The Memory Processing Unit: A Generalized Interface for End-to-End In-Memory Execution

Minh S. Q. Truong[‡†] Yiqiu Sun[†] Dawei Xiong[†] Amol Shah[†] Alexander Glass[‡]
Abraham Farrell[†] James A. Bain[‡] L. Richard Carley[‡] Saugata Ghose[†]

[†]University of Illinois Urbana-Champaign [‡]Carnegie Mellon University

***Abstract*—The *processing-using-memory* (PUM; a.k.a. *in-memory computing*) paradigm aims to eliminate data movement energy and performance costs by using memory cell interactions to directly perform computation. Given PUM's potential for large savings, prior works have proposed many different datapath microarchitectures to demonstrate how general-purpose PUM benefits a wide range of application kernels. Unfortunately, these efforts largely depend on microarchitecture-specific vector-like interfaces that (1) force many of an application's operations to be offloaded to a CPU, (2) require significant programmer effort to scale up applications to an entire memory chip, and (3) make it impractical to develop badly-needed systems software and programming tools for PUM.**

To address these three issues, we propose the *memory processing unit* (MPU), a microarchitecture-agnostic interface layer for general-purpose PUM with three components. First, we develop an MPU instruction set architecture (ISA) with instructions to facilitate application scaling and task coordination. Second, we propose an *ensemble* execution model that coordinates execution across millions of PUM vector function units and maps to most general-purpose PUM microarchitectures. Third, we design a comprehensive MPU control path that efficiently executes MPU ISA binaries across multiple ensembles, and can enable CPU-free execution of complex end-to-end applications with PUM. We demonstrate how the MPU maps to multiple previously-proposed PUM datapaths, and how it achieves average performance/energy improvements of 1.79×/3.23× for 21 data-intensive kernels over these prior works (67×/47× vs. a modern GPU), while also achieving performance and energy improvements for the complex end-to-end applications.

I. Introduction

In the last decade, there has been a surge in the number of data-centric application domains that perform data generation and/or analytics. These applications generate a large amount of data movement between compute elements (e.g., CPUs, GPUs) and data stores (e.g., caches, off-chip main memory, storage), with the movement often dominating overall performance and energy usage [20, 23, 25, 36, 51, 77]. As a result, significant academic and commercial effort has been placed on moving computation closer to these data stores [1, 20, 26, 36–38, 43, 50, 55, 60, 63, 65, 66, 73, 83, 84, 99].

Processing-using-memory (PUM; a.k.a. *in-memory computing* or IMC) [36, 88] aims to eliminate unnecessary data movement costs by using electrical interactions between memory *cells* to perform computation *without* the need for discrete CMOS logic units. (A memory cell is a small device that holds one or more bits of data, depending on the specific memory technology.) PUM takes advantage of the data-parallel nature of many modern applications by performing operations across entire columns (or rows) of data in a memory cell array at once, significantly increasing throughput to compensate for higher latencies. Across the dozens to thousands of cells per column/row in an array, and across dozens to thousands of arrays in a chip, PUM has the potential to perform millions of parallel operations each cycle, without invoking any memory I/O. As a result, PUM can reduce the total energy needed for computation by orders of magnitude.

The promised throughput and energy savings potential has led to the proposal of many different PUM *datapaths* in recent years. Researchers have shown how PUM datapaths can be constructed from arrays of conventional memories (e.g., DRAM [34, 35, 40, 69, 87], SRAM [2, 28, 31, 48], flash memory [3, 15, 33, 80]) as well as from emerging resistive memories (e.g., PCM [22, 52, 62, 70], MRAM [6–9, 47, 68, 71], ReRAM/memristors [10, 10, 19, 24, 24, 32, 39, 41, 42, 45, 57–59, 67, 81, 89, 89, 91, 91, 97, 98, 101–104], domain-wall/racetrack [79]). To enable computation, these datapaths are tightly integrated with the underlying memory technology. For example, DRAM-based PUM datapaths make use of triple-row activation to combine capacitive charge across multiple rows, while ReRAM-based PUM datapaths take advantage of state-dependent voltage divisions among memory cells connected in series to perform logical operations.

However, these PUM datapaths can achieve their theoretical throughput and energy savings only under fairly restrictive constraints that most real applications are unable to achieve in practice. First, applications must have enough data-level parallelism for dependency-free thousand-to-million way parallelism. Many data-centric applications do not have such amounts of parallelism available to exploit, but PUM datapaths are typically designed for only fixed, large degrees of parallelism. Second, applications must be able to perform PUM operations entirely within a single array. Most PUM architectures have only limited capabilities of migrating data between arrays, and poor inter-array interfaces make communication difficult to implement at scale. Third, applications should avoid operations that are not currently possible to perform in memory. The vast majority of modern applications combine data-parallel kernels with interstitial code, often scalar or with limited parallelism, to

perform essential operations such as data-driven control flow. PUM is often incapable of performing scalar operations or sophisticated control flow, causing existing datapaths to rely on an off-chip host CPU to perform these (frequent) operations. From a simplistic study that we perform (Figure 1), even if only one in 80 instructions requires the CPU, this slows down the program by 10.1×, vs. a hypothetical PUM capable of executing without CPU assistance. For typical programs, we estimate that slowdown to be on the order of 30–40×.

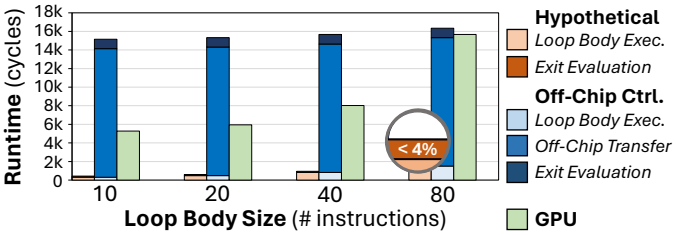


Fig. 1. Simplistic study showing a breakdown of dynamic loop execution time for RACER [97, 98] as the number of loop body instructions (back-to-back `CMPEQ`: compare equality) increases.

As a result, PUM prototypes have currently been limited to accelerating matrix–vector multiplication (MVM) for machine learning (ML) inference [37, 38, 52, 62], as ML models are able to satisfy the above conditions. Unfortunately, this leaves behind the many classes of applications (e.g., graph analysis, databases, genomics, edge analytics, extended reality) that could benefit from bitwise PUM (i.e., PUM that performs Boolean-complete logic [39, 57, 98]) were it not for these constraints. *Our goal is to design low-cost control logic that can eliminate these constraints for most bitwise PUM datapaths.*

To this end, we propose the *memory processing unit* (MPU), a technology-agnostic front end for PUM datapaths. We set three design goals for the MPU:

1) **Support end-to-end application execution.** We want to eliminate PUM datapath dependency on the host CPU, so the MPU needs to support common operations such as scalar computing and data-driven control flow.
2) **Present an easy-to-program abstraction of hardware.** Today, PUM datapaths either require the programmer to possess expert knowledge of the hardware design, or present a simple vector register interface that does not work well for control flow operations or inter-array communication. We want an interface that seems familiar to existing programming paradigms, while simplifying how programmers expose different forms of parallelism to the hardware.
3) **Work with most PUM datapaths.** If done correctly, the MPU can enable development of a software stack for PUM, which has been a substantial need for several years. Because existing interfaces are specialized to datapath microarchitectures and memory technologies, systems developers are hesitant to expend significant effort building a stack for a datapath with no guaranteed future. We want a front end with a consistent hardware-agnostic interface, which can plug into most bitwise PUM datapaths while enabling cross-datapath stack portability and future-proofing.

To achieve these design goals simultaneously, we couple a flexible multi-layer MPU abstraction with a compatible microarchitectural implementation of MPU control path logic.

As Figure 2 shows, our abstraction builds upon the vector interface exposed by many popular bitwise PUM datapaths, which maps memory arrays to multiple *vector register files* (VRFs). In many, but not all, datapaths, not all VRFs can be activated simultaneously due to specific hardware constraints, so we introduce the *RF holder* (RFH) to group together VRFs that share physical constraints (e.g., thermal activation limits, local communication networks). Our MPU runtime dynamically ensures that RFH constraints are always enforced. To support task-parallel programming, we introduce the *ensemble execution model*. An ensemble is a programmer-defined collection of VRFs that will execute the same operations in a kernel. VRFs can be added to an ensemble by a programmer without any concern for hardware constraints or physical location, with the runtime handling task dispatching and scheduling. We demonstrate how (1) this abstraction can be mapped to three very different datapaths (DRAM-based MIMDRAM [78], ReRAM-based RACER [97, 98], SRAM-based Duality Cache [31]); and (2) programmers can significantly simplify how they write PUM programs by using the abstraction, as exemplified with *ezpim*, our advanced assembler.

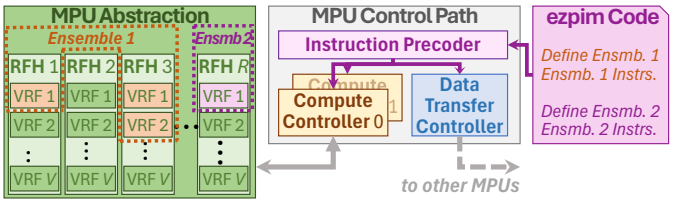


Fig. 2. MPU overview. A VRF corresponds to one or more memory arrays.

To enable the effective execution of programs based on this abstraction, we design the complete microarchitecture of a control path for the MPU, which we synthesize in a 15 nm CMOS technology. This control path (1) translates universal MPU instructions into datapath-specific micro-ops, (2) manages the hardware state of concurrent tasks as they execute on the MPU, (3) supports arbitrarily-nested data-driven control flow through the introduction of a novel SIMD gating mechanism in memory, and (4) enables message passing communications with other MPUs. Table I shows the capabilities of the MPU, compared to CPUs, GPUs, and four state-of-the-art PUM datapaths: Liquid Silicon (LS) [103], Duality Cache [31] (DC), MIMDRAM (MD) [78], and RACER (RC) [97, 98].

We evaluate the MPU across 21 data-intensive kernels for three PUM microarchitectures, and find that it improves performance and energy over their existing designs. As an example, the MPU improves RACER's performance and energy usage by an average of 1.79× and 3.23×, respectively (5.6×/11.3× for kernels with data-driven control flow), with improvements of 67×/47× vs. an NVIDIA GeForce RTX 4090 GPU. To demonstrate the potential of the MPU, we also show how it enables multiple microarchitectures to execute three end-

TABLE I
MPU FEATURES VS. PRIOR PUM DATAPATHS, CPUS, AND GPUS

○: not supported; ●: supported; ◐: partially/potentially supported

Supported Features	LS	DC	MD	RC	CPU	GPU	MPU
Complex Control Instructions							
`if-else` statements	◐	◐	◐	◐	●	●	●
Dynamic loops	○	○	○	○	●	●	●
Subroutine calls	○	○	●	○	●	●	●
Global synchronization	◐	●	○	◐	●	●	●
System-Level Abilities							
Collective communication	○	◐	◐	◐	●	◐	●
Power-density-aware scheduling	○	○	○	○	○	○	●
Runtime micro-op decoding	○	○	●	◐	◐	◐	●

to-end applications, with significant gains over both existing PUM datapaths and the RTX 4090.

We make the following contributions in this paper:

- We propose the MPU front end, which can enable end-to-end application execution across a wide range of digital PUM datapaths and alleviate programming burden at scale.
- We introduce new PUM hardware abstractions that encapsulate scheduling limits and non-contiguous parallel execution.
- We demonstrate how the MPU enables essential systems software such as parallel task coordination, a templated PUM assembler, and thermal-aware task scheduling.

II. BACKGROUND

A. Existing Execution Models

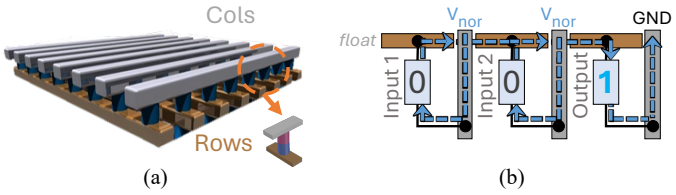
An execution model is a critical part of a computing platform's ISA. The model describes at a high level the hardware components and their runtime behaviors when executing an application. Traditional CPUs make use of what we call a *thread-centric* execution model, which maps threads that execute independent streams of instructions across components such as functional units, register files, and caches. As a comparison, GPUs make use of a *warp-centric* (SIMT) execution model, which can exploit both thread-level parallelism and data-level parallelism by grouping multiple threads into a *warp*, and managing resources and execution state at a warp granularity. The warp-centric execution model scales more efficiently across the thousands of concurrent threads executing in a GPU, and enables new techniques such as memory coalescing, lockstep execution, and warp scheduling to hide long-latency stalls.

B. Enabling Bitwise PUM Micro-Ops

PUM-enabling memory technologies rely on similar mechanisms to realize column-wide logic *in-situ*, by applying technology-specific voltage values to different columns of a memory array. While the basic mechanism is the same across most technologies (turn on multiple columns or rows, and induce an operation) and can be performed using conventional (e.g., SRAM, DRAM, NAND flash) or emerging (e.g., ReRAM, MRAM, PCM) memory technologies, the specific enabling voltages, interactions (e.g., resistance ladders in crossbars vs. charge sharing in DRAM), and resulting micro-ops (e.g., `NOR`, `AND`, `OR`, `NOT`, `IMPLY`, `XOR`) differ from one memory technology to another [54, 87]. We briefly describe how PUM can be performed in SRAM, DRAM, and ReRAM, as these are the underlying technologies of the three datapath microarchitectures that we explore in this work.

For SRAM and DRAM, PUM logic can perform *bitline computation* using *multiple-row activation* [49, 87]. Using DRAM as an example, the memory is organized into two-dimensional arrays, where cells in an array are activated one row at a time. Each cell stores charge representing one bit of data, and shares a bitline with cells in the same column in other rows (but not cells in the same row). When a DRAM cell is activated, its stored charge perturbs a preset voltage on the bitline (which is precharged, i.e., initialized, to half of the base voltage V_{DD}), and the slight shift up or down is detected by the bitline's sense amplifier, which amplifies the shift to a full V_{DD} or *GND* voltage, respectively. If three DRAM cells sharing a bitline are simultaneously activated (which DRAM PUM architectures perform using a triple-row activate, or `TRA`, micro-op), all three cells share their charge with the bitline, and the shift detected by the sense amplifier becomes a *majority vote* among the three cells. If one of these cells is preset to bit value 0, then the majority vote represents an `AND` operation between the other two cells. If one of these cells is preset to bit value 1, then the majority vote represents an `OR` operation between the other two cells. SRAM- and DRAM-based PUM architectures typically augment basic bitline computing with additional wires, CMOS modules, and/or specialized cells to enable additional micro-ops [2, 31, 40, 78, 87] (e.g., adding dual-contact cells to enable `NOT` for Ambit-based DRAM PUM [87]).

In contrast, emerging memory devices such as ReRAM are often organized into crossbars (Figure 3a), which allows for current to flow from multiple columns into the same row line (i.e., wire). Figure 3b shows how ReRAM can perform a `NOR` micro-op on an entire column of operands. For each row, by applying a voltage V_{nor} on two input cells and grounding the output cell, a current is induced that flows from V_{nor} to ground, changing the output cell's value based on the `NOR` of the bits in the two input cells.



(a) (b)

Fig. 3. (a) ReRAM crossbars with memory devices at intersection of wires; (b) asserting specific voltages to perform a `NOR` micro-op.

C. Bitwise PUM Datapath Microarchitectures

Bitwise PUM datapaths aggregate basic PUM micro-ops into more complex CPU-like instructions. While the datapaths differ significantly [54], they typically share two common traits: (1) they make use of bit-serial computation [14], and (2) they map columns (or rows) of each memory array to a *vector register* and organize these registers into one or more *vector*

register files (VRFs). Since bit-serial computation results in long instruction latencies, bitwise PUM datapaths make use of vectorization to compute across many operands in parallel. For motivation, we briefly discuss the design of two state-of-the-art bitwise PUM datapaths.

MIMDRAM. A DRAM chip is split up into multiple *subarrays* that each consist of multiple *mats* (i.e., small memory arrays). MIMDRAM [78] executes instructions on data stored within a mat, where all data bits of a vector register are mapped to a single mat. As shown in Figure 4b in Section IV, a subarray contains multiple μProgram processing engines (μPEs) that can execute a stream of vector instructions across adjacent DRAM mats. An active μPE issues an instruction (a *bbop* in MIMDRAM terminology) to the mats assigned to that μPEs. The *bbop* is broken down into triple-row activate (TRA) operations (MIMDRAM micro-ops), which simultaneously turn on multiple rows in the mat to share charge and perform a Boolean logic operation (e.g., `AND`, `OR`, `NOT`). The TRAs are issued to corresponding mats using a scoreboard scheduler.

RACER. RACER [97, 98] extends bit-serial vector computing into a *bit-pipelined* execution model, as shown in Figure 4a. Each *n*-bit operand in a vector register is striped by bit position across *n* different ReRAM memory *tiles* (e.g., tile 0 holds bit 0, tile 1 holds bit 1). As a result, one vector register spans *n* columns in *n* different tiles. Tiles belonging to the same vector registers are grouped into a RACER *pipeline*, and are interconnected using single-column *buffers* that act like pipeline registers. As a result, RACER can perform vector micro-ops (e.g., `NOR`) on bit 0 in tile 0, pass a carry-out value to tile 1, and operate on bit 1 while allowing tile 0 to work on a completely separate stream of micro-ops. Because bit-serial operations are often repeated for each bit, RACER contains *pipeline control circuitry* (PCC) that passes micro-ops from one tile to the next in a pipeline.

III. Memory Processing Unit: Overview

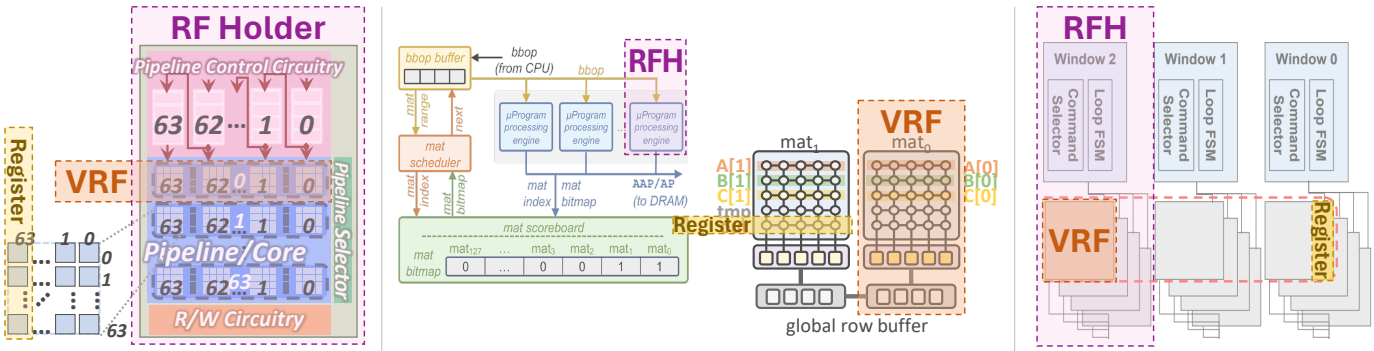
Today, each bitwise PUM datapath provides its own unique view of the hardware to the programmer. These views simultaneously (1) force programmers and system software to highly specialize binaries to the underlying datapath microarchitecture (e.g., knowing which memory arrays are physically co-located in a chip, fixing the width of vector instructions), preventing program and toolchain reuse across different datapaths; and (2) significantly restrict the datapaths to supporting the execution of only a limited number of kernels (e.g., highly parallel kernels with little to no control flow divergence). The first issue has hindered any meaningful development of sophisticated software development toolchains. The second issue has forced most end-to-end PUM-friendly applications to repeatedly alternate their execution between in-PUM kernels and short host CPU segments that perform control flow evaluation.

We introduce the *memory processing unit* (MPU; see Figure 2), a new front-end layer for bitwise PUM that, with careful design and flexibility, solves both issues. The MPU consists of a lightweight runtime to manage execution state across an entire memory chip, along with modest hardware support to improve the efficiency of the runtime and abstraction. The MPU makes four key decisions about the programmer–datapath interface, building upon a low-level mapping of *vector register files* (VRFs) to physical memory arrays.

First, *we want programmers to be able to express flexible degrees of parallelism in their code.* While existing PUM datapaths expose their underlying memory cells as vector registers, they currently force programmers to write individual vector instructions for each pair of vector registers, with no support for vector lane masking. Given that the vector width in PUM ranges from dozens to thousands of lanes, while data-level parallelism is rarely of a constant width in real programs, this makes programming complex: for small widths, programmers must manually repeat vector instructions many times in the code; for large widths, they will need to pad data (often to different degrees for separate instructions), wasting PUM data capacity and throughput. To avoid this, the MPU introduces (1) the *ensemble execution model*, where a programmer can dynamically regroup memory arrays at any point in a program, based on the application's actual needs; and (2) hardware/software support for efficient lane masking. Essentially, an ensemble is a programmer's way of telling the MPU hardware and runtime which VRFs are executing the same instructions, and can *potentially* execute simultaneously. This allows programmers to express arbitrary levels of parallelism (even scalar, i.e. single-operand, execution) that change over time, in a way that reduces code redundancy.

Second, *we must ensure that MPU program execution obeys all real-world hardware constraints, without asking the programmer to track this*. For example, RACER [97] has at least two such constraints: (1) it limits the number of active pipelines per cluster to stay within thermal density limits, and (2) it exposes a non-uniform memory access (NUMA) interface for cluster-to-cluster communication. Instead of forcing programmers to know and explicitly manage each of these constraints, the MPU introduces the *register file holder* (RFH), a generic abstraction that embodies all such hardware constraints. At design time, the system developer defines RFH mappings in hardware based on whatever constraints are required by a particular microarchitecture (see Section IV), and includes constraint management code in the MPU runtime. During program execution, the runtime enforces hardware constraints, by managing which RFHs in an ensemble can *actually* execute simultaneously to stay within the designer-defined constraints, taking this burden away from the binary.

Third, *we want to maximize the portability of MPU binaries*. The ensemble/RFH combination allows MPU applications to avoid encoding hardware constraints into the binary. We also aim to eliminate datapath-specific instructions, so that a programmer (and, eventually, a compiler) can generate binary instructions that can execute across many different datapaths. To avoid datapath-specific information, we carefully study the instructions currently exposed by popular bitwise PUM datapaths, and replace them with a common MPU ISA. Already, most PUM datapaths convert assembly instructions into low-



(a) RACER (b) MIMDRAM (c) Duality Cache

Fig. 4. Mapping RF holders and VRFs to hardware for (a) RACER, (b) MIMDRAM, (c) Duality Cache. Datapath figures reproduced from original papers.

level technology-specific *micro-ops* (e.g., NOR), and include some form of instruction-to-micro-op decoding; we propose to replace this with a universal decoder in hardware that translates MPU instructions into datapath-specific micro-ops.

Fourth, as much as possible, *we want MPU applications to perform control flow inside PUM hardware to avoid frequent off-chip CPU requests*. In the MPU control path, we introduce light hardware support that (1) significantly improves the granularity of control flow (even making scalar execution possible), and (2) efficiently tracks loop conditions. With this support, which builds on top of our per-lane masking ability, it is possible for end-to-end PUM applications to avoid the need for CPU-side control flow entirely.

IV. Integrating the MPU With PUM Datapaths

In this section, we first demonstrate how the MPU's *RF holder* and *VRF* abstractions can be mapped to different PUM datapaths (i.e., microarchitectural back ends), as shown in Figure 4. We then show how the MPU leverages the RF holder mapping to schedule the execution of ensembles while adhering to PUM-specific hardware constraints.

When designers integrate the MPU front end with a bitwise PUM datapath, they must appropriately map vector register files (VRFs) and register file holders (RFHs) to datapath hardware. We demonstrate three example mappings for RACER, MIMDRAM, and Duality Cache. A VRF is expected to map to one or more physical memory arrays in the datapath, with the designer ideally picking the smallest collection of arrays and associated peripheral components capable of vector register access. An RFH is expected to map to one or more VRFs, but its specific mapping is hardware design dependent, so designers must use it to encompass two types of constraints.

First, a designer should identify which constraints require runtime scheduling and/or throttling. For example, Figure 5 shows how thermal dissipation can limit the number of activated memory arrays per unit area in several PUM datapaths [45, 78, 98, 103]. For these datapaths, an RFH should contain multiple VRFs, where only one VRF can be activated at a time, and where the total VRF count ensures that the datapath always remains within thermal limits.

Second, the designer should identify any shared hardware components across VRFs that inherently constrain parallel

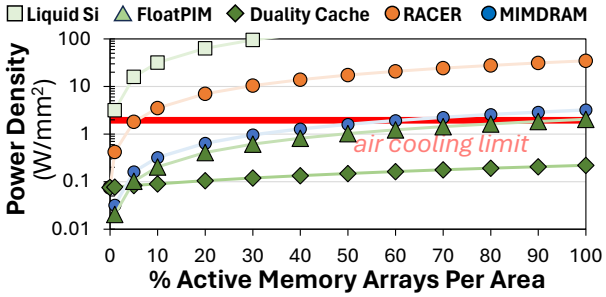


Fig. 5. Power density of PUM datapaths vs. active memory arrays.

execution. These components can include control units, peripheral circuitry, or network components, and are often shared by VRFs *in close physical proximity with each other*. For example, Duality Cache [31] does not suffer from thermal throttling in Figure 5, but is rate limited because nearby memory arrays cannot execute different instruction streams simultaneously, due to the limited number of instruction controllers.

RACER. In RACER, we map each pipeline to its own VRF. While our VRF spans multiple tiles (see Section II-C), the MPU does not have to manage tile execution individually, as RACER's pipeline control circuitry (PCC) coordinates a bit-pipelined instruction across these tiles in hardware. A w-bit word is striped across the tiles of a pipeline so that each bit occupies the same row and column in a tile. With this organization, an n-element vector register is mapped to w columns (each containing n rows) of the same address across w tiles (i.e., vector register i is mapped to Column i across all tiles). As discussed previously, pipelines/VRFs in close proximity cannot be activated at the same time because of thermal dissipation limits. Conveniently, RACER groups 64 pipelines into a single unit called a *cluster*, with pipelines in a cluster sharing one PCC. To adhere to thermal limits, we map an RFH to a single RACER cluster (Figure 4a), limiting the number of active pipelines per cluster.

MIMDRAM. MIMDRAM (see Section II-C) stores a w bit word across w consecutive memory cells in a single DRAM mat. Thus, vector register i maps to Columns $i{*}w$ through $i{*}w{+}w{-}1$, and a VRF maps to a single DRAM mat (Figure 4b). Like RACER, MIMDRAM cannot activate all VRFs in the same area due to thermal dissipation constraints. Thus, each of its

μPEs is mapped to its own RFH because each of them controls a group of mats/VRFs that are physically nearby. Micro-ops (TRAs) need to be serially applied to the w columns that store the word.

Duality Cache. The organization of Duality Cache is similar to that of MIMDRAM, with each *issue window* containing a *loop finite state machine* (FSM) that is used to deliver the micro-ops to the correct bitline. In this microarchitecture, vector register i maps to Columns $i*w$ through $i*w+w-1$, and a VRF maps to a single SRAM subarray (Figure 4c). Unlike MIMDRAM, each issue window hardware is directly connected to a specific group of SRAM subarrays, which cannot be activated simultaneously. Thus, we map an RF holder to each issue window, and the MPU can leverage the loop FSM as the vector mapper.

V. Writing an MPU Program

At the heart of MPU programming are ensembles, which as we discuss in Section III allow programmers to dynamically group VRFs together when they are executing the same task. Figure 6 shows an example of how a program can instantiate an ensemble using our MPU ISA (32-bit instructions, 64-bit data). For each block of a program (where the length of a block is left up to the programmer), the programmer instantiates an ensemble and associates it with one or more VRFs. Unlike the RFH, the VRFs in an ensemble can be located anywhere within the MPU, and need not be physically adjacent. We introduce two types of ensembles in the MPU: (1) *compute ensembles*, which enable the execution of MPU instructions across a programmer-defined ensemble; and (2) *transfer ensembles*, which enable synchronization and memory-consistent data communication. Our control path hardware and runtime (Section VI) handle ensemble execution and state management.

```
// Compute Ensemble 1
 1: COMPUTE RFH1 VRF1          header
 2: COMPUTE RFH3 VRF1
 3: COMPUTE RFH3 VRF2
 4: ADD r0 r1 r2               body
 5: SUB r2 r3 r4
 6: COMPUTE_DONE               footer

// Compute Ensemble 2
 7: COMPUTE RFH2 VRF1
 8: MUL r0 r1 r2
 9: MAC r0 r3 r4
10: COMPUTE_DONE

// Transfer Ensemble
11: MOVE RFH1 RFH2
12: MOVE RFH2 RFH3
13: MEMCPY r0 VRF0 r0 VRF0
14: MEMCPY r1 VRF0 r1 VRF0
15: MOVE_DONE
// Inter-MPU Communication
16: SEND MPU4
17: MOVE RFH1 RFH4
18: MEMCPY r1 VRF1 r1 VRF2
19: MEMCPY r1 VRF1 r1 VRF2
20: MOVE_DONE
21: SEND_DONE
```

Fig. 6. Ensemble-based example code. Back end abstraction shown in Figure 2.

A. Compute Ensembles

As shown in Lines 1–10 of Figure 6, a compute ensemble consists of three parts, and we introduce three instructions into the MPU ISA for ensemble management (Table II). The header of the compute ensemble uses one or more `COMPUTE` instructions (one per VRF) to select which VRF(s) the ensemble should use. The body consists of arithmetic instructions to be executed by all VRFs in the compute ensemble. In a departure from other data-parallel execution models, such as the warp-centric GPU SIMT model, the constituent VRFs do not assume *any* concurrent execution, enabling greater MPU scheduling flexibility.[1] The footer has a single `COMPUTE_DONE` instruction to indicate the end of ensemble execution.

Our approach allows programmers to treat compute ensembles as lightweight threads. Similar to conventional threading, the MPU does not explicitly manage dependencies across concurrent compute ensembles, resulting in shared-memory-like interleaving semantics when two ensembles access the same vector register. To allow programmers to explicitly manage dependencies between compute ensembles, we introduce an `MPU_SYNC` instruction, which is a fence for compute ensembles.

B. Transfer Ensembles

The MPU ISA includes a second kind of ensemble, the *transfer ensemble* (e.g., Lines 11–15 in Figure 6), to transfer data between VRFs. A transfer ensemble's header contains one or more `MOVE` instructions, where each instruction sets up a source RFH and destination RFH pair. For PUM datapaths with circuit-switched networks, these instructions can also set up network paths prior to performing data transfers. In the body, each `MEMCPY` instruction copies one vector register from a source VRF to a register in a destination VRF *for each RFH pair*. The footer consists of a single `MOVE_DONE` instruction.

Unlike compute ensembles, transfer ensembles must guarantee memory consistency, as a VRF can read another VRF's data during the transfer. Because PUM datapaths currently only support in-order execution, we make use of sequential consistency [61], where the MPU guarantees that instructions are executed to completion and in the order specified by the ensemble. To enforce consistency, an MPU executes only one transfer ensemble at a time. Across multiple MPUs, we enforce consistency by employing an explicit message-passing interface (Lines 16–21 in Figure 6), which also enables scalable inter-MPU communication. Inter-MPU messages are set up using `SEND` and `SEND_DONE` instructions in the sender MPU, and `RECV` in the receiving MPU, with these instructions able to realize complex communication patterns such as gather–scatter and broadcasting. To guarantee deadlock avoidance, we force MPUs with lower MPU IDs to `SEND` first, and break circular dependencies across concurrently executing transfer ensembles using our runtime.

C. Simplifying Programming With ezpim

Existing PUM datapaths are capable on their own of executing only relatively simple kernels, and rely on an external host processor to handle complex control decisions during full application execution. We design the MPU to eliminate the need for the CPU, by introducing six control instructions that programmers can use to express multi-level `for` and `while` loops, `if/else` statements, and subroutine calls (see Table II).

[1]We consciously avoid the warp-centric model for two reasons. First, it would require branches and enable signals to be evaluated by *all* VRFs, incurring significant delays due to (1) the number of VRFs, (2) thermal limits preventing all VRFs from executing concurrently, and (3) the often-slow nature of PUM condition evaluation (e.g., `CMPEQ`, compare equality). Second, it would become difficult to enable more than one ensemble/kernel concurrently, as the data store *is* the processing element, and there is no need for (or way to mimic) latency hiding techniques that assign multiple warps to a GPU core.

TABLE II
MPU BASE INSTRUCTION SET

Instruction	Description
Ensemble Deployment	
`COMPUTE <rfhID> <vrfID>`	Demarcate start of an ensemble, activate VRF `vrfID` of RFH `rfhID`
`COMPUTE_DONE`	Demarcate the end of an ensemble
`MPU_SYNC`	Sync all deployed ensembles, wait until complete to proceed
`MOVE <rfhSRC> <rfhDES>`	Demarcate the start of a move block with source RFH `rfhSRC` and destination `rfhDES`
`MOVE_DONE`	Demarcate the end of a move block
Inter-MPU Communication	
`SEND <mpuDES>`	Send an execution block to `mpuDES`
`SEND_DONE`	Demarcate the end of `SEND`
`RECV <mpuSRC>`	Service an ensemble coming from `mpuSRC`
Control Flow Instructions	
`GETMASK <rd>`	Get mask bits from mask register, put in `rd`
`SETMASK <rs>`	Copy `rs` to mask register, start predication
`UNMASK`	Stop predicated execution
`JUMP_COND <lineNum>`	Jump to `lineNum` if mask register returns all logic 0s, else go to next line
`JUMP <lineNum>`	Jump to `lineNum`
`RETURN`	Return to next line of code after where `JUMP` was called
`NOP`	Do nothing (i.e., insert a bubble)
Arithmetic Instructions	
`ADD <rs>, <rt>, <rd>`	Two's complement add (`rd` = `rs` + `rt`)
`SUB <rs>, <rt>, <rd>`	Two's complement subtract
`INC <rs>, <rd>`	Increment a number by 1 (`rd` = `rs` + 1)
`INIT0 <rd>`	Initialize `rd` with 0
`INIT1 <rd>`	Initialize `rd` with 1
`MUL <rs>, <rt>, <rd>`	Multiply (only 8-/16-/32-bit inputs)
`MAC <rs>, <rt>, <rd>`	Multiply–accumulate (`rd` += `rs` $\times$ `rt`)
`QDIV <rs>, <rt>, <rd>`	Division that returns quotient
`QRDIV <rs>, <rt>, <rd>`	Division that returns quotient in `rt` and remainder in `rt` (overwriting register)
`RDIV <rs>, <rt>, <rd>`	Division that returns remainder
`POPC <rs>, <rd>`	Population count
`RELU <rs>, <rd>`	Rectified linear unit
Comparison & Search Instructions	
`CMPEQ <rs>, <rt>`	Check equality (result in conditional register)
`CMPGT <rs>, <rt>`	Check `rs` > `rt` (result in conditional register)
`CMPLT <rs>, <rt>`	Check `rs` < `rt` (result in conditional register)
`FUZZY <rs>, <rt>, <rd>`	Fuzzy comparison, skipping bits set in `rd` (result in conditional register)
`CAS <rs>, <rt>`	Compare and swap
`MUX <rs>, <rt>, <rd>`	Multiplex (i.e., choose) `rs` or `rt` based on bitmask in `rd`
`MAX <rs>, <rt>, <rd>`	Returns larger number
`MIN <rs>, <rt>, <rd>`	Returns smaller number
Boolean & Bit Manipulation Instructions	
`AND <rs>, <rt>, <rd>`	Bitwise AND
`NAND <rs>, <rt>, <rd>`	Bitwise NAND
`NOR <rs>, <rt>, <rd>`	Bitwise NOR
`INV <rs>, <rd>`	Bitwise NOT
`OR <rs>, <rt>, <rd>`	Bitwise OR
`XOR <rs>, <rt>, <rd>`	Bitwise XOR
`XNOR <rs>, <rt>, <rd>`	Bitwise XNOR
`BFLIP <rs>, <rd>`	Reverse the order of bits
`LSHIFT <rs>, <rd>`	Left shift by 1
Data Movement Instructions	
`MEMCPY <vrfSRC> <rs>, <vrfDES> <rd>`	Copy vector register contents across VRFs (only possible during a move block)
`MOV <rs>, <rd>`	Copy vector register contents within a VRF

To reduce programmer burden in writing assembly-level control instructions, we develop a Python-based advanced assembler called *ezpim*, which can integrate MPU ISA instructions with control semantics similar to those found in high-level languages. With this support, our MPU enables a single PUM chip to perform standalone execution of end-to-end programs. We hope that future works can build upon ezpim to develop a full compiler, and envision that our work can enable PUM programs to make use of popular parallel programming frameworks such as OpenMP and MapReduce.

Comparison-Based Predication. A key issue for the programmability of SIMD architectures such as our vector-based datapaths is providing support for control flow constructs commonly found in high-level programming languages (e.g., `if`–`else` conditionals, `for` and `while` loops). A typical SIMD architecture uses a single instruction to compute on multiple data elements [30]: for fixed-width vector processing, a vector is treated as an array of data elements where each element is executed in its own *lane*. Vector-based datapaths often implement some form of predication [13], which enables architectural support to *mask* operations on a per-lane basis, by mapping a bitmask (i.e., the predicate) to the enable logic of each lane. To execute an `if`–`else` conditional on a fixed vector, where the conditional depends on data stored within a vector, the datapath executes both the `if` and the `else` body on the vector. With predication, the datapath applies a mask based on the outcome of the condition evaluation to correctly enable those lanes participating in the `if` body (while disabling the other lanes), and then inverts the mask to enable the lanes participating in the `else` body.

The MPU extends predicated execution with hardware-assisted support for dynamic loops. To facilitate this in the ISA, the MPU introduces a *conditional register*, which holds the per-lane bitmask resulting from the execution of our comparison instructions (see Table II). The contents of the conditional register (or any data register in the MPU) can be used to control hardware (Section VI-B) that (1) enables and disables individual vector lanes in the datapath, and (2) can allow for lane divergence during dynamic loops and detect when all lanes have exited the loop. While the lane masking logic sits in the MPU control path, we provide programmer support to read out the current lane mask, so that it can be modified in the datapath (e.g., to support arbitrary loop/branch nesting). We discuss how programmers can use this support below.

Dynamic Loops. Existing datapaths can evaluate static loops using loop unrolling, but this (1) does not support dynamic data-driven loop conditions (i.e., when the iteration count is unknown at compile time); (2) often requires significant manual intervention; and (3) exacerbates binary capacity pressure. The MPU avoids this by providing hardware/software support for dynamic loops. In the MPU ISA, we introduce the `JUMP_COND` instruction, which uses program-generated masks to decide which lanes should participate in the next iteration of a loop. The MPU control path contains a *mask register* that holds a bitmask, where each bit corresponds to a vector lane, and controls whether the lane is enabled or disabled. When `JUMP_COND` is invoked, the control path uses this mask to determine which vector lanes have reached the loop termination condition (using hardware support shown in Figure 7d; see Section VI-B for details), and advances past the loop when all lanes are disabled. With *ezpim*, programmers can use

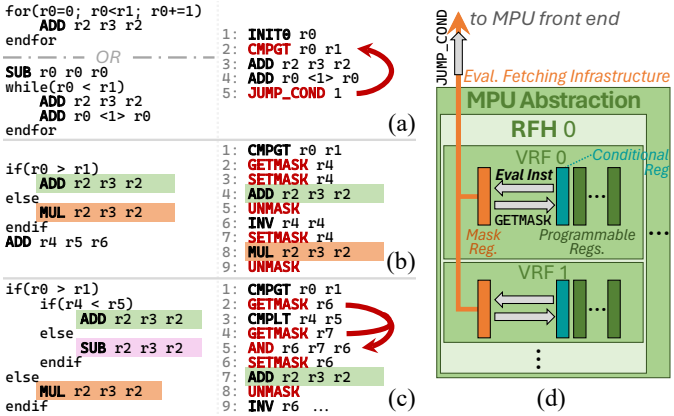


Fig. 7. *ezpim* code (left) and MPU ISA output (right) for (a) `for`/`while` loops, (b) branches, (c) nested branches; (d) MPU complex control support.

conventional `for` and `while` loops as shown in Figure 7a, which *ezpim* automatically converts into conditional evaluation and `JUMP_COND` instructions.

Branches. We can reuse the lane masking support to implement per-lane branching for a VRF. To support inline branching, we introduce the `SET_MASK` instruction, which retrieves a comparison result from a VRF register and loads it into the VRF's mask register. This allows multiple use cases, including (1) evaluating a branch condition several instructions before performing the branch, (2) statically setting the branches instead of using a branch condition, and (3) enabling arbitrary levels of branch nesting. Figure 7b shows how *ezpim* can translate in-assembly `if` and `else` commands written by a programmer into the MPU's masking instructions. If a branch mask needs to be updated (e.g., to reflect a new branch nesting level), the `GET_MASK` instruction copies the current lane mask's contents into a regular VRF register (disabling the mask lane control to ensure that all bits of the mask are copied), at which point the MPU can perform arbitrary computation to update the mask (Figure 7c). To exit the branch, the `UNMASK` instruction re-enables all lanes by setting all mask bits to 1.

Subroutine Calls. To support subroutines, we introduce a `JUMP` instruction into the MPU ISA. Unlike `JUMP_COND`, `JUMP`'s targets are not limited to the current ensemble, and can be anywhere in the binary where the subroutine call lies. The `JUMP` instruction saves the current PC into a return address stack in the control path hardware, and we add a `RETURN` instruction that pops an address from the stack upon a subroutine exit. In *ezpim*, programmers can simply define and call subroutines, and the assembler will add in the correct `JUMP`/`RETURN` instructions.

VI. MPU Control Path Hardware & Runtime

In this section, we discuss how our example control path hardware and runtime software can execute MPU code on bitwise PUM datapaths. Figure 8 shows the components of our control path: (1) a *precoder*, which stores the binary and distributes instructions to specific controllers; (2) one or more *compute controllers* (CCs), which manage the runtime state of compute ensembles; and (3) a *data transfer controller*, which handles transfer ensembles and inter-MPU communications.

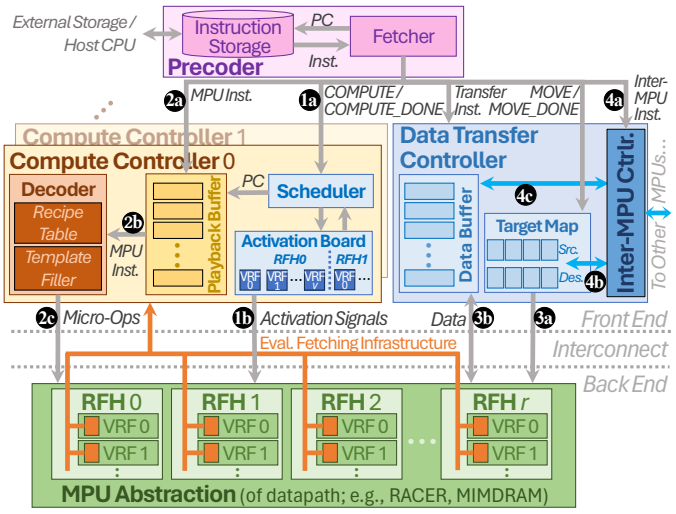


Fig. 8. MPU control path hardware for an abstracted datapath.

A. Precoder

The precoder includes an *instruction storage unit* (ISU) that stores a program binary on chip, and a *fetcher* unit that uses a program counter to retrieve the next instruction(s) from the ISU. Note that the ISU can be implemented using many different memory technologies (e.g., the same types of arrays being used for PUM data in the datapath; SRAM-based caches), with the specific choice left to the hardware designer based on platform needs. If an MPU binary exceeds the capacity of an ISU, the runtime can borrow capacity in the ISU of nearby MPUs. (In practice, we find that thanks to *ezpim* and the MPU ISA, all of our binaries fit within a single ISU.) The fetcher uses metadata from ensemble headers and footers to determine which control units it should distribute ensemble body instructions to.

B. Compute Controller

The compute controller (CC) unit executes one compute ensemble at any given time. At the start of an ensemble, the fetcher assigns a CC to the ensemble, and issues metadata to the CC (1a in Figure 8), which the CC uses to enable specific VRFs in its *activation board* (1b). Each CC's activation board has an enable bitmask for every VRF in the MPU. Once ensemble body execution starts, the fetcher issues compute instructions to the CC for the VRFs to execute. The CC commits the instructions to its *playback buffer* (2a), which allows the CC to replay a sequence of instructions multiple times if hardware constraints prevent full VRF concurrency (Section VI-C), or if the instructions contain a dynamic loop (Section V-C).

For each sequence replay (as determined by the scheduler; Section VI-C), the CC issues instructions one at a time from the playback buffer to the *instruction-to-micro-op decoder* (I2M). While different back-end datapaths have different micro-ops (Section II-C), we design a universal I2M that can support any of these. The I2M translates an instruction into m micro-ops, where the value of m depends on the specific instruction and the available micro-ops. For some PUM datapaths, a single instruction can expand into hundreds, if not thousands, of micro-ops, placing significant pressure on micro-op decoding.

To address this, the I2M uses a *recipe table* implemented as a parallel lookup table. The recipe table stores *recipes* (micro-op sequence templates) for each compute instruction, where the recipe includes micro-ops but not specific register addresses. A *template filler* in I2M uses information about the active VRFs to populate VRF-specific addresses into each micro-op in the recipe. The I2M then dispatches the filled-in micro-ops to the back end, where it is executed by the activated VRFs (2c).

The ensemble execution model allows multiple ensembles from the same program to exist concurrently. Thus, multiple CCs can exist in the MPU control path. Once the fetcher finishes issuing an ensemble subsequence to a CC, it can start issuing another subsequence from a different ensemble to a new CC. Realistically, the number of CCs an MPU can support depends on the sizes of the playback buffer and recipe table, the two most area-/power-intensive components.

Recipe Table Optimizations. While the recipe table reduces I2M pressure to a degree, a pressure point remains in that the table's capacity is practically limited to a few thousand micro-op templates. We propose three mechanisms to relieve capacity constraints, as shown in Figure 9. First, as we observe that multiple instructions often share portions of their recipes (e.g., ADD and MAC), we can add a *pointer table* that allows a recipe to point to common recipe subsequences. Second, we can include a *template lookup table*, which stores pointers to recipes in binary storage, and can dynamically cache recipes into the recipe table once an instruction is issued. Third, we can share recipe table hardware across multiple CCs.

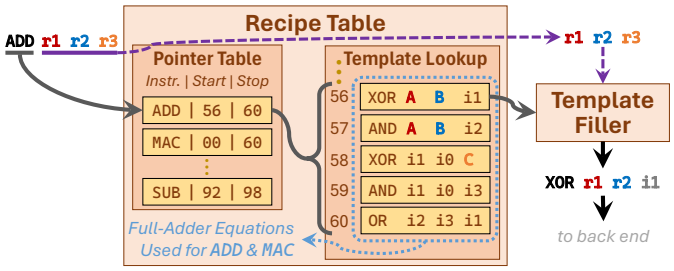


Fig. 9. Example of an optimized recipe table implementation.

Complex Control Flow. To support the control flow in Section V-C, we use an observation that many bitwise PUM datapaths add independent voltage assertion units to each row of a memory array, in order to isolate the electrical interactions of each row. The MPU leverages these units to implement vector lane masking: we add a *mask register* to each VRF in the datapath, which sits at the voltage supply lines to the memory arrays and contains one control bit per lane. For each lane (e.g., a memory array row), the mask register chooses whether the lane receives a voltage assertion required for the active operation, or is disabled (i.e., power gated).

Our in-VRF masking allows us to implement efficient hardware for data-driven predicated execution of arbitrary nesting depth. The SETMASK instruction can retrieve a bitmask from either (1) the conditional register (Section V-C) or (2) one bit of data from each element in a vector register, and copy this into the mask register to enable/disable lanes. To support control instructions such as JUMP_COND, which use the mask register contents to determine whether to continue loop iteration, we add logic to the CC called the *evaluation fetching infrastructure* (EFI; Figure 7d). The EFI sits at the interface between the CC and the datapath, and when the CC executes a control instruction such as JUMP_COND, it uses the EFI to copy the contents of the mask register into the CC and determine if any of the lanes remain enabled. If at least one lane is enabled, the EFI notifies the scheduler to update the program counter to the jump target and continue issuing micro-ops. If all lanes are disabled (i.e., the mask contains all 0s), then the jump is not taken, and the scheduler proceeds to the next instruction.

C. Scheduling Algorithm & Hardware

A key constraint in many PUM datapaths is that their power density scales proportionally to their throughput. When coupled with the density of modern memories, an unrestricted PUM datapath can easily exceed safe air cooling limits [44] and cause irreversible hardware damage. To ensure safe operation, the MPU scheduler uses vendor-provided data on the relationship between the fraction of active VRFs per RF holder, the instruction type, and expected power density, to cap the total number of activated VRFs.

Figure 10 describes the MPU scheduling algorithm in detail. We implement the scheduler in hardware, but the MPU model allows for software-based schedulers as well. As an ensemble activates VRFs for execution, the scheduler tracks the number of activated VRFs per RFH across all currently executing ensembles (Lines 3–5). If any RFH reaches its constraint-defined maximum active VRF count, the remaining VRFs for that RFH are placed in a standby queue. Once all currently active VRFs complete execution, the compute controller raises a hardware exception to the scheduler (Line 11), and if any

```
1  while True:
2    # activate all VRF addresses in each per-RFH queue
3    for queue in rf_active_queues:
4      for vrf in queue:
5        activate(vrf)
6    serve = serve_interupt_playback_full()
7    # reactivate same VRFs for next ensemble body segment
8    if (serve):
9      jump 2
10   # at ensemble end; restart if there are queued VRFs
11   serve = serve_interupt_footer()
12   if (serve):
13     if (len(waiting) != 0):
14       for (active, waiting) in
15           zip(rf_active_queues, waiting_queues):
16         active.clear()
17         thermal_counter = 0
18         # only activate a certain number of VRFs
19         if (thermal_counter < limit):
20           active.append(waiting[0])
21           waiting.pop()
22           thermal_counter++
23       jump 2
24       # if nothing left to activate, retire ensemble
25       else:
26         notify_ensemble_done()
```

Fig. 10. MPU thermal-aware scheduling algorithm.

VRFs are on the standby queue, the scheduler (1) deactivates the just-completed VRFs, (2) activates the VRFs on the queue (again enforcing RFH limits), and (3) executes the ensemble on the newly activated VRFs.

The runtime and scheduling algorithm can also assist with binary portability. To a degree, MPU binary portability is similar to that of GPU kernels: while VRFs, RFHs, and ensembles abstract away hardware specifics, the number of VRFs per RFH is specific to a datapath. To allow for portability, we encode the compile-target VRFs-per-RFH parameter in the binary, and the MPU runtime can perform some degree of RFH/VRF-to-MPU remapping if the target hardware uses a different parameter (provided enough resources are available). As is the case with GPUs, we envision that MPU binaries can benefit from some degree of autotuning support (though the search space is likely significantly smaller in scope than with GPUs).

D. Data Transfer Controller

The data transfer controller (DTC) handles the execution of one transfer ensemble at any given time. At the start of a transfer ensemble, the fetcher sends metadata from the header to the DTC, which uses the information to configure its *target map*. The target map stores pairs of source and destination RFH/VRF addresses. Once the fetcher starts sending data transfer instructions from the ensemble body, the DTC performs the transfers on one or more of the pairs in the target map (3a in Figure 8), with the number of concurrent transfers dependent on the datapath's underlying network. Optionally, the DTC may contain a *data buffer* for intermediate storage during a transfer (3b). A data buffer can help in three cases: (1) facilitating long-distance transfers, (2) enabling data pre-processing for complex transfer instructions (e.g., broadcast, transpose, vector shift, data transpose), and (3) inter-MPU message buffering.

Inter-MPU Controller. This unit is activated when a `SEND` or `RECV` message-passing instruction is fetched (4a). When sending a message to another MPU, the inter-MPU controller first requests the necessary data from the back end via the target map (4b), waits for the data to be populated in the data buffer, and then sends the data to the destination MPU (4c).

VII. Methodology

We evaluate the MPU for three PUM datapaths:

- ReRAM-based RACER [97] with OSCAR-based `NOR` [98],
- DRAM-based MIMDRAM [78], and
- SRAM-based Duality Cache [31].

We compare these to (1) *Baseline*, the original versions of the datapaths (which use the host CPU to execute non-PUM instructions); and (2) a modern 16384-core NVIDIA GeForce RTX 4090 *GPU* [75]. We also compare against a 16-core Intel Xeon Gold 6544Y CPU [46], but omit those results for brevity as the GPU always outperforms the CPU. All PUM architectures use iso-area comparisons for a 4 cm^2 chip (i.e., we reduce the number of MPUs to compensate for front-end hardware area). Table III summarizes system parameters.

TABLE III
System Parameters

MPU:RACER/MIMDRAM/Duality Cache (non-exhaustive)		
Pointer Table Entries	20	20-bit entry/opcode
Template Lookup Entries	1024	24-bit entry/micro-op template
Bits in Activation Board	512	1 bit per VRF in MPU
Playback Buffer Entries	1024	27 bits per stored instruction
Instruction Storage Cap.	2 MB	per MPU
Active VRFs Per RFH	1/256/256	due to thermal constraints
RFHs Per MPU	8	due to interconnect constraints
MPUs on Chip	497/450/12	each MPU manages 16 MB of memory
Compute Controllers	1	per MPU
Micro-Op Issue Rate	1	micro-op per cycle per MPU
Host CPU *(based on Intel Xeon Gold 6544Y [46])*		
# Cores	16	x86, OoO, 4 wide
L1 Inst./Data Caches	80 kB	each, 8-way set associative, per core
L2 Caches	2 MB	8-way set associative, per core
L3 (Last-Level) Cache	45 MB	16-way set associative, shared
Memory	8 GB	DDR3L, 64-bit bus

Applications. We start our evaluations using 21 data-intensive kernels from a wide range of application domains. We split these into four groups: (1) *basic* kernels that the RACER datapath can execute without CPU/MPU support, (2) *branch-focused* kernels with multiple nested branches, (3) *stencil* kernels that are challenging to express without a robust execution model, and (4) *complex* kernels with complex control instructions that the three PUM datapaths are unable to execute without the CPU/MPU. Then, we evaluate three complex end-to-end applications: (1) *LLMEncode*, a large language model (LLM) encoder [100]; (2) *BlackScholes*, a financial model used to price options [17, 18]; and (3) *EditDistance*, a bitap-based genome sequencing algorithm to calculate the distance of (i.e., difference between) two genome reads [86, 92, 93].

For the GPU, we work to maximize optimizations for each application, in order to present a fair and competitive comparison. All applications are written in CUDA, and make extensive use of kernel fusion and highly optimized libraries such as NVIDIA cuBLAS [74] to maximize GPU core utilization (which we verified using NVIDIA's profiling tools [76]). We make use of multiple kernel streams, overlapping compute and communication for independent streams as much as possible.

Modeling & Simulation. We extensively overhaul and modify RACER-Sim [11] to develop MASTODON (*Memory Array Simulation Testbed for Organization, Data, Operations, and Networks*) [12], a cycle-accurate simulation of the MPU that can faithfully emulate the RACER, MIMDRAM, and Duality Cache back ends. We validate the MIMDRAM and Duality Cache performance and energy statistics reported by MASTODON with data reported in the original papers [31, 78]. We synthesize critical components of the MPU using Synopsys Design Compiler [94] with the FreePDK 15 nm process [16]. Our synthesized circuitry achieves a frequency of 1 GHz, and we ensure our simulator's cycle accuracy for each MPU model by calibrating to critical paths identified by Synopsys timing tools. We integrate MASTODON with the Structural Simulation Toolkit (SST) [82, 85], leveraging existing cycle-accurate modules to faithfully model inter-MPU communication and on-chip network properties. We have open-sourced MASTODON (along with *ezpim*) under the MIT License [12]. GPU runs are performed using a real RTX 4090.

VIII. Evaluation

A. MPU Front End: Area & Power Analysis

From synthesis, we find that an MPU front end has a total area of 0.123 mm^2, with a static power of 1.22 mW and a dynamic power of 71.72 mW. Figure 11 breaks down the control path power and area. We observe that storage-based components (e.g., playback buffer, template lookup) are responsible for 53% of the total front-end area, 91% of its static power, and almost all of its dynamic power.

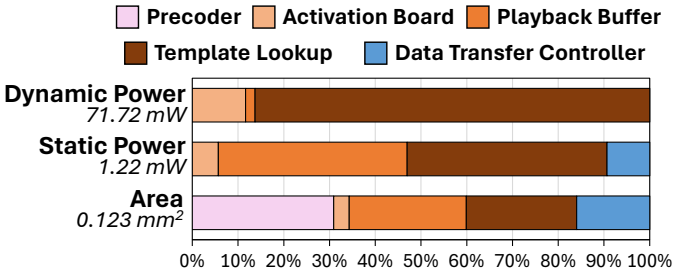


Fig. 11. Power and area breakdown for a single MPU front end.

As an example, if we augment RACER with 512 MPUs on chip, the total chip area increases from 4.00 cm^2 to 4.63 cm^2, while static power goes from 330 mW to 955 mW. The *MPU* control path consumes a maximum of 36.7 W at runtime, or 40.2% of RACER's total system power. Note that the iso-area evaluations below use fewer than 512 MPUs (see Table III).

B. MPU Improvement Analysis

Our primary goals in this experiment are to demonstrate that (1) the MPU's VRFs, RF holders, and support infrastructure are sufficiently general to integrate with multiple PUM datapaths; and (2) the MPU can deliver performance and energy improvements across all three of our evaluated back ends over their original implementations. Figure 12 shows the speedup and energy savings of the *MPU:RACER*, *MPU:MIMDRAM*, and *MPU:DualityCache* configurations. For each configuration, we normalize its performance and energy to its respective baseline (e.g., *MPU:RACER* is normalized to *Baseline:RACER*, *MPU:MIMDRAM* is normalized to *Baseline:MIMDRAM*). We make three observations from the figure.

First, all three MPU configurations achieve better performance than their respective baselines: the MPU front end speeds up RACER by 78.7%, MIMDRAM by 69.5%, and Duality Cache by 12.3%. We attribute these speedups to the fact that *Baseline* spends a considerable amount of time communicating with the host CPU, which the MPU eliminates. As an example, for the basic kernels, which lack much control flow and therefore have minimal CPU–PUM communication, the MPU incurs minor slowdowns (e.g., RACER's average slowdown is 3.1%, with no drop greater than 4.9%), predominantly due to the reduction in datapath capacity for iso-area comparisons.

In contrast, stencils and complex kernels have a significant amount of control flow. Stencils require multiple pieces of data to be retrieved based on a specific pattern. To reduce the associated control complexity, prior PUM datapaths convert stencil microkernels into matrix-multiplication-based microkernels with much simpler data fetching patterns (e.g., Toeplitz matrix transformation), but this inflates the application footprint by approximately 4×. For the complex kernels, *Baseline* datapaths are unable to execute the kernels on their own, and thus rely heavily on the CPU, while the MPU control path eliminates all of the datapath–CPU communication. As an example, on average across these two categories, *MPU:RACER* achieves 4.4× the performance of *Baseline*. In some cases (e.g., *ibert-sqrt, euclidean*), *Baseline*'s external transfer overhead is so significant that its overall performance is worse than *GPU*.

Second, all three MPU configurations achieve significant energy savings over their respective baselines: The MPU front end delivers energy savings of 3.23× for RACER (i.e., an energy reduction of 69.0%), 2.34× for MIMDRAM, and 4.07× for Duality Cache. The predominant sources of these savings are (1) the reduction in CPU energy, as we eliminate CPU–PUM communication for control operations; and (2) improvements to

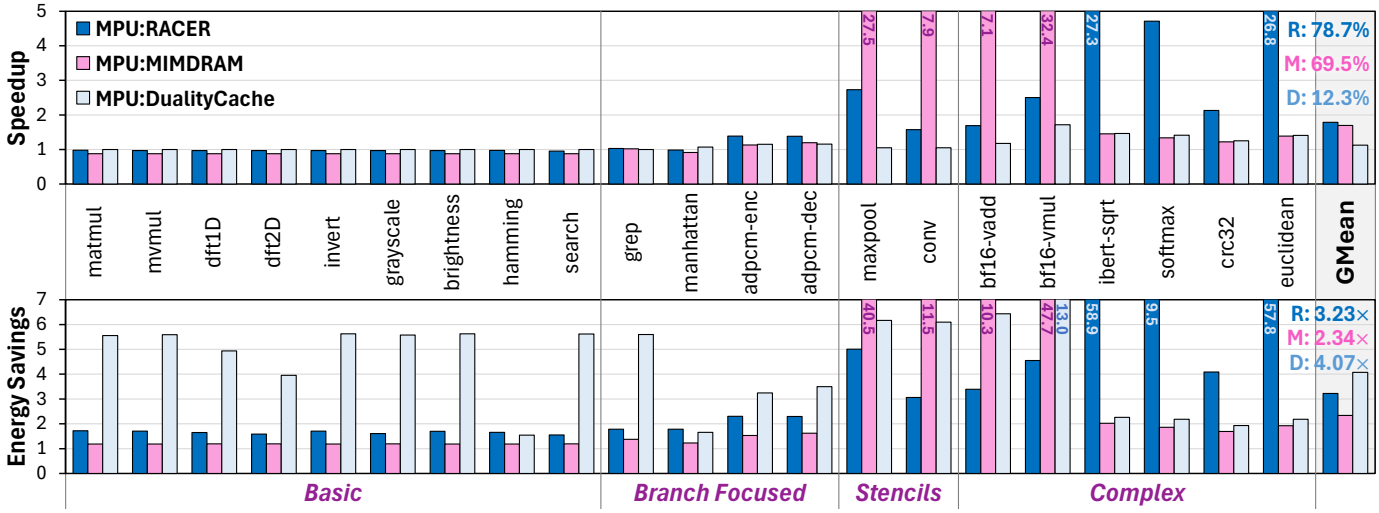


Fig. 12. Speedup (top) and energy savings (bottom) of *MPU:X* (*X* = *RACER*, *MIMDRAM*, *DualityCache*), each normalized to *Baseline:X*.

front-end and back-end processing (e.g., more efficient micro-op expansion, improved instruction reuse, built-in loop support). Even if we ignore CPU energy savings, the MPU's processing improvements reduce energy by 49.8%, 49.2%, and 22.6% for RACER, MIMDRAM, and Duality Cache, respectively.

Third, *MPU:DualityCache* has smaller improvements than *MPU:RACER* and *MPU:MIMDRAM*. This is for two reasons: (1) the Duality Cache arrays are on chip with the CPU, and thus incur lower communication costs; and (2) the limited on-chip capacity of Duality Cache (0.2 GB, due to the poor density of SRAM) forces it to spend significant time transferring data from the external memory. The MPU abstraction is most effective for applications that can fit entirely on the *MPU:DualityCache* chip (*manhattan*, *ibert-sqrt*, *softmax*, *crc32*, *euclidean*). For these kernels, *MPU:DualityCache* achieves a 31.1% speedup over *Baseline:DualityCache*, due to its dedicated single-cycle CMOS full adders that augment bitline-based computation, and its ability to activate all VRFs simultaneously (see Section IV). We note that MPU improvements are underestimated across the board, as *Baseline* does not include its (higher) costs of non-MPU binary retrieval, while *MPU* includes all such costs.

We conclude that our MPU design can effectively enable the *same* microkernels to achieve reasonable benefits across multiple datapaths with minimal code changes required.

C. Comparison With GPU

Next, we compare our MPU configurations to a modern high-performance GPU (the RTX 4090), to (1) provide context for how large the improvements in Section VIII-B are compared to executing the kernels on a conventional commercial processor; and (2) demonstrate that even with its efficient front end for parallel lockstep execution, the GPU cannot fully exploit the data-level parallelism available in these kernels. Figure 13 shows the speedup and energy savings of Baseline and MPU versions of RACER and MIMDRAM, normalized to *GPU*. We make three observations from the figure.

First, the MPU configurations outperform the GPU on average. *MPU:RACER* does so for all but one kernel, with 67× the performance of *GPU* across all 21 kernels.[2] *MPU:MIMDRAM* does so for all but three kernels, achieving a mean speedup of 156× vs. the GPU. While the underperforming kernels are part of our complex kernel group, we do observe that the MPU is able to improve performance over Baseline even for these, and even pulls *MPU:RACER* above *GPU* for *ibert-sqrt*.

Second, the kernels with the highest improvements differ between RACER and MIMDRAM. For our stencils, there exists significant bulk computation that MIMDRAM is better at exploiting, with its wider vector widths and lower bit-serial latency, hence the strong stencil improvements for *MPU:MIMDRAM*. In contrast, our complex applications benefit from more granular vectorization (e.g., loops can complete earlier for more ensembles as there is a lower chance of divergence), so *MPU:RACER* is able to achieve strong benefits.

Third, *MPU:RACER* and *MPU:MIMDRAM* achieve significant energy savings, with an average savings of 47× and 35×, respectively, over *GPU*. While much of the improvements over *Baseline* for the basic kernels is the result of eliminating idle CPU energy, the other kernels see large benefits due to the MPU's more efficient processing. For example, with complex kernels, *MPU:RACER* achieves a 6.2× and 11.3× improvement in energy over *GPU* and *Baseline*, respectively. This is because *Baseline* is bottlenecked by control instruction offloading between the CPU and the PUM datapath, which increases the overall execution time and, thus, energy consumption. We conclude that control-path hardware capable of performing complex control instructions is necessary for PUM architectures to successfully execute a broader range of applications beyond highly parallel basic blocks.

Note that while we do not show graphs due to space constraints, *MPU:DualityCache* achieves modest performance (1.6×) and energy (3.6×) improvements over *GPU*. The per-kernel benefits are mixed, with eight kernels demonstrating

[2]This conservatively assumes that RACER can have only one active VRF per RFH. If we increase this to two active VRFs, which is still within air-cooled thermal limits, *MPU:RACER* reaches speedups of 134× over *GPU*.

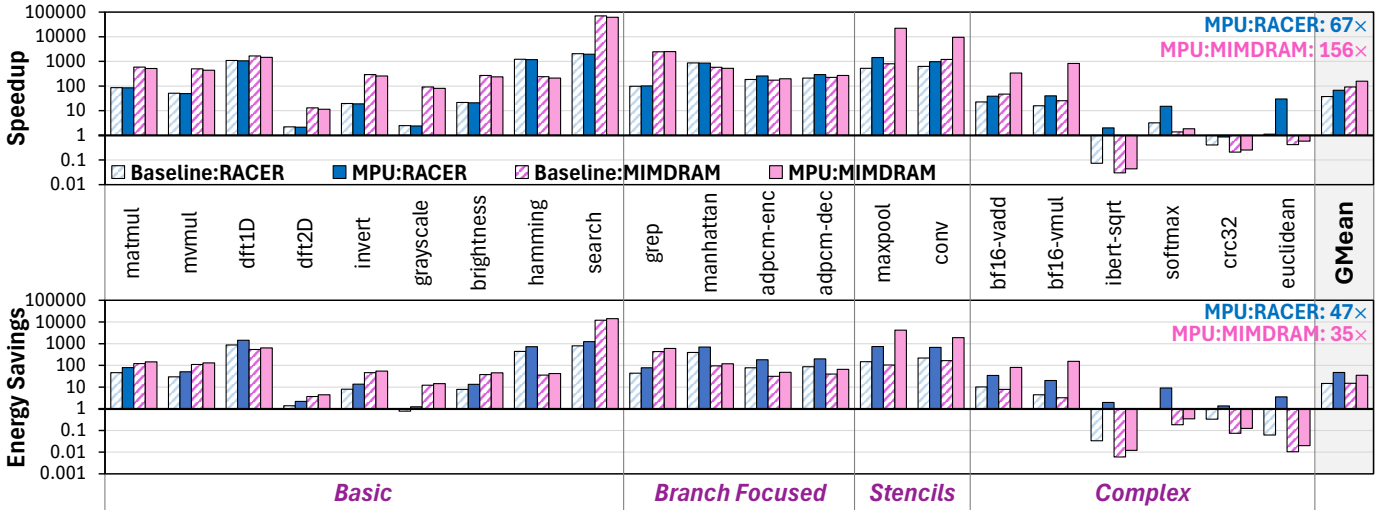


Fig. 13. Speedup (top) and energy savings (bottom) of *Baseline:X* and *MPU:X* (*X* = *RACER*, *MIMDRAM*), normalized to *GPU*; y-axis is in log scale.

sizeable improvements, while six show large slowdowns. This is in large part due to the limited memory capacity of Duality Cache and its high operation latency (14 cycles), both of which hamper its abilities (and neither of which are due to the MPU).

We conclude that MPU-based PUM architectures offer significant performance and energy gains over state-of-the-art GPUs, overcoming several drawbacks of conventional PUM.

D. End-to-End Application Analysis

Finally, we show that thanks to the efficiencies of the MPU, it is now feasible to execute entire applications end-to-end within a PUM datapath. As both *MPU:RACER* and *MPU:MIMDRAM* exhibit strong performance for both basic and complex microkernels, we investigate these MPU configurations further by running three complex end-to-end applications. Each application has multiple compute steps and multiple forms of complex collective communication, as shown in Table IV. The table also shows how *ezpim* significantly reduces complex application code size.

TABLE IV
END-TO-END APPLICATION EXECUTION STEPS ON THE MPU

Application	Compute Step	Collective Commun.	MPUs	Lines of Code Baseline	Lines of Code *ezpim*
LLMEncode	matmul, softmax layernorm, relu	gather, scatter P2P, broadcast	130	15290	1160
BlackScholes	sqrt, exp, norm CDF	P2P, broadcast	2	1059	383
EditDistance	bitwise comparisons	2-D systolic	23	5428	120

Figure 14 shows the speedup and energy savings of four configurations (*Baseline* and *MPU* for both RACER and MIMDRAM), vs. *GPU*. We make two observations from the figure.

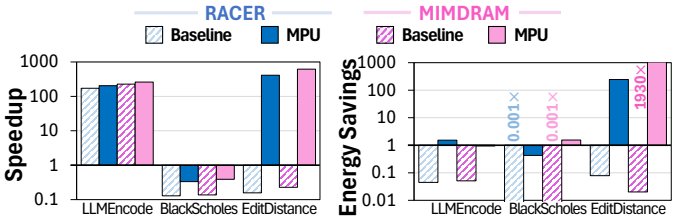


Fig. 14. End-to-end application speedup and energy savings vs. *GPU*.

First, *Baseline*'s speedup is highly dependent on the size of the basic block. For LLMEncode, which contains many large, regular, highly parallel computing steps, *Baseline* achieves large speedups over *GPU*. However, its performance decreases proportionally as the fraction of execution time spent on communication with the CPU increases, as shown in Figure 15. The figure breaks down execution time into three components: (1) MPU computation, (2) inter-MPU communication on-chip, and (3) off-chip communication with an external CPU. While the *MPU* configuration's execution time only consists of MPU computation and inter-MPU communication, the *Baseline* configuration must spend additional time transferring data between the CPU and the PUM datapath. For *EditDistance*, in which almost all execution time is attributed to off-chip communication, the cost of frequent CPU–PUM communication

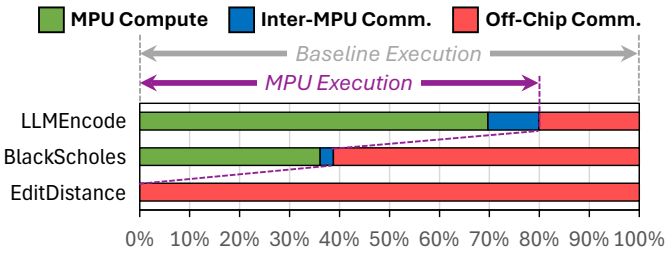


Fig. 15. Execution time breakdown for *MPU* vs. *Baseline*.

can make *Baseline* perform 7.72× worse than *GPU*. In contrast, because there is no off-chip communication for *MPU* configurations, *MPU:RACER*/*MPU:MIMDRAM* achieve speedups over *GPU* of 198×/229× and 400×/545× for *LLMEncode* and *EditDistance*, respectively. The MPU configurations still experience slowdowns with *BlackScholes* due to their extensive use of CORDIC subroutines (implemented as software-emulated subroutines), for which the GPU has significantly faster dedicated hardware. Even then, the MPU improves over *Baseline* by 2.50× for *BlackScholes*.

Second, *Baseline* can significantly undermine the energy savings of PUM platforms, and in the case of *BlackScholes* and *EditDistance* actually consumes *more* energy than *GPU*, for both RACER and MIMDRAM datapaths. For *BlackScholes*, the frequent CPU–PUM communication is attributed to its complex computing steps, such as calculating the square root, whereas *EditDistance* needs frequent communication due to its complex 2D systolic patterns, which require frequent synchronization between back-end arrays. Such frequent CPU–PUM communication severely increases *Baseline*'s execution time, and coupled with the fact that the external CPU generates additional power for the system, results in orders of magnitude higher energy consumption compared to the *GPU* and *MPU* configurations. In contrast, *MPU* with RACER/MIMDRAM achieves average energy savings of 5.4×/14.2× over *GPU*.

From these observations, we conclude that the MPU ISA and its front-end hardware are effective solutions to enable end-to-end application execution on different PUM platforms.

IX. MPU LIMITATIONS

Porting to Other Back Ends. We note that while the MPU should work for most bitwise PUM architectures, it currently does not adapt well to non-bitwise PUM approaches. For example, works such as Liquid Silicon [103] treat the memory as an FPGA, and allocate large connected strings of hardware blocks in an ASIC-like manner. Such custom hardware pipelines require custom front ends (as is the case for FPGA datapaths today), which are difficult for an MPU-like model to generalize efficiently. However, one could instantiate a datapath on top of Liquid Silicon that is compatible with the MPU ISA, facilitating MPU integration.

Beyond PUM, the MPU abstraction and front end are compatible with any processing-in-memory (PIM) architecture that presents a vector abstraction (e.g., CAPE [21]). Some PIM architectures take different approaches to expose in-memory operations to programmers: for example, GDDR6-AiM [43,

65] implements a custom front end that can support only matrix multiplication, while UPMEM [26] presents a custom API for its data processing units (DPUs). While it may not be efficient to replace such front ends with the MPU, it could be possible to extend the MPU with a meta-ISA that encompasses all of these operations (e.g., encode matrix multiply operations as multiply and accumulate micro-ops), and uses a lightweight hardware/software coordination layer to enable concurrent execution across heterogeneous types of PIM.

Completing Application Support. While the MPU can support the end-to-end execution of applications, it still lacks a number of important features that programmers expect in modern architectures: precise exception handling, function calls, and a true compiler toolchain. We believe that the MPU has made each of these significantly more feasible, and expect to implement all of these in our future work.

X. Related Works

To our knowledge, the MPU is the first PUM interface to (1) abstract away microarchitectural details of PUM datapaths with a common interface, (2) propose control logic that is compatible with most bitwise PUM datapaths, and (3) enable end-to-end program execution using in-PUM control flow. We briefly discuss closely related works below.

PUM With Warp-Centric Execution Model. Duality Cache [31] (DC) proposes an in-cache PUM microarchitecture based on a warp-centric execution model. While the work shows that it is possible to achieve significant speedup and energy savings compared to CPU/GPU platforms, while being able to program the platform using CUDA, there are three key challenges in adopting this execution model to other PUM works. First, DC can adopt the warp-centric model efficiently because of its relatively smaller capacity compared to platforms such as MIMDRAM and RACER. This allows DC to manage the activations of its SRAM subarray for in-memory instructions more effectively. Its smaller capacity effectively eliminates the need for at-scale system-level features such as synchronization and collective communication. Second, DC does not provide any high-level abstractions (e.g., VRFs, RF holders) that allow its execution model to integrate with other memory technology and microarchitectural organizations. Third, DC's front end does not support complex control behavior, making it difficult to run end-to-end applications.

System-Level Artifacts for PUM. Previous works propose non-recipe-based instruction-to-micro-op (I2M) translation routines [29, 96], with a compilation flow that identifies and translates PUM-friendly code regions from high-level representations [4, 27, 53, 90]. Many of these system-level artifacts can work on top of the MPU ISA. For example, abstractPIM [29] discusses translating a generic arithmetic instruction into technology-specific micro-ops. The MPU can make use of this translation flow to generate the micro-op sequences of instructions that appear in a compute ensemble and send them to the appropriate I2M decoders at the right time. As another example, PIMFlow [90] describes how to compile neural network graphs into processing-in-DRAM instructions. PIMFlow can use the MPU ISA as an intermediate representation to target other PUM memory technologies.

Digital PUM Datapaths. Prior works [28, 31, 40, 56, 64, 72, 78, 87, 95, 97, 98] have proposed various microarchitectural designs to perform arithmetic operations *in situ* across different memory technologies. While most offer a list of arithmetic instructions and some basic control instructions that programmers can use, none offer a robust ISA design that programmers can use to efficiently write scalable end-to-end applications. Furthermore, they do not consider how the ISA design can be applied to different PUM microarchitectures, which is essential for the development of a PUM software stack whose lifetime is not dependent on the success of any single microarchitectural design. The MPU ISA offers a stable hardware foundation through hardware abstractions that extend across many back-end implementations. It further offers a first look at what the control-path hardware should look like to efficiently support the MPU ISA and its complex control instructions.

Analog PUM Datapaths. Prior works [2, 5, 10, 24, 45, 69, 89] use resistive memory crossbars to perform in-memory analog matrix–vector multiplies (MVMs). The limited capability of these crossbars has restricted their use to matrix–multiply accelerators, primarily for machine learning. As these accelerators are designed and optimized for specific application domains with simple and highly regular operations, they do not require a complex ISA and execution model. Hand-written libraries and function calls are sufficient to leverage their abilities.

XI. Conclusion

Bitwise PUM platforms promise orders of magnitude speedup and energy savings compared to traditional computing platforms when executing data-intensive kernels. However, without a capable front end on-chip and a robust ISA, it has been challenging for these platforms to deliver their promised performance and efficiency for whole applications. We introduce the Memory Processing Unit (MPU), a microarchitecture-agnostic PUM interface that eliminates the need for an external CPU to handle complex control behaviors on behalf of the PUM platforms. Our evaluations show that when the MPU is used with the RACER, MIMDRAM, and Duality Cache datapaths, it improve the programmability, performance, and energy of control-heavy kernels and end-to-end applications. The MPU enables programmers to write microarchitecture-agnostic, end-to-end programs, establishing a foundation for a PUM software stack and making it possible for existing computing domains to adopt PUM in the future.

Acknowledgments

This work was supported by the Univ. of Illinois Center for Advanced Semiconductor Chips with Accelerated Performance (ASAP; an NSF IUCRC), a grant from the Samsung Memory Solutions Lab, the Data Storage Systems Center at Carnegie Mellon Univ., the Univ. of Illinois DREMES HYBRID Center, and NSF grant CCF-2329096. Minh Truong was supported by an Apple Ph.D. Fellowship in Integrated Systems.

REFERENCES

[1] Advanced Micro Devices, Inc., "Samsung SmartSSD," https://www.xilinx.com/applications/data-center/computational-storage/smartssd.html.

[2] S. Aga, S. Jeloka, A. Subramaniyan, S. Narayanasamy, D. Blaauw, and R. Das, "Compute Caches," in *HPCA*, 2017.

[3] V. Agrawal, T. P. Xiao, C. H. Bennett, B. Feinberg, S. Shetty, K. Ramkumar, H. Medu, K. Thekkekara, R. Chettuvetty, S. Leshner, Z. Luzada, L. Hinh, T. Phan, M. J. Marinella, and S. Agarwal, "Subthreshold Operation of SONOS Analog Memory to Enable Accurate Low-Power Neural Network Inference," in *IEDM*, 2022.

[4] H. Ahmed, P. C. Santos, J. P. C. Lima, R. F. Moura, M. A. Z. Alves, and A. C. S. Beck, "A Compiler for Automatic Selection of Suitable Processing-in-Memory Instructions," in *DATE*, 2019.

[5] J. Ahn, S. Yoo, O. Mutlu, and K. Choi, "PIM-Enabled Instructions: A Low-Overhead, Locality-Aware Processing-in-Memory Architecture," in *ISCA*, 2015.

[6] S. Angizi, Z. He, A. Awad, and D. Fan, "MRIMA: An MRAM-Based In-Memory Accelerator," *TCAD*, May 2020.

[7] S. Angizi, Z. He, and D. Fan, "PIMA-Logic: A Novel Processing-in-Memory Architecture for Highly Flexible and Energy-Efficient Logic Computation," in *DAC*, 2018.

[8] S. Angizi, Z. He, A. S. Rakin, and D. Fan, "CMP-PIM: An Energy-Efficient Comparator-based Processing-in-Memory Neural Network Accelerator," in *DAC*, 2018.

[9] S. Angizi, J. Sun, W. Zhang, and D. Fan, "AlignS: A Processing-in-Memory Accelerator for DNA Short Read Alignment Leveraging SOT-MRAM," in *DAC*, 2019.

[10] A. Ankit, I. El Hajj, S. R. Chalamalasetti, G. Ndu, M. Foltin, R. S. Williams, P. Faraboschi, W. m. Hwu, J. P. Strachan, K. Roy, and D. S. Milojicic, "PUMA: A Programmable Ultra-Efficient Memristor-Based Accelerator for Machine Learning Inference," in *ASPLOS*, 2019.

[11] ARCANA Research Group, "RACER Artifacts — GitHub Repository," https://github.com/ARCANA-Research/RACER-Artifacts/, 2021.

[12] ARCANA Research Group, "MASTODON — GitHub Repository," https://github.com/ARCANA-Research/MASTODON/, 2026.

[13] G. H. Barnes, R. M. Brown, M. Kato, D. J. Kuck, D. L. Slotnick, and R. A. Stokes, "The ILLIAC IV Computer," *TC*, Aug. 1968.

[14] K. E. Batcher, "Bit-Serial Parallel Processing Systems," *TC*, 1982.

[15] C. H. Bennett, T. P. Xiao, R. Dellana, B. Feinberg, S. Agarwal, M. J. Marinella, V. Agrawal, V. Prabhakar, K. Ramkumar, L. Hinh, S. Saha, V. Raghavan, and R. Chettuvetty, "Device-Aware Inference Operations in SONOS Non-Volatile Memory Arrays," in *IRPS*, 2020.

[16] K. Bhanushali and W. R. Davis, "FreePDK15: An Open-Source Predictive Process Design Kit for 15nm FinFET Technology," in *ISPD*, 2015.

[17] C. Bienia, S. Kumar, J. P. Singh, and K. Li, "The PARSEC Benchmark Suite: Characterization and Architectural Implications," in *PACT*, 2008.

[18] F. Black and M. Scholes, "The Pricing of Options and Corporate Liabilities," *The Journal of Political Economy*, May 1973.

[19] J. Borghetti, G. S. Snider, P. J. Kuekes, J. J. Yang, D. R. Stewart, and R. S. Williams, "Memristive Switches Enable Stateful Logic Operations via Material Implication," *Nature*, Apr. 2010.

[20] A. Boroumand, S. Ghose, Y. Kim, R. Ausavarungnirun, E. Shiu, R. Thakur, D. Kim, A. Kuusela, A. Knies, P. Raganathan, and O. Mutlu, "Google Workloads for Consumer Devices: Mitigating Data Movement Bottlenecks," in *ASPLOS*, 2018.

[21] 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 *HPCA*, 2021.

[22] M. Cassinerio, N. Ciocchini, and D. Ielmini, "Logic Computation in Phase Change Materials by Threshold and Memory Switching," *Adv. Materials*, Nov. 2013.

[23] S. Chen, Y. Jiang, C. Delimitrou, and J. F. Martínez, "PIMCloud: QoS-Aware Resource Management of Latency-Critical Applications in Clouds With Processing-in-Memory," in *HPCA*, 2022.

[24] T. Chou, W. Tang, J. Botimer, and Z. Zhang, "CASCADE: Connecting RRAMs to Extend Analog Dataflow in an End-to-End In-Memory Processing Paradigm," in *MICRO*, 2019.

[25] W. J. Dally, "Challenges for Future Computing Systems," keynote talk at HiPEAC, 2015.

[26] F. Devaux, "The True Processing in Memory Accelerator," in *HotChips*, 2019.

[27] 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 *ISCA*, 2022.

[28] C. Eckert, X. Wang, J. Wang, A. Subramaniyan, R. Iyer, D. M. Sylvester, D. T. Blaauw, and R. Das, "Neural Cache: Bit-Serial In-Cache Acceleration of Deep Neural Networks," in *ISCA*, 2018.

[29] A. Eliahu, R. Ben-Hur, R. Ronen, and S. Kvatinsky, "abstractPIM: Bridging the Gap Between Processing-In-Memory Technology and Instruction Set Architecture," in *VLSI-SOC*, 2020.

[30] M. J. Flynn, "Very High-Speed Computing Systems," *Proc. IEEE*, Dec. 1966.

[31] D. Fujiki, S. Mahlke, and R. Das, "Duality Cache for Data Parallel Acceleration," in *ISCA*, 2019.

[32] P. Gaillardon, L. Amaru, A. Siemon, E. Linn, R. Waser, A. Chattopadhyay, and G. D. Micheli, "The Programmable Logic-in-Memory (PLiM) Computer," in *DATE*, 2016.

[33] C. Gao, X. Xin, Y. Lu, Y. Zhang, J. Yang, and J. Shu, "ParaBit: Processing Parallel Bitwise Operations in NAND Flash Memory Based SSDs," in *MICRO*, 2021.

[34] F. Gao, G. Tziantzioulis, and D. Wentzlaff, "ComputeDRAM: In-Memory Compute Using Off-the-Shelf DRAMs," in *MICRO*, 2019.

[35] F. Gao, G. Tziantzioulis, and D. Wentzlaff, "FracDRAM: Fractional Values in Off-the-Shelf DRAM," in *MICRO*, 2022.

[36] S. Ghose, A. Boroumand, J. S. Kim, J. Gómez-Luna, and O. Mutlu, "Processing-in-Memory: A Workload-Driven Perspective," *IBM JRD*, Nov.–Dec. 2019.

[37] GSI Technology, Inc., "In-Place Associative Computing," https://www.gsitechnology.com/APU.

[38] GSI Technology, Inc., "Gemini® APU: Enabling High Performance Billion-Scale Similarity Search," White Paper, 2020, https://www.gsitechnology.com/sites/default/files/Whitepapers/GSIT-APU-Enabling-High-Performance-Billion-Scale-Similarity-Search-WP.pdf.

[39] S. Gupta, M. Imani, and T. Rosing, "FELIX: Fast and Energy-Efficient Logic in Memory," in *ICCAD*, 2018.

[40] N. Hajinazar, G. F. Oliveira, S. Gregorio, J. D. Ferreira, N. M. Ghiasi, M. Patel, M. Alser, S. Ghose, J. Gomez-Luna, and O. Mutlu, "SIMDRAM: A Framework for Bit-Serial SIMD Processing Using DRAM," in *ASPLOS*, 2021.

[41] S. Hamdioui, S. Kvatinsky, G. Cauwenberghs, L. Xie, N. Wald, S. Joshi, H. M. Elsayed, H. Corporaal, and K. Bertels, "Memristor for Computing: Myth or Reality?" in *DATE*, 2017.

[42] S. Hamdioui, L. Xie, H. A. D. Nguyen, M. Taouil, K. Bertels, H. Corporaal, H. Jiao, F. Catthoor, D. Wouters, L. Eike, and J. van Lunteren, "Memristor-Based Computation-in-Memory Architecture for Data-Intensive Applications," in *DATE*, 2015.

[43] 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 *MICRO*, 2020.

[44] W. Huang, M. R. Stan, S. Gurumurthi, R. J. Ribando, and K. Skadron, "Interaction of Scaling Trends in Processor Architecture and Cooling," in *SEMI-THERM*, 2010.

[45] M. Imani, S. Gupta, Y. Kim, and T. Rosing, "FloatPIM: In-Memory Acceleration of Deep Neural Network Training With High Precision," in *ISCA*, 2019.

[46] Intel Corp., "Intel Xeon Gold 6544Y," https://www.intel.com/content/www/us/en/products/sku/237569/intel-xeon-gold-6544y-processor-45m-cache-3-60-ghz/specifications.html.

[47] S. Jain, A. Ranjan, K. Roy, and A. Raghunathan, "Computing in Memory With Spin-Transfer Torque Magnetic RAM," *TVLSI*, Mar. 2018.

[48] S. Jeloka, N. B. Akesh, D. Sylvester, and D. Blaauw, "A 28 nm Configurable Memory (TCAM/BCAM/SRAM) Using Push-Rule 6T Bit Cell Enabling Logic-in-Memory," in *JSSC*, 2016.

[49] M. Kang, E. P. Kim, M.-S. Keel, and N. R. Shanbhag, "Energy-Efficient and High Throughput Sparse Distributed Memory Architecture," in *ISCAS*, 2015.

[50] L. Ke, X. Zhang, J. So, J.-G. Lee, S.-H. Kang, S. Lee, S. Han, Y. Cho, J. H. Kim, Y. Kwon, K. Kim, J. Jung, I. Yun, S. J. Park, H. Park, J. Song, J. Cho, K. Sohn, N. S. Kim, and H.-H. S. Lee, "Near-Memory Processing in Action: Accelerating Personalized Recommendation With AxDIMM," *IEEE Micro*, 2021.

[51] G. Kestor, R. Gioiosa, D. J. Kerbyson, and A. Hoisie, "Quantifying the Cost of Data Movement in Scientific Applications," in *IISWC*, 2013.

[52] R. Khaddam-Aljameh, M. Stanisavljevic, J. F. Mas, G. Karunaratne, M. Braendli, F. Liu, A. Singh, S. M. Müller, U. Egger, A. Petropoulos, T. Antonakopoulos, K. Brew, S. Choi, I. Ok, F. L. Lie, N. Saulnier, V. Chan, I. Ahsan, V. Narayanan, S. R. Nandakumar, M. L. Gallo, P. A. Francese, A. Sebastian, and E. Eleftheriou, "HERMES Core – A 14nm CMOS and PCM-Based In-Memory Compute Core Using an Array of 300ps/LSB Linearized CCO-Based ADCs and Local Digital Processing," in *VLSIT*, 2021.

[53] 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 *ASPLOS*, 2024.

[54] 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:2401.14428 [cs.AR], 2024.

[55] Y. S. Ki, "Innovation With SmartSSD for Green Computing," talk at SNIA PMCS, 2022.

[56] S. Kvatinsky, "Real Processing-in-Memory With Memristive Memory Processing Unit (mMPU)," in *ASAP*, 2019.

[57] S. Kvatinsky, D. Belousov, S. Liman, G. Satat, N. Wald, E. G. Friedman, A. Kolodny, and U. C. Weiser, "MAGIC: Memristor-Aided Logic," *TCAS II*, Sep. 2014.

[58] S. Kvatinsky, A. Kolodny, U. C. Weiser, and E. G. Friedman, "Memristor-Based IMPLY Logic Design Procedure," in *ICCD*, 2011.

[59] S. Kvatinsky, G. Satat, N. Wald, E. G. Friedman, A. Kolodny, and U. C. Weiser, "Memristor-Based Material Implication (IMPLY) Logic: Design Principles and Methodologies," in *VLSI*, 2013.

[60] Y. C. Kwon, S. H. Lee, J. Lee, S. H. Kwon, J. M. Ryu, J. P. Son, S. O, H. S. Yu, H. Lee, S. Y. Kim, Y. Cho, J. G. Kim, J. Choi, H. S. Shin, J. Kim, B. S. Phuah, H. M. Kim, M. J. Song, A. Choi, D. K. Y. Kim, E. B. Kim, D. Wang, S. Kang, Y. Ro, S. Seo, J. H. Song, J. Youn, K. Sohn, and N. S. Kim, "A 20nm 6GB Function-In-Memory DRAM, Based on HBM2 With a 1.2TFLOPS Programmable Computing Unit Using Bank-Level Parallelism, for Machine Learning Applications," in *ISSCC*, 2021.

[61] L. Lamport, "How to Make a Correct Multiprocess Program Execute Correctly on a Multiprocessor," *TC*, September 1979.

[62] M. Le Gallo, R. Khaddam-Aljameh, M. Stanisavljevic, A. Vasilopoulos, B. Kersting, M. Dazzi, G. Karunaratne, M. Brändli, A. Singh, S. M. Müller, J. Büchel, X. Timoneda, V. Joshi, M. J. Rasch, U. Egger, A. Garofalo, A. Petropoulos, T. Antonakopoulos, K. Brew, S. Choi, I. Ok, T. Philip, V. Chan, C. Silvestre, I. Ahsan, N. Saulnier, V. Narayanan, P. A. Francese, E. Eleftheriou, and A. Sebastian, "A 64-Core Mixed-Signal In-Memory Compute Chip Based on Phase-Change Memory for Deep Neural Network Inference," *Nature Electronics*, Aug. 2023.

[63] D. Lee, J. So, M. Ahn, J.-G. Lee, J. Kim, J. Cho, R. Oliver, V. C. Thummala, R. S. JV, S. S. Upadhya, M. I. Khan, and J. H. Kim, "Improving In-Memory Database Operations With Acceleration DIMM (AxDIMM)," in *DaMoN*, 2022.

[64] S. Lee, S. Lee, M. Seo, C. Park, W. Shin, and H. Lee, "NPC: A Non-Conflicting Processing-in-Memory Controller in DDR Memory Systems," in *IEEE Access*, 2024.

[65] S. Lee, K. Kim, S. Oh, J. Park, G. Hong, D. Ka, K. Hwang, J. Park, K. Kang, J. Kim, J. Jeon, N. Kim, Y. Kwon, K. Vladimir, W. Shin, J. Won, M. Lee, H. Joo, H. Choi, J. Lee, D. Ko, Y. Jun, K. Cho, I. Kim, C. Song, C. Jeong, D. Kwon, J. Jang, I. Park, J. Chun, and J. Cho, "A 1ynm 1.25V 8Gb, 16Gb/s/Pin GDDR6-Based Accelerator-in-Memory Supporting 1TFLOPS MAC Operation and Various Activation Functions for Deep-Learning Applications," in *ISSCC*, 2022.

[66] S. Lee, S. haeng Kang, J. Lee, H. Kim, E. Lee, S. Seo, H. Yoon, S. Lee, K. Lim, H. Shin, J. Kim, S. O, A. Iyer, D. Wang, K. Sohn, and N. S. Kim, "Hardware Architecture and Software Stack for PIM Based on Commercial DRAM Technology," in *ISCA*, 2021.

[67] Y. Levy, J. Briuck, Y. Cassuto, E. G. Friedman, A. Kolodny, E. Yaakobi, and S. Kvatinsky, "Logic Operations in Memory Using a Memristive Akers Array," *Microelectronics*, Nov. 2014.

[68] B. Li, L. Xia, P. Gu, Y. Wang, and H. Yang, "Merging the Interface: Power, Area, and Accuracy Co-Optimization for RRAM Crossbar-Based Mixed-Signal Computing System," in *DAC*, 2015.

[69] S. Li, D. Niu, K. T. Malladi, H. Zheng, B. Brennan, and Y. Xie, "DRISA: A DRAM-Based Reconfigurable In-Situ Accelerator," in *MICRO*, 2017.

[70] S. Li, C. Xu, Q. Zou, J. Zhao, Y. Lu, and Y. Xie, "Pinatubo: A Processing-in-Memory Architecture for Bulk Bitwise Operations in Emerging Non-Volatile Memories," in *DAC*, 2016.

[71] J. Louis, B. Hoffer, and S. Kvatinsky, "Performing Memristor-Aided Logic (MAGIC) Using STT-MRAM," in *ICECS*, 2019.

[72] A. Mamdouh, H. Geng, M. Niemier, X. S. Hu, and D. Reis, "Shared-PIM: Enabling Concurrent Computation and Data Flow for Faster Processing-in-DRAM," arXiv:2408.15489 [cs.AR], 2024.

[73] O. Mutlu, S. Ghose, J. Gómez-Luna, and R. Ausavarungnirun, "Enabling Practical Processing in and Near Memory for Data-Intensive Computing," in *DAC*, 2019.

[74] NVIDIA Corp., "cuBLAS: Basic Linear Algebra on NVIDIA GPUs," https://developer.nvidia.com/cublas/.

[75] NVIDIA Corp., "GeForce RTX 4090," https://www.nvidia.com/en-us/geforce/graphics-cards/40-series/rtx-4090/.

[76] NVIDIA Corp., *CUDA Toolkit Documentation: Profiler User's Guide*, 2025, https://docs.nvidia.com/cuda/profiler-users-guide/.

[77] G. F. Oliveira, L. O. J. Gomez-Luna, S. Ghose, N. Vijaykumar, I. Fernandez, M. Sadrosadati, and O. Mutlu, "DAMOV: A New Methodology and Benchmark Suite for Evaluating Data Movement Bottlenecks," *IEEE Access*, September 2021.

[78] 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 *HPCA*, 2024.

[79] S. Ollivier, S. Longofono, P. Dutta, J. Hu, S. Bhanja, and A. K. Jones, "CORUSCANT: Fast Efficient Processing-in-Racetrack Memories," in *MICRO*, 2022.

[80] J. Park, R. Azizi, G. F. Oliveira, M. Sadrosadati, R. Nadig, D. Novo, J. Gómez-Luna, M. Kim, and O. Mutlu, "Flash-Cosmos: In-Flash Bulk Bitwise Operations Using Inherent Computation Capability of NAND Flash Memory," in *MICRO*, 2022.

[81] M. Prezioso, F. Merrikh-Bayat, B. Hoskins, G. C. Adam, K. K. Likharev, and D. B. Strukov, "Training and Operation of an Integrated Neuromorphic Network Based on Metal-Oxide Memristors," *Nature*, May 2015.

[82] A. F. Rodrigues, K. S. Hemmert, B. W. Barrett, C. Kersey, R. Oldfield, M. Weston, R. Risen, J. Cook, P. Rosenfeld, E. Cooper-Balis, and B. Jacobs, "The Structural Simulation Toolkit," in *SIGMETRICS*, 2011.

[83] Samsung Electronics Co., Ltd., "HBM Processing in Memory," https://www.samsung.com/semiconductor/solutions/technology/hbm-processing-in-memory/.

[84] Samsung Electronics Co., Ltd., "Samsung Electronics Develops Second-Generation SmartSSD Computational Storage Drive With Upgraded Processing Functionality," https://news.samsung.com/global/samsung-electronics-develops-second-generation-smartssd-computational-storage-drive-with-upgraded-processing-functionality, July 2022.

[85] Sandia National Laboratories, "The Structural Simulation Toolkit," https://sst-simulator.org/.

[86] D. Senol Cali, G. S. Kalsi, Z. Bingöl, C. Firtina, L. Subramanian, J. S. Kim, R. Ausavarungnirun, M. Alser, J. Gómez-Luna, Juan, A. Boroumand, A. Nori, A. Scibisz, S. Subramoney, C. Alkan, S. Ghose, and O. Mutlu, "GenASM: A High-Performance, Low-Power Approximate String Matching Acceleration Framework for Genome Sequence Analysis," in *MICRO*, 2020.

[87] 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 *MICRO*, 2017.

[88] V. Seshadri and O. Mutlu, "Simple Operations in Memory to Reduce Data Movement," in *Advances in Computers*, 2017, vol. 106.

[89] A. Shafiee, A. Nag, M. N, 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 *ISCA*, 2016.

[90] Y. Shin, J. Park, S. Cho, and H. Sung, "PIMFlow: Compiler and Runtime Support for CNN Models on Processing-in-Memory DRAM," in *CGO*, 2023.

[91] L. Song, X. Qian, H. Li, and Y. Chen, "PipeLayer: A Pipelined ReRAM-Based Accelerator for Deep Learning," in *HPCA*, 2017.

[92] M. Šošić and M. Šikić, "Edlib: A C/C++ Library for Fast, Exact Sequence Alignment Using Edit Distance," *Bioinformatics*, May 2017.

[93] M. Šošić and M. Šikić, "Edlib," 2024. [Online]. Available: https://github.com/Martinsos/edlib/tree/master

[94] Synopsys, Inc., *Synopsys Design Compiler*, 2024. [Online]. Available: https://www.synopsys.com/implementation-and-signoff/rtl-synthesis-test/dc-ultra.html

[95] N. Talati, R. Ben-Hur, N. Wald, A. Haj-Ali, J. Reuben, and S. Kvatinsky, "mMPU—A Real Processing-in-Memory Architecture to Combat the von Neumann Bottleneck," in *Applications of Emerging Memory Technology: Beyond Storage*, 2019.

[96] C. Tang, C. Nie, W. Qian, and Z. He, "PIMLC: Logic Compiler for Bit-Serial Based PIM," in *DATE*, 2024.

[97] M. S. Q. Truong, E. Chen, D. Su, A. Glass, L. Shen, L. R. Carley, J. A. Bain, and S. Ghose, "RACER: Bit-Pipelined Processing Using Resistive Memory," in *MICRO*, 2021.

[98] M. S. Q. Truong, L. Shen, A. Glass, A. Hoffmann, L. R. Carley, J. A. Bain, and S. Ghose, "Adapting the RACER Architecture to Integrate Improved In-ReRAM Logic Primitives," *JETCAS*, 2022.

[99] UPMEM SAS, "Technology," https://www.upmem.com/technology/.

[100] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin, "Attention Is All You Need," in *NIPS*, 2017.

[101] L. Xie, H. A. D. Nguyen, M. Taouil, S. Hamdioui, and K. Bertels, "Fast Boolean Logic Mapped on Memristor Crossbar," in *ICCD*, 2015.

[102] J. Yu, H. A. D. Nguyen, L. Xie, M. Taouil, and S. Hamdioui, "Memristive Devices for Computation-in-Memory," in *DATE*, 2018.

[103] Y. Zha and J. Li, "Liquid Silicon: A Data-Centric Reconfigurable Architecture Enabled by RRAM Technology," in *FPGA*, 2018.

[104] Y. Zha and J. Li, "Liquid Silicon-Monona: A Reconfigurable Memory-Oriented Computing Fabric With Scalable Multi-Context Support," in *ASPLOS*, 2018.