SIMDRAM: An End-to-End Framework for Bit-Serial SIMD Computing in DRAM

*Nastaran Hajinazar[1,2]   *Geraldo F. Oliveira[1]   Sven Gregorio[1]   João Dinis Ferreira[1]
Nika Mansouri Ghiasi[1]   Minesh Patel[1]   Mohammed Alser[1]   Saugata Ghose[3]
Juan Gómez-Luna[1]   Onur Mutlu[1]

[1]ETH Zürich   [2]Simon Fraser University   [3]University of Illinois at Urbana–Champaign

Abstract

Processing-using-DRAM has been proposed for a limited set of basic operations (i.e., logic operations, addition). However, in order to enable full adoption of processing-using-DRAM, it is necessary to provide support for more complex operations. In this paper, we propose SIMDRAM, a flexible general-purpose processing-using-DRAM framework that (1) enables the efficient implementation of complex operations, and (2) provides a flexible mechanism to support the implementation of arbitrary user-defined operations. The SIMDRAM framework comprises three key steps. The first step builds an efficient MAJ/NOT representation of a given desired operation. The second step allocates DRAM rows that are reserved for computation to the operation's input and output operands, and generates the required sequence of DRAM commands to perform the MAJ/NOT implementation of the desired operation in DRAM. The third step uses the SIMDRAM *control unit* located inside the memory controller to manage the computation of the operation from start to end, by executing the DRAM commands generated in the second step of the framework. We design the hardware and ISA support for SIMDRAM framework to (1) address key system integration challenges, and (2) allow programmers to employ new SIMDRAM operations without hardware changes.

We evaluate SIMDRAM for reliability, area overhead, throughput, and energy efficiency using a wide range of operations and seven real-world applications to demonstrate SIMDRAM's generality. Our evaluations using a single DRAM bank show that (1) over 16 operations, SIMDRAM provides 2.0× the throughput and 2.6× the energy efficiency of Ambit, a state-of-the-art processing-using-DRAM mechanism; (2) over seven real-world applications, SIMDRAM provides 2.5× the performance of Ambit. Using 16 DRAM banks, SIMDRAM provides (1) 88× and 5.8× the throughput, and 257× and 31× the energy efficiency, of a CPU and a high-end GPU, respectively, over 16 operations; (2) 21× and 2.1× the performance of the CPU and GPU, over seven real-world applications. SIMDRAM incurs an area overhead of only 0.2% in a high-end CPU.

---

*Nastaran Hajinazar and Geraldo F. Oliveira are co-primary authors.

CCS Concepts

• **Computer systems organization → Other architectures**.

Keywords

Bulk Bitwise Operations, Processing-in-Memory, DRAM

1 Introduction

The increasing prevalence and growing size of data in modern applications has led to high energy and latency costs for computation in traditional computer architectures. Moving large amounts of data between memory (e.g., DRAM) and the CPU across bandwidth-limited memory channels can consume more than 60% of the total energy in modern systems [17, 104]. To mitigate such costs, the *processing-in-memory* (PIM) paradigm moves computation closer to where the data resides, reducing (and in some cases eliminating) the need to move data between memory and the processor.

There are two main approaches to PIM [43, 105]: (1) processing-near-memory, where PIM logic is added to the same die as memory or to the logic layer of 3D-stacked memory [3–5, 14, 17–19, 26, 28, 30, 41, 42, 44, 49, 55–57, 63, 66, 70, 82, 87, 96, 109, 113, 114, 119, 122, 126, 127, 152, 153]; and (2) processing-using-memory, which makes use of the operational principles of the memory cells themselves to perform computation by enabling interactions between cells [1, 23, 25, 29, 38, 40, 91, 130–132, 134, 136, 140, 141, 149]. Since processing-using-memory operates directly in the memory arrays, it benefits from the large internal bandwidth and parallelism available inside the memory arrays, which processing-near-memory solutions cannot take advantage of.

A common approach for processing-using-memory architectures is to make use of bulk bitwise computation. Many widely-used data-intensive applications (e.g., databases, neural networks, graph analytics) heavily rely on a broad set of simple (e.g., AND, OR, XOR) and complex (e.g., equality check, multiplication, addition) bitwise operations. Ambit [129, 131], an in-DRAM processing-using-memory accelerator, was the first work to propose exploiting DRAM's analog operational principles to perform bulk bitwise AND, OR, and NOT logic operations. Inspired by Ambit, many prior works have explored DRAM (as well as NVM) designs that are capable of performing in-memory bitwise operations [6, 8–12, 25, 40, 52, 59, 92, 149].

However, a major shortcoming prevents these proposals from becoming widely applicable: they support only basic operations (e.g., Boolean operations, addition) and fall short on flexibly and easily supporting new and more complex operations. Some prior works propose processing-using-DRAM designs that support more complex operations [25, 91]. However, such designs (1) require significant changes to the DRAM subarray, and (2) support only a limited and specific set of operations, lacking the flexibility to support new operations and cater to the wide variety of applications that can potentially benefit from in-memory computation.

Our goal in this paper is to design a framework that aids the adoption of processing-using-DRAM by efficiently implementing complex operations and providing the flexibility to support new desired operations. To this end, we propose SIMDRAM, an end-to-end processing-using-DRAM framework that provides the programming interface, the ISA, and the hardware support for (1) efficiently computing *complex* operations, and (2) providing the ability to implement *arbitrary* operations as required, all in an in-DRAM massively-parallel SIMD substrate. At its core, we build the SIMDRAM framework around a DRAM substrate that enables two previously-proposed techniques: (1) vertical data layout in DRAM, and (2) majority-based logic for computation.

Vertical Data Layout. Supporting bit-shift operations is essential for implementing complex computations, such as addition or multiplication. Prior works show that employing a vertical layout [6, 15, 29, 38, 40, 53, 54, 64, 137, 146] for the data in DRAM, such that all bits of an operand are placed in a single DRAM column (i.e., in a single bitline), eliminates the need for adding extra logic in DRAM to implement shifting [25, 91]. Accordingly, SIMDRAM supports efficient bit-shift operations by storing operands in a vertical fashion in DRAM. This provides SIMDRAM with two key benefits. First, a bit-shift operation can be performed by simply copying a DRAM row into another row (using RowClone [130], LISA [21], NoM [135] or FIGARO [148]). For example, SIMDRAM can perform a left-shift-by-one operation by copying the data in DRAM row *j* to DRAM row *j+1*. (Note that while SIMDRAM supports bit shifting, we can optimize many applications to avoid the need for explicit shift operations, by simply changing the row indices of the SIMDRAM commands that read the shifted data). Second, SIMDRAM enables massive parallelism, wherein each DRAM column operates as a SIMD lane by placing the source and destination operands of an operation on top of each other in the same DRAM column.

Majority-Based Computation. Prior works use majority operations to implement basic logical operations [40, 91, 129, 131] (e.g., AND, OR) or addition [6, 9, 25, 39, 40, 91]. These basic operations are then used as basic building blocks to implement the target in-DRAM computation. SIMDRAM extends the use of the majority operation by directly using only the logically complete set of majority (MAJ) and NOT operations to implement in-DRAM computation. Doing so enables SIMDRAM to achieve higher performance, throughput, and reduced energy consumption compared to using basic logical operations as building blocks for in-DRAM computation. We find that a computation typically requires fewer DRAM commands using MAJ and NOT than using basic logical operations AND, OR, and NOT.

To aid the adoption of processing-using-DRAM by flexibly supporting new and more complex operations, SIMDRAM addresses

two key challenges: (1) how to synthesize new arbitrary in-DRAM operations, and (2) how to exploit an optimized implementation and control flow for such newly-added operations while taking into account key limitations of in-DRAM processing (e.g., DRAM operations that destroy input data, limited number of DRAM rows that are capable of processing-using-DRAM, and the need to avoid costly in-DRAM copies). As a result, SIMDRAM is the first end-to-end framework for processing-using-DRAM. SIMDRAM provides (1) an effective algorithm to generate an efficient MAJ/NOT-based implementation of a given desired operation; (2) an algorithm to appropriately allocate DRAM rows to the operands of the operation and an algorithm to map the computation to an efficient sequence of DRAM commands to execute *any* MAJ-based computation; and (3) the programming interface, ISA support and hardware components required to (i) compute any new user-defined in-DRAM operation without hardware modifications, and (ii) program the memory controller for issuing DRAM commands to the corresponding DRAM rows and correctly performing the computation. Such end-to-end support enables SIMDRAM as a holistic approach that facilitates the adoption of processing-using-DRAM through (1) enabling the flexibility to support new in-DRAM operations by providing the user with a simplified interface to add desired operations, and (2) eliminating the need for adding extra logic to DRAM.

The SIMDRAM framework efficiently supports a wide range of operations of different types. In this work, we demonstrate the functionality of the SIMDRAM framework using an example set of 16 operations including (1) *N*-input logic operations (e.g., AND/OR/XOR of more than 2 input bits); (2) relational operations (e.g., equality/inequality check, greater than, maximum, minimum); (3) arithmetic operations (e.g., addition, subtraction, multiplication, division); (4) predication (e.g., if-then-else); and (5) other complex operations such as bitcount and ReLU [48]. The SIMDRAM framework is not limited to these 16 operations, and can enable processing-using-DRAM for other existing and future operations. SIMDRAM is well-suited to application classes that (i) are SIMD-friendly, (ii) have a regular access pattern, and (iii) are memory bound. Such applications are common in domains such as database analytics, high-performance computing, image processing, and machine learning.

We compare the benefits of SIMDRAM to different state-of-the-art computing platforms (CPU, GPU, and the Ambit [131] in-DRAM computing mechanism). We comprehensively evaluate SIMDRAM's reliability, area overhead, throughput, and energy efficiency. We leverage the SIMDRAM framework to accelerate seven application kernels from machine learning, databases, and image processing (VGG-13 [138], VGG-16 [138], LeNET [79], kNN [88], TPC-H [145], BitWeaving [93], brightness [47]). Using a single DRAM bank, SIMDRAM provides (1) 2.0× the throughput and 2.6× the energy efficiency of Ambit [131], averaged across the 16 implemented operations; and (2) 2.5× the performance of Ambit, averaged across the seven application kernels. Compared to a CPU and a high-end GPU, SIMDRAM using 16 DRAM banks provides (1) 257× and 31× the energy efficiency, and 88× and 5.8× the throughput of the CPU and GPU, respectively, averaged across the 16 operations; and (2) 21× and 2.1× the performance of the CPU and GPU, respectively, averaged across the seven application kernels. SIMDRAM incurs no additional area overhead on top of Ambit, and a total area overhead

of only 0.2% in a high-end CPU. We also evaluate the reliability of SIMDRAM under different degrees of manufacturing process variation, and observe that it guarantees correct operation as the DRAM process technology node scales down to smaller sizes.

We make the following key contributions:

- To our knowledge, this is the first work to propose a framework to enable efficient computation of a flexible set and wide range of operations in a massively parallel SIMD substrate built via processing-using-DRAM.
- SIMDRAM provides a three-step framework to develop efficient and reliable MAJ/NOT-based implementations of a wide range of operations. We design this framework, and add hardware, programming, and ISA support, to (1) address key system integration challenges and (2) allow programmers to define and employ new SIMDRAM operations without hardware changes.
- We provide a detailed reference implementation of SIMDRAM, including required changes to applications, ISA, and hardware.
- We evaluate the reliability of SIMDRAM under different degrees of process variation and observe that it guarantees correct operation as the DRAM technology scales to smaller node sizes.

2 Background

We first briefly explain the architecture of a typical DRAM chip. Next, we describe prior processing-using-DRAM works that SIMDRAM builds on top of (RowClone [130] and Ambit [129, 131, 134]) and explain the implications of majority-based computation.

2.1 DRAM Basics

A DRAM system comprises a hierarchy of components, as Fig. 1 shows, starting with *channels* at the highest level. A channel is subdivided into *ranks*, and a rank is subdivided into multiple *banks* (e.g., 8-16). Each bank is composed of multiple (e.g., 64-128) 2D arrays of cells known as *subarrays*. Cells within a subarray are organized into multiple *rows* (e.g., 512-1024) and multiple *columns* (e.g., 2-8 kB) [67, 83, 84]. A cell consists of an *access transistor* and a *storage capacitor* that encodes a single bit of data using its voltage level. The source nodes of the access transistors of all the cells in the same column connect the cells' storage capacitors to the same *bitline*. Similarly, the gate nodes of the access transistors of all the cells in the same row connect the cells' access transistors to the same *wordline*.
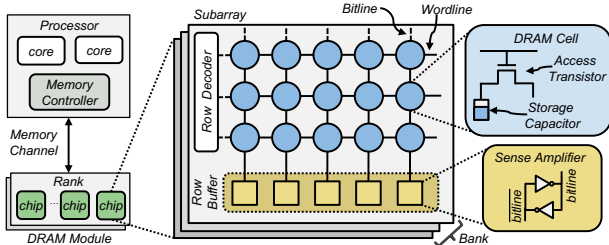

Figure 1: High-level overview of DRAM organization.

When a wordline is asserted, all cells along the wordline are connected to their corresponding bitlines, which perturbs the voltage of each bitline depending on the value stored in each cell's capacitor. A two-terminal *sense amplifier* connected to each bitline senses the voltage difference between the bitline (connected to one terminal) and a reference voltage (typically $\frac{1}{2}V_{DD}$; connected to the other

terminal) and amplifies it to a CMOS-readable value. In doing so, the sense amplifier terminal connected to the reference voltage is amplified to the *opposite* (i.e., *negated*) value, which is shown as the $\overline{\text{bitline}}$ terminal in Fig. 1. The set of sense amplifiers in each subarray forms a logical *row buffer*, which maintains the sensed data for as long as the row is *open* (i.e., the wordline continues to be asserted). A read or write operation in DRAM includes *three* steps:

(1) ACTIVATE. The *wordline* of the target row is asserted, which connects all cells along the row to their respective bitlines. Each bitline shares charge with its corresponding cell capacitor, and the resulting bitline voltage shift is sensed and amplified by the bitline's sense amplifier. Once the sense amplifiers finish amplification, the row buffer contains the values originally stored within the cells along the asserted wordline.

(2) RD/WR. The memory controller then issues read or write commands to columns within the activated row (i.e., the data within the row buffer).

(3) PRECHARGE. The capacitor is disconnected from the bitline by disabling the wordline, and the bitline voltage is restored to its quiescent state (e.g., typically $\frac{1}{2}V_{DD}$).

2.2 Processing-using-DRAM

2.2.1 In-DRAM Row Copy. RowClone [130] is a mechanism that exploits the vast internal DRAM bandwidth to efficiently copy rows inside DRAM without CPU intervention. RowClone enables copying a source row A to a destination row B in the same subarray by issuing two consecutive ACTIVATE commands to these two rows, followed by a PRECHARGE command. This command sequence is called AAP [131]. The first ACTIVATE command copies the contents of the source row A into the row buffer. The second ACTIVATE command connects the cells in the destination row B to the bitlines. Because the sense amplifiers have already sensed and amplified the source data by the time row B is activated, the data (i.e., voltage level) in each cell of row B is overwritten by the data stored in the row buffer (i.e., row A's data). Recent work [40] experimentally demonstrates the feasibility of executing in-DRAM row copy operations in unmodified off-the-shelf DRAM chips.

2.2.2 In-DRAM Bitwise Operations. Ambit [129, 131, 134] shows that simultaneously activating *three* DRAM rows (via a DRAM operation called *Triple Row Activation, TRA*) can be used to perform bitwise Boolean AND, OR, and NOT operations on the values contained within the cells of the three rows. When activating three rows, three cells connected to each bitline share charge simultaneously and contribute to the perturbation of the bitline. Upon sensing the perturbation, the sense amplifier amplifies the bitline voltage to V_{DD} or 0 if at least two of the capacitors of the three DRAM cells are charged or discharged, respectively. As such, a TRA results in a Boolean *majority operation* (*MAJ*) among the three DRAM cells on each bitline. A majority operation MAJ outputs a 1 (0) only if more than half of its inputs are 1 (0). In terms of AND (·) and OR (+) operations, a 3-input majority operation can be expressed as MAJ(A, B, C) = A · B + A · C + B · C.

Ambit implements MAJ by introducing a custom row decoder (discussed in §3.1) that can perform a TRA by simultaneously addressing three wordlines. To use this decoder, Ambit defines a new command sequence called AP, which issues (1) a TRA to compute the MAJ of three rows, followed by (2) a PRECHARGE to close all three

rows.[1] Ambit uses AP command sequences to implement Boolean AND and OR operations by simply setting one of the inputs (e.g., C) to 1 or 0. The AND operation is computed by setting C to 0 (i.e., `MAJ(A, B, 0) = A AND B`). The OR operation is computed by setting C to 1 (i.e., `MAJ(A, B, 1) = A OR B`).

To achieve functional completeness alongside AND and OR operations, Ambit implements NOT operations by exploiting the differential design of DRAM sense amplifiers. As §2.1 explains, the sense amplifier already generates the complement of the sensed value as part of the activation process ($\overline{bitline}$ in Fig. 1). Therefore, Ambit simply forwards $\overline{bitline}$ to a special DRAM row in the subarray that consists of DRAM cells with *two* access transistors, called *dual-contact cells* (DCCs). Each access transistor is connected to one side of the sense amplifier and is controlled by a separate wordline (*d-wordline* or *n-wordline*). By activating either the d-wordline or the n-wordline, the row of DCCs can provide the true or negated value stored in the row's cells, respectively.

2.2.3 Majority-Based Computation. Activating multiple rows simultaneously reduces the reliability of the value read by the sense amplifiers due to manufacturing process variation, which introduces non-uniformities in circuit-level electrical characteristics (e.g., variation in cell capacitance levels) [131]. This effect worsens with (1) an increased number of simultaneously activated rows, and (2) more advanced technology nodes with smaller sizes. Accordingly, although processing-using-DRAM can potentially support majority operations with more than three inputs (as proposed by prior works [6, 9, 92]) our realization of processing-using-DRAM uses the minimum number of inputs required for a majority operation (N=3) to maintain the reliability of the computation. In §7.5, we demonstrate via SPICE simulations that using 3-input MAJ operations provides higher reliability compared to designs with more than three inputs per MAJ operation. Using 3-input MAJ, a processing-using-DRAM substrate does not require modifications to the subarray organization (Fig. 2) beyond the ones proposed by Ambit (§3.1). Recent work [40] experimentally demonstrates the feasibility of executing MAJ operations by activating three rows in unmodified off-the-shelf DRAM chips.

3 SIMDRAM Overview

SIMDRAM is a processing-using-DRAM framework whose goal is to (1) enable the efficient implementation of complex operations and (2) provide a flexible mechanism to support the implementation of arbitrary user-defined operations. We present the subarray organization in SIMDRAM, describe an overview of the SIMDRAM framework, and explain how to integrate SIMDRAM into a system.

3.1 Subarray Organization

In order to perform processing-using-DRAM, SIMDRAM makes use of a subarray organization that incorporates additional functionality to perform logic primitives (i.e., MAJ and NOT). This subarray organization is *identical* to Ambit's [131] and is similar to DRISA's [91]. Fig. 2 illustrates the internal organization of a subarray in SIMDRAM, which resembles a conventional DRAM subarray.

---

[1]Although the 'A' in AP refers to a TRA operation instead of a conventional `ACTIVATE` command, we use this terminology to remain consistent with the Ambit paper [131], since an `ACTIVATE` command can be internally translated to a TRA operation by the DRAM chip [131].

SIMDRAM requires only minimal modifications to the DRAM subarray (namely, a small row decoder that can activate three rows simultaneously) to enable computation. Like Ambit [131], SIMDRAM divides DRAM rows into *three groups*: the **D**ata group (D-group), the **C**ontrol group (C-group) and the **B**itwise group (B-group).
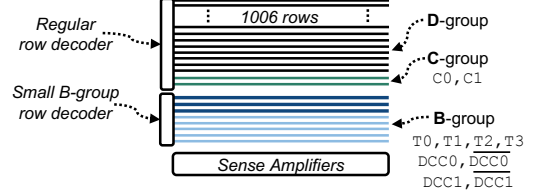
Figure 2: SIMDRAM subarray organization [131].

The D-group contains regular rows that store program or system data. The C-group consists of two constant rows, called C0 and C1, that contain all-0 and all-1 values, respectively. These rows are used (1) as initial input values for a given SIMDRAM operation (e.g., the initial carry-in bit in a full addition), or (2) to perform operations that naturally require AND/OR operations (e.g., AND/OR reductions). The D-group and the C-group are connected to the regular row decoder, which selects a single row at a time.

The B-group contains six regular rows, called T0, T1, T2, and T3; and two rows of dual-contact cells (see §2.2.2), whose d-wordlines are called DCC0 and DCC1, and whose n-wordlines are called $\overline{DCC0}$ and $\overline{DCC1}$, respectively. The B-group rows, called *compute rows*, are designated to perform bitwise operations. They are all connected to a special row decoder that can simultaneously activate three rows using a single address (i.e., perform a TRA)

Using a typical subarray size of 1024 rows [20, 67, 69, 74, 85], SIMDRAM splits the row addressing into 1006 D-group rows, 2 C-group rows, and 16 B-group rows.

3.2 Framework Overview

SIMDRAM is an end-to-end framework that provides the user with the ability to implement an *arbitrary* operation in DRAM using the AAP/AP command sequences. The framework comprises three key steps, which are illustrated in Fig. 3. The first two steps of the framework give the user the ability to efficiently implement any desired operation in DRAM, while the third step controls the execution flow of the in-DRAM computation transparently from the user. We briefly describe these steps below, and discuss each step in detail in §4.

The first step (❶ in Fig. 3a; §4.1) builds an efficient MAJ/NOT representation of a given desired operation from its AND/OR/NOT-based implementation. Specifically, this step takes as input a desired operation and uses logic optimization to minimize the number of logic primitives (and, therefore, the computation latency) required to perform the operation. Accordingly, for a desired operation input into the SIMDRAM framework by the user, the first step derives its *optimized* MAJ/NOT-based implementation.

The second step (❷ in Fig. 3a; §4.2) allocates DRAM rows to the operation's inputs and outputs and generates the required sequence of DRAM commands to execute the desired operation. Specifically, this step translates the MAJ/NOT-based implementation of the operation into AAPs/APs. This step involves (1) allocating the designated compute rows in DRAM to the operands, and (2) determining the optimized sequence of AAPs/APs that are required to perform the
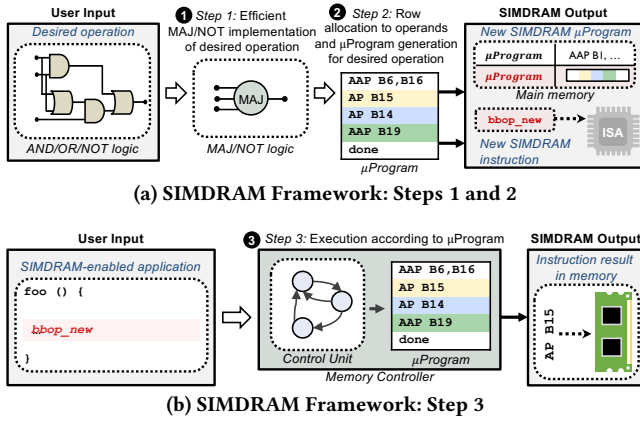
(a) SIMDRAM Framework: Steps 1 and 2

(b) SIMDRAM Framework: Step 3

Figure 3: Overview of the SIMDRAM framework.

operation. While doing so, SIMDRAM minimizes the number of AAPs/APs required for a specific operation. This step's output is a μProgram, i.e., the optimized sequence of AAPs/APs that is stored in main memory and will be used to execute the operation at runtime.

The third step (❸ in Fig. 3b; §4.3) executes the μProgram to perform the operation. Specifically, when a user program encounters a *bbop* instruction (§5.2) associated with a SIMDRAM operation, the *bbop* instruction triggers the execution of the SIMDRAM operation by performing its μProgram in the memory controller. SIMDRAM uses a *control unit* in the memory controller that transparently issues the sequence of AAPs/APs to DRAM, as dictated by the μProgram. Once the μProgram is complete, the result of the operation is held in DRAM.

3.3 Integrating SIMDRAM in a System

As we discuss in §1, SIMDRAM operates on data using a vertical layout. Fig. 4 illustrates how data is organized within a DRAM subarray when employing a horizontal data layout (Fig. 4a) and a vertical data layout (Fig. 4b). We assume that each data element is four bits wide, and that there are four data elements (each one represented by a different color). In a conventional horizontal data layout, data elements are stored in different DRAM rows, with the contents of each data element ordered from the most significant bit to the least significant bit (or vice versa) in a single row. In contrast, in a vertical data layout, the DRAM row holds only the *i*-th bit of *multiple* data elements (where the number of elements is determined by the bit width of the row). Therefore, when activating a single DRAM row in a vertical data layout organization, a *single* bit of data from each data element is read at once, which enables in-DRAM bit-serial parallel computation [6, 15, 49, 91, 134, 137].
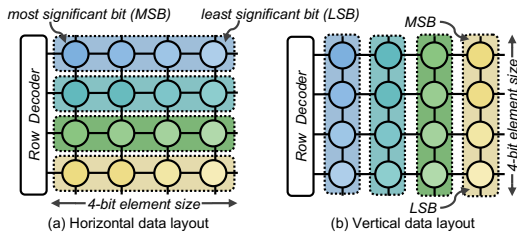

Figure 4: Data layout: horizontal vs. vertical.

To maintain compatibility with traditional system software, we store regular data in the conventional horizontal layout and provide

hardware support (explained in §5.1) to transpose horizontally-laid-out data into the vertical layout for in-DRAM computation. To simplify program integration, we provide ISA extensions that expose SIMDRAM operations to the programmer (§5.2).

4 SIMDRAM Framework

We describe the three steps of the SIMDRAM framework introduced in §3.2, using the full addition operation as a running example.

4.1 Step 1: Efficient MAJ/NOT Implementation

SIMDRAM implements in-DRAM computation using the logically-complete set of MAJ and NOT logic primitives, which requires fewer AAP/AP command sequences to perform a given operation when compared to using AND/OR/NOT. As a result, the goal of the first step in the SIMDRAM framework is to build an optimized MAJ/NOT implementation of a given operation that executes the operation using as few AAP/AP command sequences as possible, thus minimizing the operation's latency. To this end, Step 1 transforms an AND/OR/NOT representation of a given operation to an optimized MAJ/NOT representation using a transformation process formalized by prior work [7].

The transformation process uses a graph-based representation of the logic primitives, called an *AND–OR–Inverter Graph* (AOIG) for AND/OR/NOT logic, and a *Majority–Inverter Graph* (MIG) for MAJ/NOT logic. An AOIG is a logic representation structure in the form of a directed acyclic graph where each node represents an AND or OR logic primitive. Each edge in an AOIG represents an input/output dependency between nodes. The incoming edges to a node represent input operands of the node and the outgoing edge of a node represents the output of the node. The edges in an AOIG can be either regular or complemented (which represents an inverted input operand; denoted by a bubble on the edge). The direction of the edges follows the natural direction of computation from inputs to outputs. Similarly, a MIG is a directed acyclic graph in which each node represents a three-input MAJ logic primitive, and each regular/complemented edge represents one input or output to the MAJ primitive that the node represents. The transformation process consists of two parts that operate on an input AOIG.

The first part of the transformation process naively substitutes AND/OR primitives with MAJ primitives. Each two-input AND or OR primitive is simply replaced with a three-input MAJ primitive, where one of the inputs is tied to logic 0 or logic 1, respectively. This naive substitution yields a MIG that *correctly* replicates the functionality of the input AOIG, but the MIG is likely *inefficient*.

The second part of the transformation process takes the inefficient MIG and uses a greedy algorithm (see Appendix A) to apply a series of transformations that identifies how to consolidate multiple MAJ primitives into a smaller number of MAJ primitives with identical functionality. This yields a smaller MIG, which in turn requires fewer logic primitives to perform the same operation that the unoptimized MIG (and, thus, the input AOIG) performs. Fig. 5a shows the optimized MIG produced by the transformation process for a full addition operation.

4.2 Step 2: μProgram Generation

Each SIMDRAM operation is stored as a *μProgram*, which consists of a series of microarchitectural operations (μOps) that SIMDRAM uses to execute the SIMDRAM operation in DRAM. The
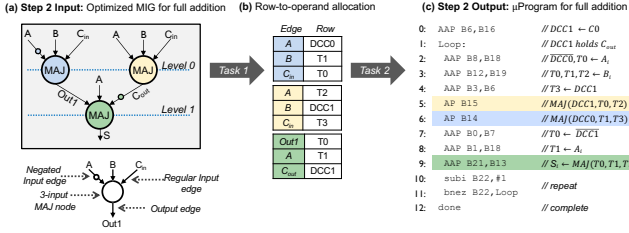
Figure 5: (a) Optimized MIG; (b) row-to-operand allocation; (c) μProgram for full addition.

goal of the second step is to take the optimized MIG generated in Step 1 and generate a μProgram that executes the SIMDRAM operation that the MIG represents. To this end, as shown in Fig. 5, the second step of the framework performs two key tasks on the optimized MIG: (1) *allocating DRAM rows to the operands*, which assigns each input operand (i.e., an incoming edge) of each MAJ node in the MIG to a DRAM row (Fig. 5b); and (2) *generating the μProgram*, which creates the series of μOps that perform the MAJ and NOT logic primitives (i.e., nodes) in the MIG, while maintaining the correct flow of the computation (Fig. 5c). In this section, we first describe the μOps used in SIMDRAM (§4.2.1). Second, we explain the process of allocating DRAM rows to the input operands of the MAJ nodes in the MIG to DRAM rows (§4.2.2). Third, we explain the process of generating the μProgram (§4.2.3).

4.2.1 SIMDRAM μOps. Fig. 6a shows the set of μOps that we implement in SIMDRAM. Each μOp is either (1) a *command sequence* that is issued by SIMDRAM to a subarray to perform a portion of the in-DRAM computation, or (2) a *control* operation that is used by the SIMDRAM control unit (see §4.3) to manage the execution of the SIMDRAM operation. We further break down the command sequence μOps into one of three types: (1) *row copy*, a μOp that performs in-DRAM copy from a source memory address to a destination memory address using an AAP command sequence; (2) *majority*, a μOp that performs a majority logic primitive on three DRAM rows using an AP command sequence (i.e., it performs a TRA); and (3) *arithmetic*, four μOps that perform simple arithmetic operations on SIMDRAM control unit registers required to control the execution of the operation (addi, subi, comp, module). We provide two control operation μOps to support loops and termination in the SIMDRAM control flow (bnez, done).
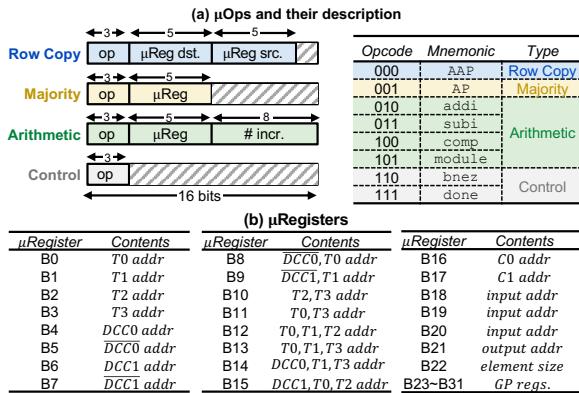

Figure 6: μOps and μRegisters in SIMDRAM.

During μProgram generation, the SIMDRAM framework converts the MIG into a series of μOps. Note that MIG represents a 1-bit-wide computation of an operation. Thus, to implement a multi-bit-wide SIMDRAM operation, the framework needs to repeat the series of the μOps that implement the MIG n times, where n is the number of bits in the operands of the SIMDRAM operation. To this end, SIMDRAM uses the arithmetic and control μOps to repeat the 1-bit-wide computation n times, transparently to the programmer.

To support the execution of μOps, SIMDRAM utilizes a set of *μRegisters* (Fig. 6b) located in the SIMDRAM control unit (§4.3). The framework uses μRegisters (1) to store the memory addresses of DRAM rows in the B-group and C-group (Fig. 3.1) of the subarray (μRegisters B0–B17), (2) to store the memory addresses of input and output rows for the computation (μRegisters B18–B22), and (3) as general-purpose registers during the execution of arithmetic and control operations (μRegisters B23–B31).

4.2.2 Task 1: Allocating DRAM Rows to the Operands. The goal of this task is to allocate DRAM rows to the input operands (i.e., incoming edges) of each MAJ node in the operation's MIG, such that we minimize the total number of μOps needed to compute the operation. To this end, we present a new allocation algorithm inspired by the linear scan register allocation algorithm [121]. However, unlike register allocation algorithms, our allocation algorithm considers two extra constraints that are specific to processing-using-DRAM: (1) performing MAJ in DRAM has destructive behavior, i.e., a TRA overwrites the original values of the three input rows with the MAJ output; and (2) the number of compute rows (i.e., B-group in Fig. 2) that are designated to perform bitwise operations is limited (there are only six compute rows in each subarray, as discussed in §3.1).

The SIMDRAM *row-to-operand allocation algorithm* receives the operation's MIG as input. The algorithm assumes that the input operands of the operation are already stored in separate rows of the D-group in the subarray using vertical layout (§3.3), before the computation of the operation starts. The algorithm then does a topological traversal starting with the leftmost MAJ node at the highest level of the MIG (e.g., level 0 in Fig. 5a), allocating compute rows to the input operands of each MAJ node in the current level of the MIG, before moving to the next lower level of the graph. The algorithm finishes once DRAM rows are allocated to all the input operands of all the MAJ nodes in the MIG. Fig. 5b shows these allocations as the output of Task 1 for the full addition example. The resulting row-to-operand allocation is then used in the second task in step two (§4.2.3) to generate the series of μOps to compute the operation that the MIG represents. We describe our row-to-operand allocation algorithm in Appendix B.

4.2.3 Task 2: Generating a μProgram. The goal of this task is to use the MIG and the DRAM row allocations from Task 1 to generate the μOps of the μProgram for our SIMDRAM operation. To this end, Task 2 (1) translates the MIG into a series of row copy and majority μOps (i.e., AAPs/APs), (2) optimizes the series of μOps to reduce the number of AAPs/APs, and (3) generalizes the one-bit bit-serial operation described by the MIG into an n-bit operation by utilizing SIMDRAM's arithmetic and control μOps.

Translating the MIG into a Series of Row Copy and Majority μOps. The allocation produced during Task 1 dictates how DRAM rows are allocated to each edge in the MIG during the

μProgram. With this information, the framework can generate the appropriate series of row copies and majority μOps to reflect the MIG's computation in DRAM. To do so, we traverse the input MIG in topological order. For *each* node, we *first* assign row copy μOps (using the AAP command sequence) to the node's edges. Then, we assign a majority μOp (using the AP command sequence) to execute the current MAJ node, following the DRAM row allocation assigned to each edge of the node. The framework repeats this procedure for all the nodes in the MIG. To illustrate, we assume that the SIM-DRAM allocation algorithm allocates DRAM rows DCC0, T1, and T0 to edges A, B, and C_{in}, respectively, of the blue node in the full addition MIG (Fig. 5a). Then, when visiting this node, we generate the following series of μOps:

```
AAP DCC0, A;        // DCC0 ← A
AAP T1, B;          // T1 ← B
AAP T0, C_in;       // T0 ← C_in
AP  DCC0, T1, T0    // MAJ(NOT(A), B, C_in)
```

Optimizing the Series of μOps. After traversing all of the nodes in the MIG and generating the appropriate series of μOps, we optimize the series of μOps by coalescing AAP/AP command sequences, which we can do in one of two cases.

Case 1: we can coalesce a series of row copy μOps if all of the μOps have the same μRegister source as an input. For example, consider a series of two AAPs that copy data array A into rows T2 and T3. We can coalesce this series of AAPs into a single AAP issued to the wordline address stored in μRegister B10 (see Fig. 6a). This wordline address leverages the special row decoder in the B-group (which is part of the Ambit subarray structure [131]) to activate multiple DRAM rows in the group at once with a single activation command. For our example, activating μRegister B10 allows the AAP command sequence to copy array A into both rows T2 and T3 at once.

Case 2: we can coalesce an AP command sequence (i.e., a majority μOp) followed by an AAP command sequence (i.e., a row copy μOp) when the destination of the AAP is one of the rows used by the AP. For example, consider an AP that performs a MAJ logic primitive on DRAM rows T0, T1, and T2 (storing the result in all three rows), followed by an AAP that copies μRegister B12 (which refers to rows T0, T1, and T2) to row T3. The AP followed by the AAP puts the majority value in all four rows (T0, T1, T2, T3). The two command sequences can be coalesced into a single AAP (AAP T3, B12), as the first ACTIVATE would automatically perform the majority on rows T0, T1, and T2 by activating all three rows simultaneously. The second ACTIVATE then copies the value from those rows into T3.

Generalizing the Bit-Serial Operation into an n-bit Operation. Once all potential μOp coalescing is complete, the framework now has an optimized 1-bit version of the computation. We generalize this 1-bit μOp series into a loop body that repeats n times to implement an n-bit operation. We leverage the arithmetic and control μOps available in SIMDRAM to orchestrate the n-bit computation. Data produced by the computation of one bit that needs to be used for computation of the next bit (e.g., the carry bit in full addition) is kept in a B-group row across the two computations, allowing for bit-to-bit data transfer without the need for dedicated shifting circuitry.

The final series of μOps produced after this step is then packed into a μProgram and stored in DRAM for future use.[2] Fig. 5c shows the final μProgram produced at the end of Step 2 for the full addition operation. The figure shows the optimized series of μOps that generates the 1-bit implementation of the full addition (lines 2–9), and the arithmetic and control μOps included to enable the n-bit implementation of the operation (lines 10–11).

Benefits of the μProgram Abstraction. The μProgram abstraction that we use to store SIMDRAM operations provides three main advantages to the framework. First, it allows SIMDRAM to minimize the total number of new CPU instructions required to implement SIMDRAM operations, thereby reducing SIMDRAM's impact on the ISA. While a different implementation could use more new CPU instructions to express finer-grained operations (e.g., an AAP), we believe that using a minimal set of new CPU instructions simplifies adoption and software design. Second, the μProgram abstraction enables a smaller application binary size since the only information that needs to be placed in the application's binary is the address of the μProgram in main memory. Third, the μProgram provides an abstraction to relieve the end user from low-level programming with MAJ/NOT operations that is equivalent to programming with Boolean logic. We discuss how a user program invokes SIMDRAM μPrograms in §5.2.

4.3 Step 3: Operation Execution

Once the framework stores the generated μProgram for a SIM-DRAM operation in DRAM, the SIMDRAM hardware can now receive program requests to execute the operation. To this end, we discuss the SIMDRAM *control unit*, which handles the execution of the μProgram at runtime. The control unit is designed as an extension of the memory controller, and is transparent to the programmer. A program issues a request to perform a SIMDRAM operation using a *bbop* instruction (introduced by Ambit [131]), which is one of the CPU ISA extensions to allow programs to interact with the SIMDRAM framework (see §5.2). Each SIMDRAM operation corresponds to a different *bbop* instruction. Upon receiving the request, the control unit loads the μProgram corresponding to the requested *bbop* from memory, and performs the μOps in the μProgram. Since all input data elements of a SIMDRAM operation may not fit in one DRAM row, the control unit repeats the μProgram i times, where i is the total number of data elements divided by the number of elements in a single DRAM row.

Fig. 7 shows a block diagram of the SIMDRAM control unit, which consists of nine main components: (1) a *bbop FIFO* that receives the *bbop*s from the program, (2) a *μProgram Memory* allocated in DRAM (not shown in the figure), (3) a *μProgram Scratchpad* that holds commonly-used μPrograms, (4) a *μOp Memory* that holds the μOps of the currently running μProgram, (5) a *μRegister Addressing Unit* that generates the physical row addresses being used by the μRegisters that map to DRAM rows (based on the μRegister-to-row assignments for B0–B17 in Fig. 6), (6) a *μRegister File* that holds the non-row-mapped μRegisters (B18–B31 in Fig. 6), (7) a *Loop Counter* that tracks the number of remaining data elements that the

---

[2]In our example implementation of SIMDRAM, a μProgram has a maximum size of 128 bytes, as this is enough to store the largest μProgram generated in our evaluations (the division operation, which requires 56 μOps, each two bytes wide, resulting in a total μProgram size of 112 bytes.)

μProgram needs to be performed on, (8) a μOp Processing FSM that controls the execution flow and issues AAP/AP command sequences, and (9) a μProgram counter (μPC). SIMDRAM reserves a region of DRAM for the μProgram Memory to store μPrograms corresponding to *all* SIMDRAM operations. At runtime, the control unit stores the most commonly used μPrograms in the μProgram Scratchpad, to reduce the overhead of fetching μPrograms from DRAM.

Figure 7: SIMDRAM control unit.

At runtime, when a CPU running a user program reaches a *bbop* instruction, it forwards the *bbop* to the SIMDRAM control unit (❶ in Fig. 7). The control unit enqueues the *bbop* in the *bbop* FIFO. The control unit goes through a four-stage procedure to execute the queued *bbop*s one at a time.

In the first stage, the control unit fetches and decodes the *bbop* at the head of the FIFO (❷). Decoding a *bbop* involves (1) setting the index of the μProgram Scratchpad to the *bbop* opcode; (2) writing the number of loop iterations required to perform the operation on all elements (i.e., the number of data elements divided by the number of elements in a single DRAM row) into the Loop Counter; and (3) writing the base DRAM addresses of the source and destination arrays involved in the computation, and the size of each data element, to the μRegister Addressing Unit.

In the second stage, the control unit copies the μProgram currently indexed in the μProgram Scratchpad to the μOp Memory (❸). At this point, the control unit is ready to start executing the μProgram, one μOp at a time.

In the third stage, the current μOp is fetched from the μOp Memory, which is indexed by the μPC. The μOp Processing FSM decodes the μOp, and determines which μRegisters are needed (❹). For μRegisters B0–B17, the μRegister Addressing Unit generates the DRAM addresses that correspond to the requested registers (see Fig. 6) and sends the addresses to the μOp Processing FSM. For μRegisters B18–B31, the μRegister File provides the register values to the μOp Processing FSM.

In the fourth stage, the μOp Processing FSM executes the μOp. If the μOp is a command sequence, the corresponding commands are sent to the memory controller's request queue (❺) and the μPC is incremented. If the μOp is a done control operation, this indicates that all of the command sequence μOps have been performed for the current iteration. The μOp Processing FSM then decrements the Loop Counter (❻). If the decremented Loop Counter is greater than zero, the μOp Processing FSM shifts the base source and destination addresses stored in the μRegister Addressing Unit to move onto the next set of data elements,[3] and resets the μPC to the first μOp in the μOp Memory. If the decremented Loop Counter equals zero, this

indicates that the control unit has completed executing the current *bbop*. The control unit then fetches the next *bbop* from the *bbop* FIFO (❼), and repeats all four stages for the next *bbop*.

4.4 Supported Operations

We use our framework to efficiently support a wide range of operations of different types. In this paper, we evaluate (in §7) a set of 16 SIMDRAM operations of five different types for *n*-bit data elements: (1) *N*-input logic operations (OR-/AND-/XOR-reduction across *N* inputs); (2) relational operations (equality/inequality check, greater-/less-than check, greater-than-or-equal-to check, and maximum/minimum element in a set); (3) arithmetic operations (addition, subtraction, multiplication, division, and absolute value); (4) predication (if-then-else); and (5) other complex operations (bitcount, and ReLU). We support four different element sizes that correspond to data type sizes in popular programming languages (8-bit, 16-bit, 32-bit, 64-bit).

5 System Integration of SIMDRAM

We discuss several challenges of integrating SIMDRAM in a real system, and how we address them: (1) data layout and how SIMDRAM manages storing the data required for in-DRAM computation in a vertical layout (§5.1); (2) ISA extensions for and programming interface of SIMDRAM (§5.2); (3) how SIMDRAM handles page faults, address translation, coherence, and interrupts (§5.3); (4) how SIMDRAM manages computation on large amounts of data (§5.4); (5) security implications of SIMDRAM (§5.5); and (6) current limitations of the SIMDRAM framework (§5.6).

5.1 Data Layout

We envision SIMDRAM as *supplementing* (not *replacing*) the traditional processing elements. As a result, a program in a SIMDRAM-enabled system can have a combination of CPU instructions and SIMDRAM instructions, with possible data sharing between the two. However, while SIMDRAM operates on vertically-laid-out data (§3.3), the other system components (including the CPU) expect the data to be laid out in the traditional horizontal format, making it challenging to share data between SIMDRAM and CPU instructions. To address this challenge, memory management in SIMDRAM needs to (1) support both horizontal and vertical data layouts in DRAM *simultaneously*; and (2) transform vertically-laid-out data used by SIMDRAM to a horizontal layout for CPU use, and vice versa. We cannot rely on software (e.g., compiler or application support) to handle the data layout transformation, as this would go through the on-chip memory controller, and would introduce significant data movement, and thus latency, between the DRAM and CPU during the transformation. To avoid data movement during transformation, SIMDRAM uses a specialized hardware unit placed between the last-level cache (LLC) and the memory controller, called the *data transposition unit*, to transform data from horizontal data layout to vertical data layout, and vice versa. The transposition unit ensures that for every SIMDRAM object, its corresponding data is in a horizontal layout whenever the data is in the cache, and in a vertical layout whenever the data is in DRAM.

Fig. 8 shows the key components of the transposition unit. The transposition unit keeps track of the memory objects that are used by SIMDRAM operations in a small cache in the transposition unit, called the *Object Tracker*. To add an entry to the Object Tracker

---

[3]The source and destination base addresses are incremented by *n* rows, where *n* is the data element size. This is because each DRAM row contains one bit of a set of elements, so SIMDRAM uses *n* consecutive rows to hold all *n* bits of the set of elements.

when allocating a memory object used by SIMDRAM, the programmer adds an initialization instruction called bbop_trsp_init (§5.2) *immediately* after the malloc that allocates the memory object (❶ in Fig. 8). Assuming a system that employs lazy allocation, the bbop_trsp_init instruction informs the operating system (OS) that the memory object is a *SIMDRAM object*. This allows the OS to perform virtual-to-physical memory mapping optimizations for the object before the allocation starts (e.g., mapping the arguments of an operation to the same row/column in the physical memory). When the SIMDRAM object's physical memory is allocated, the OS inserts the base physical address, the total size of the allocated data, and the size of each element in the object (provided by bbop_trsp_init) into the Object Tracker. As the initially-allocated data is placed in the CPU cache, the data starts in a horizontal layout until it is evicted from the cache.
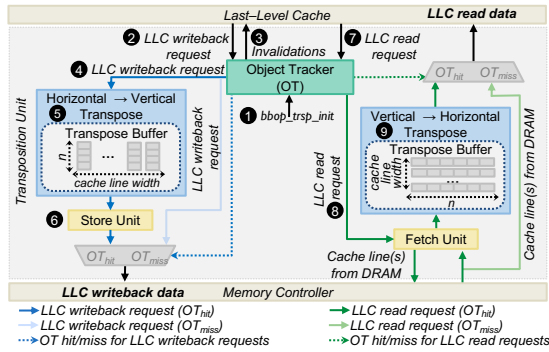

Figure 8: Major components of the data transposition unit.

SIMDRAM stores SIMDRAM objects in DRAM using a vertical layout, since this is the layout used for in-DRAM computation (§3.3). Since a vertically-laid-out n-bit element spans n different cache lines in DRAM (with each cache line in a different DRAM row), SIMDRAM partitions SIMDRAM objects into *SIMDRAM object slices*, each of which is n cache lines in size. Thus, a SIMDRAM object slice in DRAM contains the vertically-laid-out bits of as many elements as bits in a cache line (e.g., 512 in a 64 B cache line). Cache line i ($0 \leq i < n$) of an object slice contains bit i of *all* elements stored in the slice. Whenever any one data element within a slice is requested by the CPU, the entire SIMDRAM object slice is brought into the LLC. Similarly, whenever a cache line from a SIMDRAM object is written back from the LLC to DRAM (i.e., it is evicted or flushed), *all* $n - 1$ remaining cache lines of the same SIMDRAM object slice are written back as well.[4] The use of object slices ensures correctness and simplifies the transposition unit.

Whenever the LLC writes back a cache line to DRAM (❷ in Fig. 8), the transposition unit checks the Object Tracker to see whether the cache line belongs to a SIMDRAM object. If the LLC request misses in the Object Tracker, the cache line does not belong to any SIMDRAM object, and the writeback request is forwarded to the memory controller as in a conventional system. If the LLC request hits in the Object Tracker, the cache line belongs to a SIMDRAM object, and thus must be transposed from the horizontal layout to the vertical layout. An Object Tracker hit triggers two actions.

First, the Object Tracker issues invalidation requests to *all* $n - 1$ remaining cache lines of the same SIMDRAM object slice (❸ in

Fig. 8).[4] We extend the LLC to support a special invalidation request type, which sends both dirty *and* unmodified cache lines to the transposition unit (unlike a regular invalidation request, which simply invalidates unmodified cache lines). The Object Tracker issues these invalidation requests for the remaining cache lines, ensuring that *all* cache lines of the object slice arrive at the transposition unit to perform the horizontal-to-vertical transposition correctly.

Second, the writeback request is forwarded (❹ in Fig. 8) to a horizontal-to-vertical transpose buffer, which performs the bit-by-bit transposition. We design the transpose buffer (❺) such that it can transpose all bits of a horizontally-laid-out cache line in a single cycle. As the other cache lines belonging from the slice are evicted (as a result of the Object Tracker's invalidation requests) and arrive at the transposition unit, they too are forwarded to the transpose buffer, and their bits are transposed. Each horizontally-laid-out cache line maps to a specific set of bit columns in the vertically-laid-out cache line, which is determined using the physical address of the horizontally-laid-out cache line. Once all n cache lines in the SIMDRAM object slice have been transposed, the Store Unit generates DRAM write requests for each vertically-laid-out cache line, and sends the requests to the memory controller (❻).

When a program wants to read data that belongs to a SIMDRAM object, and the data is not in the CPU caches, the LLC issues a read request to DRAM (❼ in Fig. 8). If the address of the read request does not hit in the Object Tracker, the request is forwarded to the memory controller, as in a conventional system. If the address of the read request hits in the Object Tracker, the read request is part of a SIMDRAM object, and the Object Tracker sends a signal (❽) to the Fetch Unit. The Fetch Unit generates the read requests for *all* of the vertically-laid-out cache lines that belong to the same SIMDRAM object slice as the requested data, and sends these requests to the memory controller. When the request responses for the object slice's cache lines arrive, the Fetch Unit sends the cache lines to a vertical-to-horizontal transpose buffer (❾), which can transpose all bits of one vertically-laid-out cache line into the horizontally-laid-out cache lines in one cycle. The horizontally-laid-out cache lines are then inserted into the LLC. The $n - 1$ cache lines that were *not* part of the original memory request, but belong to the same object slice, are inserted into the LLC in a manner similar to conventional prefetch requests [142].

5.2 ISA Extensions and Programming Interface

The lack of an efficient and expressive programmer/system interface can negatively impact the performance and usability of the SIMDRAM framework. This would put data transposition on the critical path of SIMDRAM computation, which would cause large performance overheads. To address such issues and to enable the programmer/system to efficiently communicate with SIMDRAM, we extend the ISA with specialized SIMDRAM instructions. The main goal of the SIMDRAM ISA extensions is to let the SIMDRAM control unit know (1) what SIMDRAM operations need to be performed and when, and (2) what the SIMDRAM memory objects are and when to transpose them.

Table 1 shows the CPU ISA extensions that the SIMDRAM framework exposes to the programmer. There are three types of instructions: (1) SIMDRAM object initialization instructions, (2) instructions to perform different SIMDRAM operations, and (3) predication

---

[4]The Dirty-Block Index [128] could be adapted for this purpose.

instructions. We discuss bbop_trsp_init, our only SIMDRAM object initialization instruction, in §5.1. The CPU ISA extensions for performing SIMDRAM operations can be further divided into two categories: (1) operations with one input operand (e.g., bitcount, ReLU), and (2) operations with two input operands (e.g., addition, division, equal, maximum). SIMDRAM uses an array-based computation model, and src (i.e., src in 1-input operations and src_1, src_2 in 2-input operations) and dst in these instructions represent source and destination *arrays*. bbop_op represents the opcode of the SIMDRAM operation, while size and n represent the number of elements in the source and destination arrays, and the number of bits in each array element, respectively. To enable predication, SIMDRAM uses the bbop_if_else instruction in which, in addition to two source and one destination arrays, select represents the predicate array (i.e., the predicate, or mask, bits).

Table 1: SIMDRAM ISA extensions.

Type	ISA Format
Initialization	bbop_trsp_init address, size, n
1-Input Operation	bbop_op dst, src, size, n
2-Input Operation	bbop_op dst, src_1, src_2, size, n
Predication	bbop_if_else dst, src_1, src_2, select, size, n

Listing 1 shows how SIMDRAM's CPU ISA extensions can be used to perform in-DRAM computation, with an example code that performs element-wise addition or subtraction of two arrays (A and B) depending on the comparison of each element of A to the corresponding element of a third array (pred). Listing 1a shows the original C code for the computation, while Listing 1b shows the equivalent code using SIMDRAM operations. The lines that perform the same operations are highlighted using the same colors in both C code and SIMDRAM code. The if-then-else condition in C code is performed in SIMDRAM using a predication instruction (i.e., bbop_if_else on line 16 in Listing 1b). SIMDRAM treats the if-then-else condition as a multiplexer. Accordingly, bbop_if_else takes two source arrays and a predicate array as inputs, where the predicate is used to choose which source array should be selected as the output at the corresponding index. To this end, we first allocate two arrays to hold the addition and subtraction results (i.e., arrays D and E on line 10 in Listing 1b), and then populate them using bbop_add and bbop_sub (lines 13 and 14 in Listing 1b), respectively. We then allocate the predicate array (i.e., array F on line 11 in Listing 1b) and populate it using bbop_greater (line 15 in Listing 1b). The addition, subtraction, and predicate arrays form the three inputs (arrays D, E, F) to the bbop_if_else instruction (line 16 in Listing 1b), which stores the outcome of the predicated execution to the destination array (i.e., array C in Listing 1b).

In this work, we assume that the programmer manually rewrites the code to use SIMDRAM operations. We follow this approach when evaluating real-world applications in §7.3. We envision two programming models for SIMDRAM. In the first programming model, SIMDRAM operations are encapsulated within userspace library routines to ease programmability. With this approach, the programmer can optimize the SIMDRAM-based code to make the most out of the underlying in-DRAM computing mechanism. In the second programming model, SIMDRAM operations are transparently inserted within the application's binary using compiler assistance. Since SIMDRAM is a SIMD-like compute engine, we expect that the

```
1  int size = 65536;
2  int elm_size = sizeof(uint8_t);
3  uint8_t *A, *B, *C = (uint8_t*)malloc(size*elm_size);
4  uint8_t *pred = (uint8_t*)malloc(size*elm_size);
5  ...
6  for(int i = 0; i < size; ++i) {
7      bool cond = A[i] > pred[i];
8      if (cond)
9          C[i] = A[i] + B[i];
10     else
11         C[i] = A[i] - B[i];
12 }
```

(a) C code for vector add/sub with predicated execution

```
1  int size = 65536;
2  int elm_size = sizeof(uint8_t);
3  uint8_t *A, *B, *C = (uint8_t*)malloc(size*elm_size);
4
5  bbop_trsp_init(A,size,elm_size);
6  bbop_trsp_init(B,size,elm_size);
7  bbop_trsp_init(C,size,elm_size);
8  uint8_t *pred = (uint8_t*)malloc(size*elm_size);
9  // D, E, F store intermediate data
10 uint8_t *D, *E = (uint8_t*)malloc(size*elm_size);
11 bool *F = (bool*)malloc(size*sizeof(bool));
12 ...
13 bbop_add(D, A, B, size, elm_size);
14 bbop_sub(E, A, B, size, elm_size);
15 bbop_greater(F, A, pred, size, elm_size);
16 bbop_if_else(C, D, E, F, size, elm_size);
```

(b) Equivalent code using SIMDRAM operations

Listing 1: Example code using SIMDRAM instructions.

compiler can generate SIMDRAM code without programmer intervention in at least two ways. First, it can leverage auto-vectorization routines already present in modern compilers [34, 95] to generate SIMDRAM code, by setting the width of the SIMD lanes equivalent to a DRAM row. For example, in LLVM [78], the width of the SIMD units can be defined using the "-force-vector-width" flag [95]. A SIMDRAM-based compiler back-end can convert the LLVM intermediate representation instructions into *bbop* instructions. Second, the compiler can compose groups of existing SIMD instructions generated by the compiler (e.g., AVX2 instructions [32]) into blocks that match the size of a DRAM row, and then convert such instructions into a single SIMDRAM operation. Prior work [2] uses a similar approach for 3D-stacked PIM. We leave the design of a compiler for SIMDRAM for future work.

SIMDRAM instructions can be implemented by extending the ISA of the host CPU. This is possible since there is enough unused opcode space to support the extra opcodes that SIMDRAM requires. To illustrate, prior works [97, 98] show that there are 389 unused operation codes considering only the AVX and SSE extensions for the x86 ISA. Extending the instruction set is a common approach to interface a CPU with PIM architectures [3, 131].

5.3 Handling Page Faults, Address Translation, Coherence, and Interrupts

SIMDRAM handles four key system mechanisms as follows:
- *Page Faults:* We assume that the pages that are touched during in-DRAM computation are already present and pinned in DRAM. In case the required data is not present in DRAM, we rely on

the conventional page fault handling mechanism to bring the required pages into DRAM.

- *Address Translation:* Virtual memory and address translation are challenging for many PIM architectures [4, 44, 120]. SIMDRAM is relieved of such challenge as it operates directly on physical addresses. When the CPU issues a SIMDRAM instruction, the instruction's virtual memory addresses are translated into their corresponding physical addresses using the same translation lookaside buffer (TLB) lookup mechanisms used by regular load/store operations.
- *Coherence:* Input arrays to SIMDRAM may be generated or modified by the CPU, and the data updates may reside only in the cache (e.g., because the updates have not yet been written back to DRAM). To ensure that SIMDRAM does not operate on stale data, programmers are responsible for flushing cache lines [13, 61] modified by the CPU. SIMDRAM can leverage coherence optimizations tailored to PIM to improve overall performance [18, 19].
- *Interrupts:* Two cases where an interrupt could affect the execution of a SIMDRAM operation are (1) on an application context switch, and (2) on a page fault. In case of a context switch, the control unit's context needs to be saved and then restored later when the application resumes execution. We do not expect to encounter a page fault during the execution of a SIMDRAM operation since, as previously mentioned, pages touched by SIMDRAM operations are expected to be loaded into and pinned in DRAM.

5.4 Handling Limited Subarray Size

SIMDRAM operates on data placed within the same subarray. However, a single subarray stores only several megabytes of data. For example, a subarray with 1024 rows and a row size of 8 kB can only store 8 MB of data. Therefore, SIMDRAM needs to use a mechanism that can efficiently move data within DRAM (e.g., across DRAM banks and subarrays). SIMDRAM can exploit (1) RowClone Pipelined Serial Mode (PSM) [130] to copy data between two banks by using the internal DRAM bus, or (2) Low-Cost Inter-Linked Subarrays (LISA) [21] to copy rows between two subarrays within the same bank. We evaluate the performance overheads of using both mechanisms in §7.6. Other mechanisms for fast in-DRAM data movement [135, 148] can also enhance SIMDRAM's capability.

5.5 Security Implications

SIMDRAM and other similar in-DRAM computation mechanisms that use dedicated DRAM rows to perform computation may increase vulnerability to RowHammer attacks [36, 68, 72, 103, 106]. We believe, and the literature suggests, that there should be robust and scalable solutions to RowHammer, orthogonally to our work (e.g., BlockHammer [150], PARA [71], TWiCe [86], Graphene [116]). Exploring RowHammer prevention and mitigation mechanisms in conjunction with SIMDRAM (or other PIM approaches) requires special attention and research, which we leave for future work.

5.6 SIMDRAM Limitations

We note three key limitations of the current version of the SIM-DRAM framework:

- *Floating-Point Operations:* SIMDRAM supports only integer and fixed-point operations. Enabling floating-point operations in-DRAM while maintaining low area overheads is a challenge. For example, for floating-point addition, the IEEE 754 FP32 format [58] requires shifting the mantissa by the difference of the

exponents of elements. Since each bitline stores a data element in SIMDRAM, shifting the value stored in one bitline without compromising the values stored in other bitlines at low cost is currently infeasible.

- *Operations That Require Shuffling Data Across Bitlines:* Different from prior works (e.g., DRISA [91]), SIMDRAM does *not* add any extra circuitry to perform bit-shift operations. Instead, SIMDRAM stores data in a vertical layout and can perform explicit bit-shift operations (if needed) by orchestrating row copies. Even though this approach enables SIMDRAM to implement a large range of operations, it is not possible to perform shuffling and reduction operations *across bitlines* without the inclusion of dedicated bit-shifting circuitry. This is due to the lack of physical connections across bitlines, which can be solved by building a bit-shift engine near the sense amplifiers.
- *Synchronization Between Concurrent In-DRAM Operations:* SIM-DRAM can be easily modified to enable concurrent execution of *distinct operations* across different subarrays in DRAM. However, this would require the implementation of software or hardware synchronization primitives to orchestrate the computation of a single task across different subarrays. Ideas that are similar to SynCron [45] can be beneficial.

6 Methodology

We implement SIMDRAM using the gem5 simulator [16] and compare it to a real multicore CPU (Intel Skylake [60]), a real high-end GPU (NVIDIA Titan V [112]), and a state-of-the-art processing-using-DRAM mechanism (Ambit [131]). In all our evaluations, the CPU code is optimized to leverage AVX-512 instructions [32]. Table 2 shows the system parameters we use in our evaluations. To measure CPU performance, we implement a set of timers in `sys/time.h` [144]. To measure CPU energy consumption, we use Intel RAPL [51]. To measure GPU performance, we implement a set of timers using the `cudaEvents` API [22]. We capture GPU kernel execution time that excludes data initialization/transfer time. To measure GPU energy consumption, we use the `nvml` API [111]. We report the average of five runs for each CPU/GPU data point, each with a warmup phase to avoid cold cache effects. We implement Ambit on gem5 and validate our implementation rigorously with the results reported in [131]. We use the same vertical data layout in our Ambit and SIMDRAM implementations, which enables us to (1) evaluate all 16 SIMDRAM operations in Ambit using their equivalent AND/OR/NOT-based implementations, and (2) highlight the benefits of Step 1 in the SIMDRAM framework (i.e., using an optimized MAJ/NOT-based implementation of the operations). Our synthetic throughput analysis (§7.1) uses 64M-element input arrays.

Table 2: Evaluated system configurations.

Intel Skylake CPU [60]	x86 [61], 16 cores, 8-wide, out-of-order, 4 GHz; *L1 Data + Inst. Private Cache:* 32 kB, 8-way, 64 B line; *L2 Private Cache:* 256 kB, 4-way, 64 B line; *L3 Shared Cache:* 8 MB, 16-way, 64 B line; *Main Memory:* 32 GB DDR4-2400, 4 channels, 4 ranks
NVIDIA Titan V GPU [112]	6 graphics processing clusters, 5120 CUDA Cores; 80 streaming multiprocessors, 1.2 GHz base clock; *L2 Cache:* 4.5 MB L2 Cache; *Main Memory:* 12 GB HBM [63, 82]
Ambit [131] and SIMDRAM	gem5 system emulation; x86 [61], 1-core, out-of-order, 4 GHz; *L1 Data + Inst. Cache:* 32 kB, 8-way, 64 B line; *L2 Cache:* 256 kB, 4-way, 64 B line; *Memory Controller:* 8 kB row size, FR-FCFS [107, 154] scheduling *Main Memory:* DDR4-2400, 1 channel, 1 rank, 16 banks

We evaluate three different configurations of SIMDRAM where 1 (*SIMDRAM:1*), 4 (*SIMDRAM:4*), and 16 (*SIMDRAM:16*) banks out of all the banks in one channel (16 banks in our evaluations) have SIM-DRAM computation capability. In the SIMDRAM 1-bank configuration, our mechanism exploits 65536 (i.e., size of an 8 kB row buffer) SIMD lanes. Conventional DRAM architectures exploit bank-level parallelism (BLP) to maximize DRAM throughput [73–75, 81, 108]. The memory controller can issue commands to different banks (one-per-cycle) on the same channel such that banks can operate in parallel. In SIMDRAM, banks in the same channel can operate in parallel, just like conventional banks. Therefore, to enable the required parallelism, SIMDRAM requires no more modifications. Accordingly, the number of available SIMD lanes, i.e., SIMDRAM's computation capability, increases by exploiting BLP in SIMDRAM configurations (i.e., the number of available SIMD lanes in the 16-bank configuration is 16×65536).[5]

7 Evaluation

We demonstrate the advantages of the SIMDRAM framework by evaluating: (1) SIMDRAM's throughput and energy consumption for a wide range of operations; (2) SIMDRAM's performance benefits on real-world applications; (3) SIMDRAM's performance and energy benefits over a closely-related processing-using-cache architecture [38]; and (4) the reliability of SIMDRAM operations. Finally, we evaluate three key overheads in SIMDRAM: in-DRAM data movement, data transposition, and area cost.

7.1 Throughput Analysis

Fig. 9 (left) shows the normalized throughput of all 16 SIMDRAM operations (§4.4) compared to those on CPU, GPU, and Ambit (normalized to the multicore CPU throughput), for an element size of 32 bits. We provide the absolute throughput of the baseline CPU (in GOps/s) in each graph. We classify each operation based on how the latency of the operation scales with respect to element size n.[6] Class 1, 2, and 3 operations scale linearly, logarithmically, and quadratically with n, respectively. Fig. 9 (right) shows how the average throughput across all operations of the same class scales relative to element size. We evaluate element sizes of 8, 16, 32, 64 bits. We normalize the figure to the average throughput on a CPU.
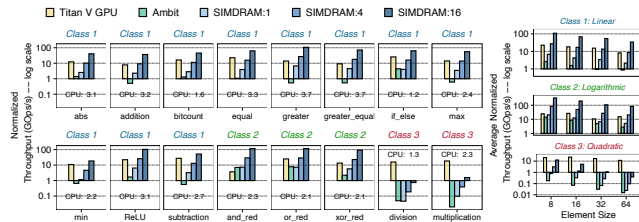

Figure 9: Normalized throughput of 16 operations. SIM-DRAM:*X* uses *X* DRAM banks for computation.

We make four observations from Fig. 9. First, we observe that SIMDRAM outperforms the three state-of-the-art baseline systems i.e., CPU/GPU/Ambit. Compared to CPU/GPU, SIMDRAM's throughput is 5.5×/0.4×, 22.0×/1.5×, and 88.0×/5.8× that of the CPU/GPU, averaged across all 16 SIMDRAM operations for 1, 4,

and 16 banks, respectively. To ensure fairness, we compare Ambit, which uses a single DRAM bank in our evaluations, only against *SIMDRAM:1*.[7] Our evaluations show that *SIMDRAM:1* outperforms Ambit by 2.0×, averaged across all 16 SIMDRAM operations. Second, SIMDRAM outperforms the GPU baseline when we use more than four DRAM banks for *all* the linear and logarithmic operations. *SIMDRAM:16* provides 5.7× (9.3×) the throughput of the GPU across all linear (logarithmic) operations, on average. *SIMDRAM:16*'s throughput is 83× (189×) and 45.2× (19.9×) that of CPU and Ambit, respectively, averaged across all linear (logarithmic) operations. Third, we observe that both the multicore CPU baseline and GPU outperform *SIMDRAM:1*, *SIMDRAM:4*, and *SIMDRAM:16* *only* for the division and multiplication operations. This is due to the quadratic nature of our bit-serial implementation of these two operations. Fourth, as expected, we observe a drop in the throughput for *all* operations with increasing element size, since the latency of each operation increases with element size. We conclude that SIMDRAM significantly outperforms all three state-of-the-art baselines for a wide range of operations.

7.2 Energy Analysis

We use CACTI [102] to evaluate SIMDRAM's energy consumption. Prior work [131] shows that each additional simultaneous row activation increases energy consumption by 22%. We use this observation in evaluating the energy consumption of SIMDRAM, which requires TRAs. Fig. 10 compares the energy efficiency (Throughput per Watt) of SIMDRAM against the GPU and Ambit baselines, normalized to the CPU baseline. We provide the absolute Throughput per Watt of the baseline CPU in each graph. We make four observations. First, SIMDRAM significantly increases energy efficiency for *all* operations over *all* three baselines. SIMDRAM's energy efficiency is 257×, 31×, and 2.6× that of CPU, GPU, and Ambit, respectively, averaged across all 16 operations. The energy savings in SIMDRAM directly result from (1) avoiding the costly off-chip round-trips to load/store data from/to memory, (2) exploiting the abundant memory bandwidth within the memory device, reducing execution time, and (3) reducing the number of TRAs required to compute a given operation by leveraging an optimized majority-based implementation of the operation. Second, similar to our results on throughput (§7.1), the energy efficiency of SIMDRAM reduces as element size increases. However, the energy efficiency of the CPU or GPU does not. This is because (1) for all SIMDRAM operations, the number of TRAs increases with element size; and (2) CPU and GPU can fully utilize their wider arithmetic units with larger (i.e., 32- and 64-bit) element sizes. Third, even though SIMDRAM multiplication and division operations scale poorly with element size, the SIMDRAM implementations of these operations are significantly more energy-efficient compared to the CPU and GPU baselines, making SIMDRAM a competitive candidate even for multiplication and division operations. Fourth, since *both* SIMDRAM's throughput and power consumption increase proportionally to the number of banks, the Throughput per Watt for SIMDRAM 1-, 4-, and 16-bank configurations is the same. We conclude that SIMDRAM is more energy-efficient than all three state-of-the-art baselines for a wide range of operations.

---

[5]SIMDRAM computation capability can be further increased by enabling and exploiting subarray-level parallelism in each bank [20, 21, 65, 74].

[6]Appendix C discusses the scalability of each operation.

[7]Ambit's throughput scales proportionally to bank count, just like SIMDRAM's.
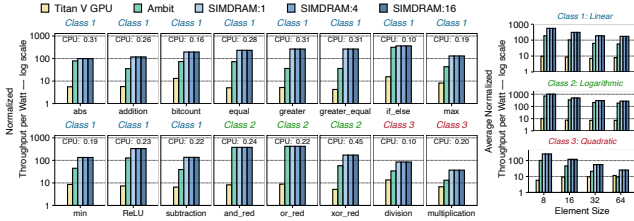
Figure 10: Normalized energy efficiency of 16 operations.

7.3 Effect on Real-World Kernels

We evaluate SIMDRAM with a set of kernels that represent the behavior of selected important real-world applications from different domains. The evaluated kernels come from databases (TPC-H query 1 [145], BitWeaving [93]), convolutional neural networks (LeNET-5 [79], VGG-13 [138], VGG-16 [138]), classification algorithms (k-nearest neighbors [88]), and image processing (brightness [47]). These kernels rely on many of the basic operations we evaluate in §7.1. We provide a brief description of each kernel and the SIMDRAM operations it utilizes in Appendix D.

Fig. 11 shows the performance of SIMDRAM and our baseline configurations for each kernel, normalized to that of the multi-core CPU. We make four observations. First, *SIMDRAM:16* greatly outperforms the CPU and GPU baselines, providing 21× and 2.1× the performance of the CPU and GPU, respectively, on average across all seven kernels. SIMDRAM has a maximum performance of 65× and 5.4× that of the CPU and GPU, respectively (for the BitWeaving kernel in both cases). Similarly, *SIMDRAM:1* provides 2.5× the performance of Ambit (which also uses a single bank for in-DRAM computation), on average across all seven kernels, with a maximum of 4.8× the performance of Ambit for the TPC-H kernel. Second, even with a single DRAM bank, SIMDRAM *always* outperforms the CPU baseline, providing 2.9× the performance of the CPU on average across all kernels. Third, *SIMDRAM:4* provides 2× and 1.1× the performance of the GPU baseline for the BitWeaving and brightness kernels, respectively. Fourth, despite GPU's higher multiplication throughput compared to SIMDRAM (§7.1), *SIMDRAM:16* outperforms the GPU baseline even for kernels that heavily rely on multiplication (Appendix D) (e.g., by 1.03× and 2.5× for kNN and TPC-H kernels, respectively). This speedup is a direct result of exploiting the high in-DRAM bandwidth in SIMDRAM to avoid the memory bottleneck in GPU caused by the large amounts of intermediate data generated in such kernels. We conclude that SIMDRAM is an effective and efficient substrate to accelerate many commonly-used real-world applications.
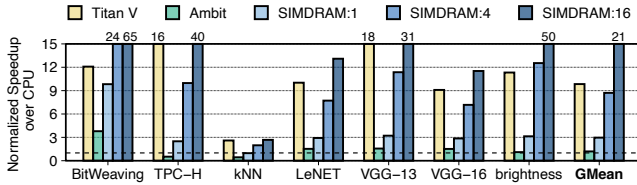

Figure 11: Normalized speedup of real-world kernels.

7.4 Comparison to DualityCache

We compare SIMDRAM to DualityCache [38], a closely-related processing-using-cache architecture. DualityCache is an in-cache computing framework that performs computation using discrete logic elements (e.g., logic gates, latches, muxes) that are added to the SRAM peripheral circuitry. In-cache computing approaches (such as DualityCache) need data to be brought into the cache first, which requires extra data movement (and even more if the working set of the application does not fit in the cache) compared to in-memory computing approaches (like SIMDRAM).

Fig. 12 (top) compares the latency of SIMDRAM against DualityCache [38] for the subset of operations that *both* SIMDRAM and DualityCache support (i.e., addition, subtraction, multiplication, and division). In this experiment, we study three different configurations. First, *DualityCache:Ideal* has *all* data required for DualityCache residing in the cache. Therefore, results for *DualityCache:Ideal* do *not* include the overhead of moving data from DRAM to the cache, making it an unrealistic configuration that needs the data to already reside and fit in the cache. Second, *DualityCache:Realistic* includes the overhead of data movement from DRAM to the cache. Both DualityCache configurations compute on an input array of 45 MB. Third, *SIMDRAM:16*. For all three configurations, we use the same cache size (35 MB) as the original DualityCache work [38] to provide a fair comparison. As shown in the figure, SIMDRAM greatly outperforms DualityCache when data movement is realistically taken into account. *SIMDRAM:16* outperforms *DualityCache:Realistic* for all four operations (by 52.9×, 52.4×, 1.8×, and 2.1× for addition, subtraction, multiplication, and division respectively, on average across all element sizes). SIMDRAM's performance improvement comes at a much lower area overhead compared to DualityCache. DualityCache (including its peripherals, transpose memory unit, controller, miss status holding registers, and crossbar network) has an area overhead of 3.5% in a high-end CPU, whereas SIMDRAM has an area overhead of only 0.2% (§7.8). As a result, SIMDRAM can actually fit a significantly higher number of SIMD lanes in a given area compared to DualityCache. Therefore, SIMDRAM's performance improvement per unit area would be much larger than that we observe in Fig. 12. We conclude that SIMDRAM achieves higher performance at lower area cost over DualityCache, when we consider DRAM-to-cache data movement.
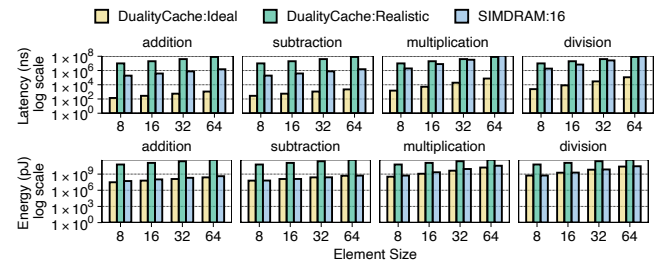
Figure 12: Latency and energy to execute 64M operations.

Fig. 12 (bottom) shows the energy consumption of *DualityCache:Realistic*, *DualityCache:Ideal*, and *SIMDRAM:16* when performing 64M addition, subtraction, multiplication, and division operations. We make two observations. First, compared to *DualityCache:Ideal*, *SIMDRAM:16* increases average energy consumption by 60%. This is because while the energy per bit to perform computation in DRAM (13.3 nJ/bit [101, 147]) is smaller than the energy per bit to perform computation in the cache (60.1 nJ/bit [29]), the DualityCache implementation of each operation requires fewer

iterations than its equivalent SIMDRAM implementation. Second, *SIMDRAM:16* reduces average energy by 600× over *DualityCache:Realistic* because *DualityCache:Realistic* needs to load *all* input data from DRAM, incurring high energy overhead (a DRAM access consumes 650× the energy-per-bit of a DualityCache operation [29, 38]). In contrast, SIMDRAM operates on data that is already present in DRAM, eliminating any data movement overhead. We conclude that SIMDRAM is much more efficient than DualityCache, when cache-to-DRAM data movement is realistically considered.

7.5 Reliability

We use SPICE simulations to test the reliability of SIMDRAM for different technology nodes and varying amounts of process variation. At the core of SIMDRAM, there are two back-to-back triple-row activations (TRAs). Table 3 shows the characteristics of TRA and two back-to-back TRAs (TRAb2b) for the 45, 32, and 22 nm technology nodes. We compare these with the reliability of quintuple-row activations (QRAs), used by prior works [6, 9] to implement bit-serial addition. We use the reference 55 nm DRAM model from Rambus [123] and scale it based on the ITRS roadmap [62, 147] to model smaller technology nodes following the PTM transistor models [110]. The goal of our analysis is to understand the reliability trends for TRA and QRA operations with technology scaling. For each technology node and process variation amount, we run Monte-Carlo simulations for 10^4 iterations.

Table 3: Process variation's effect on TRA/QRA failure rates.

	Variation (%)	± 0	± 5	± 10	± 20
45 nm	TRA Failure (%)	0	0	0.02	3.01
	TRAb2b Failure (%)	0	0	0.04	5.93
	QRA Failure (%)	0	0	0.35	6.54
32 nm	TRA Failure (%)	0	0	0.35	3.90
	TRAb2b Failure (%)	0	0	0.69	7.64
	QRA Failure (%)	0	0.42	6.33	11.52
22 nm	TRA Failure (%)	0	0	0.42	4.50
	TRAb2b Failure (%)	0	0	0.84	8.83
	QRA Failure (%)	error	error	error	error

We make four observations. First, for all process variation ranges, TRA and TRAb2b perform more reliably than QRA. Specifically, TRA and TRAb2b perform without errors for 5% variation. Second, while moving from 45 nm to 32 nm, we observe that the error rate of QRA increases faster than than that of TRA, making QRA less reliable as the technology node size reduces. Third, for TRA and TRAb2b in 22 nm, we observe a similar trend of increased error rate while still having zero error rate for 5% process variation. In our simulations, QRA does not perform correctly in the projected 22 nm DRAM. For example, MAJ(11100) always leads to the incorrect outcome of '0'. This is because charge sharing between five capacitors in QRA does not lead to enough voltage on the bitline for the sense amplifier to pull up the bitline to the value '1'. We believe that proposals based on QRA require changes to the circuit elements (e.g., transistors in the sense amplifier) to enable correct operation in the 22 nm technology node. Fourth, a TRA can fail depending on the amount of manufacturing process variation. We observe that a TRA starts to fail when process variation is larger than 10%, for all technology nodes. Since SIMDRAM operations are executed within a DRAM module, it is quite challenging to leverage existing in-DRAM or in-memory-controller error correction mechanisms [99, 100, 117, 118]. The same problem exists for

other processing-using-DRAM mechanisms [6, 9, 25, 27, 90, 91, 129–134, 143, 149]. We conclude that the TRA operations SIMDRAM relies on are much more scalable and variation-tolerant than QRA operations some prior works rely on. We leave a study of reliability solutions for future work.

7.6 Data Movement Overhead

There may be cases where the output of a SIMDRAM operation that is used as an input to a subsequent operation does not reside in the same subarray as other inputs. For example, consider the computation $C = OP(A, B)$. If the output of the SIMDRAM operation OP is an input to a subsequent SIMDRAM operation, C needs to move to the same subarray as the other inputs of the subsequent operation, before the operation can start. Fig. 13 shows the distribution of the worst-case latency overhead of moving the output of each of our 16 SIMDRAM operations with 8-, 16-, 32-, and 64-bit element sizes in *SIMDRAM:1* to a different subarray within the same bank, i.e., *intra-bank* (using LISA [21]) or a different bank, i.e., *inter-bank* (using RowClone PSM [130]). We make two observations. First, intra-bank data movement (Fig. 13, left) results in only 0.39% latency overhead, averaged across all 16 SIMDRAM operations and four different element sizes (max. 1.52% for 8-bit reduction, min. 0.001% for 64-bit multiplication). Second, inter-bank data movement (Fig. 13, right) results in 17.5% latency overhead, averaged across all 16 SIMDRAM operations and four different element sizes (max. 68.7% for 8-bit reduction, min. 0.03% for 64-bit multiplication). We observe that the latency overhead of moving data, as a fraction of the total computation latency *decreases* with element size, because the computation latency of each SIMDRAM operation increases with element size. We conclude that while efficient data movement is a challenge in processing-in-memory architectures that rely on moving and aligning operands, the performance overhead of data movement in SIMDRAM stays within an acceptable range even under worst-case assumptions.
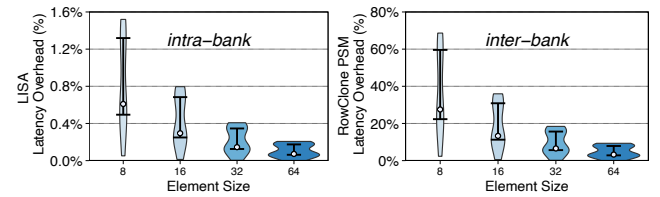

Figure 13: Latency overhead distribution of worst-case intra-bank (left) and inter-bank (right) data movement for *SIMDRAM:1*. Error bars depict the 25th and 75th percentiles.

7.7 Data Transposition Overhead

Transposition of the data in one subarray can overlap with in-DRAM computation in another subarray. As a result, if the data required for in-DRAM computation spans over multiple subarrays, only the transposition of the data in the first subarray is on the critical path of SIMDRAM execution. The data in each remaining subarray is then transposed simultaneously with the in-DRAM computation in the previous subarray.

To better understand the overhead of transposing data, we evaluate the worst-case latency of data transposition, which is when SIMDRAM's data initially resides in the cache in a horizontal layout. Before the computation of the SIMDRAM operation can start, this

data needs to be transposed to a vertical layout and transferred to DRAM, incurring additional latency. Fig. 14 shows this worst-case data transposition latency and the distribution of latency overhead of data transposition in *SIMDRAM:1* across all 16 SIMDRAM operations, as a function of element size. We make three observations. First, in *SIMDRAM:1* (*SIMDRAM:16*), data transposition incurs 7.1% (44.6%) latency overhead across all SIMDRAM operations (min. 0.03% (0.55%) for 64-bit multiplication, max. 38.9% (91.1%) for 8-bit AND-reduction and OR-reduction). As shown in §7.1, for all the evaluated element sizes, *SIMDRAM:1* (*SIMDRAM:16*) outperforms the CPU and GPU baselines by 5.5× and 0.4× (88.0× and 5.8×) on average across all 16 SIMDRAM operations, respectively. Even when we include the data transposition overhead, *SIMDRAM:1* (*SIMDRAM:16*) still outperforms both the CPU and GPU baselines by 4.0× and 0.24× (20.0× and 1.4×) on average across all 16 SIMDRAM operations. Our analysis for kernels that represent the behavior of real-world applications (§7.3) *already includes* the data transposition overhead. Second, the data transposition latency significantly increases with element size (by 9.7× from 8-bit elements to 64-bit elements). The number of cache lines that need to be transposed increases linearly with element size, which, in turn, increases the total transposition latency. Third, even though the transposition latency increases with element size, the transposition overhead as a fraction of the total latency *decreases* with element size, because the latency of each SIMDRAM operation also increases with element size. Since the transposition of data in each subarray is overlapped with the computation in another subarray, the increase in transposition latency is amortized over an even higher increase in the SIMDRAM operation latency. We conclude that SIMDRAM can efficiently perform in-DRAM computation even when worst-case data transposition overhead is taken into account.
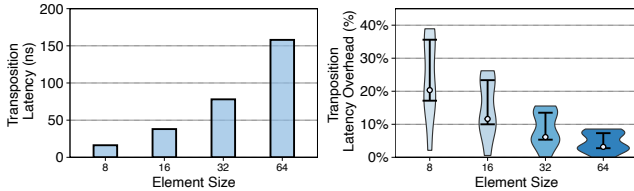

Figure 14: Worst-case latency (left) and worst-case latency overhead distribution (right) of data transposition in 16 SIM-DRAM operations for *SIMDRAM:1*. Error bars depict the 25th and 75th percentiles, and a bubble depicts the 50th percentile.

7.8 Area Overhead

We use CACTI [102] to evaluate the area overhead of the primary components in the SIMDRAM design using a 22 nm technology node. SIMDRAM does not introduce any modifications to DRAM circuitry other than those proposed by Ambit, which has an area overhead of <1% in a commodity DRAM chip [131]. Therefore, SIMDRAM's area overhead over Ambit is only two structures in the memory controller: the control and transposition units.

Control Unit Area Overhead. The main components in the SIMDRAM control unit are the (1) *bbop* FIFO, (2) μProgram Scratchpad, (3) μOp Memory. We size the *bbop* FIFO and μProgram Scratchpad to 2 kB each. The size of the *bbop* FIFO is enough to hold up to 1024 *bbop* instructions, which we observe is more than enough for

our real-world applications. The size of the μProgram Scratchpad is large enough to store the μPrograms for all 16 SIMDRAM operations that we evaluate in the paper (16 μPrograms × 128 B max per μProgram). We use a 128 B scratchpad for the μOp Memory.[2] We estimate that the SIMDRAM control unit area is 0.04 mm^2.

Transposition Unit Area Overhead. The primary components in the transposition unit are (1) the Object Tracker and (2) two transposition buffers. We use an 8 kB fully-associative cache with a 64-bit cache line size for the Object Tracker. This is enough to store 1024 entries in the Object Tracker, where each entry holds the base physical address of a SIMDRAM object (19 bits), the total size of the allocated data (32 bits), and the size of each element in the object (6 bits). Each transposition buffer is 4 kB, to transpose up to a 64-bit SIMDRAM object (64-bit × 64 B). We estimate the transposition unit area is 0.06 mm^2. Considering the area of the control and transposition units, SIMDRAM has an area overhead of only 0.2% compared to the die area of an Intel Xeon E5-2697 v3 CPU [38]. We conclude that SIMDRAM has low area cost.

8 Related Work

To our knowledge, SIMDRAM is the first end-to-end framework that supports in-DRAM computation flexibly and transparently to the user. We highlight SIMDRAM's key contributions by contrasting it with state-of-the-art processing-in-memory designs.

Processing-near-Memory (PnM) within 3D-Stacked Memories. Many recent works (e.g., [3, 4, 17–19, 26, 28, 30, 31, 41, 42, 45, 46, 50, 55, 56, 66, 70, 89, 96, 109, 113, 120, 125–127, 152]) explore adding logic directly to the logic layer of 3D-stacked memories (e.g., High-Bandwidth Memory [63, 82], Hybrid Memory Cube [57]). The implementation of SIMDRAM is considerably simpler, and relies on minimal modifications to commodity DRAM chips.

Processing-using-Memory (PuM). Prior works propose mechanisms wherein the memory arrays themselves perform various operations in bulk [6, 9, 25, 27, 90, 91, 129–134, 143, 149]. SIM-DRAM supports a much wider range of operations (compared to [6, 9, 25, 90, 91, 130, 131, 149]), at lower computational cost (compared to [131, 149]), at lower area overhead (compared to [91]), and with more reliable execution (compared to [6, 9]).

Processing-in-Cache. Recent works [1, 29, 38] propose in-SRAM accelerators that take advantage of the SRAM bitline structures to perform bit-serial computation in caches. SIMDRAM shares similarities with these approaches, but offers a significantly lower cost per bit by exploiting the high density and low cost of DRAM technology. We show the large performance and energy advantages of SIMDRAM compared to DualityCache [38] in §7.4.

Frameworks for PIM. Few prior works tackle the challenge of providing end-to-end support for PIM. We describe these frameworks and their limitations for in-DRAM computing. DualityCache [38] is an end-to-end framework for in-cache computing. DualityCache utilizes the CUDA/OpenAcc programming languages [22, 115] to generate code for an in-cache mechanism that executes a fixed set of operations in a single-instruction multiple-thread (SIMT) manner. Like SIMDRAM, DualityCache stores data in a vertical layout through the bitlines of the SRAM array. It treats each bitline as an independent execution thread and utilizes a crossbar network to allow inter-thread communication across bitlines. Despite its benefits, employing DualityCache in DRAM is not straightforward

for two reasons. First, extending the DRAM subarray with the crossbar network utilized by DualityCache in SRAM to allow inter-thread communication would impose a prohibitive area overhead in DRAM (9× the DRAM subarray area). Second, as an in-cache computing solution, DualityCache does *not* account for the limitations of in-DRAM computing, i.e., DRAM operations that destroy input data, limited number of DRAM rows that are capable of processing-using-DRAM, and the need to avoid costly in-DRAM copies. We have already shown that SIMDRAM achieves higher performance at lower area overhead than DualityCache, when DRAM-to-cache data movement is realistically taken into account (§7.4).

Two prior works propose frameworks targeting ReRAM devices. Hyper-AP [151] is a framework for associative processing using ReRAM. Since Hyper-AP targets associative processing, the proposed framework is *fundamentally* different from SIMDRAM. IMP [37] is a framework for in-situ ReRAM operations. Like DualityCache, the IMP framework depends on particular structures of the ReRAM array (such as analog-to-digital/digital-to-analog converters) to perform computation and, thus, is not applicable to an in-DRAM substrate that performs bulk bitwise operations. Moreover, DualityCache, Hyper-AP, and IMP each have a rigid ISA that enables only a limited set of in-memory operations (DualityCache supports 16 in-memory operations, while both Hyper-AP and IMP support 12). In contrast, SIMDRAM is the first framework for PuM that is flexible, providing a methodology that allows new operations to be integrated and computed in memory as needed. In summary, SIMDRAM fills the gap for a flexible end-to-end framework that targets processing-using-DRAM.

9 Conclusion

We introduce SIMDRAM, a massively-parallel general-purpose processing-using-DRAM framework that (1) enables the efficient implementation of a wide variety of operations in DRAM, in SIMD fashion, and (2) provides a flexible mechanism to support the implementation of arbitrary user-defined operations. SIMDRAM introduces a new three-step framework to enable efficient MAJ/NOT-based in-DRAM implementation for complex operations of different categories (e.g., arithmetic, relational, predication), and is applicable to a wide range of real-world applications. We design the hardware and ISA support for SIMDRAM framework to (1) address key system integration challenges, and (2) allow programmers to employ new SIMDRAM operations without hardware changes. We experimentally demonstrate that SIMDRAM provides significant performance and energy benefits over state-of-the-art CPU, GPU, and PuM systems. We hope that future work builds on our framework to further ease the adoption and improve the performance and efficiency of processing-using-DRAM architectures and applications.

Acknowledgments

References

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

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

[3] J. Ahn, S. Hong, S. Yoo, O. Mutlu, and K. Choi, "A Scalable Processing-in-Memory Accelerator for Parallel Graph Processing," in *ISCA*, 2015.

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

[5] B. Akin, F. Franchetti, and J. C. Hoe, "Data Reorganization in Memory Using 3D-Stacked DRAM," in *ISCA*, 2016.

[6] M. F. Ali, A. Jaiswal, and K. Roy, "In-Memory Low-Cost Bit-Serial Addition Using Commodity DRAM Technology," in *TCAS-I*, 2019.

[7] L. Amaru, P.-E. Gaillardon, and G. Micheli, "Majority-Inverter Graph: A Novel Data-Structure and Algorithms for Efficient Logic Optimization," in *DAC*, 2014.

[8] S. Angizi, N. A. Fahmi, W. Zhang, and D. Fan, "PIM-Assembler: A Processing-in-Memory Platform for Genome Assembly," in *DAC*, 2020.

[9] S. Angizi and D. Fan, "GraphiDe: A Graph Processing Accelerator Leveraging in-DRAM-Computing," in *GLSVLSI*, 2019.

[10] S. Angizi and D. Fan, "ReDRAM: A Reconfigurable Processing-in-DRAM Platform for Accelerating Bulk Bit-Wise Operations," in *ICCAD*, 2019.

[11] S. Angizi, Z. He, and D. Fan, "DIMA: A Depthwise CNN In-Memory Accelerator," in *ICCAD*, 2018.

[12] S. Angizi, Z. He, F. Parveen, and D. Fan, "IMCE: Energy-Efficient Bitwise In-Memory Convolution Engine for Deep Neural Network," in *ASP-DAC*, 2018.

[13] ARM Ltd., *Cortex-A8 Technical Reference Manual*, 2010.

[14] O. O. Babarinsa and S. Idreos, "JAFAR: Near-Data Processing for Databases," in *SIGMOD*, 2015.

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

[16] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood, "The gem5 Simulator," *Comput. Archit. News*, 2011.

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

[18] A. Boroumand, S. Ghose, B. Lucia, K. Hsieh, K. Malladi, H. Zheng, and O. Mutlu, "LazyPIM: An Efficient Cache Coherence Mechanism for Processing-in-Memory," *CAL*, 2017.

[19] A. Boroumand, S. Ghose, M. Patel, H. Hassan, B. Lucia, R. Ausavarungnirun, K. Hsieh, N. Hajinazar, K. T. Malladi, H. Zheng, and O. Mutlu, "CoNDA: Efficient Cache Coherence Support for Near-Data Accelerators," in *ISCA*, 2019.

[20] K. K. Chang, D. Lee, Z. Chishti, A. R. Alameldeen, C. Wilkerson, Y. Kim, and O. Mutlu, "Improving DRAM Performance by Parallelizing Refreshes with Accesses," in *HPCA*, 2014.

[21] K. K. Chang, P. J. Nair, D. Lee, S. Ghose, M. K. Qureshi, and O. Mutlu, "Low-Cost Inter-Linked Subarrays (LISA): Enabling Fast Inter-Subarray Data Movement in DRAM," in *HPCA*, 2016.

[22] J. Cheng, M. Grossman, and T. McKercher, *Professional CUDA C Programming*. John Wiley & Sons, 2014.

[23] P. Chi, S. Li, C. Xu, T. Zhang, J. Zhao, Y. Liu, Y. Wang, and Y. Xie, "PRIME: A Novel Processing-in-Memory Architecture for Neural Network Computation in ReRAM-Based Main Memory," in *ISCA*, 2016.

[24] L. Deng, "The MNIST Database of Handwritten Digit Images for Machine Learning Research [Best of the Web]," *IEEE Signal Processing Magazine*, 2012.

[25] Q. Deng, L. Jiang, Y. Zhang, M. Zhang, and J. Yang, "DrAcc: A DRAM Based Accelerator for Accurate CNN Inference," in *DAC*, 2018.

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

[27] P. Dlugosch, D. Brown, P. Glendenning, M. Leventhal, and H. Noyes, "An Efficient and Scalable Semiconductor Architecture for Parallel Automata Processing," *TPDS*, 2014.

[28] M. Drumond, A. Daglis, N. Mirzadeh, D. Ustiugov, J. Picorel, B. Falsafi, B. Grot, and D. Pnevmatikatos, "The Mondrian Data Engine," in *ISCA*, 2017.

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

[30] A. Farmahini-Farahani, J. H. Ahn, K. Morrow, and N. S. Kim, "NDA: Near-DRAM Acceleration Architecture Leveraging Commodity DRAM Devices and Standard Memory Modules," in *HPCA*, 2015.

[31] I. Fernandez, R. Quislant, E. Gutiérrez, O. Plata, C. Giannoula, M. Alser, J. Gómez-Luna, and O. Mutlu, "NATSA: A Near-Data Processing Accelerator for Time Series Analysis," in *ICCD*, 2020.

[32] N. Firasta, M. Buxton, P. Jinbo, K. Nasri, and S. Kuo, "Intel AVX: New Frontiers in Performance Improvements and Energy Efficiency," Intel Corp., 2008, white paper.

[33] J. D. Foley, F. D. Van, A. Van Dam, S. K. Feiner, J. F. Hughes, E. Angel, and J. Hughes, *Computer Graphics: Principles and Practice*, 1996.

[34] Free Software Foundation, "GNU Project: Auto-Vectorization in GCC," https://gcc.gnu.org/projects/tree-ssa/vectorization.html.

[35] J. Friedman, T. Hastie, and R. Tibshirani, *The Elements of Statistical Learning (2nd Edition)*. Springer-Verlag, 2008.

[36] P. Frigo, E. Vannacci, H. Hassan, V. van der Veen, O. Mutlu, C. Giuffrida, H. Bos, and K. Razavi, "TRRespass: Exploiting the Many Sides of Target Row Refresh," in *IEEE S&P*, 2020.

[37] D. Fujiki, S. Mahlke, and R. Das, "In-Memory Data Parallel Processor," in *ASPLOS*, 2018.

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

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

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

[41] M. Gao and C. Kozyrakis, "HRL: Efficient and Flexible Reconfigurable Logic for Near-Data Processing," in *HPCA*, 2016.

[42] M. Gao, J. Pu, X. Yang, M. Horowitz, and C. Kozyrakis, "TETRIS: Scalable and Efficient Neural Network Acceleration with 3D Memory," in *ASPLOS*, 2017.

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

[44] S. Ghose, K. Hsieh, A. Boroumand, R. Ausavarungnirun, and O. Mutlu, "The Processing-in-Memory Paradigm: Mechanisms to Enable Adoption," in *Beyond-CMOS Technologies for Next Generation Computer Design*. Springer, 2019, preprint available at arXiv:1802.00320 [cs.AR].

[45] C. Giannoula, N. Vijaykumar, N. Papadopoulou, V. Karakostas, I. Fernandez, J. Gómez-Luna, L. Orosa, N. Koziris, G. Goumas, and O. Mutlu, "SynCron: Efficient Synchronization Support for Near-Data-Processing Architectures," in *HPCA*, 2021.

[46] M. Gokhale, B. Holmes, and K. Iobst, "Processing in Memory: The Terasys Massively Parallel PIM Array," *Computer*, 1995.

[47] R. C. Gonzalez and R. E. Woods, *Digital Image Processing*, 2nd ed. Addison-Wesley, 2002.

[48] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. MIT Press, 2016.

[49] P. Gu, S. Li, D. Stow, R. Barnes, L. Liu, Y. Xie, and E. Kursun, "Leveraging 3D Technologies for Hardware Security: Opportunities and Challenges," in *GLSVLSI*, 2016.

[50] P. Gu, X. Xie, Y. Ding, G. Chen, W. Zhang, D. Niu, and Y. Xie, "iPIM: Programmable In-Memory Image Processing Accelerator Using Near-Bank Architecture," in *ISCA*, 2020.

[51] M. Hähnel, B. Döbel, M. Völp, and H. Härtig, "Measuring Energy Consumption for Short Code Paths Using RAPL," *SIGMETRICS*, 2012.

[52] Z. He, L. Yang, S. Angizi, A. S. Rakin, and D. Fan, "Sparse BD-Net: A Multiplication-Less DNN with Sparse Binarized Depth-Wise Separable Convolution," *JETC*, 2020.

[53] W. D. Hillis and L. W. Tucker, "The CM-5 Connection Machine: A Scalable Supercomputer," *CACM*, 1993.

[54] W. D. Hillis, "The Connection Machine," Ph.D. dissertation, Massachusetts Inst. of Technology, 1988.

[55] K. Hsieh, E. Ebrahimi, G. Kim, N. Chatterjee, M. O'Connor, N. Vijaykumar, O. Mutlu, and S. W. Keckler, "Transparent Offloading and Mapping (TOM) Enabling Programmer-Transparent Near-Data Processing in GPU Systems," in *ISCA*, 2016.

[56] K. Hsieh, S. Khan, N. Vijaykumar, K. K. Chang, A. Boroumand, S. Ghose, and O. Mutlu, "Accelerating Pointer Chasing in 3D-Stacked Memory: Challenges, Mechanisms, Evaluation," in *ICCD*, 2016.

[57] Hybrid Memory Cube Consortium, "Hybrid Memory Cube Specification Rev. 2.1," 2014.

[58] IEEE, "IEEE Standard for Floating-Point Arithmetic," Standard 754-2019, 2019.

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

[60] Intel Corp., "6th Generation Intel Core Processor Family Datasheet," http://www.intel.com/content/www/us/en/processors/core/.

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

[62] International Technology Roadmap for Semiconductors, "ITRS Reports," http://www.itrs2.net/itrs-reports.html, 2015.

[63] JEDEC Solid State Technology Assn., "JESD235C: High Bandwidth Memory (HBM) DRAM," January 2020.

[64] B. A. Kahle and W. D. Hillis, "The Connection Machine Model CM-1 Architecture," Thinking Machines Corp., Tech. Rep., 1989.

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

[66] D. Kim, J. Kung, S. Chai, S. Yalamanchili, and S. Mukhopadhyay, "Neurocube: A Programmable Digital Neuromorphic Architecture with High-Density 3D Memory," in *ISCA*, 2016.

[67] J. Kim, M. Patel, H. Hassan, and O. Mutlu, "Solar-DRAM: Reducing DRAM Access Latency by Exploiting the Variation in Local Bitlines," in *ICCD*, 2018.

[68] J. Kim, M. Patel, A. G. Yağlıkçı, H. Hassan, R. Azizi, L. Orosa, and O. Mutlu, "Revisiting RowHammer: An Experimental Analysis of Modern DRAM Devices and Mitigation Techniques," in *ISCA*, 2020.

[69] J. S. Kim, M. Patel, H. Hassan, L. Orosa, and O. Mutlu, "D-RaNGe: Using Commodity DRAM Devices to Generate True Random Numbers with Low Latency and High Throughput," in *HPCA*, 2019.

[70] J. S. Kim, D. Senol Cali, H. Xin, D. Lee, S. Ghose, M. Alser, H. Hassan, O. Ergin, C. Alkan, and O. Mutlu, "GRIM-Filter: Fast Seed Location Filtering in DNA Read Mapping Using Processing-in-Memory Technologies," in *APBC*, 2018.

[71] Y. Kim, R. Daly, J. Kim, C. Fallin, j. Lee, D. Lee, C. Wilkerson, K. Lai, and O. Mutlu, "RowHammer: Reliability Analysis and Security Implications," in *ISCA*, 2014.

[72] Y. Kim, R. Daly, J. Kim, C. Fallin, J. H. Lee, D. Lee, C. Wilkerson, K. Lai, and O. Mutlu, "Flipping Bits in Memory Without Accessing Them: An Experimental Study of DRAM Disturbance Errors," in *ISCA*, 2014.

[73] Y. Kim, M. Papamichael, O. Mutlu, and M. Harchol-Balter, "Thread Cluster Memory Scheduling: Exploiting Differences in Memory Access Behavior," in *MICRO*, 2010.

[74] Y. Kim, V. Seshadri, D. Lee, J. Liu, and O. Mutlu, "A Case for Exploiting Subarray-Level Parallelism (SALP) in DRAM," in *ISCA*, 2012.

[75] Y. Kim, W. Yang, and O. Mutlu, "Ramulator: A Fast and Extensible DRAM Simulator," *CAL*, 2015.

[76] A. Krizhevsky, "Convolutional Deep Belief Networks on CIFAR-10," https://www.cs.toronto.edu/~kriz/conv-cifar10-aug2010.pdf, 2010.

[77] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet Classification with Deep Convolutional Neural Networks," in *NIPS*, 2012.

[78] C. Lattner and V. Adve, "LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation," in *CGO*, 2004.

[79] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner, "LeNet-5, Convolutional Neural Networks," http://yann.lecun.com/exdb/lenet, 2015.

[80] Y. Lecun, L. D. Jackel, L. Bottou, C. Cartes, J. S. Denker, H. Drucker, U. Müller, E. Säckinger, P. Simard, V. Vapnik, and et al., "Learning Algorithms For Classification: A Comparison On Handwritten Digit Recognition," in *Neural Networks: The Statistical Mechanics Perspective*, 1995.

[81] C. J. Lee, V. Narasiman, O. Mutlu, and Y. N. Patt, "Improving Memory Bank-Level Parallelism in the Presence of Prefetching," in *MICRO*, 2009.

[82] D. Lee, S. Ghose, G. Pekhimenko, S. Khan, and O. Mutlu, "Simultaneous Multi-Layer Access: Improving 3D-Stacked Memory Bandwidth at Low Cost," *ACM TACO*, 2016.

[83] D. Lee, S. Khan, L. Subramanian, S. Ghose, R. Ausavarungnirun, G. Pekhimenko, V. Seshadri, and O. Mutlu, "Design-Induced Latency Variation in Modern DRAM Chips: Characterization, Analysis, and Latency Reduction Mechanisms," in *SIGMETRICS*, 2017.

[84] D. Lee, Y. Kim, G. Pekhimenko, S. Khan, V. Seshadri, K. Chang, and O. Mutlu, "Adaptive-Latency DRAM: Optimizing DRAM Timing for the Common-Case," in *HPCA*, 2015.

[85] D. Lee, Y. Kim, V. Seshadri, J. Liu, L. Subramanian, and O. Mutlu, "Tiered-Latency DRAM: A Low Latency and Low Cost DRAM Architecture," in *HPCA*, 2013.

[86] E. Lee, I. Kang, S. Lee, G. E. Suh, and J. H. Ahn, "TWiCe: Preventing Row-Hammering by Exploiting Time Window Counters," in *ISCA*, 2019.

[87] J. H. Lee, J. Sim, and H. Kim, "BSSync: Processing Near Memory for Machine Learning Workloads with Bounded Staleness Consistency Models," in *PACT*, 2015.

[88] Y. Lee, "Handwritten Digit Recognition Using k-Nearest-Neighbor, Radial-Basis Function, and Backpropagation Neural Networks," *Neural Computation*, 1991.

[89] M. Lenjani, P. Gonzalez, E. Sadredini, S. Li, Y. Xie, A. Akel, S. Eilert, M. R. Stan, and K. Skadron, "Fulcrum: A Simplified Control and Access Mechanism Toward Flexible and Practical In-Situ Accelerators," in *HPCA*, 2020.

[90] S. Li, A. O. Glova, X. Hu, P. Gu, D. Niu, K. T. Malladi, H. Zheng, B. Brennan, and Y. Xie, "SCOPE: A Stochastic Computing Engine for DRAM-Based In-Situ Accelerator," in *MICRO*, 2018.

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

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

[93] Y. Li and J. M. Patel, "BitWeaving: Fast Scans for Main Memory Data Processing," in *SIGMOD*, 2013.

[94] X. Lin, C. Zhao, and W. Pan, "Towards Accurate Binary Convolutional Neural Network," in *NIPS*, 2017.

[95] LLVM Project, "Auto-Vectorization in LLVM — LLVM 12 Documentation," https://llvm.org/docs/Vectorizers.html.

[96] G. H. Loh, N. Jayasena, M. Oskin, M. Nutter, D. Roberts, M. Meswani, D. P. Zhang, and M. Ignatowski, "A Processing in Memory Taxonomy and a Case for Studying Fixed-Function PIM," in *WoNDP*, 2013.

[97] B. Lopes, R. Auler, R. Azevedo, and E. Borin, "ISA Aging: A X86 Case Study," in *WIVOSCA*, 2013.

[98] B. C. Lopes, R. Auler, L. Ramos, E. Borin, and R. Azevedo, "SHRINK: Reducing the ISA Complexity via Instruction Recycling," in *ISCA*, 2015.

[99] Y. Luo, S. Govindan, B. Sharma, M. Santaniello, J. Meza, A. Kansal, J. Liu, B. Khessib, K. Vaid, and O. Mutlu, "Characterizing Application Memory Error Vulnerability to Optimize Data Center Cost via Heterogeneous-Reliability Memory," in *DSN*, 2014.

[100] J. Meza, Q. Wu, S. Kumar, and O. Mutlu, "Revisiting Memory Errors in Large-Scale Production Data Centers: Analysis and Modeling of New Trends from the Field," in *DSN*, 2015.

[101] Micron Technology, Inc., "Calculating Memory System Power for DDR3," *Technical Note TN-41-01*, 2015.

[102] N. Muralimanohar, R. Balasubramonian, and N. P. Jouppi, "CACTI 6.0: A Tool to Model Large Caches," HP Laboratories, Tech. Rep. HPL-2009-85, 2009.

[103] O. Mutlu, "The RowHammer Problem and Other Issues We May Face as Memory Becomes Denser," in *DATE*, 2017.

[104] O. Mutlu, S. Ghose, J. Gomez-Luna, and R. Ausavarungnirun, "Processing Data Where It Makes Sense: Enabling In-Memory Computation," *MICPRO*, 2019.

[105] O. Mutlu, S. Ghose, J. Gómez-Luna, and R. Ausavarungnirun, "A Modern Primer on Processing in Memory," in *Emerging Computing: From Devices to Systems - Looking Beyond Moore and Von Neumann*. Springer, 2021.

[106] O. Mutlu and J. S. Kim, "RowHammer: A Retrospective," *TCAD*, 2019.

[107] O. Mutlu and T. Moscibroda, "Stall-Time Fair Memory Access Scheduling for Chip Multiprocessors," in *MICRO*, 2007.

[108] O. Mutlu and T. Moscibroda, "Parallelism-Aware Batch Scheduling: Enhancing Both Performance and Fairness of Shared DRAM Systems," in *ISCA*, 2008.

[109] L. Nai, R. Hadidi, J. Sim, H. Kim, P. Kumar, and H. Kim, "GraphPIM: Enabling Instruction-Level PIM Offloading in Graph Computing Frameworks," in *HPCA*, 2017.

[110] NIMO Group, Arizona State Univ., "Predictive Technology Model," http://ptm.asu.edu/, 2012.

[111] NVIDIA Corp., "NVIDIA Management Library (NVML)," https://developer.nvidia.com/nvidia-management-library-nvml.

[112] NVIDIA Corp., "NVIDIA Titan V," https://www.nvidia.com/en-us/titan/titan-v/.

[113] G. F. Oliveira, P. C. Santos, M. A. Z. Alves, and L. Carro, "NIM: An HMC-Based Machine for Neuron Computation," in *ARC*, 2017.

[114] G. F. Oliveira, J. Gómez-Luna, L. Orosa, S. Ghose, N. Vijaykumar, I. Fernandez, M. Sadrosadati, and O. Mutlu, "A New Methodology and Open-Source Benchmark Suite for Evaluating Data Movement Bottlenecks: A Near-Data Processing Case Study," in *SIGMETRICS*, 2021.

[115] OpenACC Organization, "The OpenACC® Application Programming Interface, Version 3.1," 2020.

[116] Y. Park, W. Kwon, E. Lee, T. J. Ham, J. H. Ahn, and J. W. Lee, "Graphene: Strong yet Lightweight Row Hammer Protection," in *MICRO*, 2020.

[117] M. Patel, J. Kim, H. Hassan, and O. Mutlu, "Understanding and Modeling On-Die Error Correction in Modern DRAM: An Experimental Study Using Real Devices," in *DSN*, 2019.

[118] M. Patel, J. Kim, T. Shahroodi, H. Hassan, and O. Mutlu, "Bit-Exact ECC Recovery (BEER): Determining DRAM On-Die ECC Functions by Exploiting DRAM Data Retention Characteristics," in *MICRO*, 2020.

[119] A. Pattnaik, X. Tang, A. Jog, O. Kayiran, A. K. Mishra, M. T. Kandemir, O. Mutlu, and C. R. Das, "Scheduling Techniques for GPU Architectures with Processing-in-Memory Capabilities," in *PACT*, 2016.

[120] J. Picorel, D. Jevdjic, and B. Falsafi, "Near-Memory Address Translation," in *PACT*, 2017.

[121] M. Poletto and V. Sarkar, "Linear Scan Register Allocation," *TOPLAS*, 1999.

[122] S. H. Pugsley, J. Jestes, H. Zhang, R. Balasubramonian, V. Srinivasan, A. Buyuktosunoglu, A. Davis, and F. Li, "NDC: Analyzing the Impact of 3D-Stacked Memory+Logic Devices on MapReduce Workloads," in *ISPASS*, 2014.

[123] Rambus Inc., "Rambus Power Model," https://www.rambus.com/energy/.

[124] M. Rastegari, V. Ordonez, J. Redmon, and A. Farhadi, "XNOR-Net: ImageNet Classification Using Binary Convolutional Neural Networks," in *ECCV*, 2016.

[125] P. C. Santos, G. F. Oliveira, J. P. Lima, M. A. Z. Alves, L. Carro, and A. C. S. Beck, "Processing in 3D Memories to Speed Up Operations on Complex Data Structures," in *DATE*, 2018.

[126] P. C. Santos, G. F. Oliveira, D. G. Tomé, M. A. Z. Alves, E. C. Almeida, and L. Carro, "Operand Size Reconfiguration for Big Data Processing in Memory," in *DATE*, 2017.

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

[128] V. Seshadri, A. Bhowmick, O. Mutlu, P. B. Gibbons, M. A. Kozuch, and T. C. Mowry, "The Dirty-Block Index," in *ISCA*, 2014.

[129] V. Seshadri, K. Hsieh, A. Boroumabd, D. Lee, M. A. Kozuch, O. Mutlu, P. B. Gibbons, and T. C. Mowry, "Fast Bulk Bitwise AND and OR in DRAM," *CAL*, 2015.

[130] V. Seshadri, Y. Kim, C. Fallin, D. Lee, R. Ausavarungnirun, G. Pekhimenko, Y. Luo, O. Mutlu, P. B. Gibbons, M. A. Kozuch, and T. C. Mowry, "RowClone: Fast and Energy-Efficient in-DRAM Bulk Data Copy and Initialization," in *MICRO*, 2013.

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

[132] V. Seshadri and O. Mutlu, "The Processing Using Memory Paradigm: In-DRAM Bulk Copy, Initialization, Bitwise AND and OR," arXiv:1610.09603 [cs.AR], 2016.

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

[134] V. Seshadri and O. Mutlu, "In-DRAM Bulk Bitwise Execution Engine," arXiv:1905.09822 [cs.AR], 2019.

[135] S. H. SeyyedAghaei Rezaei, R. Ausavarungnirun, M. Sadrosadati, O. Mutlu, and M. Daneshtalab, "NoM: Network-on-Memory for Inter-Bank Data Transfer in Highly-Banked Memories," *CAL*, 2020.

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

[137] W. Shooman, "Parallel Computing with Vertical Data," in *EJCC*, 1960.

[138] K. Simonyan and A. Zisserman, "Very Deep Convolutional Networks for Large-Scale Image Recognition," arXiv:1409.1556 [cs.CV], 2014.

[139] M. Soeken, S. Shirinzadeh, P.-E. Gaillardon, L. G. Amarú, R. Drechsler, and G. De Micheli, "An MIG-Based Compiler for Programmable Logic-in-Memory Architectures," in *DAC*, 2016.

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

[141] L. Song, Y. Zhuo, X. Qian, H. Li, and Y. Chen, "GraphR: Accelerating Graph Processing Using ReRAM," in *HPCA*, 2018.

[142] S. Srinath, O. Mutlu, H. Kim, and Y. N. Patt, "Feedback Directed Prefetching: Improving the Performance and Bandwidth-Efficiency of Hardware Prefetchers," in *HPCA*, 2007.

[143] A. Subramaniyan and R. Das, "Parallel Automata Processor," in *ISCA*, 2017.

[144] The Open Group, "The Single UNIX Specification, Version 2," https://pubs.opengroup.org/onlinepubs/7908799/xsh/systime.h.html, 1997.

[145] Transaction Processing Performance Council, "TPC-H," http://www.tpc.org/tpch/.

[146] L. W. Tucker and G. G. Robertson, "Architecture and Applications of the Connection Machine," *Computer*, 1988.

[147] T. Vogelsang, "Understanding the Energy Consumption of Dynamic Random Access Memories," in *MICRO*, 2010.

[148] Y. Wang, L. Orosa, X. Peng, Y. Guo, S. Ghose, M. Patel, J. S. Kim, J. G. Luna, M. Sadrosadati, N. M. Ghiasi, and O. Mutlu, "FIGARO: Improving System Performance via Fine-Grained In-DRAM Data Relocation and Caching," in *MICRO*, 2020.

[149] X. Xin, Y. Zhang, and J. Yang, "ELP2IM: Efficient and Low Power Bitwise Operation Processing in DRAM," in *HPCA*, 2020.

[150] A. G. Yağlıkçı, M. Patel, J. S. Kim, R. Azizibarzoki, A. Olgun, L. Orosa, H. Hassan, J. Park, K. Kanellopoullos, T. Shahroodi, S. Ghose, and O. Mutlu, "BlockHammer: Preventing RowHammer at Low Cost by Blacklisting Rapidly-Accessed DRAM Rows," in *HPCA*, 2021.

[151] Y. Zha and J. Li, "Hyper-AP: Enhancing Associative Processing Through A Full-Stack Optimization," in *ISCA*, 2020.

[152] D. Zhang, N. Jayasena, A. Lyashevsky, J. L. Greathouse, L. Xu, and M. Ignatowski, "TOP-PIM: Throughput-Oriented Programmable Processing in Memory," in *HPDC*, 2014.

[153] Q. Zhu, T. Graf, H. E. Sumbul, L. Pileggi, and F. Franchetti, "Accelerating Sparse Matrix-Matrix Multiplication with 3D-Stacked Logic-in-Memory Hardware," in *HPEC*, 2013.

[154] W. Zuravleff and T. Robinson, "Controller for a Synchronous DRAM That Maximizes Throughput by Allowing Memory Requests and Commands to Be Issued Out of Order," U.S. Patent No. 5,630,096, 1997.

APPENDIX

A AIG-to-MIG Conversion

The conversion from AND/OR/NOT representation of an operation to its MAJ/NOT representation relies on a set of transformation rules that are derived from the characteristics of the MAJ operation. Table 4 lists the set of transformation rules that we use to synthesize a circuit for a desired operation with MAJ and NOT gates. We use full addition as a running example to describe the process of synthesizing a MAJ/NOT-based circuit, starting from an AND/OR/NOT representation of the circuit and using the transformation rules. We obtain MAJ/NOT-based circuits for other SIMDRAM operations following the same method. In a later step (§4.2), we translate a MAJ/NOT-based circuit to sequences of AAPs/APs operations.

Table 4: MAJ/NOT transformation Rules [7].

Commutativity (C)	$M(x, y, z) = M(y, x, z) = M(z, y, x)$
Majority (M)	if$(x = y) : M(x, y, z) = x = y$ if$(x = \overline{y}) : M(x, y, z) = z$
Associativity (A)	$M(x, u, M(y, u, z)) = M(z, u, M(y, u, x))$
Distributivity (D)	$M(x, y, M(u, v, z)) = M(M(x, y, u), M(x, y, v), z)$
Inverter Propagation (I)	$\overline{M}(x, y, z) = M(\overline{x}, \overline{y}, \overline{z})$
Relevance (R)	$M(x, y, z) = M(x, y, z_{x/\overline{y}})$
Complementary Associativity (CA)	$M(x, u, M(y, \overline{u}, z)) = M(x, u, M(y, x, z))$

Fig. 15a shows the optimized AND/OR/Inverter (i.e., AND/OR/NOT) Graph (AOIG) representation of a full addition (i.e., $F = A + B + C_{in}$). As shown in Fig. 15b, the naive way to transform the AOIG to a Majority/Inverter (i.e., MAJ/NOT) Graph (MIG) representation, is to replace every AND and OR primitive with a three-input MAJ primitive where the third input is 0 or 1, respectively. The resulting MIG is in fact Ambit's [131] representation of the full addition. While the AOIG in Fig. 15a is optimized for AND/OR/NOT operations, the resulting MIG in

Fig. 15b can be further optimized by exploiting the transformation rules of the MAJ primitive (Table 4, replicated from [7]). The MIG optimization is performed in two key steps: (1) node reduction, and (2) MIG reshaping.

Node reduction. In order to optimize the MIG in Fig. 15b, the first step is to reduce the number of MAJ nodes in the MIG. As shown in Table 1, rules **M** and **D** reduce the number of nodes in a MIG if applied from left to right (i.e., $M_{L \to R}$) and from right to left (i.e., $D_{R \to L}$), respectively. $M_{L \to R}$ replaces a MAJ node with a single value, and $D_{R \to L}$ replaces three MAJ nodes with two MAJ nodes in the MIG. The node reduction step applies $M_{L \to R}$ and $D_{R \to L}$ as many times as possible to reduce the the number of MAJ operations in the MIG. We can see in Fig. 15b that none of the two rules are applicable in the particular case of the full addition MIG. Therefore, Fig 15b remains unchanged after applying node reduction.

MIG reshaping. When no further node reduction is possible, we *reshape* the MIG in an effort to enable more node reduction opportunities by repeatedly using two sets of rules: (1) rules $M_{R \to L}$, $D_{L \to R}$, and **R** to temporarily inflate the MIG and create more node reduction opportunities with the help of the new nodes, and (2) rules **A** and **CA**, to exchange variables between adjacent nodes. Note that in this step, rules **M** and **D** are applied in the reverse direction compared to the previous step (i.e., node reduction step) which results in increasing the number of nodes in the MIG. We now describe the MIG reshaping process for the full addition example (Fig. 15b). For simplicity, we first assume that the entire MIG is represented as function **F** that computes the full addition of the input operands **A** and **B**. Then, we apply rule $M_{R \to L}$ while introducing variable **X** to the MIG (as $F = M(F, x, \overline{x})$) without impacting the functionality of the MIG (Fig. 15c). We then apply the same rule again, and replace **X** with a new MAJ node while introducing variable **Y** (Fig. 15d). Next, by applying rule $D_{L \to R}$, we introduce a
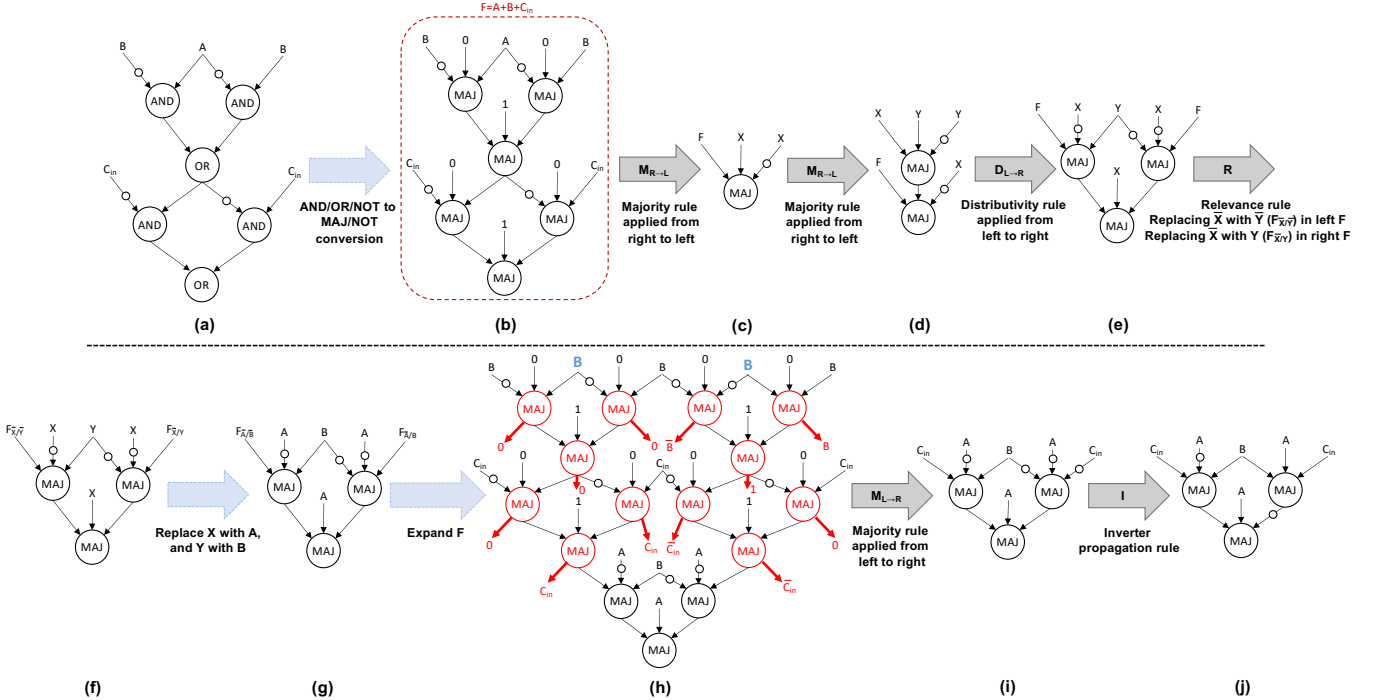

Figure 15: Synthesizing SIMDRAM circuit for a full addition.

new MAJ node and distribute the function **F** across the two MAJ nodes (Fig. 15e). Now, by applying rule **R** to the function **F** on the left, variable $\overline{\mathbf{X}}$ is replaced with variable $\overline{\mathbf{Y}}$ in the function **F** on the left. Similarly, by applying rule **R** to the function **F** on the right, variable $\overline{\mathbf{X}}$ is replaced with variable **Y** in the function **F** on the right (Fig. 15f). At this point, since rule $\mathbf{M}_{R \rightarrow L}$ holds with any given two variables, we can safely replace **X** and **Y** with variables **A** and **B**, respectively (Figure 15g). Next, we expand function **F** (Fig. 15h) and the variables replaced as a result of the previous rule are highlighted in blue. As shown in Fig. 15h, the resulting graph after expanding function **F** has multiple node reduction opportunities using rule $\mathbf{M}_{L \rightarrow R}$ and starting from the top of the graph. The nodes that can be eliminated using this rule are marked in red and the replacing value is indicated with a red arrow leaving the node. Fig. 15i shows the same MIG after resolving all the node reductions. We next use rule **I** to remove all three NOT primitives in the rightmost MAJ node. The final optimized MIG that is shown in Figure 15j requires only 3 MAJ primitives to perform the full addition operation, as opposed to the 6 we started with (in Fig. 15b).

The node reduction step followed by the MIG reshaping step are repeated (for a predefined number of times) until we achieve an optimized MIG that requires minimal number of MAJ operations to perform the desired in-DRAM operation. The process of converting an operation to a MAJ-based implementation can be automated as suggested by prior work [7, 139].

B Row-to-Operand Allocation

Algorithm 1 describes SIMDRAM's row-to-operand allocation procedure. To enable in-DRAM computation, our allocation algorithm copies (i.e., maps) input operands for each MAJ node in the MIG from D-group rows (where the operands normally reside) into compute rows. However, due to the limited number of compute rows, the allocation algorithm cannot allocate DRAM rows to all input operands from all MAJ nodes at once. To address this issue, the allocation algorithm divides the allocation process into *phases*. Each phase allocates as many compute rows to operands as possible. For example, because no rows are allocated yet, the initial phase (Phase 0) has all six compute rows available for allocation (i.e., the rows are vacant), and can allocate up to six input operands to the compute rows. A phase is considered finished when either (1) there are not enough vacant compute rows to allocate all input operands for the next logic primitive that needs to be computed, or (2) there are no more MAJ primitives left to process in the MIG. The phase information is used when generating the µProgram for the MIG in Task 2 of Step 2 of SIMDRAM framework (§4.2.3), where µOps for all MAJ primitives in phase i are generated prior to the MAJ primitives in phase $i + 1$. Knowing that all the MAJ primitives in phase i are performed before the next phase $i + 1$ starts, the allocation algorithm can safely reuse the compute rows for use in phase $i + 1$, without worrying about the output of a MAJ primitive being overwritten by a new row-to-operand allocation.

We now describe the row-to-operand allocation algorithm in detail, using the MIG for full addition in Fig. 5a as an example of a MIG being traversed by the algorithm. The allocation algorithm starts at Phase 0. Throughout its execution, the algorithm maintains (1) the list of free compute rows (rows in the B-group of the subarray, shown in Fig. 2) that are available for allocation (B_rows and

B_rows_DCC in Algorithm 1, initialized in lines 3–4); and (2) the list of row-to-operand allocations associated with each MAJ node, tagged with the phase number that the allocations were performed in ($row\_operand\_allocation$ in Algorithm 1). Once a row-to-operand allocation is performed, the algorithm removes the compute row used for the allocation from the list of the free compute rows, and adds the new allocation to the list of row-to-operand allocations generated in that phase for the corresponding MAJ node. The algorithm follows a simple procedure to allocate compute rows to the input operands of the MAJ nodes in the MIG. The algorithm does a topological traversal starting with the leftmost MAJ node in the highest level of the MIG (e.g., Level 0 in Fig. 5a), and traverses all the MAJ nodes in each level, before moving to the next lower level of the graph.

Algorithm 1 SIMDRAM's Row-to-Operand Allocation Algorithm.

```
1: Input: MIG G = (V, E)                  ▷ Majority-Inverter Graph G nodes <vertex, edge>
2: Output: row_operand_allocation                      ▷ Allocation map of rows to operands

3: B_rows ← {T0, T1, T2, T3}
4: B_rows_DCC ← {DCC0, DCC1}
5: phase ← 0
6: row_operand_allocation_map ← ∅

7: for each level in G do
8:   for each V in G[level] do
9:     for each input edge in E[V] do
10:       Search for input edge's parent
11:       if input edge has no parents then
12:         if input edge is negated then
13:           Allocate row in B_rows_DCC to input edge
14:           Remove allocated row from B_rows_DCC
15:         else
16:           Allocate row in B_rows to input edge
17:           Remove allocated row from B_rows
18:       else
19:         if input edge is negated then
20:           Map allocated parent row in B_rows_DCC to input edge
21:         else
22:           Map allocated parent row in B_rows to input edge
23:       if B_rows and B_rows_DCC are empty then
24:         phase ← phase + 1
25:         B_rows ← {T0, T1, T2, T3}
26:         B_rows_DCC ← {DCC0, DCC1}
27:       row_operand_allocation ← (input edge, allocated row, phase)
```

Case 1 (lines 11–17) · Case 2 (lines 18–22) · Case 3 (lines 23–26)

For each of the three input edges (i.e., operands) of any given MAJ node, the algorithm checks for the following three possible cases and performs the allocation accordingly:

Case 1: if the edge is not connected to another MAJ node in a higher level of the graph (line 11 in Algorithm 1), i.e., the edge does not have a parent (e.g., the three edges entering the blue node in Fig. 5a), and a compute row is available, the input operand associated with the edge is considered to be a source input, and is currently located in the D-group rows of the subarray. As a result, the algorithm copies the input operand associated with the edge from its D-group row to the first available compute row. Note that if the edge *is* complemented, i.e., the input operand is negated (e.g., the edge with operand A for the blue node in Fig. 5a), the algorithm allocates the first available compute row with dual contact cells (DCC0 or DCC1) to the input operand of the edge (lines 12–14 in Algorithm 1). If the edge is *not* complemented (e.g., the edge with operand B for the blue node in Fig. 5a), a regular compute row is allocated to the input operand (lines 15–17 in Algorithm 1).

Case 2: if the edge is connected to another MAJ node in a higher level of the graph (line 18 in Algorithm 1), the edge has a parent node and the value of the input operand associated with the edge equals the result of the parent node, which is available in the compute rows that hold the result of the parent MAJ node. As a result, the

algorithm maps the input operand of the edge to a compute row that holds the result of its parent node (lines 19–22 in Algorithm 1). **Case 3:** if there are no free compute rows available, the algorithm considers the phase as *complete* and continues the allocations in the next phase (lines 23–26 in Algorithm 1).

Once DRAM rows are allocated to all the edges connected to a MAJ node, the algorithm stores the row-to-operand allocation information of the three input operands of the MAJ node in *row_operand_allocation* (line 27 in Algorithm 1) and associates this information with the MAJ node and the phase number that the allocations were performed in. The algorithm finishes once DRAM rows are allocated to *all* the input operands of all the MAJ nodes in the MIG. Fig. 5b shows these allocations as the output of Task 1 for the full addition example. The resulting *row_operand_allocation* is then used in Task 2 of Step 2 of the SIMDRAM framework (§4.2.3) to generate the series of μOps to compute the operation that the MIG represents.

C   Scalability of Operations

Table 5 lists the semantics and the total number of AAP/APs required for each of the 16 SIMDRAM operations that we evaluate in this paper (§7) for input element(s) of size *n*. Each operation is classified based on how the latency of the operation scales with respect to the element size *n*. Class 1, 2, and 3 operations scale linearly, logarithmically, and quadratically with *n*, respectively.

Table 5: Evaluated SIMDRAM operations (for *n*-bit data).

Type	Operation	# AAPs/APs	Class	Semantics
Arithmetic	abs	$10n - 2$	Linear	$dst = (src > 0)?\ src : -(src)$
	addition	$8n + 1$	Linear	$dst = src_1 + src_2$
	bitcount	$\Omega = 8n - 8\log_2(n+1)$	Linear	$\sum_{i=0}^{n} src(i)$
		$O = 8n$		
	division	$8n^2 + 12n$	Quadratic	$dst = \frac{src_1}{src_2}$
	max	$10n + 2$	Linear	$dst = (src_1 > src_2)?\ src_1 : src_2$
	min	$10n + 2$	Linear	$dst = (src_1 < src_2)?\ src_1 : src_2$
	multiplication	$11n^2 - 5n - 1$	Quadratic	$dst = src_1 \times src_2$
	ReLU	$3n + ((n-1)\ mod\ 2)$	Linear	$dst = (src \geq 0)?\ src : 0$
	subtraction	$8n + 1$	Linear	$dst = src_1 - src_2$
Predication	if_else	$7n$	Linear	$dst = (sel)?\ src_1 : src_2$
Reduction	and_reduction	$5\lfloor \frac{n}{2} \rfloor + 2$	Logarithmic	$Y = src(1) \wedge src(2) \wedge src(3)$
	or_reduction	$5\lfloor \frac{n}{2} \rfloor + 2$	Logarithmic	$Y = src(1) \vee src(2) \vee src(3)$
	xor_reduction	$6\lfloor \frac{n}{2} \rfloor + 1$	Logarithmic	$Y = src(1) \oplus src(2) \oplus src(3)$
Relational	equal	$4n + 3$	Linear	$dst = (src_1 == src_2)$
	greater	$3n + 2$	Linear	$dst = (src_1 > src_2)$
	greater_equal	$3n + 2$	Linear	$dst = (src_1 \geq src_2)$

D   Evaluated Real-World Applications

Convolutional Neural Networks (CNNs). CNNs [52, 77, 124] are used in many classification tasks such as image and handwriting classification. CNNs are often computationally intensive as they use many general-matrix-multiplication (GEMM) operations using floating-point operations for each convolution. Prior works [52, 94, 124] demonstrate that instead of the costly floating-point multiplication operations, convolutions can be performed using a series of bitcount, addition, shift, and XNOR operations. In this work, we use the XNOR-NET [124] implementations of VGG-13, VGG-16, and LeNET provided by [52], to evaluate the functionality of SIMDRAM. We modify these implementations to make use of SIMDRAM's bitcount, addition, shift, and XNOR operations. We evaluate all three networks for inference using two different datasets: VGG-13 and VGG-16 (using CIFAR-10 [76]), and LeNet-5 (using MNIST [24]).

k-Nearest Neighbor Classifier (kNN). We use a kNN classifier to solve the handwritten digits recognition problem [80]. The kNN classifier finds a group of k objects in the input set using a simple distance algorithm such as Euclidean distance [35]. In our evaluations, we use SIMDRAM to implement the Euclidean distance algorithm entirely in DRAM. We evaluate a kNN algorithm using the MNIST dataset [24] with 3000 training images and 1000 testing images. We quantize the inputs using an 8-bit representation.

Database. We evaluate SIMDRAM using two different database workloads. First, we evaluate a simple table scan query 'select count(*) from T where c1 <= val <= c2' using the BitWeaving algorithm [93]. Second, we evaluate the performance of the TPC-H [145] scheme using query 01, which executes many arithmetic operations, including addition and multiplication. For our evaluation, we follow the column-based data layout employed in [126] and use a scale factor of 100.

Brightness. We use a simple image brightness algorithm [33] to demonstrate the benefits of the SIMDRAM predication operation. The algorithm evaluates if a given brightness value is larger than 0. If so, it increases the pixel value of the image by the brightness value. Before assigning the new brightness value to the pixel, the algorithm verifies if the new pixel value is between 0 and 255. In our SIMDRAM implementation, we use both addition and predication operations.