A gem5-based Simulation Framework for Computing-in-DRAM

Alexander Kusnezoff[1], João Paulo C. de Lima[1,2], Jeronimo Castrillon[1,2], Asif Ali Khan[1]
[1]*TU Dresden*, [2]*ScaDS.AI*
Dresden, Germany
alexander.kusnezoff@mailbox.tu-dresden.de, {joao.lima, jeronimo.castrillon, asif_ali.khan}@tu-dresden.de

***Abstract*—Computing-in-Memory using DRAM (CIMD) has demonstrated substantial energy and throughput gains for memory-bound workloads consisting of bulk-bitwise operations, by performing computation directly within DRAM subarrays. Realizing CIMD, however, requires a redesign of the memory controller and careful mapping of operands onto the memory arrays. Presently, accurate FPGA-based testbeds exist but they are costly and labor-intensive, while open-source simulators are largely trace-based and cannot execute full application runs. The only full-application CIMD simulator available, included with MIMDRAM [1], is built on an outdated version of gem5 and its toolchains. We present gem5-CIMD, a full-system CIMD simulation framework built on the latest gem5 simulation framework. We extend the simulated instruction set from four bitwise operations to sixteen CIM instructions spanning arithmetic, relational, conditional, and utility operations across a range of data types and bitwidths (4 to 64 bits) . A complete memory-management stack – comprising multi-level huge-page support, a CIM-aware allocator, and a subarray-aware address-mapping scheme – ensures that CIM operands are co-located in the same DRAM subarray. In addition, gem5-CIMD is aware of the DRAM refresh interval and faithfully models and schedules the mandatory refresh operations when needed. We provide a CIM standard library intended as a compiler target and evaluate gem5-CIMD on multiple case studies including an end-to-end KNN workload. The simulator sources, including the evaluated workloads, are available at https://github.com/tud-ccc/gem5-CIMD.**

***Index Terms*—processing-in-memory, processing-using-DRAM, computing-in-memory, logic-in-memory, gem5, architectural simulation, DRAM, compiler infrastructure**

I. Introduction

Data movement, not computation, dominates the energy and latency budget of modern memory-bound workloads [2]. Across machine learning, database, and graph-processing applications, moving operands between memory and the processor is the principal bottleneck, accounting for over 62% of total system energy consumption [2]–[4]. This gap between compute throughput and memory bandwidth continues to widen as Moore's Law continues to slow down. Processing-in-memory (PIM) addresses this directly by performing computation within, or close to, the memory that stores the data, thus reducing or fully eliminating data transfers [3], [5].

PIM has been demonstrated across various memory technologies and computational paradigms, targeting a broad range of application domains [5]. Among them, computing-in-memory using DRAM (CIMD)[1] is especially attractive for applications requiring bulk-bitwise logic and arithmetic operations. By simultaneously *activating* multiple rows, CIMD repurposes the analog charge-sharing behavior of DRAM sense amplifiers to implement basic logic operations directly within the array [6]–[8], from which the full set of bitwise and arithmetic operations can be derived [9]. For instance in the Ambit scheme [6], simultaneously activating three rows causes the charge accumulated on each column's bitline to reflect the *bitwise majority* of the three stored values and a subsequent *precharge* operation closes all rows and writes the majority result back to each of them. This strictly requires operands to be column-aligned within the same subarray. Other architectures such as SIMDRAM [9], MIMDRAM [1], PRADA [8], and FCDRAM [7] demonstrate that a wide range of operations can be expressed as sequences of such *activate* and *precharge* (AP) primitives, collectively referred to as microprograms [9], and executed *in-place*.

To leverage CIMD in practice requires error-prone low-level programming or a novel compilers. There is a growing body of work for the latter, where high-level compilation frameworks take an application kernel as input and generate its corresponding microprograms [10]–[14]. For validation, these works either rely on ad-hoc, in-house simulators or on trace-driven memory-only simulators. The few available hardware testbeds, such as PiDRAM [15], run on FPGA prototypes which are accurate, but costly to acquire and laborious to configure, placing them out of reach for most researchers.

Simulation is the natural alternative, offering fine-grained control over system configuration, inexpensive design-space exploration, and flexible integration into compiler pipelines. Existing CIMD simulators, however, fall short. Ramulator [16]- and NVMain-based simulators [17] operate on memory traces and cannot execute full application runs, capturing the DRAM device in isolation while ignoring the full-system aspects. To the best of our knowledge, the only simulator capable of running complete CIMD applications is the one released with MIMDRAM [1].[2] However, this simulator is built on an outdated version of gem5 (2015), requiring outdated Python and GCC toolchains, and has been isolated from nearly a decade of upstream advances in memory-controller modeling, cache-coherence support, and system emulation. Beyond the fundamental AP and AAP operations, the simulator exposes

[1]Also referred to as Processing-using-DRAM (PuD).

[2]https://github.com/CMU-SAFARI/MIMDRAM

TABLE I
FEATURES COMPARISON IN GEM5-CIMD VS. MIMDRAM.

Feature	MIMDRAM	gem5-CIMD
gem5 version	2015-era	v24 (2025)
Supported CIM ISA	4 ops	16 ops
Same-subarray operand check	×	✓
Same-mat operand check	×	✓
Memory management	×	✓
Refresh during microprograms	×	✓
AP/AAP statistics counters	×	✓

only four bitwise primitives (AND, OR, NOT, XOR) and lacks the memory-management infrastructure needed to guarantee subarray co-location of operands. Arithmetic operations such as addition or multiplication can only be expressed by manually decomposing them into raw AP and AAP sequences — requiring the programmer to implement carry propagation and manage temporary DRAM rows explicitly — with no ISA-level abstraction for arithmetic or relational operations.

This paper proposes gem5-CIMD, a CIMD simulation framework built on the latest gem5 (v24) [18] that addresses the above-mentioned limitations. Table I compares gem5-CIMD with the baseline MIMDRAM implementation, highlighting the newly added features. Concretely, our contributions are as follows:

Extended instruction set: gem5-CIMD extends the instruction set from four bitwise operations to sixteen CIMD instructions, spanning arithmetic, relational, conditional, and utility operations at arbitrary bitwidths from 4 to 64 bits — each backed by a library of 75 precomputed Ambit [6] microprograms covering all supported bitwidths, executed with cycle-accurate DRAM timing.

Full-system support: We implement multi-level huge-page table support in gem5's syscall-emulation mode, a CIM-aware memory allocator, and a subarray-aware address-mapping scheme that guarantees operands sharing a computation are placed in the same DRAM subarray.

Realistic refresh modeling: Microprograms in CIMD quickly become longer than a typical DRAM refresh interval, e.g., a 64-bit addition expands to 1,781 AP/AAP commands. We observed that this is an aspect that has been neglected so far in existing simulators. By omitting mandatory interleaved refresh operations, such simulators yields optimistic performance estimates. To account for this, gem5-CIMD models intermediate refreshes explicitly.

Library support: We expose the full instruction set as a C/C++ standard library intended as a compiler target. We validate gem5-CIM against the MIMDRAM simulator, and evaluate multiple case studies including bitwise logic, arithmetic, relational, and conditional operations as well as an end-to-end KNN classifier as a qualitative proof of concept.

The remainder of this paper is organized as follows. Section II provides a short background on CIMD and the gem5 simulator. Section III describes the gem5-CIMD framework. Section IV presents the evaluation, and Section V concludes the paper.

II. BACKGROUND AND RELATED WORK

A. CIM using DRAM

A DRAM device is organized hierarchically into channels, ranks, banks, and subarrays. Each subarray is a two-dimensional grid of cells, where each cell consists of an access transistor and a storage capacitor holding one bit of data. Cells are arranged at the intersections of wordlines (rows) and bitlines (columns), with a sense amplifier connected to each bitline. A memory controller (MC) translates each physical address into a (rank, bank, subarray, row, column) tuple and drives the device with a small command set. A typical read access proceeds as follows: activate (`ACT`) command raises the target wordline, connecting the storage capacitors of the selected row to their respective bitlines; the stored charge shifts the precharged bitline voltage ($0.5\,V_{DD}$) by a small $\pm\delta$, which the sense amplifiers detect and amplify to full V_{DD} or GND, latching the row into the row buffer. Read/write (`RD`/`WR`) commands then transfer data between the row buffer and the memory bus. Finally, precharge (`PRE`) closes the row and restores all bitlines to $0.5\,V_{DD}$ in preparation for the next access. Separately, `REF` commands must be issued periodically at refresh interval t_{REFI} to restore charge lost to leakage; otherwise, the stored data gets corrupted. All commands are governed by timing constraints—including t_{RAS}, t_{RCD}, and t_{RC}—that bound the minimum time between operations and guarantee correct device behavior.

CIMD repurposes the default behavior of DRAM either by deliberately violating timing constraints [7], [8], or by combining timing violations with DRAM commands reordering [6], [19], to perform computation in-place. RowClone [19] copies a row in place by issuing two back-to-back `ACT`s to the same subarray followed by a `PRE` (`ACT`$\langle src \rangle$, `ACT`$\langle dst \rangle$, `PRE`, abbreviated AAP), so the source row's values overwrite the destination cells without leaving the array. For computations, the exact compute mechanism, i.e., DRAM commands and their order, depends on the CIMD architecture. For instance, in Ambit [6], activating three rows simultaneously — triple-row activation (TRA) — followed by a precharge operations (AP) makes the sense amplifiers resolve the bitwise majority of the three operands, yielding $\mathrm{MAJ}(a, b, c)$. Pinning one input to `0` or `1` gives `AND` and `OR`, respectively, and a dual-contact cell (DCC) provides `NOT`. FCDRAM [7] and PRADA [8] achieve the same effect on unmodified commodity DRAM purely through timing-violating `ACT`/`PRE` sequences. Because all participating rows must reside in the same subarray (or neighboring subarrays, as in FCDRAM) and TRA is destructive, operands are typically first RowClone-copied into reserved compute rows before each operation. Higher-level and more complex operations are represented by decomposing them in longer sequences of AP and AAP commands, which are usually referred to as *microprograms*.

Figure 1 shows the microprogram for a logical `AND` between two rows. Deconstructing large operations into such

```
// AND(a,b) = MAJ(a,b,0) -> c
AAP  a,  T0       // a  -> T0
AAP  b,  T1       // b  -> T1
AAP  C0, T2       // 0  -> T2
AP   T0, T1, T2   // MAJ(a,b,0) = AND(a,b)
AAP  T0, c        // result -> c

// NOT(a) -> c
AAP  a,   DCC0     // a->DCC0, ~a->DCC1
AAP  DCC1, c       // ~a -> c
```

Fig. 1. Ambit microprograms for AND and NOT [6]. T0–T2: B-Group compute rows; C0: constant-zero row; DCC: dual-contact cell (stores x and $\neg x$). All rows must reside in the same subarray.

fundamental logic operations can thus quickly lead to large sequences of memory commands. As an example, a 64-bit addition expands to 1,781 AP/AAP commands, which is why faithfully accounting for interleaved REF commands is essential to correct timing simulation.

B. The gem5 Simulator

gem5 [18] is a modular, cycle-accurate simulator widely used in computer architecture research. It executes unmodified binaries and reports detailed per-component statistics, including a per-rank DRAM energy breakdown across ACT, PRE, read, and write commands. Target systems are described in Python configuration scripts that expose CPU, cache, and memory parameters such as DDR timings, banks per rank, capacity, and the address-mapping scheme. gem5 offers two execution modes: Full-System (FS) mode boots a complete OS image, while Syscall-Emulation (SE) mode intercepts system calls with lightweight stubs and replaces OS structures such as the page table with its own EmulationPageTable, simplifying setup and reducing runtime.

The memory subsystem is structured as a class hierarchy that gem5-CIMD extends. AbstractMemory is the base class for all memory devices and handles functional memory accesses; MemInterface holds the address-mapping policy and shared parameters; and DRAMInterface specializes it with DRAM timing parameters and helpers such as activateBank() and prechargeBank() that issue the underlying ACT/PRE commands. MemCtrl schedules requests onto the interface. This layer is where the CIM command primitives and microprogram execution are introduced.

C. Related Work

Validating the output of a CIMD compiler requires either a hardware testbed or a simulator. FPGA-based platforms such as PiDRAM [15] and DRAM-Bender [20] expose a DRAM interface for issuing arbitrary, timing-relaxed command sequences and reproduce real device behavior, but they are costly to acquire and laborious to configure, placing them out of reach for most compiler developers. Software simulators lower this barrier, yet existing options are limited: Ramulator-PIM [16] and NVMain-PIM [17] model the DRAM device in isolation, operating on memory-command traces and ignoring the non-memory instructions of a real program, so they cannot run full applications.

To our knowledge, MIMDRAM [1] is the only CIMD simulator that executes complete binaries. Its gem5 base, however, dates to 2015 – predating formal gem5 versioning – isolating it from a decade of upstream advances, and it exposes only four bitwise primitives (AND, OR, NOT, XOR). Its public release also omits several mechanisms described in the paper that are essential for executing real CIMD workloads, such as same-subarray operand checks, a subarray-aware address-mapping scheme (the default bank-interleaving scatters the elements of an array across banks), a reserved address space with CIM-aware allocation, and statistics for executed CIM operations. gem5-CIMD implements all these features using the latest gem5, providing a full-application platform for validating hand-written or compiler-generated CIMD code.

III. The gem5-CIMD Framework

gem5-CIMD ports and extends the simulator released with MIMDRAM [1]. The porting itself required non-trivial changes to the code base, including rewriting most of the API function calls (e.g., DRAMPacket→MemPacket, DRAMCtrl→MemInterface+MemCtrl etc.). Beyond the port, we add new features in gem5-CIMD that are essential for modern CIMD systems. These include an extended 16-instruction ISA, subarray-level operands placement, memory management, and DRAM-refresh interleaving during long microprograms.

Figure 2 shows an overview of gem5-CIMD. An input program (either hand-written or compiler-generated code) invokes the CIM standard library (§III-A), which emits custom x86 opcodes via the ISA extension (§III-B). The CPU pipeline fetches and decodes these as memory-referencing instructions, forwards them through the cache hierarchy with cache-line invalidation, and delivers a ROWOP request to the memory controller, which decodes it into AP/AAP sequences executed in the DRAM subarray (§III-C). Orthogonally, cimMalloc() allocates operands through the memory management stack (§III-D), guaranteeing that all operands participating in a CIM operation are co-located in the same DRAM subarray.

A. CIM Standard Library

The CIM standard library is the primary interface between user code and the gem5-CIMD framework. It exposes two layers; a memory-management layer (cimMalloc/cimFree) and a computation layer providing C/C++ wrappers around the sixteen CIM instructions. Each wrapper emits the corresponding custom x86 opcode via inline assembly, passing operand addresses through a fixed register convention: RDI = destination, RSI = first source, RDX = second source, RCX = element count, R8 = element bitwidth[3]. The bitwidth parameter enables sub-byte operand support (e.g., INT4) without any changes to the underlying hardware model. Table II lists all provided routines.

Figure 3 compares the two representations of the same element-wise INT4 addition kernel. The yellow-highlighted

[3]As one would expect, not all registers are used in every CIM instruction.

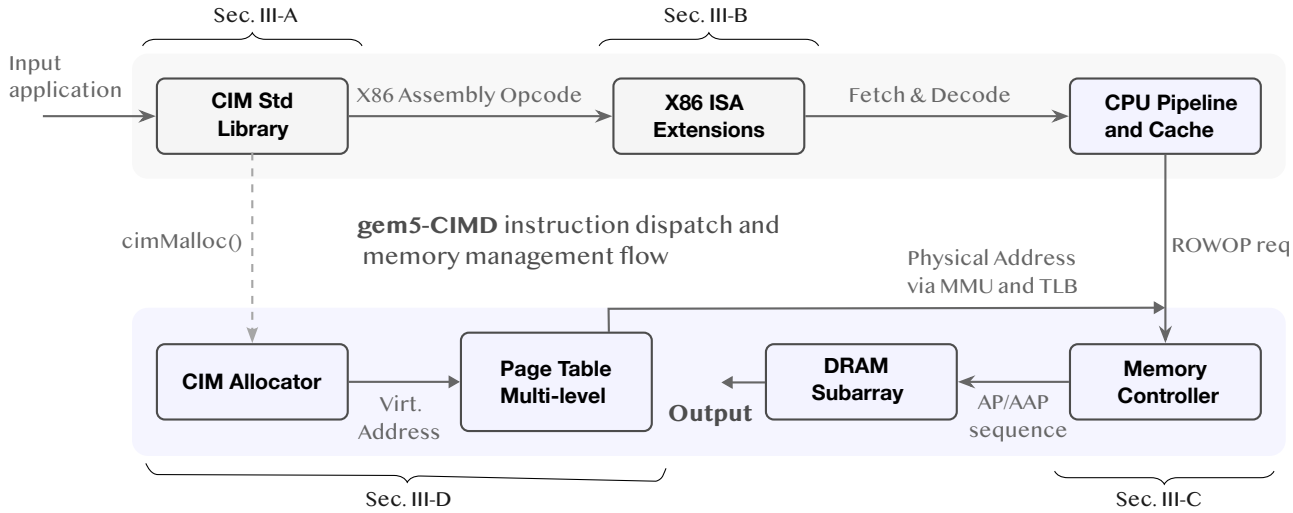


Fig. 2. gem5-CIMD framework overview. The execution flow (top, left to right) traces a CIM instruction from the input application through the ISA and cache hierarchy. The bottom part shows the memory management including huge pages allocation and address translation. The memorycontroller decodes instructions into AP/AAP sequences, and uses the phyical address to execute commands in the DRAM subarray.

TABLE II
CIM STANDARD LIBRARY ROUTINES.

Category	Routines
Memory	`cimMalloc`, `cimFree`
Bitwise	`rowand`, `rowor`, `rownot`, `rowxor`
Arithmetic	`rowadd`, `rowsub`, `rowmult`, `rowdiv`
Relational	`rowmin`, `rowmax`, `rowequal`, `rowgreater`, `rowgreater_equal`
Conditional	`rowif_else`
Utility	`rowabs`, `rowbitcount`

regions highlight how a single `rowadd` call in (a) requires the programmer to manually write the entire carry-ripple decomposition in (b). MIMDRAM does provide `AAP_VECTORS`/`AP_VECTOR` macros that iterate over bit positions rather than fully unrolling all AP/AAP commands, but the programmer must still express every step of the carry logic explicitly . This becomes more complex with operation complexity, e.g., a 32-bit multiplication still reduces to one library call in (a) yet expands to over 18K AP/AAP commands in (b), making manual authoring both error-prone and impractical. For complete application workloads such as the KNN classifier evaluated in Section IV, expressing the computation as raw AP/AAP sequences and manually handling their reduction across columns becomes infeasible altogether.

B. ISA Extension

gem5-CIMD extends the x86 ISA with sixteen new instructions using opcodes in the `0x66 0x0F 0x38 0x4x` range, covering bitwise, arithmetic, relational, conditional, and utility operations at bitwidths 4, 8, 16, 32, and 64 bits – backed by a library of total 16 (instructions) × (5 bitwidths) = 80 precomputed Ambit microprograms. When the CPU decodes a CIM instruction, it creates a memory request with a `ROWOP` flag and attaches a `RowOpPayload` carrying the operation type, operand addresses, element count, and bitwidth. For operations with multiple operands, `splitRowOp()` generates independent sub-requests for each address, each undergoing a separate TLB translation. At the cache level, all cache lines overlapping with the result row are invalidated before the request is forwarded, ensuring that subsequent CPU reads get the correct result rather than a stale cached value.

C. Memory Controller and Refresh Handling

The memory controller (MC) is extended at two levels. `MemCtrl` is extended to decode incoming `ROWOP` requests. It calls `decodePacket()` for each operand address to obtain the precise DRAM address (rank, bank, subarray, row) and asserts that all operands map to the same bank, rank, and subarray — a hard requirement of Ambit-based CIM. The bus state machine is modified to prioritize pending CIM operations before resuming regular read/write accesses. Per-rank statistics counters track the number of TRA commands (APs, see §II-A) and RowClone operations (AAPs) executed, enabling detailed performance analysis.

`DRAMInterface` is extended to support two new primitives. `apBank()` performs a TRA (AP) operation, computing MAJ(a, b, c) across three simultaneously activated rows; and `aapBank()` performs a RowClone (AAP), copying a source row to a destination row using the overlapped-activate timing parameter t_{WLOV} . For basic bitwise operations (AND, OR, NOT, XOR), the decomposition into AP/AAP sequences is hardcoded. For complex operations (arithmetic, relational, conditional), `executeAmbitMicroprogram()` dispatches to a library of 75 precomputed microprograms, one per operation per supported bitwidth. This design decision in intentional, i.e., gem5-CIMD is not designed to optimize the microprogram generation. Instead it in intended to be

```cpp
#include "cim_core.h"

int main() {
  /* int8_t: min C type; BITS=4 sets CIM op width */
  const size_t N = 1024, BITS = 4;
  int8_t *a   = cimMalloc(N*sizeof(int8_t), 0);
  /* same for *b and *dst */

  for (size_t i = 0; i < N; ++i) // Initialization
    { a[i] = (int8_t)i;  b[i] = 2*(int8_t)i; }
  rowadd(dst, a, b, N, BITS); /* dst[i]=a[i]+b[i] */
  /* display / consume output */
  cimFree(a);  cimFree(b);  cimFree(dst);
  return 0;
}
```

(a) gem5-CIMD: C++ standard library

```cpp
// No cim_core.h; ADD not in MIMDRAM ISA.

int main() {
  const size_t N = 1024, BITS = 4;
  /* malloc: no automatic subarray co-location */
  int8_t *a   = malloc(N*sizeof(int8_t));
  /* same for *b and *dst */

  for (size_t i = 0; i < N; ++i) // Initialization
    { a[i] = (int8_t)i;  b[i] = 2*(int8_t)i; }

  /* MIMDRAM provides AAP_VECTORS/AP_VECTOR macros that
     loop over bits (not fully unrolled), but programmer
     must still write this bitwise operations and carry-
     ripple logic manually. */

  AAP a, T0       // a      -> T0
  AAP b, T1       // b      -> T1
  AAP C0, T2       // cin=0  -> T2  (carry for bit 0)
  AP T0, T1, T2   // carry  = MAJ(a,b,cin)
  AAP T0, DCC0     // carry  -> DCC0/DCC1
  AAP a, T0       // reset  T0 = a
  AAP b, T1       // reset  T1 = b
  AP T0, T1, DCC1 // sum    = XOR(a,b,cin)
  AAP T0, dst      // sum    -> dst
  // carry[i] feeds bit i+1. ~8 ops/bit = ~36 for INT4
  /* display / consume output */
  free(a);  free(b);  free(dst);
  return 0;
}
```

(b) MIMDRAM: AP/AAP microprogram [6], [9]

Fig. 3. The same element-wise `INT4` addition in gem5-CIMD (a) and MIMDRAM (b). Red: allocation—`cimMalloc` guarantees subarray co-location automatically; `malloc` does not. Yellow: computation—one `rowadd` call vs. the full carry-ripple decomposition in (b); MIMDRAM's `AAP_VECTORS`/`AP_VECTOR` macros loop over bit positions but the programmer writes the carry logic manually . `T0`–`T2`: B-Group rows; `C0`: const-zero row; `DCC`: dual-contact cell (x and $\neg x$).

orthogonal to the logic synthesis part and faithfully simulate the provided microprograms.

gem5-CIMD also faithfully models refresh operations. Specifically, in longer microprograms such as a 64-bit addition, which requires 1,781 AP/AAP steps, the execution time of these commands far exceeds the refresh interval $t_{REFI} \approx 7.8\,\mu s$ of a typical DRAM. Our framework monitors elapsed time during microprogram execution and inserts synchronous `REF` commands whenever the remaining time before the next required refresh falls below a safety threshold, each consuming t_{RFC} cycles[4]. Omitting these refreshes, as in the MIMDRAM simulator, yields optimistic latency estimates for long-running CIM operations.

D. Memory Management Stack

CIM requires all operands of a single operation to reside in the same DRAM subarray. Standard `malloc` provides no such guarantee, and the default bank-interleaving address mapping actively works against it by scattering consecutive array elements across different banks. gem5-CIMD addresses this through three coordinated mechanisms.

1) Address mapping.: A new `RaBaSuRoCh` (Rank, Bank, Subarray, Row, Column) mapping scheme is added to `DRAMInterface`. It decomposes physical addresses such that consecutive bytes within a mat row share the same subarray, ensuring that properly aligned CIM arrays are physically co-located. This scheme is applied uniformly to all memory accesses to avoid mapping conflicts between CIM and non-CIM traffic.

2) Huge-page Pool: A DRAM subarray typically spans several megabytes, far exceeding a standard 4 KiB page. A reserved pool of 2 MiB huge pages is pre-allocated at a fixed base address (`CIM_BASE_ADDR`), backed by a custom `mmapCim` system call (syscall 500) emulated in gem5's SE mode. Unlike standard `mmap`, `mmapCim` allocates pages eagerly from *contiguous physical memory*, eliminating demand-paging non-determinism. To support variable page sizes, gem5's `EmulationPageTable` is replaced with a multi-level page table mimicking the Linux PGD/P4D/PUD/PMD/PTE hierarchy; 2 MiB pages terminate the walk at the PMD level, bypassing the PTE level.

3) CIM-aware Allocator: `cimMalloc(size, mat_label)` provides fine-grained allocation within the huge-page pool. The `mat_label` parameter allows a compiler to co-locate operands that participate in the same CIM operation by assigning them identical labels. The allocator verifies at runtime that all co-labeled operands land in the same 256 KiB DRAM mat — a finer-grained check than the subarray-level assertion in the MC. Internally, each 2 MiB huge page is divided into eight 256 KiB mats; allocation metadata tracks free row ranges per mat and coalesces freed blocks to prevent fragmentation. `cimFree()` returns blocks to the per-mat free list.

IV. Evaluation

This section presents our experimental setup and evaluation results. We validate functional correctness of gem5-CIMD against the MIMDRAM simulator and demonstrate end-to-end simulation capability with a KNN classifier.

[4]The exact value of t_{RFC} depends on the DRAM technology. For DDR4, $t_{RFC} = 350$ ns, as per the gem5 code base

TABLE III
GEM5-CIMD SYSTEM CONFIGURATION.

Component	Configuration
CPU model	X86TimingSimpleCPU
L1 I/D-cache	64 KiB, 4-way, 64 B lines, 2 cycles
L2 cache	512 KiB, 8-way, 64 B lines, 9 cycles
Memory	DDR4_2400_8x8, 1 channel, 1 rank
Memory capacity	8192 MiB
Address mapping	RaBaSuRoCh
Simulation mode	SE (Syscall Emulation)

TABLE IV
CYCLE COUNT COMPARISON OF GEM5-CIMD VS. MIMDRAM (INT32, 1 K ELEMENTS).

Kernel	MIMDRAM	gem5-CIMD	Δ (%)
add32	13,640,000	15,378,640	+12%
sub32	55,720,000	62,210,640	+11.6%
mul32	1,044,110,500	956,744,960	−8.4%
eq32	18,730,000	21,286,880	+13.6%

A. Experimental Setup

We evaluate gem5-CIMD using gem5's Syscall Emulation (SE) mode with an `X86TimingSimpleCPU`. Table III summarises the simulated system configuration. The memory subsystem uses `DDR4_2400_8x8` with a single channel and rank. All CIM operands reside in a reserved huge-page pool mapped via `RaBaSuRoCh` to guarantee same-subarray placement.

Our workload set covers the full gem5-CIMD instruction set including four bitwise operations (`rowand`, `rowor`, `rownot`, `rowxor`), four arithmetic operations (`rowadd`, `rowsub`, `rowmult`, `rowdiv`), five relational operations, one conditional (`rowif_else`), and two utility operations (`rowabs`, `rowbitcount`). We evaluate integer bitwidths of 8, 16, and 32 bits with array sizes of 8 K and 40 K elements. The KNN workload is used as an end-to-end application case study.

B. Validation against MIMDRAM Simulator

We validate gem5-CIMD against MIMDRAM's simulator in terms of execution cycles, using the microbenchmarks in the MIMDRAM repository and refresh handling fidelty.

1) Head-to-Head Comparison with MIMDRAM: For a direct comparison with MIMDRAM, we run four microworkloads from the MIMDRAM repository with identical vector dimensions (1K elements) on both simulators at INT32, and compare the number of simulated cycles (execution time), as reported in Table IV.

The cycle counts show a mismatch in the range of ±14%. This is expected because gem5-CIMD, compared to MIMDRAM, significantly reduces the number of CPU instruction fetches and decodes due to its extended ISA. Conversely, it increases the execution cycles by modeling mandatory refresh operations. Nonetheless, the results confirm that gem5-CIMD faithfully reproduces MIMDRAM's timing model for not only simple logical operations but also for relatively longer microprograms, as reported in the table.

2) Misalignment Detection: A fundamental requirement of Ambit-based CIM is that all operands of a single instruction must reside in the same DRAM subarray. MIMDRAM relies on the programmer (or compiler) to ensure correct operand placement, and silently produces incorrect results if operands are misaligned. Contrarily, gem5-CIMD enforces with a runtime assertion and the program execution fails if this condition is violated.

To demonstrate this, we allocate two 32 K-element INT32 arrays using standard `malloc` instead of `cimMalloc` and execute `ROWAND` on both simulators. Under MIMDRAM's default bank-interleaving address mapping, consecutive array elements are scattered across different banks. The simulator completes without error but produces bit-corrupted output. Under gem5-CIMD, the memory controller calls `decodePacket()` for each operand address and checks that all operands map to the same bank, rank, and subarray. The misaligned allocation triggers an assertion at write-queue insertion time.

3) Refresh Handling Fidelity: For short microprograms (bitwise operations, $\leq$ 96 AP/AAP steps for INT32), gem5-CIMD and MIMDRAM produce comparable cycle counts. However, for longer microprograms where the total microprogram runtime exceeds the DRAM refresh interval ($t_{REFI} \approx 7.8\,\mu$s), MIMDRAM's estimations are optimistic as it does not model the mandatory refresh commands.

We quantify this for 64-bit addition (1,781 AP/AAP steps). At DDR4-2400 timings, each AP command takes approximately $t_{RAS} + t_{RP} \approx 44$ ns while each AAP takes $t_{RAS} + t_{RP} + t_{WLOV} \approx 48$ ns, giving a total microprogram duration of ∼84 μs — exceeding t_{REFI} by a factor of ∼10.8. gem5-CIMD therefore inserts $\lfloor 84\,\mu\text{s}/7.8\,\mu\text{s} \rfloor = 10$ synchronous `REF` commands, each consuming $t_{RFC} \approx 350$ ns, adding ∼3.5 μs of overhead per operation.

C. Case Studies

Figure 4 shows the evaluation results (throughput) of 11 microworkloads for different operands dimensions (8K, 40K, 500K) and different data types (8, 16, 32). We do not provide any comparison to CPU/GPU systems because that is not the focus of this paper – CIM gains against these architectures have been demonstrated by prior research. Our focus is instead to show that the simulation results are correct (validation in Section IV-B1) and that the simulator can handle running a diverse set of workloads.

To demonstrate the practical applicability of gem5-CIMD, we implemented a K-Nearest Neighbors (KNN) classifier as an end-to-end workload. The KNN algorithm computes the Euclidean distance between a query point and all training samples and then selects the K nearest neighbors via majority voting. The distance computation, which dominates runtime, is embarrassingly parallel across training samples, making it an ideal candidate for CIMD acceleration.

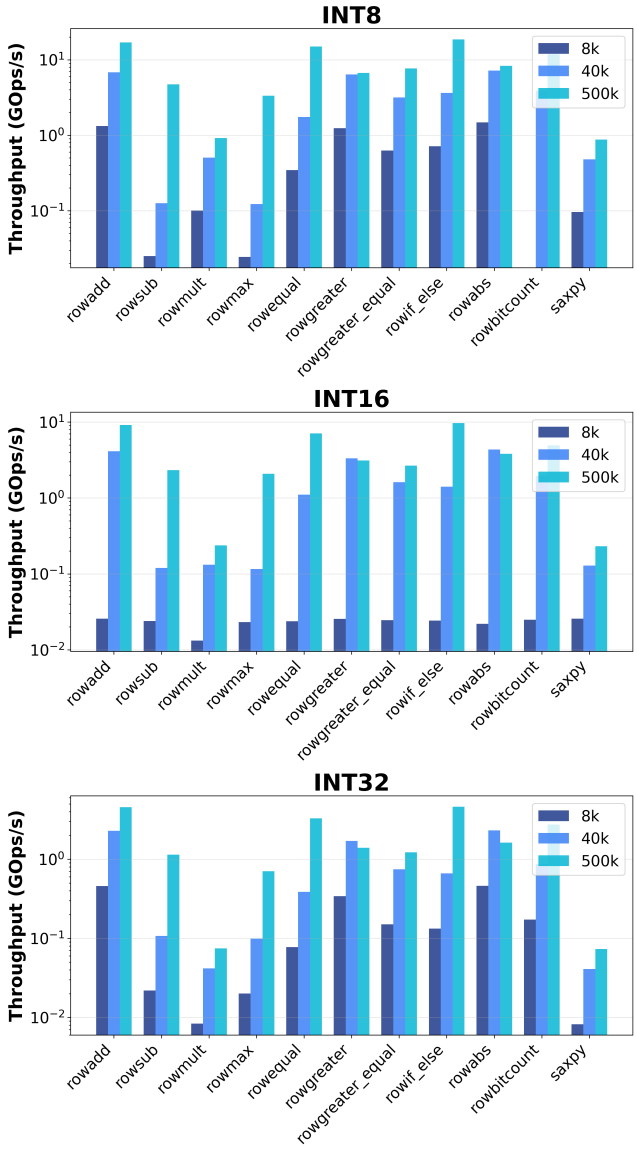


Fig. 4. : Throughput (GOps/s) of selected CIM workloads

The implementation uses a Structure-of-Arrays (SoA) data layout where each feature column is stored as a separate CIM-allocated array. For a dataset with F features and N training samples, F arrays of N elements each are allocated in the same DRAM subarray. The squared Euclidean distance for all training samples is computed simultaneously using CIM operations:

```
// For each feature f:
// diff = query - train
rowsub(diff, query_broadcast[f], train[f], N, BITS);
// sq = diff^2
rowmult(sq, diff, diff, N, BITS);
// dist += sq
rowadd(dist, dist, sq, N, BITS);
```

Each CIM instruction operates on all N training samples in parallel within a single DRAM subarray. The host CPU handles the top-K selection and majority vote, which are inherently sequential and operate on a small number of candidates. The sources of the KNN application are `tests/test-progs/cim/src/combined_knn.cpp`.

V. Conclusion and Future Work

We presented gem5-CIMD, a full-system CIMD simulation framework built on the latest gem5 version (v24). Building upon the MIMDRAM simulator, we ported the infrastructure to the current API and extended it by (1) increasing the number of supported instructions from four bitwise primitives to sixteen CIM instructions spanning arithmetic, relational, conditional, and utility operations, (2) implementing a memory-management stack including novel CIM-specific memory allocation and release instructions that guarantees that CIM operands are co-located in the same DRAM subarray, a constraint imposed by the Ambit scheme, and (3) faithfully modelling the mandatory DRAM refresh commands, which particularly affects execution time when running long microprograms. gem5-CIMD also introduces a CIM standard library which exposes the full instruction set as a C/C++ compiler target. Validation against MIMDRAM confirms cycle-accurate equivalence for common microworkloads. An end-to-end KNN workload demonstrates that the framework can execute real CIM applications with full system emulation.

Building on this foundation, we plan to extend gem5-CIMD in the following directions. The current implementation uses gem5's SE mode; FS mode support would require a custom Linux kernel image with CIM system calls and a device driver managing the huge-page pool, enabling OS-level workloads and multi-process scenarios. Importantly, all the simulated results reported in this paper are for a single bank. We plan to model a more realistic DRAM memory, make the data layouts explicit and configurable, perform CIM in parallel in different banks, and faithfully model the data movement between banks. We also plan to extend gem5-CIMD to support other ISAs (RISC-V and ARM) and other CIMD implementations (PRADA [8] and FCDRAM [7]).

Acknowledgment

This work is partially funded by the AI competence center ScaDS.AI Dresden/Leipzig (01IS18026A-D), and by the German Research Council (DFG) through the HetCIM project (502388442), and the TRR 404 Active 3D (528378584).

References

[1] G. F. Oliveira, A. Olgun, A. G. Yağlıkçı, F. N. 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 computing," in *2024 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 2024, pp. 186–203.

[2] A. Boroumand, S. Ghose, Y. Kim, R. Ausavarungnirun, E. Shiu, R. Thakur, D. Kim, A. Kuusela, A. Knies, P. Ranganathan *et al.*, "Google workloads for consumer devices: Mitigating data movement bottlenecks," in *Proceedings of the twenty-third international conference on architectural support for programming languages and operating systems*, 2018, pp. 316–331.

[3] O. Mutlu, A. Olgun, G. F. Oliveira, and I. E. Yuksel, "Memory-centric computing: Recent advances in processing-in-dram," in *2024 IEEE International Electron Devices Meeting (IEDM)*. IEEE, 2024, pp. 1–4.

[4] D. Lee, B. Hyun, T. Kim, and M. Rhu, "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, 2024, pp. 627–642.

[5] A. A. Khan, J. P. C. De Lima, H. Farzaneh, and J. Castrillon, "The landscape of compute-near-memory and compute-in-memory: A research and commercial overview," *arXiv preprint arXiv:2401.14428*, 2024.

[6] 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 50th Annual IEEE/ACM International Symposium on Microarchitecture*, 2017, pp. 273–287.

[7] İ. E. Yüksel, Y. C. Tuğrul, A. Olgun, F. N. Bostancı, A. G. Yağlıkçı, G. F. Oliveira, H. Luo, J. Gómez-Luna, M. Sadrosadati, and O. Mutlu, "Functionally-complete boolean logic in real dram chips: Experimental characterization and analysis," in *2024 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 2024, pp. 280–296.

[8] H. Shin, R. Park, and J. W. Lee, "A processing-using-memory architecture for commodity dram devices with enhanced compatibility and reliability," in *Proceedings of the 43rd IEEE/ACM International Conference on Computer-Aided Design*, 2024, pp. 1–10.

[9] 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 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, 2021, pp. 329–345.

[10] D. Guo, M. Gholamrezaei, M. Hofmann, A. Venkat, Z. Zhang, and K. Skadron, "Pimsynth: A unified compiler framework for bit-serial processing-in-memory architectures," *IEEE Computer Architecture Letters*, 2025.

[11] X. Peng, Y. Wang, and M.-C. Yang, "Chopper: A compiler infrastructure for programmable bit-serial simd processing using memory in dram," in *2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 2023, pp. 1275–1288.

[12] A. A. Khan, H. Farzaneh, K. F. A. Friebel, C. Fournier, L. Chelini, and J. Castrillon, "Cinm (cinnamon): A compilation infrastructure for heterogeneous compute in-memory and compute near-memory paradigms," in *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 4*, 2024, pp. 31–46.

[13] Y. Shin, J. Park, S. Cho, and H. Sung, "Pimflow: Compiler and runtime support for cnn models on processing-in-memory dram," in *Proceedings of the 21st ACM/IEEE International Symposium on Code Generation and Optimization*, 2023, pp. 249–262.

[14] P. Yang, S. Durvasula, I. Fernandez, M. Sadrosadati, O. Mutlu, G. Pekhimenko, and C. Giannoula, "A tensor compiler for processing-in-memory architectures," *arXiv preprint arXiv:2511.15503*, 2025.

[15] A. Olgun, J. G. Luna, K. Kanellopoulos, B. Salami, H. Hassan, O. Ergin, and O. Mutlu, "Pidram: A holistic end-to-end fpga-based framework for processing-in-dram," *ACM Transactions on Architecture and Code Optimization*, vol. 20, no. 1, pp. 1–31, 2022.

[16] Y. Kim, W. Yang, and O. Mutlu, "Ramulator: A fast and extensible DRAM simulator," *IEEE Computer Architecture Letters*, vol. 15, no. 1, pp. 45–49, 2016.

[17] SU-JonesLab, "NVMain-PIM," https://github.com/SU-JonesLab/NVMain-PIM, 2025.

[18] J. Lowe-Power, A. M. Ahmad, A. Akram, M. Alian, R. Amslinger, M. Andreozzi, A. Armejach, N. Asmussen, B. Beckmann, S. Bharadwaj, G. Black, G. Bloom, B. R. Bruce, D. R. Carvalho, J. Castrillon, L. Chen, N. Derumigny, S. Diestelhorst, W. Elsasser, C. Escuin, M. Fariborz, A. Farmahini-Farahani, P. Fotouhi, R. Gambord, J. Gandhi, D. Gope, T. Grass, A. Gutierrez, B. Hanindhito, A. Hansson, S. Haria, A. Harris, T. Hayes, A. Herrera, M. Horsnell, S. A. R. Jafri, R. Jagtap, H. Jang, R. Jeyapaul, T. M. Jones, M. Jung, S. Kannoth, H. Khaleghzadeh, Y. Kodama, T. Krishna, T. Marinelli, C. Menard, A. Mondelli, M. Moreto, T. Mück, O. Naji, K. Nathella, H. Nguyen, N. Nikoleris, L. E. Olson, M. Orr, B. Pham, P. Prieto, T. Reddy, A. Roelke, M. Samani, A. Sandberg, J. Setoain, B. Shingarov, M. D. Sinclair, T. Ta, R. Thakur, G. Travaglini, M. Upton, N. Vaish, I. Vougioukas, W. Wang, Z. Wang, N. Wehn, C. Weis, D. A. Wood, H. Yoon, and Éder F. Zulian, "The gem5 simulator: Version 20.0+," 2020.

[19] V. Seshadri, Y. Kim, C. Fallin, D. Lee, R. Ausavarungnirun, G. Pekhimenko, Y. Luo, O. Mutlu, P. B. Gibbons, M. A. Kozuch *et al.*, "Rowclone: Fast and energy-efficient in-dram bulk data copy and initialization," in *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*, 2013, pp. 185–197.

[20] A. Olgun, F. N. Bostanci, A. G. Yaglikci, Y. C. Tugrul, G. F. d. O. Bera, H. Hassan, O. Ergin, and O. Mutlu, "DRAM Bender: An extensible and versatile FPGA-based infrastructure to easily test state-of-the-art DRAM chips," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 42, no. 12, pp. 5098–5112, 2023.