Proteus: Achieving High-Performance Processing-Using-DRAM with Dynamic Bit-Precision, Adaptive Data Representation, and Flexible Arithmetic

Geraldo F. Oliveira[†] Mayank Kabra[†] Yuxin Guo[‡] Kangqi Chen[†]
A. Giray Yağlıkçı[†] Melina Soysal[†] Mohammad Sadrosadati[†]
Joaquin O. Bueno[⋆] Saugata Ghose[▽] Juan Gómez-Luna[§] Onur Mutlu[†]

[†] *ETH Zürich* [‡] *Cambridge University* [⋆] *Universidad de Córdoba*
[▽] *Univ. of Illinois Urbana-Champaign* [§] *NVIDIA Research*

Abstract

Processing-using-DRAM (PUD) is a paradigm where the analog operational properties of DRAM are used to perform bulk logic operations. While PUD promises high throughput at low energy and area cost, we uncover three limitations of existing PUD approaches that lead to significant inefficiencies: (*i*) static data representation, i.e., two's complement with fixed bit-precision, leading to *unnecessary computation* over useless (i.e., inconsequential) data; (*ii*) support for *only* throughput-oriented execution, where the high latency of individual PUD operations can *only* be hidden in the presence of bulk data-level parallelism; and (*iii*) high latency for high-precision (e.g., 32-bit) operations.

To address these issues, we propose *Proteus*, the first hardware framework that addresses the high execution latency of bulk bitwise PUD operations by implementing a data-aware runtime engine for PUD. *Proteus* reduces the latency of PUD operations in three different ways: (*i*) *Proteus dynamically* reduces the bit-precision (and thus the latency and energy consumption) of PUD operations by exploiting narrow values (i.e., values with many leading zeros or ones); (*ii*) *Proteus concurrently executes* independent in-DRAM primitives belonging to a *single* PUD operation across *multiple* DRAM arrays; (*iii*) *Proteus chooses and uses* the most appropriate data representation and arithmetic algorithm implementation for a given PUD instruction *transparently* to the programmer.

We compare *Proteus* to different state-of-the-art computing platforms (CPU, GPU, and the SIMDRAM PUD architecture) for twelve real-world applications. Even when using only a *single* DRAM bank, *Proteus* provides (*i*) 17×, 7.3×, and 10.2× higher performance per mm^2; and (*ii*) 90.3×, 21×, and 8.1× lower energy consumption than CPU, GPU, and SIMDRAM, respectively. We open-source *Proteus* at https://github.com/CMU-SAFARI/Proteus.

ICS '25, June 8–11, 2025, Salt Lake City, UT, USA

ACM ISBN 979-8-4007-1537-2/2025/06
https://doi.org/10.1145/3721145.3730420

CCS Concepts

• **Computer systems organization** → **Other architectures**; • **Hardware** → **Memory and dense storage**.

Keywords

processing-in-memory, memory systems, DRAM, energy efficiency, parallelism

ACM Reference Format:
Geraldo F. Oliveira, Mayank Kabra, Yuxin Guo, Kangqi Chen, A. Giray Yağlıkçı, Melina Soysal, Mohammad Sadrosadati, Joaquin Olivares Bueno, Saugata Ghose, Juan Gómez-Luna, Onur Mutlu . 2025. *Proteus*: Achieving High-Performance Processing-Using-DRAM with Dynamic Bit-Precision, Adaptive Data Representation, and Flexible Arithmetic . In *2025 International Conference on Supercomputing (ICS '25), June 8–11, 2025, Salt Lake City, UT, USA.* ACM, New York, NY, USA, 20 pages. https://doi.org/10.1145/3721145.3730420

1 Introduction

Processing-in-memory (PIM) [1–12] aims to alleviate the ever-growing cost of moving data between computing units (e.g., CPU, GPU) and memory units (e.g., DRAM). In PIM architectures, computation is done by adding logic *near* memory arrays, i.e., processing-near-memory (PNM) [13–98], or by *using* the analog operational properties of the memory arrays, i.e., processing-using-memory (PUM) [66, 99–139]). Prior works [66, 101–107, 110, 114–117, 119, 120, 126, 129, 130, 132, 133, 140, 141] show the potential and feasibility of processing-using-DRAM (PUD), by using DRAM circuitry to implement in-DRAM row copy [104, 110, 130, 142], Boolean [101, 103, 107, 117], and arithmetic [103, 106, 119, 120, 131–133, 141, 143–149] operations. PUD systems often employ a *bulk bit-serial* execution model [102], where each Boolean primitive operates across entire DRAM rows, with each row containing one bit from many input operands. The predefined sequence of DRAM commands that implements an operation are stored in a *μProgram* [143].

We uncover three shortcomings that significantly limit the performance and efficiency of PUD architectures. First, they employ a **rigid and static data representation**, which is *inefficient* for bit-serial execution. Existing PUD engines typically employ a *fixed* bit-precision, statically-defined (commonly two's complement) data representation for *all* PUD

operations. This rigid and static data format introduces inefficiencies in a bit-serial execution model, where bits of a data word are *individually* and *sequentially* processed. Since many applications store data in data representation formats that exceed the necessary precision [150–159] (e.g., 8-bit values stored in a 32-bit integer), a significant fraction of PUD computation is wasted on processing inconsequential bits, such as leading zeros or ones (e.g., sign-extension bits), causing significant latency and energy overhead. Second, PUD architectures provide *only* a **throughput-oriented execution model with limited latency tolerance**. PUD operations are composed of bitwise (bit-serial) in-DRAM primitives (e.g., in-DRAM majority and `NOT` [101, 117]), making individual PUD operations inherently slow due to the need to operate on each bit serially to perform an operation with a data width larger than one. To compensate for this latency, PUD architectures adopt a *throughput-oriented execution model* that distributes large amounts of data across multiple DRAM subarrays and DRAM banks. However, this approach is effective *only* when sufficient data-level parallelism is available to amortize the high latency of bit-serial execution of an individual PUD primitive. In scenarios where data-level parallelism is limited, this throughput-oriented execution model fails to hide the latency of individual in-DRAM primitives, potentially leading to performance degradation [141]. Third, bit-serial PUD architectures face **scalability challenges for high-precision operations**. PUD systems suffer from increased latency as the bit-precision grows [143]. Due to their bit-serial nature, the latencies of arithmetic PUD operations scale linearly or quadratically with the bit-precision [143].

Our *goal* in this work is to overcome the three limitations of PUD architectures that stem from the naive use of a bit-serial execution model. To this end, we propose *Proteus*,[1] an efficient data-aware runtime framework that *dynamically* adjusts the bit-precision and, based on that, *chooses* and *uses* the most appropriate data representation and arithmetic algorithm implementation for a given PUD operation. *Proteus* builds on three *key ideas*. To solve the **first limitation** (i.e., rigid and static data representation), *Proteus reduces* the bit-precision for PUD operations by leveraging *narrow values* (i.e., values with many leading zeros or ones). As several works observe [150–159], programmers often overprovision the bit-precision used to store operands, using large data types (e.g., a 32-bit or 64-bit integer) to store small (i.e., narrow, e.g., 4-bit, 8-bit) values. Based on this observation, *Proteus* exploits dynamic narrow values to reduce the bit-precision of a PUD operation to that of the best-fitting number of bits, thereby avoiding costly in-DRAM operations on inconsequential bits, which improves overall performance and energy efficiency.

To solve the **second limitation** (i.e., throughput-oriented execution with limited latency tolerance), *Proteus parallelizes* the execution of *independent* in-DRAM primitives in a PUD operation by leveraging DRAM's internal organization combined with *bit-level parallelism*. We make the *key observation* that many in-DRAM primitives that compose a PUD operation (e.g., an in-DRAM addition) can be executed *concurrently* across different bits of a data word. For example, executing an n-bit in-DRAM addition (i.e., $\{A_{n-1}, \ldots, A_0\}+\{B_{n-1}, \ldots, B_0\}$) in a bit-serial manner requires serially performing three majority-of-three (`MAJ3`) operations per bit i to compute the *sum* and propagate the carry to bit i+1. However, only one of these operations (i.e., the carry propagation from bit i to bit $i+1$) truly requires serialization, while the other two `MAJ3` operations can be *concurrently executed across* the n bit positions of a data word. To exploit this observation, *Proteus* scatters the n bits of a data word across *multiple* DRAM subarrays (i.e., $subarray_0 \leftarrow \{A_0, B_0\}, \ldots, subarray_{n-1} \leftarrow \{A_{n-1}, B_{n-1}\}$) and employs subarray-level parallelism (SALP) [161] to enable each subarray i to *concurrently* execute the in-DRAM primitive associated with bit i, thereby hiding the high latency of individual in-DRAM primitives in a PUD operation over the many bits of the target data word. To propagate intermediate data (e.g., carry bits) across DRAM subarrays, *Proteus* leverages LISA [162], a low-cost DRAM design that enables fast inter-subarray data movement at DRAM row granularity.

To solve the **third limitation** (i.e., scalability challenges for high-precision operations), *Proteus* exploits an alternative data representation for high-precision computation. Concretely, we use the *redundant binary representation (RBR)* [163–167] (where multiple-digit combinations represent the same value), for high-precision (e.g., 32-bit or 64-bit) PUD computations. PUD execution can take advantage of two properties of RBR-based arithmetic: (*i*) operations no longer need to propagate carry bits through the full width of the data (e.g., RBR-based addition limits carry propagation to at most two digits [168]), and (*ii*) operation latency is *independent* of the bit-precision.

Based on these three key ideas, we design *Proteus* as a three-component hardware runtime framework for high-performance PUD computation that *transparently* (from the user/programmer) selects, for a given PUD operation, the (*i*) bit-precision, (*ii*) fastest arithmetic algorithm for latency- or throughput-oriented PUD execution, and (*iii*) most efficient data format (e.g., two's complement or redundant binary [163–167]). First, we build a *Parallelism-Aware μProgram Library*, consisting of hand-tuned algorithmic implementations of key arithmetic operations that take into account

[1] *Proteus* is a shape-shifting, prophetic sea god from Greek mythology [160], known for his ability to elude capture by changing forms. Our *Proteus changes* the bit-precision of PUD operations to improve performance.

SALP [161] to implement PUD operations using (*i*) both bit-serial and bit-parallel algorithms and (*ii*) both two's complement and RBR data representation formats. The *Parallelism-Aware μProgram Library* contains a collection of possible implementations of PUD operations, each of which with different predetermined latency and energy requirements that depend on a given bit-precision. The latency and energy each μProgram consumes is stored within an easily-accessible *Pre-Loaded Cost Model Lookup Tables (LUTs)* alongside the μProgram in the *Parallelism-Aware μProgram Library*.

Second, we devise a new *Dynamic Bit-Precision Engine* to identify the appropriate initial bit-precision for a given PUD operation. We implement the *Dynamic Bit-Precision Engine* by augmenting prior works' *Data Transposition Unit* [141, 143]. Before PUD execution, the *Data Transposition Unit* captures and transposes (from the standard horizontal data layout to the PUD vertical data layout) cache lines that are about to be evicted from the last-level cache (LLC) to DRAM and belong to a PUD memory object, which is *previously-identified* based on its memory address range. During this process, our *Dynamic Bit-Precision Engine scans* the content of the evicted cache line to identify the largest value belonging to a PUD memory object. Third, when a PUD operation is issued, the *μProgram Select Unit* probes the *Dynamic Bit-Precision Engine* to identify the most suitable bit-precision for the PUD operation and, based on the *Pre-Loaded Cost Model LUTs* within the *μProgram Select Unit*, selects the best performing μProgram from the *Parallelism-Aware μProgram Library*.

Key Results. We compare *Proteus* to different state-of-the-art computing platforms (state-of-the-art CPU, GPU, and SIMDRAM [143]). We comprehensively evaluate *Proteus*' performance for twelve real-world applications [169–172]. Even when using only a *single* DRAM bank, *Proteus* provides (*i*) 17×, 7.3×, and 10.2× higher performance per mm^2; and (*ii*) 90.3×, 21×, and 8.1× lower energy consumption than CPU, GPU, and SIMDRAM, respectively, on average across all twelve real-world applications. *Proteus* incurs low area cost on top of a DRAM chip (1.6%) and CPU die (0.03%).

We make the following major contributions:

- We identify three major shortcomings of existing bit-serial PUD architectures: (*i*) rigid and static data representation, (*ii*) throughput-oriented execution model with limited latency tolerance, and (*iii*) scalability challenges for high-precision operations.
- We propose *Proteus*, a three-component data-aware hardware runtime framework for high-performance PUD computation that *transparently* (from the programmer), for a given PUD operation, *dynamically* adjusts the bit-precision, and based on that, *chooses* and *uses* the most appropriate data representation (e.g., two's complement or RBR) and the most appropriate arithmetic algorithm implementation for latency- or throughput-oriented PUD execution.
- We extensively evaluate *Proteus* for twelve real-world applications, showing that *Proteus* outperforms a state-of-the-art PUD framework (SIMDRAM), CPU, and GPU while incurring low area cost to the system.
- We open-source *Proteus* at https://github.com/CMU-SAFARI/Proteus. An extended version of this paper is available at [173].

2 Background

2.1 DRAM Organization & Operation

DRAM Organization. Fig. 1 shows the hierarchy of a DRAM system. A *DRAM module* (Fig. 1a) has several (e.g., 8–16) DRAM chips. A *DRAM chip* (Fig. 1b) has multiple banks (e.g., 8–16). A *DRAM bank* (Fig. 1c) has (*i*) multiple (e.g., 64–128) 2D arrays of DRAM cells known as *DRAM subarrays* [101, 102, 104, 161, 162, 174–198]; (*ii*) a *global row decoder* and a *global address latch* that select a row of cells in a subarray through *global wordlines*; (*iii*) *column select logic* (CSL) that selects portions (e.g., 64-bit) of the row; and (*iv*) a set of *global sense amplifiers* (GSAs) [102, 105, 162, 182, 187, 193, 199, 200], also sometimes called the global row buffer [120, 141, 161, 201–204], that transfers the selected fraction of the data from the row through *global bitlines*. Each subarray (Fig. 1d) contains (*i*) multiple rows (e.g., 512–1024) and columns (e.g., 2–8 kB [199, 204, 205]) of DRAM cells, (*ii*) a *local row decoder* that activates a *local wordline*, and (*iii*) a *local row buffer* containing a row of *sense amplifiers* (SAs; ❶ in Fig. 1d) to latch data from an activated row. A DRAM cell (❷) consists of an access capacitor, which connects a transistor that stores the data value with a *local bitline* shared by all cells in the same column. Modern DRAM employs an *open bitline architecture* [206, 207], fitting only enough SAs in one local row buffer to latch half a row (❸).

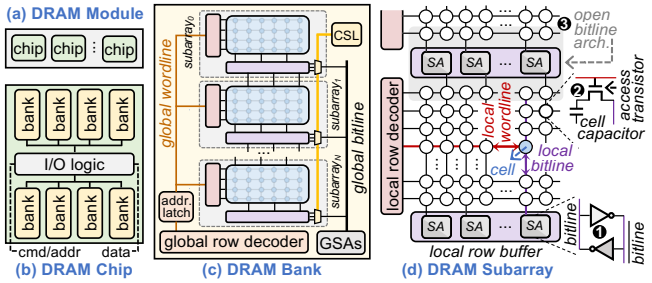


Figure 1: DRAM organization.

DRAM Operation. The memory controller issues three commands to service a DRAM request. The first command,

ACTIVATE (ACT), connects each DRAM cell in a row to its local bitline, and the cell's transistor shares its charge with the bitline to shift the bitline voltage higher (or lower) if the cell stores a '1' ('0'). The local row buffer amplifies the shifts to CMOS-readable values (simultaneously restoring charge to the DRAM cell). The latency from the start of activation until charge restoration is called t_{RAS}. The second command, READ (RD), returns a cache line of data from the local row buffer. The third command, PRECHARGE (PRE), disconnects DRAM cells from the bitlines, and returns the bitlines to their reference voltage. The precharge latency is called t_{RP}.

2.2 Processing-Using-DRAM

In-DRAM Row Copy. RowClone [104] enables copying a row *A* to a row *B* in the *same* subarray by issuing two consecutive ACTs to these two rows, followed by a PRE. This command sequence is called AAP. LISA [162] enables the execution of in-DRAM row copy operations across DRAM rows in *different* subarrays by connecting local row buffers of neighboring subarrays using isolation transistors.

In-DRAM Bitwise Operations. Ambit [101, 117] shows that a simultaneous *triple row activation (TRA)* can perform *in-DRAM* bitwise MAJ3/AND/OR operations. Ambit implements TRA using a custom row decoder, and introduces a new command called AP that issues a TRA followed by a PRE. Ambit also provides a mechanism to perform bitwise NOT operations [101]. SIMDRAM [143] builds on top of Ambit to implement and expose high-level in-DRAM operations (*μPrograms*). A μProgram consists of a sequence of AAPs (row copies) and APs that are generated offline, and exposed to the programmer as new *bbop* (bulk-bitwise operation) instructions. To implement carry propagation, SIMDRAM employs a *vertical* data layout, where all bits of a data word are stored in a single DRAM column, and executes each *bbop* instruction bit-serially. Such an execution model allows SIMDRAM to perform *implicit bit-shift* operations via in-DRAM row copies. Across this paper, we use the following terminology: (*i*) a *PUD operation* refers to the target computation that DRAM executes (e.g., addition, row copy); (*ii*) an *in-DRAM/PUD primitive* refers to the sequence of AAPs/APs in a μProgram; and (*iii*) a *PUD instruction* refers to a *bbop* instruction that the user/compiler uses to trigger a PUD operation.

3 Motivation

***Limitation 1*: Static Data Representation.** PUD architectures naively utilize conventional data formats (e.g., two's complement) and fixed operand bit-precision (e.g., 32-bit integers) to implement bit-serial computation. However, because bit-serial latency directly increases with bit-precision, these architectures experience subpar performance since an application's data with small dynamic range (i.e., narrow values) are often stored in large data formats [150–157] that waste most of the bit-precision. Note that data values often become narrow dynamically at runtime. Narrow values have been exploited in many scenarios, e.g., cache compression [150–152, 157, 158, 208–213], register files [153, 155, 214–219], logic synthesis & circuit optimizations [220–224], neural network quantization [225, 226], error tolerance [153, 227, 228].

***Opportunity 1:* Narrow Values for PUD Computation.** Narrow values can be exploited to reduce the bit-precision of a PUD operation to that of the best-fitting number of bits, thereby, improving overall performance. We quantify the required bit-precision in PUD-friendly real-world applications in Fig. 2. We define as *required bit-precision* the minimum number of bits required to represent the input operands of the PUD operation. We collect the required bit-precision dynamically in three main steps: we (*i*) instrument loops in applications that can be auto-vectorized using LLVM's loop auto-vectorization pass [229–232] (since prior work [141] shows that such loops are well-suited for PUD execution) to output the data values such loops use (i.e., we collect the data values of *each* array that is used as input/output of an auto-vectorized arithmetic instruction across the auto-vectorized loops), (*ii*) execute the application to completion, (*iii*) post-process the output file containing the loop information data to calculate the required bit-precision.

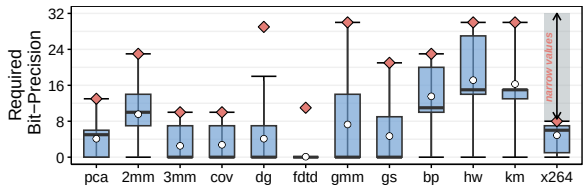


Figure 2: Required bit-precision distribution for input/output arrays of auto-vectorized instructions in loops across 12 applications. The box represents the 25th to 75th percentiles, with whiskers extending to the smallest/largest precision (with a diamond at the largest precision and a bubble at the mean precision).

We make two observations. First, all our real-world applications display a *significant* amount of narrow values. In such applications, the input bit-precision can be reduced from the native 32-bit to 20-bit (min. of 8-bit, max. of 30-bit) on average across *all* applications. By doing so, the performance of the underlying PUD architecture can improve by 1.6×, in case the application utilizes linearly-scaling PUD operations (such as addition [143]), or 2.6×, in case the application utilizes quadratically-scaling PUD operations (such as multiplication [143]). Second, the bit-precision significantly

varies across data arrays within a given application. This indicates the need for a mechanism that can *dynamically* identify the target bit-precision for a given PUD operation (similar to prior works that leverage narrow values for tasks other than PUD [150–157, 214, 215, 233, 234]). As prior work points out [154], static compiler analyses *cannot* identify the bit-precision of dynamically allocated and initialized data arrays. We investigated several prior compiler works [220, 235–238] that perform bit-width identification. However, such works are limited to identifying the bit-precision of statically-allocated variables.

Limitation 2: **Throughput-Oriented Execution.** Existing PUD architectures favor throughput-oriented execution as DRAM parallelism can partially hide the activation latency in a μProgram. To further improve throughput, prior works [141, 143, 144] use bank-level parallelism (BLP) to (*i*) distribute μPrograms across DRAM banks [143], or (*ii*) parallelize data writing and PUD computation of *different* μPrograms targeting *different* banks [144]. However, such approaches cannot reduce the latency of a *single* μProgram.

Opportunity 2: **DRAM Parallelism for Latency-Oriented Execution.** We make the *key observation* that several primitives in a μProgram (i.e., `AAP`/`AP` primitives that execute in-DRAM row copy or in-DRAM `MAJ3`/`NOT` operations) can be executed concurrently, as they are independent of one another. Fig. 3 shows this opportunity for a two-bit addition. In conventional bit-serial execution (Fig. 3a), *all* bits of the input arrays A and B are placed in a *single* DRAM subarray. Because of that, *all* in-DRAM primitives in a μProgram are *serialized*, enabling the execution of only a single bit-position at a time. In our example, (*i*) DRAM cycles ❶–❸ execute `MAJ3`/`NOT` operations over the *least-significant bits* (LSBs) of the input arrays A and B, i.e., A_0 and B_0; and afterwards (*ii*) DRAM cycles ❹–❻ execute `MAJ3`/`NOT` operations over the *most-significant bits* (MSBs) of the input arrays A and B, i.e., A_1 and B_1. However, the only *inter-bit* dependency in the μProgram is the *carry propagation* (ⓘ in Fig. 3a). In contrast, we can leverage *bit-level parallelism* to *concurrently* execute bit-independent in-DRAM primitives *across multiple* DRAM subarrays. In our example, we can reduce the overall latency of the bit-serial PUD addition operation by (*i*) *distributing* the individual bits of data-elements from arrays A and B across two DRAM subarrays (i.e., $subarray_0 \leftarrow \{A_0, B_0\}$; $subarray_1 \leftarrow \{A_1, B_1\}$), (*ii*) executing the required in-DRAM row copies (not shown) and `MAJ3`/`NOT` operations for the LSBs (DRAM cycles ❶–❸ in Fig. 3b) and MSBs (DRAM cycles ❷–❹ in Fig. 3b) *concurrently*, and (*iii*) serializing *only* the carry generated from the LSBs to the MSBs of the input arrays (ⓘⓘ in Fig. 3b).

Besides reducing the latency of bit-serial PUD operations, carefully distributing individual bit positions across different

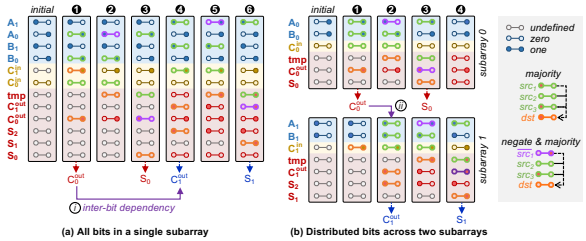


Figure 3: Simplified bit-serial PUD addition of two input arrays A and B, each of which with two-bit data elements using (a) one and (b) two DRAM subarrays.

DRAM subarrays enables the efficient realization of latency-friendly *bit-parallel* PUD arithmetic operations. By mapping each bit position of a data element to a distinct subarray, our PUD substrate can *concurrently* perform bitwise operations across all bits of the operand, thereby fully exploiting the parallelism inherent to bit-parallel arithmetic algorithms.

Limitation 3: **High-Precision Computation.** PUD suffers from high latency for high bit-precision operations. For example, even when employing multiple (i.e., 16) parallel DRAM banks, SIMDRAM's throughput for 32-bit and 64-bit division is 0.8× and 0.5× that of a 16-core CPU system [143]. This is because the latency of bit-serial multiplication and division scales *quadratically* with the bit-precision.

Opportunity 3: **Alternative Data Representation for High-Precision Computation.** The high latency associated with high-precision computation is an *inherent* property of coupling the binary numeral system with bit-serial computation. We investigate an alternative data representation, i.e. the *redundant binary representation (RBR)* [163–167], for high-precision computation. RBR is a positional number system where each bit-position i, which encodes 2^i, is represented by two bits that can take on a value $v \in \{-1, 0, 1\}$, such that the magnitude of bit-position i is $v \times 2^i$. PUD execution can take advantage of two key properties of RBR-based arithmetic: (*i*) the operations no longer need to propagate carry bits through the full width of the data (e.g., RBR-based addition limits carry propagation to *at most* two places [168]), and (*ii*) the operation latency is *independent* of the bit-precision.

Goal. Our *goal* in this work is to mitigate the three limitations of PUD architectures that arise due to the naive use of a bit-serial execution model. To do so, we aim to *fully* exploit the opportunities that DRAM's internal parallelism and dynamic bit-precision can provide to reduce the latency and energy of PUD operations. Concretely, we aim to enable (*i*) adaptive data-representation formats (two's complement and RBR) for PUD operands and (*ii*) flexible execution of different arithmetic algorithm implementations (bit-serial and bit-parallel) for PUD instructions.

4 *Proteus* Overview

Fig. 4 provides a high-level overview of *Proteus*' framework, its main components, and execution flow. *Proteus* is composed of three main components: (*i*) *Parallelism-Aware µProgram Library*, (*ii*) *Dynamic Bit-Precision Engine*, and (*iii*) *µProgram Select Unit*. These components are implemented in hardware inside the DRAM memory controller. The *Parallelism-Aware µProgram Library* and *µProgram Select Unit* are part of Proteus *Control Unit*.

4.1 Main Components of *Proteus*

Parallelism-Aware µProgram Library (§5.2). *Proteus* incorporates a *Parallelism-Aware µProgram Library* (ⓐ in Fig. 4) that consists of (*i*) hand-optimized implementations of different µPrograms for key PUD operations (each with different performance and bit-precision trade-offs), and (*ii*) *Cost Model Logic*. For each operation, we implement multiple µPrograms (§5.2.2) that use different (*i*) bit-serial or bit-parallel algorithms and (*ii*) data representation formats (i.e., two's complement or RBR). Each µProgram uses a novel data mapping that enables the *concurrent* execution of multiple independent primitives across bits (§5.2.1). The performance of each µProgram depends on the bit-precision, and the *Cost Model Logic* selects the best-performing µProgram for a given operation and target bit-precision. The *Cost Model Logic* comprises of *Pre-Loaded Cost Model LUTs*, which list the most-suitable µProgram for each bit-precision, and *Select Logic* to identify the target LUT for a *bbop* instruction. We *empirically* measure the throughput and energy efficiency of µPrograms in *Parallelism-Aware µProgram Library* while scaling the target bit-precision to populate the *Pre-Loaded Cost Model LUTs*.

Dynamic Bit-Precision Engine (§5.3). *Proteus*' *Dynamic Bit-Precision Engine* (ⓑ in Fig. 4) aims to identify the dynamic range of *memory objects* associated with a PUD operation. To do so, we dynamically identify (in hardware) the *largest* input operand a PUD's memory object stores. In state-of-the-art PUD architectures [110, 131, 132, 143, 144], cache lines belonging to a PUD's memory object need to be transposed from the traditional horizontal data layout to a vertical data layout *prior* to the execution of a PUD operation. To efficiently perform such data transformation, SIMDRAM [143] implements a *Data Transposition Unit*, which hides the data transposition latency by overlapping cache line evictions and data layout transformation. The *Data Transposition Unit* consists of an *Object Tracker* table (a small cache that keeps track of memory objects that are used by PUD operations) and *Data Transposition Engines*. The user/compiler informs the *Object Tracker* of PUD's memory objects using a specialized instruction called `bbop_trsp_init`. *Proteus* leverages such a *Data Transposition Unit* to dynamically identify in hardware the largest value in a PUD's memory object by adding: (*i*) a *new field* in the *Data Transposition Unit* called *maximum value*, which stores the largest value in a given memory object; and (*ii*) a *Dynamic Bit-Precision Engine*, which scans the data elements of evicted cache lines, identifies the largest data value across all data elements and updates the stored *maximum value* entry in the *Data Transposition Unit*.

µProgram Select Unit (§5.4). *Proteus*' *µProgram Select Unit* (ⓒ in Fig. 4) identifies the appropriate bit-precision for a PUD operation based on the operation's input data. The *µProgram Select Unit* has of a (*i*) *Bit-Precision Calculator Unit*, which evaluates the target bit-precision based on the input operands of the PUD operation and their associated maximum values, and (*ii*) buffers to store the selected µProgram.

4.2 Execution Flow

Proteus works in five main steps. In the first step (❶ in Fig. 4), the programmer/compiler utilizes specialized instructions (i.e., `bbop_trsp_init`) to (*i*) register in the *Object Tracker* the address, size, and initial bit-precision for each memory object used as an input, output, or temporary operand in a PUD operation; and (*ii*) execute a PUD operation over previously-registered memory objects. When issuing an arithmetic *bbop* instruction, the programmer/compiler indicates whether or not dynamic bit-precision is enabled or disabled for that *bbop* instruction. When dynamic bit-precision is disabled, *Proteus*' *Dynamic Bit-Precision Engine* is turned off, and the *µProgram Select Unit* utilizes the user-provided bit-precision for the upcoming PUD operation related to the issued *bbop* instruction. In the second step, if the *Dynamic Bit-Precision Engine* is enabled, it intercepts evicted cache lines belonging to previously registered memory objects (❷) and identifies the largest value stored in the cache line. If the identified value is *larger than* the current maximum value stored in the *Object Tracker*, the *Dynamic Bit-Precision Engine* updates the *Object Tracker* with the up-to-date value (❸). The second step is repeated for all cache lines belonging to the memory objects registered in the *Object Tracker*. As in SIMDRAM [143], our system employs lazy allocation and maintains data coherence for PUD memory objects through cache line flushing, using the `clflush` instruction [239]. Thus, all memory objects initially reside within the CPU caches, and prior to PUD execution, all cache lines belonging to a PUD operation are evicted to DRAM, which allows *Proteus*' *Dynamic Bit-Precision Engine* to access *all* data elements of a PUD operation prior to computation. In the third step, the host CPU dispatches the arithmetic *bbop* instruction (❹) to *Proteus*' Control Unit. In the fourth step, *Proteus*' *Control Unit* receives the *bbop* instruction from the CPU and the maximum values from the *Dynamic Bit-Precision Engine* (❺), which are used as inputs to the *µProgram Select Unit*. Based on

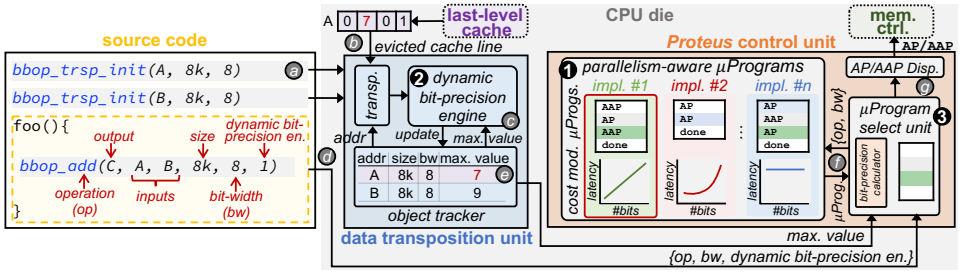


Figure 4: Overview of the *Proteus* framework.

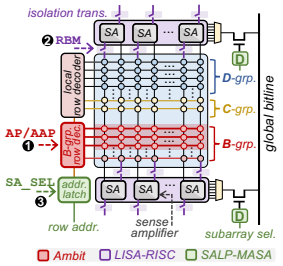


Figure 5: Subarray layout.

this information, the *Bit-Precision Calculator Unit* computes the target bit-precision and probes the *Parallelism-Aware μProgram Library* (❻), which returns the best-performing μProgram and data format representation for the target PUD operation and its associated bit-precision. In the fifth step, the *μProgram Select Unit* dispatches the sequence of AAPs/APs in the selected μProgram to DRAM (❼). When the host CPU reads back PUD memory objects (not shown in the figure), *Proteus* (*i*) performs the necessary data format conversions either from the reduced bit-precision to the user's specified bit-precision or from RBR to two's complement (thus maintaining system compatibility), and (*ii*) prepares the *Dynamic Bit-Precision Engine* for future accesses by resetting the current maximum data value stored in the *Object Tracker*.

5 *Proteus* Implementation

5.1 Subarray Organization

Performing Logic Primitives with Ambit. *Proteus* reuses the subarray organization of Ambit [101] and SIMDRAM [143] (shown in Fig. 5) to enable logic primitive execution with only small subarray modifications. DRAM rows are divided into three groups: (*i*) the **D**ata group (D-group), containing regular rows that store program data; (*ii*) the **C**ontrol group (C-group), containing two rows pre-initialized with all-'0' and all-'1' values; and (*ii*) the **B**itwise group (B-group), containing six rows (called *compute rows*) to perform bitwise operations. The B-group rows are all connected to a special row decoder that can simultaneously activate three rows when performing an AP and two when performing an AAP (❶ in Fig. 5).

Inter-Subarray Data Copy with LISA. *Proteus* leverages LISA-RISC [162], which dynamically connects adjacent subarrays using isolation transistors, to propagate intermediate data across subarrays. LISA-RISC works in four steps: (*i*) activate the source row in the source subarray (latency: t_{RAS}); (*ii*) use the LISA *row buffer movement* command (RBM, ❷ in Fig. 5) to turn on isolation transistors, which copies data from the source subarray's local row buffer (LRB) to the destination subarray's LRB (latency: t_{RBM}, 5 ns [162]); (*iii*) activate the destination row, to save the contents of the destination LRB into the destination row (latency: t_{RAS}); and (*iv*) precharge the bank (latency: t_{RP}). Due to DRAM's open bitline architecture [206, 207], each LRB stores half of the row, so we must perform steps (*ii*)–(*iv*) twice to copy both halves of the row.

Enabling Subarray-Level Parallelism with SALP. To enable the concurrent execution of bit-independent primitives in a μProgram, *Proteus* leverages SALP [161]. SALP-MASA (*Multitude of Activated Subarrays*) allows multiple subarrays in a bank to be activated concurrently by (*i*) pushing the global row-address latch to individual subarrays, (*ii*) adding a designated-bit latch (**D** in Fig. 5) to each subarray to ensure that only a single subarray's row buffer is connected to the global bitline, and (*iii*) routing a new global wire (called *subarray select*), controlled by a new DRAM command (SA_SEL, ❸ in Fig. 5), allowing the memory controller to set/clear each designated-bit latch.

5.2 Parallelism-Aware μProgram Library

5.2.1 One-Bit Per-Subarray (OBPS) Data Mapping. To reduce the latency of PUD operations (§3), *Proteus* employs a specialized data mapping called *one-bit per-subarray (OBPS)*. Bit-serial PUD architectures can employ three data mappings, as Fig. 6 illustrates: (*i*) all-bits in one-subarray (ABOS), (*ii*) all-bits per-subarray (ABPS), and (*iii*) OBPS. Assume an example DRAM bank with four subarrays, a DRAM row size of three and an input array *A* with six two-bit data elements.

First, the *ABOS* data mapping stores *all* six two-bit data elements in *one* DRAM subarray (Fig. 6a). This data mapping limits the parallelism available for PUD execution to that of a *single* DRAM subarray. In our example, the latency of executing a single PUD primitive over *all* data elements of the input array *A* is four PUD cycles (as shown in Fig. 6a).[2] Second, the *ABPS* data mapping distributes *all* bits of multiple sets of the input array across *multiple* DRAM subarrays (Fig. 6b),

[2]We refer to a *PUD cycle* as the end-to-end latency required to execute a single AAP/AP in-DRAM primitive.

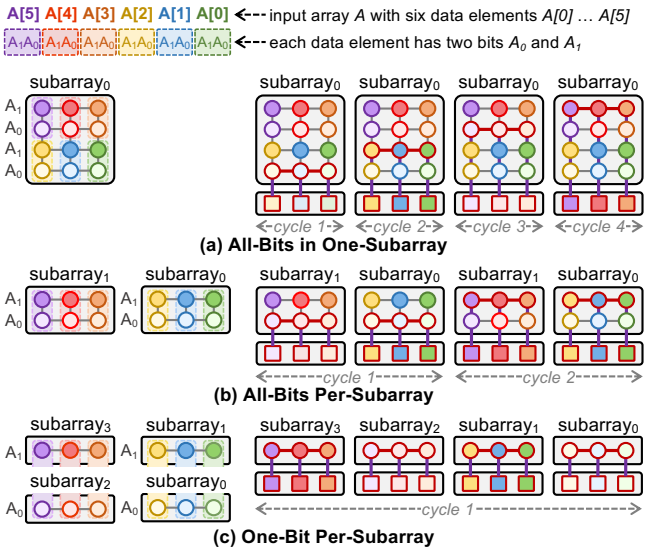


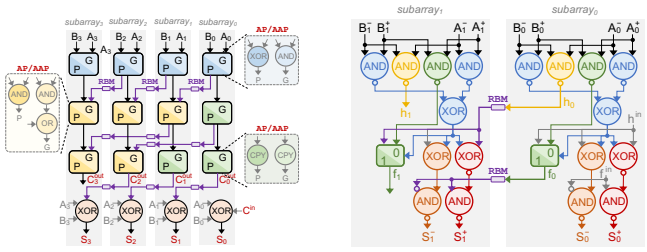
Figure 6: Three data mappings for bit-serial computing.

allowing a PUD primitive to execute concurrently on different portions of the input data stored in each subarray by exploiting *data-level parallelism*. In our example, the latency of executing a single PUD primitive over *all* data elements of the input array A while employing the ABPS data mapping is two PUD cycles (as shown in Fig. 6b). This is because, although execution across data elements can be parallelized by distributing them across multiple DRAM subarrays, the PUD system must still serialize the execution of PUD primitives across different bit positions of each data element, since *all bits of a given data element* are co-located within a single DRAM subarray under ABPS. Third, the OBPS data mapping distributes each of the m individual bits of a given data element of the input array to m DRAM subarrays (Fig. 6c), i.e., $subarray_0 \leftarrow \{A_0\}, \ldots, subarray_{m-1} \leftarrow \{A_{m-1}\}$, allowing a PUD primitive to execute concurrently on different bits of the input array stored in each subarray by exploiting *bit-level parallelism*.[3] In our example, the latency of executing a single PUD primitive over *all* data elements of the input array A while employing the OBPS data mapping is *only* a single PUD cycle (as shown in Fig. 6c).

5.2.2 µProgram Library Implementation. *Proteus* leverages the subarray organization in Fig. 5 and our OBPS data mapping (Fig. 6) to implement parallelism-aware µPrograms for key arithmetic operations (e.g., addition, multiplication). We implement three classes of algorithms for arithmetic PUD computations: *bit-serial*, *bit-parallel*, and *RBR-based algorithms*. In *Proteus*, each µProgram implementation (i) has an associated *µProgram_addr*, and (ii) is stored in a reserved memory space in DRAM (i.e., *µProgram Memory*).

Bit-Serial Algorithms. We optimize µPrograms for bit-serial arithmetic operations (i.e., addition, subtraction, division, and multiplication) by concurrently executing independent AAPs/APs across different DRAM subarrays. Fig. 3b illustrates such a process for addition using the OBPS data mapping (the process is analogous for other arithmetic operations). *Proteus* implements a ripple-carry adder using majority gates in two main steps. First, *Proteus* utilizes SALP-MASA to concurrently execute the appropriate row copies and majority operations across N different subarrays. Second, *Proteus* utilizes LISA-RISC to pipeline the carry propagation process (ⓘⓘ in Fig. 3b) from $subarray_i$ (e.g., C^0_{out}) to $subarray_{i+1}$ (e.g., C^1_{in}). This process repeats for all N bits in the input operand. *Proteus* reduces the latency of executing an N-bit bit-serial addition from $8N + 1$ AAP/AP cycles [143] to $2N + 7$ AAP/AP cycles + $2(N - 1)$ RBM cycles.

Bit-Parallel Algorithms. We implement bit-parallel variants of our µPrograms that leverage *carry-lookahead logic* to decouple the calculation of the carry bits and arithmetic logic (e.g., addition). Carry-lookahead logic can identify if any arithmetic on a bit will *generate* a carry (e.g., both operands bits are '1' for an addition), or if it will *propagate* the carry value (e.g., only one operand bit for an addition is a '1'). For N-bit operands, this reduces time complexity compared to ripple-carry logic from $O(N)$ to $O(\log N)$, where N is the number of bits in the input operands. We implement several carry-lookahead algorithms in *Proteus*, including the carry-select [240], Kogge–Stone [241], Ladner–Fischer [242], and Brent–Kung [243] adders, as building blocks to implement subtraction, multiplication, and division. Fig. 7a shows an example *Proteus* implementation of a Kogge–Stone adder.



(a) 4-bit Kogge–Stone adder **(b) 2-bit RBR adder**

Figure 7: *Proteus*' implementation of different adders. Bits A_i and B_i are stored *vertically* in the same DRAM bitline of subarray i using the OBPS data mapping.

In the first step, *Proteus* performs $2N$+4 inter-subarray data copies (using LISA-RISC) to copy the *generate* and *propagate*

[3] If the number of subarrays is smaller than the target bit-precision, OBPS *evenly* distributes the bits of input operands across the available subarrays.

bits from $subarray_i$ to $subarray_{i+1}$. In the second step, *Proteus* performs a series of Boolean operations (using AAPs/APs) to compute the next generate and propagate bits in parallel (using SALP-MASA) across *all* DRAM subarrays. These two steps repeat for $log(N)$ iterations. The latency of executing an N-bit bit-parallel addition using *Proteus* is $3log_2N + 13$ AAP/AP cycles + $2N + 4$ RBM cycles. Even though the bit-parallel algorithms have a lower time complexity than the bit-serial algorithms, the former can require more inter-subarray copies, i.e., $2N+4$ RBM cycles for bit-parallel algorithms versus $2(N-1)$ RBM cycles for bit-serial algorithms.

RBR-Based Algorithms. Fig. 7b illustrates *Proteus*' implementation of a two-bit RBR-based adder [244]. The adder operates in three steps. First, each digit i generates an intermediate value h_i, computed *only* from the corresponding input digit i. Second, the output value f_i at digit i is computed as a function of both the current digit and the preceding intermediate value h_{i-1}. Third, the *sum* at digit i depends on the current digit, h_{i-1}, and f_{i-1}. To propagate intermediate results between digits, *Proteus* uses RBM commands to transfer the values of h_i and f_i from $subarray_i$ to $subarray_{i+1}$. The RBR-based addition executes with a constant latency of 34 AAPs/APs cycles and 8 RBM cycles. Beyond addition, *Proteus* reuses the same RBR-based adder design to support additional arithmetic operations, including subtraction and multiplication in the RBR format.

5.2.3 Cost Model Logic Implementation. Fig. 8 depicts the hardware design of the *Cost Model Logic*. The *Cost Model Logic* has two main components: (*i*) one LUT per PUD operation, and (*ii*) *Select Logic*. Each LUT row represents a different bit-precision, and stores the index of the best-performing µProgram in the library for that operation–precision combination. We *empirically* sized each LUT to contain 64 eight-bit rows (i.e., supporting up to 64-bit computation, and indexing up to 64 different µProgram implementations per PUD operation). The *Cost Model Logic* works in four main CPU cycles. It receives as input the *bit-precision* (6 bits) and the *bbop_op* opcode (4 bits) of the target PUD operation. In the first cycle, the *bit-precision* indexes all the LUTs in parallel (❶ in Fig. 8), selecting the best-performing *µProgram_id* for the given bit-precision for all implemented PUD operations (❷). The *Cost Model Logic* can *quickly* query the LUTs since they consist of a few (i.e., 16) small (i.e., 64 B in size) SRAM arrays indexed in parallel. In the second cycle, based on the 4-bit *bbop_op* opcode, the *Select Logic* chooses the appropriate *µProgram_id* (❸). In the third cycle, the *µProgram_id* is concatenated with the *bbop_op* opcode to form the *µProgram_addr* (❹). In the fourth cycle, the *µProgram_addr* indexes and fetches the best-performing µProgram from the *µProgram Scratchpad* (❺). If the target µProgram is not loaded in the *µProgram Scratchpad*, the *Cost Model Logic* fetches it from the *µProgram Memory* (not shown). We estimate, using CACTI [245], that the access latency and energy per access of the 64 B SRAM array (used in our *Cost Model Logic*) is of 0.07 ns (i.e., less than 1 CPU cycle) and 0.000 04 nJ.

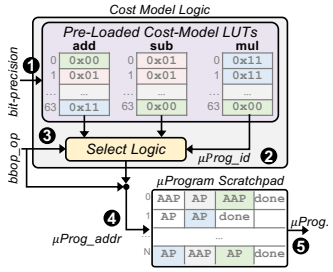


Figure 8: *Proteus*' Cost Model Logic.

5.2.4 Pareto Analysis. We conduct a performance and energy Pareto analysis to populate the *Pre-Loaded Cost Model LUTs*. We model each µProgram using an analytical cost model that takes as input the target bit-precision, the number of elements used during computation, and the number of DRAM subarrays available. The analytical cost model outputs the throughput (in GOPs/s) and energy efficiency (in throughput/Watt) for each µProgram in the *Parallelism-Aware µProgram Library*. We highlight our analyses for two main operations (i.e., addition and multiplication) since they represent linearly and quadratically-scaling PUD operations, respectively. The analyses for subtraction and division follow similar observations. In our analyses, we evaluate a SIMDRAM-like PUD architecture using the three data mapping schemes described in Fig. 6. We assume a DRAM bank with 64 PUD-capable DRAM subarrays and a subarray with 65,536 columns. We vary the number of input elements as multiples of the number of DRAM columns per subarray (from 1 DRAM subarray with 64K input elements to 64 DRAM subarrays with 4M input elements).

Linearly-Scaling PUD Operations. Fig. 9 shows the throughput (y-axis; top) and energy efficiency (y-axis; bottom) of six µProgram implementations for a linearly-scaling PUD operation (i.e., integer addition) for different bit-precision values (x-axis). Each subplot depicts the different input data sizes we use in our analysis. For this analysis, we implement the following addition algorithms: ripple-carry adder (RCA), carry-select adder (CSA) [240], Brent-Kung adder [243], Kogge–Stone adder [241], Ladner-Fischer adder [242], using (*i*) both two's complement and RBR data format representations; and (*ii*) ABOS, ABPS, and OBPS data mappings. Note that the bit-parallel adder can *only* be implemented using the OBPS data mappings. We make two observations. First, in terms of throughput, the best-performing

adder implementation varies depending on the target bit-precision and number of input elements. The achievable throughput ultimately depends on a combination of the number of AAPs/APs that can be concurrently executed across DRAM subarrays and the number of inter-DRAM subarray operations required to implement the adder. In general, we empirically observe that as the input data size increases (see subplots' titles), the number of inter-DRAM subarray operations also increases and eventually dominates the overall execution time. For *small bit-precision* and *small input size* (i.e., bit-precision smaller than 8, and fewer than 256K input elements), the bit-serial RCA using the OBPS data mapping provides the highest throughput, while for *large bit-precision* and *small input size* (i.e., bit-precision larger than 8, and fewer than 256K input elements), the RBR adder using the OBPS data mapping provides the highest throughput. For large-enough input sizes (i.e., more than 1M input elements), employing the ABPS data mapping leads to the highest throughput, independent of the bit-precision. This is because when more DRAM subarrays are involved in the execution of the target PUD operation, the inter-subarray data transfers dominate overall execution time in the OBPS implementations. Second, in terms of energy efficiency, the bit-serial implementation of RCA provides the best throughput/Watt for ABOS, ABPS, and OBPS, independent of the bit-precision and input size. This is because (*i*) the number of AAPs/APs performed to execute RCA is the same *independent* of the data mapping, and (*ii*) the energy the bit-parallel algorithms consume is dominated by inter-subarray operations, which is *not* present in bit-serial implementations.

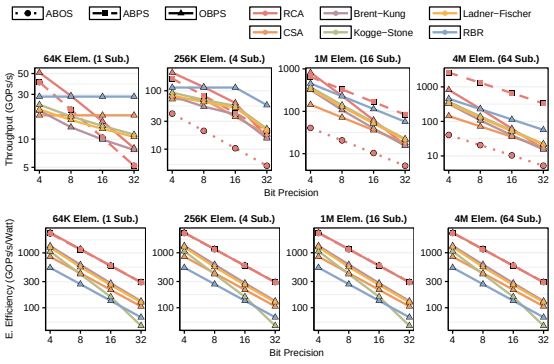


Figure 9: Pareto analysis for throughput (top) and energy efficiency (bottom) for PUD addition operations. Dotted lines represent ABOS; dashed lines represent ABPS; straight lines represent OBPS data mapping.

Quadratically-Scaling PUD Operations. Fig. 10 shows the throughput (top) and energy efficiency (bottom) of six μProgram implementations for a quadratically-scaling PUD operation (i.e., integer multiplication). We implement PUD multiplication operations as a triplet composed of (*i*) the multiplication method (i.e., Booth's multiplication algorithm [246] or the divide-and-conquer Karatsuba [247] multiplication); (*ii*) different methods for addition (i.e., bit-serial RCA, bit-parallel Ladner-Fischer [242], and RBR-based adder); and (*iii*) data mappings (i.e., ABOS, ABPS, and OBPS). Note that PUD multiplication operations that use bit-parallel and RBR-based adders can *only* be implemented using the OBPS data mapping. We make two observations. First, in terms of throughput, the best-performing multiplier implementation varies depending on the bit-precision and number of input elements. For small bit-precision and small input size (i.e., bit-precision smaller than 8, and fewer than 64K input elements), Booth's bit-serial multiplication with ABOS data mapping provides the highest throughput, while for medium bit-precision and small input size (i.e., bit-precision from 8 to 16 and fewer than 64K input elements), Booth's bit-parallel multiplication with the OBPS data mapping provides the highest throughput. For high bit-precision and small-to-medium input size (i.e., bit-precision larger than 32 and fewer than 256K input elements), RBR-based multiplication using OBPS data mapping provides the highest throughput. For large-enough input sizes (i.e., larger than 1M input elements), employing Booth's bit-serial RCA-based multiplication using ABPS data mapping leads to the highest throughput, independent of the bit-precision. Second, in terms of energy efficiency, Booth's bit-serial RCA-based multiplication implementation provides the best throughput/Watt for ABOS, ABPS, and OBPS, independent of the bit-precision and input size, since (*i*) the number of AAPs/APs required to execute the addition step is the same regardless of the data mapping and (*ii*) the energy of the bit-parallel-based algorithms is dominated by the large number of inter-subarray operations.

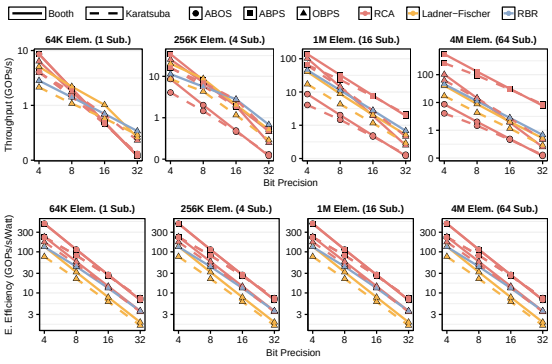


Figure 10: Pareto analysis for throughput (top) and energy efficiency (bottom) for multiplication. Straight lines represent the Booth's multiplication method [246]; dashed lines represent the Karatsuba [247] multiplication method.

5.2.5 Non-Arithmetic PUD Operations. We also equip *Proteus*' *Parallelism-Aware μProgram Library* with SIMDRAM's implementations of non-arithmetic PUD operations [143], including (*i*) N-bit logic operations (i.e., AND/OR/XOR of more than two input bits), (*ii*) relational operations (i.e., equality/inequality check, greater than, maximum, minimum), (*iii*) predication, and (*iv*) bitcount and ReLU [248].

5.3 Dynamic Bit-Precision Engine

The *Dynamic Bit-Precision Engine* comprises a simple reconfigurable n-bit comparator and a finite state machine (FSM). For each evicted cache line, the FSM probes the *Object Tracker* and identifies if the incoming evicted cache line belongs to a PUD's memory object. If it does, the FSM executes four operations. First, it reads the bit-precision value (specified by the `bbop_trsp_init` instruction) and the current maximum value stored in the *Object Tracker* for the given memory object. Second, it uses the bit-precision value to configure the n-bit comparator. Third, it inputs to the n-bit comparator all n-bit values in the incoming cache line (one at a time) and the current maximum value. Fourth, after all the n-bit values are processed, if any value in the incoming cache line is larger than the current maximum value, the FSM sends an update signal to the *Object Tracker* alongside the new maximum value. The energy cost of identifying the largest element in a 64 B cache line is 0.0016 nJ [249]. That represents an increase in 0.084% in the energy of an LLC eviction [81, 250, 251], which *needs* to happen prior to PUD execution regardless.

5.4 μProgram Select Unit

Calculating Bit-Precision. The *μProgram Select Unit* needs to address two scenarios when calculating the bit-precision for PUD operations: *vector-to-vector* PUD operations, and *vector-to-scalar* reduction PUD operations. In *vector-to-vector*, the target PUD operation implements a parallel *map* operation, in which inputs and outputs are data vectors. For such operations, the bit-precision can be computed *a priori*, using the maximum values the *Dynamic Bit-Precision Engine* provides, *even* in the presence of chains of PUD operations. In such a case, the *Bit-Precision Calculation Engine* updates the *Object Tracker* with the maximum possible output value for *each* PUD in the chain. For example, assume a kernel that executes `D[i]=(A[i]+B[i])×C[i]` as follows:

```
bbop_add(tmp, A, B, 8k, 8, 1);   // tmp ← A + B
bbop_mul(D, tmp, C, 8k, 8, 1);   // D ← tmp × C
```

Assume that the maximum value of `A`, `B`, and `C` are 3, 6, and 2, respectively. In this case, the *μProgram Select Unit* (*i*) computes the bit-precision for the addition operation as $\lceil \log_2(3+6) \rceil = 4\ bits$; (*ii*) updates the *Object Tracker* entry of `tmp` with the maximum value of the addition operation (i.e., 9); (*iii*) computes the bit-precision for the multiplication operation as $\lceil \log_2(9 \times 2) \rceil = 5\ bits$ using an n-bit scalar ALU; (*iv*) updates the *Object Tracker* entry of `D` with the maximum value of the multiplication (i.e., 18).

In *vector-to-scalar* reduction, the PUD operation implements a parallel *reduction* operation, where the inputs are vectors and the output is a scalar value. In this case, the bit-precision *cannot* be computed with *only* the maximum input operands without causing *overprovisioning*, since in a reduction, each element contributes to the bit-precision of the scalar output. Therefore, for *vector-to-scalar* reduction PUD operations, the *μProgram Select Unit* needs to (*i*) fetch from DRAM the row containing the carry-out bits produced during partial steps[4] of the PUD reduction; (*ii*) evaluate if a partial step generated an overflow (i.e., check if any carry-out bit is '1'); and (*iii*) increment the bit-precision for the next partial step if overflow is detected.

Hardware Design. The *μProgram Select Unit* comprises of simple hardware units: (*i*) an *n-bit ALU* to compute the target bit-precision, (*ii*) a *Fetch Unit* to generate load instructions for carry re-evaluation, and (*iii*) a *μProgram Buffer* to store the currently running μProgram.

6 Methodology

We implement *Proteus* using an in-house cycle-level simulator (which we open-source at [252]) and compare it to a real multicore CPU (Intel Comet Lake [253]), a real high-end GPU (NVIDIA A100 using CUDA and tensor cores [254]), and a simulated state-of-the-art PUD framework (SIMDRAM [143]). In our evaluations, the CPU code uses AVX-512 instructions [255]. Our simulator is rigorously validated against SIMDRAM [143] and MIMDRAM [141]'s gem5 [256] implementation [257]. The simulator (*i*) is cycle-level accurate with regard to DRAM commands and (*ii*) accounts for the data movement cost of cache line eviction on a per-cycle basis. Our simulation accounts for the additional latency imposed by SALP [161] on `ACT` commands, i.e., the extra circuitry required to support SALP incurs an extra latency of 0.028 ns to an `ACT` [258], which is less than 0.11% extra latency of an `AAP`. To verify the functional correctness of our applications, our simulation infrastructure performs functional verification over application's data when performing PUD operations. We did *not* observe any difference from the golden outputs. We open-source our simulation infrastructure at https://github.com/CMU-SAFARI/Proteus.

Table 1 shows the system parameters we use in our evaluations. To measure CPU energy consumption, we use Intel RAPL [259]. We capture GPU kernel execution time that excludes data initialization/transfer time. We use the `nvml` API [260] to measure GPU energy consumption. We

[4] *Proteus* implements PUD reduction operations using *reduction trees* [141]. Thus, a partial step refers to a level of the reduction tree.

use CACTI 7.0 [245] to evaluate *Proteus* and SIMDRAM energy consumption, where we take into account that each additional simultaneous row activation increases energy consumption by 22% [101, 143]. We evaluate two SIMDRAM configurations: (*i*) SIMDRAM with SALP [161] and static bit-precision (*SIMDRAM-SP*), and (*ii*) SIMDRAM with SALP and *Proteus*' *Dynamic Bit-Precision Engine* (*SIMDRAM-DP*). In both configurations, the system implements only the 16 μPrograms proposed in SIMDRAM (i.e., there is *no Parallelism-Aware μProgram Library* enabled). We evaluate four *Proteus* configurations: (*i*) *Proteus LT-SP* and (*ii*) *Proteus EN-SP*, where *Proteus* selects the *lowest latency* (LT) and *lowest energy* (EN) consuming μProgram, respectively, using the statically profiled bit-precision from Fig. 2; (*iii*) *Proteus LT-DP* and (*iv*) *Proteus EN-DP*, where *Proteus* executes the *lowest latency* and *lowest energy* consuming μProgram with dynamically chosen bit-precision. We use 64 subarrays in *only one* DRAM bank for our PUD evaluations.

Table 1: Evaluated system configurations.

Intel Comet Lake CPU [261] (Real System)	x86 [239], 16 cores, 8-wide, out-of-order, 3.8 GHz; *L1 Data + Inst. Private Cache:* 256 kB, 8-way, 64 B line; *L2 Private Cache:* 2 MB, 4-way, 64 B line; *L3 Shared Cache:* 16 MB, 16-way, 64 B line; *Main Memory:* 64 GB DDR4-2133, 4 channels, 4 ranks
NVIDIA A100 GPU [254] (Real System)	7 nm technology node; 826 mm^2 die area [254]; 6912 CUDA cores; 432 tensor cores, 108 streaming multiprocessors, 1.4 GHz base clock; *L2 Cache:* 40 MB L2 Cache; *Main Memory:* 40 GB HBM2 [193, 262]
SIMDRAM [143] & *Proteus* (Simulated)	gem5-based in-house simulator [252, 257]; x86 [239]; 1 out-of-order core @ 4 GHz (*only* for instruction offloading); *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 [263, 264] *Main Memory:* DDR5-5200 [265], 1 channel, 1 rank, 16 banks

Real-World Applications. We select twelve workloads from four popular benchmark suites in our real-workload analysis (as Table 2 describes). We manually modified each workload to (*i*) identify loops that can benefit from PUD computation, i.e., loops that are memory-bound and that can leverage single-instruction multiple-data (SIMD) parallelism and (*ii*) use the appropriate *bbop* instructions. To identify loops that can leverage SIMD parallelism, we use the MIMDRAM compiler [257] for identification and generation of PUD instructions, which uses LLVM's loop auto-vectorization engine [229–232] as a profiling tool that outputs SIMD-safe loops in an application. We use the clang compiler [229] to compile each application while enabling the loop auto-vectorization engine and its loop vectorization report (i.e., `-O3 -Rpass-analysis=loop-vectorize -Rpass=loop-vectorize`). We observe that applications with regular and wide data parallelism (e.g., applications operating over large dense vectors) are better suited for SIMD-based PUD systems. We select applications from various domains, including linear algebra and stencil computing (i.e., 2mm, 3mm, doitgen, fdtd-apml, gemm, gramschmidt from Polybench [169]), machine learning (i.e., pca from Phoenix [171], covariance from Polybench [169], kmeans and backprop from Rodinia [172]), and image/video processing (i.e., heartwall from Rodinia [172] and 525.x264_r from SPEC 2017 [170]). Since our baseline PUD substrate (SIMDRAM) does *not* support floating-point, we manually modify the selected floating-point-heavy PUD-friendly loops to operate on fixed-point data arrays. We use the largest input dataset available for each benchmark.

Table 2: Evaluated applications.

Benchmark Suite	Application (Short Name)	Peak GPU Util. (%)	Total Mem. Footprint (GB)	Bit-Precision {min, max}	PUD Instrs.†
Phoenix [171]	pca (pca)	–	1.91	{8, 8}	D, S, M, R
Polybench [169]	2mm (2mm)	98	4.77	{13, 25}	M, R
	3mm (3mm)	100	26.7	{12, 12}	M, R
	covariance (cov)	100	7.63	{23, 23}	D, S, R
	doitgen (dg)	92	33.08	{10, 11}	M, C, R
	fdtd-apml (fdtd)	–	36.01	{11, 13}	D, M, S, A
	gemm (gmm)	98	22.89	{12, 24}	M, R
	gramschmidt (gs)	66	22.89	{12, 13}	M, D, R
Rodinia [172]	backprop (bp)	–	22.50	{13, 13}	M, R
	heartwall (hw)	48	0.03	{17, 17}	M, R
	kmeans (km)	36	1.23	{17, 17}	S, M, R
SPEC 2017 [170]	525.x264_r (x264)	–	0.15	{1, 8}	A, R

† D = division, S = subtraction, M = multiplication, A = addition, R = reduction, C = copy

7 Evaluation

7.1 Real-World Application Analysis

Performance. Fig. 11 shows the CPU, GPU, SIMDRAM, and *Proteus* performance for twelve real-world applications. As in prior works [103, 119, 120, 266, 267], we report area-normalized results (i.e., performance per mm^2) for a fair comparison. We make four observations. First, *Proteus significantly* outperforms all three baseline systems. On average across all twelve applications, *Proteus LT-DP* (*Proteus EN-DP*) achieves 17× (11.2×), 7.3× (4.8×), and 10.2× (6.8×) the performance per mm^2 of the CPU, GPU, and SIMDRAM, respectively. Second, we observe that equipping SIMDRAM with *Proteus*' *Dynamic Bit-Precision Engine* to leverage narrow values for PUD execution *significantly* improves overall performance. On average, *SIMDRAM-DP* provides 6.3× the performance per mm^2 of *SIMDRAM-SP*. Third, *Proteus*' ability to adapt the μProgram depending on the target bit-precision further improves overall performance by 1.6× that of *SIMDRAM-DP*. Fourth, *Proteus*' *Dynamic Bit-Precision Engine* further increases performance by 46%, over *Proteus* with static bit-precision. This happens because for statically profiled bit-precision, we *must* round the bit-precision up to the nearest power-of-two, as high-level programming languages (e.g., C/C++) are *inherently* constrained by the two's complement data representation.

Energy. Fig. 12 shows the end-to-end energy reduction the GPU, SIMDRAM, and *Proteus* provide compared to the baseline CPU for twelve applications. We make four observations.

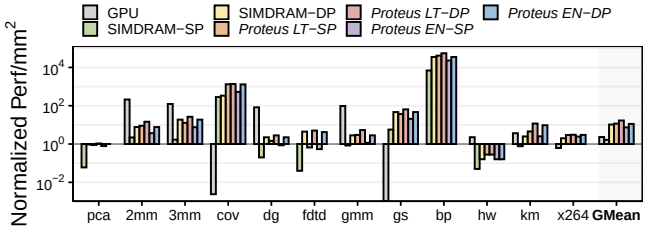


Figure 11: CPU-normalized performance per mm^2 for twelve real-world applications. Phoenix [171] and SPEC2017 [170] do *not* provide GPU implementations of pca and x264.

First, *Proteus significantly* reduces energy consumption compared to all three baseline systems. On average across all twelve applications, *Proteus EN-DP* (*Proteus LT-DP*) provides 90.3× (27×), 21× (6.3×), and 8.1× (2.5×) lower energy consumption than CPU, GPU, and *SIMDRAM-SP*, respectively. Second, enabling *Proteus*' *Dynamic Bit-Precision Engine* and *Parallelism-Aware μProgram Library* allows *Proteus* to reduce energy consumption by an average of 8× and 1.02× compared to PUD substrates with statically-defined bit-precision (*SIMDRAM-SP*) and bit-serial *only* arithmetic (*SIMDRAM-DP*), respectively. Third, compared to *SIMDRAM-DP*, *Proteus LT-DP increases* energy consumption by 3.3×, on average. This is because the highest performance implementation of a PUD operation often leads to an increase in the number of AAPs/APs required for PUD computing. In many cases, the energy associated with inter-subarray data copies (employed in RBR and bit-parallel algorithms) leads to an *increase* in energy consumption. Fourth, the *Dynamic Bit-Precision Engine* further reduces *Proteus*' energy consumption by 58%, compared to *Proteus* with static bit-precision.

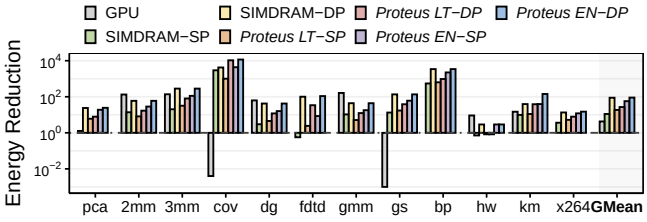


Figure 12: End-to-end energy reduction compared to the baseline CPU for twelve applications.

7.2 *Proteus* vs. Tensor Cores in GPUs

We compare the performance and energy efficiency of our real-world applications that perform general matrix-matrix multiplication (GEMM) operations while running on the tensor cores in the NVIDIA A100 GPU and *Proteus* for narrow data precision input operands (i.e. 4-bit and 8-bit integers). To do so, we (*i*) identify the subset of our real-world applications that mainly perform GEMM operations and therefore are suitable for the A100's tensor core engines; and (*ii*) re-implement such workloads using optimized instructions (from NVIDIA's CUTLASS [268]) to perform tensor GEMM operations on the A100 GPU tensor cores. Re-implementing the GPU workloads is necessary since GPU tensor core instructions are *not* automatically produced via the standard CUDA code our workloads use and there is *no* reference implementation available from the original benchmark suites targeting tensor core GPUs. We employ A100's all 432 tensor cores during GPU execution.

Fig. 13 shows the tensor cores, SIMDRAM, and *Proteus* performance per mm^2 (Fig. 13, top) and energy efficiency (i.e., performance per Watt in Fig. 13, bottom) for three GEMM-heavy real-world applications using 8-bit (`int8`) and 4-bit (`int4`) data types. Values are normalized to those obtained on real GPU tensor cores. We make two observations. First, *Proteus* significantly improves performance per mm^2 and energy efficiency compared to both tensor cores and SIMDRAM across all applications and data types. On average across the three applications, *Proteus* provides (*i*) 20×/43× and 8×/21× the performance per mm^2 and (*ii*) 484×/767× and 9.8×/25× the performance per Watt of the tensor cores and SIMDRAM, respectively, using `int8`/`int4` data types. *Proteus* and SIMDRAM are capable of outperforming the tensor cores of the A100 GPU for narrow data precisions since both the throughput and the energy efficiency of bit-serial PUD architectures *increase* quadratically for multiplication operations as the bit-precision *decreases* [143]. Second, we observe that by employing dynamic bit-precision and adaptive arithmetic computation, *Proteus* further improves the performance and energy gains that SIMDRAM provides compared to the A100 GPU's tensor cores, even improving performance compared to the tensor cores in cases where SIMDRAM fails to do so (i.e., for `gmm`).

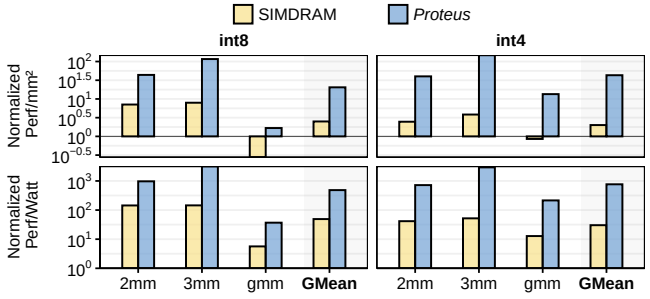


Figure 13: Performance per mm^2 (top) and performance per Watt (bottom) of GEMM-intensive real-world applications using `int8` and `int4`, normalized to the same metric measured on 432 NVIDIA A100 tensor cores.

7.3 Area Analysis

DRAM Chip Area and Storage Overhead. We use CACTI 7.0 [245] to evaluate the area overhead of the primary components in the *Proteus* design using a 22 nm technology node. *Proteus* does *not* introduce any modifications to the DRAM array circuitry other than those proposed by (*i*) Ambit, which has an area overhead of <1% in a commodity DRAM chip [101]; (*ii*) LISA, which has an area overhead of 0.6% in a commodity DRAM chip [162]; and (*iii*) SALP, which has an area overhead of 0.15% in a commodity DRAM chip [161]. We reserve less than 1 DRAM row (i.e., 6.25 kB in an 8 GB) to store our implemented µPrograms. In total, we implement 50 µPrograms, each of which takes 128 B of DRAM space.

CPU Area Overhead. We size the *Parallelism-Aware µProgram Library* to contain: (*i*) 16 64 B LUTs, each LUT holding a 8-bit *µProgram_id*); (*ii*) one 2 kB *µProgram Scratchpad* Memory. The size of the *Parallelism-Aware µProgram Library* is enough to hold one LUT per SIMDRAM PUD operations and address 2^8 different µProgram implementations. The size of the µProgram Scratchpad is large enough to store the µPrograms for all 16 SIMDRAM operations. We use a 128 B scratchpad for the *Dynamic Bit-Precision Engine*. Using CACTI, we estimate that the *Proteus Control Unit* area is 0.03 mm^2. *Proteus*' *Data Transposition Unit* (one per DRAM channel) uses an 8 kB fully-associative cache with a 128-bit cache line size for the *Object Tracker*, and two 4 kB transposition buffers. Using CACTI, we estimate the *Data Transposition Unit* area is 0.06 mm^2. Considering the area of the control and transposition units, *Proteus* has an area overhead of only 0.03% compared to the die area of an Intel Xeon E5-2697 v3 CPU [113].

8 Related Work

To our knowledge, *Proteus* is the first system that can transparently execute PUD operations with the best bit-precision, data representation, and algorithm arithmetic implementation. We highlight *Proteus*' key contributions by contrasting them with state-of-the-art PIM designs.

Processing-Using-DRAM. Prior works propose different ways of implementing PUD operations [101, 103, 104, 106, 107, 110, 120, 127, 132, 133, 141, 143, 147, 269]. Such works could benefit from *Proteus*' dynamic bit-precision selection and alternative data representation and algorithms, since they all assume a static bit-precision and algorithmic implementation. AritPIM [270] provides a collection of bit-parallel and bit-serial algorithms for PUM arithmetic. Compared to AritPIM, *Proteus* (*i*) extends AritPIM's set of bit-parallel algorithms for PUD; (*ii*) evaluates different data mapping and format representations that lead to further performance and energy improvements; (*iii*) proposes a framework that can dynamically adapt to the bit-precision of the operation.

Using Bit-Slicing Compilers & Early Termination for PIM. Prior works [144, 271] propose bit-slicing compilers for bit-serial PIM computation. In particular, CHOPPER [144] improves SIMDRAM's programming model by leveraging bit-slicing compilers and employing optimizations to reduce the latency of a µProgram. Compared to CHOPPER, *Proteus* has two main advantages. First, *Proteus* improves µProgram performance by leveraging the DRAM parallelism within a single DRAM bank via SALP. Second, although bit-slicing compilers can naturally adapt to different bit-precision values, they require the programmer to specify the target bit-precision manually. In contrast, *Proteus dynamically* identifies the most suitable bit-precision transparently from the programmer. Some other prior works (e.g., [272, 273]) propose techniques to realize dynamic bit-precision-based PUM operations for different memory technologies. Compared to these, *Proteus*' main novelty lies in realizing dynamic bit-precision of bit-serial and bit-parallel operation in the context of DRAM/majority-based PUD systems.

9 Conclusion

We introduce *Proteus*, a data-aware hardware runtime framework that addresses the high execution latency of bulk bit-wise PUD operations. To do so, *Proteus dynamically* adjusts the bit-precision of PUD operations by exploiting narrow values, and, based on that, *chooses* and *uses* the most appropriate data representation (i.e., two's complement or redundant-binary representation) and arithmetic algorithm implementation (i.e., bit-serial or bit-parallel) for PUD systems. We demonstrate that *Proteus* provides large performance and energy benefits over state-of-the-art CPU, GPU, and PUD systems. The source code of *Proteus* is freely available at https://github.com/CMU-SAFARI/Proteus.

Acknowledgments

We thank the anonymous reviewers of ASPLOS 2024, ISCA 2024, MICRO 2024, ASPLOS 2025, and ICS 2025 for their feedback. We thank the SAFARI Research Group members for providing a stimulating intellectual environment. We acknowledge the generous gifts from our industrial partners, including Google, Huawei, Intel, and Microsoft. This work is supported in part by the ETH Future Computing Laboratory (EFCL), Huawei ZRC Storage Team, Semiconductor Research Corporation, AI Chip Center for Emerging Smart Systems (ACCESS), sponsored by InnoHK funding, Hong Kong SAR, and European Union's Horizon programme for research and innovation [101047160 - BioPIM]. An extended version of this paper is available at [173].

References

[1] S. Ghose, A. Boroumand *et al.*, "Processing-in-Memory: A Workload-Driven Perspective," *IBM JRD*, 2019.

[2] O. Mutlu, S. Ghose *et al.*, "A Modern Primer on Processing in Memory," in *Emerging Computing: From Devices to Systems — Looking Beyond Moore and Von Neumann.* Springer, 2021.

[3] G. F. Oliveira, J. Gómez-Luna *et al.*, "DAMOV: A New Methodology and Benchmark Suite for Evaluating Data Movement Bottlenecks," *IEEE Access*, 2021.

[4] S. Ghose, K. Hsieh *et al.*, "The Processing-in-Memory Paradigm: Mechanisms to Enable Adoption," in *Beyond-CMOS Technologies for Next Generation Computer Design*, 2019.

[5] O. Mutlu, S. Ghose *et al.*, "Processing Data Where It Makes Sense: Enabling In-Memory Computation," *MicPro*, 2019.

[6] O. Mutlu, S. Ghose *et al.*, "Enabling Practical Processing in and Near Memory for Data-Intensive Computing," in *DAC*, 2019.

[7] O. Mutlu and L. Subramanian, "Research Problems and Opportunities in Memory Systems," *SUPERFRI*, 2014.

[8] O. Mutlu, "Memory Scaling: A Systems Architecture Perspective," in *IMW*, 2013.

[9] G. H. Loh, N. Jayasena *et al.*, "A Processing in Memory Taxonomy and a Case for Studying Fixed-Function PIM," in *WoNDP*, 2013.

[10] R. Balasubramonian, J. Chang *et al.*, "Near-Data Processing: Insights from a MICRO-46 Workshop," *IEEE Micro*, 2014.

[11] H. S. Stone, "A Logic-in-Memory Computer," *IEEE TC*, 1970.

[12] A. Saulsbury, F. Pong, and A. Nowatzyk, "Missing the Memory Wall: The Case for Processor/Memory Integration," in *ISCA*, 1996.

[13] A. Farmahini-Farahani, J. H. Ahn *et al.*, "NDA: Near-DRAM Acceleration Architecture Leveraging Commodity DRAM Devices and Standard Memory Modules," in *HPCA*, 2015.

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

[15] F. Devaux, "The True Processing in Memory Accelerator," in *Hot Chips*, 2019.

[16] N. M. Ghiasi, J. Park *et al.*, "GenStore: A High-Performance and Energy-Efficient In-Storage Computing System for Genome Sequence Analysis," in *ASPLOS*, 2022.

[17] J. Gómez-Luna, I. El Hajj *et al.*, "Benchmarking Memory-Centric Computing Systems: Analysis of Real Processing-in-Memory Hardware," in *CUT*, 2021.

[18] J. Gómez-Luna, I. E. Hajj *et al.*, "Benchmarking a New Paradigm: An Experimental Analysis of a Real Processing-in-Memory Architecture," arXiv:2105.03814 [cs.AR], 2021.

[19] J. Gómez-Luna, I. El Hajj *et al.*, "Benchmarking a New Paradigm: Experimental Analysis and Characterization of a Real Processing-in-Memory System," *IEEE Access*, 2022.

[20] C. Giannoula, N. Vijaykumar *et al.*, "SynCron: Efficient Synchronization Support for Near-Data-Processing Architectures," in *HPCA*, 2021.

[21] G. Singh, D. Diamantopoulos *et al.*, "NERO: A Near High-Bandwidth Memory Stencil Accelerator for Weather Prediction Modeling," in *FPL*, 2020.

[22] S. Lee, K. Kim *et al.*, "A 1ynm 1.25V 8Gb, 16Gb/s/pin GDDR6-based Accelerator-in-Memory Supporting 1TFLOPS MAC Operation and Various Activation Functions for Deep-Learning Applications," in *ISSCC*, 2022.

[23] L. Ke, X. Zhang *et al.*, "Near-Memory Processing in Action: Accelerating Personalized Recommendation with AxDIMM," *IEEE Micro*, 2021.

[24] C. Giannoula, I. Fernandez *et al.*, "SparseP: Towards Efficient Sparse Matrix Vector Multiplication on Real Processing-in-Memory Architectures," in *SIGMETRICS*, 2022.

[25] H. Shin, D. Kim *et al.*, "McDRAM: Low Latency and Energy-Efficient Matrix Computations in DRAM," *IEEE TCADICS*, 2018.

[26] S. Cho, H. Choi *et al.*, "McDRAM v2: In-Dynamic Random Access Memory Systolic Array Accelerator to Address the Large Model Problem in Deep Neural Networks on the Edge," *IEEE Access*, 2020.

[27] A. Denzler, R. Bera *et al.*, "Casper: Accelerating Stencil Computation using Near-Cache Processing," arXiv:2112.14216 [cs.AR], 2021.

[28] H. Asghari-Moghaddam, Y. H. Son *et al.*, "Chameleon: Versatile and Practical Near-DRAM Acceleration Architecture for Large Memory Systems," in *MICRO*, 2016.

[29] D. Patterson, T. Anderson *et al.*, "A Case for Intelligent RAM," *IEEE Micro*, 1997.

[30] D. G. Elliott, M. Stumm *et al.*, "Computational RAM: Implementing Processors in Memory," *Design and Test of Computers*, 1999.

[31] M. A. Z. Alves, P. C. Santos *et al.*, "Saving Memory Movements Through Vector Processing in the DRAM," in *CASES*, 2015.

[32] S. L. Xi, O. Babarinsa *et al.*, "Beyond the Wall: Near-Data Processing for Databases," in *DaMoN*, 2015.

[33] W. Sun, Z. Li *et al.*, "ABC-DIMM: Alleviating the Bottleneck of Communication in DIMM-Based Near-Memory Processing with Inter-DIMM Broadcast," in *ISCA*, 2021.

[34] K. K. Matam, G. Koo *et al.*, "GraphSSD: Graph Semantics Aware SSD," in *ISCA*, 2019.

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

[36] M. Hall, P. Kogge *et al.*, "Mapping Irregular Applications to DIVA, a PIM-Based Data-Intensive Architecture," in *SC*, 1999.

[37] M. A. Z. Alves, P. C. Santos *et al.*, "Opportunities and Challenges of Performing Vector Operations Inside the DRAM," in *MEMSYS*, 2015.

[38] E. Lockerman, A. Feldmann *et al.*, "Livia: Data-Centric Computing Throughout the Memory Hierarchy," in *ASPLOS*, 2020.

[39] J. Ahn, S. Hong *et al.*, "A Scalable Processing-in-Memory Accelerator for Parallel Graph Processing," in *ISCA*, 2015.

[40] L. Nai, R. Hadidi *et al.*, "GraphPIM: Enabling Instruction-Level PIM Offloading in Graph Computing Frameworks," in *HPCA*, 2017.

[41] A. Boroumand, S. Ghose *et al.*, "Google Workloads for Consumer Devices: Mitigating Data Movement Bottlenecks," in *ASPLOS*, 2018.

[42] A. Boroumand, S. Ghose *et al.*, "LazyPIM: An Efficient Cache Coherence Mechanism for Processing-in-Memory," *CAL*, 2017.

[43] D. Zhang, N. Jayasena *et al.*, "TOP-PIM: Throughput-Oriented Programmable Processing in Memory," in *HPDC*, 2014.

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

[45] J. S. Kim, D. S. Cali *et al.*, "GRIM-Filter: Fast Seed Location Filtering in DNA Read Mapping Using Processing-in-Memory Technologies," *BMC Genomics*, 2018.

[46] M. Drumond, A. Daglis *et al.*, "The Mondrian Data Engine," in *ISCA*, 2017.

[47] P. C. Santos, G. F. Oliveira *et al.*, "Operand Size Reconfiguration for Big Data Processing in Memory," in *DATE*, 2017.

[48] G. F. Oliveira, P. C. Santos *et al.*, "NIM: An HMC-Based Machine for Neuron Computation," in *ARC*, 2017.

[49] J. Ahn, S. Yoo *et al.*, "PIM-Enabled Instructions: A Low-Overhead, Locality-Aware Processing-in-Memory Architecture," in *ISCA*, 2015.

[50] M. Gao, J. Pu *et al.*, "TETRIS: Scalable and Efficient Neural Network Acceleration with 3D Memory," in *ASPLOS*, 2017.

[51] D. Kim, J. Kung *et al.*, "Neurocube: A Programmable Digital Neuromorphic Architecture with High-Density 3D Memory," in *ISCA*, 2016.

[52] P. Gu, S. Li *et al.*, "Leveraging 3D Technologies for Hardware Security: Opportunities and Challenges," in *GLSVLSI*, 2016.

[53] A. Boroumand, S. Ghose *et al.*, "CoNDA: Efficient Cache Coherence Support for Near-Data Accelerators," in *ISCA*, 2019.

[54] K. Hsieh, E. Ebrahimi *et al.*, "Transparent Offloading and Mapping (TOM) Enabling Programmer-Transparent Near-Data Processing in GPU Systems," in *ISCA*, 2016.

[55] D. S. Cali, G. S. Kalsi *et al.*, "GenASM: A High-Performance, Low-Power Approximate String Matching Acceleration Framework for Genome Sequence Analysis," in *MICRO*, 2020.

[56] S. H. Pugsley, J. Jestes *et al.*, "NDC: Analyzing the Impact of 3D-Stacked Memory+Logic Devices on MapReduce Workloads," in *ISPASS*, 2014.

[57] A. Pattnaik, X. Tang *et al.*, "Scheduling Techniques for GPU Architectures with Processing-in-Memory Capabilities," in *PACT*, 2016.

[58] B. Akin, F. Franchetti, and J. C. Hoe, "Data Reorganization in Memory Using 3D-Stacked DRAM," in *ISCA*, 2015.

[59] K. Hsieh, S. Khan *et al.*, "Accelerating Pointer Chasing in 3D-Stacked Memory: Challenges, Mechanisms, Evaluation," in *ICCD*, 2016.

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

[61] A. Boroumand, S. Ghose *et al.*, "Mitigating Edge Machine Learning Inference Bottlenecks: An Empirical Study on Accelerating Google Edge Models," arXiv:2103.00768 [cs.AR], 2021.

[62] A. Boroumand, S. Ghose *et al.*, "Google Neural Network Models for Edge Devices: Analyzing and Mitigating Machine Learning Inference Bottlenecks," in *PACT*, 2021.

[63] A. Boroumand, S. Ghose *et al.*, "Polynesia: Enabling High-Performance and Energy-Efficient Hybrid Transactional/Analytical Databases with Hardware/Software Co-Design," in *ICDE*, 2022.

[64] A. Boroumand, S. Ghose *et al.*, "Polynesia: Enabling Effective Hybrid Transactional/Analytical Databases with Specialized Hardware/Software Co-Design," arXiv:2103.00798 [cs.AR], 2021.

[65] A. Boroumand, "Practical Mechanisms for Reducing Processor-Memory Data Movement in Modern Workloads," Ph.D. dissertation, Carnegie Mellon University, 2020.

[66] M. Besta, R. Kanakagiri *et al.*, "SISA: Set-Centric Instruction Set Architecture for Graph Mining on Processing-in-Memory Systems," in *MICRO*, 2021.

[67] I. Fernandez, R. Quislant *et al.*, "NATSA: A Near-Data Processing Accelerator for Time Series Analysis," in *ICCD*, 2020.

[68] G. Singh, G. *et al.*, "NAPEL: Near-Memory Computing Application Performance Prediction via Ensemble Learning," in *DAC*, 2019.

[69] Y.-C. Kwon, S. H. Lee *et al.*, "A 20nm 6GB Function-in-Memory DRAM, Based on HBM2 with a 1.2 TFLOPS Programmable Computing Unit using Bank-Level Parallelism, for Machine Learning Applications," in *ISSCC*, 2021.

[70] S. Lee, S.-h. Kang *et al.*, "Hardware Architecture and Software Stack for PIM Based on Commercial DRAM Technology: Industrial Product," in *ISCA*, 2021.

[71] D. Niu, S. Li *et al.*, "184QPS/W 64Mb/mm^2 3D Logic-to-DRAM Hybrid Bonding with Process-Near-Memory Engine for Recommendation System," in *ISSCC*, 2022.

[72] Q. Zhu, T. Graf *et al.*, "Accelerating Sparse Matrix-Matrix Multiplication with 3D-Stacked Logic-in-Memory Hardware," in *HPEC*, 2013.

[73] E. Azarkhish, C. Pfister *et al.*, "Logic-Base Interconnect Design for Near Memory Computing in the Smart Memory Cube," *IEEE VLSI*, 2016.

[74] E. Azarkhish, D. Rossi *et al.*, "Neurostream: Scalable and Energy Efficient Deep Learning with Smart Memory Cubes," *TPDS*, 2018.

[75] Q. Guo, N. Alachiotis *et al.*, "3D-Stacked Memory-Side Acceleration: Accelerator and System Design," in *WoNDP*, 2014.

[76] J. P. C. de Lima, P. C. Santos *et al.*, "Design Space Exploration for PIM Architectures in 3D-Stacked Memories," in *CF*, 2018.

[77] B. Akın, J. C. Hoe, and F. Franchetti, "HAMLeT: Hardware Accelerated Memory Layout Transform within 3D-Stacked DRAM," in *HPEC*, 2014.

[78] Y. Huang, L. Zheng *et al.*, "A Heterogeneous PIM Hardware-Software Co-Design for Energy-Efficient Graph Processing," in *IPDPS*, 2020.

[79] G. Dai, T. Huang *et al.*, "GraphH: A Processing-in-Memory Architecture for Large-Scale Graph Processing," *TCAD*, 2018.

[80] J. Liu, H. Zhao *et al.*, "Processing-in-Memory for Energy-Efficient Neural Network Training: A Heterogeneous Approach," in *MICRO*, 2018.

[81] P.-A. Tsai, C. Chen, and D. Sanchez, "Adaptive Scheduling for Systems with Asymmetric Memory Hierarchies," in *MICRO*, 2018.

[82] P. Gu, X. Xie *et al.*, "iPIM: Programmable In-Memory Image Processing Accelerator using Near-Bank Architecture," in *ISCA*, 2020.

[83] A. Farmahini-Farahani, J. H. Ahn *et al.*, "DRAMA: An Architecture for Accelerated Processing Near Memory," *Computer Architecture Letters*, 2014.

[84] H. Asghari-Moghaddam, A. Farmahini-Farahani *et al.*, "Near-DRAM Acceleration with Single-ISA Heterogeneous Processing in Standard Memory Modules," *IEEE Micro*, 2016.

[85] J. Huang, R. R. Puli *et al.*, "Active-Routing: Compute on the Way for Near-Data Processing," in *HPCA*, 2019.

[86] C. D. Kersey, H. Kim, and S. Yalamanchili, "Lightweight SIMT Core Designs for Intelligent 3D Stacked DRAM," in *MEMSYS*, 2017.

[87] J. Li, X. Wang *et al.*, "PIMS: A Lightweight Processing-in-Memory Accelerator for Stencil Computations," in *MEMSYS*, 2019.

[88] J. S. Kim, D. Senol *et al.*, "GRIM-Filter: Fast Seed Filtering in Read Mapping using Emerging Memory Technologies," arXiv:1708.04329 [q-bio.GN], 2017.

[89] A. Boroumand, S. Ghose *et al.*, "LazyPIM: Efficient Support for Cache Coherence in Processing-in-Memory Architectures," arXiv:1706.03162 [cs.AR], 2017.

[90] Y. Zhuo, C. Wang *et al.*, "GraphQ: Scalable PIM-Based Graph Processing," in *MICRO*, 2019.

[91] M. Zhang, Y. Zhuo *et al.*, "GraphP: Reducing Communication for PIM-Based Graph Processing with Efficient Data Partition," in *HPCA*, 2018.

[92] H. Lim and G. Park, "Triple Engine Processor (TEP): A Heterogeneous Near-Memory Processor for Diverse Kernel Operations," *TACO*, 2017.

[93] E. Azarkhish, D. Rossi *et al.*, "A Case for Near Memory Computation Inside the Smart Memory Cube," in *EMS*, 2016.

[94] M. A. Z. Alves, M. Diener *et al.*, "Large Vector Extensions Inside the HMC," in *DATE*, 2016.

[95] J. Jang, J. Heo *et al.*, "Charon: Specialized Near-Memory Processing Architecture for Clearing Dead Objects in Memory," in *MICRO*, 2019.

[96] R. Nair, S. F. Antao *et al.*, "Active Memory Cube: A Processing-in-Memory Architecture for Exascale Systems," *IBM JRD*, 2015.

[97] R. Hadidi, L. Nai *et al.*, "CAIRO: A Compiler-Assisted Technique for Enabling Instruction-Level Offloading of Processing-in-Memory," *TACO*, 2017.

[98] P. C. Santos, G. F. Oliveira *et al.*, "Processing in 3D Memories to Speed Up Operations on Complex Data Structures," in *DATE*, 2018.

[99] P. Chi, S. Li *et al.*, "PRIME: A Novel Processing-in-Memory Architecture for Neural Network Computation in ReRAM-Based Main Memory," in *ISCA*, 2016.

[100] A. Shafiee, A. Nag *et al.*, "ISAAC: A Convolutional Neural Network Accelerator with In-Situ Analog Arithmetic in Crossbars," in *ISCA*, 2016.

[101] V. Seshadri, D. Lee *et al.*, "Ambit: In-Memory Accelerator for Bulk Bitwise Operations Using Commodity DRAM Technology," in *MICRO*, 2017.

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

[103] S. Li, D. Niu *et al.*, "DRISA: A DRAM-Based Reconfigurable In-Situ Accelerator," in *MICRO*, 2017.

[104] V. Seshadri, Y. Kim *et al.*, "RowClone: Fast and Energy-Efficient In-DRAM Bulk Data Copy and Initialization," in *MICRO*, 2013.

[105] 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.

[106] Q. Deng, L. Jiang *et al.*, "DrAcc: A DRAM Based Accelerator for Accurate CNN Inference," in *DAC*, 2018.

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

[108] L. Song, Y. Zhuo *et al.*, "GraphR: Accelerating Graph Processing Using ReRAM," in *HPCA*, 2018.

[109] L. Song, X. Qian *et al.*, "PipeLayer: A Pipelined ReRAM-Based Accelerator for Deep Learning," in *HPCA*, 2017.

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

[111] C. Eckert, X. Wang *et al.*, "Neural Cache: Bit-Serial In-Cache Acceleration of Deep Neural Networks," in *ISCA*, 2018.

[112] S. Aga, S. Jeloka *et al.*, "Compute Caches," in *HPCA*, 2017.

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

[114] V. Seshadri, D. Lee *et al.*, "Buddy-RAM: Improving the Performance and Efficiency of Bulk Bitwise Operations Using DRAM," arXiv:1611.09988 [cs.AR], 2016.

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

[116] V. Seshadri, Y. Kim *et al.*, "RowClone: Accelerating Data Movement and Initialization Using DRAM," arXiv:1805.03502 [cs.AR], 2018.

[117] V. Seshadri, K. Hsieh *et al.*, "Fast Bulk Bitwise AND and OR in DRAM," *CAL*, 2015.

[118] S. Li, C. Xu *et al.*, "Pinatubo: A Processing-in-Memory Architecture for Bulk Bitwise Operations in Emerging Non-Volatile Memories," in *DAC*, 2016.

[119] J. D. Ferreira, G. Falcao *et al.*, "pLUTo: In-DRAM Lookup Tables to Enable Massively Parallel General-Purpose Computation," arXiv:2104.07699 [cs.AR], 2021.

[120] J. D. Ferreira, G. Falcao *et al.*, "pLUTo: Enabling Massively Parallel Computation in DRAM via Lookup Tables," in *MICRO*, 2022.

[121] M. Imani, S. Gupta *et al.*, "FloatPIM: In-Memory Acceleration of Deep Neural Network Training with High Precision," in *ISCA*, 2019.

[122] Z. He, L. Yang *et al.*, "Sparse BD-Net: A Multiplication-Less DNN with Sparse Binarized Depth-Wise Separable Convolution," *JETC*, 2020.

[123] J. Park, R. Azizi *et al.*, "Flash-Cosmos: In-Flash Bulk Bitwise Operations Using Inherent Computation Capability of NAND Flash Memory," in *MICRO*, 2022.

[124] M. S. Truong, L. Shen *et al.*, "Adapting the RACER Architecture to Integrate Improved In-ReRAM Logic Primitives," *JETCAS*, 2022.

[125] M. S. Truong, E. Chen *et al.*, "RACER: Bit-Pipelined Processing Using Resistive Memory," in *MICRO*, 2021.

[126] A. Olgun, M. Patel *et al.*, "QUAC-TRNG: High-Throughput True Random Number Generation Using Quadruple Row Activation in Commodity DRAMs," in *ISCA*, 2021.

[127] J. S. Kim, M. Patel *et al.*, "D-RaNGe: Using Commodity DRAM Devices to Generate True Random Numbers With Low Latency and High Throughput," in *HPCA*, 2019.

[128] J. S. Kim, M. Patel *et al.*, "The DRAM Latency PUF: Quickly Evaluating Physical Unclonable Functions by Exploiting the Latency-Reliability Tradeoff in Modern Commodity DRAM Devices," in *HPCA*, 2018.

[129] F. N. Bostancı, A. Olgun *et al.*, "DR-STRaNGe: End-to-End System Design for DRAM-Based True Random Number Generators," in *HPCA*, 2022.

[130] A. Olgun, J. G. Luna *et al.*, "PiDRAM: A Holistic End-to-End FPGA-Based Framework for Processing-in-DRAM," *TACO*, 2022.

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

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

[133] S. Li, A. O. Glova *et al.*, "SCOPE: A Stochastic Computing Engine for DRAM-Based In-Situ Accelerator," in *MICRO*, 2018.

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

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

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

[137] L. Orosa, Y. Wang *et al.*, "CODIC: A Low-Cost Substrate for Enabling Custom In-DRAM Functionalities and Optimizations," in *ISCA*, 2021.

[138] M. Sharad, D. Fan, and K. Roy, "Ultra Low Power Associative Computing with Spin Neurons and Resistive Crossbar Memory," in *DAC*, 2013.

[139] S. H. S. Rezaei, M. Modarressi *et al.*, "NoM: Network-on-Memory for Inter-Bank Data Transfer in Highly-Banked Memories," *CAL*, 2020.

[140] I. E. Yuksel, Y. C. Tuğrul *et al.*, "Functionally-Complete Boolean Logic in Real DRAM Chips: Experimental Characterization and Analysis," in *HPCA*, 2024.

[141] G. F. Oliveira, A. Olgun *et al.*, "MIMDRAM: An End-to-End Processing-Using-DRAM System for High-Throughput, Energy-Efficient and Programmer-Transparent Multiple-Instruction Multiple-Data Computing," in *HPCA*, 2024.

[142] I. E. Yuksel, Y. C. Tugrul *et al.*, "Simultaneous Many-Row Activation in Off-the-Shelf DRAM Chips: Experimental Characterization and Analysis," in *DSN*, 2024.

[143] N. Hajinazar, G. F. Oliveira *et al.*, "SIMDRAM: A Framework for Bit-Serial SIMD Processing Using DRAM," in *ASPLOS*, 2021.

[144] X. Peng, Y. Wang, and M.-C. Yang, "CHOPPER: A Compiler Infrastructure for Programmable Bit-Serial SIMD Processing Using Memory In DRAM," in *HPCA*, 2023.

[145] M. Zhou, W. Xu *et al.*, "TransPIM: A Memory-Based Acceleration via Software-Hardware Co-Design for Transformer," in *HPCA*, 2022.

[146] J. Park, J. Choi *et al.*, "AttAcc! Unleashing the Power of PIM for Batched Transformer-Based Generative Model Inference," in *ASPLOS*, 2024.

[147] Q. Deng, Y. Zhang *et al.*, "LAcc: Exploiting Lookup Table-Based Fast and Accurate Vector Multiplication in DRAM-Based CNN Accelerator," in *DAC*, 2019.

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

[149] H. Shin, R. Park, and J. W. Lee, "A Processing-using-Memory Architecture for Commodity DRAM Devices with Enhanced Compatibility and Reliability," in *ICCAD*, 2024.

[150] G. Pekhimenko, V. Seshadri *et al.*, "Base-Delta-Immediate Compression: Practical Data Compression for On-Chip Caches," in *PACT*, 2012.

[151] A. R. Alameldeen and D. A. Wood, "Adaptive Cache Compression for High-Performance Processors," in *ISCA*, 2004.

[152] M. M. Islam and P. Stenstrom, "Characterization and Exploitation of Narrow-Width Loads: The Narrow-Width Cache Approach," in *CASES*, 2010.

[153] O. Ergin, O. Unsal *et al.*, "Exploiting Narrow Values for Soft Error Tolerance," *CAL*, 2006.

[154] D. Brooks and M. Martonosi, "Dynamically Exploiting Narrow Width Operands to Improve Processor Power and Performance," in *HPCA*, 1999.

[155] O. Ergin, D. Balkan *et al.*, "Register Packing: Exploiting Narrow-Width Operands for Reducing Register File Pressure," in *MICRO*, 2004.

[156] M. Budiu, M. Sakr *et al.*, "BitValue Inference: Detecting and Exploiting Narrow Bitwidth Computations," in *Euro-Par*, 2000.

[157] P. R. Wilson, S. F. Kaplan, and Y. Smaragdakis, "The Case for Compressed Caching in Virtual Memory Systems," in *USENIX ATC*, 1999.

[158] G. Pekhimenko, "Practical Data Compression for Modern Memory Hierarchies," Ph.D. dissertation, Carnegie Mellon University, 2016.

[159] G. Pekhimenko, V. Seshadri *et al.*, "Linearly Compressed Pages: A Low-Complexity, Low-Latency Main Memory Compression Framework," in *MICRO*, 2013.

[160] Homer, *The Odyssey*, B. Knox, Ed. Penguin Classics, 2006.

[161] Y. Kim, V. Seshadri *et al.*, "A Case for Exploiting Subarray-Level Parallelism (SALP) in DRAM," in *ISCA*, 2012.

[162] K. K. Chang, P. J. Nair *et al.*, "Low-Cost Inter-Linked Subarrays (LISA): Enabling Fast Inter-Subarray Data Movement in DRAM," in *HPCA*, 2016.

[163] C. Guest and T. K. Gaylord, "Truth-Table Look-Up Optical Processing Utilizing Binary and Residue Arithmetic," *Applied Optics*, 1980.

[164] D. S. Phatak and I. Koren, "Hybrid Signed-Digit Number Systems: A Unified Framework for Redundant Number Representations with Bounded Carry Propagation Chains," *TC*, 1994.

[165] M. Lapointe, H. T. Huynh, and P. Fortier, "Systematic Design of Pipelined Recursive Filters," *TC*, 1993.

[166] J. Olivares, J. Hormigo *et al.*, "SAD Computation based on Online Arithmetic for Motion Estimation," *Microprocessors and Microsystems*, 2006.

[167] J. Olivares, J. Hormigo *et al.*, "Minimum Sum of Absolute Differences Implementation in a Single FPGA Device," in *FPL*, 2004.

[168] M. D. Brown and Y. N. Patt, "Using Internal Redundant Representations and Limited Bypass to Support Pipelined Adders and Register Files," in *HPCA*, 2002.

[169] L.-N. Pouchet, "PolyBench: The Polyhedral Benchmark Suite," https://www.cs.colostate.edu/~pouchet/software/polybench/.

[170] Standard Performance Evaluation Corp., "SPEC CPU2017 Benchmarks," http://www.spec.org/cpu2017/.

[171] R. M. Yoo, A. Romano, and C. Kozyrakis, "Phoenix Rebirth: Scalable MapReduce on a Large-Scale Shared-Memory System," in *IISWC*, 2009.

[172] S. Che, M. Boyer *et al.*, "Rodinia: A Benchmark Suite for Heterogeneous Computing," in *IISWC*, 2009.

[173] G. F. Oliveira, M. Kabra *et al.*, "*Proteus*: Enabling High-Performance Processing-Using-DRAM with Dynamic Bit-Precision, Adaptive Data Representation, and Flexible Arithmetic," arXiv:2501.17466 [cs.AR], 2025.

[174] H. Hassan, M. Patel *et al.*, "CROW: A Low-Cost Substrate for Improving DRAM Performance, Energy Efficiency, and Reliability," in *ISCA*, 2019.

[175] S. Ghose, T. Li *et al.*, "Demystifying Complex Workload–DRAM Interactions: An Experimental Study," in *SIGMETRICS*, 2020.

[176] S. Ghose, A. G. Yaglikçi *et al.*, "What Your DRAM Power Models Are Not Telling You: Lessons from a Detailed Experimental Study," in *SIGMETRICS*, 2018.

[177] Y. Kim, W. Yang, and O. Mutlu, "Ramulator: A Fast and Extensible DRAM Simulator," *CAL*, 2016.

[178] T. Zhang, K. Chen *et al.*, "Half-DRAM: A High-Bandwidth and Low-Power DRAM Architecture from the Rethinking of Fine-Grained Activation," in *ISCA*, 2014.

[179] H. Hassan, G. Pekhimenko *et al.*, "ChargeCache: Reducing DRAM Latency by Exploiting Row Access Locality," in *HPCA*, 2016.

[180] D. Lee, Y. Kim *et al.*, "Tiered-Latency DRAM: A Low Latency and Low Cost DRAM Architecture," in *HPCA*, 2013.

[181] K. K. Chang, A. G. Yağlıkçı *et al.*, "Understanding Reduced-Voltage Operation in Modern DRAM Devices: Experimental Characterization, Analysis, and Mechanisms," in *SIGMETRICS*, 2017.

[182] K. K. Chang, "Understanding and Improving the Latency of DRAM-Based Memory Systems," Ph.D. dissertation, Carnegie Mellon University, 2017.

[183] K. K. Chang, A. Kashyap *et al.*, "Understanding Latency Variation in Modern DRAM Chips: Experimental Characterization, Analysis, and Optimization," in *SIGMETRICS*, 2016.

[184] K. K.-W. Chang, D. Lee *et al.*, "Improving DRAM Performance by Parallelizing Refreshes with Accesses," in *HPCA*, 2014.

[185] D. Lee, Y. Kim *et al.*, "Adaptive-Latency DRAM: Optimizing DRAM Timing for the Common-Case," in *HPCA*, 2015.

[186] D. Lee, S. Khan *et al.*, "Reducing DRAM Latency by Exploiting Design-Induced Latency Variation in Modern DRAM Chips," in *SIGMETRICS*, 2017.

[187] D. Lee, "Reducing DRAM Latency at Low Cost by Exploiting Heterogeneity," Ph.D. dissertation, Carnegie Mellon University, 2016.

[188] D. Lee, L. Subramanian *et al.*, "Decoupled Direct Memory Access: Isolating CPU and IO Traffic by Leveraging a Dual-Data-Port DRAM," in *PACT*, 2015.

[189] J. Liu, B. Jaiyen *et al.*, "An Experimental Study of Data Retention Behavior in Modern DRAM Devices: Implications for Retention Time Profiling Mechanisms," in *ISCA*, 2013.

[190] J. Liu, B. Jaiyen *et al.*, "RAIDR: Retention-Aware Intelligent DRAM Refresh," in *ISCA*, 2012.

[191] V. Seshadri, T. Mullins *et al.*, "Gather-Scatter DRAM: In-DRAM Address Translation to Improve the Spatial Locality of Non-Unit Strided Accesses," in *MICRO*, 2015.

[192] E. Ipek, O. Mutlu *et al.*, "Self-Optimizing Memory Controllers: A Reinforcement Learning Approach," in *ISCA*, 2008.

[193] D. Lee, S. Ghose *et al.*, "Simultaneous Multi-Layer Access: Improving 3D-Stacked Memory Bandwidth at Low Cost," *TACO*, 2016.

[194] R. H. Dennard, "Field-Effect Transistor Memory," 1968, US Patent 3,387,286.

[195] B. Keeth, R. J. Baker *et al.*, *DRAM Circuit Design: Fundamental and High-Speed Topics*. John Wiley & Sons, 2007.

[196] M. Patel, "Enabling Effective Error Mitigation In Memory Chips That Use On-Die Error-Correcting Codes," Ph.D. dissertation, ETH Zürich, 2022.

[197] H. Hassan, "Improving DRAM Performance, Reliability, and Security by Rigorously Understanding Intrinsic DRAM Operation," Ph.D. dissertation, ETH Zürich, 2022.

[198] J. M. O'Connor, "Energy Efficient High Bandwidth DRAM for Throughput Processors," Ph.D. dissertation, The University of Texas at Austin, 2021.

[199] D. Lee, S. Khan *et al.*, "Design-Induced Latency Variation in Modern DRAM Chips: Characterization, Analysis, and Latency Reduction Mechanisms," in *SIGMETRICS*, 2017.

[200] V. Seshadri, "Simple DRAM and Virtual Memory Abstractions to Enable Highly Efficient Memory Subsystems," Ph.D. dissertation, Carnegie Mellon University, 2016.

[201] Y. Wang, L. Orosa *et al.*, "FIGARO: Improving System Performance via Fine-Grained In-DRAM Data Relocation and Caching," in *MICRO*, 2020.

[202] A. Olgun, F. Bostanci *et al.*, "Sectored DRAM: A Practical Energy-Efficient and High-Performance Fine-Grained DRAM Architecture," *TACO*, 2024.

[203] J. S. Kim, "Improving DRAM Performance, Security, and Reliability by Understanding and Exploiting DRAM Timing Parameter Margins," Ph.D. dissertation, Carnegie Mellon University, 2020.

[204] J. Kim, M. Patel *et al.*, "Solar-DRAM: Reducing DRAM Access Latency by Exploiting the Variation in Local Bitlines," in *ICCD*, 2018.

[205] C. Kim, D. Burger, and S. W. Keckler, "An Adaptive, Non-Uniform Cache Structure for Wire-Delay Dominated On-Chip Caches," in *ASPLOS*, 2002.
[206] K.-N. Lim, W.-J. Jang *et al.*, "A 1.2 V 23nm 6F2 4Gb DDR3 SDRAM with Local-Bitline Sense Amplifier, Hybrid LIO Sense Amplifier and Dummy-Less Array Architecture," in *ISSCC*, 2012.
[207] T. Takahashi, T. Sekiguchi *et al.*, "A Multigigabit DRAM Technology with 6F2 Open-Bitline Cell, Distributed Overdriven Sensing, and Stacked-Flash Fuse," *JSSC*, 2001.
[208] G. Duan and S. Wang, "Exploiting Narrow-Width Values for Improving Non-Volatile Cache Lifetime," in *DATE*, 2014.
[209] C. Molina, C. Aliagas *et al.*, "Non Redundant Data Cache," in *ISLPEL*, 2003.
[210] J. Kong and S. W. Chung, "Exploiting Narrow-Width Values for Process Variation-Tolerant 3-D Microprocessors," in *DAC*, 2012.
[211] G. Pekhimenko, E. Bolotin *et al.*, "Toggle-Aware Compression for GPUs," *CAL*, 2015.
[212] G. Pekhimenko, E. Bolotin *et al.*, "A Case for Toggle-Aware Compression for GPU Systems," in *HPCA*, 2016.
[213] G. Pekhimenko, T. Huberty *et al.*, "Exploiting Compressed Block Size as an Indicator of Future Reuse," in *HPCA*, 2015.
[214] X. Wang and W. Zhang, "GPU Register Packing: Dynamically Exploiting Narrow-Width Operands to Improve Performance," in *TrustCom*, 2017.
[215] J. Hu, S. Wang, and S. G. Ziavras, "In-Register Duplication: Exploiting Narrow-Width Value for Improving Register File Reliability," in *DSN*, 2006.
[216] S. Wang, J. Hu *et al.*, "Exploiting Narrow-Width Values for Thermal-Aware Register File Designs," in *DATE*, 2009.
[217] M. Özsoy, Y. O. Koçberber *et al.*, "Dynamic Register File Partitioning in Superscalar Microprocessors for Energy Efficiency," in *ICCD*, 2010.
[218] S. Mittal, H. Wang *et al.*, "Design and Analysis of Soft-Error Resilience Mechanisms for GPU Register File," in *VLSID*, 2017.
[219] O. Ergin, "Exploiting Narrow Values for Energy Efficiency in the Register Files of Superscalar Microprocessors," in *PATMOS*, 2006.
[220] M. Canesche, R. Ferreira *et al.*, "A Polynomial Time Exact Solution to the Bit-Aware Register Binding Problem," in *CC*, 2022.
[221] A. Canis, J. Choi *et al.*, "From Software to Accelerators with LegUp High-Level Synthesis," in *CASES*, 2013.
[222] C. Pilato and F. Ferrandi, "Bambu: A Modular Framework for the High Level Synthesis of Memory-Intensive Applications," in *FPL*, 2013.
[223] Y. Onur Koçberber, Y. Osmanlıoğlu, and O. Ergin, "Exploiting Narrow Values for Faster Parity Generation," *Microelectronics International*, 2009.
[224] Y. Osmanlioglu, Y. O. Koçberber, and O. Ergin, "Reducing Parity Generation Latency Through Input Value Aware Circuits," in *GLSVLSI*, 2009.
[225] M. Jang, J. Kim *et al.*, "ENCORE Compression: Exploiting Narrow-width Values for Quantized Deep Neural Networks," in *DATE*, 2022.
[226] J. Albericio, A. Delmás *et al.*, "Bit-Pragmatic Deep Neural Network Computing," in *MICRO*, 2017.
[227] I. B. Karsli, P. Reviriego *et al.*, "Enhanced Duplication: A Technique to Correct Soft Errors in Narrow Values," *CAL*, 2012.
[228] O. Ergin, O. Unsal *et al.*, "Reducing Soft Errors Through Operand Width Aware Policies," *TDSC*, 2008.
[229] C. Lattner, "LLVM and Clang: Next Generation Compiler Technology," in *BSDCan*, 2008.
[230] S. Sarda and M. Pandey, *LLVM Essentials*. Packt Publishing Ltd, 2015.
[231] B. C. Lopes and R. Auler, *Getting Started with LLVM Core Libraries*. Packt Publishing Ltd, 2014.
[232] A. Sampson, "LLVM for Grad Students," https://tinyurl.com/y3tyb7z2.
[233] M. H. Lipasti, B. R. Mestan, and E. Gunadi, "Physical Register Inlining," in *ISCA*, 2004.
[234] G. H. Loh, "Exploiting Data-Width Locality to Increase Superscalar Execution Bandwidth," in *MICRO*, 2002.
[235] M. Stephenson, J. Babb, and S. Amarasinghe, "Bidwidth Analysis with Application to Silicon Compilation," in *PLDI*, 2000.
[236] R. E. Rodrigues, V. H. S. Campos, and F. M. Q. Pereira, "A Fast and Low-Overhead Technique to Secure Programs Against Integer Overflows," in *CGO*, 2013.
[237] V. H. S. Campos, R. E. Rodrigues *et al.*, "Speed and Precision in Range Analysis," in *SBLP*, 2012.
[238] J. Cong, Y. Fan *et al.*, "Bitwidth-Aware Scheduling and Binding in High-level Synthesis," in *ASP-DAC*, 2005.
[239] Intel Corp., *Intel® 64 and IA-32 Architectures Software Developer's Manual, Vol. 3*, 2016.
[240] O. J. Bedrij, "Carry-Select Adder," *IEEE TC*, 1962.
[241] P. M. Kogge and H. S. Stone, "A Parallel Algorithm for the Efficient Solution of a General Class of Recurrence Equations," *IEEE TC*, 1973.
[242] R. E. Ladner and M. J. Fischer, "Parallel Prefix Computation," *JACM*, 1980.
[243] Brent and Kung, "A Regular Layout for Parallel Adders," *IEEE TC*, 1982.
[244] H. Makino, Y. Nakase *et al.*, "An 8.8-ns 54/SPL Times/54-bit Multiplier with High Speed Redundant Binary Architecture," *JSSC*, 1996.
[245] P. Shivakumar and N. P. Jouppi, "CACTI 3.0: An Integrated Cache Timing, Power, and Area Model," Compaq Computer Corporation, Tech. Rep. 2001/2, 2001.
[246] A. D. Booth, "A Signed Binary mMultiplication Technique," *The Quarterly Journal of Mechanics and Applied Mathematics*, 1951.
[247] A. A. Karatsuba and Y. P. Ofman, "Multiplication of Many-Digital Numbers by Automatic Computers," in *USSR Academy of Sciences*, 1962.
[248] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. MIT Press, 2016.
[249] S. Han, X. Liu *et al.*, "EIE: Efficient Inference Engine on Compressed Deep Neural Network," in *ISCA*, 2016.
[250] SAFARI Research Group, "DAMOV Benchmark Suite and Simulation Framework," https://github.com/CMU-SAFARI/DAMOV.
[251] N. Muralimanohar, R. Balasubramonian, and N. Jouppi, "Optimizing NUCA Organizations and Wiring Alternatives for Large Caches with CACTI 6.0," in *MICRO*, 2007.
[252] SAFARI Research Group, "Proteus Simulation Framework," https://github.com/CMU-SAFARI/Proteus.
[253] Intel Corp., "6th Generation Intel Core Processor Family Datasheet," http://www.intel.com/content/www/us/en/processors/core/.
[254] NVIDIA, "NVIDIA A100 Tensor Core GPU Architecture. White Paper," https://tinyurl.com/53a8easc, 2020.
[255] N. Firasta, M. Buxton *et al.*, "Intel AVX: New Frontiers in Performance Improvements and Energy Efficiency," Intel Corp., 2008, white paper.
[256] N. Binkert, B. Beckmann *et al.*, "The gem5 Simulator," *Comput. Archit. News*, 2011.
[257] SAFARI Research Group, "MIMDRAM Simulation Framework," https://github.com/CMU-SAFARI/MIMDRAM.
[258] H. Hassan, A. Olgun *et al.*, "A Case for Self-Managing DRAM Chips: Improving Performance, Efficiency, Reliability, and Security via Autonomous in-DRAM Maintenance Operations," arXiv:2207.13358 [cs.AR], 2022.
[259] M. Hähnel, B. Döbel *et al.*, "Measuring Energy Consumption for Short Code Paths Using RAPL," *SIGMETRICS*, 2012.
[260] NVIDIA Corp., "NVIDIA Management Library (NVML)," https://developer.nvidia.com/nvidia-management-library-nvml.

[261] Intel Corp., "10th Generation Intel Core Processor Family Datasheet," https://tinyurl.com/4fh5ze38.
[262] D. U. Lee, K. W. Kim *et al.*, "A 1.2V 8Gb 8-Channel 128GB/s High-Bandwidth Memory (HBM) Stacked DRAM with Effective Microbump I/O Test Methods Using 29nm Process and TSV," in *ISSCC*, 2014.
[263] O. Mutlu and T. Moscibroda, "Stall-Time Fair Memory Access Scheduling for Chip Multiprocessors," in *MICRO*, 2007.
[264] W. K. 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 5 630 096, 1997.
[265] JEDEC, *JESD79-5: DDR5 SDRAM Standard*, 2020.
[266] H. Lee, M. Kim *et al.*, "3D-FPIM: An Extreme Energy-Efficient DNN Acceleration System Using 3D NAND Flash-Based In-Situ PIM Unit," in *MICRO*, 2022.
[267] R. Zhou, S. Tabrizchi *et al.*, "P-PIM: A Parallel Processing-in-DRAM Framework Enabling Row Hammer Protection," in *DATE*, 2023.
[268] NVIDIA Corp., "NVIDIA/cutlass: CUDA Templates for Linear Algebra Subroutines," https://github.com/NVIDIA/cutlass.
[269] R. Zhou, A. Roohi *et al.*, "FlexiDRAM: A Flexible In-DRAM Framework to Enable Parallel General-Purpose Computation," in *ISLPED*, 2022.
[270] O. Leitersdorf, D. Leitersdorf *et al.*, "AritPIM: High-Throughput In-Memory Arithmetic," *IEEE Trans. Emerg. Topics Comput.*, 2023.
[271] A. Arora, A. Bhamburkar *et al.*, "CoMeFa: Deploying Compute-in-Memory on FPGAs for Deep Learning Acceleration," *Trans. Reconfigurable Technol. Syst.*, 2023.
[272] H. Caminal, Y. Chronis *et al.*, "Accelerating Database Analytic Query Workloads Using an Associative Processor," in *ISCA*, 2022.
[273] S. S. Wong, C. C. Tamarit, and J. F. Martínez, "PUMICE: Processing-using-Memory Integration with a Scalar Pipeline for Symbiotic Execution," in *DAC*, 2023.