# pLUTo

## Enabling Massively Parallel Computation in DRAM via Lookup Tables

Ferreira, Joao Dinis; Falcao, Gabriel; Gomez-Luna, Juan; Alser, Mohammed; Orosa, Lois; Sadrosadati, Mohammad; Kim, Jeremie S.; Oliveira, Geraldo F.; Shahroodi, Taha; More Authors

**DOI**
10.1109/MICRO56248.2022.00067

**Publication date**
2022

**Document Version**
Final published version

**Published in**
Proceedings - 2022 55th Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 2022

**Citation (APA)**
Ferreira, J. D., Falcao, G., Gomez-Luna, J., Alser, M., Orosa, L., Sadrosadati, M., Kim, J. S., Oliveira, G. F., Shahroodi, T., & More Authors (2022). pLUTo: Enabling Massively Parallel Computation in DRAM via Lookup Tables. In *Proceedings - 2022 55th Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 2022* (pp. 900-919). (Proceedings of the Annual International Symposium on Microarchitecture, MICRO; Vol. 2022-October). IEEE. https://doi.org/10.1109/MICRO56248.2022.00067

2022 55th IEEE/ACM International Symposium on Microarchitecture (MICRO) | 978-1-6654-6272-3/22/$31.00 ©2022 IEEE | DOI: 10.1109/MICRO56248.2022.00067

# pLUTo: Enabling Massively Parallel Computation in DRAM via Lookup Tables

João Dinis Ferreira[§] Gabriel Falcao[†] Juan Gómez-Luna[§] Mohammed Alser[§]
Lois Orosa[§▽] Mohammad Sadrosadati[§] Jeremie S. Kim[§] Geraldo F. Oliveira[§]
Taha Shahroodi[‡] Anant Nori[⋆] Onur Mutlu[§]

[§]*ETH Zürich* [†]*IT, University of Coimbra* [▽]*Galicia Supercomputing Center* [‡]*TU Delft* [⋆]*Intel*

*Data movement between the main memory and the processor is a key contributor to execution time and energy consumption in memory-intensive applications. This* data movement bottleneck *can be alleviated using Processing-in-Memory (PiM). One category of PiM is Processing-using-Memory (PuM), in which computation takes place* inside *the memory array by exploiting intrinsic analog properties of the memory device. PuM yields high performance and energy efficiency, but existing PuM techniques support a limited range of operations. As a result, current PuM architectures cannot efficiently perform some complex operations (e.g., multiplication, division, exponentiation) without large increases in chip area and design complexity.*

*To overcome these limitations of existing PuM architectures, we introduce* **pLUTo** ***(processing-using-memory with lookup table (LUT) operations)****, a DRAM-based PuM architecture that leverages the high storage density of DRAM to enable the massively parallel storing and querying of lookup tables (LUTs). The key idea of pLUTo is to replace complex operations with low-cost, bulk memory reads (i.e., LUT queries) instead of relying on complex extra logic.*

*We evaluate pLUTo across 11 real-world workloads that showcase the limitations of prior PuM approaches and show that our solution outperforms optimized CPU and GPU baselines by an average of* 713× *and* 1.2×*, respectively, while simultaneously reducing energy consumption by an average of* 1855× *and* 39.5×*. Across these workloads, pLUTo outperforms state-of-the-art PiM architectures by an average of* 18.3×*. We also show that different versions of pLUTo provide different levels of flexibility and performance at different additional DRAM area overheads (between 10.2% and 23.1%). pLUTo's source code and all scripts required to reproduce the results of this paper are openly and fully available at* `https://github.com/CMU-SAFARI/pLUTo`.

## 1. Introduction

*Processing-in-Memory* (PiM) is a promising paradigm that augments a system's memory with compute capability [1–5] to alleviate the *data movement bottleneck* between processing and memory units [2, 6–17]. PiM architectures can be classified into two categories [1, 18]: 1) **Processing-near-Memory (PnM)**, where computation takes place in dedicated processing elements (e.g., accelerators [11, 19–43], processing cores [11, 30–32, 41, 44–58], reconfigurable logic [59–63]) placed near the memory array (e.g., [11, 19–41, 44–67]), and 2) **Processing-using-Memory (PuM)**, where computation takes place *inside* the memory array by exploiting intrinsic analog operational properties of the memory device (e.g., [41, 55, 68–91]).

In DRAM-based PnM, data is transferred from the DRAM array to nearby processors or specialized accelerators, which could be 1) part of the DRAM chip, but separate from the DRAM array [22, 36–38, 42, 43, 51–54, 56, 58], e.g., near the DRAM banks, 2) integrated into the logic layer of 3D-stacked memories [11, 20, 21, 23–25, 28–32, 34, 39, 44–50, 55, 57, 59, 60, 66, 67], or 3) inside the memory controller [35, 92, 93]. PnM enables the design of flexible substrates that support a diverse range of operations. However, the performance, efficiency, and scalability of near-bank PnM architectures [22, 36–38, 42, 43, 51–54, 56, 58] can be limited by design and fabrication challenges, such as 1) the difficulty in designing complex logic due to the limited number of DRAM metal layers [94, 95], and 2) the inefficiency of the DRAM process for the implementation of digital logic due to its heavy optimization for memory density [51, 54]. In 3D-stacked memories, the logic layer's limited area and thermal budgets impose additional constraints. All these design and fabrication issues lead to generally very simple PnM execution engines, which are unable to exploit the entire DRAM bandwidth [15, 47, 54].

In contrast, PuM architectures enable computation *within* the memory array. The key benefit of PuM architectures is that *data does not leave the memory array during computation.* As a result, PuM architectures can provide high compute throughput by performing operations in a bulk parallel manner, often at the granularity of memory rows. Prior PuM works [70, 72, 74, 75, 79, 82, 84, 96, 97] propose mechanisms for the execution of bulk bitwise operations (e.g., bitwise `MAJority,AND,OR,NOT`) [72, 74, 78, 80, 82–85, 87, 91, 98] and bulk arithmetic operations [70, 75, 79, 96, 97]. However, these proposals have two important limitations: 1) the execution of some complex operations (e.g., multiplication, division) incurs high latency and energy consumption [75], and 2) other complex operations (e.g., exponentiation, trigonometric functions) are not even supported.

We aim to overcome these two limitations of prior PuM architectures in this work. To this end, we employ *LUT-based computing*, i.e., the use of memory read operations *(LUT queries)* to retrieve the results of complex operations from lookup tables that hold precomputed values. Concretely, a *LUT query* is a *memory read operation* that, for a given input value $x$, returns $f(x)$, i.e., the result of applying some function $f$ to the input $x$. Many PuM architectures [96, 97, 99] exploit LUT-based computing to improve the performance of a few complex operations. However, *no* prior work supports the general-purpose execution of LUT-based complex operations.

978-1-6654-6272-3/22/$31.00 ©2022 IEEE
DOI 10.1109/MICRO56248.2022.00067

Our **goal** in this work is to *extend the functionality of DRAM-based PuM systems to provide support for general-purpose execution of complex operations.* **To this end**, we propose pLUTo*: processing-using-memory with lookup table (LUT) operations,* a DRAM-based PuM architecture that leverages LUT-based computing via bulk querying of LUTs to perform complex operations beyond the scope of prior DRAM-based PuM proposals. pLUTo introduces a novel LUT-querying mechanism, the *pLUTo LUT Query*, which enables the simultaneous querying of multiple LUTs stored in a single DRAM subarray. In pLUTo, the number of elements stored in each LUT may be as large as the number of rows in each DRAM subarray (e.g., 512–1024 rows [100–102]). pLUTo requires the following two modest modifications to DRAM hardware: 1) ***row sweeping logic***, which enables the *sweeping* of DRAM rows, i.e., the successive activation of consecutive rows in a DRAM subarray; 2) ***match logic***, which identifies *matches* between the *elements* in the input row and the *index* of the currently active row in the subarray that holds multiple copies of one or more LUTs. We describe three pLUTo designs: pLUTo-BSA *(Buffered Sense Amplifier)*, pLUTo-GSA *(Gated Sense Amplifier)*, and pLUTo-GMC *(Gated Memory Cell)*. These designs achieve different performance, energy efficiency, and area overhead trade-offs.

To enable the seamless integration of pLUTo with the system, we methodically describe the changes that allow programmers to offload their applications to pLUTo. These changes comprise 1) pLUTo ISA instructions that enable support for each of the DRAM operations required for pLUTo's operation, 2) the pLUTo Library, an API library that includes routines that programmers can use to conveniently express pLUTo operations at a high level of abstraction, 3) the pLUTo Compiler, which analyzes an application's data dependency graph to plan the in-memory placement and alignment of data, and 4) the pLUTo Controller, a modified memory controller that supports the execution of pLUTo ISA instructions.

We evaluate pLUTo on a diverse range of real-world arithmetic, bitwise logic, cryptographic, image processing, and neural network workloads that demonstrate the limitations of existing PuM architectures and how pLUTo is able to overcome them. These workloads include 1) bitwise (`AND/OR/XOR`), arithmetic (addition, multiplication), and nonlinear operations (substitution tables, image binarization, and color grading); and 2) a quantized neural network. We compare pLUTo to state-of-the-art processor-centric architectures (CPU [103], GPU [104], FPGA [105]) and PiM architectures (PnM [67], PuM [75, 79, 84]). Our evaluations show that pLUTo consistently and considerably outperforms these five baselines in performance and energy consumption.

We make the following **key contributions**:

- We introduce pLUTo, a new DRAM-based PuM architecture that introduces support for general-purpose operations through the use of bulk lookup table (LUT) queries.
- We propose three different and new pLUTo designs with varying trade-offs in performance, energy efficiency, and DRAM area overhead.
- We describe the end-to-end system integration of pLUTo, including 1) ISA instructions, 2) an API library, 3) a compiler, and 4) modifications to the memory controller.
- We experimentally demonstrate that pLUTo significantly outperforms CPU [103], GPU [104], FPGA [105], and PiM [67, 75, 79, 84] baselines across a wide variety of real-world bitwise logic, arithmetic, cryptographic, image processing, and neural network workloads.
- We open-source pLUTo's source code and all scripts required to reproduce the results presented in this paper on `https://github.com/CMU-SAFARI/pLUTo`.

## 2. Background

This section describes the hierarchical organization of DRAM and provides an overview of relevant prior work we build on. We refer the reader to prior work [79, 84, 100, 106–108] for more detailed descriptions of DRAM operation.

### 2.1. DRAM Background

DRAM is organized hierarchically: each DRAM module consists of multiple chips, banks, and subarrays, as Figure 1 shows. A DRAM *module* (Figure 1a) consists of multiple DRAM *chips* (Figure 1b), each of which contains multiple DRAM *banks* (e.g., 8 for DDR3 [109], 16 for DDR4 [110]) and I/O logic. Each DRAM bank (Figure 1c) is divided into DRAM *subarrays* [100] (Figure 1d), which are two-dimensional arrays of DRAM *cells* (Figure 1e). DRAM subarrays in a bank share peripheral circuitry (e.g., a global row decoder and a global row buffer). Each DRAM cell contains one cell capacitor and one access transistor. The cell capacitor encodes a single bit as stored electrical charge. The access transistor connects the cell capacitor to the *bitline* wire. Each bitline is shared by all DRAM cells in a column and is connected to a *sense amplifier* (*SA* in Figure 1d). The set of sense amplifiers in a subarray makes up the *local row buffer*.

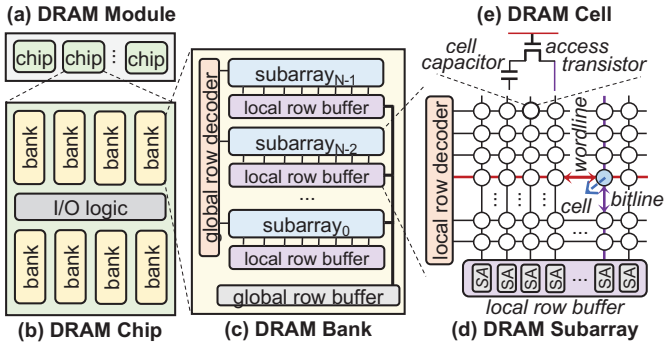


**Figure 1: Internal organization of a DRAM module.**

Reading and writing data in DRAM occurs over three phases. *1) Row Activation.* The memory controller initiates a memory request by issuing an activate (`ACT`) command together with a DRAM row address to the target DRAM bank. Once the DRAM chip receives the `ACT` command, it asserts the corresponding wordline to activate the DRAM row. The row activation process happens in three main steps. First, the wordline of the accessed row is driven with high voltage, turning on the row's access

transistors and creating a path for *charge sharing* between each DRAM cell and its bitline. This process induces a voltage fluctuation, $\pm\delta$, that affects the voltage level of the *precharged* (i.e., set to $V_{DD}/2$) bitline. If the cell is charged, the bitline voltage becomes $V_{DD}/2+\delta$. Otherwise, it becomes $V_{DD}/2-\delta$. Second, each sense amplifier in the local row buffer *amplifies* its bitline's voltage fluctuation ($\pm\delta$) until the bitline voltage reaches either $V_{DD}$ or 0. Third, the original voltage level of each cell capacitor in the activated row is *restored*, since each access transistor allows charge to flow between each bitline and each corresponding DRAM cell in the activated row. Once the row activation process is complete, the local row buffer contains the values originally stored in the cells along the asserted wordline.

*2) Reading/Writing.* The memory controller issues read (`RD`) or write (`WR`) commands together with a DRAM *column address* to read or write chunks of the data latched in the local row buffer. In the case of a read, data in the corresponding columns is sent to the CPU through the DRAM's I/O circuitry and the memory bus. In the case of a write, data received from the CPU modifies the corresponding columns' bitline (and thus cell) voltages.

*3) Precharging.* The memory controller issues a precharge (`PRE`) command to free up the local row buffer and allow the activation of other DRAM rows. To achieve this, the wordline is de-asserted, which turns off the access transistors along a DRAM row, and the subarray's bitlines are precharged (i.e., restored to $V_{DD}/2$).

### 2.2. Enhanced DRAM Architectures

pLUTo optimizes key operations by incorporating the following previous proposals for enhanced DRAM architectures.

**Intra-Subarray Data Copy.** The *RowClone-FPM* (Fast Parallel Mode) [81] operation enables data to be copied between two DRAM rows belonging to the same DRAM subarray. This is achieved via two consecutive activations: first to the source row, then to the destination row. Doing so asserts the destination row's wordline while the contents of the source row are already in the subarray's row buffer, which causes the entire row buffer's contents to be written to the destination row.

**Inter-Subarray Data Copy.** The *LISA-RBM* (Row Buffer Movement) operation [108] copies the contents of one local row buffer to another local row buffer in a different subarray in the same bank without relying on the external memory channel. This is achieved by linking neighboring subarrays with isolation transistors.

**Subarray-Level Parallelism.** *MASA* [100] is a mechanism that enables subarray-level parallelism by overlapping the latency of memory accesses directed to different subarrays' bitlines. MASA enables multiple rows in different subarrays to be activated and to be accessed (read or written to) in parallel.

**Bulk Bitwise Operations.** *Ambit* [84] is a PuM architecture that introduces native support for bulk bitwise logic operations (`MAJority,AND,OR,NOT`) between rows in a DRAM subarray. Ambit uses the *triple row activation* primitive (which concurrently asserts three wordlines, leading to the execution of the majority function between the contents of three DRAM rows) and the copy operation enabled by RowClone [81] to enable these simple, row-granularity bitwise operations.

**Shifting.** *DRISA* [79] is a PuM architecture that features support for intra-row shifting in DRAM. Using this mechanism, the contents of a DRAM row can be shifted by 1 or 8 bits at a time, at the cost of *one* `ACT-ACT-PRE` [84] command sequence.

## 3. Motivation

Our **goal** in this work is to *extend the functionality of Processing-using-Memory (PuM) architectures to provide support for the general-purpose execution of complex operations.* In particular, pLUTo is motivated by the following two key observations. First, state-of-the-art PuM architectures [41, 55, 68–91, 96] provide very high performance and energy efficiency by mitigating data movement, but they *only* support a limited range of operations. For example, prior DRAM-based PuM accelerators only support the execution of basic operations (e.g., bitwise logic, addition) [70, 74, 75, 79, 82, 84, 96, 97] or require long sequences of DRAM commands to support more complex operations (e.g., multiplication, division) [75]. Second, lookup tables (LUTs) enable the replacement of complex computations with cheaper LUT query operations (i.e., memory reads). pLUTo improves prior PuM works by leveraging their best features (i.e., high parallelism, reduced data movement) and addressing their main drawbacks (i.e., reduced range of supported operations and low performance for complex operations). We achieve this via the introduction of the *pLUTo LUT Query* operation (described in Section 4.1), which enables the bulk querying of all the values in a given input DRAM row.

## 4. An Overview of pLUTo

The key contribution of pLUTo is the *pLUTo LUT Query*, an operation that enables the bulk execution of a large number of LUT queries *inside* a DRAM subarray. Each individual *LUT query* is defined as a *memory read operation* that, given an *input value* $x$, returns as its *output value* the result of applying some arbitrary function $f$ to $x$, i.e., $f(x)$. Building on the pLUTo LUT Query, pLUTo employs *LUT-based computing* (i.e., *the replacement of complex operations with equivalent LUT queries*) to perform computation under the Processing-using-Memory paradigm. LUT-based computing requires that each complex operation to be replaced with a LUT query be *deterministic*; in other words, the behavior of the function $f$ being replaced with a LUT query *should only depend on its input value x.* The construction of a LUT requires a one-time effort of computing all its values, i.e., all *LUT elements*.

Figure 2 shows an overview of the DRAM structures required to perform a *pLUTo LUT Query*.[1] First, the *source subarray* (❶ in Figure 2) stores the *LUT query input vector* (ⓘ in Figure 2), which consists of a set of $N$-bit LUT indices associated with LUT elements. Second, the *pLUTo Match Logic* (❷) comprises a set of comparators that identify matches between

[1] Figure 2 assumes that pLUTo has been implemented with the pLUTo-BSA design (described in Section 5.1). However, the key ideas of the pLUTo LUT Query described in this section apply to all three pLUTo designs described in Sections 5.1 to 5.3.

1) the row index of the currently activated row in the pLUTo-enabled subarray, and 2) each LUT index in the LUT query input vector (i.e., the source subarray's row buffer). Third, the *pLUTo-enabled row decoder* (❸) enables the successive activation of consecutive DRAM rows in the pLUTo-enabled subarray with a single DRAM command. It also outputs the row index of the currently activated row as input to the pLUTo Match Logic. Fourth, the *pLUTo-enabled subarray* (❹) stores multiple vertical copies of a given LUT (ⓘⓘ), which consists of $M$-bit LUT elements. Fifth, the *pLUTo-enabled row buffer* (❺) allows the reading of individual LUT elements from the activated row in the *pLUTo-enabled subarray*. This is possible by extending the DRAM sense amplifier design of the pLUTo-enabled row buffer with switches controlled by the pLUTo Match Logic (using the *matchline* signal). Sixth, the *flip-flip (FF) buffer* (❻) enables pLUTo to temporarily store select LUT elements by copying them from the pLUTo-enabled row buffer, conditioned on the output of the pLUTo Match Logic following each row activation. Seventh, a LISA-RBM operation copies the entire contents of the FF buffer (i.e., the LUT query output vector, ⓘⓘⓘ) into the destination row buffer, i.e., the row buffer of the *destination subarray* (❼).

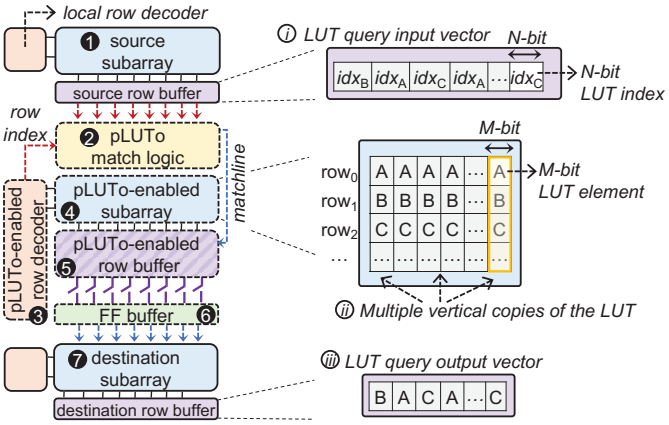


**Figure 2: Main components of pLUTo.**

In contrast to the bit-serial paradigm employed by prior PuM architectures (e.g., SIMDRAM [75]), pLUTo operates in a *bit-parallel* manner; in other words, the bits that make up each LUT element (e.g., `A`) are stored *horizontally* (i.e., in adjacent bitlines), and all the copies of each LUT element (i.e., `{A,A,...,A}`) take up *one whole row* in the depicted pLUTo-enabled subarray (ⓘⓘ).[2]

## 4.1. The pLUTo LUT Query

The pLUTo LUT Query enables all the elements stored in a source row buffer to simultaneously be used to query a LUT. We illustrate the pLUTo LUT Query using a simple example that employs a small LUT to store the first four prime numbers (i.e., `{2,3,5,7}`) at LUT indices `{0,1,2,3}`, as shown in Figure 3a.

[2] Throughout this example, we assume that all `{A,B,C,...}` values are 8 bits wide, although this bit width is a parameter of each pLUTo LUT Query and may vary (see Section 6.2).

In our example, the user-defined LUT query will return the $\{2^{nd}, 1^{st}, 2^{nd}, 4^{th}\}$ prime numbers, which corresponds to a LUT query input vector `[1,0,1,3]` and an expected LUT query output vector `[3,2,3,7]` (i.e., $\{2^{nd}, 1^{st}, 2^{nd}, 4^{th}\}$ prime numbers $\Rightarrow$ `{LUT[1],LUT[0],LUT[1],LUT[3]} = {3,2,3,7}`). Note that, in this example, *four* individual lookup operations are performed by a *single* pLUTo LUT Query.

Four copies of this LUT are stored in a pLUTo-enabled subarray, as shown in Figure 3b: each row $i$ contains repeated copies of the element corresponding to the entry at the $i$-th index of the LUT. pLUTo performs the pLUTo LUT Query operation in five steps. First, the memory controller loads the LUT query input vector from the source subarray (not shown) into the source row buffer (❶ in Figure 3). Second, the memory controller issues a *pLUTo Row Sweep* operation (Section 5.1.1) to consecutively activate all four rows in the pLUTo-enabled subarray that hold LUT elements, in order (i.e., the row indices to be activated are `{#0,#1,#2,#3}`). After each row activation during a pLUTo Row Sweep operation, the pLUTo Match Logic identifies matches between 1) the row index of the currently activated row in the pLUTo-enabled subarray, and 2) each element of the LUT query input vector (i.e., the source row buffer). The aim of this procedure is to *allow* for consecutive LUT elements, in turn, to be copied to the FF buffer, *if they are part of the final output row* for the ongoing pLUTo LUT Query operation. Consider the activation of row `#0` (❷), which creates four copies of `LUT[0]` (i.e., the LUT element with the value 2) in the pLUTo-enabled row buffer (❸). Concurrently with this row activation, the pLUTo Match Logic 1) identifies a match between the index of the currently activated row (`#0`) and the *second* LUT index in the LUT query input vector (❹), and 2) asserts the matchlines corresponding to the *second element* in the pLUTo-enabled row buffer. As a result, the switch at the *second LUT element* in the pLUTo-enabled row buffer is closed, enabling the LUT element to be copied to the *second position in the FF buffer* (❺).

Third, the activation of row index `#1` illustrates how multiple LUT indices may be matched at once: in this case, the LUT element of `LUT[1]` is required by *both the first and the third* positions of the LUT query input vector (❻). Therefore, the pLUTo Match Logic asserts the matchlines corresponding to *both the first and third* LUT elements, which copies both LUT elements into the FF buffer (❼). Fourth, the pLUTo LUT Query operation progresses by activating row `#2`, which produces *no* matches with LUT indices stored in the LUT Query input vector (❽). As a result, *no* LUT elements are copied into the FF buffer (❾). Fifth, when activating row `#3`, the pLUTo Match Logic identifies a match between row index `#1` and the *fourth* element of the LUT query input vector (❿), which leads to the copy of the LUT element `LUT[3]` (i.e., 7) into the *fourth* position in the FF buffer (⓫).

At this point, the pLUTo Row Sweep operation has been completed, and the FF buffer holds the results of the pLUTo LUT Query. The contents of the FF buffer are then copied to the *destination row buffer* (not shown in Figure 3) using a LISA-RBM command (see Section 2.2).

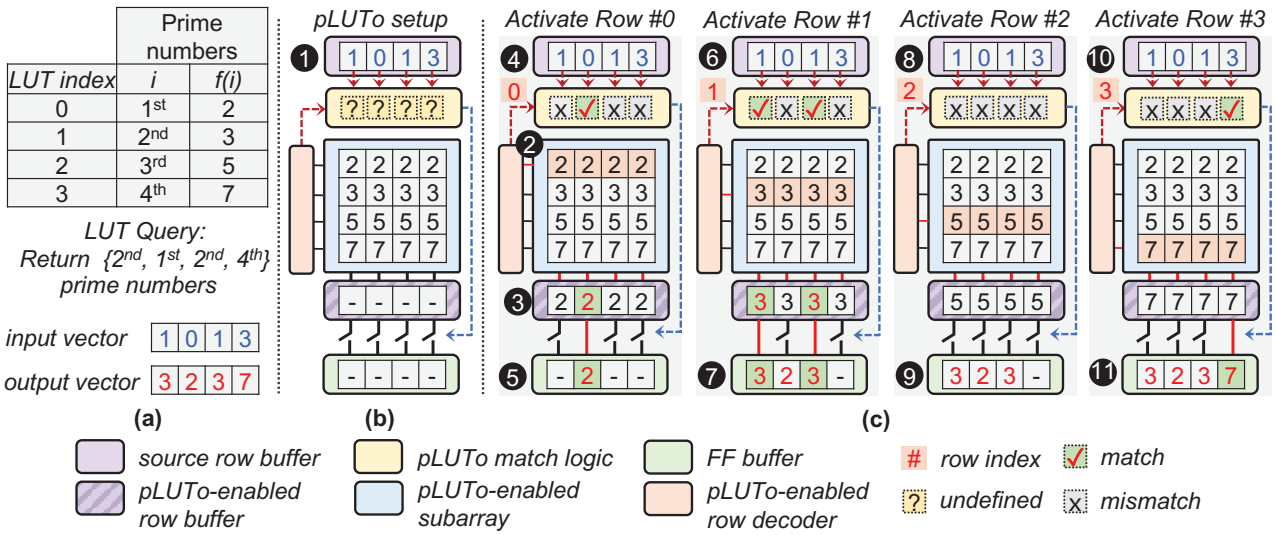


**Figure 3: A pLUTo LUT Query: (a) a LUT containing the first four prime numbers and an example user-specified LUT query, (b) setup of pLUTo's main components prior to the execution of the pLUTo LUT Query, and (c) steps of the pLUTo LUT Query. This pLUTo LUT Query returns into the destination row buffer (not depicted) the i-th prime number for each LUT index in the source row buffer.**

## 5. pLUTo's Hardware Design

This section describes the hardware design of pLUTo that enables the pLUTo LUT Query operation. First, we propose three different pLUTo architectures. Each of these architectures provides a different trade-off between performance, energy efficiency, and area overhead: 1) pLUTo-BSA *(Buffered Sense Amplifier)* incurs moderate hardware overhead and provides intermediate performance and energy efficiency gains; 2) pLUTo-GSA *(Gated Sense Amplifier)* incurs the lowest hardware overhead but provides the lowest performance and energy efficiency; 3) pLUTo-GMC *(Gated Memory Cell)* incurs the highest hardware overhead but provides the highest performance and energy efficiency. Second, we describe the synergistic integration of the novel components of pLUTo with prior PuM-based operations [79, 84, 108] and subarray-level parallelism [100].

### 5.1. pLUTo-BSA *(Buffered Sense Amplifier)* Design

To enhance a DRAM subarray with support for the execution of pLUTo LUT Queries, we modify the DRAM subarray's row decoder and local row buffer to implement the *pLUTo Match Logic*. Uniquely, pLUTo-BSA employs the *key idea* of relying on a secondary row buffer (the *FF buffer*, see Section 5.1.3) to store matching LUT elements during a pLUTo Row Sweep.

**5.1.1. pLUTo-Enabled Row Decoder.** The pLUTo-enabled row decoder enhances the DRAM row decoder by introducing support for the *pLUTo Row Sweep* operation. The pLUTo Row Sweep extends the self-refresh operation (already present in commodity DRAM [111–113]) to activate consecutive rows quickly. With support for the pLUTo Row Sweep operation, pLUTo activates all the rows in the pLUTo-enabled subarray that store LUT elements during a pLUTo LUT Query operation *via a single new DRAM command.* The latency of the pLUTo Row Sweep is equal to $(\texttt{tRCD} + \texttt{tRP}) \times LUT_{\#Elems}$, where tRCD (≈ 12.5 ns in DDR4 [110]) is the time that must elapse to ensure that the sense amplifiers can reliably amplify the voltage perturbation on the bitline, tRP (≈ 12.5 ns in DDR4 [110]) is the time that must elapse between a PRE command and the next ACT command, and $LUT_{\#Elems}$ is the total number of rows swept.

**5.1.2. pLUTo Match Logic.** As shown in Figure 2a, we implement the *pLUTo Match Logic* between the source subarray and the pLUTo-enabled subarray. This logic comprises a set of *comparators*; there are as many comparators in the pLUTo Match Logic as there are *elements* in the source row buffer. Every *i*-th comparator in the pLUTo Match Logic receives as input the following two *N*-bit values, where *N* is *the bit width of each LUT element:* 1) the row index of the currently activated row in the pLUTo-enabled subarray, and 2) the *i*-th element in the source subarray's row buffer. Each comparator outputs an *N*-bit value (the *matchlines* in pLUTo's design) that depends on the result of the comparison between its two *N*-bit inputs: if the two inputs exactly match, all *N* matchlines at the output are driven high; otherwise, all matchlines are driven low.

**5.1.3. pLUTo-Enabled Row Buffer.** Performing the pLUTo LUT Query operation as described in Section 4.1 requires a mechanism to perform many fine-grained (i.e., LUT-element-wise) operations throughout a pLUTo Row Sweep, to both 1) read data from the row buffer of the pLUTo-enabled subarray, and 2) write the result of the pLUTo LUT Query operation to some output buffer. To realize this functionality, which commodity DRAM does *not* support, we connect one flip-flop (FF) to every sense amplifier in the pLUTo-enabled row buffer using a *matchline-controlled switch* (*m-c switch,* shown in Figure 4a). Each m-c switch is closed only if there is a match between the row index of the currently activated row in the pLUTo-enabled subarray and the corresponding LUT index in the LUT Query input vector. The complete row of FFs constitutes an *FF buffer*, which gives pLUTo-BSA *(Buffered Sense Amplifier)* its name. In this design, when a sense amplifier reads a DRAM cell's value, this value is also immediately written into the corresponding FF, but only if the corresponding *matchline* signal is high (i.e., if the m-c switch connected to the FF is closed).

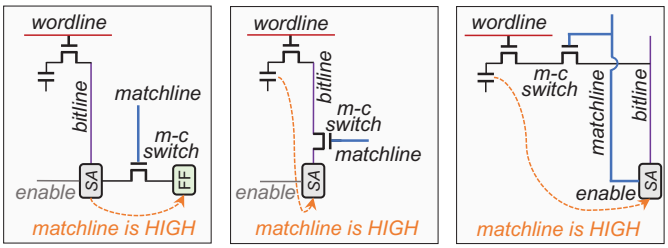


**Figure 4: The three pLUTo designs. *m-c switch* stands for matchline-controlled switch. Orange-dashed lines show how charge flows in case the matchline signal is asserted.**

**5.1.4. Analysis of pLUTo-BSA.** pLUTo-BSA's design entails the following pLUTo LUT Query throughput, pLUTo LUT Query energy consumption, and area overhead values:

- *Throughput:* pLUTo-BSA's maximum pLUTo LUT Query throughput (in *number of LUT queries per second*, LUTs/s) for a single pLUTo-enabled subarray depends on the number of LUT indices that fit in the source subarray ($\frac{RowSize_{bytes}}{LUT_{ElementSize}}$) and the latency of a pLUTo Row Sweep ($(\texttt{tRCD}+\texttt{tRP}) \times LUT_{\#Elems}$). Thus, pLUTo-BSA's maximum throughput is
$$BSA_{Throughput} = \frac{RowSize_{bits}/input_{bit\ width}}{(\texttt{tRCD}+\texttt{tRP}) \times LUT_{\#Elems}}\ LUTs/s.$$
- *Energy Consumption:* The energy pLUTo-BSA consumes during a pLUTo Row Sweep depends on the energy consumed by each DRAM row activation/precharge and $LUT_{\#Elems}$, the total number of rows swept in the pLUTo-enabled subarray:
$$BSA_{Energy} = (\texttt{E}_{\texttt{RCD}}+\texttt{E}_{\texttt{RP}}) \times LUT_{\#Elems}.$$
- *Area Overhead:* The area overhead of pLUTo-BSA includes the area of the pLUTo Match Logic ($RowSize_{bytes} \times Area_{ByteComp}$), matchline-controlled switches ($RowSize_{bits} \times Area_{m-c\ switch}$), and FF buffer ($RowSize_{bits} \times Area_{FF}$). Thus, pLUTo-BSA's area overhead is
$$\begin{aligned} BSA_{Area} = {} & RowSize_{bytes} \times Area_{ByteComp} + \\ & RowSize_{bits} \times Area_{m-c\ switch} + \\ & RowSize_{bits} \times Area_{FF}. \end{aligned}$$

## 5.2. pLUTo-GSA (*Gated Sense Amplifier*) Design

*pLUTo-GSA* differs from pLUTo-BSA in its pLUTo-enabled row buffer design and in its implementation of the pLUTo Row Sweep operation. pLUTo-GSA's *key idea* is to use the sense amplifier as a buffer that stores *only* the LUT elements indicated as a match by the pLUTo Match Logic. By doing this, pLUTo-GSA eliminates the need for a secondary buffer (such as the FF buffer in pLUTo-BSA) to store LUT elements during a pLUTo Row Sweep operation. pLUTo-GSA provides reduced area overhead over pLUTo-BSA, at the expense of reduced throughput and energy efficiency.

**5.2.1. pLUTo-GSA Row Buffer.** Each sense amplifier in pLUTo-GSA's row buffer is *gated* from its bitline by a matchline-controlled switch (Figure 4b). This switch (i.e., isolation transistor) incurs a lower area overhead than the FF buffer used by pLUTo-BSA, resulting in pLUTo-GSA's higher area efficiency. During a pLUTo LUT Query, if the matchline signal is high (i.e., the switch is closed), the sense amplifier is electrically connected to its bitline, thus amplifying bitline voltage perturbations normally. However, if the matchline signal is low (i.e., the switch is open), the sense amplifier does *not* respond to the perturbation induced by its corresponding DRAM cell, which leads to the loss of the cell's contents during row activation (i.e., implementing pLUTo-GSA's row buffer leads to destructive reads). This represents a potential for LUT data loss, which means that *a LUT must be loaded into the pLUTo-enabled subarray before every pLUTo LUT Query in pLUTo-GSA*, leading to performance overheads compared to pLUTo-BSA.

**5.2.2. The pLUTo Row Sweep.** The latency of the pLUTo Row Sweep operation is lower in pLUTo-GSA than in pLUTo-BSA. This is because 1) row activations in pLUTo-GSA's pLUTo Row Sweep do *not* require the full-fledged activation process required to read DRAM data, but *only the triggering* of the charge sharing process, and 2) the pLUTo Row Sweep does *not* require a precharge (PRE) command to be issued after every activate (ACT) command, since each unmatched bitline remains in a precharged state (i.e., the voltage in each bitline remains at $V_{DD}/2$) *until* the pLUTo Match Logic registers a match. Instead, a single PRE command may be issued at the end of the pLUTo Row Sweep. However, row activations in pLUTo-GSA's pLUTo Row Sweep lead to LUT data loss for unmatched elements: when an *unmatched* DRAM cell capacitor is discharged, its charge level will *not* be restored since the matchline-controlled switch leaves the path between the bitline and the sense amplifier open.

The total time required to perform a pLUTo Row Sweep in pLUTo-GSA is equal to $\texttt{tRCD} \times N + \texttt{tRP}$, where tRCD ($\approx$ 12.5 ns in DDR4 [110]) is the time that must elapse between the DRAM chip receiving an ACT command and the sense amplifier finishing sensing the voltage perturbation in the bitline, tRP ($\approx$ 12.5 ns in DDR4 [110]) is the precharge time, and $LUT_{\#Elems}$ is the total number of rows swept. This is about half the time a pLUTo Row Sweep requires in pLUTo-BSA.[3] However, due to the destruction of row contents during the pLUTo Row Sweep, the calculation of the average latency of a pLUTo LUT Query in pLUTo-GSA must also factor in the latency of loading data into a pLUTo-enabled subarray before *every* pLUTo Row Sweep ($LISA_{RBM} \times LUT_{\#Elems}$). As a result, while the latency of the pLUTo Row Sweep operation is lower in pLUTo-GSA than in pLUTo-BSA, the latency of the *total* pLUTo LUT Query is higher in pLUTo-GSA than in pLUTo-BSA.

**5.2.3. Analysis of pLUTo-GSA.** pLUTo-GSA's design entails the following pLUTo LUT Query throughput, pLUTo LUT Query energy consumption, and area overhead values:

- *Throughput:* pLUTo-GSA's maximum throughput (in *number of LUT queries per second*, LUTs/s) for a single pLUTo-enabled subarray depends on the number of LUT indices that fit in the source subarray ($\frac{RowSize_{bytes}}{LUT_{ElementSize}}$), the latency of a pLUTo Row Sweep operation ($\texttt{tRCD} \times LUT_{\#Elems} + \texttt{tRP}$),

[3] The ratio between the latencies of the pLUTo Row Sweep in pLUTo-BSA and pLUTo-GSA is given by $\frac{(\texttt{tRCD}+\texttt{tRP})\times N}{\texttt{tRCD}\times N+\texttt{tRP}}$. Under the assumption that $\texttt{tRCD} \approx \texttt{tRP}$, this simplifies to $\frac{2\times N}{1+N}$, which approaches 2 for large values of $N$.

and the latency of loading LUT elements into the pLUTo-enabled subarray, since LUT elements are destroyed after each pLUTo Row Sweep ($LISA_{RBM} \times LUT_{\#Elems}$). Thus, the maximum throughput achievable with pLUTo-GSA is

$$GSA_{Throughput} = \frac{RowSize_{bits}/input_{bit\ width}}{LISA_{RBM} \times LUT_{\#Elems} + (\texttt{tRCD} \times LUT_{\#Elems} + \texttt{tRP})}\ LUTs/s.$$

- *Energy Consumption:* The energy pLUTo-GSA consumes during a pLUTo Row Sweep operation depends on the number of elements in the LUT and the energy consumed by a DRAM row activation and precharge operation:

$$GSA_{Energy} = \texttt{E}_{LISA_{RBM}} \times LUT_{\#Elems} + \texttt{E}_{\texttt{RCD}} \times LUT_{\#Elems} + \texttt{E}_{\texttt{RP}}.$$

- *Area Overhead:* The area overhead of pLUTo-GSA's design includes the area of the pLUTo Match Logic ($RowSize_{bytes} \times Area_{ByteComp}$), and matchline-controlled switches ($RowSize_{bits} \times Area_{m-c\ switch}$). Thus, the total area overhead of pLUTo-GSA is

$$GSA_{Area} = RowSize_{bytes} \times Area_{ByteComp} + RowSize_{bits} \times Area_{m-c\ switch}.$$

## 5.3. pLUTo-GMC *(Gated Memory Cell)* Design

*pLUTo-GMC* provides higher throughput and energy efficiency over pLUTo-BSA, at the expense of increased area overhead. pLUTo-GMC differs from pLUTo-BSA in its DRAM cell design, pLUTo-enabled row buffer design, and pLUTo Row Sweep implementation. Similarly to pLUTo-GSA, *pLUTo-GMC*'s *key idea* is to use the sense amplifier in the pLUTo-enabled subarray as a buffer to store the matched LUT elements during a pLUTo LUT Query (instead of adding a new buffer, as pLUTo-BSA does). However, in contrast to pLUTo-GSA, row activations during the pLUTo Row Sweep in pLUTo-GMC are *not* destructive, since charge is allowed to flow from the DRAM cell to the bitline *only if* there is a match between the row index and the LUT indices in the source subarray. To do so, pLUTo-GMC adds an extra transistor in *each DRAM cell* of the pLUTo-enabled subarray. As such, pLUTo-GMC is the most intrusive to the subarray design (because it changes the DRAM cell itself).

**5.3.1. pLUTo-GMC DRAM Cell.** pLUTo-GMC implements a 2T1C DRAM cell instead of the conventional 1T1C design (described in Section 2.1). An additional transistor connects the DRAM cell's access transistor to the bitline. The output of the pLUTo Match Logic (i.e., the matchline) controls the additional transistor in each cell, as shown in Figure 4c. The matchline signal thus controls whether a cell in an activated row shares charge with the bitline. This significantly reduces the overall movement of charge, since charge only flows between the DRAM cell and the bitline if the pLUTo Match Logic outputs a match during the pLUTo LUT Query operation, which reduces the overall energy consumption of pLUTo-GMC during a pLUTo Row Sweep compared to both pLUTo-BSA and pLUTo-GSA.

**5.3.2. pLUTo-GMC Row Buffer.** In pLUTo-GMC, additional matchline-controlled switches exist between each sense amplifier and its enable signal. The role of these switches is to ensure that, when a row is activated, the sense amplifier connected to a given DRAM cell only senses the bitline voltage if *both the wordline and the matchline signals are high*. Without this safeguard, the sense amplifiers would be activated when cells in the active row are not connected to the bitline (i.e., when the wordline signal is high for a given row, but the matchline signals are low for one or more cells in that row), which would lead to undefined behavior. In addition, these matchline-controlled switches enable pLUTo-GMC to perform back-to-back activations *without needing to precharge the subarray.* This happens because, when a matchline signal is low, the corresponding bitline behaves as if it had remained inactive, which keeps it in its precharged state. Conversely, *only when the matchline is high* does the sense amplifier become enabled.

**5.3.3. The pLUTo Row Sweep.** pLUTo-GMC optimizes the pLUTo Row Sweep operation by introducing the ability to perform back-to-back activations *without* the need to precharge the bitlines. Leveraging this optimization, pLUTo-GMC outperforms pLUTo-BSA in the pLUTo Row Sweep by almost 2×.[4] To achieve this optimization, pLUTo-GMC adopts the following two key design features. First, a sense amplifier is only enabled when there is a match in the corresponding pLUTo Match Logic. This means that an activation *only* perturbs a bitline if the associated matchline signal is high, and that the voltage in the bitlines is *kept* at $V_{DD}/2$ (i.e., in the precharged state) if the matchline signal is low. Second, since each source row element necessarily only has one match in a LUT, the sense amplifier is only enabled for a single row activation during an entire pLUTo LUT Query. Therefore, we can guarantee that back-to-back row activations will *not* open the gating transistors of any two cells sharing the same bitline, and thus will not destroy the data in the cell. As in pLUTo-GSA, the total time required to perform a pLUTo Row Sweep in pLUTo-GMC is $\texttt{tRCD} \times LUT_{\#Elems} + \texttt{tRP}$. In addition, due to matchline-controlled switches (Section 5.1.3), pLUTo-GMC does *not* destroy the data in the LUTs; this translates into significant performance gains, as there is no need to repeatedly load LUT data into the subarray.

**5.3.4. Analysis of pLUTo-GMC.** pLUTo-GMC's design entails the following pLUTo LUT Query throughput, pLUTo LUT Query energy consumption, and area overhead values:

- *Throughput:* pLUTo-GMC's maximum throughput (in *number of LUT queries per second*, LUTs/s) for a single pLUTo-enabled subarray depends on the number of LUT indices that fit in the source subarray ($RowSize_{bits}/input_{bit\ width}$) and the latency of a pLUTo Row Sweep operation ($\texttt{tRCD} + \texttt{tRP} \times LUT_{\#Elems}$). Thus, the maximum throughput achievable with pLUTo-GMC is

$$GMC_{Throughput} = \frac{RowSize_{bits}/input_{bit\ width}}{\texttt{tRCD} \times LUT_{\#Elems} + \texttt{tRP}}\ LUTs/s.$$

- *Energy Consumption*: The energy pLUTo-GMC consumes during a pLUTo LUT Query operation depends on the number of elements in the LUT and the energy consumed by a DRAM row activation and precharge operation:

$$GMC_{Energy} = \texttt{E}_{\texttt{RCD}} \times LUT_{\#Elems} + \texttt{E}_{\texttt{RP}}.$$

[4] See Footnote 3. The latency of the pLUTo Row Sweep in pLUTo-GSA and pLUTo-GMC is the same.

- *Area Overhead:* The area overhead of pLUTo-GMC's design includes the area of the pLUTo Match Logic ($RowSize_{bytes} \times Area_{ByteComp}$) and matchline-controlled switches ($\#Rows \times RowSize_{bits} \times Area_{m-c\ switch}$). Thus, the total area overhead of pLUTo-GMC is

$$GMC_{Area} = RowSize_{bytes} \times Area_{ByteComp} + \#Rows \times RowSize_{bits} \times Area_{m-c\ switch}.$$

## 5.4. Summary of pLUTo Architectures

While the pLUTo-BSA design provides a balanced trade-off between performance, energy efficiency, and area overhead, the system designer could prefer to optimize for one of these three metrics in isolation. To provide this added flexibility, we described pLUTo-GSA and pLUTo-GMC, two additional pLUTo designs with different trade-offs in performance, energy efficiency, and area overhead. Table 1 summarizes the trade-offs of each of the proposed pLUTo designs.

**Table 1: Comparison of pLUTo designs' core attributes. Bold cells represent key benefits of a pLUTo design compared to others.** $N$ **corresponds to LUT elements (i.e.,** $LUT_{\#Elems}$**).**

| | pLUTo-BSA | pLUTo-GSA | pLUTo-GMC |
|---|---|---|---|
| **Area Efficiency** | Medium | **High** | Low |
| **Throughput** | Medium | Low | **High** |
| **Energy Efficiency** | Medium | Low | **High** |
| **Destructive Reads** | **No** | Yes | **No** |
| **LUT Data Loading** | **Once** | After every use | **Once** |
| **Query Latency** | $(\mathtt{tRCD} + \mathtt{tRP}) \times N$ | $LISA_{RBM} \times N + \mathtt{tRCD} \times N + \mathtt{tRP}$ | $\mathtt{tRCD} \times N + \mathtt{tRP}$ |
| **Query Energy** | $(\mathtt{E_{RCD}} + \mathtt{E_{RP}}) \times N$ | $\mathtt{E}_{LISA_{RBM}} \times N + \mathtt{E_{RCD}} \times N + \mathtt{E_{RP}}$ | $\mathtt{E_{RCD}} \times N + \mathtt{E_{RP}}$ |

Table 1 and the expressions for throughput, energy consumption and area overhead derived in Sections 5.1.4, 5.2.3 and 5.3.4 enable three key observations. First, pLUTo-GMC provides the highest throughput of the three designs ($GMC_{Throughput} > BSA_{Throughput} > GSA_{Throughput}$). pLUTo-GMC achieves this by 1) eliminating the need to issue `PRE` commands following every row activation in the pLUTo-enabled subarray (as in pLUTo-BSA), and 2) eliminating the need to load all LUT elements before each pLUTo LUT Query operation (as required by pLUTo-GSA, due to the destructive row activations required by its pLUTo Row Sweep operation). Second, GMC provides the highest energy efficiency of the three designs ($GMC_{Energy} < BSA_{Energy} < GSA_{Energy}$). pLUTo-GMC achieves this by 1) eliminating the energy overhead associated with the issuance of `PRE` commands following every row activation in the pLUTo-enabled subarray (as in pLUTo-BSA), and 2) eliminating the energy overhead associated with loading the LUTs before each pLUTo LUT Query (as in pLUTo-GSA). Third, GSA incurs the smallest area overhead of the three designs ($GSA_{Area} < BSA_{Area} < GMC_{Area}$). pLUTo-GSA's area overhead is minimized by making as few modifications to the DRAM array as possible. In particular, pLUTo-GSA does *not* employ the logic components required for the operation of pLUTo-BSA (i.e., the FF buffer) or pLUTo-GMC (i.e., per-cell matchline-controlled switches).

We conclude that pLUTo-GSA is the most well-suited design to minimize area overhead, pLUTo-GMC is the most well-suited design to maximize either performance or energy efficiency, and pLUTo-BSA provides a trade-off point that offers intermediate throughput, energy efficiency, and area overhead metrics.

## 5.5. Subarray-Level Parallelism

Since the lookup operations of different input values are independent of one another, many pLUTo LUT Queries can be executed simultaneously across *multiple* subarrays by exploiting *subarray-level parallelism* (SALP) [100], as described in Section 2.2. Two important use cases benefit from the distribution of LUT queries across multiple subarrays: 1) LUT query input vectors with a very large number of LUT indices can be partitioned across multiple source subarrays to be queried simultaneously; and 2) independent pLUTo LUT Queries (possibly belonging to different threads or applications) can be executed concurrently.

The achievable degree of subarray-level parallelism is limited by the `tFAW` DRAM timing constant [109, 110], which corresponds to the duration of the time window during which *at most four* `ACT` commands can be issued, per DRAM rank. This constraint protects against the deterioration of the DRAM reference voltage, although DRAM manufacturers have been able to mitigate it substantially in commodity DRAM chips in recent years [114], as well as to perform a targeted reduction of this parameter specifically for PiM architectures where it becomes a performance bottleneck [115]. These advances suggest that this parameter may not limit pLUTo's scalability severely.

## 5.6. Limitations of pLUTo's Subarray Design

For a single-subarray pLUTo LUT Query, the number of LUT elements can scale up to *the number of rows in the subarray.* To query LUTs with a greater number of elements, it is possible to partition a pLUTo LUT Query across subarrays. Note that partitioning the query *does not increase* latency (since multiple subarrays operate simultaneously), but *does increase* energy consumption $N$-fold, for a pLUTo LUT Query distributed across $N$ subarrays. For this reason, the design of pLUTo is not well suited for executing large-bit-width lookup queries. We leave the potential exploration of alternative designs that address this limitation for future work.

## 5.7. The Role of pLUTo in the PiM Landscape

As discussed in Section 1, PnM and PuM are complementary approaches: the former enables flexible substrates that support a diverse range of operations, while the latter yields maximal performance and energy efficiency benefits. pLUTo does *not* aim to replace prior PuM proposals. Instead, it addresses an important gap in the literature and enables PuM to support more complex operations, as many applications require. Ideally, a real-world PiM system would combine the strengths of different proposals: for example, relying on SIMDRAM [75] for addition, pLUTo for trigonometric functions and bit counting operations, and near-memory general-purpose cores [67] for serial reduction and other irregular tasks. Mapping application segments to their most suitable PiM substrates is a rich area for future work.

## 6. System Integration

This section describes the system integration stack that enables pLUTo to operate seamlessly with the host system. There are four key components in this stack: 1) the *pLUTo ISA* (Section 6.1), a set of instructions that express i) pLUTo Row Sweep operations (`pluto_op`), ii) bitwise logic operations [84], iii) bit- and byte-level shifting operations [79], and iv) data movement operations involving multiple DRAM rows [108]; 2) the *pLUTo library* (Section 6.2), a set of routines that implement complex operations (i.e., operations that involve several pLUTo ISA instructions); 3) the *pLUTo Compiler* (Section 6.3), which analyzes data dependences (necessary for data allocation and alignment) and translates pLUTo library routines to pLUTo ISA instructions; 4) the *pLUTo Controller* (Section 6.4), a modified DRAM controller that supports the execution of pLUTo ISA instructions (i.e., given a `pluto_op`, the pLUTo Controller carries out the corresponding pLUTo Row Sweep operation through a series of `ACT` and `PRE` DRAM commands). The description of the entire system integration stack is depicted by Figure 5, which shows an end-to-end example of pLUTo's operation, from reference C code (a) to in-memory computation (e). This example describes the implementation of the multiply-and-add ($A \odot B + C$) operation between three vectors: `Row A` (2-bit elements), `Row B` (2-bit elements), and `Row C` (4-bit elements). Section 6.5 describes the methods of creating the LUTs used in pLUTo LUT Queries. Finally, Section 6.6 discusses the limitations of our proposed system integration of pLUTo and how we can mitigate these limitations.

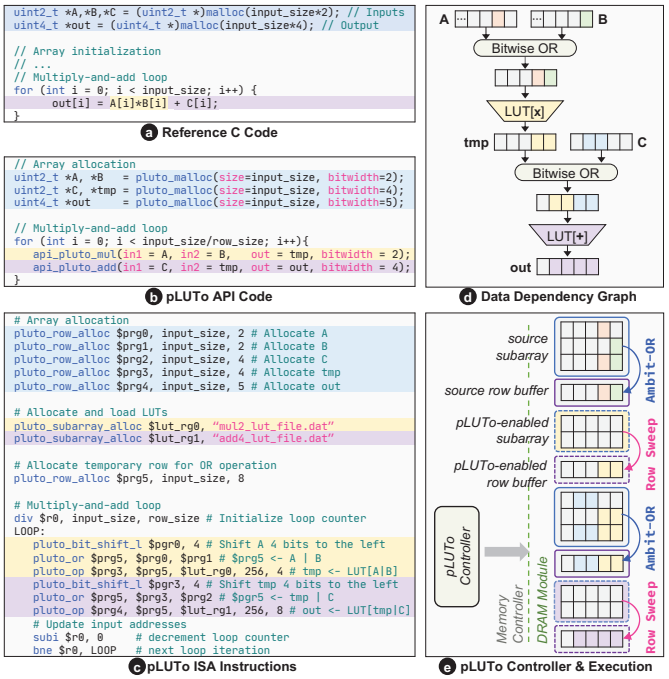


**Figure 5: pLUTo's system integration stack. An example is shown for the C code displayed in a. Subsequent steps are shown in top-down, left-to-right order: b implementation using pLUTo's API Library, c the transformation of the pLUTo API code performed by the pLUTo Compiler, d data dependency graph analysis, e the role of the pLUTo Controller and in-memory execution.**

### 6.1. The pLUTo ISA

We propose ISA extension instructions that express the operations required by pLUTo to perform in-memory computation. The instructions in the pLUTo ISA manipulate special-purpose *pLUTo registers* that keep track of the currently allocated pLUTo data structures. We describe ISA instructions for 1) allocating memory, 2) querying LUTs (the `pluto_op`), and 3) manipulating data (in-memory bitwise logic [84], bit shifting [79], and data copy [108] operations), as summarized in Table 2. Figure 5 c shows the translation of instructions from the reference program (a) into a sequence of pLUTo ISA instructions.

**Table 2: Summary of pLUTo ISA extension instructions.**

| Operation | Instruction | Proposed in |
|---|---|---|
| pLUTo Register Allocation | `pluto_row_alloc dst, size, bitwidth`<br>`pluto_subarray_alloc dst, num_rows, lut_file` | This work<br>This work |
| pLUTo Row Sweep | `pluto_op dst, src, lut_subarr, lut_size, lut_bitw` | This work |
| Bitwise Logic Operations | `pluto_{not, and, or} dst, src1, scr2` | [84] |
| Bit- and Byte-Level Shifting | `pluto_bit_{shift_l, shift_r} src, #N`<br>`pluto_byte_{shift_l, shift_r} src, #N` | [79]<br>[79] |
| In-DRAM Data Movement | `pluto_move dst, src` | [108] |

**pLUTo Registers.** pLUTo's instructions operate at the granularity of *contiguously allocated DRAM rows* (for both LUT query input and output vectors) and *contiguously allocated DRAM subarrays* (pLUTo-enabled subarrays that hold LUTs). To guarantee that the physical memory addresses of all DRAM rows involved in a pLUTo LUT Query operation are contiguously allocated in the DRAM array, we define two data structures (*row registers* and *subarray registers*) that separately capture these two abstractions. Each *pLUTo Row Register* is a special-purpose architectural register that identifies a DRAM *row* to be used either as the input or the output of a pLUTo LUT Query. Each *pLUTo Subarray Register* is a special-purpose architectural register that identifies a LUT-holding DRAM *subarray* to be used in a pLUTo LUT Query. These two types of registers are used as arguments of pLUTo's ISA instructions where appropriate. The allocation of pLUTo Registers is performed by the operating system via a call to a pLUTo allocation routine (see "Memory Allocation", Section 6.1). The allocation of both register types is recorded in an in-memory allocation table, which the pLUTo Controller (Section 6.4) accesses to derive the *physical memory addresses* required to issue DRAM commands during the execution of pLUTo LUT Queries.

**Memory Allocation.** We introduce two instructions to enable the compiler or the programmer to allocate pLUTo Registers. The first instruction, `pluto_row_alloc`, allocates the memory space (as a *whole number* of memory rows) to be used by a source or destination row involved in the execution of pLUTo ISA instructions. This instruction has two inputs (`size` and `bitwidth`) and one output (`dst`). It sets `dst` to a valid pLUTo Row Register that is used to reference the allocated `size`-byte memory row(s) whose elements are `bitwidth`-bits wide. `bitwidth` is only a meaningful parameter for data structures used as *inputs*, and is equal to $\log_2(\texttt{lut\_size})$, where `lut_size` is number of elements in the LUT to query.

The second instruction, `pluto_subarray_alloc`, allocates memory space corresponding to consecutive rows belonging to a single subarray, in which the LUT required by a `pluto_op` will be stored. This instruction has two inputs (`num_rows` and `lut_file`) and one output (`dst`). It sets `dst` to a valid pointer that references the allocated DRAM subarray. `num_rows` is the number of rows to be reserved, i.e., the number of elements in the associated LUT, and `lut_file` is a memory location that holds the LUT data to be stored in the allocated subarray.

**LUT Querying.** A pLUTo LUT Query (Section 4.1) uniquely maps to a `pluto_op` instruction, which has three inputs (`src`, `lut_subarr`, `lut_size`, and `lut_bitw`) and one output (`dst`). Here, `dst` and `src` are the pLUTo Row Registers of the destination and source rows. `lut_subarr` is the physical address of the pLUTo-enabled subarray where the LUT to query is stored. `lut_size` is the number of LUT elements, i.e., the number of rows to sweep. `lut_size` must be a power of two: more specifically, `lut_size` $:= 2^N$, where $N$ is the bit width of each source row value; for example, a 4-bit-input LUT contains $2^4 = 16$ elements, and thus requires the sweeping of 16 rows. `lut_bitw` specifies *the bit width of the LUT elements*,[5] i.e., the *width* of the match logic's comparators for this `pluto_op`. A `pluto_op` instruction always operates at the granularity of a DRAM row; as a result, operating on $S$ input bytes requires $\lceil \frac{S}{DRAM_{row\ size}} \rceil$ `pluto_op` instructions.

**Bit Manipulation.** pLUTo requires bit manipulation operations proposed by prior works [79, 84, 108], as shown in Table 2. We use these operations to align source row values (bit shifting using `pluto_bit_*` and `pluto_byte_*`), merge operands between source rows (bitwise `OR` using `pluto_or`), apply bit masks to input and output rows (bitwise `AND` using `pluto_and`), and copy rows in-memory (row buffer to row buffer data copy using `pluto_move`).

## 6.2. The pLUTo Library

The pLUTo library encompasses 1) *computation* routines, which the programmer may conveniently use to express operations at a high level of abstraction, and 2) a routine for *memory allocation* (`pluto_alloc`), which the programmer may use to instantiate the data structures involved in pLUTo's operation (i.e., the source and destination rows and the LUT-holding subarrays).

**Computation.** Examples of pLUTo Library computation routines include common operations (e.g., `api_pluto_add` and `api_pluto_mul` express addition and multiplication). Figure 5 ⓑ contains a code example with pLUTo library calls (`api_pluto_add`, `api_pluto_mul`) in place of the addition and multiplication operations in the reference code (Figure 5 ⓐ). Each of the routines in the pLUTo library translates into a predetermined, constant sequence of pLUTo ISA instructions. For example, the 4-bit addition operation

```
api_pluto_add(in1,in2,out,bitwidth=4)
```

*always* corresponds to the following sequence of pLUTo ISA instructions:

```
pluto_or temp,in1,in2
pluto_op dst,temp,add4_lut,lut_size=256,lut_bitw=8
```

Here, `dst` denotes the destination row to which the result will be stored, `temp` holds the result of the bitwise `OR` operation that combines the two source rows, `add4_lut` denotes the subarray that holds the LUT with the results for the 4-bit addition, and `lut_size` and `lut_bitw` are uniquely determined by the bit width of this operation (256 and 8, respectively, in this example). pLUTo library routines always assume a specific data alignment (e.g., the `pluto_add` operation assumes that the left and right operands are concatenated before performing the LUT query). However, these routines *do not explicitly guarantee this alignment;* instead, the responsibility of ensuring correct input operand alignment is assumed by the pLUTo Compiler, as explained in Section 6.3.

**Memory Allocation.** To abstract the low-level memory allocation instructions defined in Section 6.1, the pLUTo Library implements the `pluto_malloc` routine, defined as `pluto_malloc(size,bitwidth)`. Here, `size` is the number of bits to be allocated, and `bitwidth` is the bit width of each element (either an input/output value, or a LUT element). Based on the dependences between the arguments of this function, the pLUTo Compiler (Section 6.3) is able to infer a sequence of pLUTo ISA instructions (i.e., `pluto_row_alloc`, `pluto_subarray_alloc`) that are equivalent to it. ⓑ and ⓒ in Figure 5 show an example of this compilation process.

## 6.3. The pLUTo Compiler

The role of the pLUTo Compiler is to identify the dependences between operands used by pLUTo library routines to ensure the correct allocation (i.e., `pluto_alloc_*`) and alignment of these operands. The compiler may achieve operand alignment by inserting additional pLUTo ISA instructions to perform bit shifting (`pluto_{bit,byte}_*`), bit masking (`pluto_and`), and row merging (`pluto_or`) operations as needed, *in addition to the pLUTo ISA instructions specified by each pLUTo Library routine*. As an example of the role of the compiler, consider the multiplication between the 2-bit elements of arrays `A` and `B` using the `api_pluto_mul` instruction, as shown in Figure 5 ⓒ. The translation from the `api_pluto_mul` call to pLUTo ISA instructions yields only a `pluto_or` and a `pluto_op`, and therefore does *not* guarantee that each value from `A` is combined with its counterpart from `B` to create the value to be queried in the LUT. In this example, the compiler guarantees correct operand alignment by performing the operations shown in ⓓ, namely 1) shifting the contents of input row `A` to the left by two bits, and 2) merging the result of operation 1) with input row `B` using a bitwise `OR`.

## 6.4. The pLUTo Controller

The pLUTo Controller, which extends the DRAM controller, executes the pLUTo ISA instructions that are either 1) specified by the programmer using pLUTo Library routines, or 2) inserted by the pLUTo Compiler to ensure correct operand alignment. Each

[5] `lut_bitw` can only be greater than or equal to $N$. If `lut_bitw` $> N$, the source row values will be zero-padded: as an example, for $N = 1$ and `lut_bitw` = 8, the 1-bit values `{0,1}` would be zero-padded to a width of 8 bits (i.e., `{00000000,00000001}`) and used to query a 2-entry LUT whose elements may be any 8-bit value (e.g., `{00000000,11111111}`).

of these ISA instructions translates into either i) a predefined sequence of ACT and PRE commands, which the pLUTo Controller stores in an internal ROM (for the execution of bitwise logic operations, bit- and byte-level shifting, and in-DRAM data movement operations); or ii) a pLUTo Row Sweep command.

The pLUTo Controller's hardware consists of: 1) a small internal ROM that maps each pLUTo ISA instruction to appropriate DRAM commands; 2) a small register file that holds pLUTo Row Registers; and 3) a finite state machine that decodes pLUTo ISA instructions, gathers the physical addresses of all operands of a pLUTo ISA instruction and controls the execution flow of pLUTo's in-memory operations. The hardware and operation of the pLUTo Controller resemble those of SIMDRAM's Control Unit [75], and thus incur negligible area overhead on the host CPU die (< 0.08%).

### 6.5. Loading LUT Data

To load the LUTs required by pLUTo LUT Queries, it is necessary to 1) *allocate LUT subarrays* (using the `pluto_subarray_alloc` operation) that are adjacent or in close physical proximity to the source and destination rows, and 2) *load the LUTs* into these subarrays. The loading of LUTs may take place in one of three ways, which we quantitatively compare in Section 8.5:

**1. First-Time Generation.** The first time a LUT is required, its elements must be computed from scratch. Optionally, these values may then be saved in memory for later reuse. This procedure is similar to operand memoization [116] and presents an opportunity for potential further optimizations, as described in prior works [116–120]. We leave the exploration of more complex memoization strategies for future work.

**2. Loading From Memory.** If a LUT already exists in memory, the most efficient way to reuse it is by copying it to the designated pLUTo-enabled subarray using LISA-RBM [108] (if the source and destination subarrays are in close physical proximity) or a CPU-mediated copy operation.

**3. Loading From Secondary Storage.** If a LUT is stored in secondary storage (e.g., at compile time, or following First-Time Generation), it may be loaded into the main memory at runtime using a direct memory access (DMA) operation.

### 6.6. Limitations of the System Integration Stack

**Address Translation.** Ensuring the physical proximity between the source row, the LUT-holding subarray, and the destination row requires knowledge of the physical address mapping of the involved DRAM subarrays, banks, and ranks. Two possible approaches to achieve this are 1) via the help of a memory controller that can provide this information, and 2) via an *a priori* reverse-engineering effort that allows the memory mapping scheme to be recovered [121].

**Coherence.** pLUTo does *not* provide means to enforce coherence between the data stored in pLUTo subarrays and the data stored in other locations in the system (e.g., CPU caches). For this reason, programmers are responsible for preventing data decoherence stemming from modifications by simultaneous CPU and pLUTo operations (e.g., using instructions to flush cache lines belonging to memory addresses that pLUTo will operate on). pLUTo can leverage coherence optimizations tailored to PiM to improve overall performance [32, 48, 122].

## 7. Methodology

We evaluate the three proposed pLUTo designs: pLUTo-GSA, pLUTo-BSA, and pLUTo-GMC. Unless stated otherwise, our implementations assume the parallel operation of 16 subarrays with 8 kB row buffers for DDR4 memory [100], and 512 subarrays with 256 B row buffers for 3D-stacked (3DS) [65] memory. These two design points are comparable since the volume of data processed *per operation* is identical in both cases: $16 \times 8\ kB = 512 \times 256\ B = 128\ kB$. We compare each pLUTo design against four baselines: 1) a state-of-the-art CPU, 2) a state-of-the-art GPU, 3) a simulated Processing-near-Memory (PnM) accelerator, and 4) a simulated FPGA. Table 3 shows the main parameters we use in our evaluations.

**Table 3: Configuration of the simulated system.**

| Parameter | Configuration |
|---|---|
| Main Memory | DDR4 2400 MHz, 8 GB, 1-channel, 1-rank, 4-bank groups, 4-banks per bank group, 512 rows per subarray, 8 kB per row; timings 17-17-17 (14.16 ns) |
| PnM | HMC Model [67] with support for bitwise operations [84] and bit shifting [79]; on-die core with 1.25 GHz clock, 10 W TDP |
| FPGA | Zynq® UltraScale+ MPSoC ZCU102 [105] |
| pLUTo | 16-subarray parallelism [100] unless stated otherwise; unthrottled rate of row activations (`tFAW` = 0$s$) |

### 7.1. Evaluation Frameworks

**Baselines.** We evaluated the CPU and GPU baselines on a real system equipped with an Intel® Xeon Gold 5118 [103] CPU and an NVIDIA® GeForce RTX 3080 Ti [104] GPU. The CPU versions of our evaluated workloads employ SSE2 and SSE4 Intel® Streaming SIMD Extensions. We evaluate the FPGA baseline using high-level synthesis (HLS) implementations created with Vitis 2020.1 [123] and Vivado 2020.1 [124], and perform post-synthesis simulation for a Xilinx Zynq UltraScale ZCU102 FPGA [105]. For the evaluation of the PnM baseline, we simulate an HMC-based system [67] with support for bulk bitwise operations as described in Ambit [84] and bit shifting as described in DRISA [79]. We simulate various configurations of pLUTo on DDR4 [110] and HMC [67] memory models using a custom-built simulator, which we have made publicly available at [125, 126] under an open-source license.

**pLUTo.** Our simulator estimates the performance of pLUTo operations by parsing the sequence of memory commands required to perform them and enforcing the memory's timing parameters. The simulator then outputs the total time elapsed and energy consumed to complete the operations.

**Energy and Area.** We evaluate the energy consumption and area overhead of pLUTo configurations using CACTI 7 [127] DDR4 and HMC models. These models supply the energy consumption of each memory command and the area of each memory component. Using these values, we extrapolate pLUTo's

energy consumption and area overhead by considering the transistor count associated with the logic required to implement its functionality, including 1) the addition of the match logic, 2) modifications to the subarray architecture and memory controller, and 3) the addition of the pLUTo controller.

### 7.2. Workloads

Table 4 shows the names and characteristics of the workloads we evaluate. We select these workloads because 1) they exemplify general-purpose, real-world functions that cannot be efficiently executed by previous Processing-using-Memory architectures [75, 84] (e.g., substitution tables [128, 129], polynomial division [130]), 2) they include segments that are well-suited for LUT-based computation, and 3) their typical working sets are much larger than the cache size of modern systems.

**Table 4: Evaluated workloads.**

| Name | Parameters |
|---|---|
| Vector Addition, LUT-based [131] | Element width: 4 bits |
| Vector Point-Wise Multiplication [131] | Q Format: Q1.7, Q1.15 |
| Row-Level Bitwise Logic Operations [84] | # LUT entries: 4 |
| Bit Counting [130] | BC-4: 4 bits, 16-entry LUT; BC-8: 8 bits, 256-entry LUT |
| CRC-8/16/32 [130] | Packet size: 128 B |
| Salsa20 [128], VMPC [129] | Packet size: 512 B |
| Image Binarization (ImgBin) [132] | 3-channel 8-bit image, 936000 pixels; threshold: 50% |
| Color Grading (ColorGrade) [133] | One 3-channel 8-bit image, 936000 pixels; 8-bit to 8-bit |

## 8. Evaluation

In this section, we evaluate pLUTo's reliable and correct operation (Section 8.1), performance (Section 8.2), energy consumption (Section 8.3), and area overhead (Section 8.4). We also carry out performance sensitivity analyses to assess the cost of loading LUTs (Section 8.5), the scalability of pLUTo (Section 8.6), `tFAW`'s impact on performance (Section 8.7), and the effect of varying degrees of subarray-level parallelism (Section 8.8). Finally, we discuss how pLUTo compares to various prior approaches (Section 8.9).

### 8.1. Reliability and Correctness

We perform circuit-level SPICE simulations to verify that the modifications required by each of the three pLUTo designs do not compromise the correct and reliable operation of DRAM. We model the effect of activating a DRAM row in unmodified DRAM and in each of the three designs of pLUTo. We model DRAM cells based on Low-Power 22nm Metal Gate PTM transistors [134], and conduct Monte Carlo simulations of 100 runs, where the process variation is assumed to be 5%. Figure 6 shows the results of these simulations. Our results show that the proposed changes in each of the three pLUTo designs *do not introduce* errors in DRAM operation. The observed disturbances in the final bitline voltage following a row activation correspond to only 0.9% of the reference voltage value.

We make three key observations. First, the correctness of the row activation is not affected in any of the proposed designs, since the bitline voltage reaches the value required to trigger the activation in all cases. Second, in all pLUTo designs, the activation time is *not* affected by the introduced DRAM modifications. Third, the activation procedure is the noisiest for pLUTo-GSA. This is expected due to the operational principles of pLUTo-GSA (Section 5.2), whereby the contents of DRAM cells in consecutive rows are shared with the bitline *without* precharging the array after each activation until the end of the pLUTo Row Sweep. However, we observe correct row activation behavior even in this case.

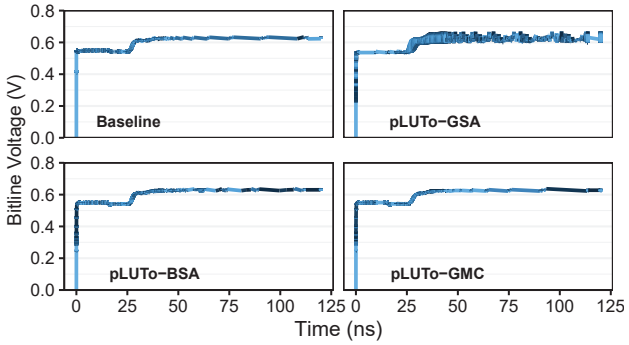


**Figure 6: Bitline voltage level over time in response to wordline activation at $time = 0$. Shades of blue indicate different runs of our Monte Carlo simulation.**

### 8.2. Performance

Figure 7 shows the performance of GPU, PnM, and pLUTo systems, normalized to the baseline CPU. For the DDR4 (3DS) implementation, the performance of pLUTo-GSA, pLUTo-BSA, and pLUTo-GMC mechanisms, on average across all workloads, is 357×/0.6×/9.2× (496×/0.8×/12.7×), 713×/1.2×/18.3× (990×/1.6×/25.4×), and 1413×/2.3×/36.2× (1962×/3.2×/50.3×) that of the CPU/GPU/PnM baselines, respectively. All pLUTo designs achieve performance comparable to or higher than that of the GPU, and consistently outperform the PnM baseline. We make two key observations. First, the 3DS-based pLUTo designs consistently outperform their DDR4 counterparts by 38% on average across the three pLUTo designs. This is due to HMC's faster row activations, which lead to faster pLUTo LUT Queries. Second, the CRC workloads show the smallest overall benefit from execution in pLUTo. The speedup in these workloads is bottlenecked by a serial reduction step, which must be performed in the CPU (pLUTo-DDR4) or in the logic layer of HMC (pLUTo-3DS). Nevertheless, the acceleration of the parallel portion of the CRC workloads still allows most pLUTo designs to achieve performance comparable to or higher than that of the GPU. We conclude that pLUTo significantly improves the performance of a variety of workloads compared to both processor-centric and PnM architectures.

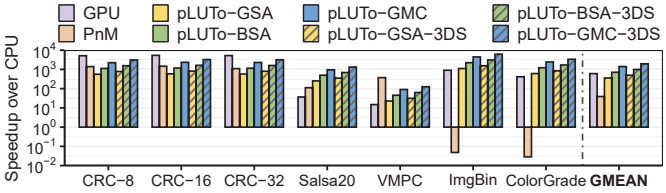


**Figure 7: Speedup of GPU, PnM, and pLUTo relative to the baseline CPU. The y-axis uses a logarithmic scale; higher is better.**

**8.2.1. Performance per Area.** Figure 8 shows the speedup per unit area of GPU and pLUTo systems, normalized to the baseline CPU. The area overhead of pLUTo-3DS designs is calculated assuming an area overhead of 4.4 $mm^2$ per vault [11, 48, 67]. For the DDR4 (3DS) implementation, the performance per unit area of pLUTo-GSA, pLUTo-BSA, and pLUTo-GMC mechanisms, on average across all workloads, is 426×/441× (12405×/12856×), 801×/830× (24747×/25646×), and 1504×/1558× (39245×/40670×) that of the CPU/GPU baselines, respectively. We make three key observations. First, all pLUTo designs provide substantially higher performance per unit area than both the CPU and the GPU (4283× and 2577×, respectively, on average across all pLUTo designs), and consistently outperform both baselines by a wide margin. This improvement is considerably greater than the one observed when considering performance in isolation (Section 8.2) and highlights the potential benefits of scaling pLUTo's design further with larger DRAM systems. Second, pLUTo-3DS designs are more area-efficient than their DDR4 counterparts across all workloads. This is a consequence of the large available area and 3D density in the HMC substrate. Third, we observe pLUTo's greatest improvements in Salsa20 and VMPC (respectively 2970×/50591× and 273×/106151× the performance per unit area of the CPU/GPU, on average across all pLUTo designs). These workloads are very memory-intensive, and are therefore well-suited for in-memory execution. In contrast, for the Salsa20 and VMPC workloads, the GPU baseline provides performance per unit area results that fall below even those of the baseline CPU, which highlights the negative impact of data movement bottlenecks in the execution of memory-intensive workloads under a processor-centric computing paradigm.

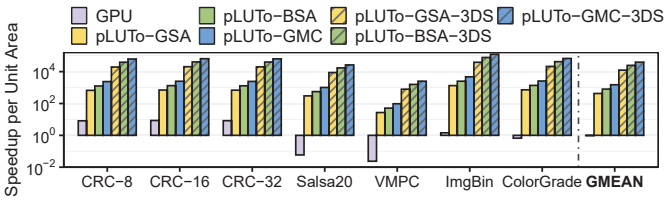


**Figure 8: Speedup of GPU and pLUTo relative to CPU, normalized to area. The y-axis uses a logarithmic scale; higher is better.**

**8.2.2. Comparison with FPGA.** Figure 9 shows the performance of the evaluated pLUTo systems normalized to the baseline FPGA. We observe that pLUTo outperforms the FPGA baseline across all workloads we evaluate. For the DDR4 (3DS) implementation, the performance of pLUTo-GSA, pLUTo-BSA, and pLUTo-GMC mechanisms, on average across all workloads, is 160× (111×), 274× (190×), and 459× (318×) that of the FPGA baseline, respectively. The most significant gains are associated with workloads that rely on smaller LUTs (e.g., BC4, ImgBin), and the smallest gains correspond to operations with large input bit widths (e.g., MUL16). We conclude that, although both the FPGA and pLUTo rely on LUT-based computation, the former's access to data in memory is still limited by main memory bandwidth. In contrast, pLUTo exploits much higher main memory bandwidth via the pLUTo LUT Query, leading to overall higher performance.

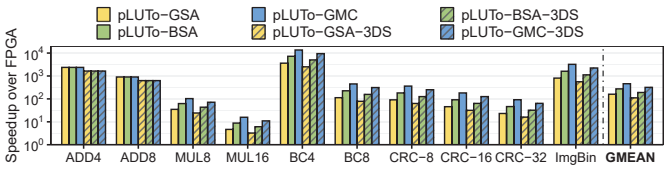


**Figure 9: pLUTo speedup relative to the baseline FPGA. The y-axis uses a logarithmic scale; higher is better.**

## 8.3. Energy Consumption

Figure 10 shows the energy consumed by the GPU and pLUTo systems when executing the evaluated workloads, normalized to the baseline CPU. pLUTo's energy consumption depends on the total number of DRAM operations required by the executed pLUTo ISA instructions (Table 2), and therefore *does not vary with different degrees of subarray-level parallelism.* For the DDR4 (3DS) implementation, pLUTo-GSA, pLUTo-BSA, and pLUTo-GMC systems, on average across all workloads, consume 1361.7×/29× (154.3×/3.3×) 1855×/39.5× (235.8×/5×) 3071.4×/65.3× (430.8×/9.2×) less energy than the CPU/GPU baselines, respectively.

We make two key observations. First, the energy savings enabled by pLUTo are considerably greater in workloads that are especially memory-intensive (e.g., VMPC) or require simple operations (e.g., ImgBin), but becomes lower as workload complexity increases (e.g., CRC-8/16/32). This trend is consistent with our observations from Section 8.2.1. Second, in some of the workloads (e.g., CRC-8/16/32, Salsa20), the energy consumption values of all three pLUTo designs are similar to each other. This is due to the relatively small number of pLUTo Row Sweep operations required to execute these workloads, which are *not* enough to highlight the differences in energy consumption of each pLUTo design. As a result, the overall impact of the improved efficiency of the pLUTo Row Sweep in pLUTo-BSA and pLUTo-GMC relative to pLUTo-GSA becomes less pronounced. We conclude that pLUTo significantly reduces energy consumption compared to processor-centric architectures for various workloads.

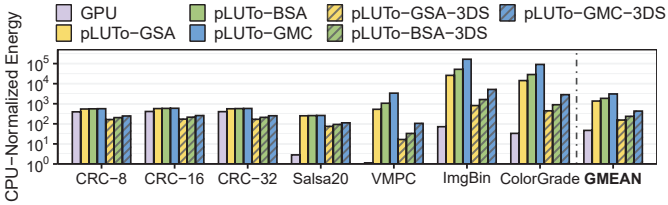


**Figure 10: Energy consumption of GPU and pLUTo compared to the CPU. The y-axis uses a logarithmic scale; higher is better.**

## 8.4. Area Overhead

Table 5 shows the estimated area of the baseline DRAM and three pLUTo designs, broken down by DRAM component. These estimates are derived from transistor count estimates and rely on the DRAM area models provided by CACTI 7 [127].

**pLUTo-GSA.** The estimated area overhead of the matchline-controlled switch (shown in Figure 4b) is 20% of the area of a sense amplifier per bitline. The total area overhead of pLUTo-GSA is 10.2% of the DRAM chip area.

**pLUTo-BSA.** The estimated area overhead of the matchline-controlled switch and the FF (shown in Figure 4a) is 60% of the area taken up by sense amplifiers in the base DRAM chip. The total area overhead of pLUTo-BSA is 16.7% of the DRAM chip area.

**pLUTo-GMC.** The estimated area overhead of the matchline-controlled switch per 2T1C DRAM cell (shown in Figure 4c) is 25%. The total area overhead of pLUTo-GMC is 23.1% of the DRAM chip area.

**Table 5: Area breakdown for DRAM and the three pLUTo designs.**

| | | Base DRAM | pLUTo-GSA | pLUTo-BSA | pLUTo-GMC |
|---|---|---|---|---|---|
| Area ($mm^2$) | DRAM Cell | 45.23 | 45.23 | 45.23 | **56.53** |
| | Local WL driver | 12.45 | 12.45 | 12.45 | 12.45 |
| | Match Logic | - | **4.61** | **4.61** | **4.61** |
| | Match Lines | - | **0.02** | **0.02** | **0.02** |
| | Sense Amp | 11.40 | **13.67** | **18.23** | 11.40 |
| | Row Decoder | 0.16 | **0.47** | **0.47** | **0.47** |
| | Column Decoder | 0.01 | 0.01 | 0.01 | 0.01 |
| | Other | 0.99 | 0.99 | 0.99 | 0.99 |
| | Total | 70.23 | 77.44 (+10.2%) | 82.00 (+16.7%) | 86.47 (+23.1%) |

The area overheads of the match logic and match lines described in Section 5.1.2, which are identical for all three designs, are shown separately in Table 5. The Row Decoder overhead includes that of the logic required for the pLUTo Row Sweep operation. The only pLUTo design that requires modifications to the DRAM cell design is pLUTo-GMC. Its per-cell overhead is indicated in the DRAM Cell row of Table 5. In the baseline system, DRAM cell access transistors take up approximately 15.1 $mm^2$. The overhead associated with these transistors doubles in pLUTo-GMC's 2T1C cell.

## 8.5. LUT Loading Overhead

Figure 11 shows the fraction of total computation time spent loading LUT data (y-axis), as a function of the total volume of data queried (x-axis). We evaluate two scenarios for the loading of LUTs, as described in Section 6.5: *Loading from Memory* and *Loading from Secondary Storage*. For the first scenario, we assume a DDR4 memory bandwidth of 19.2 GB/s [135]; for the second scenario, we assume an M.2 SSD bandwidth of 7500 MB/s [136]. As an example of how to interpret this plot, if data is *loaded from memory* (i.e., from DRAM), when querying around 20 MB of data (x-axis), approximately 10% of the LUT query execution time (y-axis) is spent loading the LUT's data into DRAM (accordingly, 90% of the execution time is spent performing pLUTo LUT Queries).

We make three key observations. First, it is sufficient to process 1.9 MB of data in the DDR4-based scenario (◆ in Figure 11) for the LUT loading time to equal the LUT query time. This observation illustrates that the cost of loading LUTs into DRAM can be quickly amortized, even for small volumes of input data. Second, as the volume of data to be processed increases, the fraction of time spent loading the LUTs into memory quickly decreases, both for the DDR4- and the SSD-based scenarios. For example, for an input of 120 MB of data (▲ in Figure 11), the fraction of time spent loading LUTs is only about 2% in the case of DDR4. Third, although loading LUTs from the SSD takes longer than doing so from DRAM, this difference does *not* significantly impact the volume data that needs to be queried to amortize the LUT loading time. We conclude that loading LUTs both from DRAM and from a secondary storage device (i.e., M.2 SSD) are well-suited and practical approaches to support pLUTo's operation.

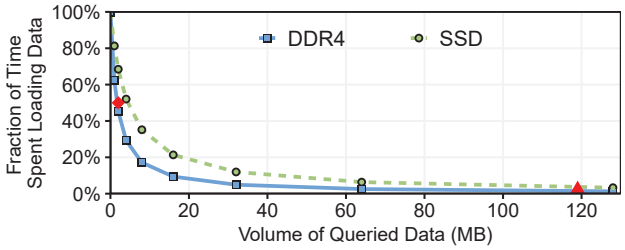


**Figure 11: Fraction of time spent loading LUTs (from DDR4 DRAM [135] and M.2 SSD [136]) versus the volume of LUT input data.**

## 8.6. Scalability Analysis

This section analyzes the scalability of pLUTo's LUT query operation. Our goals are 1) to fundamentally understand the performance limits of the pLUTo LUT Query, and 2) to study the suitability of different PiM architectures for multiplication, a commonly used operation. First, in Figure 12a, we show an analysis of the throughput and energy consumption scaling of the three proposed pLUTo designs, as determined by the equations and timing parameters in Table 1 and by the equations derived in Sections 5.1.4, 5.2.3 and 5.3.4. Second, in Figure 12b, we compare the energy efficiency (in # `Multiplications`/$J$) of three systems: 1) pLUTo-BSA, 2) a bit-serial PuM mechanism (SIMDRAM [75]), and 3) our baseline PnM device, while varying the bit width of the operands involved in the multiplication. We note that the multiplication operation is especially costly for SIMDRAM to execute, as discussed in Section 8.9.

We make two key observations from the figures. First, all three pLUTo designs provide high throughput and low energy consumption for small LUT query sizes ($N \leq 8$). This happens

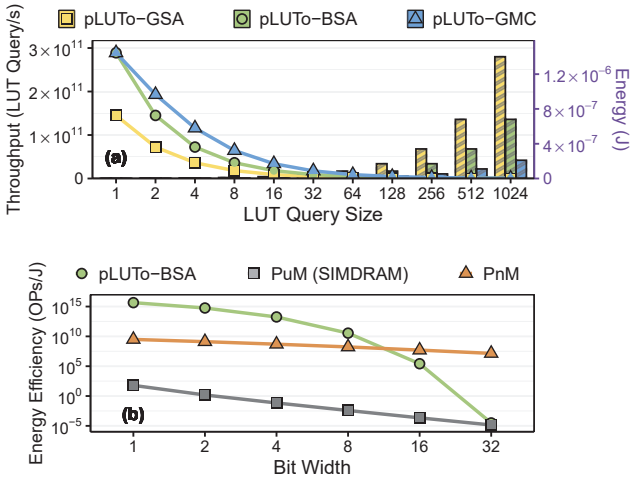


**Figure 12: Scalability analysis for pLUTo's LUT query operation: (a) shows LUT query throughput (lines in the primary/left y-axis) and energy consumption (bars in the secondary/right y-axis) for the three pLUTo designs while varying LUT query size; (b) compares the energy efficiency (in OPs/J) of pLUTo-BSA against a prior PuM mechanism (SIMDRAM [75]) and the PnM baseline.**

because, as Table 1 shows, the latency and energy consumption of a pLUTo Row Sweep increase linearly with the LUT query size (i.e., consequently, the throughput of a pLUTo Row Sweep *decreases* linearly with the LUT query size). Therefore, pLUTo achieves its highest throughput and lowest energy consumption for small LUT query sizes. Second, pLUTo provides higher energy efficiency than the alternative SIMDRAM and PnM Processing-in-Memory architectures for low-precision multiplication (bit width $\leq 8$ bits). Executing multiplication in pLUTo leads to better energy efficiency than in SIMDRAM for all evaluated bit widths. This happens because executing bit-serial multiplications (as SIMDRAM does) incurs a quadratic number of DRAM activations [75]. We conclude that pLUTo is well-suited to perform low-bit-width LUT queries, which can be adopted alongside alternative solutions (e.g., SIMDRAM and PnM) to take full advantage of the underlying DRAM substrate.

## 8.7. Impact of `tFAW` on Performance

The `tFAW` timing parameter limits the activation rate in a DRAM chip to meet power and reliability constraints. Since the activation operation is central to pLUTo, it is important to evaluate the impact of this parameter on performance. Figure 13 shows the effect on the performance of a single pLUTo LUT Query of varying `tFAW` between 0% and 100% of its *nominal value* (i.e., the actual `tFAW` value in the DRAM chip we model, 13.328 ns), across our examined workloads. When `tFAW` = 0%, it is possible to issue an *unlimited* number of activations concurrently; when `tFAW` = 50%, it is possible to issue *twice* as many activations per unit of time as commodity DRAM permits; when `tFAW` = 100%, the number of activations issued per unit of time corresponds to *exactly* as many as commodity DRAM permits.

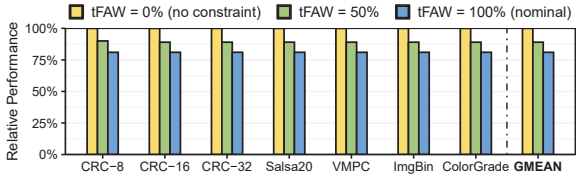


**Figure 13: The impact of tFAW on the performance of pLUTo.**

We make two key observations. First, the performance loss is approximately 10% for `tFAW` = 50%, and approximately 20% for `tFAW` = 100%, compared to not having the `tFAW` limitation at all. Even accounting for this performance penalty of `tFAW`, pLUTo still outperforms the CPU and GPU baselines, as shown in Section 8.2. Second, the performance penalty is very similar across all of the evaluated workloads for the same value of `tFAW`. We conclude that, despite `tFAW`'s limited impact on pLUTo, the use of more powerful charge pumps could further relax power constraints of pLUTo-capable DRAMs.

## 8.8. Effect of Subarray-Level Parallelism

We evaluate the three pLUTo designs (pLUTo-GSA, pLUTo-BSA, pLUTo-GMC) with varying degrees of subarray-level parallelism for both DDR4 and 3D-stacked memory. Figure 14 plots the speedup (averaged across all evaluated workloads) of each configuration against the baseline CPU. We make two key observations. First, due to the different row buffer sizes

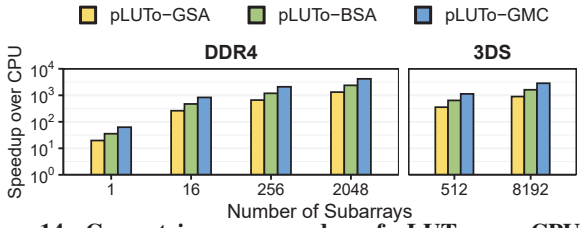


**Figure 14: Geometric mean speedup of pLUTo over CPU, for varying degrees of subarray-level parallelism. The y-axis uses a logarithmic scale; higher is better.**

of DDR4 (8 kB) and 3DS (256 B) memories (as discussed in Section 7), the same degree of subarray-level parallelism results in higher speedup for pLUTo-DDR4 than pLUTo-3DS. Second, performance scaling is approximately proportional to the number of subarrays operating in parallel in both cases, provided that the size of the input to be queried is sufficiently large. This linear relationship between subarray count and performance improvement shows that 1) although the volume of internal data movement required by pLUTo's operation increases with the degree of subarray-level parallelism, this factor does *not* limit pLUTo's scalability, and 2) pLUTo should be configured to operate with as high a degree of subarray-level parallelism as the memory technology supports.[6]

## 8.9. Comparison With Prior PuM Systems

As demonstrated in Table 6 and discussed in Section 3, prior DRAM-based PuM architectures (e.g., [79, 84, 96]) achieve very high throughput and energy efficiency, but do so while supporting a very limited range of operations. These works can address this limitation by exploiting alternatives to conventional bit-parallel algorithms. For example, it is possible to efficiently realize arithmetic operations in Ambit [84] using bit-serial algorithms [75]. We show that the additional flexibility afforded by pLUTo's native support for LUT operations allows it to outperform prior PuM architectures. We substantiate this claim with Table 6, which shows the latency of each operation of interest under Ambit [84], SIMDRAM [75], LAcc [96], DRISA [79] and pLUTo. In each case, we assume the use of ideal data layout for each system as defined in the original works (e.g., bit-parallel for pLUTo, bit-serial for SIMDRAM) and report the best-case *achievable* performance for each design.[7] As an example, bitwise operations between input sets `A` ('$a_1a_2$...') and `B` ('$b_1b_2$...'), under pLUTo's LUT-based paradigm require that input operands be shuffled ('$a_1b_1a_2b_2$...'); in contrast, the input sets `A` and `B` are ideally stored in two separate DRAM rows for all prior PuM designs (Ambit [84], SIMDRAM [75], LAcc [96], DRISA [79]). To make fair comparisons, the memory capacity for each design is such that the area overheads for all designs remain in a narrow range (the area values across the five PuM comparison points average 62.5 $mm^2$ ± 5.2 $mm^2$), which is similar to the area of commodity DRAM devices (≈70.23 $mm^2$, see Table 5). As a result, due to its inferior storage density, the capacity of DRISA [79] is limited to 2 GB.

[6] As discussed in Section 8.3, energy consumption is *not* affected by different degrees of subarray-level parallelism.

[7] Supporting each architecture's ideal data layout requires system-level changes.

**Table 6: Comparison of operations supported by pLUTo vs. prior PuM. All performance per area and energy efficiency values are normalized to pLUTo-BSA with 4-subarray parallelism.**

| | Ambit [84] | SIMDRAM [75] | LAcc [96] | DRISA [79] | pLUTo-BSA |
|---|---|---|---|---|---|
| Capacity | 8 GB | 8 GB | 8 GB | 2 GB | 8 GB |
| Area ($mm^2$) | 61.0 | 61.1 | 54.8 | 65.2 | 70.5 |
| Power ($W$) | 5.3 | 5.3 | 5.3 | 98.0 | 11 |
| NOT ($ns$) | 135.0 | 135.0 | 135.0 | 207.6 | 105.0 |
| AND ($ns$) | 270.0 | 270.0 | 270.0 | 415.2 | 165.0 |
| OR ($ns$) | 270.0 | 270.0 | 270.0 | 415.2 | 165.0 |
| XOR ($ns$) | 585.0 | 585.0 | 450.0 | 691.9 | 165.0 |
| XNOR ($ns$) | 585.0 | 585.0 | 450.0 | 691.9 | 165.0 |
| **Performance Per Area (higher is better)** | **0.54** | **0.54** | **0.67** | **0.37** | **1.00** |
| **Energy Efficiency (higher is better)** | **0.54** | **0.54** | **0.67** | **0.02** | **1.00** |
| 4-bit Addition ($ns$) | 5081.0 | 1585.0 | 1142.3 | 1756.5 | 1920.0 |
| 4-bit Multiplication ($ns$) | 19065.0 | 7451.0 | 5365.4 | 8250.1 | 1920.0 |
| 4-bit Bit Counting ($ns$) | 2936.0 | 1156.0 | - | 6649.9 | 120.0 |
| 8-bit Bit Counting ($ns$) | 6901.0 | 2696.0 | - | 13580.0 | 1920.0 |
| **Performance Per Area (higher is better)** | **0.34** | **0.45** | **1.00**$*$ | **0.17** | **1.00** |
| **Energy Efficiency (higher is better)** | **0.69** | **0.94** | **2.00**$*$ | **0.02** | **1.00** |
| 6-bit to 2-bit LUT Query ($ns$) | - | - | - | - | 480.0 |
| 8-bit to 8-bit LUT Query ($ns$) | - | - | - | - | 1920.0 |
| 8-bit Binarization ($ns$) | - | - | - | - | 1920.0 |
| 8-bit Exponentiation ($ns$) | - | - | - | - | 1920.0 |

– indicates that the operation is *not* supported by the proposed mechanism.
$*$ indicates that the result was obtained from partial data.

We draw three key observations from Table 6. First, due to their complexity, some operations (e.g., binarization, exponentiation) *cannot* be implemented in a time-efficient manner using any prior design. In pLUTo, it is possible to perform exponentiation with high efficiency when operating on small bit widths (for best results, up to 8 bits). Second, pLUTo's throughput for bitwise logic operations matches or exceeds that of all prior works. Third, pLUTo consistently outperforms all four other approaches for most of the considered operations in performance (absolute and normalized to area) and energy efficiency. This improvement is not universal: for instance, pLUTo slightly lags behind all baselines for 4-bit addition. We conclude that pLUTo achieves its main goal of extending the range and complexity of operations that the DRAM-based PuM paradigm supports.

## 9. Case Study: Quantized Neural Networks

Building on the observation that pLUTo is especially well-suited for executing low-bit-width operations (Section 8.9), we evaluate the benefits of pLUTo on neural networks quantized to 1 and 4 bits, an emerging machine learning application that is especially useful for power-limited devices [137–139]. We evaluate a quantized version of the LeNet-5 network [140] to classify digits from the MNIST dataset [140] as a proof of concept. For this evaluation, the CPU and FPGA baselines are those described in Section 7.1; the GPU is a data-center-grade NVIDIA P100 [141], which was specifically developed for machine learning applications. Table 7 shows the inference times for CPU, GPU, FPGA, and pLUTo-BSA. pLUTo-BSA outperforms the CPU (10×, 30× for 1-bit, 4-bit inference), the GPU (2×, 7×) and the FPGA (6×, 19×) in inference time. pLUTo's performance improvements for these operations result from the bulk querying of input values using only short sequences of DRAM commands to perform bitwise logic operations. Simultaneously, pLUTo provides large energy savings over both the CPU (110×, 109×), the GPU (80×, 81×) and the FPGA (15×, 16×), for both 1- and 4-bit precision. The key reason behind this increase in energy efficiency is the reduction in overall data movement (since many operations are performed in-place) and the energy efficiency of DRAM commands. We conclude that pLUTo is well-suited to accelerate quantized neural network inferencing and to reduce the energy cost of this workload, especially in heavily energy-constrained devices, such as edge and IoT devices.

**Table 7: LeNet-5 inference times (in µs) and energy (in mJ) for CPU, GPU, FPGA, and pLUTo.**

| Bit Width | Accuracy [138] | CPU | | GPU | | FPGA | | pLUTo-BSA | |
|---|---|---|---|---|---|---|---|---|---|
| | | Time | Energy | Time | Energy | Time | Energy | Time | Energy |
| 1 bit | 97.4 % | 249 | 2.2 | 56 | 1.6 | 141 | 0.3 | 23 | 0.02 |
| 4 bits | 99.1 % | 997 | 8.7 | 224 | 6.5 | 563 | 1.3 | 30 | 0.08 |

## 10. Related Work

To our knowledge, pLUTo is the first work that enables the efficient bulk querying of lookup tables (LUTs) inside DRAM to enable the execution of complex operations. In this section, we describe relevant prior works.

**Processing-using-Memory (PuM).** Many prior works propose various forms of Processing-using-Memory [41, 55, 68–92, 96, 98, 108, 142–193]. All these approaches provide significant performance and energy improvements, but focus mainly on a reduced set of operations (e.g., data movement [81, 108], bulk bitwise operations [68, 78, 84, 188], or neural network acceleration [70, 71, 79, 96]). By combining the pLUTo LUT Query with fast and efficient bitwise logic [84] and bit shifting operations [79] enabled by these prior works, pLUTo supports a much wider range of operations. While pPIM [97] and LAcc [96], for example, leverage dedicated LUT hardware for neural network acceleration, the pLUTo LUT Query is suitable for a greater range of operations (by supporting a broader set of input-output configurations, with greater performance and energy efficiency). DRAF [99] employs a DRAM-based FPGA-like LUT-based computing paradigm that allows it to outperform an FPGA baseline in area and energy efficiency; however, DRAF lags in throughput and latency. In contrast, pLUTo enables high-throughput LUT queries without compromising energy efficiency and with small overhead (between 10.2% and 23.1%, for different versions of pLUTo) on the storage density of the DRAM array.

**Processing-near-Memory (PnM).** 3D-stacked memories [67, 194, 195] enable the stacking of memory layers on top of a logic layer [11, 20, 21, 23–25, 28–32, 34, 39, 44–50, 55, 57, 59, 60, 66, 196–212]. This technology provides higher bandwidth compared to 2D DRAM. pLUTo is *complementary* to 3D-stacked memory: the two can be combined as shown in Section 8.

**Content-Addressable Memories (CAMs).** CAM-based accelerators (e.g., [192, 213–219]) return the address of matched data given an input query and can therefore be used to perform LUT-based computing. Most CAMs are SRAM-based and provide low area density compared to DRAM-based memories. DRAM-based CAMs also exist [220–222], but require a greater number of transistors per cell than pLUTo-GMC, our most expensive design.

**Automata Processors (APs).** APs are specialized processing engines that support the hardware-native execution of nondeterministc-finite-automata [223, 224]. These automata may be used for pattern recognition tasks and the querying of unstructured data sources, but have a relatively narrow domain of applications and are therefore not well-suited for the offloading of complex functions by memoization.

## 11. Conclusion

We introduced pLUTo, a new DRAM-based Processing-using-Memory architecture that enables the storage and bulk querying of lookup tables completely in-DRAM. We build pLUTo based on the key observation that enabling bulk lookup-table-query operations inside DRAM enables the efficient execution of complex operations with high performance and energy efficiency. We describe 1) the hardware design of three different pLUTo architectures, each providing a different trade-off between performance, energy efficiency, and area overhead, and 2) the necessary system integration support to enable the execution of in-DRAM pLUTo operations. Our evaluations show that pLUTo significantly outperforms the baseline CPU-, GPU-, FPGA-, PnM-, and PuM-based systems in terms of execution time, performance per area, and energy consumption. We hope that future work explores new ways of taking advantage of pLUTo and in-DRAM LUT-based computing to provide even greater performance and energy benefits for more applications that can take advantage of the PuM paradigm.

## Acknowledgments

We thank the anonymous reviewers of HPCA (2020), ISCA (2020, 2021, 2022), ICCD (2020), MICRO (2021, 2022), and ASPLOS (2022) for their valuable comments and feedback. We thank the SAFARI Research Group members for their valuable feedback and the stimulating intellectual environment they provide. We acknowledge the generous gifts provided by our industrial partners: Google, Huawei, Intel, Microsoft, and VMware. This work was supported in part by the Semiconductor Research Corporation (SRC) and the ETH Future Computing Laboratory.

This work was supported in part by the Instituto de Telecomunicações and the Fundação para a Ciência e a Tecnologia (FCT), under grant numbers UIDB/50008/2020-UIDP/50008/2020 and EXPL/EEI-HAC/1511/2021.

## References

[1] S. Ghose *et al.*, "The Processing-in-Memory Paradigm: Mechanisms to Enable Adoption," in *Beyond-CMOS Technologies for Next Generation Computer Design*, 2019.
[2] O. Mutlu *et al.*, "Processing Data Where It Makes Sense: Enabling In-Memory Computation," in *Microprocessors and Microsystems*, 2019.
[3] O. Mutlu *et al.*, "Enabling Practical Processing in and Near Memory For Data-Intensive Computing," in *DAC*, 2019.
[4] S. Ghose *et al.*, "Processing-in-Memory: A Workload-Driven Perspective," in *IBM J. Res. Dev.*, 2019.
[5] P. Siegl *et al.*, "Data-Centric Computing Frontiers: A Survey on Processing-in-Memory," in *MEMSYS*, 2016.
[6] D. Pandiyan and C. Wu, "Quantifying the Energy Cost of Data Movement for Emerging Smart Phone Workloads on Mobile Platforms," in *IISWC*, 2014.
[7] S. Kanev *et al.*, "Profiling a Warehouse-Scale Computer," in *ISCA*, 2015.
[8] I. Paul *et al.*, "Harmonia: Balancing Compute and Memory Power in High-Performance GPUs," in *ISCA*, 2015.
[9] M. Ware *et al.*, "Architecting for Power Management: The IBM® Power7™ Approach," in *HPCA*, 2010.
[10] C. Lefurgy *et al.*, "Energy Management for Commercial Servers," in *Computer*, 2003.
[11] A. Boroumand *et al.*, "Google Workloads for Consumer Devices: Mitigating Data Movement Bottlenecks," in *ASPLOS*, 2018.
[12] T. Vogelsang, "Understanding the Energy Consumption of Dynamic Random Access Memories," in *MICRO*, 2010.
[13] W. A. Wulf and S. A. McKee, "Hitting the Memory Wall: Implications of the Obvious," in *CAN*, 1995.
[14] B. Dally, "The Path to Exascale Computing," https://bit.ly/3CIzc1l, 2015.
[15] G. F. Oliveira *et al.*, "DAMOV: A New Methodology and Benchmark Suite for Evaluating Data Movement Bottlenecks," *IEEE Access*, 2021.
[16] O. Mutlu, "Memory Scaling: A Systems Architecture Perspective," in *IMW*, 2013.
[17] O. Mutlu and L. Subramanian, "Research Problems and Opportunities in Memory Systems," *SUPERFRI*, 2014.
[18] O. Mutlu *et al.*, "A Modern Primer on Processing in Memory," in *Emerging Computing: From Devices to Systems - Looking Beyond Moore and Von Neumann*, 2021.
[19] L. Nai *et al.*, "GraphPIM: Enabling Instruction-Level PIM Offloading in Graph Computing Frameworks," in *HPCA*, 2017.
[20] J. S. Kim *et al.*, "GRIM-Filter: Fast Seed Location Filtering in DNA Read Mapping using Processing-in-Memory Technologies," in *APBC*, 2018.
[21] J. Ahn *et al.*, "PIM-Enabled Instructions: A Low-Overhead, Locality-Aware Processing-in-Memory Architecture," in *ISCA*, 2015.
[22] M. Gao *et al.*, "TETRIS: Scalable and Efficient Neural Network Acceleration with 3D Memory," in *ASPLOS*, 2017.
[23] D. Kim *et al.*, "Neurocube: A Programmable Digital Neuromorphic Architecture with High-Density 3D Memory," in *ISCA*, 2016.
[24] D. S. Cali *et al.*, "GenASM: A High-Performance, Low-Power Approximate String Matching Acceleration Framework for Genome Sequence Analysis," in *MICRO*, 2020.
[25] B. Akin *et al.*, "Data Reorganization in Memory Using 3D-Stacked DRAM," in *ISCA*, 2016.
[26] K. Hsieh *et al.*, "Accelerating Pointer Chasing in 3D-Stacked Memory: Challenges, Mechanisms, Evaluation," in *ICCD*, 2016.
[27] O. O. Babarinsa and S. Idreos, "JAFAR: Near-Data Processing for Databases," in *SIGMOD*, 2015.
[28] A. Boroumand *et al.*, "Mitigating Edge Machine Learning Inference Bottlenecks: An Empirical Study on Accelerating Google Edge Models," arXiv:2103.00768 [cs.AR], 2021.
[29] A. Boroumand *et al.*, "Google Neural Network Models for Edge Devices: Analyzing and Mitigating Machine Learning Inference Bottlenecks," in *PACT*, 2021.
[30] A. Boroumand *et al.*, "Polynesia: Enabling High-Performance and Energy-Efficient Hybrid Transactional/Analytical Databases with Hardware/Software Co-Design," in *ICDE*, 2022.
[31] A. Boroumand *et al.*, "Polynesia: Enabling Effective Hybrid Transactional/Analytical Databases with Specialized Hardware/Software Co-Design," arXiv:2103.00798 [cs.AR], 2021.
[32] A. Boroumand, "Practical Mechanisms for Reducing Processor-Memory Data Movement in Modern Workloads," Ph.D. dissertation, Carnegie Mellon University, 2020.
[33] N. M. Ghiasi *et al.*, "GenStore: A High-Performance and Energy-Efficient In-Storage Computing System for Genome Sequence Analysis," in *ASPLOS*, 2022.
[34] I. Fernandez *et al.*, "NATSA: A Near-Data Processing Accelerator for Time Series Analysis," in *ICCD*, 2020.
[35] G. Singh *et al.*, "NERO: A Near High-Bandwidth Memory Stencil Accelerator for Weather Prediction Modeling," in *FPL*, 2020.
[36] S. Lee *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.
[37] Y.-C. Kwon *et al.*, "A 20nm 6GB Function-in-Memory DRAM, Based on HBM2 with a 1.2TFLOPS Programmable Computing Unit Using Bank-Level Parallelism, for Machine Learning Applications," in *ISSCC*, 2021.
[38] S. Lee *et al.*, "Hardware Architecture and Software Stack for PIM Based on Commercial DRAM Technology: Industrial Product," in *ISCA*, 2021.
[39] D. Niu *et al.*, "184QPS/W 64Mb/$mm^2$ 3D Logic-to-DRAM Hybrid Bonding with Process-Near-Memory Engine for Recommendation System," in *ISSCC*, 2022.
[40] P. Rosenfeld, "Performance Exploration of the Hybrid Memory Cube," Ph.D. dissertation, University of Maryland, 2014.
[41] G. F. Oliveira *et al.*, "Accelerating Neural Network Inference with Processing-in-DRAM: From the Edge to the Cloud," *IEEE Micro*, 2022.
[42] S. Cho *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.
[43] H. Shin *et al.*, "McDRAM: Low Latency and Energy-Efficient Matrix Computations in DRAM," *IEEE TCADICS*, 2018.
[44] J. Ahn *et al.*, "A Scalable Processing-in-Memory Accelerator for Parallel Graph Processing," in *ISCA*, 2015.
[45] A. Boroumand *et al.*, "LazyPIM: An Efficient Cache Coherence Mechanism for Processing-in-Memory," in *CAL*, 2016.
[46] D. Zhang *et al.*, "TOP-PIM: Throughput-Oriented Programmable Processing in Memory," in *HPDC*, 2014.
[47] M. Drumond *et al.*, "The Mondrian Data Engine," in *ISCA*, 2017.

[48] A. Boroumand *et al.*, "CoNDA: Efficient Cache Coherence Support for Near-Data Accelerators," in *ISCA*, 2019.
[49] K. Hsieh *et al.*, "Transparent Offloading and Mapping (TOM): Enabling Programmer-Transparent Near-Data Processing in GPU Systems," in *ISCA*, 2016.
[50] A. Pattnaik *et al.*, "Scheduling Techniques for GPU Architectures with Processing-in-Memory Capabilities," in *PACT*, 2016.
[51] F. Devaux, "The True Processing in Memory Accelerator," in *HC*, 2019.
[52] J. Gómez-Luna *et al.*, "Benchmarking Memory-Centric Computing Systems: Analysis of Real Processing-in-Memory Hardware," in *CUT*, 2021.
[53] J. Gómez-Luna *et al.*, "Benchmarking a New Paradigm: An Experimental Analysis of a Real Processing-in-Memory Architecture," arXiv:2105.03814 [cs.AR], 2021.
[54] J. Gómez-Luna *et al.*, "Benchmarking a New Paradigm: An Experimental Analysis of a Real Processing-in-Memory Architecture," *IEEE Access*, 2022.
[55] M. Besta *et al.*, "SISA: Set-Centric Instruction Set Architecture for Graph Mining on Processing-in-Memory Systems," in *MICRO*, 2021.
[56] C. Giannoula *et al.*, "SynCron: Efficient Synchronization Support for Near-Data-Processing Architectures," in *HPCA*, 2021.
[57] G. Singh *et al.*, "NAPEL: Near-Memory Computing Application Performance Prediction via Ensemble Learning," in *DAC*, 2019.
[58] C. Giannoula *et al.*, "SparseP: Towards Efficient Sparse Matrix Vector Multiplication on Real Processing-in-Memory Systems," in *SIGMETRICS*, 2022.
[59] M. Gao and C. Kozyrakis, "HRL: Efficient and Flexible Reconfigurable Logic for Near-Data Processing," in *HPCA*, 2016.
[60] P. C. Santos *et al.*, "Operand Size Reconfiguration for Big Data Processing in Memory," in *DATE*, 2017.
[61] G. F. Oliveira *et al.*, "NIM: An HMC-Based Machine for Neuron Computation," in *ARC*, 2017.
[62] A. Farmahini-Farahani *et al.*, "NDA: Near-DRAM Acceleration Architecture Leveraging Commodity DRAM Devices and Standard Memory Modules," in *HPCA*, 2015.
[63] L. Ke *et al.*, "Near-Memory Processing in Action: Accelerating Personalized Recommendation with AxDIMM," *IEEE Micro*, 2021.
[64] D. U. Lee *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.
[65] J. T. Pawlowski, "Hybrid Memory Cube (HMC)," in *HCS*, 2011.
[66] G. H. Loh, "3D-Stacked Memory Architectures for Multi-Core Processors," in *ISCA*, 2008.
[67] Hybrid Memory Cube Consortium *et al.*, "Hybrid Memory Cube Specification 2.1," *Retrieved from micron.com*, 2014.
[68] S. Aga *et al.*, "Compute Caches," in *HPCA*, 2017.
[69] P. Chi *et al.*, "PRIME: A Novel Processing-in-Memory Architecture for Neural Network Computation in ReRAM-Based Main Memory," in *ISCA*, 2016.
[70] Q. Deng *et al.*, "DrAcc: A DRAM Based Accelerator for Accurate CNN Inference," in *DAC*, 2018.
[71] C. Eckert *et al.*, "Neural Cache: Bit-Serial In-Cache Acceleration of Deep Neural Networks," in *ISCA*, 2018.
[72] J. Park *et al.*, "Flash-Cosmos: In-Flash Bulk Bitwise Operations Using Inherent Computation Capability of NAND Flash Memory," in *MICRO*, 2022.
[73] D. Fujiki *et al.*, "Duality Cache for Data Parallel Acceleration," in *ISCA*, 2019.
[74] F. Gao *et al.*, "ComputeDRAM: In-Memory Compute Using Off-The-Shelf DRAMs," in *MICRO*, 2019.
[75] N. Hajinazar *et al.*, "SIMDRAM: A Framework for Bit-Serial SIMD Processing Using DRAM," in *ASPLOS*, 2021.
[76] Z. He *et al.*, "Sparse BD-Net: A Multiplication-Less DNN with Sparse Binarized Depth-Wise Separable Convolution," *JETC*, 2020.
[77] M. Imani *et al.*, "FloatPIM: In-Memory Acceleration of Deep Neural Network Training with High Precision," in *ISCA*, 2019.
[78] S. Li *et al.*, "Pinatubo: A Processing-in-Memory Architecture for Bulk Bitwise Operations in Emerging Non-Volatile Memories," in *DAC*, 2016.
[79] S. Li *et al.*, "DRISA: A DRAM-Based Reconfigurable In-Situ Accelerator," in *MICRO*, 2017.
[80] V. Seshadri *et al.*, "Buddy-RAM: Improving the Performance and Efficiency of Bulk Bitwise Operations Using DRAM," arXiv:1611.09988 [cs:AR], 2016.
[81] V. Seshadri *et al.*, "RowClone: Fast and Energy-Efficient In-DRAM Bulk Data Copy and Initialization," in *MICRO*, 2013.
[82] V. Seshadri *et al.*, "Fast Bulk Bitwise AND and OR in DRAM," in *CAL*, 2015.
[83] 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.
[84] V. Seshadri *et al.*, "Ambit: In-Memory Accelerator For Bulk Bitwise Operations Using Commodity DRAM Technology," in *MICRO*, 2017.
[85] V. Seshadri and O. Mutlu, "Simple Operations in Memory to Reduce Data Movement," in *Adv. Comput.*, 2017.
[86] V. Seshadri *et al.*, "RowClone: Accelerating Data Movement and Initialization Using DRAM," arXiv:1805.03502 [cs.AR], 2018.
[87] V. Seshadri and O. Mutlu, "In-DRAM Bulk Bitwise Execution Engine," arXiv:1905.09822 [cs.AR], 2019.
[88] A. Shafiee *et al.*, "ISAAC: A Convolutional Neural Network Accelerator with In-Situ Analog Arithmetic in Crossbars," in *ISCA*, 2016.
[89] L. Song *et al.*, "PipeLayer: A Pipelined ReRAM-Based Accelerator for Deep Learning," in *HPCA*, 2017.
[90] L. Song *et al.*, "GraphR: Accelerating Graph Processing Using ReRAM," in *HPCA*, 2018.
[91] X. Xin *et al.*, "ELP2IM: Efficient and Low Power Bitwise Operation Processing in DRAM," in *HPCA*, 2020.
[92] V. Seshadri *et al.*, "Gather-Scatter DRAM: In-DRAM Address Translation to Improve the Spatial Locality of Non-Unit Strided Accesses," in *MICRO*, 2015.
[93] M. Hashemi *et al.*, "Accelerating Dependent Cache Misses With an Enhanced Memory Controller," in *ISCA*, 2016.
[94] D. Weber *et al.*, "Current and Future Challenges of DRAM Metallization," in *IITC*, 2005.
[95] Y. Peng *et al.*, "Design, Packaging, and Architectural Policy Co-Optimization for DC Power Integrity in 3D DRAM," in *DAC*, 2015.
[96] Q. Deng *et al.*, "LAcc: Exploiting Lookup Table-Based Fast and Accurate Vector Multiplication in DRAM-Based CNN Accelerator," in *DAC*, 2019.
[97] P. R. Sutradhar *et al.*, "pPIM: A Programmable Processor-in-Memory Architecture with Precision-Scaling For Deep Learning," in *CAL*, 2020.
[98] S. Angizi and D. Fan, "ReDRAM: A Reconfigurable Processing-in-DRAM Platform for Accelerating Bulk Bit-Wise Operations," in *ICCAD*, 2019.
[99] M. Gao *et al.*, "DRAF: A Low-Power DRAM-Based Reconfigurable Acceleration Fabric," in *ISCA*, 2016.
[100] Y. Kim *et al.*, "A Case for Exploiting Subarray-Level Parallelism (SALP) in DRAM," in *ISCA*, 2012.
[101] J. Kim *et al.*, "Solar-DRAM: Reducing DRAM Access Latency by Exploiting the Variation in Local Bitlines," in *ICCD*, 2018.
[102] D. Lee *et al.*, "Design-Induced Latency Variation in Modern DRAM Chips: Characterization, Analysis, and Latency Reduction Mechanisms," in *SIGMETRICS*, 2017.
[103] Intel, "Intel® Xeon® Gold 5118 Processor Specifications." [Online]. Available: https://intel.ly/3e3ZUXL
[104] NVIDIA, "NVIDIA GeForce RTX 3080 Ti Graphics Card." [Online]. Available: https://bit.ly/3Rl05Nr
[105] Xilinx, Inc., "ZCU102 Evaluation Board: User Guide," https://bit.ly/3Ky0TvJ, 2019.
[106] D. Lee *et al.*, "Tiered-Latency DRAM: A Low Latency and Low Cost DRAM Architecture," in *HPCA*, 2013.
[107] D. Lee *et al.*, "Decoupled Direct Memory Access: Isolating CPU and IO Traffic by Leveraging a Dual-Data-Port DRAM," in *PACT*, 2015.
[108] K. K. Chang *et al.*, "Low-Cost Inter-Linked Subarrays (LISA): Enabling Fast Inter-Subarray Data Movement in DRAM," in *HPCA*, 2016.
[109] "JEDEC Solid State Technology Assn.", "JESD79-3D: DDR3 SDRAM Standard," July 2012.
[110] "JEDEC Solid State Technology Assn.", "JESD79-4B: DDR4 SDRAM Standard," July 2017.
[111] P. S. Lazar and S. C. Oh, "DRAM with Total Self Refresh And Control Circuit," May 25 2004, US Patent 6,741,515.
[112] V. C. Patel *et al.*, "DRAM Power Management With Self-Refresh," Nov. 15 1994, US Patent 5,365,487.
[113] T. Y. Kim, "Self-Refresh Apparatus and Method," Sep. 27 2005, US Patent 6,950,364.
[114] Micron, "Micron Collaborates with Broadcom to Solve DRAM Timing Challenge, Delivering Improved Performance for Networking Customers," https://bit.ly/3RmxI0L, 2013.
[115] M. He *et al.*, "Newton: A DRAM-Maker's Accelerator-in-Memory (AiM) Architecture for Machine Learning," in *MICRO*, 2020.
[116] A. Suresh *et al.*, "Compile-Time Function Memoization," in *CC*, 2017.
[117] C. Wilcox *et al.*, "Mesa: Automatic Generation of Lookup Table Optimizations," in *IWMSE*, 2011.
[118] L. Besnard *et al.*, "A Framework for Automatic and Parameterizable Memoization," *SoftwareX*, 2019.
[119] P. Pinto and J. M. Cardoso, "A Methodology and Framework for Software Memoization of Functions," in *CF*, 2021.
[120] G. F. Oliveira *et al.*, "Employing Classification-Based Algorithms for General-Purpose Approximate Computing," in *DAC*, 2018.
[121] A. Barenghi *et al.*, "Software-Only Reverse Engineering of Physical DRAM Mappings for Rowhammer Attacks," in *IVSW*, 2018.
[122] A. Boroumand *et al.*, "LazyPIM: Efficient Support for Cache Coherence in Processing-in-Memory Architectures," in *CAL*, 2017.
[123] Xilinx, Inc., "Vitis 2020.1," https://bit.ly/3ABqwqN, 2020.
[124] Xilinx, Inc., "Vivado 2020.1," https://bit.ly/3ABQfzl, 2020.
[125] SAFARI Research Group, "pLUTo GitHub Repository," https://github.com/CMU-SAFARI/pLUTo.
[126] SAFARI Research Group, "pLUTo Zenodo Repository," https://doi.org/10.5281/zenodo.694205.
[127] R. Balasubramonian *et al.*, "CACTI 7: New Tools for Interconnect Exploration in Innovative Off-Chip Memories," in *TACO*, 2017.
[128] D. J. Bernstein, "Salsa20 Specification," https://bit.ly/3CJsnMO, 2005.
[129] B. Zoltak, "VMPC One-Way Function and Stream Cipher," in *FSE*, 2004.
[130] H. S. Warren, *Hacker's Delight*. Addison-Wesley, 2013.
[131] Q. Wang *et al.*, "AUGEM: Automatically Generate High Performance Dense Linear Algebra Kernels on X86 CPUs," in *SC*, 2013.
[132] MathWorks, "`imbinarize` Specification," https://bit.ly/3pZR9AU, 2022.
[133] Apple Inc., *Final Cut Pro User Guide*, 2022, ch. "Apply LUTs in Final Cut Pro", https://apple.co/3R7eHj4.
[134] Nanoscale Integration and Modeling (NIMO) Group, ASU, "Predictive Technology Model (PTM)," http://ptm.asu.edu/, 2012.
[135] *CT8G4SFS8 DDR4 SDRAM SODIMM*, Crucial, 2016, Rev. 04/01/16.
[136] *CS3140 Solid State Drives*, PNY Technologies, Inc., 2012, ver. 01-08-21.

[137] I. Hubara *et al.*, "Quantized Neural Networks: Training Neural Networks With Low Precision Weights and Activations," *JMLR*, 2017.
[138] S. Khoram and J. Li, "Adaptive Quantization of Neural Networks," in *ICLR*, 2018.
[139] A. Garofalo *et al.*, "PULP-NN: Accelerating Quantized Neural Networks on Parallel Ultra-Low-Power RISC-V Processors," *Philos. Trans. R. Soc. A*, 2020.
[140] Y. LeCun *et al.*, "Gradient-Based Learning Applied to Document Recognition," *Proc. IEEE*, 1998.
[141] NVIDIA, "NVIDIA Tesla P100." [Online]. Available: https://bit.ly/3CMcpld
[142] A. Akerib *et al.*, "Using Storage Cells to Perform Computation," 2012, US Patent 8,238,173.
[143] S. Angizi *et al.*, "Design and Evaluation of A Spintronic In-Memory Processing Platform for Nonvolatile Data Encryption," in *TCAD*, 2017.
[144] S. Angizi *et al.*, "Energy Efficient In-Memory Computing Platform Based on 4-Terminal Spin Hall Effect-Driven Domain Wall Motion Devices," in *GLSVLSI*, 2017.
[145] S. Angizi and D. Fan, "IMC: Energy-Efficient In-Memory Convolver for Accelerating Binarized Deep Neural Network," in *NCS*, 2017.
[146] S. Angizi *et al.*, "RIMPA: A New Reconfigurable Dual-Mode In-Memory Processing Architecture with Spin Hall Effect-Driven Domain Wall Motion Device," in *ISVLSI*, 2017.
[147] S. Angizi *et al.*, "CMP-PIM: An Energy-Efficient Comparator-Based Processing-in-Memory Neural Network Accelerator," in *DAC*, 2018.
[148] S. Angizi *et al.*, "DIMA: A Depthwise CNN In-Memory Accelerator," in *ICCAD*, 2018.
[149] S. Angizi *et al.*, "IMCE: Energy-Efficient Bit-Wise In-Memory Convolution Engine for Deep Neural Network," in *ASP-DAC*, 2018.
[150] S. Angizi *et al.*, "PIMA-Logic: A Novel Processing-in-Memory Architecture for Highly Flexible and Energy-Efficient Logic Computation," in *DAC*, 2018.
[151] S. Angizi *et al.*, "AlignS: A Processing-in-Memory Accelerator for DNA Short Read Alignment Leveraging SOT-MRAM," in *DAC*, 2019.
[152] S. Angizi and D. Fan, "Deep Neural Network Acceleration in Non-Volatile Memory: A Digital Approach," in *NANOARCH*, 2019.
[153] S. Angizi and D. Fan, "GraphiDe: A Graph Processing Accelerator Leveraging In-DRAM-Computing," in *GLSVLSI*, 2019.
[154] S. Angizi *et al.*, "GraphS: A Graph Processing Accelerator Leveraging SOT-MRAM," in *DATE*, 2019.
[155] S. Angizi *et al.*, "ParaPIM: A Parallel Processing-in-Memory Accelerator for Binary-Weight Deep Neural Networks," in *ASP-DAC*, 2019.
[156] S. Angizi *et al.*, "Exploring DNA Alignment-in-Memory Leveraging Emerging SOT-MRAM," in *GLSVLSI*, 2020.
[157] S. Angizi *et al.*, "PIM-Aligner: A Processing-in-MRAM Platform for Biological Sequence Alignment," in *DATE*, 2020.
[158] S. Angizi *et al.*, "PIM-Assembler: A Processing-in-Memory Platform for Genome Assembly," in *DAC*, 2020.
[159] D. Fan, "Low Power In-Memory Computing Platform with Four Terminal Magnetic Domain Wall Motion Devices," in *NANOARCH*, 2016.
[160] D. Fan and S. Angizi, "Energy Efficient In-Memory Binary Deep Neural Network Accelerator With Dual-Mode SOT-MRAM," in *ICCD*, 2017.
[161] D. Fan *et al.*, "Leveraging Spintronic Devices for Ultra-Low Power In-Memory Computing: Logic and Neural Network," in *MWSCAS*, 2017.
[162] D. Fan *et al.*, "In-Memory Computing With Spintronic Devices," in *ISVLSI*, 2017.
[163] P.-E. Gaillardon *et al.*, "The Programmable Logic-in-Memory (PLiM) Computer," in *DATE*, 2016.
[164] P. Gu *et al.*, "DLUX: A LUT-Based Near-Bank Accelerator for Data Center Deep Learning Training Workloads," in *TCAD*, 2020.
[165] S. Hamdioui *et al.*, "Memristor Based Computation-in-Memory Architecture for Data-Intensive Applications," in *DATE*, 2015.
[166] S. Hamdioui *et al.*, "Memristor for Computing: Myth or Reality?" in *DATE*, 2017.
[167] Z. He *et al.*, "Exploring STT-MRAM Based In-Memory Computing Paradigm with Application of Image Edge Extraction," in *ICCD*, 2017.
[168] Z. He *et al.*, "High Performance and Energy-Efficient In-Memory Computing Architecture Based on SOT-MRAM," in *NANOARCH*, 2017.
[169] Z. He *et al.*, "Leveraging Dual-Mode Magnetic Crossbar for Ultra-Low Energy In-Memory Data Encryption," in *GLSVLSI*, 2017.
[170] M. Kang *et al.*, "An Energy-Efficient VLSI Architecture for Pattern Recognition via Deep Embedding of Computation in SRAM," in *ICASSP*, 2014.
[171] S. Kvatinsky *et al.*, "Memristor-Based Imply Logic Design Procedure," in *ICCD*, 2011.
[172] S. Kvatinsky *et al.*, "Memristor-Based Material Implication (IMPLY) Logic: Design Principles and Methodologies," in *VLSI*, 2013.
[173] S. Kvatinsky *et al.*, "MAGIC–Memristor-Aided Logic," in *TCAS II*, 2014.
[174] P. V. Lea, "Apparatuses and Methods for In-Memory Operations," 2019, US Patent 10,268,389.
[175] Y. Levy *et al.*, "Logic Operations in Memory using a Memristive Akers Array," in *Microelectronics Journal*, 2014.
[176] S. Li *et al.*, "SCOPE: A Stochastic Computing Engine for DRAM-Based In-Situ Accelerator," in *MICRO*, 2018.
[177] T. A. Manning, "Apparatuses and Methods for Comparing Data Patterns in Memory," 2018, US Patent 9,934,856.
[178] F. Parveen *et al.*, "Hybrid Polymorphic Logic Gate with 5-Terminal Magnetic Domain Wall Motion Device," in *ISVLSI*, 2017.
[179] F. Parveen *et al.*, "Low Power In-Memory Computing Based on Dual-Mode SOT-MRAM," in *ISLPED*, 2017.
[180] F. Parveen *et al.*, "HielM: Highly Flexible In-Memory Computing using STT MRAM," in *ASP-DAC*, 2018.
[181] F. Parveen *et al.*, "IMCS2: Novel Device-to-Architecture Co-Design For Low-Power In-Memory Computing Platform using Coterminous Spin Switch," in *IEEE Trans. Magn.*, 2018.
[182] A. S. Rakin *et al.*, "PIM-TGAN: A Processing-in-Memory Accelerator for Ternary Generative Adversarial Networks," in *ICCD*, 2018.
[183] A. K. Ramanathan *et al.*, "Look-Up Table Based Energy Efficient Processing in Cache Support for Neural Network Acceleration," in *MICRO*, 2020.
[184] S. H. S. Rezaei *et al.*, "NoM: Network-on-Memory for Inter-Bank Data Transfer in Highly-Banked Memories," in *CAL*, 2020.
[185] Y. Tian *et al.*, "ApproxLUT: A Novel Approximate Lookup Table-Based Accelerator," in *ICCAD*, 2017.
[186] L. Wu *et al.*, "DRAM-CAM: General-Purpose Bit-Serial Exact Pattern Matching," in *CAL*, 2022.
[187] L. Xie *et al.*, "Fast Boolean Logic Mapped on Memristor Crossbar," in *ICCD*, 2015.
[188] X. Xin *et al.*, "ROC: DRAM-Based Processing with Reduced Operation Cycles," in *DAC*, 2019.
[189] L. Yang *et al.*, "A Flexible Processing-in-Memory Accelerator for Dynamic Channel-Adaptive Deep Neural Networks," in *ASP-DAC*, 2020.
[190] J. Yu *et al.*, "Memristive Devices for Computation-in-Memory," in *DATE*, 2018.
[191] J. T. Zawodny and G. E. Hush, "Apparatuses and Methods to Reverse Data Stored in Memory," 2018, US Patent 9,959,923.
[192] Y. Zha and J. Li, "Hyper-AP: Enhancing Associative Processing through a Full-Stack Optimization," in *ISCA*, 2020.
[193] H. Zhao *et al.*, "Apparatuses and Methods to Control Body Potential in Memory Operations," 2017, US Patent 9,536,618.
[194] D. U. Lee *et al.*, "A 1.2 V 8 Gb 8-Channel 128 Gb/S High-Bandwidth Memory (HBM) Stacked DRAM With Effective I/O Test Circuits," in *JSSC*, 2014.
[195] D. Lee *et al.*, "Simultaneous Multi-Layer Access: Improving 3D-Stacked Memory Bandwidth at Low Cost," *ACM TACO*, 2016.
[196] A. O. Glova *et al.*, "Near-Data Acceleration of Privacy-Preserving Biomarker Search with 3D-Stacked Memory," in *DATE*, 2019.
[197] C. Xie *et al.*, "Processing-in-Memory Enabled Graphics Processors for 3D Rendering," in *HPCA*, 2017.
[198] C. D. Kersey *et al.*, "Lightweight SIMT Core Designs for Intelligent 3D Stacked DRAM," in *MEMSYS*, 2017.
[199] Q. Guo *et al.*, "3D-Stacked Memory-Side Acceleration: Accelerator and System Design," in *WoNDP*, 2014.
[200] R. Hadidi *et al.*, "Performance Implications of NoCs on 3D-Stacked Memories: Insights from the Hybrid Memory Cube," in *ISPASS*, 2018.
[201] P. Liu *et al.*, "3D-Stacked Many-Core Architecture for Biological Sequence Analysis Problems," *IJPP*, 2017.
[202] S. H. Pugsley *et al.*, "NDC: Analyzing the Impact of 3D-Stacked Memory+Logic Devices on MapReduce Workloads," in *ISPASS*, 2014.
[203] J. P. C. de Lima *et al.*, "Design Space Exploration for PIM Architectures in 3D-Stacked Memories," in *CF*, 2018.
[204] P. C. Santos *et al.*, "Processing in 3D Memories to Speed Up Operations on Complex Data Structures," in *DATE*, 2018.
[205] Q. Zhu *et al.*, "A 3D-Stacked Logic-in-Memory Accelerator for Application-Specific Data Intensive Computing," in *3DIC*, 2013.
[206] Q. Zhu *et al.*, "Accelerating Sparse Matrix-Matrix Multiplication with 3D-Stacked Logic-in-Memory Hardware," in *HPEC*, 2013.
[207] E. Azarkhish *et al.*, "A Case for Near Memory Computation Inside the Smart Memory Cube," in *EMS*, 2016.
[208] J. Jang *et al.*, "Charon: Specialized Near-Memory Processing Architecture for Clearing Dead Objects in Memory," in *MICRO*, 2019.
[209] P.-A. Tsai *et al.*, "Adaptive Scheduling for Systems with Asymmetric Memory Hierarchies," in *MICRO*, 2018.
[210] G. F. Oliveira *et al.*, "A Generic Processing in Memory Cycle Accurate Simulator Under Hybrid Memory Cube Architecture," in *SAMOS*, 2017.
[211] J. Picorel *et al.*, "Near-Memory Address Translation," in *PACT*, 2017.
[212] R. Hadidi *et al.*, "CAIRO: A Compiler-Assisted Technique for Enabling Instruction-Level Offloading of Processing-in-Memory," *TACO*, 2017.
[213] H. Caminal *et al.*, "CAPE: A Content-Addressable Processing Engine," in *HPCA*, 2021.
[214] H. Caminal *et al.*, "Accelerating Database Analytic Query Workloads using an Associative Processor," in *ISCA*, 2022.
[215] L. Yavits *et al.*, "GIRAF: General Purpose In-Storage Resistive Associative Framework," *TPDS*, 2021.
[216] E. Garzón *et al.*, "AIDA: Associative In-Memory Deep learning Accelerator," *IEEE Micro*, 2022.
[217] R. Kaplan *et al.*, "PRINS: Processing-in-Storage Acceleration of Machine Learning," *TNANO*, 2018.
[218] A. Morad *et al.*, "Resistive GP-SIMD Processing-in-Memory," *TACO*, 2016.
[219] R. Kaplan *et al.*, "A Resistive CAM Processing-in-Storage Architecture for DNA Sequence Alignment," *IEEE Micro*, 2017.
[220] V. Patel, "DRAM CAM Memory," Aug. 8 2006, US Patent 7,088,603.
[221] A. Makosiej *et al.*, "CAM Memory Cell," Mar. 14 2019, US Patent App. 16/083,314.
[222] K. A. Batson *et al.*, "DRAM CAM Cell with Hidden Refresh," Aug. 6 2002, US Patent 6,430,073.
[223] A. Subramaniyan and R. Das, "Parallel Automata Processor," in *ISCA*, 2017.
[224] K. Wang *et al.*, "An Overview of Micron's Automata Processor," in *CODES*, 2016.

# A. Artifact Appendix

## A.1. Abstract

Our artifacts span two components of the evaluation of pLUTo: 1) SPICE simulations (Section 8.1), and 2) throughput and energy consumption estimations (Sections 8.2 to 8.8). We evaluate the correct circuit-level functionality of pLUTo cells using DRAM cell models based on Low-Power 22nm Metal Gate PTM transistors [134]. We evaluate the performance and energy consumption of pLUTo using a custom Python-based analytical timing and energy model. To aid the reproducibility of our results, we provide a Jupyter Notebook that may be used to automatically 1) run the Python-based performance and energy model, and 2) plot the ensuing results. The artifact repository is available at `https://doi.org/10.5281/zenodo.6942058` and at `https://github.com/CMU-SAFARI/pLUTo`.

## A.2. Artifact Check-List (Meta-Information)

- **Program: LTSpice, Python**
- **Metrics: voltage level, cycle count, energy consumption**
- **Experiments: DRAM cell operation, evaluated workloads**
- **Disk space requirements: 10 MB**
- **Time required to prepare the workflow: 1 hour**
- **Time required to complete the experiments: 1 hour**
- **Publicly available: yes**
- **Code license: MIT License**
- **DOI: 10.5281/zenodo.6942058**

## A.3. Description

**A.3.1. How to Access.** The repository containing the artifacts may be accessed at `https://doi.org/10.5281/zenodo.6942058` or at `https://github.com/CMU-SAFARI/pLUTo`.

**A.3.2. Software Dependencies.** The reproduction of our artifacts requires LTSpice, Python 3, and NumPy. The interactive artifact generation walkthrough further requires the Jupyter, pandas, SciPy, and Matplotlib Python libraries.

## A.4. Installation

No installation is required. The simulator may be launched either directly as a Python script or via the provided interactive Jupyter Notebook. Detailed step-by-step directions for achieving this are provided in the repository and in Appendix A.5.

## A.5. Experiment Workflow

**A.5.1. SPICE Simulation.** Please follow the following instructions to reproduce these results:

1. Download LTSpice.
2. Open the `.asc` files and run a Monte Carlo simulation.
3. Probe the bitline voltage by clicking the bitline node. A similar (results are stochastic) plot to Figure 6 will appear.

**A.5.2. Performance and Energy Model.** These results can be easily reproduced by following the step-by-step instructions in the provided `sim_walkthrough.ipynb` file. This Jupyter Notebook will automatically generate three output CSV files containing the performance and energy values for each of the 6 (DDR4- and 3DS-based pLUTo-GSA, pLUTo-BSA, and pLUTo-GMC) configurations of pLUTo.

## A.6. Evaluation and Expected Results

**A.6.1. SPICE Simulation.** Inspection of the resulting bitline voltages should reveal similar results to those shown in Figure 6 (small variations are expected due to the randomness inherent to the Monte Carlo simulation).

**A.6.2. Performance and Energy Model.** The execution of the provided simulator should result in the creation of files identical to the provided `pluto_sim/pysim_reference/pLUTo_*.csv`. In addition, when executing the provided Jupyter Notebook, plots depicting the same data as shown in Figures 7 to 12 should appear.