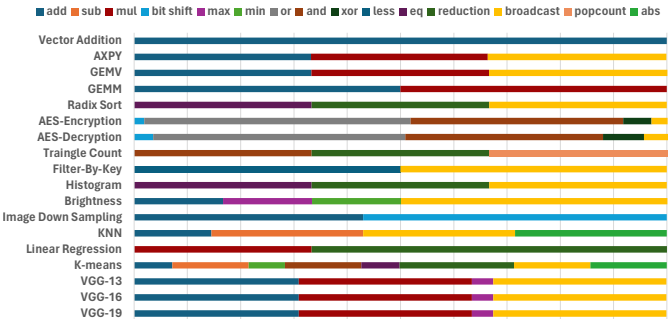


Fig. 8: PIM operation frequency distribution, in terms of percentage of total operations in that benchmark.

compared to both CPU and GPU. Fulcrum achieves the second-best performance, followed by bank-level PIM. A similar trend is observed in terms of energy reduction.

AXPY. AXPY, collected from InSituBench [37], is a linear algebra kernel [5] that scales a vector and adds it to a second vector. This involves both multiplication and addition (Figure 8). Fetching the second (addition) vector operand can be pipelined with the scaling. Fulcrum offers an efficient approach for executing multiplication, outperforming both bit-serial, which suffers from quadratic multiplication latency, and bank-level, which is constrained by the narrow GDL width. Consequently, Fulcrum achieves the highest speedup and energy reduction for AXPY compared to CPU and GPU.

Matrix-vector Multiplication (GEMV). GEMV [5] is a fundamental operation in linear algebra. Fulcrum, with its efficient execution of multiplication−−a key operation in GEMV−−outperforms both bit-serial and bank-level PIM, as shown in Figures 9 and 10a. As with AXPY, bit-serial PIM experiences a slowdown compared to the GPU due to its quadratic multiplication latency, while bank-level PIM, constrained by the narrow GDL width, shows a slight slowdown relative to the GPU.

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.

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 only the counting phase on PIM and the sorting phase on the host CPU. This approach introduces significant host latency, causing the PIM variants to show only slight speedup over the CPU and significant slowdown and energy consumption compared to the GPU.

AES—Advanced Encryption Standard. We implemented AES-256 in ECB mode [15], which processes input in 16-byte state buffers over 14 rounds of logical and look-up operations.In our PIM implementation, the look-up table is realized using logic gates [25], making it well-suited for bit-serial processing. Similarly, since the Fulcrum and Bank-level architectures lack the capability to store the look-up table in a buffer, they also implement it using logic gates. Bit-serial has higher performance compared to Fulcrum and Bank-level as it performs well in logical operations and also has higher parallelism. Also, Fulcrum has takes advantage of sub-array level parallelism compared to bank-level which makes it faster. Bit-serial implementation gains a speedup over CPU, however the GPU outperforms all of the PIM architectures, because AES has higher computation intensity per byte.

Triangle Count. Triangle counting is a graph algorithm [8] that determines the number of triangles (three mutually connected vertices) in a graph. To map the triangle counting algorithm onto PIM, a series of AND, popcount, and reduction sum is employed [69]. Among the three PIM variants, bit-serial demonstrates the best execution latency for these operations (Section VII), 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 compared to the CPU and GPU.

Filter-By-Key. One of the most common operations in data analytics is scanning a database table to select records that match a specific predicate. In our implementation, we only scan and fetch selected records from one column, and

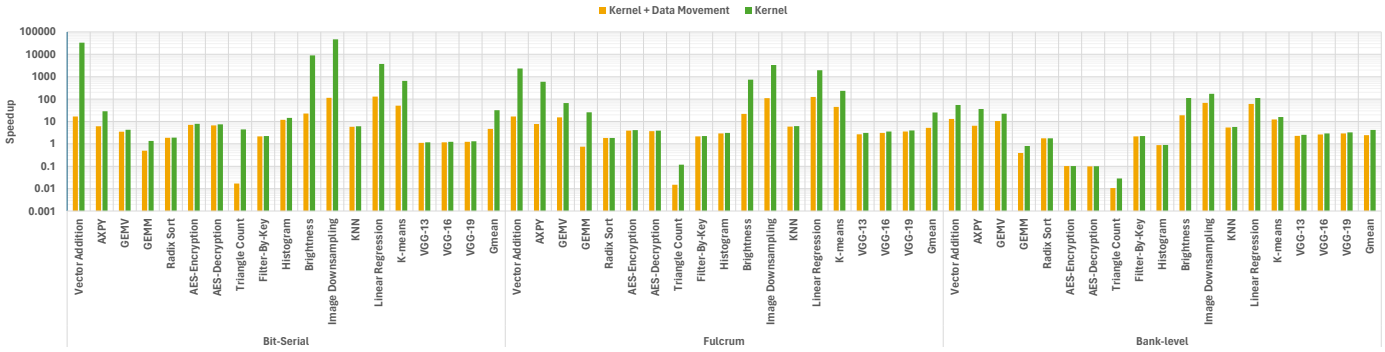
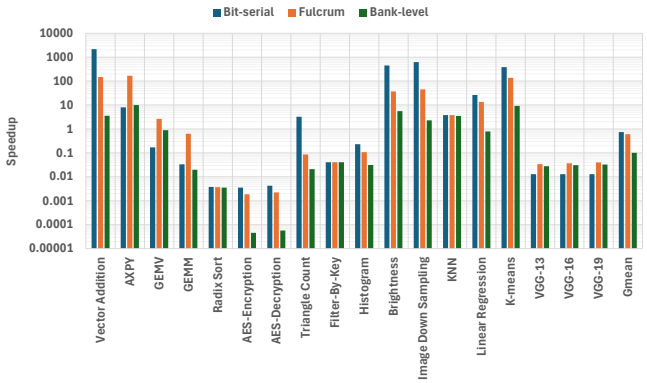
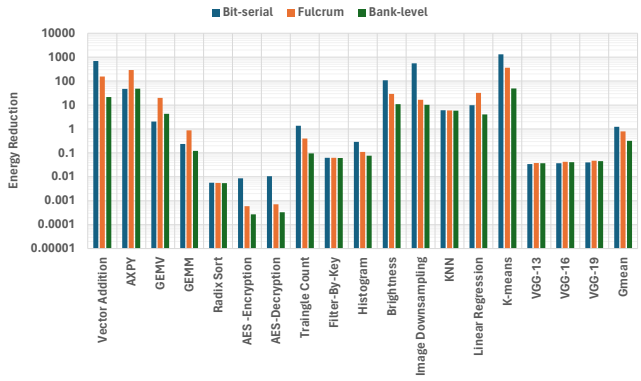


Fig. 9: Speedup of three different PIM variants with 32 ranks over baseline CPU.



(a) Speedup over baseline GPU.



(b) Energy efficiency of PIM architectures vs. GPU.

Fig. 10: Comparison of performance and energy of three different PIM architectures with 32 ranks over GPU.

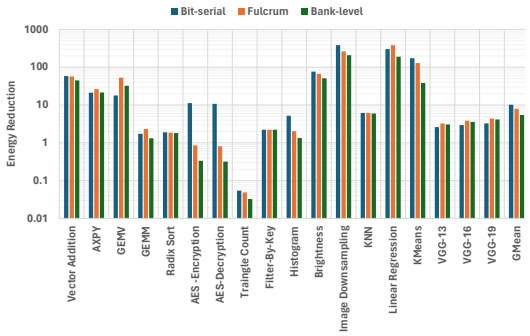


Fig. 11: Energy efficiency of PIM architectures vs. CPU.

1% of the records match the predicate. PIM performs the filtering on the DRAM side and generates a bitmap of the matched items, and this step obtains high speedup on all three architectures, but then the host CPU must fetch the bitmap, and only then can it iterate through the bitmap to gather the selected items in the column. This gathering phase is the bottleneck, constituting 31% of the runtime in the CPU baseline and 99% for PIM (Figure 7).Consequently, none of the PIM variants achieve a speedup over the GPU, though they show a small speedup over the CPU, with the energy results following a similar trend. Higher speedup would be expected if the selected items consisted of more than a single field, since the filtering would lead to eliminating more data fetching.

Histogram. Histogram computes the distribution of RGB values in a 24-bit bitmap file [6], which is modeled after Phoenix [61]. To mitigate the challenges of random access to maximize PIM execution, we extract the pixels for each color channel and sequentially traverse the key-value pairs (0-255), using the equality operation to group together all instances for an individual channel and perform a reduction. Reduction is in PIM and becomes the limiting factor, especially for bit-serial. While all PIM variants demonstrate speedup and energy-reduction over the CPU, all variants experience slowdown and energy-inefficiency compared to the GPU.

Brightness. Brightness, modeled after the SIMDRAM benchmark [26], increments each of the RGB values for every pixel of a 24-bit bitmap file by a set parameter [6]. 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, exploiting the available subarray parallelism, achieves speedup both with and without data movement over CPU and over GPU. For a similar reason, PIM is much more energy efficient compared to both CPU and GPU.

Image Downsampling. PIMbench adapts box filtering for uncompressed bitmap images [40] as an image downsampling candidate. Box filtering sets each pixel in the output image to the average of a box of input pixels [6]. In our implementation, the image is scaled to half its size by applying addition and bit

shifting operations (Figure 8). Since PIM can execute both of these operations optimally, all three PIM variants outperform the CPU and GPU in runtime, while also demonstrating significant energy reductions.

KNN. K-nearest neighbors (KNN) is a supervised learning algorithm which uses the Manhattan distance proximity metric to classify individual points. Our PIM implementation 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 7 the sorting and classification phase contributes a significant percentage of the PIM execution latency. Despite this, we observed modest speedups and improved energy efficiency over CPU and GPU implementations.

Linear Regression. PIMbench implements a 2D variant of the linear regression statistical technique [60]. 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 (as shown in Figure 8), Fulcrum and bit-serial exhibit similar performance, and all three PIM variants achieve speedup over both the CPU and GPU.

K-means. K-means [41], [43] is a widely used 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 efficieny gains over both CPU and GPU.

VGG. VGG [65] is a convolutional neural network variant [7], consisting of an input layer, several hidden layers (including convolution, ReLU, max-pooling, and dense layers), and a softmax output layer. We present three different VGG variants: VGG-13, VGG-16, and VGG-19. The difference among these variants lies in the depth of the network, specifically the number of convolution layers. Running an end-to-end VGG network in PIM is challenging because each layer requires some data preprocessing, such as padding. To address this, we decompose the VGG networks into smaller kernels that correspond to each hidden layer. To maximize parallelism, the input images are processed in batches in the PIM.

PIM can support ReLU, max-pooling, and dense layers, the softmax layer is executed on the host CPU because it requires floating-point operations, which PIMeval does not support yet. Additionally, parts of the convolution layer, such as aggregating the final results, are also executed on the host, as these involve strided access patterns, which are costly in PIM due to the need for inter-bank or inter-subarray communication. As a result, PIM execution is bottlenecked by host execution, leading to moderate speedups and energy efficiency improvements over the CPU for all three VGG variants and across architectures. However, the GPU outperforms PIM significantly in terms of performance and also energy for all PIM architectures.

IX. Discussion & Future Work

We use the same benchmark implementation to evaluate all three PIM architectures modeled here. This approach was chosen to demonstrate the portability of a single implementation across different PIM variants. However, this comes with a limitation: the implementation may not fully exploit architecture-specific optimizations. This limitation could be partially addressed with further optimization of the operation sequence, data layout, or algorithm for each architecture, and architecture-specific PIM API calls may help. Ultimately, a compiler capable of generating device-specific operations using our high-level APIs, or directly generating low-level opcodes for the target architecture will be most helpful. Exploring this direction has been left as a future work.

Another observation is that, even though memory capacity is same (Figure 13), increasing rank count has huge impact on the performance of bit-parallel PIM architectures (Fulcrum, bank-level) because parallel processing unit increases (increased #subarrays and #banks). Bit-serial on the other hand does not benefit as much if the input size is not large enough to fill all the columns of the DRAM rows, because either way it needs to open multiple rows. Fulcrum and bank-level, being bit-parallel, greatly benefits from increased parallelism. However, some benchmarks are unable to realize the benefits of more ranks. In some cases, such as with Radix Sort, the bottleneck is the host interaction. In other cases, such as with GEMV on bit-serial and Fulcrum, the problem size we chose is too small to realize the benefits of more ranks - the vectors are not long enough to utilize all the available subarrays. For the input size used in the experiments in the preceding section, Fulcrum utilizes only 56% of the active subarrays with 8 ranks. As a result, in Figure 12, as we keep increasing the number of ranks, Fulcrum's GEMV performance does not scale beyond 8. Furthermore, bit-serial uses only 15% of the available subarrays, even with only 1 rank, because the vertical layout requires data to be laid out in batches of rows corresponding to the data type's bitwidth. Hence, bit-serial does not exhibit any rank scaling for GEMV. (Figure 12, 13). This is a shortcoming that we unfortunately did not realize in time to correct it for this paper, but the results for AXPY give an approximate sense of the scalability that could be obtained for larger problem sizes. For GEMM, increasing the rank count from 32 to 64, which increases the bank-level

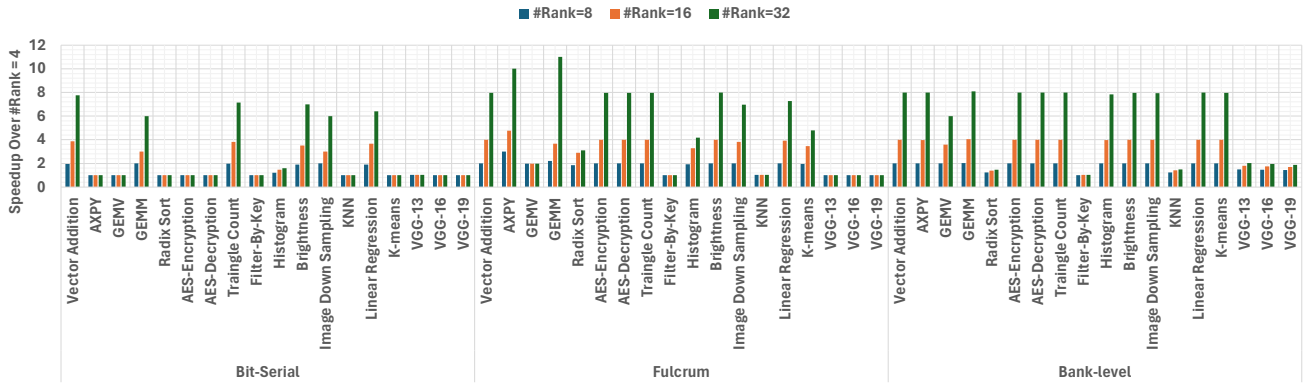


Fig. 12: Rank sensitivity analysis across benchmarks, excluding data movement latency, with capacity scaling by ranks.

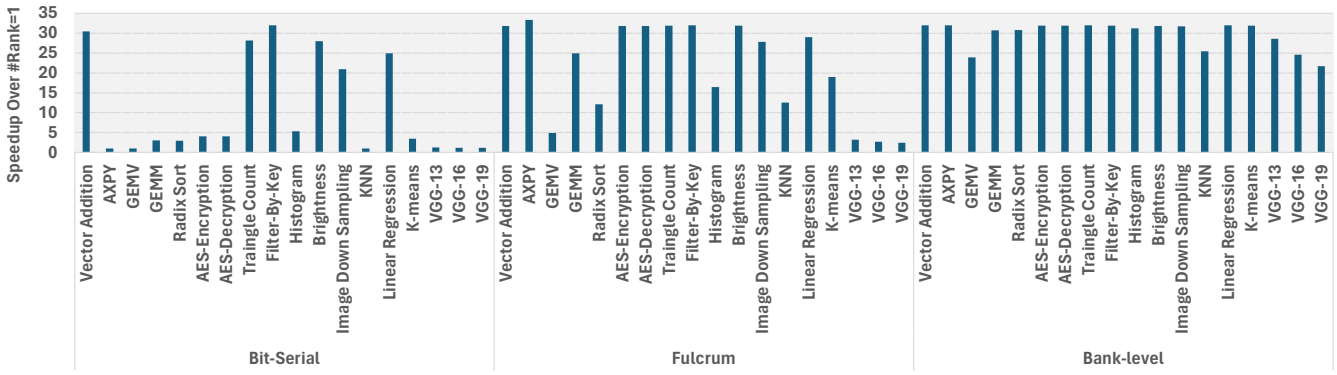


Fig. 13: Rank (1 vs. 32) sensitivity analysis across different benchmarks excluding data movement latency, with same capacity.

parallelism, improves the performance of bank-level PIM, making it more competitive with the GPU. Notably, all three PIM variants achieve energy reductions compared to both CPU and GPU. A comprehensive exploration of problem size is an essential direction for future work. A further consideration is that many use cases call for smaller problem sizes, requiring batching to utilize the full PIM computation bandwidth.

It is also worth mentioning that the benchmark diversity analysis (Figure 1) indicates that some benchmarks are quite similar, and could likely be excluded. Examples include vector-add/image-downsampling, three VGG benchmarks, and AES encryption/decryption. This repetition also skews Gmean results toward those benchmarks' behaviors.

PIMeval is already being extended to support various forms of analog bit-serial PIM. Interesting areas for future work include modeling 3D memories such as HBM and modeling wider SIMD operation in the Fulcrum-style and bank-level approaches, both of which will likely change the tradeoffs observed here; exploration of problem sizes and batching; gem5 integration; a flexible area modeling approach that supports diverse PIM architectures, and further extension and analysis of the benchmark suite. We are continuing to extend PIMbench with additional kernels, such as prefix sum, transitive closure, PCA and string match, additional machine-learning and graph algorithms, and sparse algorithms (which are not easily supported in bit-serial PIM).

X. Conclusions

This paper introduces *PIMbench*- a new benchmark suite, *PIMeval*- a flexible simulation and energy modeling framework, and a new *PIM API*. The paper uses PIMbench and PIMeval to compare the performance of three different PIM architecture: DRAM-AP, a subarray-level bit-serial PIM leveraging row-wide bit-slice operation and supporting associative processing; Fulcrum, a scalar subarray-level PIM; and a bank-level version based on Fulcrum. Our results show that subarray-level Fulcrum provides the best balance of parallelism (via subarray-level parallelism) and flexibility (excelling in multiplication and other more complex operations), and achieves the highest geometric mean performance among the three architectures, outperforming the CPU by about 5.2X (including data transfer overheads). In contrast, for many benchmarks, none of the PIM architectures consistently outperform an A100 GPU, due to various overheads, such as data movement to change data layout for PIM processing. In terms of energy, most benchmarks do show energy reduction compared to the CPU, with a Gmean of 5-10X energy reduction, but results are more mixed for the GPU, with a Gmean of about 2X energy reduction for both subarray-level techniques, but the bank-level approach unable to beat the GPU.

Acknowledgements

We extend our gratitude to the anonymous reviewers for their valuable feedback. We also thank Marzieh Lenjani for her suggestions in modeling Fulcrum.

References

[1] AMD, "AMD EPYC 9124 16-core processor," 2023, accessed: 2024-06-02. [Online]. Available: https://www.amd.com/en/products/cpu/amd-epyc-9124

[2] R. Balasubramonian, A. B. Kahng, N. Muralimanohar, A. Shafiee, and V. Srinivas, "Cacti 7: New tools for interconnect exploration in innovative off-chip memories," *ACM Transactions on Architure and Code Optimization (TACO)*, vol. 14, no. 2, pp. 1–25, June 2017.

[3] S. Beamer, K. Asanovic, and D. A. Patterson, "The GAP benchmark suite," *arXiv*, vol. arXiv:1508.03619, Aug. 2015.

[4] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti *et al.*, "The gem5 simulator," *ACM Special Interest Group on Computer Architecture (SIGARCH) Computer Architecture News*, vol. 39, no. 2, pp. 1–7, May 2011.

[5] L. S. Blackford, A. Petitet, R. Pozo, K. Remington, R. C. Whaley, J. Demmel, J. Dongarra, I. Duff, S. Hammarling, G. Henry *et al.*, "An updated set of basic linear algebra subprograms (BLAS)," *ACM Transactions on Mathematical Software (TOMS)*, vol. 28, no. 2, pp. 135–151, 2002.

[6] J. Blow, "Mipmapping, part 1," http://number-none.com/product/Mipmapping,

[7] M. Bojarski, D. Del Testa, D. Dworakowski, B. Firner, B. Flepp, P. Goyal, L. D. Jackel, M. Monfort, U. Muller, J. Zhang *et al.*, "End to end learning for self-driving cars," *arXiv preprint arXiv:1604.07316*, 2016.

[8] P. Burkhardt, "Graphing trillions of triangles," *Information Visualization*, vol. 16, no. 3, pp. 157–166, 2017. [Online]. Available: https://doi.org/10.1177/1473871616666393

[9] H. Caminal, Y. Chronis, T. Wu, J. M. Patel, and J. F. Martínez, "Accelerating database analytic query workloads using an associative processor," in *Proceedings of the Annual International Symposium on Computer Architecture (ISCA)*, June 2022, pp. 623–637.

[10] H. Caminal, K. Yang, S. Srinivasa, A. K. Ramanathan, K. Al-Hawaj, T. Wu, V. Narayanan, C. Batten, and J. F. Martínez, "CAPE: A content-addressable processing engine," in *Proceedings of the IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, Feb. 2021, pp. 557–569.

[11] K. K. Chang, P. J. Nair, D. Lee, S. Ghose, M. K. Qureshi, and O. Mutlu, "Low-Cost Inter-Linked Subarrays (LISA): Enabling fast inter-subarray data movement in DRAM," in *Proceedings of the IEEE International Symposium on High Performance Computer Architecture (HPCA)*, Mar. 2016, pp. 568–580. [Online]. Available: http://ieeexplore.ieee.org/document/7446095/

[12] L. Dagum and R. Menon, "OpenMP: An industry-standard API for shared-memory programming," *IEEE Computational Science and Engineering (CSE)*, vol. 5, no. 1, pp. 46–55, 1998.

[13] S. Das. (2013) A SWAR algorithm for popcount. Accessed: 2024-06-02. [Online]. Available: https://www.playingwithpointers.com/blog/swar.html

[14] A. Devic, S. B. Rai, A. Sivasubramaniam, A. Akel, S. Eilert, and J. Eno, "To PIM or not for emerging general purpose processing in DDR memory systems," in *Proceedings of the Annual International Symposium on Computer Architecture (ISCA)*, June 2022, pp. 231–244.

[15] M. J. Dworkin, "Recommendation for block cipher modes of operation: Methods and techniques," in *NIST Special Publication 800-38A*, 2001.

[16] D. G. Elliott, M. Stumm, W. M. Snelgrove, C. Cojocaru, and R. McKenzie, "Computational RAM: Implementing processors in memory," *IEEE Design & Test of Computers*, vol. 16, no. 1, pp. 32–41, 1999.

[17] T. Finkbeiner, G. Hush, T. Larsen, P. Lea, J. Leidel, and T. Manning, "In-memory intelligence," *IEEE Micro*, vol. 37, no. 4, pp. 30–38, 2017.

[18] R.-V. Foundation. (2019) The RISC-V instruction set manual. Accessed: 2024-06-02. [Online]. Available: https://riscv.org/wp-content/uploads/2019/12/riscv-spec-20191213.pdf

[19] ——. (2021) RISC-V bit-manipulation ISA specification. Accessed: 2024-06-02. [Online]. Available: https://github.com/riscv/riscv-bitmanip/blob/main/bitmanip/_popcount.adoc

[20] B. R. Gaeke, P. Husbands, X. S. Li, L. Oliker, K. A. Yelick, and R. Biswas, "Memory-intensive benchmarks: IRAM vs. cache-based machines," in *Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS)*, Apr. 2002, p. 7 pp.

[21] F. Gao, G. Tziantzioulis, and D. Wentzlaff, "ComputeDRAM: In-memory compute using off-the-shelf DRAMs," in *Proceedings of the Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, Oct. 2019, pp. 100–113.

[22] S. Ghose, A. G. Yaglikçi, R. Gupta, D. Lee, K. Kudrolli, W. X. Liu, H. Hassan, K. K. Chang, N. Chatterjee, A. Agrawal, M. O'Connor, and O. Mutlu, "What your DRAM power models are not telling you: Lessons from a detailed experimental study," *Proceedings of the ACM on Measurement and Analysis of Computer Systemns*, vol. 2, no. 3, pp. 1–41, Dec 2018.

[23] S. Gueron, "Intel advanced encryption standard (AES) new instructions set," *Intel Corporation*, vol. 128, 2010.

[24] J. Gómez-Luna, I. El Hajj, I. Fernandez, C. Giannoula, G. F. Oliveira, and O. Mutlu, "Benchmarking memory-centric computing systems: Analysis of real processing-in-memory hardware," in *Proceedings of the International Green and Sustainable Computing Conference (IGSC)*, Oct. 2021, pp. 1–7.

[25] O. Hajihassani, S. K. Monfared, S. H. Khasteh, and S. Gorgin, "Fast AES implementation: A high-throughput bitsliced approach," *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, vol. 30, no. 10, pp. 2211–2222, 2019.

[26] 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, "SIMDRAM: a framework for bit-serial SIMD processing using DRAM," in *Proceedings of the ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Apr. 2021, pp. 329–345.

[27] M. He, C. Song, I. Kim, C. Jeong, S. Kim, I. Park, M. Thottethodi, and T. N. Vijaykumar, "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)*, Oct. 2020, pp. 372–385.

[28] Hybrid Memory Cube Consortium (HMCC), *Hybrid Memory Cube Specification 2.1*, Nov. 2015, https://www.hybridmemorycube.org/.

[29] B. Jacob, D. Wang, and S. Ng, *Memory systems: cache, DRAM, disk*. Morgan Kaufmann, 2010.

[30] JEDEC Solid State Technology Association, *High Bandwidth Memory (HBM) Specification*, 2023, https://www.jedec.org/standards-documents/docs/jesd235b.

[31] M. Jung, D. M. Mathew, F. Zulian, C. Weis, and N. Wehn, "A new bank sensitive DRAMPower model for efficient design space exploration," in *Proceedings of the International Workshop on Power and Timing Modeling, Optimization and Simulation (PATMOS)*, Sep. 2016, pp. 283–288.

[32] A. A. Karatsuba and Y. P. Ofman, "Multiplication of many-digital numbers by automatic computers," in *Doklady Akademii Nauk*, vol. 145, no. 2. Russian Academy of Sciences, 1962, pp. 293–294.

[33] Y. Kim, W. Yang, and O. Mutlu, "Ramulator: A fast and extensible DRAM simulator," *IEEE Computer Architecture Letters (CAL)*, vol. 15, no. 1, pp. 45–49, 2015.

[34] D. Lee, B. Hyun, T. Kim, and M. Rhu, "Analysis of data transfer bottlenecks in commercial PIM systems: A study with UPMEM-PIM," *IEEE Computer Architecture Letters (CAL), to appear*, 2024.

[35] M. Lenjani, A. Ahmed, and K. Skadron, "Pulley: An algorithm/hardware co-optimization for in-memory sorting," *IEEE Computer Architecture Letters (CAL)*, vol. 21, no. 2, pp. 109–112, 2022.

[36] M. Lenjani, A. Ahmed, M. Stan, and K. Skadron, "Gearbox: A case for supporting accumulation dispatching and hybrid partitioning in PIM-based accelerators," in *Proceedings of the Annual International Symposium on Computer Architecture (ISCA)*, June 2022, pp. 218–230.

[37] M. Lenjani, P. Gonzalez, E. Sadredini, S. Li, Y. Xie, A. Akel, S. Eilert, M. R. Stan, and K. Skadron, "Fulcrum: A simplified control and access mechanism toward flexible and practical in-situ accelerators," in *Proceedings of the IEEE International Symposium on High Performance Computer Architecture (HPCA)*, Feb. 2020, pp. 556–569.

[38] S. Li, D. Niu, K. T. Malladi, H. Zheng, B. Brennan, and Y. Xie, "DRISA: A DRAM-based reconfigurable in-situ accelerator," in *Proceedings of the Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, Oct. 2017, pp. 288–301.

[39] S. Li, Z. Yang, D. Reddy, A. Srivastava, and B. Jacob, "DRAMsim3: A cycle-accurate, thermal-capable DRAM simulator," *IEEE Computer Architecture Letters (CAL)*, vol. 19, no. 2, pp. 106–109, 2020.

[40] N. Liesch, "The bmp file format," https://www.ece.ualberta.ca/~elliott/ee552/studentAppNotes/2003_w/misc/bmp_file_format/bmp_file_format.htm.

[41] S. Lloyd, "Least squares quantization in PCM," *IEEE Transactions on Information Theory (TIT)*, vol. 28, no. 2, pp. 129–137, 1982.

[42] A. Maćkiewicz and W. Ratajczak, "Principal components analysis (pca)," *Computers & Geosciences*, vol. 19, no. 3, pp. 303–342, 1993.

[43] J. MacQueen *et al.*, "Some methods for classification and analysis of multivariate observations," in *Proceedings of the Fifth Berkeley Symposium on Mathematical Statistics and Probability*, vol. 1, no. 14. Oakland, CA, USA, 1967, pp. 281–297.
[44] M. Marazzi, T. Sachsenweger, F. Solt, P. Zeng, K. Takashi, M. Yarema, and K. Razavi, "Hifi-DRAM: Enabling high-fideltiy DRAM research by uncovering sense amplifiers with IC imaging," in *Proceedings of the ACM/IEEE International Symposium on Computer Architecture (ISCA)*, June 2024.
[45] Micron, "TN-40-07: Calculating Memory Power for DDR4 SDRAM," 2017.
[46] G. E. Moore, "Cramming more components onto integrated circuits," *Proceedings of the IEEE (Proc. IEEE)*, vol. 86, no. 1, pp. 82–85, 1998.
[47] S. Mosanu, M. Z. Sakib, T. Tracy, E. Cukurtas, A. Ahmed, P. Ivanov, S. Khan, K. Skadron, and M. Stan, "PiMulator: A fast and flexible processing-in-memory emulation platform," in *Proceedings of the Design, Automation & Test in Europe Conference (DATE)*, 2022, pp. 1473–1478.
[48] F. Murtagh and P. Contreras, "Algorithms for hierarchical clustering: an overview," *Wiley Interdisciplinary Reviews (WIREs): Data Mining and Knowledge Discovery*, vol. 2, no. 1, pp. 86–97, 2012.
[49] O. Mutlu, S. Ghose, J. Gómez-Luna, and R. Ausavarungnirun, "A modern primer on processing in memory," in *Emerging Computing: From Devices to Systems: Looking Beyond Moore and Von Neumann*. Springer, 2022, pp. 171–243.
[50] B. Nichols, D. Buttlar, and J. P. Farrell, *Pthreads Programming*. O'Reilly Media, Inc., 1996.
[51] NIST, "Advanced encryption standard (AES)," Federal Information Processing Standards Publication, p. 0311, 2001, accessed: 2024-08-14. [Online]. Available: https://csrc.nist.gov/pubs/fips/197/final
[52] NVIDIA Corporation, *NVIDIA CUB Library*, accessed: 2024-06-03. [Online]. Available: https://nvlabs.github.io/cub/
[53] ——, *NVIDIA cuBLAS Library*, accessed: 2024-06-03. [Online]. Available: https://developer.nvidia.com/cublas
[54] ——, *NVIDIA Thrust Library*, accessed: 2024-06-03. [Online]. Available: https://developer.nvidia.com/thrust
[55] ——, "Nvidia A100 tensor core GPU," 2020, accessed: 2024-06-02. [Online]. Available: https://www.nvidia.com/en-us/data-center/a100/
[56] G. F. Oliveira, A. Olgun, A. G. Yağlıkçı, F. Bostancı, J. Gómez-Luna, S. Ghose, and O. Mutlu, "MIMDRAM: An end-to-end processing-using-DRAM system for high-throughput, energy-efficient and programmer-transparent multiple-instruction multiple-data processing," in *Proceedings of the IEEE International Symposium on Computer Architecture (HPCA)*, 2024, pp. 186–203.
[57] OpenSSL Project, "OpenSSL: Cryptography and SSL/TLS Toolkit," https://www.openssl.org, accessed: 2024-08-14.
[58] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga *et al.*, "Pytorch: An imperative style, high-performance deep learning library," *Advances in Neural Information Processing Systems (NeurIPS)*, vol. 32, 2019.
[59] D. Patterson, T. Anderson, N. Cardwell, R. Fromm, K. Keeton, C. Kozyrakis, R. Thomas, and K. Yelick, "A case for intelligent RAM: IRAM," *IEEE Micro*, vol. 17, no. 2, pp. 34–44, Mar./Apr. 1997.
[60] K. Pearson, "Vii. mathematical contributions to the theory of evolution.—iii. regression, heredity, and panmixia," *Philosophical Transactions of the Royal Society of London. Series A, containing papers of a mathematical or physical character*, no. 187, pp. 253–318, 1896.
[61] C. Ranger, R. Raghuraman, A. Penmetsa, G. Bradski, and C. Kozyrakis, "Evaluating MapReduce for multi-core and multiprocessor systems," in *Proceedings of the IEEE International Symposium on High Performance Computer Architecture (HPCA)*, Feb. 2007, pp. 13–24.
[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, "Ambit: In-memory accelerator for bulk bitwise operations using commodity DRAM technology," in *Proceedings of the Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, Oct. 2017, pp. 273–287.
[63] V. Seshadri and O. Mutlu, "In-DRAM bulk bitwise execution engine," *arXiv*, vol. abs/1905.09822, 2019. [Online]. Available: http://arxiv.org/abs/1905.09822
[64] A. Shafiee, A. Nag, N. Muralimanohar, R. Balasubramonian, J. P. Strachan, M. Hu, R. S. Williams, and V. Srikumar, "Isaac: A convolutional neural network accelerator with in-situ analog arithmetic in crossbars," in *Proceedings of the ACM/IEEE International Symposium on Computer Architecture (ISCA)*, June 2016, pp. 14–26.
[65] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," *arXiv preprint arXiv:1409.1556*, 2014.
[66] SK hynix Inc., *AXDIMM: Acceleration DIMM Specification*, 2023, https://www.skhynix.com/.
[67] H. S. Stone, "A logic-in-memory computer," *IEEE Transactions on Computers (TC)*, vol. 100, no. 1, pp. 73–78, 1970.
[68] T. Vogelsang, "Understanding the energy consumption of dynamic random access memories," in *Proceedings of the Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, Dec. 2010, pp. 363–374.
[69] X. Wang, J. Yang, Y. Zhao, X. Jia, R. Yin, X. Chen, G. Qu, and W. Zhao, "Triangle counting accelerations: From algorithm to in-memory computing architecture," *IEEE Transactions on Computers*, vol. 71, no. 10, p. 2462–2472, Oct 2022. [Online]. Available: https://doi.org/10.1109/TC.2021.3131049
[70] Y. Wang, A. Davidson, Y. Pan, Y. Wu, A. Riffel, and J. D. Owens, "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)*, Mar. 2016, pp. 1–12.
[71] Z. Wang, X. Zhang, Y. Zhang, and S. Qing, "OpenBLAS: An optimized BLAS library," *ACM Transactions on Mathematical Software (TOMS)*, vol. 39, no. 3, pp. 1–14, 2013.
[72] L. Wu, R. Sharifi, M. Lenjani, K. Skadron, and A. Venkat, "Sieve: Scalable in-situ DRAM-based accelerator designs for massively parallel k-mer matching," in *Proceedings of the ACM/IEEE Annual International Symposium on Computer Architecture (ISCA)*, June 2021, pp. 251–264.
[73] L. Wu, R. Sharifi, A. Venkat, and K. Skadron, "DRAM-CAM: General-purpose bit-serial exact pattern matching," *IEEE Computer Architecture Letters (CAL)*, vol. 21, no. 2, pp. 89–92, 2022.
[74] S. Xu, X. Chen, Y. Wang, Y. Han, X. Qian, and X. Li, "PIMSim: A flexible and detailed processing-in-memory simulator," *IEEE Computer Architecture Letters (CAL)*, vol. 18, no. 1, pp. 6–9, 2018.
[75] R. M. Yoo, A. Romano, and C. Kozyrakis, "Phoenix rebirth: Scalable MapReduce on a large-scale shared-memory system," in *Proceedings of the IEEE International Symposium on Workload Characterization (IISWC)*, Oct. 2009, pp. 198–207.
[76] C. Yu, S. Liu, and S. Khan, "Multipim: A detailed and configurable multi-stack processing-in-memory simulator," *IEEE Computer Architecture Letters (CAL)*, vol. 20, no. 1, pp. 54–57, 2021.
[77] T. Zhang, K. Chen, C. Xu, G. Sun, T. Wang, and Y. Xie, "Half-DRAM: A high-bandwidth and low-power DRAM architecture from the rethinking of fine-grained activation," in *Proceedings of the ACM/IEEE 41st International Symposium on Computer Architecture (ISCA)*, June 2014, pp. 349–360.

Appendix A
Artifact Appendix

This paper presents a PIM benchmark suite along with a PIM modeling framework, both implemented in C++. The artifact requires an AMD or Intel x86 CPU with at least 256GB of memory, running Ubuntu version 22.04 or later, and a minimum of 2GB disk space. The software requirements are Python3 and g++ 11.4 or newer. There are no external data dependencies. Each benchmark can be executed with default parameters, producing output similar to Listing 3, which demonstrates the metrics used in the paper. The default parameters are selected to ensure each benchmark runs with a manageable data size and completes within a reasonable time. The data size used for each benchmark can be found in the Table I.

A. Artifact check-list (meta-information)

- **Binary:** Binaries are not included; they need to be built using the approach mentioned.
- **Hardware:** AMD or Intel X86 CPU having at least 256GB memory with Ubuntu versions greater than or equal 22.04.
- **Metrics:** Estimated runtimes for: i) kernel execution, ii) host execution, and iii) data copy overhead between host and device.

- **How much disk space required (approximately)?:** At least 2GB.
- **How much time is needed to complete experiments (approximately)?:** 168 Hrs
- **Publicly available?:** Yes (https://github.com/UVA-LavaLab/PIMeval-PIMbench)
- **Code licenses:** MIT License
- **Archived (provide DOI)?:** 10.5281/zenodo.13243685

B. Description

1) How to access: The repository can be cloned from: https://github.com/UVA-LavaLab/PIMeval-PIMbench.

2) Hardware dependencies: AMD or Intel x86 CPU with a minimum of 256GB memory running Ubuntu version 22.04 or later.

3) Software dependencies: python3, g++ 11.4 or newer.

C. Installation

User should clone the repository from Github. The directory structure is as follows:

- **PIMBench:** Contains benchmark implementations of the PIMBench suite.
- **bit-serial:** Contains bit-serial microcode implementation.
- **libpimeval:** Contains the PIMeval simulator implementation.
- **tests:** Contains test files for testing PIMeval functionality.
- **third_party:** Stores third-party code. Currently, the simulator does not use any third-party libraries.
- **.gitignore:** Prevents including executables and similar files in the git repository.
- **LICENSE:** Contains the license information.
- **Makefile:** A common makefile that builds both the simulator and benchmarks.
- **README.md:** Provides detailed instructions to build and run the simulator and benchmarks.
- **build_run.sh:** A shell script that sequentially builds the simulator for three different PIM variants mentioned in the paper, navigates to each benchmark directory, and executes each benchmark.

Listing 2 provides instructions for running and testing individual benchmarks for the three different PIM devices discussed in this paper. For users who wish to execute all benchmarks across the three PIM devices, the *build_run.sh* script can be used. It is important to note that this process may take several days to complete.

```
git clone https://github.com/UVA-LavaLab/PIMeval-PIMbench
cd PIMeval-PIMbench/
#make and test for bit-serial
make -j PIM_SIM_TARGET=PIM_DEVICE_BITSIMD_V_AP
cd PIMbench/<app_directory>/PIM
./<executable_name>.out
#make and test for fulcrum
make clean
make -j PIM_SIM_TARGET=PIM_DEVICE_FULCRUM
cd PIMbench/<app_directory>/PIM
./<executable_name>.out
#make and test for bank-level
make clean
make -j PIM_SIM_TARGET=PIM_DEVICE_BANK_LEVEL
cd PIMbench/<app_directory>/PIM
./<executable_name>.out
```

Listing 2: Getting Started

D. Evaluation and expected results

Upon executing each benchmark for each PIM device, the output displays PIM statistics, including the runtime for the PIM kernel and the runtime for data movement. Listing 3 provides an example output for the vector addition. For benchmarks involving host execution, the output also includes the *host elapsed time*. When comparing each benchmark with the CPU baseline, we sum these three runtimes to obtain the total benchmark runtime, which is then used to calculate the speedup. For comparisons with the GPU baseline, we add the PIM kernel runtime and the host elapsed time, and use this combined time to calculate the speedup against the GPU.

```
./vec-add.out
Running Vector Add on PIM for vector length: 2048

PIM-Info: Current Device = PIM_FUNCTIONAL, Simulation Target = PIM_DEVICE_FULCRUM
PIM-Info: Config: #ranks = 4, #bankPerRank = 128, #subarrayPerBank = 32, #
      rowsPerSubarray = 1024, #colsPerRow = 8192
PIM-Info: Aggregate every two subarrays as a single core
PIM-Info: Created PIM device with 8192 cores of 2048 rows and 8192 columns.
PIM-Info: Created thread pool with 11 threads.
----------------------------------------
PIM Params:
           PIM Device Type Enum : PIM_FUNCTIONAL
         PIM Simulation Target : PIM_DEVICE_FULCRUM
 Rank, Bank, Subarray, Row, Col : 4, 128, 32, 1024, 8192
            Number of PIM Cores : 8192
        Number of Rows per Core : 2048
        Number of Cols per Core : 8192
                Typical Rank BW : 25.600000 GB/s
                  Row Read (ns) : 28.500000
                 Row Write (ns) : 43.500000
                      tCCD (ns) : 3.000000
Data Copy Stats:
        Host to Device   : 16384 bytes
        Device to Host   : 8192 bytes
        Device to Device : 0 bytes
        TOTAL --------- : 24576 bytes  0.000224ms Runtime 0.001602mj Energy

PIM Command Stats:
    PIM-CMD :         CNT EstimatedRuntime(ms)   EstimatedEnergyConsumption(mJ)
add.int32.h :           1            0.001660                           0.004197
TOTAL ----- :           1            0.001660                           0.004197
----------------------------------------
```

Listing 3: Example Output for Vector Add Benchmark

E. Notes

Running all the benchmarks for the three different PIM architectures may take several days. To expedite the process, users can opt to run only the benchmarks mentioned in the paper with very small data sizes. In this case, users will need to manually navigate to each benchmark directory and refer to the help text to provide the different input parameters for each benchmark.

This project is in under active development. Please check our github for the most recent updates.