# RACER: Bit-Pipelined Processing Using Resistive Memory

Minh S. Q. Truong
Carnegie Mellon University

Eric Chen
Carnegie Mellon University

Deanyone Su
Carnegie Mellon University

Alexander Glass
Carnegie Mellon University

Liting Shen
Carnegie Mellon University

L. Richard Carley
Carnegie Mellon University

James A. Bain
Carnegie Mellon University

Saugata Ghose
University of Illinois Urbana-Champaign

## ABSTRACT

To combat the high energy costs of moving data between main memory and the CPU, recent works have proposed to perform *processing-using-memory* (PUM), a type of processing-in-memory where operations are performed on data *in situ* (i.e., right at the memory cells holding the data). Several common and emerging memory technologies offer the ability to perform bitwise Boolean primitive functions by having interconnected cells interact with each other, eliminating the need to use discrete CMOS compute units for several common operations. Recent PUM architectures extend upon these Boolean primitives to perform bit-serial computation using memory. Unfortunately, several practical limitations of the underlying memory devices restrict how large emerging memory arrays can be, which hinders the ability of conventional bit-serial computation approaches to deliver high performance in addition to large energy savings.

In this paper, we propose RACER, a cost-effective PUM architecture that delivers high performance and large energy savings using small arrays of resistive memories. RACER makes use of a *bit-pipelining* execution model, which can pipeline bit-serial $w$-bit computation across $w$ small tiles. We fully design efficient control and peripheral circuitry, whose area can be amortized over small memory tiles without sacrificing memory density, and we propose an ISA abstraction for RACER to allow for easy program/compiler integration. We evaluate an implementation of RACER using NOR-capable ReRAM cells across a range of microbenchmarks extracted from data-intensive applications, and find that RACER provides 107×, 12×, and 7× the performance of a 16-core CPU, a 2304-shader-core GPU, and a state-of-the-art in-SRAM compute substrate, respectively, with energy savings of 189×, 17×, and 1.3×.

**ACM Reference Format:**
Minh S. Q. Truong, Eric Chen, Deanyone Su, Alexander Glass, Liting Shen, L. Richard Carley, James A. Bain, and Saugata Ghose. 2021. RACER: Bit-Pipelined Processing Using Resistive Memory. In *MICRO'21: 54th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO '21), October 18–22, 2021, Virtual Event, Greece*. ACM, New York, NY, USA, 17 pages. https://doi.org/10.1145/3466752.3480071



*MICRO '21, October 18–22, 2021, Virtual Event, Greece*

ACM ISBN 978-1-4503-8557-2/21/10.
https://doi.org/10.1145/3466752.3480071

## 1 INTRODUCTION

Several modern applications (e.g., graph processing, genome sequencing, video processing, machine learning) routinely process large amounts of data. This large-scale data processing requires a high amount of data movement between the main memory and the CPU in a conventional computer. Unfortunately, data movement can consume as much as two orders of magnitude more energy than that needed to process the data [20, 46], and can be responsible for a majority of energy consumed for several modern applications [14]. Traditionally, data movement costs have been mitigated through the use of on-chip caches, which exploit locality to eliminate main memory accesses for data recently brought on-chip. However, the memory access patterns of many of these modern applications exhibit little locality, minimizing the benefits of caching.

To avoid data movement overheads for such applications, recent works have proposed new architectures based on the principle of *processing-using-memory* (PUM) [30, 75]. PUM architectures take advantage of electrical interactions between interconnected memory cells to perform primitive computational functions, in addition to the original role of the cells as data storage. Examples of these primitives include various families of Boolean-complete operators (e.g., [31, 43, 50, 74]) and multi-bit dot products (e.g., [8, 17, 76, 81]). Such primitives can be performed *in situ* on the data (i.e., the data never has to leave the memory cell). The principles of PUM have been demonstrated using a wide range of memory technologies, including more conventional DRAM [29, 32, 58, 74] and SRAM [1, 23, 26, 43], and emerging technologies such as resistive memories [5–8, 17, 28, 31, 33, 34, 38, 50–52, 55, 59, 76, 81, 90, 92, 94].[1]

Ideally, PUM architectures would be able to eliminate *all* data movement, thanks to their ability to perform *in situ* operations (which are performed using one or more primitives). A number of practical considerations prevent this from happening, such as (1) the wiring interconnects between cells, and (2) the exact underlying *in-situ* logic primitives enabled by a particular architecture. Data movement over the interconnect topology is likely to exist in any PUM architecture at a reasonable scale, as (a) the cost and complexity of enabling direct interactions between any two arbitrary cells is expected to be prohibitive; and (b) modern memories consist of many separate arrays of cells in order to achieve cost-effective capacity scaling, and require some form of interconnect hierarchy to enable cross-array communications. The data movement due to limitations of the underlying primitives, however, can vary significantly from architecture to architecture. As one example, Ambit [74], an in-DRAM processing architecture, can support

[1] In this work, we use the term *resistive memory* to refer broadly to resistance-based non-volatile memories (e.g., PCM, MRAM, ReRAM), while we use ReRAM to refer specifically to oxide-based switches (often referred to as memristors).

AND and OR operations in place, but must copy data to special rows of cells within a memory array to perform NOT operations. A second example involves PUM architectures that augment memory arrays with closely-placed CMOS compute units. As we discuss in Section 2.3, Boolean operators are often used in PUM architectures to perform *bit-serial* computation (i.e., a multi-bit operation is performed one bit at a time), which results in a high latency for several essential operations (e.g., addition, multiplication). To mitigate these latencies, several PUM architectures add dedicated compute units such as adders, shifters, and multipliers near the memory arrays [1, 8, 17, 38, 58, 76]. Unfortunately, adding either special memory cells or dedicated compute units can increase fabrication costs significantly, due to the reduced memory densities of the resulting chips. Dedicated compute units comes with a further cost of having to "convert" data every time the CMOS compute units are used, as the data stored in the memory cell is typically not at a voltage and/or current that is directly compatible with CMOS logic. The conversion requires the use of components such as sense amplifiers, transimpedance amplifiers, and/or analog/digital converters, which can consume non-trivial power and area [56].

In this work, we explore the feasibility of designing a PUM architecture that can minimize the need for architecture-induced data movement (i.e., without the need for special cells or discrete CMOS compute units). As prior works have shown the promise of bit-serial computation for PUM [23, 26, 29, 32, 38, 58], we aim to find alternative ways of compensating for the long latencies of serialization without significantly increasing the power or area footprints. Resistive memories arranged in a *crossbar* topology (where a memory cell sits at every intersection of row wires and column wires, with the wires directly connected to cell components) provide an opportunity to perform many bit-serial operations at once in a single array. In a resistive crossbar array, it is possible to take an entire column of the array and perform a bitwise Boolean primitive with an entire second column of the array. As a result, for a column of size $n$, we can potentially increase the throughput by a factor of $n$, which can amortize the latency of bit-serial operations if $n$ is large enough. There are, however, practical limitations to how large $n$ can become. While prior works have shown that crossbars made of ReRAM (as an example) can set $n$ as large as 1024, we show in Section 3.3 that $n$ needs to be much smaller when performing primitives on entire columns, due to the additional current generated by each cell on the column wire. Even projecting into the future, it will be difficult to achieve a value of $n$ that is much larger than 128 with existing wire materials. A larger value would result in a current that exceeds the wire's maximum current carrying capacity.

*Our goal* in this work is to co-design an architecture and its required circuitry for high-throughput, low-energy PUM, while working within the practical limitations of resistive crossbar memories. This requires us to design the architecture under three key constraints. First, our architecture should be built using small resistive memory tiles whose column lengths remain small enough to keep currents within the carrying capacities of existing wires. Second, our architecture should use an execution model that can deliver high throughput despite the small size of the tiles. Third, our architecture should aim to minimize the need for sense amplifiers, transimpedance amplifiers, or analog/digital converters whenever performing operations.

To this end, we propose *RACER* (Resistive Accelerated Computation for Energy Reduction). RACER takes advantage of a novel execution model that we call *bit-pipelining*. Column-wide bit-serial operations allow an architecture to operate on $n$ words at a time for a column size of $n$. Bit-pipelining allows us to pipeline bit-serial computation across multiple sets of words in parallel. For a word width of $w$ bits, bit-pipelining allows the architecture to operate on $w \times n$ words at a time. In RACER, we group $w$ small tiles together to form a pipeline (we set $w$ to 64 in this work to support up to 64-bit computation) that can support bit-pipelining. We stripe each bit of a $w$-bit word across all $w$ tiles in a pipeline, which allows us to leverage two observations about bit-serial computation. First, many bit-serial operations communicate information from one bit position to the next (e.g., the carry bits in addition). To enable this bit-to-bit communication in RACER, we add a one-column-wide *buffer* between every adjacent pair of tiles in a pipeline. Each buffer is made of the same resistive memory cells as the tiles themselves, and has configurable switches that connect it to at most one of its adjacent tiles at any time. Second, bit-serial operations tend to perform the same series of Boolean primitives on each bit position. As a result, we can enable the efficient pipelining of many operations by generating the primitives for only a single tile, and then propagating the sequence of primitives from tile to tile.

We design RACER to be highly scalable, to cater to the needs of different platforms. We group 64 pipelines together into a RACER *cluster*, which forms the basic unit of scalability. Each cluster has its own control units and peripheral circuitry, and can operate independently of other clusters. A RACER chip can consist of one or more clusters, which are connected together by a data sharing network for inter-cluster communication, where distributed chip-level network controllers coordinate the communication.

At the architecture level, we design RACER to serve as a co-processor that can accelerate a range of data operations in memory. To ease programmer burden, and to support a wide range of underlying resistive memory technologies (and their corresponding primitives), we propose a lightweight ISA that exposes only the high-level operations (including both bit-pipelined operations and other operations that RACER is optimized for) to programmers and/or compilers. The RACER ISA supports operations on a range of word sizes, from 8-bit words to 64-bit words.

RACER is compatible with any resistive memory crossbar, including MRAM and ReRAM. We evaluate a specific implementation of RACER that makes use of ReRAM crossbars based on MAGIC [50], which supports a single Boolean NOR primitive. We synthesize and simulate all of the circuitry for RACER, and demonstrate that it can operate within reasonable power limits and thermal densities at a 333 MHz frequency, with an 8 GB RACER chip (consisting of 4096 clusters) fitting within a 4 $cm^2$ area. Using a range of data-intensive microbenchmarks extracted from real-world applications, we quantitatively compare the performance and energy consumption of RACER to (1) a 16-core Xeon CPU baseline with conventional DRAM [39], (2) the Xeon CPU with on-chip high-bandwidth embedded MRAM, (3) a modern NVIDIA GPU with 2304 shader cores [68], and (4) the Duality Cache SRAM-based PUM architecture [26]. We show that RACER outperforms all four of these state-of-the-art systems (with an average performance of 107× that of the baseline, 71× that of the CPU with embedded MRAM, 12× that of the GPU,

and 6.7× that of Duality Cache), while delivering significant energy savings (189× over the baseline, 94× over the CPU with embedded MRAM, 17× over the GPU, and 1.3× over Duality Cache).

We make the following contributions in this work:

- We forecast how key ReRAM technology parameters are expected to scale in coming years. We use this forecast to determine how this impacts architectural design.
- We develop a novel execution model for small-tile-based processing-using-memory, called bit-pipelining. Bit-pipelining exploits communication patterns and command reuse in bit-serial computation to significantly increase throughput, while using lightweight control circuitry.
- We propose RACER, a hardware design and corresponding ISA abstraction for high-throughput, high-energy-efficiency processing using small memory tiles. RACER is compatible with any logic-capable resistive memory technology, and we design its control and peripheral circuits to maintain high memory density.

## 2 BACKGROUND

In this section, we provide a brief discussion on the underlying technologies that we build RACER on top of.

### 2.1 Crossbar Topology for Memories

As DRAM scaling issues continue to be difficult to solve [45, 64, 65], researchers have been developing a number of emerging memory alternatives, which are starting to reach commercial deployment [15, 63, 93]. These alternatives include resistive memories such as MRAM, PCRAM, and ReRAM. While the specific design of a memory cell depends on the particular technology, each memory cell generally has a *selector element* to enable access to it, and a *storage* element that holds the data. Such cells are referred to as 1S1R (one selector, one resistor) devices.

Device and circuit researchers have envisioned a way to integrate these emerging memory technologies fully into the *back-end-of-line* (BEOL) CMOS process, by using a *crossbar* array topology for access control [93], as shown in Figure 1a for ReRAM. In a crossbar, columns of metal wires connects to the selectors of all cells in the array in one layer, and rows of metal wires connects to the memory storage elements in a second layer, where one memory cell sits at the intersection of each row and each column. In a crossbar array, a single cell can be selected by asserting predetermined selection voltages on one row wire and one column wire.

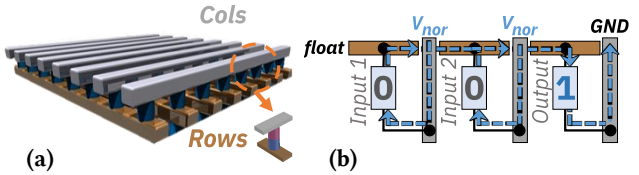


**Figure 1: (a) ReRAM crossbar: selectors shown in purple, resistive switches in blue; (b) NOR operation: current in blue.**

The crossbar topology enables the elimination of CMOS selection transistors that are required for more traditional 1T1R (one transistor, one resistor) devices [15, 93]. While current 1T1R device prototypes have more reliable selector elements, there has been significant research in recent years on improving the reliability of CMOS-less selector elements (e.g., for 1S1R devices [44, 78]). Devices with CMOS-less selector elements are fully BEOL-compatible, and leave the underlying CMOS available for logic that interacts with the memory cells (e.g., controllers, additional compute units).

### 2.2 ReRAM Devices

A redox-based RAM (ReRAM) cell is a non-volatile resistive memory device that stores data in an oxide-based switching filament, where a reduction–oxidation (redox) chemical reaction changes the oxidation state of the filament. The data is stored in the form of a resistance value, which can be changed by applying a switching voltage that triggers a redox reaction [88]. A common high-density ReRAM cell configuration is 1S1R, where a CMOS-less selector is placed in series with the oxide-based memory switch. Two-terminal ReRAM cells can be arranged into a crossbar, as shown in Figure 1a.

Each cell can store one or more bits' worth of data, where the bit count is a function of the resistance range of the memory switch. Prior work proposes to store multiple bits of data per cell [53], which can enable analog multi-bit operations such as dot products. However, given the poor stability and repeatability of programmed resistance changes relative to the total range of resistance changes possible [82], *we treat ReRAM cells as digital devices that each store only a single bit of data.* While this prevents us from using multi-bit operations, it ensures higher reliability and avoids the need for complex analog-to-digital converters (ADCs) during processing.

### 2.3 NOR-Based Computation with ReRAM

Prior work shows that through a careful choice of column selection voltages, ReRAM devices can interact with each other to perform NOR operations [50]. Figure 1b shows a NOR operation for three cells in the same row, by applying a column selection voltage of $V_{nor}$ to the two input cells and a column selection voltage of *GND* to the output cell, and keeping the row selection voltage floating (i.e., keeping the row line electrically disconnected from any voltage source). As illustrated by the current flow in the figure, floating the row line allows the input cells to determine the row line voltage, which is then used to program the output cell. This behavior can be extended to entire columns of input cells, by floating *all* row selection voltages in an array, which enables bitwise NOR for any two columns of data (they need not be adjacent columns). As NOR operations are *logically complete*, (i.e., any arbitrary Boolean function can be expressed using a combination of NOR operations), we use NOR-capable ReRAM as the building block of our architecture.

Boolean operators are commonly exploited in PUM architectures using *bit-serial* (i.e., bit-sliced) computation [1, 8, 50]. Bit-serial computation takes one word of data and performs the computation one bit at a time. A bit-serial operation can be broken down into two functions that are computed for every bit: (1) a *local* function, whose output does not need to propagate across bits; and (2) a *global* function, whose output is fed into the functions that compute the next bit in sequence. A commonly-known example of bit-serial computation is the ripple carry adder (RCA), where the addition is computed bit-by-bit starting with the least-significant bit (LSB). In RCA, the local function for bit $i$ generates the sum bit $i$, while the global function for bit $i$ generates the carry out from bit $i$, which is rippled (i.e., propagated) to bit $i + 1$.

# 3 MOTIVATION

We explore three key aspects that impact the design of *practical* crossbar-based PUM architectures. First, we discuss how the high latencies of bit-serial computation can potentially be amortized through the use of whole-column computation (Section 3.1). Second, to understand how large we can make the crossbar columns, we project how ReRAM arrays will scale in the future due to device-level innovations (Section 3.2). Third, using these projections, we discuss why column sizes are limited by inherent technological constraints, and restrict the ability of whole-column computation to fully amortize bit-serialization latencies (Section 3.3).

## 3.1 Computing on Entire Columns of Data

While bit-serial operations are a natural fit for in-memory Boolean primitive, bit-serialization comes with a significant latency penalty. For example, in order to perform addition using NOR-capable ReRAM, we would need to perform 22 NOR primitives per bit for a full addition that takes in two operand bits and one carry-in bit, and produces a sum and carry-out bit. For a NOR-capable ReRAM device where each NOR primitive takes 1 ns, a 64-bit bit-serial ripple carry addition would take 1408 ns, which is orders of magnitude higher than the latency of a 64-bit CMOS carry-lookahead-based adder (5.5 FO4 [66], or 60.5 ps in 45 nm technology [41]).

To enable bit-serial-based architectures with reasonable performance, prior works take advantage of the ability to perform a Boolean primitive on an *entire column* of a crossbar at once (see Section 2.3). For a column of length $n$, $n$ bit-serial additions can be performed in parallel. Thus, high-enough parallelism compensates for the high latency. For a large enough $n$, this potentially allows bit-serial PUM addition to achieve similar or greater throughput than a conventional CPU, without spending energy on data movement.

## 3.2 Scaling ReRAM Arrays

To understand how large we can practically make $n$, we need to consider the physical structure of emerging memory, how it is likely to scale, and the constraints that will be imposed on the architecture. Using literature on fabricated devices and projected device properties [16, 62, 67, 77, 79, 84, 85, 91], we develop Table 1. The table shows the capabilities of current ReRAM devices, conservatively-scaled attributes that researchers expect are within easy reach for fabrication in the near future, and aggressively-scaled attributes that are expected to be possible but require further research.

## 3.3 Crossbar Size Limitations

Unfortunately, the physics of ReRAM devices places significant constraints on the array size. To determine a realistic array size, we examine three key constraints. First, wire resistances can limit the array size, since these resistances increase as an array is scaled to a smaller manufacturing process technology (e.g., for a fixed-size crossbar with dimensions $n \times n$). The cell resistances can be made larger to compensate for increasing wire resistances (see Section 3.2), so this constraint is generally weak, and allows for $n > 1024$ [18]. Second, sneak path leakage currents can limit the array size. Sneak paths are a well-documented issue in crossbars (e.g., [15]), and limit the array size based on the on/off ratio of the selector element. As a result, it is challenging to build an array with $n > 1024$. Third, the current carrying capacity of the array wires can limit the array size. Estimates of the maximum current allowed in aggressively-scaled wires vary. At smaller feature sizes (e.g., 15 nm), individual nanoscale experiments suggest current densities of $1 \times 10^8$ A/cm$^2$ [27, 36]. However, VLSI technology projections that consider the statistics of failure suggest a much lower limit of $1 \times 10^6$ A/cm$^2$ or even less [60]. This suggests a maximum current ($I_{max}$) of 2–200 µA. Even the smallest of these values would allow for accessing a single cell in a crossbar of any size, (e.g., typical single-cell write currents, $I_w$ are in the range of 1–10 µA).

While sneak path leakage currents are the limiting factor for single-cell access, this changes when we perform primitives on entire columns. Accessing all $n$ elements in a column simultaneously places $n$ times the current on the wire, because the cells are connected in parallel and the column access voltage stays constant. As a result, *the limiting factor for array size in an architecture with whole-column computation is the wire's current carrying capacity*. For such architectures, it will be very difficult to build arrays where $n > 200$ (i.e., $n = I_{max}/I_w$), and we will likely be limited to even smaller array sizes when failure statistics are considered.

Such small memory arrays pose a major problem for practical fabrication. A perennial concern for memory arrays is the *peripheral circuitry*, which is used to select cells/columns, assert read/write voltages, and sense cell state through measurements of resistance (e.g., to enable I/O with other components). In a PUM architecture, peripheral circuits would also be responsible for managing whatever computation takes place. From an area efficiency perspective, larger memory arrays are favored because they allow the peripheral circuits (which typically grow linearly with $n$) to be amortized over a greater array area (which grows with $n^2$). Unfortunately, for smaller arrays (e.g., where $n \leq 128$), traditional peripheral circuitry would be significantly larger than the size of the actual cell array, leading to very poor area efficiency.

We conclude that a practical PUM architecture needs to (1) use small memory arrays to stay within technology constraints, and (2) rethink the peripheral circuitry to avoid poor memory densities.

**Table 1: ReRAM technology scaling trends (†: derived from other attributes).**

| Attribute | Current Technology | Conservatively Scaled | Aggressively Scaled |
|---|---|---|---|
| Switch Latency (ns) | 10 [16, 67] | 2 [84] | 0.5 |
| Switch Voltage (V) | 2 [91] | 1.5 | 1 |
| Switch Energy† (pJ) | 0.8 | 0.09 | 0.0064 |
| On Resistance (kΩ) | 10 [62, 79] | 50 | 500 |
| Off Resistance (MΩ) | 1 [62, 79, 85] | 5 | 100 |
| BEOL Position | M3–M5 | M3–M5 | M3–M5 |
| Access Topology | 1T1R [77] | 1S1R [84] | 1S1R |

# 4 RACER CLUSTER DESIGN

We set out to design a new architecture that can deliver the performance that large arrays would achieve under whole-column operations, using the small arrays that technology constrains us to. To this end, we propose RACER. The basic unit in RACER is a *cluster*, which contains several *tiles* (i.e., small ReRAM arrays, which we size to $64 \times 64$ in this work) that work together to execute a series of commands. A cluster contains all of the necessary components to store data in tiles, perform logic operations, and perform I/O.

In this section, we describe key cluster design choices, including (1) *buffers* to facilitate inter-tile data transfer, (2) our *bit-pipelining* execution model across multiple tiles, (3) control circuitry, and (4) I/O circuitry. We discuss the programmer interface in Section 5.

## 4.1 Communicating Using Buffers

A key technique to enabling higher throughput with small tiles is to provide an efficient mechanism to move a vector of data from one tile to another without relying on cross-domain converters. Unfortunately, while we could connect adjacent tiles together using controllable pass gates, this effectively creates a double-width tile, and can result in current values that exceed the wire carrying capacities discussed in Section 3.3. When multiple sets of pass gates are enabled simultaneously, many adjacent tiles may be fused together in a chain, effectively creating a large array.

Instead, we introduce *buffers*, as shown in Figure 2a. A buffer consists of a $1 \times 64$ column of ReRAM cells, and is connected to two neighboring tiles with pass gates to each neighbor (one pass gate per row). RACER's control circuitry ensures that a buffer can connect to only *one* of its neighboring tiles at a time, ensuring that the largest fused crossbar at any point is $65 \times 64$. Inter-tile communication copies data into and out of buffers,[2] as shown in Figure 2b. Our topology of interleaving tiles and buffers allows us to perform a transfer from all tiles to an attached buffer in parallel.

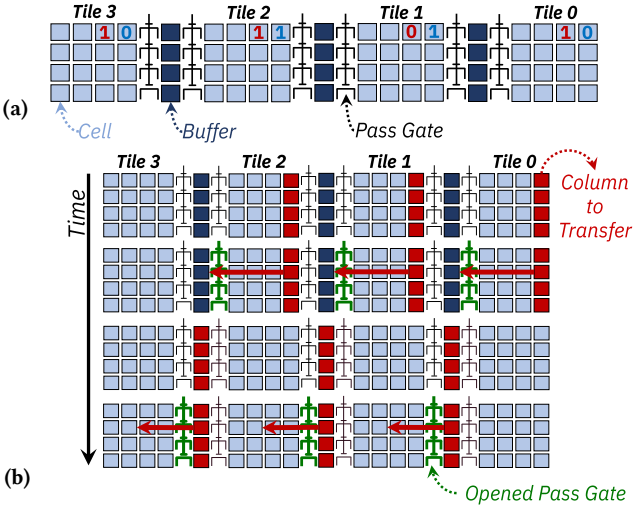


**Figure 2: (a) Tile and buffer design, showing two values (1101 in red, 0110 in blue); (b) inter-tile data transfer.**

## 4.2 Bit-Pipelining

RACER makes use of the repetitive logic behavior of bit-serial operations to enable our high-throughput execution model, called *bit-pipelining*. To enable bit-pipelining, RACER stores data in a *bit-striped* format, where the bits of a $w$-bit word are distributed across $w$ tiles. Each bit of the word occupies the same row and column coordinate in every tile, as shown in Figure 2a. As RACER supports up to 64-bit word sizes, we group the 64 tiles that hold the bits of the same 64-bit word together into a tile *pipeline*.

[2]In RACER, we program one column of each tile to always hold zeroes. We can then NOR any other column of data with this zero column to perform a copy operation.

This bit-striped format allows us to take advantage of a key insight in many bit-serial operations: *the same operations are applied sequentially to each bit position.* Bit-serial computation in Tile $t$ typically takes the form of using operands in Tile $t$, including data generated by the computation in Tile $t-1$, generating a result for the current bit position, and propagating some portion of the result to Tile $t+1$ (which can then perform the same operations). In RACER, each tile's bit-serial computation is represented as a sequence of *micro-ops* (instructions that specify the column address to perform NOR primitives on). We exploit this bit-serial behavior to improve the efficiency of micro-op execution in two ways.

First, we design our controllers to let a tile reuse the micro-ops previously executed by a neighboring tile. As a result, our control circuitry no longer needs to generate instructions for every tile in a pipeline. Instead, it generates the micro-op sequence for a single tile, and then configures the pipeline to automatically propagate this sequence from tile to tile (see Section 4.3).

Second, after a tile finishes executing its current micro-op sequence and passes the sequence to its neighbor, the tile is free to work on a new micro-op sequence while its neighbor tile performs the just-passed micro-op sequence. This effectively allows RACER to perform pipelined execution across each bit of an operand. This is akin to the pipeline in a CPU (albeit at a much finer granularity), where RACER's tiles take the place of processing stages while its buffers take the place of pipeline registers.

## 4.3 Pipeline Control Circuitry

Figure 3 shows the control circuitry used to perform bit-pipelining. In RACER, we support 8-bit, 16-bit, 32-bit, and 64-bit operands. Given that the smallest supported granularity is 8 bits, we divide the control circuitry up into *byte groups*. Within a byte group, micro-op sequences are *always* propagated from one tile to another. To enable micro-op sequence propagation from tile to tile, we implement a series of circular FIFO queues, one per tile, which we call *micro-op queues*. Because a tile can only execute one NOR primitive every clock cycle in RACER, Tile $t$ may need to wait multiple cycles for Tile $t-1$ to produce its needed data and transfer it to the shared buffer (for use in Tile $t$'s computation). While Tile $t$ waits for this data, its micro-op queue starts gathering the micro-ops already completed by Tile $t-1$. To do this, the control circuitry includes a bus that connects the head of Tile $t-1$'s instruction queue to the tail of Tile $t$'s queue. Micro-ops are propagated between the queues *every* cycle. The start of execution on Tile $t$ is triggered the cycle after an inter-tile copy micro-op is inserted in the tile's micro-op queue (i.e., when the copy is executed by Tile $t-1$), as the inter-tile copy indicates that the intermediate data from bit $t-1$ is now ready for use during the computation of bit $t$.

The byte group contains two other components besides the micro-op queues. First, a *broadcast bus* bypasses the micro-op queues, allowing RACER to send the same micro-op to every tile in the byte group simultaneously. This is used by operations that execute the same micro-op but do not propagate data from tile to tile (e.g., a multi-bit Boolean operation). Second, a *direction reversal switch* allows us to reverse the direction that micro-ops flow from tile to tile, allowing instruction propagation from MSB to LSB instead of from LSB to MSB. Within a byte group, the direction

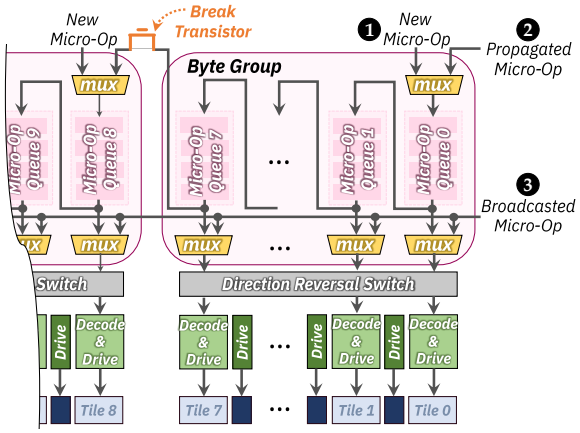


**Figure 3: Byte group circuitry.**

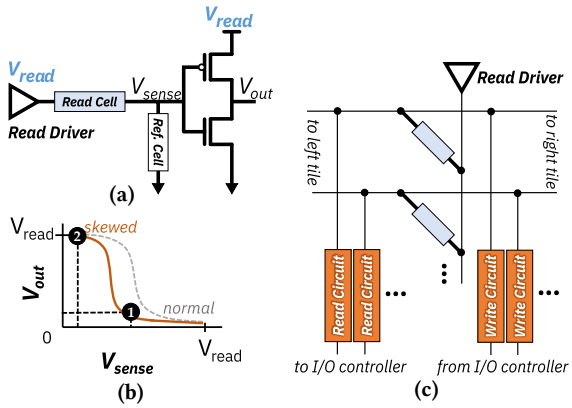


**Figure 4: (a) Read circuitry for one cell in a buffer; (b) voltage transfer characteristics graph of a skewed inverter used in the read circuitry; (c) read/write circuitry for a buffer.**

reversal switch connects Micro-Op Queue 0 to either Tile 0 (in forward mode) or Tile 7 (in reverse mode), Micro-Op Queue 1 to either Tile 1 (forward mode) or Tile 6 (reverse mode), etc. This reversal is useful for bit-serial operations that start at the MSB tile, such as the rectified linear unit (ReLU) or magnitude comparison.

To support micro-op sequence propagation for operands greater than 8 bits, we can connect byte groups together dynamically. To enable this connection, we support three ways of populating the first queue in a byte group (as shown in Figure 3): new micro-op insertion (❶ in the figure), which allows one or more byte groups to receive a new micro-op; micro-op propagation (❷), which receives a micro-op from the last queue of the previous byte group when a break transistor is turned on; and micro-op broadcast (❸).

When a micro-op reaches the head of the micro-op queue, it is dispatched to a *decode & drive* unit, as shown in Figure 3. Each micro-op is made up of (1) three 6-bit fields to indicate the input and output columns for the NOR primitive, and (2) a 2-bit buffer control field that determines if either of the buffers should be connected to the tile (i.e., if the micro-op is for an inter-tile copy, with the two bits indicating whether to copy to the left, copy to the right, or not copy). The decode & drive unit uses these fields to determine which three columns to enable in order to execute the NOR primitive, and uses a voltage selector to assert the correct voltages. We find empirically that a 32-micro-op instruction queue length is sufficient to hold all of the micro-op sequences for our operations (see Section 5.1).

## 4.4 Read/Write Circuitry

We build lightweight circuitry to read data out of and write data into tiles, designing this circuitry carefully to avoid diminishing the ReRAM density. Figure 4a shows our circuitry for reading from an ReRAM cell. We use a voltage divider (consisting of the cell to be read and a reference ReRAM cell) and a skewed inverter to determine if the cell holds a logic 0 or a logic 1, by detecting the voltage $V_{sense}$. The reference ReRAM cell is fixed to a low resistance state, so when a read voltage $V_{read}$ is applied to the cell to be read, $V_{sense}$ will be one of two values (as shown in Figure 4b): ❶ $\approx 0.5 V_{read}$, if the read cell has low resistance (representing logic 1), as there is an equal voltage drop over the read cell and the reference cell; or ❷ $\approx 0$ V, if the cell has high resistance (representing logic 0), as most of the voltage drops over the read cell.

If we were to connect the read circuit to a tile, the voltage divider would not work efficiently, as it would detect the full resistance of the cell being read as well as partial resistances from other cells on the wire. To avoid this scenario, we perform reads only on buffers. Each cell in a buffer is connected to its own read circuit, allowing us to potentially read out the entire contents of a buffer in one cycle.

For the write circuitry, we also cannot write directly to a column in a tile without adding extra peripheral circuitry. Without this additional circuitry, a write with the current decoder design would populate new data into *every* column of a tile. This additional peripheral circuitry would require significant area, diminishing the memory density. Instead, similar to the read circuits, we attach our write circuits only to buffer cells. The write circuit contains a write driver, which asserts a voltage that changes the buffer's resistance. An entire column of data can be written into the buffer in two cycles: the first cycle presets the entire buffer to all zeros, and the second cycle writes logic 1 to all enabled cells.

Figure 4c shows the buffer-level read/write circuit layout. We discuss how to connect this to an I/O controller in Section 5.2.

## 4.5 Control Amortization & Cluster Scaling

So far, we discuss the components needed to control a single RACER pipeline. Unfortunately, the control circuitry required to enable bit-pipelining and support reads/writes consumes significantly more area (308×) than the 64 tiles in a pipeline Similarly, adding the decode & drive unit to every tile would consume over 30 W of static power for an 8 GB chip, and every pipeline would expose 4096 read/write ports to the I/O controller. To avoid these expensive scaling factors, we share components across *multiple* pipelines, leading us to include 64 pipelines in each cluster.

In RACER, we share a single set of pipeline control circuitry and decode & drive units across all 64 pipelines. In other words, a single instruction queue $t$ and its corresponding decode & drive unit is shared across 64 tiles (every Tile $t$ in the 64 pipelines). We add a lightweight *pipeline selector* circuit to choose which of the 64 pipelines receives commands from the decode & drive unit. While this means that only a single pipeline in a cluster can be active at a time, this reduces the thermal density of RACER to 0.02 W/mm$^2$, well within the limits tolerable by conventional cooling methods.

To reduce the number of read/write ports, we multiplex the ports on a 64-bit bus, with each wire dedicated to a single buffer in the pipeline. Every cycle, the bus reads out one bit per buffer, which across all 64 buffers corresponds to all of the bits of a bit-striped 64-bit word. Again, only the active pipeline can perform I/O, so we need only a single bus for an entire cluster. The bus serves as the communication interface to the cluster.

Figure 5 shows the final design of a cluster, including all shared circuits. The cluster serves as the basic block of RACER. A RACER chip can contain an arbitrary number of clusters. For example, a chip for resource-constrained platforms may have only a single cluster, while a chip for larger platforms may contain thousands of clusters. We discuss inter-cluster communication in Section 5.2.

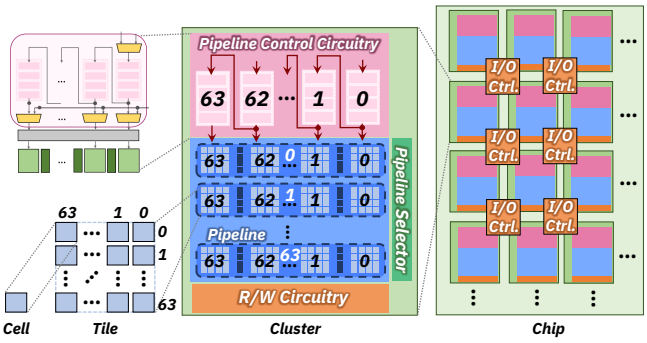


**Figure 5: RACER cluster and chip design.**

# 5 RACER ARCHITECTURE

In this section, we describe a programming abstraction and communication network that allows multiple clusters to work together to execute a data-intensive workload. We discuss this abstraction in the context of a system where RACER is integrated as a co-processor (serving as both main memory and a PUM platform), though RACER can potentially serve as a standalone architecture.

## 5.1 RACER Instruction Set Architecture

We hide the complexity of the underlying RACER components by introducing the notion of a *RACER core*. Each RACER core is physically mapped to a unique RACER pipeline, and exposes vector register sets to the programmer that provide low-latency access to the 32 kB of memory cells within the pipeline. Because a pipeline can support multiple bit granularities, we use a technique similar to vector extensions for the x86 ISA: we provide different vector register sets for each word length (byte/half/single/double sets for 8-/16-/32-/64-bit lengths, respectively), but these vector register sets map to the same underlying memory locations, as shown in Figure 6a. For a RACER chip with an 8 GB capacity (4096 clusters, 64 pipelines per cluster), there are 256K RACER cores.

To program different RACER cores, programmers first identify which cores they want to call. Programmers can then execute programs using the RACER ISA. Table 2 lists the different operations that RACER exposes in its ISA. While most of these instructions support all of our word lengths, `POPC`, `MUL`, and `MAC` cannot operate on 64-bit inputs, as their output data would be larger than 64 bits and would exceed the width of our pipeline. RACER supports primitive predicated branch execution using the `MUX` instruction.

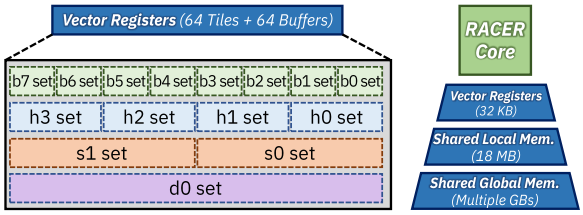


**Figure 6: (a) Vector register set mapping to a pipeline; (b) RACER programming abstraction.**

**Table 2: RACER ISA (†: non-bit-pipelined operations).**

| Op. | Description/Notes | Op. | Description/Notes |
|---|---|---|---|
| *Arithmetic Operations* | | | |
| ADD | Two's complement add | ABS | Absolute value |
| SUB | Two's complement subtract | MUX | Multiplex (i.e., choose) |
| POPC | Population count | RELU | Rectified linear unit |
| CMPEQ | Check equality | LSHIFT | Left shift by 1 |
| FUZZY | Fuzzy search | RSHIFT | Right shift by 1 |
| MUL† | Multiply (only 8-/16-/32-bit) | SQRT† | CORDIC square root |
| MAC† | Multiply–accumulate | SIN† | CORDIC sine |
| DIV† | Division (returns quotient & remainder) | COS† | CORDIC cosine |
| MAX | Searches for the maximum number | EXP† | CORDIC exponent |
| MIN | Searches for the minimum number | CAS | Compare and swap |
| *Boolean Operations* | | | |
| NOR | Bitwise NOR | OR | Bitwise OR |
| NAND | Bitwise NAND | AND | Bitwise AND |
| NOT | Bitwise NOT | XOR | Bitwise XOR |
| *Data Transfer Operations* | | | |
| MOV | <**MOV** *buff*[*dst*] = *buff*[*src*]> Moves data stored in buffers of core *src* to buffers of core *dst* | SHIFT | <**SHIFT** *stride*> Parallel data shift *dst* = *src* + *stride* |
| *Configuration Operations* | | | |
| SET | <**SET** *start*, *stop*, *stride*> Turns on RACER core *i* for *i* ∈ *range*(*start*, *stop*, *stride*) | UNSET | Turns off all RACER cores that are active |

## 5.2 Data Sharing Network

Because each RACER core only has low-latency access to 32 kB of data (i.e., its own vector registers), it may need to access data stored on other cores for certain applications. To enable core-to-core communication, we design a lightweight, distributed network for data sharing. Recall that each cluster has a 64-bit read/write bus (Section 4.5). As we stamp out multiple clusters next to each other in a grid within a chip, we design an I/O controller that provides non-uniform memory access (NUMA) to the other clusters in the chip, laid out as shown in Figure 5.

Each I/O controller connects four neighboring clusters together, enabling local communication across the clusters (and across different pipelines in the same cluster). Each cluster is connected to four different I/O controllers, as shown in Figure 5. Thus, each RACER core has local access to nine clusters (except for cores in clusters along a chip edge), providing 18 MB of local memory.

The I/O controllers enable access to non-neighboring clusters by communicating with each other. Each local I/O controller works independently, avoiding the need for centralized hardware (and thus avoiding a potential communication bottleneck). Instead, to enable cross-chip communication, we connect all of the I/O controllers together using a mesh network. Each network request contains (1) source/destination core IDs and (2) a priority level, both of which are used to make local routing decisions at each network hop. We observe in the applications we study that most communication is local, allowing us to use such a simple global network.

To enable off-chip communication, each I/O controller is connected to a 512-bit chip-wide shared bus that interfaces with the

CPU. A controller can burst up to eight cache lines of data (i.e., 512 B) to the CPU on this bus. Our circuit simulations show that moving 512 B of data from an I/O controller to the CPU interface takes 16 ns, and can support a peak bandwidth of 32 GB/s.

RACER exposes NUMA to the programming interface as two tiers of shared memory: local memory and global memory (Figure 6b). The ISA provides two types of data transfer operations, shown in Table 2: (1) `MOV`, a scalar operation that can move all of the data stored in the 64 buffers of a core to another core; and (2) `SHIFT`, which shifts all data stored in all active cores' buffers to new cores.

## 5.3 Support for Non-Pipelined Operations

While RACER is optimized for bit-pipelined operations, we can use it to enable a number of shift-based, non-pipelined operations (integer multiply, multiply–accumulate, integer divide, trigonometric functions and square root using CORDIC [86]). A characteristic of non-pipelined operations is that each tile in a pipeline can either (1) execute a unique micro-op not previously executed by another tile (hence the micro-op cannot be pipelined), or (2) remain idle for a number of cycles before executing a unique micro-op. While the execution behavior for these operations is more complicated than that of bit-serial operations, we can leverage RACER's bit-striping and ReRAM buffers to efficiently perform these operations.

In non-pipelined mode, RACER configures the micro-op queues (Section 4.3) to store only one micro-op at any time. For every cycle, Micro-Op Queue $t$ copies its current micro-op to Micro-Op Queue $t+1$, and then overwrites its existing micro-op with the micro-op that it receives from Micro-Op Queue $t-1$. This effectively turns the micro-op queues of a byte group into an eight-register scan chain, where it takes eight cycles for all micro-ops to reach the correct queues. Although the performance of non-pipelined mode is significantly lower than that of bit-pipelining due to the lack of micro-op reuse, RACER can still enable high-performance execution of these operations without adding new logic blocks.

Some non-pipelined operations such as multiply or multiply–accumulate do not require all eight tiles in a byte group to be active at the same time. For such operations, the controller issues special all-zero-bit micro-ops to disable the decoder & drive units of certain tiles in the byte group. All eight byte groups of a RACER core can populate their scan chains in parallel, and the core can issue a new set of micro-ops to its 64 tiles every eight cycles.

We briefly describe two operations that make use of RACER's non-pipelined mode. Although these operations are based on different algorithms, they both take advantage of RACER's low-latency, low-overhead shift abilities. By employing bit striping across tiles, we can implement shift operations in programs simply by using inter-tile data transfers through the buffers. This avoids the need for any dedicated circuitry for shift operations, which a number of prior PUM architectures require (as they typically store all bits of a number in a single array). RACER's `LSHIFT` and `RSHIFT` operations perform bulk shifting, as they are applied to entire columns, and can be performed in parallel by multiple tiles in a pipeline.

**Wallace-Tree-Based Multiplication**. RACER performs $n$-bit multiplication ($C = A \cdot B$) in three steps, similar to tree-reduction-based CMOS multipliers: (1) it generates partial products, (2) it reduces the partial products using the Wallace tree reduction algorithm [24], and (3) it performs a final ripple carry addition (RCA) of the last two rows of partial products. In Step 1, $n$ rows of partial products are generated from the multiplier ($A$) and multiplicand ($B$). In Step 2, a predetermined micro-op sequence of full adder equations is sent to the micro-op queues to reduce $n$ rows of partial products to two rows, based on the Wallace tree algorithm. In Step 3, a single RCA is sent to the tiles to add the last two rows into the output $C$.

Figure 7 shows an example with $A$=0101; $B$=1110. First, RACER uses `LSHIFT` and `RSHIFT` operations to create shifted copies of $A$ and $B$ (❶ in the figure; $B$ not shown for clarity). We refer to $A$'s left-shifted-by-$i$ copy on line $i$ as $A_i$. Second, RACER ANDs all bits of $A_i$ with the $i^{\text{th}}$ bit of $B$ to generate four partial products (❷). Third, RACER performs column-wise adds based on the Wallace tree reduction algorithm (❸). The addition reduces the number of partial product lines to 3 (❹). Next, the carry bits are left-shifted by one (❺). The tree reduction (❸–❺) repeats until only two partial products remain (❻). A final RCA (❼) generates the product $C$ (❽).

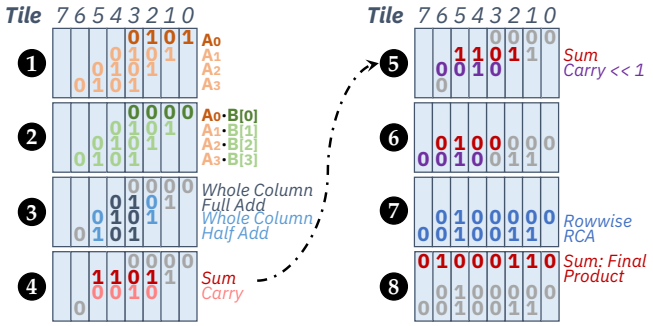


**Figure 7: Multiplication in RACER (buffers not shown).**

This sequence of operations is analogous to the operations performed by a Wallace-tree-based CMOS multiplier. The key difference between the two is that RACER performs all full adds using the same set of tiles, whereas a CMOS multiplier requires multiple physical full adder circuits arranged in a tree-like structure, consuming a significant amount of power and area. A secondary difference is that RACER can adapt to other types of tree-based multiplication, such as the Dadda multiplier [19], without needing to physically rewire the circuit, which cannot be done in CMOS.

**CORDIC-Based Functions**. The CORDIC algorithm [86] is a method of iteratively updating three input values $x$, $y$, and $z$ over multiple iterations of bitwise operations to perform complex mathematical functions such as sine, cosine, or square root. The exact function performed depends on the initial values of $x$, $y$, and $z$. CORDIC performs the following calculations at each iteration $i$:

$$x[i+1] = x[i] - \sigma_i 2^{-i} y[i] \quad (1)$$

$$y[i+1] = y[i] + \sigma_i 2^{-i} x[i] \quad (2)$$

$$z[i+1] = z[i] + \sigma_i tan^{-1}(2^{-i}) \quad (3)$$

The value $\sigma_i$ is the sign of $z_i$ (i.e., either 1 or 0). In RACER, we precompute two sets of potential outputs (i.e., $x[i+1]$, $y[i+1]$, $z[i+1]$) predicated on the value of $\sigma_i$. A `MUX` operation selects the correct set of outputs for that iteration. The values of $2^{-i}y[i]$ and $2^{-i}x[i]$ (Equations 1 and 2) can be computed in RACER using `RSHIFT` operations. Because $x$, $y$, and $z$ are stored as two's complement numbers, RACER can perform the necessary additions and

subtractions for the update equations using RCA. The only value that cannot be computed is $tan^{-1}(2^{-i})$ (Equation 3), which requires the $tan^{-1}$ function. We decide to store a lookup table populated with precomputed entries for this function. A RACER core loads the lookup table values into its vector registers before performing the iterative update. We empirically set the number of iterations and the number of lookup table entries to 12, to achieve a balance between reasonable output accuracy and the storage/latency overheads.

## 5.4 Example Application: grep

We describe how an 8 GB RACER chip executes *grep*, which counts the number of instances of an 8-bit word in 64 files, each of size 58,720,256 words (about 1700 pages single spaced). In RACER, we split grep into three phases: (1) local *grep* within each RACER core, (2) partial result reduction across the cores in a cluster (intra-cluster *grep*), and (3) result reduction across the entire chip.

Algorithm 1 illustrates the first phase of *grep*, as executed on a single RACER core. Because we are operating on an 8-bit word size, the program operates on eight byte sets (b[0]–b[7]) with 64 vectors each (denoted as v[i], where $0 \leq i < 64$). We reserve seven vectors per byte set for the intermediate values generated during addition, and one vector per byte set to store the search pattern (v[56]), leaving 56 vectors available per byte set for computation. For each character in a file, we store the results of a bitwise comparison with the search pattern. Hence, 28 vectors store the file words, while 28 other vectors store the results. We add the results together across all eight byte sets to find the number of matches in the core.

**Algorithm 1:** Single-core *grep* within a pipeline

```
   // Turn on all cores to do the following
1  SET 0, 262144, 1 ;
   // 8 byte sets compute in parallel
2  for parallel s in range(0,7) do
       // Compare all words in set to vector 56
3      for i in range(0,27) do
4          CMP b[s]v[i], b[s]v[i+28], b[s]v[56];
5      end
       // Count number of matches in the same set
6      for i in range(0,27) do
7          ADD b[s]v[0], b[s]v[0], b[s]v[i];
8      end
9  end
   // Reduce partial results amongst 8 sets
10 for s in range(0,27) do
11     ADD b[0]v[0], b[0]v[0],b[s]v[0];
12 end
   // Partial results stored in b[0]v[0]
13 UNSET;
```

Figure 8 shows the second and third phases of *grep*, where all partial results are added together across multiple clusters. To simplify the figure, we show this using a small 4×4-cluster chip in the figure, but the principle can be applied to RACER chips with much larger dimensions. From the single-core *grep* algorithm, each cluster holds 64 partial results, one per core. All 16 clusters can perform the intra-cluster *grep* program (G; the second phase of *grep*) in parallel, which brings 64 partial results into the same core using the cluster's designated I/O controller, and then perform 64 additions (❶ in Figure 8). Each cluster then holds one partial result. RACER uses the data sharing network to transfer the partial results of every second cluster column to its neighboring cluster, such that the neighbor now holds two partial results (❷). Then, an addition is

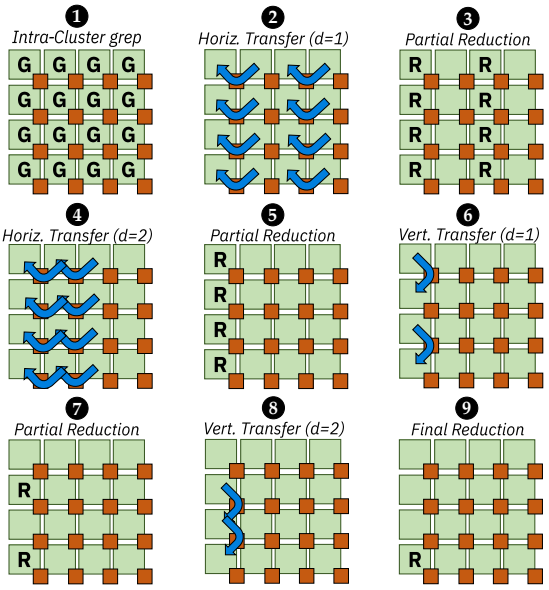


**Figure 8: RACER *grep* reduction operations, illustrating intra-cluster *grep* (G) and partial result reduction (R). Clusters shown in green, and I/O controllers shown in orange.**

performed in each tile with two partial results to reduce the number of partial results from 16 to 8 (❸). At this point, only every second cluster holds the partial results. Hence, in order to put two partial results in a single cluster again to perform the next reduction, the partial results from one cluster have to hop through two I/O controllers before they reach the destination cluster (❹), at which point the partial results can be added together (❺). This horizontal bulk data transfer between columns of clusters is performed using RACER's `SHIFT` instruction. Note that because the data sharing network is a 2D mesh, horizontal (and vertical) partial reductions can take multiple hops $d$, where $d = 1, 2, 4, 8, \ldots$. In the example, $d = 1, 2$. After the horizontal reduction, we are left with four partial results in our example, all of which are in the same cluster column as each other. We can then reduce these partial results to one number, by consolidating partial results in the vertical direction (❻–❽). Finally, after the reductions are complete, the result in the bottom left cluster is the result of *grep* for the entire chip (❾).

Our simulation of grep on RACER (with an 8 GB chip) shows that almost 85% of the time and 99% of the energy is spent on the single-core and intra-cluster *grep* phases, indicating that the RACER data sharing network provides low-overhead inter-core communication.

## 5.5 System-Level Considerations

**Memory Addressing**. RACER currently requires a programmer to use physical addresses for execution (since RACER cores are tied to physical addresses). Like other works, we do not implement virtual memory for RACER, as virtual memory for processing-in-memory faces several complex challenges [2, 12, 30, 37, 49, 80]. We leave a full-feature virtual memory abstraction for RACER as future work.

**Exceptions**. RACER's execution model can incorporate support for precise exceptions. The model uses SIMD execution, where a fleet of RACER cores are activated in parallel to execute the same set of

instructions, and a new fleet is not activated until the current fleet finishes. If an active core needs to pause because of an exception, (1) the exception is recorded and the current set of instructions is saved by the RACER core controller; (2) the active cores without exceptions keep executing until the `UNSET` instruction is reached; and (3) the paused core rolls back if required by the exception handler, and replays the saved instructions.

# 6 METHODOLOGY

**Modeling & Simulation**. We model RACER at the device, circuit, and architecture levels. We develop a highly-detailed Verilog-A model of an ReRAM cell partially based on the aggressively-scaled device parameters in Table 1, while still enforcing the current carrying limit of the wire as discussed in Section 3 (1 ns switching latency for the resistive element, 1 : 500 ratio of on–off resistances, 0.0128 pJ per switch transition). We synthesize RACER's CMOS-domain components using Synopsis Design Compiler with FreePDK 15 nm [11] to estimate the area, power, and critical path latency of each component, along with delay and power models for the wires connecting these components together (inserting signal repeaters for long wires when necessary). To evaluate microbenchmarks, we develop RACER-Sim, a detailed in-house simulator for RACER that incorporates the data collected from our Verilog-A model and synthesized circuits. This simulator faithfully models (1) execution at a cell granularity, and (2) all data movement and communication across the data sharing network. We have open-sourced RACER-Sim [9], and discuss how to run it in the appendix.

**Comparison Points**. We quantitatively compare RACER to four state-of-the-art systems: (1) **Baseline**, a 16-core CPU modeled after the Intel Xeon Platinum 8253 [39], which uses a conventional off-chip DRAM for main memory; (2) **eMRAM**, a variant of *Baseline* that replaces the off-chip DRAM with a high-bandwidth (333.3 GB/s for reads) connection to *on-chip* embedded magnetic RAM (MRAM) [22, 87]; (3) **RTX2070**, an NVIDIA GPU with 2304 shader cores [68]; and (4) **DC**, the Duality Cache [26] architecture proposed for in-SRAM computation, with a 35 MB cache capacity (the same as proposed in the original work) and a connection to off-chip DRAM for main memory. Table 3 summarizes major parameters of the comparison points. The eMRAM configuration represents a future technology where multi-gigabyte MRAM banks can potentially be stacked on top of a CPU, eliminating the off-chip data movement bottleneck and enabling sophisticated compute *near* memory. We optimistically assume that the MRAM is fully multi-ported (with one read port and one write port per last-level cache subbank in the CPU), to minimize contention. Each port is 512 bits wide, to allow for a single-cycle transfer of an entire cache line from the MRAM peripheral circuitry to the last-level cache (vs. 4 DRAM cycles for DDRx DRAM). We evaluate Baseline and eMRAM using MARSSx86 [71], McPAT [57], and DRAMSim2 [73], making significant modifications to DRAMSim2 to support our aggressive embedded MRAM technology (we have open-sourced this modified version [9]). We use a real Geforce RTX 2070 GPU with the nvprof profiler [70] to gather execution time and average power. We optimistically account for only kernel execution time, ignoring memory allocation and host setup latencies. We evaluate DC by faithfully replicating all latencies and energy values reported in the original work [26] from architectural simulations and circuit models (we validate the circuit models), and developing optimistic arithmetic models for DC's performance and latency.

**Table 3: System parameters.**

| Configuration | Parameters |
|---|---|
| RACER-1024/4096 | *No. Clusters*: 32 × 32/64 × 64, 333.3 MHz, 2/8 GB capacity<br>*Cluster–Cluster Bandwidth*: 1 GB/s<br>*Total Mesh Bandwidth*: 1024/4096 GB/s<br>*Interface w/ Host*: DDR-like, 32 GB/s BW, bus width: 64 bit |
| Baseline | x86 [40], 16 cores, 4-wide, OoO, 2.2 GHz, TDP: 125 W<br>*L1I/D Cache*: 16 × 32 kB private, 8-way assoc.<br>*L2 Cache*: 16 × 16 MB private, 16-way assoc.<br>*L3 Cache*: 22 MB shared, 11-way assoc.<br>*Main Memory*: DDR3L [42], 8 GB, bus width: 64 bit |
| eMRAM | Same CPU configuration as Baseline<br>*Main Memory*: DDR-like, 8 GB, bus width: 512 bit<br>read: 3 ns, write: 10.5 ns<br>IDD4R: 170.9 mA, IDD4W: 267.2 mA |
| RTX2070 | Turing [69], 2304 CUDA cores, 1.4 GHz, TDP: 175 W<br>*L1I/D Cache*: 64 kB per SM; *L2 Cache*: 4 MB<br>*Main Memory*: GDDR6, 8 GB, bus width: 256 bit |
| DC | *L3 Duality Cache* [26]: 35 MB, 2.5 GHz<br>*Compute Energy*: 10.26 pJ/cycle/32 B (scaled to 15 nm)<br>*Access Energy*: 5.73 pJ/cycle/32 B (scaled to 15 nm) |

**Microbenchmarks**. We evaluate RACER using 13 data-intensive microbenchmarks, which span multiple application fields: (1) image processing (*brightness*, *rgb2gray*), (2) linear algebra (*mmul*, *mvmul*, *pagerank*), (3) signal processing (*dftSparse*, *dftDense*), (4) classification (*manhattan*, *hamming*, *lenet5*), and (5) string matching (*grep*, *exactMatch*, *fuzzyMatch*). We compile optimized 16-thread x86 ISA versions of our benchmarks for Baseline and eMRAM using gcc [25] with the `-O3` flag. We use nvcc with the `-O3` flag to compile GPU kernels. The kernels are based on highly-optimized algorithms for GPUs, and we carefully design them to minimize branch divergence and maximize memory coalescing. Due to their custom ISAs, we hand-compile microbenchmarks for RACER and DC, optimizing the data mapping to maximize locality for each architecture.[3]

# 7 EVALUATION

We perform iso-area comparisons to our four comparison points (Baseline, eMRAM, RTX2070, and DC; see Section 6). We compare Baseline, eMRAM, and RTX2070 to **RACER-4096**, a 4 $cm^2$ RACER chip with 4096 clusters that uses less area than a Xeon Platinum 8153 CPU [89] and a Geforce RTX 2070 GPU [83]. We compare DC to **RACER-1024**, a RACER chip with 1024 clusters that fits within the 1.4 $cm^2$ area of a 35 MB Duality Cache implemented as a last-level cache (LLC) [26]. We obtain the LLC area and power from McPAT, with appropriate scaling to the 15 nm technology node. Given the smaller memory capacity of RACER-1024, we run our microbenchmarks with reduced data sets that fit within a 2 GB main memory footprint, and normalize these results to Baseline runs of the reduced-data-set microbenchmark (denoted as **Baseline–2GB**).

## 7.1 Area & Circuit Synthesis Analysis

Table 4 shows a breakdown of the area, static power, and dynamic power consumed by each component of the RACER circuitry. We use *back-end-of-line* (BEOL) integration, where the ReRAM cell

[3]Optimal data mapping is a challenging problem for most processing-in-memory architectures, and we leave the investigation of automated mapping to future work.

**Table 4: Costs of RACER cluster components and global corner-to-corner (C2C) wiring with repeaters.**

| Component Name | Area ($\mu m^2$) | Delay (ps) | Power ($\mu W$) Static | Power ($\mu W$) Dynamic | # of Instances Per Cluster |
|---|---|---|---|---|---|
| Tile C2C Wiring | – | 19 | 0 | 482 | – |
| Cluster C2C Wiring | – | 140 | 0 | 714 | – |
| Chip C2C Wiring | – | $16\times10^3$ | 0 | $17\times10^3$ | – |
| 1 64 × 64 ReRAM Tile | 3.7 | – | – | – | 4096 |
| 64 Tiles & 64 Buffers | 240 | – | – | – | 64 |
| Bit-Pipeline Control | $74\times10^3$ | 100 | 677 | 775 | 1 |
| Decode & Drive Unit | 277 | 27 | 3.11 | 126 | 64 |
| Pipeline Selector | 76 | 4 | 0.54 | 73 | 1 |
| Selector Passgates | 0.001 | 0 | 0 | $2\times10^{-5}$ | $64\times2\times64\times64$ |
| I/O Controller | 9600 | 174 | 128 | $8\times10^3$ | 1 |
| Driver Passgate | 0.001 | 0 | 0 | $2\times10^{-5}$ | $64\times3\times64\times64$ |
| R/W Circuit | 0.001 | 1 | 0 | $1\times10^{-5}$ | $64\times64$ |
| **1 Cluster + 1 I/O Ctrl.** | $84\times10^3$ | – | 800 | 790 | – |
| **RACER-1024** | $1.4\times10^8$ | – | $82\times10^4$ | $97\times10^5$ | – |
| **RACER-4096** | $4\times10^8$ | – | $33\times10^5$ | $30\times10^6$ | – |

materials can be deposited on metal layers 3–5, with the remaining layers available for building traditional CMOS transistors. Each ReRAM cell occupies a $4\,F^2$ area in a crossbar topology, and we design a cluster such that the ReRAM array, all cluster control, and peripheral circuitry fit within an approximately similar footprint to improve area efficiency. Figure 9 shows a preliminary floorplan of the different circuit components, including all wiring. Throughout the circuit design process and construction of the RACER chips, we conservatively enforce a maximum power budget of 80 W. From our circuit synthesis, we find that RACER's circuit critical path is 2.6 ns, including wire delay, with each cluster dissipating 0.8 mW of static power. To include a modest guardband, we conservatively clock RACER at 333 MHz (i.e., a 3 ns cycle time).

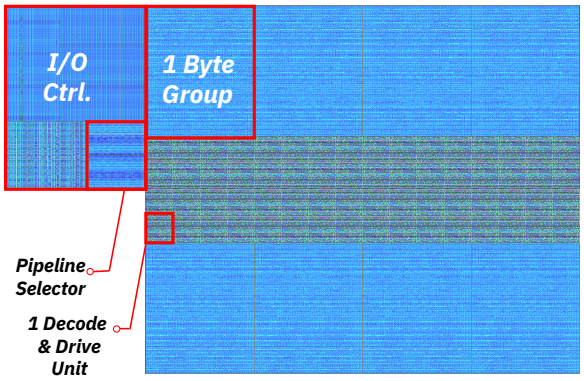


**Figure 9: Preliminary floorplan of a RACER cluster.**

## 7.2 Performance Analysis

Figure 10 shows the performance for each of our evaluated systems. We make five observations from the figure.

First, eMRAM achieves modest performance improvements over Baseline, with an average speedup of 1.5×. This is because several benchmarks are able to take advantage of the high on-chip bandwidth provided by eMRAM. We observe a direct correlation between increased bandwidth utilization and increased performance. For example, *rgb2gray*'s bandwidth utilization increases from 5.4 GB/s to 7.9 GB/s. The increased utilization, combined with alleviating bursty port contention that *rgb2gray* experiences in Baseline (due to per-subbank ports), allows the microbenchmark to achieve a 2.9× speedup. However, despite providing eight times the bandwidth of the DRAM used in Baseline, eMRAM does not achieve a

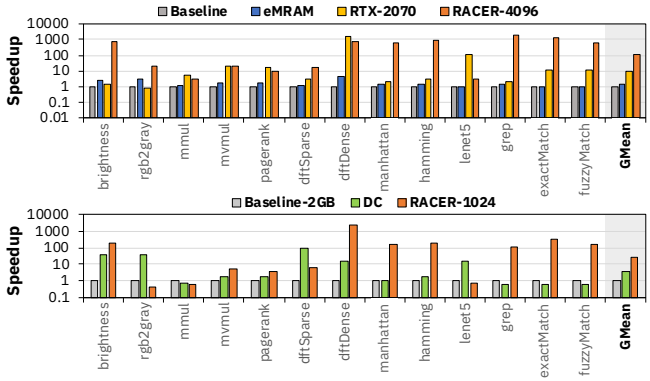


**Figure 10: Speedup vs. Baseline (top) and Baseline–2GB (bottom).**

commensurate increase in performance. This is because the caches act as a bottleneck for many of these applications (which we verify with cache sensitivity studies; not shown due to space limits), and thus the increased eMRAM bandwidth is only of limited use. Thus, we conclude that simply eliminating off-chip accesses and exposing high bandwidth are not enough to deliver large performance increases for data-intensive applications.

Second, RACER enables significant performance improvements over Baseline, with an average speedup of 107×. Instead of relying on temporal and spatial locality in memory access patterns, many microbenchmarks on RACER benefit from *operand locality*, where the operands of the same operation are stored in the same pipeline to make use of bit-pipelining. This reduces the need to stall for memory transfers over the data sharing network, which would potentially increase the memory access latency significantly. *brightness*, *dftDense*, *manhattan*, *hamming*, *grep*, *exactMatch*, and *fuzzyMatch* are all able to take advantage of operand locality.

RACER also benefits from providing three different levels of parallelism for bit-serial operations: (1) tile-level parallelism (a NOR primitive operates on 64 words at a time), (2) pipeline-level parallelism (bit-pipelining enables 64 primitives to execute in parallel in a single pipeline), and (3) *cluster-level parallelism* (with $n$ clusters, we can turn on $n$ pipelines concurrently). As a result, when operand locality is maximized and there is enough data, RACER-4096 can operate on as many as 16M words concurrently, easily amortizing the long latencies of bit-serialization.

Third, microbenchmarks consisting of mainly non-bit-pipelined operations still benefit from RACER's potential for parallelized execution. *rgb2gray*, *mmul*, *mvmul*, *dftSparse*, and *lenet5* achieve a reasonable average speedup of 8.9× compared to *Baseline*. This is notably lower than the speedup of other benchmarks because of the heavy use of non-pipelined operations. Recall from Section 5.3 that we need to disable micro-op reuse for non-pipelined operations. This requires micro-ops to be brought into the byte group serially. Non-piplined operations cannot hide this front-end serialization, while being unable to benefit from pipeline-level parallelism. However, RACER still provides tile-/cluster-level opportunities for parallelism, and eliminates data movement and cache access latencies that exist in Baseline. For example, even though 95.7% of the execution time of *lenet5* is spent on non-pipelined operations (primarily `MAC`), it can inference across 85 images in parallel on RACER-4096, as a single image requires only seven clusters.

Fourth, RACER achieves an average speedup of 12× compared to RTX2070. For streaming applications such as *brightness* and *rgb2gray*, RACER-4096 outperforms RTX2070 because it can process data in-place, while RTX2070 needs to copy data from host memory to the GPU global memory, and later transfer it from global memory to the shader cores. *lenet5* performs significantly better on RTX2070, due to the GPU kernel's ability to exploit significant weight reuse (reducing memory transfer overheads).

Fifth, RACER significantly outperforms DC, which is a state-of-the-art PUM architecture that relies on discrete logic to avoid the high bit-serialization latencies for common operations. DC achieves a modest average speedup of 3.8× compared to Baseline–2GB, despite offering many opportunities for parallel PUM computation. This is because DC uses SRAM cells and their associated peripheral circuitry, both of which are significantly larger than their ReRAM counterparts. As a result, in the same area that RACER can store and process 2 GB of data, DC, can only store and process 35 MB. Even though we map data and computation to DC to maximize locality, DC still requires many accesses to off-chip DRAM to swap in different working sets of data. For microbenchmarks such as *lenet5* and *rgb2gray*, DC significantly outperforms RACER because these microbenchmarks do not require many off-chip accesses. *lenet5* can store all of its weight values in DC throughout the entire execution, requiring only a small amount of image data to be brought in from DRAM. In contrast, benchmarks such as *grep* perform significantly better on RACER than on DC. On DC, *grep* spends 99.8% of its execution time waiting for new data to come in from DRAM. In contrast, RACER's distributed approach to computing *grep* (Section 5.4) allows it to extract significant parallelism and highly-local memory accesses. Averaged across all of our microbenchmarks, RACER achieves a 6.7× speedup over DC.

We conclude that RACER's bit-pipelining execution model, and its supporting control circuitry, effectively extract very high levels of parallelism, resulting in significant performance improvements.

## 7.3 Energy Analysis

Figure 11 shows the normalized energy savings for our evaluated systems. We make four observations from the figure.

First, eMRAM achieves modest system energy reductions by eliminating DRAM-specific processes such as precharge, activation, and refresh. This allows eMRAM to reduce memory power by 20× compared to Baseline, in addition to eliminating off-chip bus energy. However, eMRAM does not eliminate the cost of data traversing through the cache hierarchy, limiting the memory power reduction.

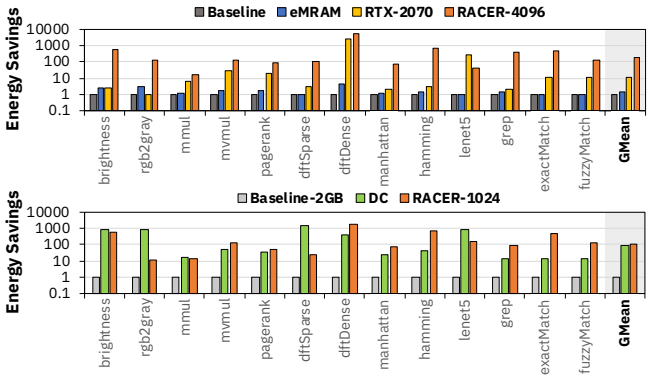


**Figure 11: Energy savings, normalized to Baseline (top) and Baseline–2GB (bottom).**

Second, RACER provides large energy savings over Baseline, with an average savings of 189×. These savings go beyond simply eliminating data movement. In Baseline, the memory and cache subsystems consume 12% of the system energy. RACER achieves a much larger reduction thanks to its low-power design, with no dedicated logic, no need for costly voltage sensing for local memory operations, and efficient amortization of costly components over multiple pipelines (Section 4.5). As a result, RACER-4096 consumes an average power of 31 W (including 3 W of static power), compared to 54 W for Baseline even with power gating. When this lower power is combined with RACER-4096's faster execution time, the energy drops significantly compared to Baseline.

Third, RACER achieves an average energy savings of 17× over RTX2070. RTX2070 achieves significant energy savings compared to Baseline because of its significantly faster execution time. However, RACER consumes less power than the GPU (31 W vs. 45 W) while providing even faster execution times.

Fourth, RACER uses less energy than DC for some microbenchmarks, while using more energy for others. As a result, RACER has a modest average energy savings of 1.3× over DC. We study two closely-related microbenchmarks, *dftSparse* and *dftDense*, to understand the different classes of energy consumption. *dftSparse* performs 32K separate 64-point transforms, which allows the microbenchmark to reuse its discrete Fourier transform (DFT) matrix 32K times. As we saw with performance, DC can exploit this locality to avoid off-chip memory accesses, greatly improving its efficiency. In contrast, *dftDense* performs a single transform on a 9216-point input, which requires DC to continuously stream in the matrix from DRAM. RACER's much denser memory array allows for the entire matrix to be stored on-chip, resulting in its much lower energy consumption. Across all microbenchmarks that trigger frequent data swapping in DC (*mmul*, *mvmul*, *pagerank*, *dftDense*, *manhattan*, *hamming*, *grep*, *exactMatch*, *fuzzyMatch*), RACER can achieve a 4.9× energy savings compared to DC.

We conclude that RACER is a highly-energy-efficient platform for computation.

## 7.4 Lifetime Analysis

One limitation on the practicality of RACER is the endurance of emerging non-volatile memories. We explore the desired memory cyclability for RACER (assuming 24/7 operation) by examining the ReRAM buffers, which incur the greatest write frequency. We assume that buffer writes occur at most every other cycle (to allow the data to be read out). Since only one pipeline in a cluster is active at a time, we assume that we will cycle through all 64 pipelines in the cluster roughly equally over the memory's lifetime. We calculate that RACER's buffers would incur $8.61 \times 10^{14}$ writes over a 10-year lifetime. If we factor in real-world usage (our microbenchmarks have a buffer write activity factor of only 0.0446), the typical-case lifetime write count is $7.68 \times 10^{13}$.

As a comparison point, state-of-the-art ReRAM devices can endure $10^{12}$ writes before failing [61]. We believe that given current industrial investments in ReRAM, and the nascent nature of the

technology, device-level advancements will achieve the endurance desired by RACER in the future. Alternatively, RACER can be built on top of contemporary technologies with better endurance, such as MRIMA [4] (which makes use of MRAM).

### 7.5 Comparison to CASCADE

We briefly compare RACER-4096 to an area-equivalent state-of-the-art CASCADE [17] accelerator, which uses ReRAM to perform analog dot products for neural networks. CASCADE does not support most of RACER's operations, and has a fixed neural-network-centric dataflow, so we can compare performance only using the *mmul*, *mvmul*, *dftSparse*, *dftDense*, and *lenet5* microbenchmarks. For these benchmarks, RACER-4096 achieves a speedup over CASCADE of 6×, 4×, 0.01×, 1.5×, and 0.09×, respectively (not shown).

We observe that while *mmul*, *mvmul*, and *dftDense* are memory-bound on CASCADE due to their large working set sizes, and significant swapping with external memory, *dftSparse* and *lenet5* do not require data swapping, and fully benefit from the fast analog dot product. We conclude that RACER trades off some of the performance of neural network accelerators in order to provide much broader acceleration opportunities.

## 8 RELATED WORK

To our knowledge, RACER is the first tile-based, high-performance, energy-efficient PUM architecture that addresses the circuit-level limitations of whole-column computations in small resistive crossbars. We highlight our key contributions over prior PUM works.

**Processing-Using-SRAM**. Several works enable computation in caches (e.g, [1, 23, 26, 43]). These architectures leverage high locality for data already stored in cache subbanks, but their limited storage capacity incurs frequent data accesses to DRAM for applications with large working sets. RACER's scalable design allows gigabytes' worth of data to be stored on-chip, and its efficient mesh network provides high-performance access to the full on-chip memory. We quantitatively compare RACER to Duality Cache [26] in Section 7.

**Processing-Using-DRAM**. Recent works propose to perform computation in DRAM [29, 32, 58, 74]. Due to the storage of data as a capacitive charge, DRAM cells require additional support to reliably perform operations using memory, such as dedicated rows with special bitlines to enable NOT primitives [32, 74]. Furthermore, given the large row size (8 kB in DDR3/DDR4), performing bit-serial operations directly on row-wise data can be inefficient, forcing the architectures to either (1) transpose data during compute so that operands are stored on a single bitline [32], which adds significant latency and energy; or (2) add dedicated shift logic to enable bit-serial operations without transposing data [58]. Unlike DRAM-based PUM architectures, RACER does not need to transpose data, and does not require modifications to the memory array.

**Processing-Using-PCM/MRAM**. Existing architectures that use PCM or MRAM as their compute-enabling technology have been primarily based on 1T1R devices [5–7, 56]. While 1T1R devices currently demonstrate better reliability than crossbar arrays, the addition of a CMOS access transistor greatly reduces the area available to implement peripheral circuitry, significantly reducing the memory density. Recent research (e.g., [44, 78]) has greatly improved the reliability of crossbars, making 1T1R-based architectures less desirable. RACER can be built on top of transistor-less PCM/MRAM crossbar technologies, as these resistive memory devices have similar high-level constraints and features as ReRAM.

**Analog Processing-Using-ReRAM**. Crossbars enable the ability to perform multiple dot products in parallel using ReRAM, by mapping (1) the input matrix values to multi-bit ReRAM cells, and (2) the input vector to analog voltages applied on every column. PUM architectures that use this primitive (e.g., [8, 17, 76, 81]) typically target neural network inference. However, they sacrifice memory density because they require: (1) area-intensive digital/analog converters to perform analog dot products, and (2) dedicated CMOS logic near ReRAM tiles to reduce the partial results generated in the tile. Moreover, multi-bit analog operations are difficult to perform reliably in ReRAM, because of device non-linearity that requires significant precision to discern between adjacent bit representations. In contrast, RACER uses an all-digital approach that avoids the need for costly supplemental logic components and significantly increases reliability, while offering a much larger set of PUM operations that can handle a wide range of data-intensive applications. We briefly compare RACER to CASCADE [17] in Section 7.5.

**Digital Processing-Using-ReRAM**. Several PUM architectures demonstrate the ability to perform logic using ReRAM crossbars (e.g., [28, 31, 33, 34, 38, 50, 51, 55, 90, 92]). However, as we discuss in Section 3, these architectures are often limited in throughput they can achieve without the assistance of discrete logic elements, due to practical limits on array sizes when performing whole-column operations. RACER's bit-pipelining execution model and controller circuitry provide a way to enable high-throughput, low-energy bit-serial computation using small tiles.

**Bit-Striping**. Several prior works make use of bit-striping (i.e., bit-planing) for applications such as data compression [3, 47], data analysis [13, 48], and efficient storage for resistive memory [21, 35, 72]. To our knowledge, RACER is the first work to build a general-purpose pipelined execution model on top of bit-striping.

## 9 CONCLUSION

We propose RACER, a highly-efficient architecture for processing using resistive memory. Resistive memories such as ReRAM have technology limitations that will prevent their memory arrays from scaling to large sizes for the foreseeable future. This is at odds with the need for large arrays in traditional PUM architectures, which use large whole-column operations to amortize the costs of bit-serial computation. RACER uses a novel execution model called bit-pipelining, along with a highly-scalable architecture, to deliver high performance with small arrays. Our evaluations show that RACER provides significant performance and energy benefits over state-of-the-art systems. We hope that RACER can enable the practical use of resistive memories for PUM.

## ACKNOWLEDGMENTS

We thank Raghav Gupta, Yuezhang Zou, and Shivani Prasad for their feedback on this work. This work was funded in part by a seed grant from the Wilton E. Scott Institute for Energy Innovation, and by the Data Storage Systems Center at Carnegie Mellon University.

# A ARTIFACT APPENDIX

## A.1 Abstract

We are releasing a set of artifacts [9] that capture the performance and energy consumption of RACER, our processing-using-memory (PUM) architecture.

The first artifact, **RACER-Sim**, is a custom simulator that calculates the latency, throughput, and energy of applications executing using a tiled crossbar memory, using the same architecture proposed for RACER, but not limited to ReRAM-only memory devices. For inputs, RACER-Sim takes in (1) device-level latency and energy costs for the different arithmetic operations supported by the RACER architecture (with these costs derived from accurate device-level modeling), and (2) a program written in the RACER ISA (for which we currently hand-compile microbenchmarks). We use RACER-Sim to model the performance and energy consumption of the two RACER configurations evaluated in the paper (*RACER-4096* and *RACER-1024* in Section 7).

The second artifact is **MARSS-CPU**, an *unmodified* version of the MARSSx86 architectural simulatorMARSSx86 [71] with custom configuration files. The simulator is bundled with DRAMSim2 [73] and McPAT [57], which provide accurate models of the DRAM subsystem and power/energy, respectively. We use MARSS-CPU evaluate the performance and energy of the two conventional CPU configurations evaluated in the paper (*Baseline* and *Baseline–2GB* in Section 7).

The third artifact is **MARSS-eMRAM**, a *modified* version of MARSSx86, which models an embedded main memory stacked on top of the CPU. We provide configuration files to model state-of-the-art embedded MRAM, though the configuration can be modified to model other embedded memory devices. We use MARSS-eMRAM to evaluate the performance and energy of the embedded MRAM configuration in the paper (*eMRAM* in Section 7).

## A.2 Artifact Checklist (Meta-Information)

- **Programs**: RACER-Sim, MARSSx86, DRAMSim2, McPAT
- **Output**: Performance and energy results for every benchmark
- **Experiments**: (1) Run `racer-sim` simulations to obtain RACER-4096 and RACER-1024 results, (2) run MARSS-CPU to obtain Baseline and Baseline–2GB results, (3) run MARSS-eMRAM to obtain eMRAM results, (4) extract speedups and energy savings from simulation results and compare to baselines.
- **Disk space required (approximate)**: 10 GB
- **Time needed to complete experiments (approximate)**: Generating all baseline microbenchmark results will take approximately 200 hours using MARSSx86's checkpointing feature and batch mode.
- **Publicly available**: Yes
- **Code license:** MIT for RACER-Sim, GNU GPLv2 for MARSSx86, BSD for DRAMSim2 and McPAT
- **Workflow framework used**: Custom shell and Python scripts to execute simulators and extract results from simulator runs
- **Archived**: https://doi.org/10.5281/zenodo.5495803, with disk images and a virtual machine (VM) available at https://doi.org/10.5281/zenodo.5495019

## A.3 Description

*A.3.1 How to Access.* We maintain a publicly-available repository where we have open-sourced all of our artifacts [9], with snapshots of the repository archived. We also provide an archive [10] that contains (1) disk images for use with our artifacts and (2) a virtual machine with our artifacts precompiled and ready to execute.

*A.3.2 Software Dependencies.* Ubuntu 14.04.5, Python 3 ($\geq$ 3.6), Bash $\geq$ 4.3, GNU Make $\geq$ 3.4, Linux kernel version $\geq$ 3.13, SCons 2.3

## A.4 Installation

The RACER-Artifacts repository includes the following directories:

- `./marss-cpu/`: MARSSx86 with an unmodified version of DRAMSim2
- `./marss-emram/`: MARSSx86 with a version of DRAMSim2 modified to simulate on-chip embedded MRAM
- `./mcpat`: McPAT source code used for Baseline, Basleline–2GB, and eMRAM
- `./racer-sim/`: RACER-Sim files
- `./racer-sim/src/`: Contains the source code for RACER-Sim
- `./racer-sim/cost/`: Contains spreadsheets detailing the device- and circuit-level costs of RACER
- `./programs/`: Contains microbenchmarks written in C++ and in the RACER ISA, used as inputs for Baseline/Baseline–2GB/eMRAM and RACER-4096/RACER-1024, respectively
- `./scripts/`: Contains scripts to setup and process simulation outputs

**Step 1: RACER-Sim**. The RACER simulator does not require any further software packages other than the default Python 3.6 installation.

**Step 2: DRAMSim2**. Installation guide can be found at https://github.com/dramninjasUMD/DRAMSim2. We have provided the correct versions of DRAMSim2 in `./marss-cpu/` and `./marss-emram/`. Compile the DRAMSim2 shared library object with the following command from inside the DRAMSim2 directory (repeat this for each MARSSx86 directory):

```
make libdramsim.so
```

**Step 3: MARSSx86**. Installation guide can be found at https://github.com/dramninjasUMD/marss.dramsim. Once completed, do the following:

- Download and unzip the disk images `ubuntu-natty.qcow2` and `MRAM-ubuntu-natty.qcow2` required for the simulator from the images archive [10].
- Compile `marss.dramsim` with the following command:

```
scons -Q c=16 dramsim=<libdramsim.so location>
```

The disk images that we provide already contain the necessary microbenchmark checkpoints. MARSSx86 can run Baseline and Baseline–2GB using the `ubuntu-natty.qcow2` image, and can run eMRAM with the `MRAM-ubuntu-natty.qcow2` image. Users can point the `marss.dramsim` simulators to the downloaded images by specifying the images' locations in the file `./marss.dramsim/util/util.cfg`. Step 5 of the installation process, and Step 2 of the experiment workflow (Section A.5), can be skipped if users need to use only the provided checkpoints.

**Step 4: McPAT**. Installation guide can be found at https://github.com/HewlettPackard/mcpat. Compile with `make`.

**Step 5: Compile Microbenchmark Binaries for MARSSx86** ***(can be skipped if using provided checkpoints)***. C++ versions of the microbenchmarks are available in the `./programs/marss/src/` folder. Run `make` in that folder, which will create the microbenchmarks' binaries in `./programs/marss/bin/`. Note that the microbenchmarks' source files require the macro

`#include ptlcalls.h` to perform checkpointing. The `ptlcalls.h` file can be found in the `marss.dramsim/ptlsim/tools/` directory (users will need to change the include path appropriately to point to the correct file). Then, the binaries can be uploaded to the disk image with the following command from the `./scripts/upload2image/` folder:

```
bash upload.sh
```

Please specify the disk image and mounting locations in `./scripts/-upload2image/upload.sh`.

## A.5 Experiment Workflow

**Step 1: Run RACER-Sim to Generate RACER-4096 and RACER-1024 Results**. Our scripts can automatically run RACER-Sim and collect all microbenchmark latency and dynamic energy costs with the following commands in the `./scripts/racer/` folder:

```
bash runracer1024.sh

bash runracer4096.sh
```

**Step 2: Create Checkpoints for MARSSx86** ***(can be skipped if using provided checkpoints)***. We recommend reading through the detailed tutorial [54] on how to checkpoint and run batch mode for MARSSx86.

Before running checkpointing, users need to update the `qemu_img` variable in `./marss-cpu/marss.dramsim/util/create_checkpoint.py` to the location of the disk image. To create a checkpoint, from the installed `./marss-cpu/marss.dramsim/` directory, enter the following command:

```
./util/create_checkpoint.py
```

Users can delete the provided checkpoint in the disk images and replace it with their own using the following command:

```
qemu-img snapshot -d <checkpoint name> <disk image
name>
```

To list available checkpoints in the image, run the following command:

```
qemu-img info <disk image name>
```

**Step 3: Run MARSSx86 to Generate Baseline, Baseline–2GB, and eMRAM Results**. Before running the simulations and obtaining the results, users will have to specify (1) the location where MARSSx86 can create result files, using the `-d` flag; and (2) how many threads the simulation can run on, using the `-n` flag. The flag variables need to be edited in both the `./marss-cpu/marss.dramsim/runbench.sh` file (for running Baseline) and the `./marss-cpu/marss.dramsim/runbench2G.sh` file (for running Baseline–2GB). The simulations can then be run with the following commands:

```
bash runbench.sh

bash runbench2G.sh
```

Creating checkpoints for and generating the eMRAM results can be done in the same manner as listed above for the Baseline runs, by editing and running `./marss-emram/marss.dramsim/runbench.sh`

**Step 4: Post-Process DRAM Numbers**. To obtain the DRAM's interconnect throughput and average power necessary to calculate the baselines' energy cost, execute the following command:

```
python3 ./scripts/postprocessing/grabPower.py <DRAM
log file>
```

The log file is generated as a part of running `marss.dramsim`.

**Step 5: Post-Process CPU Numbers**. The total cycle time of baselines are available from the `yml` file created when `marss.dramsim` runs. To extract the times, run:

```
python3 ./scripts/postprocessing/grabCycle.py <yml
file>
```

To obtain the CPU average power, we need to convert the `yml` output file to McPAT's `xml` format. Inside the installed `marss.dramsim` directory, run the following command:

```
./util/marss2mcpat.py -marss <yml file> -xml_in
Xeon.xml –cpu_mode user -o <unique xml name>
```

Then, power numbers can be obtained by running the following command in the installed `./mcpat/` directory:

```
./mcpat -infile <xml file produced previously>
-print_level 5
```

## A.6 Evaluation and Expected Results

There are eight attributes we obtain from running the simulations:

- **From `./scripts/racer/runracer<1024/4096>.sh`**:
  (1) RACER-1024/RACER-4096 energy
  (2) RACER-1024/RACER-4096 total latency
- **From `./scripts/postprocessing/grabCycle.py`**:
  (3) Baseline/Baseline–2GB/eMRAM cycle time
- **From `./scripts/postprocessing/grabPower.py`**:
  (4) Baseline/Baseline–2GB/eMRAM DRAM average power
  (5) Baseline/Baseline–2GB/eMRAM off-chip interconnect throughput
- **From McPAT**:
  (6) Baseline/Baseline–2GB/eMRAM core power (including L1/L2 cache power)
  (7) Baseline/Baseline–2GB/eMRAM L3 cache power
  (8) Baseline/Baseline–2GB/eMRAM interconnect power

We provide a comparison script that quickly calculates the speedups and energy savings based on the eight attributes above:

```
python3 ./scripts/compare/compare.py <attribute csv
file>
```

Each row of the CSV file contains the attributes for each microbenchmark. Run `./scripts/compare/example.csv`, which should generate the reported 107× speedup and 189× energy savings for RACER-4096 compared to Baseline. We expect the users' simulation runs to yield identical geometric mean speedups and energy savings to those reported in Section 7.

# REFERENCES


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

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

[3] I. Alam, S. Pal, and P. Gupta. 2019. Compression With Multi-ECC: Enhanced Error Resiliency for Magnetic Memories. In *MEMSYS*.

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

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

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

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

[8] 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. 2019. PUMA: A Programmable Ultra-Efficient Memristor-Based Accelerator for Machine Learning Inference. In *ASPLOS*.

[9] ARCANA Research Group. 2021. RACER Artifacts — Zenodo Repository. https://github.com/ARCANA-Research/RACER-Artifacts/. archived at https://doi.org/10.5281/zenodo.5495803.

[10] ARCANA Research Group. 2021. RACER Images — Zenodo Repository. https://doi.org/10.5281/zenodo.5495019.

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

[12] D. Bhattacharjee and A. Chattopadhyay. 2016. Delay-Optimal Technology Mapping for In-Memory Computing Using ReRAM Devices. In *ICCAD*.

[13] B. Bonney, R. Ives, D. Elter, and Y. Du. 2004. Iris Pattern Extraction Using Bit Planes and Standard Deviation. In *ACSSC*.

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

[15] A. Chen. 2016. A Review of Emerging Non-Volatile Memory (NVM) Technologies and Applications. *SSE* (Nov. 2016).

[16] Y.-S. Chen, T.-Y. Wu, P.-J. Tzeng, P.-S. Chen, H.-Y. Lee, C.-H. Lin, F. Chen, and M.-J. Tsai. 2009. Forming-Free $HfO_2$ Bipolar RRAM Device With Improved Endurance and High-Speed Operation. In *VLSIT*.

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

[18] A. Ciprut and E. G. Friedman. 2017. Modeling Size Limitations of Resistive Crossbar Array With Cell Selectors. *TVLSI* (Jan. 2017).

[19] L. Dadda. 1965. Some Schemes for Parallel Multipliers. *Alta Frequenza* (Aug. 1965).

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

[21] N. Dastanova, S. Duisenbay, O. Krenstinskaya, and A. P. James. 2018. Bit-Plane Extracted Moving-Object Detection Using Memristive Crossbar-CAM Arrays for Edge Computing Image Devices. *IEEE Access* (2018).

[22] X. Dong, W. Wu, G. Sun, Y. Xie, H. Li, and Y. Chen. 2008. Circuit and Microarchitecture Evaluation of 3D Stacking Magnetic RAM (MRAM) as a Universal Memory Replacement. In *DAC*.

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

[24] A. El-Gamal, D. Gluss, J. Greene, J. Reyneri, and P.-H. Ang. 1986. A CMOS 32b Wallace Tree Multiplier-Accumulator. In *ISSCC*.

[25] Free Software Foundation, Inc. [n. d.]. GCC, the GNU Compiler Collection. https://gcc.gnu.org/.

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

[27] S. Fujisawa, T. Kikkawa, and T. Kizuka. 2003. Direct Observation of Electromigration and Induced Stress in Cu Nanowire. *JJAP Letters* (Dec. 2003).

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

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

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

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

[32] 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. 2021. SIMDRAM: A Framework for Bit-Serial SIMD Processing Using DRAM. In *ASPLOS*.

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

[34] 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. 2015. Memristor-Based Computation-in-Memory Architecture for Data-Intensive Applications. In *DATE*.

[35] Z. He, S. Angizi, and D. Fan. 2017. Exploring STT-MRAM Based In-Memory Computing Paradigm With Application of Image Edge Extraction. In *ICCD*.

[36] M. C. Hersam, A. C. F. Hoole, S. J. O'Shea, and M. E. Welland. 1998. Potentiometry and Repair of Electrically Stressed Nanowires Using Atomic Force Microscopy. *APL* (Feb. 1998).

[37] R. B. Hur, N. Wald, N. Talati, and S. Kvatinsky. 2020. SIMPLER MAGIC: Synthesis and Mapping of In-Memory Logic Executed in a Single Row to Improve Throughput. *TCAD* (Oct. 2020).

[38] M. Imani, S. Gupta, Y. Kim, and T. Rosing. 2018. FloatPIM: In-Memory Acceleration of Deep Neural Network Training With High Precision. In *ISCA*.

[39] Intel Corp. [n. d.]. Intel Xeon Platinum 8253. https://ark.intel.com/content/www/us/en/ark/products/192465/intel-xeon-platinum-8253-processor-22m-cache-2-20-ghz.html.

[40] Intel Corp. 2016. *Intel® 64 and IA-32 Architectures Software Developer's Manual, Vol. 3.*

[41] International Technology Roadmap for Semiconductors. 2015. ITRS 2.0, 2015 Edition. http://www.itrs2.net/.

[42] JEDEC Solid State Technology Assn. 2013. *JESD79-3-1A.01: Addendum No. 1 to JESD79-3 — 1.35 V DDR3L-800, DDR3L-1066, DDR3L-1333, DDR3L-1600, and DDR3L-1866.*

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

[44] S. H. Jo, T. Kumar, S. Narayanan, W. D. Lu, and H. Nazarian. 2015. 3D-Stackable Crossbar Resistive Memory Based on Field Assisted Superlinear Threshold (FAST) Selector. In *IEDM*.

[45] U. Kang, H.-S. Yu, C. Park, H. Zheng, J. Halbert, K. Bains, S. Jang, and J. Choi. 2014. Co-Architecting Controllers and DRAM to Enhance DRAM Process Scaling. In *The Memory Forum*.

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

[47] J. Kim, M. Sullivan, E. Choukse, and M. Erez. 2016. Bit-Plane Compression: Transforming Data for Better Compression in Many-Core Architectures. In *ISCA*.

[48] J. Kim, M. Sullivan, E. Choukse, and M. Erez. 2019. A Novel Convolution Computing Paradigm Based on NOR Flash Array With High Computing Speed and Energy Efficiency. In *ITCS*.

[49] E. Kultursay, M. Kandemir, A. Sivasubramaniam, and O. Mutlu. 2013. Evaluating STT-RAM as an Energy-Efficient Main Memory Alternative. In *ISPASS*.

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

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

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

[53] S. Lee, Y.-B. Kim, M. Chang, K. M. Kim, C. B. Lee, J. H. Hur, G.-S. Park, D. Lee, M.-J. Lee, C. J. Kim, U.-I. Chung, I.-K. Yoo, and K. Kim. 2012. Multi-Level Switching of Triple-Layered TaOx RRAM With Excellent Reliability for Storage Class Memory. In *VLSIT*.

[54] T. S. Lehman, Q. Wang, S. M. Zahedi, and B. C. Lee. 2014. Datacenter Simulation Methodologies: MARSSx86 and DRAMSim2. https://www.seas.upenn.edu/~leebcc/tutorial/dsm14/01-sim.pdf.

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

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

[57] S. Li, J. H. Ahn, R. D. Strong, J. B. Brockman, D. M. Tullsen, and N. P. Jouppi. 2009. McPAT: An Integrated Power, Area, and Timing Modeling Framework for Multicore and Manycore Architectures. In *MICRO*.

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

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

[60] J. Lienig. 2013. Electromigration and Its Impact on Physical Design in Future Technologies. In *ISPD*.

[61] Q. Luo, J. Yu, X. Zhang, K. H. Xue, J. H. Yuan, T. Gong, H. Lv, X. Xu, P. Yuan, J. Yin, L. Tai, S. Long, Q. Liu, X. Miao, J. Li, and M. Liu. 2019. $Nb_{1-x}O_2$ Based Universal Selector With Ultra-High Endurance (>$10^{12}$), High Speed (10 ns) and Excellent $V_{th}$ Stability. In *VLSIT*.

[62] H. Lv, X. Xu, P. Yuan, D. Dong, T. Gong, J. Liu, Z. Yu, P. Huang, K. Zhang, C. Huo, C. Chen, Y. Xie, Q. Luo, S. Long, Q. Liu, J. Kang, D. Yang, S. Yin, S. Chiu, and M. Liu. 2017. BEOL-Based RRAM With One Extra-Mask for Low Cost, Highly Reliable Embedded Application in 28 nm Node and Beyond. In *IEDM*.

[63] A. Makarov, V. Sverdlov, and S. Selberherr. 2012. Emerging Memory Technologies: Trends, Challenges, and Modeling Methods. *Microelectronics Reliability* (Apr. 2012).

[64] J. A. Mandelman, R. H. Dennard, G. B. Bronner, J. K. DeBrosse, R. Divakaruni, Y. Li, and C. J. Radens. 2002. Challenges and Future Directions for the Scaling of Dynamic Random-Access Memory (DRAM). *IBM JRD* (Mar. 2002).

[65] O. Mutlu. 2013. Memory Scaling: A Systems Architecture Perspective. In *IMW*.

[66] S. Naffziger. 1996. A Sub-Nanosecond 0.5µm 64b Adder Design. In *ISSCC*.

[67] C. Nguyen, C. Cagli, E. Vianello, A. Perisco, G. Molas, G. Reimbold, Q. Rafhay, and G. Ghibaudo. 2013. Advanced 1T1R Test Vehicle for RRAM Nanosecond-Range Switching-Time Resolution and Reliability Assessment. In *IISWC*.

[68] NVIDIA Corp. [n. d.]. GeForce RTX 2070. https://www.nvidia.com/en-us/geforce/graphics-cards/rtx-2070/.

[69] NVIDIA Corp. 2018. *NVIDIA Turing GPU Architecture*. White Paper WP-09183-00. https://images.nvidia.com/aem-dam/en-zz/Solutions/design-visualization/technologies/turing-architecture/NVIDIA-Turing-Architecture-Whitepaper.pdf.

[70] NVIDIA Corp. 2021. *CUDA Toolkit Documentation: Profiler*. https://docs.nvidia.com/cuda/profiler-users-guide/.

[71] A. Patel, F. Afram, S. Chen, and K. Ghose. 2011. MARSS: A Full System Simulator for Multicore x86 CPUs. In *DAC*.

[72] I. Richter, K. Pas, X. Guo, R. Patel, J. Liu, E. Ipek, and E. G. Friedman. 2015. Memristive Accelerator for Extreme Scale Linear Solvers. In *GOMACTech*.

[73] P. Rosenfeld, E. C.-Balis, and B. Jacob. 2011. DRAMSim2: A Cycle Accurate Memory System Simulator. *CAL* (Mar. 2011).

[74] V. Seshadri, D. Lee, T. Mullins, H. Hassan, A. Boroumand, J. Kim, M. A. Kozuch, O. Mutlu, P. B. Gibbons, and T. C. Mowry. 2017. Ambit: In-Memory Accelerator for Bulk Bitwise Operations Using Commodity DRAM Technology. In *MICRO*.

[75] V. Seshadri and O. Mutlu. 2017. Simple Operations in Memory to Reduce Data Movement. In *Advances in Computers*. Vol. 106. Elsevier.

[76] A. Shafiee, A. Nag, N. Muralimanohar, R. Balasubramonian, J. P. Strachan, M. Hu, R. S. Williams, and V. Srikumar. 2016. ISAAC: A Convolutional Neural Network Accelerator With In-Situ Analog Arithmetic in Crossbars. In *ISCA*.

[77] W. C. Shen, C. Yu. Mei, Y. D. Chih, S. Sheu, M. Tsai, Y. King, and C. J. Lin. 2012. High-K Metal Gate Contact RRAM (CRRAM) in Pure 28 nm CMOS Logic Process. In *IEDM*.

[78] R. S. Shenoy, G. W. Burr, K. Virwani, B. Jackson, A. Padilla, P. Narayanan, C. T. Rettner, R. M. Shelby, D. S. Bethune, K. V. Raman, M. Brightsky, E. Joseph, P. M. Rice, T. Topuria, A. J. Kellock, B. Kurdi, and K. Gopalakrishnan. 2014. MIEC (Mixed-Ionic-Electronic-Conduction)-Based Access Devices for Non-Volatile Crossbar Memory Arrays. *SST* (Oct. 2014).

[79] S. Sheu, P. Chiang, W. Lin, H. Lee, P. Chen, Y. Chen, T. Wu, F. T. Chen, K. Su, M. Kao, K. Cheng, and M. Tsai. 2009. A 5ns Fast Write Multi-Level Non-Volatile 1K Bits RRAM Memory With Advance Write Scheme. In *VLSIC*.

[80] S. Shirinzadeh, M. Soeken, P.-E. Gaillardon, and R. Drechsler. 2018. Logic Synthesis for RRAM-Based In-Memory Computing. *TCAD* (Jul. 2018).

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

[82] Y.F. Tan, Y.-T Su amd M.-C Chen, T.-C. Chang, T-M. Tsai, Y.-T. Tseng, C.-C. Yang, H.-X. Zheng, W.-C. Chen, and C.-C. Lin. 2019. The Influence of Temperature on Set Voltage for Different High Resistance State in 1T1R Devices. *APEX* (Feb. 2019).

[83] Tech Power Up. [n. d.]. GeForce RTX 2070. https://www.techpowerup.com/gpu-specs/geforce-rtx-2070.c3252.

[84] A. Torrezan, J. P. Strachn, G. Medeiros-Ribeiro, and R. S. Williams. 2011. Sub-Nanosecond Switching Of A Tantalum Oxide Memristor. In *Nanotechnology*.

[85] X. A. Tran, B. Gao, J. F. Kang, L. Wu, Z. R. Wang, Z. Fang, K. L . Pey, Y. C. Yeo, A. Y. Du, B. Y. Nguyen, M. F. Li, and H. Y. Yu and. 2011. High Performance Unipolar AlOy/HfOx/Ni Based RRAM Compatible With Si Diodes for 3D Application. In *VLSIT*.

[86] J. Volder. 1959. The CORDIC Computing Technique. In *WJCC*.

[87] J. Wang, X. Dong, and Y. Xie. 2014. Enabling High-Performance LPDDRx-Compatible MRAM. In *ISLPED*.

[88] H.-S. P. Wong, H.-Y. Lee, S. Yu, Y.-S. Chen, Y. Wu, P.-S. Chen, B. Lee, F. T. Chen, and M.-J. Tsai. 2012. Metal-Oxide RRAM. *Proc. IEEE* (Jun. 2012).

[89] x86 CPUs' Guide. [n. d.]. Intel Xeon Platinum 8153. http://www.x86-guide.net/en/cpu/Intel-Xeon-Platinum-8153-cpu-no6299.html.

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

[91] K. G. Young-Fisher, G. Bersuker, B. Butcher, A. Padovani, L. Larcher, D. Veksler, and D. C. Gilmer. 2013. Leakage Current-Forming Voltage Relation and Oxygen Gettering in HfOx RRAM Devices. *EDL* (May 2013).

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

[93] S. Yu and Y. P. Chen. 2016. Emerging Memory Technologies: Recent Trends and Prospects. *SCC-M* (Spring 2016).

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