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* [†]*Instituto de Telecomunicações, University of Coimbra* [‡]*TU Delft* [⋆]*Intel Corporation*

ABSTRACT

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 throughput and efficiency, but supports a limited range of operations. As a result, PuM architectures cannot efficiently perform some complex operations (e.g., multiplication, division, exponentiation) without sizeable increases in chip area and design complexity.

To overcome this limitation in DRAM-based PuM architectures, we introduce ***pLUTo* (processing-using-memory with lookup table (LUT) operations)**, a DRAM-based PuM architecture that leverages the high area density of DRAM to enable the massively parallel storing and querying of lookup tables (LUTs). LUTs enable pLUTo to efficiently execute complex operations in-memory via memory reads (i.e., LUT queries) instead of relying on complex extra logic or performing long sequences of DRAM commands. pLUTo outperforms the optimized CPU and GPU baselines in performance/energy efficiency by an average of 1960×/307× and 4.2×/4× across the evaluated workloads, and by 33×/8× and 110×/80× for the LeNet-5 quantized neural network. pLUTo outperforms a state-of-the-art PiM baseline by 50×/342× in performance/energy efficiency.

1. INTRODUCTION

Processing-in-Memory (PiM) is a promising paradigm that augments a system's memory with compute capability [51, 53, 91, 92, 119] to alleviate the *data movement bottleneck* between the processing units and the memory units of computing systems [27, 35, 68, 82, 92, 98, 105, 123, 127, 130]. PiM architectures can be classified into one of two categories [53]: 1) **Processing-near-Memory (PnM)**, where computation takes place in processing elements located *near* the memory array [34, 38, 76, 81, 87, 106, 111], and 2) **Processing-using-Memory (PuM)**, where computation takes place *inside* the memory array by exploiting intrinsic analog properties of the memory device [36, 49, 57, 85, 112, 114].

In PnM architectures, data is transferred from the DRAM to nearby processors or specialized accelerators, which are either 1) part of the DRAM chip, but separate from the memory array [38, 76], e.g., near the DRAM banks, or 2) integrated into the logic layer of 3D-stacked memories [34, 87]. PnM enables the design of flexible substrates that support a diverse range of operations. However, the design and fabrication of PnM architectures is subject to significant challenges that limit their performance and scalability. For example, in near-bank PnM architectures [38, 76], the limited number of metal layers in DRAM [107, 129] impedes the fabrication of fast logic transistors [38, 55] for the near-bank processors. In 3D-stacked memories, the logic layer's limited area and thermal budget impose additional constraints. All these design and fabrication issues lead to wimpy PnM cores, which cannot exploit the entire DRAM bandwidth [40, 55, 97].

In contrast, PuM architectures enable computation *within* the memory array. The key benefit of PuM architectures is that *data does not leave the memory unit 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 propose mechanisms for the execution of bitwise operations (e.g., `AND/OR/XOR`) [49, 112, 114] and arithmetic operations [36, 37, 85, 121]. However, these proposals result in high latency and energy consumption for the execution of some complex operations (e.g., multiplication, division) [57], while other operations (e.g., exponentiation, binarization) are not supported. One approach for PuM architectures to increase the range and efficiency of supported operations is to perform computation using lookup tables (LUTs) [37, 50]. Under *LUT-based computing*, the results of complex functions are precomputed or memoized and stored in a LUT. A *LUT query* is a *memory read operation* that returns the result of a function $f(x)$ for an input value x. Several proposed PuM architectures [37, 50, 121] exploit LUT-based computing to improve the performance of specific complex operations. However, none of these proposals supports the general-purpose execution of LUT-based complex operations.

Our goal in this work is to *extend the functionality of DRAM-based PuM to provide support for complex operations via the bulk querying of LUTs.* **To this end**, we propose *pLUTo: processing-using-memory with lookup table (LUT) operations,* a DRAM-based PuM architecture that leverages LUT-based computing to perform complex operations beyond the scope of prior 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 memory array. Each LUT may contain up to as many entries as there are rows in each memory subarray (for DDR4, about 512 rows [72]). The design of pLUTo requires

the following two reasonable modifications to DRAM hardware: 1) ***row sweeping logic***, which enables the *sweeping* of memory rows, i.e., the successive activation of consecutive memory rows in a memory array; 2) ***match logic***, which identifies *matches* between the *elements* in the input row and the *index* of the currently active row in the subarray which holds the LUTs. We describe three pLUTo designs: pLUTo-GSA, pLUTo-BSA, and pLUTo-GMC. These designs achieve different trade-offs between performance, energy efficiency, and area overhead. To enable a seamless integration of pLUTo with the system, we describe the stack of changes that allows a programmer to offload their applications to pLUTo: this encompasses 1) the development of source code which includes pLUTo ISA instructions, 2) a compiler that analyzes the application's data dependency graph to determine the optimal intra- and inter-row data allocation mapping, and 3) a pLUTo controller and runtime library that monitor the allocation of pLUTo data structures and oversee the execution of pLUTo operations throughout the application's runtime.

We evaluate pLUTo for a diverse range of operations, including bitwise (`AND/OR/XOR`), arithmetic (4-bit addition / multiplication), and nonlinear functions (substitution tables, image binarization, and LUT-based color grading). We compare pLUTo to state-of-the-art processor-centric architectures (CPU [65], GPU [95], FPGA [133]) and PiM architectures (PnM [34], PuM [57, 85, 114]) across four arithmetic operations (4- and 8-bit addition, 8- and 16-bit fixed-point multiplication), seven workloads (e.g., 8-, 16-, and 32-bit CRC, Image Binarization, Image Color Grading), and one quantized neural network case study. Our evaluations show that pLUTo consistently outperforms these baselines, especially when normalizing to area overhead. We make the following **key contributions**:

- We introduce pLUTo, a DRAM-based PuM architecture that supports general-purpose LUT-based general-purpose / complex operations.
- We propose three different pLUTo designs with varying trade-offs in area overhead, energy efficiency, and performance.
- We describe the end-to-end system integration of pLUTo, which encompasses a set of ISA extensions, the specification for the compilation procedure of pLUTo source code, and dedicated hardware structures that augment DRAM with the required bookkeeping abilities.
- We evaluate pLUTo using arithmetic, bitwise logic, cryptographic, image processing, and neural network workloads and compare it to CPU [65], GPU [95], FPGA [133], and recent PiM [34, 57, 85, 114] architectures.

2. BACKGROUND

This section describes the hierarchical organization of DRAM and provides an overview of relevant prior work.

2.1 DRAM Background

A DRAM chip contains multiple memory *banks* (8 for DDR3, 16 for DDR4), and I/O circuitry. As shown in Figure 1, each bank is further divided into *subarrays*, which are two-dimensional arrays of memory cells. The DRAM subarrays in a bank share peripheral circuitry (e.g., row decoders). Each *cell* contains one capacitor and one access transistor. The capacitor encodes a single bit as stored electrical charge. The memory cell transistor connects the capacitor to the *bitline* wire. Each bitline is shared by all memory cells in a column and connects to a *sense amplifier*. The set of sense amplifiers in a subarray makes up the *local row buffer*.

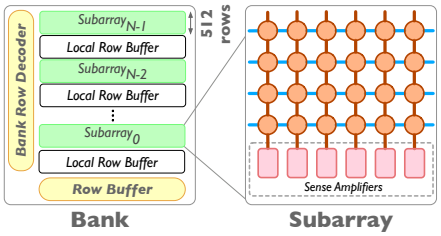

Figure 1: The internal organization of DRAM banks.

Reading and writing data in DRAM occurs over three phases: 1) Activation, 2) Reading/Writing, 3) Precharging. During Activation, the wordline of the accessed row is driven high. This turns on the row's *access transistors* and creates a path for charge sharing between each memory cell and its bitline. This process induces a voltage fluctuation ($\pm\delta$) in the bitline, which is originally set at $V_{DD}/2$. If the cell is charged, the bitline voltage becomes $V_{DD}/2+\delta$. Otherwise, it becomes $V_{DD}/2-\delta$. To read the cell value, the sense amplifiers in the local row buffer amplify the fluctuation ($\pm\delta$) recorded by the bitline during Activation. Simultaneously, the original charge level is restored to the memory cell's capacitor. After reading, data is sent to the CPU through the I/O circuitry and the memory bus. During Precharging, the access transistors are turned off, and all bitlines are restored to $V_{DD}/2$.

2.2 DRAM Extensions

pLUTo optimizes key operations by incorporating the following previous proposals for enhanced DRAM architectures.
Inter-Subarray Data Copy. The *LISA-RBM* (Row Buffer Movement) operation [32] copies the contents of one row buffer to another row buffer in a different subarray, without relying on the external memory channel. This is achieved by linking neighboring subarrays with isolation transistors.
Subarray-Level Parallelism. *MASA* [72] is a mechanism that enables subarray-level parallelism by overlapping the latency of memory accesses directed to different subarrays.
Bitwise Operations. *Ambit* [114] introduces support for bulk bitwise logic operations between rows in a DRAM subarray.
Shifting. *DRISA* [85] introduces support for intra-row shifting in DRAM. Using this mechanism, the contents of a memory row may be shifted by 1 or 8 bits at a time, with a cost of *one* Activate-Activate-Precharge (AAP) command sequence.

3. MOTIVATION

Our goal in this work is to introduce support for the in-memory execution of general-purpose operations. In particular, pLUTo is motivated by the following two key observations. First, state-of-the-art PuM architectures [37, 85, 114] provide very high throughput and energy efficiency by mitigating data movement, but only support a limited range of operations. For example, state-of-the-art in-situ DRAM-based specialized accelerators only support the execution

of basic operations (e.g., addition, bitwise logic) [85, 114] or employ long sequences of DRAM commands for more complex operations (e.g., multiplication, division) [57]. Second, LUTs can replace complex computations with cheaper lookup queries (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/lack of support for complex operations), and achieves this via the introduction of the pLUTo LUT Query operation, which enables the simultaneous querying of all the values in a given input memory row in bulk.

4. AN OVERVIEW OF pLUTo

The key contribution of pLUTo is the *pLUTo LUT Query*, a mechanism that enables the simultaneous execution of a large number of table lookups *inside* the memory array. To understand this operation, we first discuss the data layout of the involved operands in memory, as shown in Figure 2. Figure 2 (a) shows an overview of the DRAM structures required to support pLUTo's operation. Three subarrays are involved: 1) the *source subarray*, which stores input data for the pLUTo LUT Query, 2) the *pLUTo-enabled subarray*, which stores the LUTs to query, and 3) the *destination subarray*, to which the output data will be copied. In contrast to the bit-serial paradigm employed by prior PuM architectures (e.g., SIMDRAM [57]), pLUTo operates in a *bit-parallel* manner: in other words, the bits that make up the value `A` in Figure 2 (b) are stored *horizontally* (i.e., in adjacent bitlines), and the multiple copies of `A` (i.e., `{A,A,...,A}`) take up *one row* in the depicted pLUTo-enabled subarray.

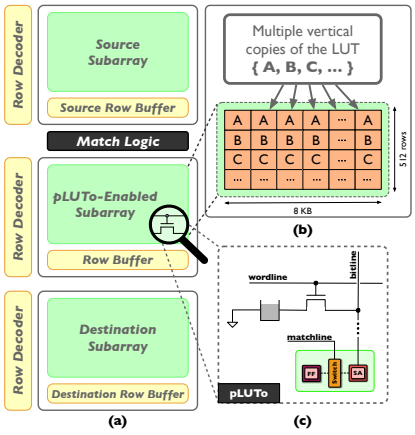

Figure 2: (a) Main components of pLUTo. (b) LUT layout inside a pLUTo subarray. (c) pLUTo cell and sense amplifier.

The pLUTo LUT Query. The pLUTo LUT Query enables all the elements stored in a *source row* to query a LUT. We illustrate the pLUTo LUT Query using a small example, which employs a small LUT to store four prime numbers $\{2,3,5,7\}$ at indices $\{0,1,2,3\}$, as shown in Figure 3 (a). Four copies of this LUT are stored in a pLUTo-enabled subarray, as shown in Figure 3 (b): each row i contains repeated copies of the element corresponding to the entry at the i-th index of the LUT. pLUTo performs a pLUTo LUT Query with an example input vector with values $\{1,0,1,3\}$, for which the output will be the prime number sequence $\{3,2,3,7\}$, in three steps. First, the memory controller loads the input values from the source subarray (not shown) into the source row buffer, as shown in Figure 3 (b). Second, the memory controller issues a *row sweep* operation (see Section 5.1.1) to quickly activate all rows that hold LUT entries in sequence; after each activation, the match logic checks for matches between the elements of the source row buffer (i.e., the input vector of LUT indices) and the index of the currently activated row in the pLUTo-enabled subarray. If there is a match (see ❶ in Figure 3 (c)), the contents of the activated row *at the matching locations* are copied to the destination row buffer (see ❷ in Figure 3 (c)). Third, the results of the pLUTo LUT Query are copied to the destination row using a LISA-RBM command (see ❸ in Figure 3 (c)), as described in Section 2.2.

5. pLUTo: A PUM ARCHITECTURE

We introduce the architecture of pLUTo, which enables the execution of basic pLUTo LUT Queries, and how we combine it with available PuM-based operations [32, 72, 85, 114].

5.1 pLUTo Architecture

To enhance a DRAM subarray with pLUTo operations, we 1) change the row decoders and row buffer, and 2) build a *match logic* unit that compares the values of each entry in the source row with the value of the currently activated row and identifies matches.

5.1.1 pLUTo-Enabled Row Decoder

The pLUTo row decoder enhances the DRAM row decoder by introducing support for the *row sweep* operation. Row sweep extends the self-refresh operation which already exists in commodity DRAM [77] to activate many consecutive rows quickly. The latency of the row sweep operation is equal to $(\texttt{tRAS} + \texttt{tRP}) \times N$. This operation enables the matching and querying of data across many rows.

5.1.2 pLUTo-Enabled Row Buffer

In DRAM, each sense amplifier in the row buffer can only read one value following the activation of a memory row. This is not sufficient for the row sweeping operation, since it is necessary to store intermediate results as they become available following each row activation. To enhance the DRAM row buffer with support for this, we connect one additional flip-flop (FF) to every sense amplifier in the row buffer via a switch (shown in Figure 2 (c)). The set of FFs constitutes a *FF buffer*. If a switch is enabled by the *matchline* signal, data in the sense amplifier is copied to the corresponding FF. Since pLUTo controls each matchline independently, data in the row buffer can be partially written to the FF buffer.

5.1.3 pLUTo-Enabled Match Logic

As shown in Figure 2 (a), we implement match logic between subarrays that consist of an array of two-input comparators: 1) the index of the currently activated row in a pLUTo-enabled subarray and 2) an element in another *source* subarray. Each comparator outputs N matchline signals, where N is the bit width of each LUT entry. Each matchline connects to the switch belonging to the corresponding FF in the

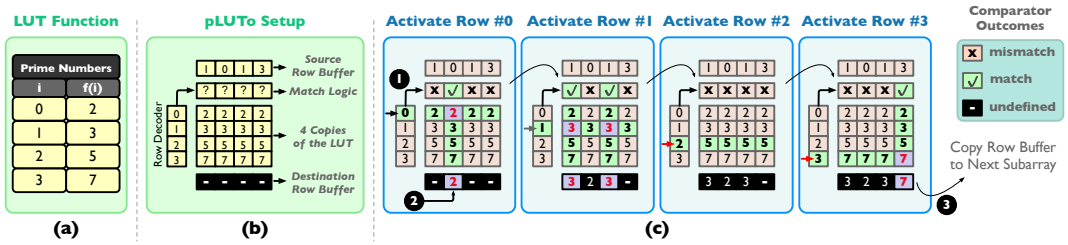

Figure 3: A pLUTo LUT Query: (a) a LUT containing the first four prime numbers, (b) the pLUTo-enabled subarray, and (c) steps of the pLUTo LUT Query. This pLUTo LUT Query returns the i-th prime number for each element in the input row.

pLUTo-enabled row buffer, as shown in Figure 2 (c). If the two inputs of the comparator match, each of the N output matchlines from the comparator are driven high. Otherwise, the matchlines are driven low.

5.2 Subarray-Level Parallelism

pLUTo operations are able to provide very high throughput, with reasonable latency. Therefore, it is important to develop strategies that allow the maximization of the number of concurrent pLUTo operations. One way to realize this objective is to leverage parallelism *across* subarrays, as described in Section 2.2, to support the simultaneous execution of multiple pLUTo LUT Queries. The achievable parallelism is limited by the `tFAW` timing parameter specified by the JEDEC DRAM Standard [66, 67], which determines the time window during which *at most four* activate commands may be issued, per DRAM rank. This constraint prevents the deterioration of the DRAM reference voltage, although DRAM manufacturers have been able to mitigate it in recent years [90].

5.3 Alternative pLUTo Architectures

While the proposed pLUTo design provides a good performance/energy/overhead trade-off, there may be situations in which a system designer would prefer to optimize for one of these three goals in isolation. In order to provide this added flexibility, we describe two variants to the pLUTo architecture with different trade-offs in throughput, area efficiency, and energy efficiency: 1) pLUTo-GSA (Gated Sense Amplifier) and 2) pLUTo-GMC (Gated Memory Cell). We refer to the design presented in Section 5.1 as pLUTo-BSA (Buffered Sense Amplifier). Table 1 qualitatively tabulates the trade-offs of each design. Each design can be used in largely the same way as pLUTo-BSA. We explain the subtleties associated with each design next.

Table 1: Comparison of pLUTo designs' core attributes.

	pLUTo-GSA	pLUTo-BSA	pLUTo-GMC
Area Efficiency	**High**	Medium	Low
Throughput	Low	Medium	**High**
Energy Efficiency	Low	Medium	**High**
Destructive Reads	Yes	**No**	**No**
Data Loading	After every use	**Once**	**Once**
Query Latency	$\texttt{tRC} \times N + \texttt{tRP}$	$(\texttt{tRAS} + \texttt{tRP}) \times N$	$\texttt{tRC} \times N + \texttt{tRP}$
Query Energy	$E_{RC} \times N + E_{RP}$	$(E_{RAS} + E_{RP}) \times N$	$E_{RC} \times N + E_{RP}$

5.3.1 pLUTo-GSA: Gated Sense Amplifier

pLUTo-GSA provides superior area efficiency over pLUTo-BSA, at the expense of reduced throughput and energy efficiency. pLUTo-GSA differs from pLUTo-BSA in its row buffer design and implementation of the row sweep operation.

pLUTo-GSA Row Buffer. Each sense amplifier in the pLUTo-GSA's row buffer includes a switch controlled by the matchline and connects the sense amplifier to the bitline (Figure 4 (a)). When the switch is enabled, the sense amplifier can sense the value on the bitline. Since pLUTo-GSA does not use an FF buffer to aggregate data, this design has a smaller area overhead than pLUTo-BSA. However, activating cells connected to bitlines that are not attached to their respective sense amplifier (i.e., the matchline is driven low by the match logic) destroys their contents. Activations performed during the *row sweep* operation are thus potentially destructive, and hence LUTs must be loaded into the pLUTo-enabled subarrays before *every* pLUTo LUT Query.

Row Sweep Operation. Since row activations are destructive in pLUTo-GSA, the row sweep operation does *not* require issuing a precharge command following each activation. Instead, it is possible to issue a single precharge command at the end of the row sweep operation. The total time required to perform a row sweep in pLUTo-GSA is therefore equal to $\texttt{tRC} \times N + \texttt{tRP}$, where `tRC` is the minimum enforced time between two consecutive activate commands, `tRP` is the precharge time, and N is the total number of rows swept. This is about half the time a row sweep operation requires in pLUTo-BSA.

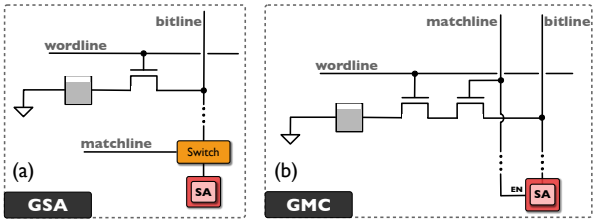

Figure 4: pLUTo-GSA (a) and pLUTo-GMC (b) designs.

5.3.2 pLUTo-GMC: Gated Memory Cell

pLUTo-GMC provides superior throughput and energy efficiency over pLUTo-BSA, at a higher area overhead. pLUTo-GMC differs from the pLUTo-BSA in its DRAM cell design, row buffer design, and implementation of the row sweep operation.

pLUTo-GMC DRAM Cell. pLUTo-GMC implements 2T1C

memory cells instead of the conventional 1T1C DRAM cell design. The matchline controls the additional transistor in each 2T1C memory cell, as shown in Figure 4 (b). This enables fine-grained control of which cells in an activated row share charge with their respective bitline and significantly minimizes the energy consumption of this design, but also requires a higher area overhead.

pLUTo-GMC Row Buffer. pLUTo-GMC places an additional switch between the sense amplifier and its enable signal, which is enabled by the matchline signal (shared by cells attached to the same bitline). This switch ensures that the sense amplifier does not drive a value on the bitline if the cell is not also attached to the bitline. This enables pLUTo-GMC to both perform back-to-back activations without a precharge and save on energy costs for enabling the sense amplifier.

Row Sweep Operation in pLUTo-GMC. Two key features of our pLUTo-GMC design enable it to perform the row sweep operation almost twice as fast as pLUTo-BSA, by using back-to-back activations without precharging. First, a sense amplifier is only enabled when there is a match in the corresponding match logic. This means that activations *only* disturb bitlines whose associated matchline signals are driven high and the remaining bitlines are *maintained* at the nominal bitline voltage level (in the precharged state). Second, since a LUT query 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 row sweep in pLUTo-GMC is equal to $\texttt{tRC} \times N + \texttt{tRP}$. In addition, due to gating transistors, pLUTo-GMC does not destroy the data in the LUTs; this translates into considerable performance gains, as there is no need to load LUT data to the subarray repeatedly.

5.4 Limitations of the Subarray Design

For a single-subarray pLUTo LUT Query, LUT entries can be increased up to the number of rows in the subarray. To query LUTs with a greater number of entries, it is possible to partition a pLUTo LUT Query across subarrays. Note that partitioning the query does not increase latency (since pLUTo operates on multiple subarrays 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 the execution of large-bit-width lookup queries. We reserve the potential exploration of alternative designs that address this limitation for future work.

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 reaps throughput and efficiency benefits by operating on data without transferring it outside the memory unit. pLUTo is not meant to replace prior PuM proposals. Instead, pLUTo addresses an important gap in the literature and inches PuM closer to supporting general-purpose operations required by a wide range of applications. Ultimately, a system with support for PnM and PuM should combine the benefits of both approaches: for example, relying on SIMDRAM [57] for multiplication, pLUTo for trigonometric functions, and near-memory general-purpose cores [34] for serial reduction. The same approach can be used to maximize performance by performing fine-grained offloading to GPUs, FPGAs, or other dedicated accelerators so that the system as a whole can perform optimally for each application segment. Performing this mapping is complex, and there is no well-established methodology in the literature to achieve it. This research direction warrants dedicated exploration, which we reserve for future work.

6. SYSTEM INTEGRATION

This section describes the system integration stack that enables pLUTo to interoperate seamlessly with the host system.

6.1 Extending the ISA: the `pluto_op`

All pLUTo LUT Queries (defined in Section 4) executed in a pLUTo-enabled DRAM have a one-to-one correspondence with a `pluto_op` instruction, defined as follows:

```
pluto_op(src, dst, lut_subarr, lut_size, lut_bitw)
```

`src` and `dst` are the physical addresses of the source and destination rows, respectively. `lut_subarr` is the physical address of the pLUTo-enabled subarray where the LUT is stored. `lut_size` is the number of LUT entries, i.e., the number of rows to sweep; 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 entries, which coincides with the width to be used by the match logic for this `pluto_op`; this value must be given in addition to `lut_size`, for example, due to the possibility that the 2^N LUT entries are zero-padded and therefore have wider bit widths than N. The `pluto_op` instruction always operates at the granularity of DRAM rows, such that operating on an S-byte input requires $\lceil \frac{S}{\textit{DRAM row size}} \rceil$ `pluto_op` instructions.

6.2 pLUTo API Code

Figure 5 shows a simple example of the implementation of the $\texttt{A} \odot \texttt{B} + \texttt{C}$ multiply-and-add operation between three vectors `A` (2-bit), `B` (2-bit), and `C` (4-bit). The reference implementation is shown in Figure 5 ⓐ. For added programmer convenience, the pLUTo API library provides pseudo-instructions (e.g., `pluto_add`, `pluto_mul`) which the compiler can automatically translate to pLUTo instructions (i.e., `pluto_op`, bit shifting, bitwise logic). The programmer may extend the pLUTo API with additional pseudo-instructions by specifying the desired mapping between inputs and outputs using a LUT. To express this program using pLUTo instructions, the programmer leverages an extensible API that describes SIMD operations as shown in Figure 5 ⓑ. The conversion of the original code requires: 1) the allocation of the associated arrays (inputs `{A,B,C}`, the temporary output `tmp`, and the final output `out`); 2) the execution of vector operations relying on the provided abstractions (for the current example, element-wise multiplication and element-wise addition).

6.3 Operation Analysis

The compiler considers data dependencies between pLUTo operands and automatically determines 1) the DRAM row where they are stored, and 2) the DRAM intra-row data layout (i.e., the alignment of input elements in the input row,

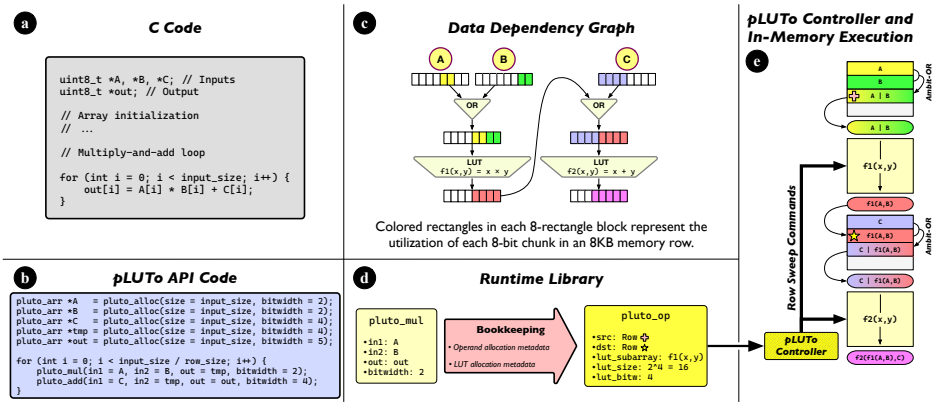

Figure 5: pLUTo's system integration stack. An example is shown for the sample C code displayed in a. All subsequent steps are shown, top-down, left-to-right: b implementation using pLUTo API instructions, c data dependency graph analysis, d the role of the runtime library, e the role of the pLUTo controller and in-memory execution.

and the position of the significant bits in each input element). An example of one such optimization process is shown in Figure 5 c, with the corresponding memory mapping depicted in Figure 5 e. In the first stage of this example, 2-bit operands `A` and `B` are stored in the same subarray to perform a LUT-based element-wise multiplication. The 2 bits of information are stored in bits `[3:2]` for every byte of `A`; likewise, each element is stored in bits `[1:0]` for every byte of `B`. The intra-row data layout of these two operands allows them to be combined simply by performing an in-DRAM bitwise `Ambit-OR` [114] operation. In the second stage, the result of the first pLUTo LUT Query is copied to the subarray where operand `C` is stored. A similar procedure then takes place again, this time combining the result of the previous operation (whose 4-bit values are stored in bits `[3:0]` of every byte in the memory row) with operand `C` (whose 4-bit values are stored in bits `[7:4]` of every byte in the memory row). The output of this 4-bit element addition is a 5-bit value, as shown in the final result of Figure 5 c. This analysis reveals three types of operations, which we now describe.

One-Shot Operations. Many common functions can be directly expressed with LUTs (e.g., arithmetic and nonlinear operations). In such cases, it is possible to automatically translate the API pseudo-instruction by 1) defining the function in question at compile time (e.g., the LUT to be used for color grading) 2) memoizing the results for all inputs, and 3) storing the ensuing results in a LUT.

Two-Operand Functions. Another common type of operation involves the combination of two inputs. Consider the element-wise product of two vectors, as shown in the first pLUTo pseudo-instruction provided in Figure 5 b. As in the one-shot scenario, it is possible to perform a one-time effort to memoize all results for any possible input and to perform the associated LUT queries. In this case, however, there is a challenge concerning ideal data mapping. Since the inputs to the LUT must be derived by combining two independent source arrays (stored in different memory rows), an additional step is required to combine them. This step can be trivially performed with a bitwise `OR` operation if, as shown in Figure 5 c, the intra-byte alignment of the data in the two input vectors allows it. Therefore, one of the roles of the pLUTo compilation framework concerns the analysis of these data dependencies to determine the ideal mapping of data in advance. This is a well-understood problem in the domain of compilers (e.g., data dependency verification during register allocation [3]), and it is, therefore, straightforward to derive mappings that optimize the in-memory movement of data for real-world-sized sequences of pLUTo operations. In situations where it is not possible to immediately obtain the ideal mapping, an additional bit shifting step is required before the execution of the `OR`-based combination step. Since pLUTo supports bit-shifting operations using the intra-lane shifting design proposed in [85], this operation can be performed *in-situ* with low overhead.

Early Termination of the Row Sweep. During a pLUTo LUT Query, if the LUTs to query collectively contain fewer entries than the number of rows in the pLUTo-enabled subarray where they are stored, it is possible to reduce the latency and energy cost of the pLUTo LUT Query operation by terminating the row sweep early. In effect, the number of rows to sweep is determined by *the bit width of each input element to be queried*. One convenient upshot of this is that pLUTo performs especially well for querying small LUTs. Each increase of 1 bit per input element causes the LUTs to contain twice as many entries and, therefore, approximately doubles the latency and energy consumption of the pLUTo LUT Query.

6.4 Data Dependency Graph: Source Data

As shown in Figure 3 and discussed in Section 4, the pLUTo LUT Query requires that 1) the source row buffer, 2) the LUT subarray, and 3) the output row buffer be adjacent in memory. In addition, all input elements for a pLUTo LUT Query must reside in the same memory row.

For this reason, it is vital to designate dedicated memory data structures to hold input/output data in a way that maximizes row utilization and, by extension, LUT utilization and operation efficiency. To this end, we define a custom memory allocation operation, the `pluto_alloc`, which allocates memory addresses while considering the above requirements. This operation reserves space in memory for an operand, and to do so receives 1) the size of this operand, in bytes, and 2)

the bitwidth of each vector element. The allocation procedure then ensures that the vectors are allocated from the beginning of a memory row and interleaved in a way that ensures the preservation of free space for the storage of the LUTs where the pLUTo LUT Query takes place.

6.5 Data Dependency Graph: LUT Data

The LUTs associated with specific application segments must be loaded to the appropriate locations in DRAM. This is a four-step process: 1) identify the degree of parallelism supported by the memory; 2) coordinating the placement of LUT data such that it is in proximity to the subarrays that store source data; 3) compute the memory addresses for these LUTs, 4) copy the LUT contents to these memory locations upon application start-up as soon as this data is available. The data to be stored in LUTs may become available in one of three ways:

Generation From Scratch. The first time a LUT is generated, all its values are computed from scratch. This procedure can be performed with lazy execution for applications that require LUTs which cannot be pre-computed.

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 [32].

Loading From Secondary Storage. LUTs can be generated at compile time and stored together with the application binary file. These LUTs can be loaded with the application code from secondary storage into the main memory at runtime using a direct memory access (DMA) operation. We evaluate this process in Section 8.5.

6.6 Runtime Library

As shown in Figure 5 ⓓ, the runtime library translates the pLUTo pseudo-instructions it receives as input to `pluto_op` instructions, which can be parsed by the pLUTo controller that issues the sequence of row sweep DRAM commands required to execute the pLUTo LUT Query. To achieve this, the runtime library holds bookkeeping records for 1) which operands, and 2) which LUTs are mapped in which memory locations. Using this information, this library is able to perform a direct translation to the `{src, dst, lut_subarray, lut_size, lut_bitwidth}` tuple required to construct each atomic `pluto_op` operation.

6.7 The pLUTo Controller

The pLUTo Controller, which is integrated with the DRAM controller, completes the pLUTo system integration stack by 1) enabling the pLUTo LUT Query operation mode for DRAM subarrays before the execution of pLUTo LUT Queries (this signals to the subarray that the match logic should be used and that a row sweep sequence is about to take place), 2) translating `pluto_op` operations to a sequence of DRAM `activate` and `precharge` commands, and 3) disabling the pLUTo LUT Query operation mode for DRAM subarrays upon completion of the operation (which ensures that these subarrays can once again be used exclusively for data storage and retrieval).

6.8 Limitations of the System Integration Stack

Address Translation. Ensuring the physical proximity of the intervening source, LUT, and destination memory addresses associated with a pLUTo LUT Query requires knowledge of their corresponding physical addresses. The use of our proposed `pluto_alloc` operation overcomes this challenge by ensuring the correct layout and placement of data in memory. The operating system can be made aware of the physical mapping of the involved memory subarrays, banks, and ranks, either 1) by employing the help of a memory controller which can provide this information as required, or 2) with an *a priori* reverse-engineering effort that allows the memory mapping scheme to be recovered (e.g., as proposed by [23]). Since the memory allocations are registered in the CPU's TLB (given that the pLUTo instructions are issued to the pLUTo controller by the CPU), the CPU cannot overwrite pLUTo-specific data since the two co-exist in the same memory address space.

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, GPU memory, accelerator memory, secondary storage). For this reason, coherence is managed implicitly by locking any CPU-side changes to data structures currently allocated to pLUTo (this is similar to the approaches employed by prior GPU programming mechanisms [30, 131]).

7. METHODOLOGY

We evaluate the three proposed designs of pLUTo (pLUTo-GSA, pLUTo-BSA, and pLUTo-GMC). Our implementations assume the parallel operation of 16 subarrays with 8 kB row buffers for DDR4 pLUTo and 512 subarrays with 256 B row buffers for 3D-stacked (HMC-based [106]) pLUTo. Despite the difference in subarray count and row buffer size between the DDR4 and 3DS configurations, they provide *the same effective parallelism at the operation level* and therefore constitute comparable design points. We compare the performance of each pLUTo configuration against four baselines: 1) a state-of-the-art CPU implementation, 2) a state-of-the-art GPU implementation, 3) a near-data processing (NDP) accelerator, 4) an FPGA.

7.1 Evaluation Frameworks

We evaluate the CPU and GPU baselines on real systems. We evaluate the FPGA baseline using high-level synthesis (HLS) implementations of the evaluated workloads, created with Vitis 2020.1 [134] and Vivado 2020.1 [135] and perform a post-synthesis simulation for a state-of-the-art Xilinx Zynq UltraScale ZCU102 FPGA [133]. For the evaluation of the NDP baseline, we simulate an HMC-based system [34] with support for bulk bitwise operations as described in [114] and shifting as described in [85]. We simulate various configurations of pLUTo on both DDR4 [67] and HMC [34] memory models using a custom-built simulator, which we plan to release under an open-source license. 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 to complete the sequence of commands and the energy consumption of the memory during the operation. We evaluate the energy consumption and area overhead of pLUTo configurations using CACTI 7 [22] 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 extrapolated 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. Unless stated otherwise, the evaluation of all pLUTo designs employs 16-subarray parallelism. Table 2 shows the main parameters that we use in our evaluations.

Table 2: System configuration for simulations.

Parameter	Configuration
Processor	Intel® Xeon Gold 5118 [65]
ISA Extensions	Intel® Streaming SIMD (SSE2, SSE4)
Last-Level Cache	64-Byte cache line, 8-way set-associative, 16.5MB
Main Memory	DDR4 2400MHz, 8GB, 1-channel, 1-rank, 4-bank groups, 4-banks per bank group, 512 rows per subarray, 8 kB per row
Main Memory Timings	17-17-17 (14.16 ns)
GPU1	NVIDIA® GeForce RTX 2080 Ti [95]
GPU2	NVIDIA® GeForce RTX 3080 Ti [96]
NDP	HMC Model [34], 1.25 GHz on-die core clock, 10 W on-die core TDP, with support for bitwise operations [114] and bit shifting [85].
FPGA	Zynq® UltraScale+ MPSoC ZCU102 [133]
pLUTo	16-subarray parallelism unless stated otherwise

7.2 Workloads

Table 3 shows the names and characteristics of the workloads we analyze. The authors chose these workloads because they exemplify general-purpose, real-world functions that cannot be efficiently executed by previous Processing-using-Memory architectures and can be replaced with LUT-based operations. These operations include 1) substitution tables, as required by cipher algorithms such as Salsa20 and VMPC, 2) polynomial division, required by the CRC algorithm, and 3) image binarization, which in prior bulk bitwise accelerator proposals require several bit masking steps and large sequences of bitwise logical operations. All workloads used 100MB of input data except when stated otherwise in the table below. We determined this size to represent modern applications while simultaneously highlighting the limitations of the existing memory hierarchy by limiting the cache utilization. For this input size, the relative time spent loading LUTs relative to computation is about 2.5%.

8. EVALUATION

In this section, we evaluate pLUTo's correct operation (Section 8.1), performance (Section 8.2), energy efficiency (Section 8.3), and area overhead (Section 8.4). We also carry out performance sensitivity analyses to understand the cost of loading LUTs (Section 8.5). Finally, we discuss how pLUTo compares to prior works (Section 8.8).

8.1 SPICE Simulation

We performed circuit-level SPICE simulations using DRAM cell models based on Low-Power 22nm Metal Gate PTM transistors [94] to verify the correct operation of pLUTo. Our results show that the proposed changes *do not introduce* errors in DRAM operation. The observed disturbances to the final voltage in the bitline after the activation of a row correspond to 0.9% of the reference value.

We draw two key observations. First, the bitline achieves the rows are correctly activated in all cases, without substantially affecting the required activation time. Second, as expected, the activation procedure is noisiest for pLUTo-GSA. However, correct behavior is observed even in this case.

Table 3: Evaluated workloads.

Name	Parameters
Vector Addition, LUT-based [125]	Element width: 4 bits
Vector Point-Wise Multiplication [125]	Q Format: Q1.7, Q1.15
Bitwise Logic (NOT, AND, OR, XOR, XNOR)	# LUT entries: 4
Bit Counting	BC-4: 4 bits, 16-entry LUT; BC-8: 8 bits, 256-entry LUT
CRC-8/16/32 [128]	Packet size: 128 bytes
Salsa20 [25], VMPC [144]	Packet size: 512 bytes
Image Binarization	3-channel 8-bit image, 936000 pixels; threshold: 50%
Color Grading	One 3-channel 8-bit image, 936000 pixels; 8-bit to 8-bit

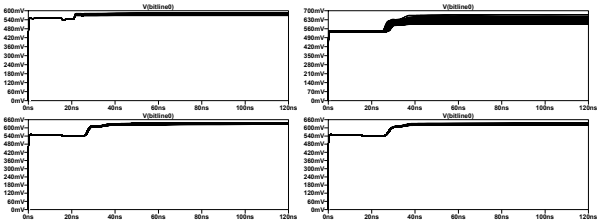

Figure 6: SPICE Simulations of the bitline voltage in response to a row activation, for a baseline DRAM and for the three designs of pLUTo. The plots depict a Monte Carlo simulation of 100 runs, where the process variation is assumed to be 5%. Clockwise from the top left: Unmodified DRAM, pLUTo-GSA, pLUTo-GMC, pLUTo-BSA.

8.2 Performance

Figure 7 plots absolute speedups relative to the CPU, GPU, and NDP baselines for the considered pLUTo design points. We make three key observations. First, we observe that, for the DDR4 (3DS) implementation, the pLUTo-BSA/pLUTo-GMC designs outperform the GPU baseline by 1.53× (2.12×) / 18.38× (25.52×), and the NDP baseline by 3.02× (4.20×) / 36.34× (50.52×), respectively; pLUTo-GSA (both DDR4- and 3DS-based) performs comparably to the GPU and outperforms the NDP baseline. Second, we observe that the 3DS-based pLUTo consistently outperforms its DDR4 counterparts due to HMC's lower value of `tRC` and faster overall LUT querying times. Third, we observe that 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 (2D pLUTo) or in the logic layer (pLUTo-3DS). Nevertheless, the acceleration of the parallel portion of the CRC workloads still allows nearly all pLUTo design points to achieve performance comparable to that of the GPU baseline.

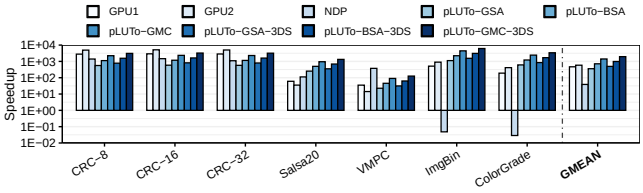

Figure 7: Speedup of GPU, NDP, and pLUTo relative to the baseline CPU. The y-axis uses a logarithmic scale; higher is better.

8.2.1 *Performance per Area*

Figure 8 shows the speedup per unit area of each pLUTo configuration relative to the CPU, NDP, and GPU baselines. Area values for the CPU/GPU baselines refer to each device's total chip area. In contrast, area values for the NDP baselines refer to the entire area available in the logic layer of 3D-stacked memories (4.4 mm^2 per vault in HMC-like devices [27, 28, 34]). We make three key observations. First, all pLUTo designs outperform the GPU and NDP. This improvement is considerably more significant than the one observed in Figure 7 when normalized to the area of each design. Second, the performance improvement of pLUTo-3DS is less noticeable in this plot. This leads to the observation that the performance of pLUTo is roughly proportional to the area of the memory technology in which it is implemented, at least for the well-established DDR4 and HMC memory technologies considered in our evaluation. Third, we especially observe improvements for the most memory-intensive workloads, which for this set are Salsa20 and VMPC. This demonstrates the benefit of performing computation in-memory for data-intensive workloads.

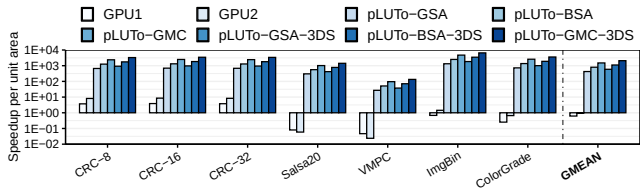

Figure 8: Normalized speedup of GPU, NDP, and pLUTo relative to CPU. The y-axis uses a logarithmic scale; higher is better.

8.2.2 *Comparison with FPGA*

FPGAs often provide superior performance and energy efficiency. Since they implement logic functionality using LUTs, as pLUTo does, it is important to compare the performance of the two approaches. Figure 9 shows the performance of pLUTo compared to the baseline FPGA implementation. We observe that pLUTo can outperform the baseline in this scenario for all considered workloads. 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).

8.3 Energy Efficiency

Figure 10 shows the energy efficiency achieved when executing each of the evaluated workloads on different pLUTo configurations. We report the energy consumption for the GPU and NDP baselines and all proposed 2D and 3D pLUTo designs. We make two key observations. First, the average energy consumption of the GPU implementation is higher than pLUTo but lower than pLUTo-3DS. This is because HMC memory comprises smaller rows, which increases the overall number of activations required and, consequently, the overall energy consumption. Second, we observe that pLUTo can outperform the GPU for most simple operations (e.g., ImgBin) but falls short as operation complexity increases (e.g., CRC). This trend is consistent with our observations from earlier in this section regarding workload performance.

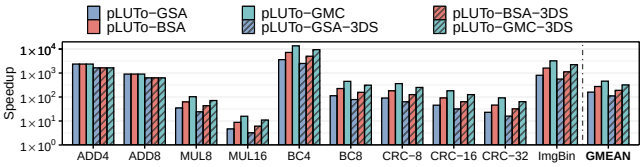

Figure 9: pLUTo speedup relative to FPGA baseline.

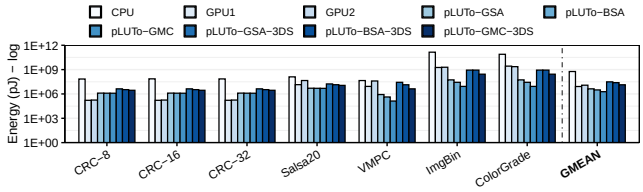

Figure 10: Energy efficiency of GPU, NDP, and pLUTo compared to the baseline CPU. The y-axis uses a logarithmic scale; higher is better.

8.4 Area Overhead

Table 4 shows a breakdown of the estimated area overheads per DRAM component. These estimates are derived from transistor counts and the area models of CACTI 7 [22].

pLUTo-BSA. We estimate that the sense amplifier switch and FF (shown in Figure 2 (c)) incur a 60% area overhead for the sense amplifiers. The total overhead of pLUTo-BSA is 16.7% of the DRAM chip area.

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

pLUTo-GSA. The estimated area overhead of the switch (shown in Figure 4 (b)) 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.

Table 4: Area breakdown of a commodity DRAM chip and of the three pLUTo designs (GSA, BSA, GMC).

		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 5.1.3, which are identical in for all three designs, are shown separately. The Row Decoder overhead includes that of the logic required for the row sweep operation. The only pLUTo design that requires modifications to the DRAM cell design is pLUTo-GMC. Its overhead is indicated in the DRAM Cell row of Table 4. In the baseline system, the memory cell transistors take up approximately 15.1 mm^2. This overhead doubles in the implementation of the 2T1C cell used by pLUTo-GMC.

8.5 LUT Loading Overhead

Figure 11 shows the fraction of total computation time spent loading LUT data (y-axis), relative to the total volume of data that is queried (x-axis). For example, to query around 20 MB worth of data from scratch (x-axis), roughly 10% of the execution time would be spent loading the data into memory (y-axis), and 90% of the execution time would be spent performing pLUTo LUT Queries. We make two key observations. First, we observe that it is sufficient to process 1.9 MB of data (◆ in Figure 11) for the LUT loading time to equal the LUT query time. Second, we observe that, as the volume of data to be processed increases, the fraction of time spent loading the LUTs into memory quickly decreases. For example, for 120 MB (▲ in Figure 11), the fraction of time spent with LUT loading is about 2%.

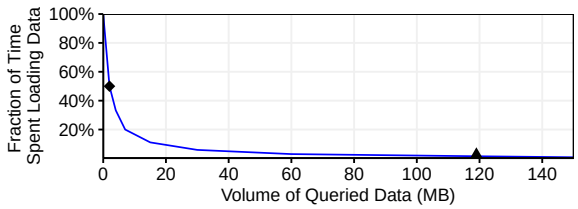

Figure 11: Fraction of time spent setting up LUTs as a function of the volume of data to be queried.

8.6 Impact of `tFAW` on Performance

The `tFAW` timing parameter is another limiter of the activation rate in a DRAM chip to meet power constraints. Since the activation operation is central to pLUTo, it is important to evaluate the impact of this parameter on performance. While we discussed methods for reducing this constraint (in Section 5.2), we consider how `tFAW` would affect pLUTo's performance in the case that `tFAW` cannot be relaxed. Figure 12 shows the effects of varying `tFAW` (between 0% and 100% of its nominal value) in a commodity DDR4 memory module on the performance of a single pLUTo LUT Query, across our examined workloads. We make two key observations. First, the performance loss is around 10% for `tFAW` = 50%, and around 20% for nominal `tFAW` (i.e., requiring no relaxation of power constraints). Even accounting for this performance penalty, pLUTo would still outperform the CPU baseline, and rival the GPU baseline. Second, we note that the performance penalties are very similar across all of the considered workloads, for the same value of `tFAW`. Despite the limited impact of `tFAW` on pLUTo, the use of more powerful charge pumps could further relax power constraints and therefore reduce the required value of `tFAW` in pLUTo-capable DRAMs, bringing the actual performance results closer to the ones reported in Figures 7 and 8.

8.7 Throughput with Subarray Parallelism

To present a more realistic internal bandwidth when using pLUTo, we evaluate the three pLUTo designs (i.e., pLUTo-GSA, pLUTo-BSA, pLUTo-GMC) with varying degrees of subarray-level parallelism for both DDR4 and 3D-stacked memory. Figure 13 plots the speedups (averaged across all evaluated workloads) of each configuration against the baseline CPU. We make three observations. First, due to the different row size in DDR4 (i.e., 8KB) and 3D memory (i.e., 256B), DDR4 provides higher speedup than 3D memory when utilizing the same degree of subarray-level parallelism. As discussed in Section 7, a fair direct comparison only can be made between the configuration pairs {DDR4-16, HMC-512} and {DDR4-256, HMC-8192}. Second, performance scaling is very close to proportional to the number of subarrays operating in parallel in both cases. The reason why the maximum theoretical performance cannot be obtained in practice is that there is an additional overhead associated with in-memory data movement when computation is partitioned across multiple subarrays. Third, performance is approximately constant when normalized to area across all pLUTo designs. This validates the scalability of pLUTo to operate in as many subarrays in parallel as the memory technology supports.

8.8 Comparison With Prior Works

As summarily demonstrated in Table 5 and discussed in Section 3, prior PuM architectures (e.g., [37, 85, 114]) 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 [114] using bit-serial algorithms. Nevertheless, we argue that the additional flexibility afforded by pLUTo's native support for LUT operations allows it to outperform prior PuM architectures in meaningful and substantive ways. We substantiate this claim with Table 5, which shows the time for each operation of interest, under each of the architectures mentioned above. In each case, we assume the use of ideal data layouts and report the best-case *achievable* performance for each design. For example, in the case of bitwise operations between input sets `A` ('a_1a_2...') and `B` ('b_1b_2...'), under the LUT-based paradigm all input operands are shuffled ('$a_1b_1a_2b_2$...'), and for all prior PuM designs input sets `A` and `B` are ideally stored in two separate memory rows. We note that changes at the

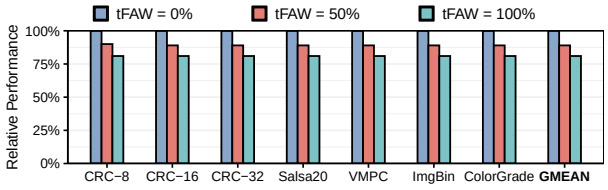

Figure 12: The impact of different values of tFAW on pLUTo's performance.

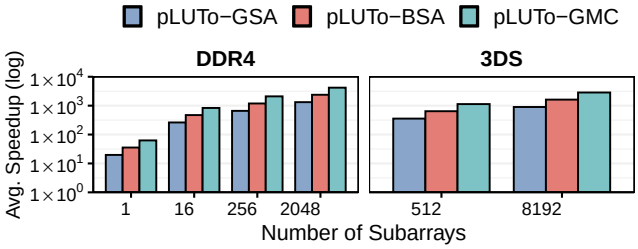

Figure 13: Geometric mean speedup of pLUTo over CPU, for varying degrees of subarray-level parallelism.

system level are required to support the ideal data mapping schemes that maximize pLUTo's performance. To maximize fairness, the memory capacity for each design is such that the area overheads for all designs remain in a narrow range, similar to the overhead of commodity DRAM devices.

Table 5: Comparison of operations supported by pLUTo vs. prior PuM. Performance per area and energy efficiency values are normalized to pLUTo-BSA with 4-subarray parallelism.

	Ambit [114]	**SIMDRAM** [57]	**LAcc** [37]	**DRISA** [85]	**pLUTo**
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
Binarization (*ns*)	-	-	-	-	1920.0
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 conclusions from Table 5. First, we observe that due to their complexity some operations (e.g., binarization, exponentiation) cannot be implemented in a time-efficient manner using any prior designs. 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, we observe that pLUTo performs bitwise logic operations at rates that match or exceed those of all prior works. This result is consequential since it shows that, with proper data alignment, LUT-based computing outperforms even specialized designs. Third, we observe that pLUTo consistently outperforms all three 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 in the case of 4-bit addition.

8.9 Scalability Analysis

This section analyzes the scalability of pLUTo's LUT query operation. Our goals are 1) to fundamentally understand the performance of the pLUTo LUT Query, and 2) to study the suitability of different PiM architectures for a commonly used operation. We adopt a two-pronged approach to achieve this goal. First, in Figure 14(a), we show an analysis of the theoretical throughput and energy consumption scaling of the three proposed pLUTo designs (following the equations and the timing parameters given in Table 1) with that of our baseline 2D DRAM device. Second, in Figure 14(b), we compare the energy efficiency (in OP/J) of a representative operation (multiplication), for three systems: (i) pLUTo-BSA, (ii) a bit-serial PuM mechanism (SIMDRAM [57]), and (iii) our baseline NDP device, while varying the element size of the operands involved in the multiplication.

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 ($N < 8$). This limitation results primarily from the exponential growth in the number of activations that need to be performed for every linear increase in input bit widths. Second, we observe that the pLUTo provides higher energy efficiency than both alternative PIM solutions (SIMDRAM and NDP) for low-precision multiplications (element size < 8 bits). Executing multiplications in pLUTo leads to better energy efficiency than in SIMDRAM for all evaluated element sizes. This happens because executing bit-serial multiplications incurs a quadratic number of DRAM activations [57]. We conclude that pLUTo is well-suited to perform low-bit-width LUT queries, which can be adopted alongside alternative solutions (e.g., NDP) to take full advantage of the underlying DRAM substrate.

9. CASE STUDY: BNN

As shown in Section 8, pLUTo is especially well-suited for executing limited-precision operations efficiently since these operations can be carried out using small LUTs. Building on this observation, this section validates the applicability of pLUTo for quantized neural networks, an emerging machine learning application. We evaluate a quantized version of the LeNet-5 network to classify digits from the MNIST dataset as proof-of-concept. The inference times for CPU, GPU, and pLUTo are shown in Table 6. For this evaluation, the CPU is unchanged from Table 2; the GPU is a server-grade NVIDIA P100, commonly used for machine learning applications, with 16GB of dedicated memory. We make two key observations. First, pLUTo-16 outperforms both the CPU (10×, 30× for 1-bit, 4-bit), the GPU (2×, 7×) and the FPGA (6×, 19×) in inference time. This is because pLUTo

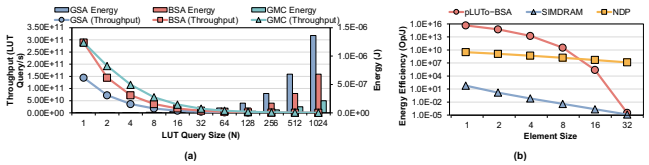

Figure 14: Scalability analysis for pLUTo's LUT query operations: (a) shows the LUT query throughput for the three pLUTo designs while varying the LUT query size; (b) compares the energy efficiency (in OP/J) of pLUTo-BSA against a prior PuM mechanism (SIMDRAM [57]) and the NDP baseline.

operations on reduced bit width data are especially efficient since they can be performed in place as a short sequence of DRAM commands. Second, pLUTo-16 also achieves considerable energy savings over both the CPU (110×, 109×), the GPU (80×, 81×) and the FPGA (15×, 16×), for both 1- and 4-bit precision. This reduction can be attributed to the overall mitigation in data movement since most operations are performed in place. These results strengthen the case for using pLUTo in heavily energy-constrained devices, such as IoT and other edge devices.

Table 6: LeNet-5 inference times (in μs) and energy (in mJ) for CPU, GPU, FPGA and pLUTo.

Bit Width	Accuracy [70]	CPU		GPU		FPGA		pLUTo-BSA-16	
		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 to propose a mechanism to enable the efficient bulk querying of LUTs inside DRAM to enable the PuM-based execution of complex operations. In this section, we describe relevant prior works.

Processing-using-Memory (PuM). Many prior works propose various forms of compute-capable memory [1, 2, 4–21, 26, 28, 32, 33, 36, 37, 39, 41–49, 51, 52, 54, 56–64, 69, 72–75, 78–80, 83–86, 89, 93, 99–102, 108–110, 112–118, 122, 126, 132, 136–141, 143]. All these approaches provide significant performance and energy improvements, but focus only on a reduced set of operations, e.g. data movement [32, 113], bulk bitwise operations [1, 86, 114, 136] or neural network acceleration [36, 37, 41, 85]. By combining the in-memory pLUTo LUT Query with the fast and efficient bitwise logic and shifting operations enabled by these prior works, pLUTo supports a much greater range of operations. While pPIM [121] and LAcc [37], 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 throughput and efficiency.) In contrast to pPIM and LAcc, DRAF [50] employs a DRAM-based FPGA-like LUT-based computing paradigm. DRAF outperforms FPGA execution in area and energy efficiency but lags in throughput and latency. In contrast, pLUTo enables high-throughput LUT queries without compromising energy efficiency. Furthermore, pLUTo subarrays can operate as a conventional storage medium when not partaking in a pLUTo LUT Query, so there is no compromise in DRAM capacity, aside from the area increase for the modifications required by pLUTo.

Processing-near-Memory (PnM). 3D-stacked memories [2, 29, 71, 104, 142] enable the stacking of memory layers atop a logic layer. 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.

Miscellaneous Processing-in-Memory (PiM). Recent works have enabled PiM via the efficient querying or searching of data in memory. **Content-Addressable Memories** (CAMs) (e.g., CAPE [31]) return the address of matched data. given an input query, and therefore can be used to enable a form of LUT-based computing. Most CAMs are SRAM-based and provide low area density compared to DRAM-based memories. DRAM-based CAMs also exist [24, 88, 103], but require a greater number of transistors per cell than pLUTo-GMC, our most oversized design. Another approach entails the development of specialized **Automata Processors** (APs) [120, 124]. These processors enable unrivaled querying of vast, unstructured data sources, but do not provide the high-throughput lookup mechanism enabled by the pLUTo LUT Query and therefore are not well-suited for the offloading of complex functions by memoization.

11. CONCLUSION

We introduced pLUTo, a DRAM-based PuM architecture that enables the storage and bulk query of lookup tables. We build pLUTo based on the key observation that enabling in-DRAM LUT-based provides a flexible substrate to execute complex operations within DRAM efficiently. We describe the hardware design of three different pLUTo architectures, each one targeting different performance metrics (i.e., throughput, energy, and area) and the necessary system integration support to enable the execution of in-DRAM pLUTo operations. Our evaluations show that pLUTo outperforms the baseline CPU-, GPU-, FPGA-, PnM- and PuM-based systems. We believe the adoption of pLUTo in real systems may provide significant performance and energy improvements in applications designed to take advantage of it.

REFERENCES

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

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

[3] A. V. Aho, R. Sethi, and J. D. Ullman, *Compilers, Principles, Techniques*. Addison Wesley, 1986.

[4] A. Akerib, A. Oren, E. Ehrman, and M. Meyassed, "Using Storage Cells To Perform Computation," 2012, US Patent 8,238,173.

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

[6] S. Angizi and D. Fan, "IMC: Energy-Efficient In-Memory Convolver For Accelerating Binarized Deep Neural Network," in *NCS*, 2017.

[7] S. Angizi and D. Fan, "Deep Neural Network Acceleration In Non-Volatile Memory: A Digital Approach," in *NANOARCH*, 2019.

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

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

[10] S. Angizi, Z. He, N. Bagherzadeh, and D. Fan, "Design And Evaluation Of A Spintronic In-Memory Processing Platform For Nonvolatile Data Encryption," in *IEEE TCAD*, 2017.

[11] S. Angizi, Z. He, and D. Fan, "Energy Efficient In-Memory Computing Platform Based On 4-Terminal Spin Hall Effect-Driven Domain Wall Motion Devices," in *GLSVLSI*, 2017.

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

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

[14] S. Angizi, Z. He, and D. Fan, "ParaPIM: A Parallel Processing-In-Memory Accelerator For Binary-Weight Deep Neural

Networks," in *ASP-DAC*, 2019.

[15] S. Angizi, Z. He, F. Parveen, and D. Fan, "Rimpa: A New Reconfigurable Dual-Mode In-Memory Processing Architecture With Spin Hall Effect-Driven Domain Wall Motion Device," in *ISVLSI*, 2017.

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

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

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

[19] S. Angizi, J. Sun, W. Zhang, and D. Fan, "GraphS: A Graph Processing Accelerator Leveraging SOT-MRAM," in *DATE*, 2019.

[20] S. Angizi, J. Sun, W. Zhang, and D. Fan, "PIM-Aligner: A Processing-In-Mram Platform For Biological Sequence Alignment," in *DATE*, 2020.

[21] S. Angizi, W. Zhang, and D. Fan, "Exploring DNA Alignment-In-Memory Leveraging Emerging SOT-MRAM," in *GLSVLSI*, 2020.

[22] R. Balasubramonian, A. B. Kahng, N. Muralimanohar, A. Shafiee, and V. Srinivas, "Cacti 7: New Tools For Interconnect Exploration In Innovative Off-Chip Memories," in *TACO*, 2017.

[23] A. Barenghi, L. Breveglieri, N. Izzo, and G. Pelosi, "Software-Only Reverse Engineering Of Physical DRAM Mappings For Rowhammer Attacks," in *IVSW*. IEEE, 2018.

[24] K. A. Batson, R. E. Busch, and G. S. Koch, "DRAM CAM cell with hidden refresh," Aug. 6 2002, US Patent 6,430,073.

[25] D. J. Bernstein, "Salsa20 Specification," http://www.ecrypt.eu.org/stream/salsa20pf.html, 2005.

[26] D. Bhattacharjee, R. Devadoss, and A. Chattopadhyay, "Revamp: Reram Based Vliw Architecture For In-Memory Computing," in *DATE*, 2017.

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

[28] A. Boroumand, S. Ghose, M. Patel, H. Hassan, B. Lucia, R. Ausavarungnirun, K. Hsieh, N. Hajinazar, K. T. Malladi, H. Zheng *et al.*, "Conda: Efficient Cache Coherence Support For Near-Data Accelerators," in *ISCA*, 2019.

[29] A. Boroumand, S. Ghose, M. Patel, H. Hassan, B. Lucia, K. Hsieh, K. T. Malladi, H. Zheng, and O. Mutlu, "LazyPIM: An Efficient Cache Coherence Mechanism For Processing-In-Memory," in *CAL*, 2016.

[30] A. R. Brodtkorb, T. R. Hagen, and M. L. Sætra, "Graphics Processing Unit (GPU) Programming Strategies And Trends In GPU Computing," *Journal of Parallel and Distributed Computing*, vol. 73, no. 1, pp. 4–13, 2013.

[31] H. Caminal, K. Yang, S. Srinivasa, A. K. Ramanathan, K. Al-Hawaj, T. Wu, V. Narayanan, C. Batten, and J. F. Martínez, "CAPE: A Content-Addressable Processing Engine," in *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 2021, pp. 557–569.

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

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

[34] H. M. C. Consortium *et al.*, "Hybrid Memory Cube Specification 2.1," *Retrieved from micron.com*, 2014.

[35] B. Dally, "The Path To Exascale Computing," http://images.nvidia.com/events/sc15/pdfs/SC5102-path-exascale-computing.pdf, 2015.

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

[37] Q. Deng, Y. Zhang, M. Zhang, and J. Yang, "LAcc: Exploiting Lookup Table-Based Fast And Accurate Vector Multiplication In DRAM-Based CNN Accelerator," in *DAC*, 2019.

[38] F. Devaux, "The True Processing In Memory Accelerator," in *HC*, 2019.

[39] J. Draper, J. Chame, M. Hall, C. Steele, T. Barrett, J. LaCoss, J. Granacki, J. Shin, C. Chen, C. W. Kang *et al.*, "The Architecture Of The Diva Processing-In-Memory Chip," in *ICS*, 2002.

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

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

[42] D. Fan, "Low Power In-Memory Computing Platform With Four Terminal Magnetic Domain Wall Motion Devices," in *NANOARCH*, 2016.

[43] D. Fan and S. Angizi, "Energy Efficient In-Memory Binary Deep Neural Network Accelerator With Dual-Mode SOT-MRAM," in *ICCD*, 2017.

[44] D. Fan, S. Angizi, and Z. He, "In-Memory Computing With Spintronic Devices," in *ISVLSI*, 2017.

[45] D. Fan, Z. He, and S. Angizi, "Leveraging Spintronic Devices For Ultra-Low Power In-Memory Computing: Logic And Neural Network," in *MWSCAS*, 2017.

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

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

[48] D. Gao, T. Shen, and C. Zhuo, "A Design Framework For Processing-In-Memory Accelerator," in *SLIP*, 2018.

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

[50] M. Gao, C. Delimitrou, D. Niu, K. T. Malladi, H. Zheng, B. Brennan, and C. Kozyrakis, "Draf: A Low-Power DRAM-Based Reconfigurable Acceleration Fabric," in *ISCA*, 2016.

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

[52] S. Ghose, K. Hsieh, A. Boroumand, R. Ausavarungnirun, and O. Mutlu, "Enabling The Adoption Of Processing-In-Memory: Challenges, Mechanisms, Future Research Directions," in *Beyond-CMOS Technologies for Next Generation Computer Design*, 2018.

[53] S. Ghose, K. Hsieh, A. Boroumand, R. Ausavarungnirun, and O. Mutlu, "The Processing-In-Memory Paradigm: Mechanisms To Enable Adoption," in *Beyond-CMOS Technologies for Next Generation Computer Design*, 2019.

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

[55] J. Gómez-Luna, I. E. Hajj, I. Fernandez, C. Giannoula, G. F. Oliveira, and O. Mutlu, "Benchmarking a New Paradigm: An Experimental Analysis of a Real Processing-in-Memory Architecture," *arXiv preprint arXiv:2105.03814*, 2021.

[56] P. Gu, X. Xie, S. Li, D. Niu, H. Zheng, K. T. Malladi, and Y. Xie, "DLUX: A LUT-Based Near-Bank Accelerator For Data Center Deep Learning Training Workloads," in *TCAD*, 2020.

[57] N. Hajinazar, G. F. Oliveira, S. Gregorio, J. D. Ferreira, N. M. Ghiasi, M. Patel, M. Alser, S. Ghose, J. Gómez-Luna, and O. Mutlu, "SIMDRAM: A Framework For Bit-Serial Simd Processing Using DRAM," in *HPCA*, 2021.

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

[59] S. Hamdioui, L. Xie, H. A. Du Nguyen, M. Taouil, K. Bertels, H. Corporaal, H. Jiao, F. Catthoor, D. Wouters, L. Eike *et al.*, "Memristor Based Computation-In-Memory Architecture For Data-Intensive Applications," in *DATE*, 2015.

[60] Z. He, S. Angizi, and D. Fan, "Exploring Stt-Mram Based In-Memory Computing Paradigm With Application Of Image Edge Extraction," in *ICCD*, 2017.

[61] Z. He, S. Angizi, F. Parveen, and D. Fan, "High Performance And Energy-Efficient In-Memory Computing Architecture Based On SOT-MRAM," in *NANOARCH*, 2017.

[62] Z. He, S. Angizi, F. Parveen, and D. Fan, "Leveraging Dual-Mode Magnetic Crossbar For Ultra-Low Energy In-Memory Data Encryption," in *GLSVLSI*, 2017.

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

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

[65] Intel, "Intel® Xeon® Gold 5118 Processor Specifications." [Online]. Available: https://www.intel.com/content/www/us/en/products/sku/120473/intel-xeon-gold-5118-processor-16-5m-cache-2-30-ghz/specifications.html

[66] JEDEC, "DDR3 SDRAM Standard, Jesd79-3d," https://www.jedec.org/standards-documents/docs/jesd-79-3d, 2012.

[67] JEDEC, "DDR4 SDRAM Standard, Jesd79-4b," https://www.jedec.org/standards-documents/docs/jesd79-4a, 2017.

[68] S. Kanev, J. P. Darago, K. Hazelwood, P. Ranganathan, T. Moseley, G.-Y. Wei, and D. Brooks, "Profiling A Warehouse-Scale Computer," in *ISCA*, 2015.

[69] M. Kang, M.-S. Keel, N. R. Shanbhag, S. Eilert, and K. Curewitz, "An Energy-Efficient Vlsi Architecture For Pattern Recognition Via Deep Embedding Of Computation In Sram," in *ICASSP*, 2014.

[70] S. Khoram and J. Li, "Adaptive Quantization Of Neural Networks," in *ICLR*, 2018.

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

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

[73] S. Kvatinsky, D. Belousov, S. Liman, G. Satat, N. Wald, E. G. Friedman, A. Kolodny, and U. C. Weiser, "Magic—Memristor-Aided Logic," in *TCAS II*, 2014.

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

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

[76] Y.-C. Kwon, S. H. Lee, J. Lee, S.-H. Kwon, J. M. Ryu, J.-P. Son, O. Seongil, H.-S. Yu, H. Lee, S. Y. Kim *et al.*, "A 20nm 6gb Function-In-Memory DRAM, Based On Hbm2 With A 1.2 Tflops Programmable Computing Unit Using Bank-Level Parallelism, For Machine Learning Applications," in *ISSCC*, 2021.

[77] P. S. Lazar and S. C. Oh, "DRAM With Total Self Refresh And Control Circuit," 2004, US Patent 6,741,515.

[78] P. V. Lea, "Apparatuses And Methods For In-Memory Operations," 2019, US Patent 10,268,389.

[79] B. C. Lee, E. Ipek, O. Mutlu, and D. Burger, "Phase Change Memory Architecture And The Quest For Scalability," in *CACM*, 2010.

[80] B. C. Lee, P. Zhou, J. Yang, Y. Zhang, B. Zhao, E. Ipek, O. Mutlu, and D. Burger, "Phase-Change Technology And The Future Of Main Memory," in *MICRO*, 2010.

[81] D. U. Lee, K. W. Kim, K. W. Kim, H. Kim, J. Y. Kim, Y. J. Park, J. H. Kim, D. S. Kim, H. B. Park, J. W. Shin *et al.*, "A 1.2v 8gb 8-Channel 128 Gb/S High-Bandwidth Memory (Hbm) Stacked DRAM With Effective Microbump I/O Test Methods Using 29nm Process And Tsv," in *ISSCC*, 2014.

[82] C. Lefurgy, K. Rajamani, F. Rawson, W. Felter, M. Kistler, and T. W. Keller, "Energy Management For Commercial Servers," in *Computer*, 2003.

[83] Y. Levy, J. Bruck, Y. Cassuto, E. G. Friedman, A. Kolodny, E. Yaakobi, and S. Kvatinsky, "Logic Operations In Memory Using A Memristive Akers Array," in *Microelectronics Journal*, 2014.

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

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

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

[87] G. H. Loh, "3D-Stacked Memory Architectures For Multi-Core Processors," in *ISCA*, 2008.

[88] A. Makosiej, A. Amara, C. Anghel, and N. Gupta, "CAM Memory Cell," Mar. 14 2019, US Patent App. 16/083,314.

[89] T. A. Manning, "Apparatuses And Methods For Comparing Data Patterns In Memory," 2018, US Patent 9,934,856.

[90] Micron, "Micron Collaborates With Broadcom To Solve DRAM Timing Challenge, Delivering Improved Performance For Networking Customers," http://investors.micron.com/static-files/3e9669f9-7186-481c-8594-dca7e992a0b2, 2013.

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

[92] O. Mutlu, S. Ghose, J. Gómez-Luna, and R. Ausavarungnirun, "Processing Data Where It Makes Sense: Enabling In-Memory Computation," in *Microprocessors and Microsystems*, 2019.

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

[94] Nanoscale Integration and Modeling (NIMO) Group, ASU, "Predictive Technology Model (PTM)," http://ptm.asu.edu/, 2012.

[95] NVIDIA, "NVIDIA GeForce RTX 2080 Ti Graphics Card." [Online]. Available: https://www.nvidia.com/en-eu/geforce/graphics-cards/rtx-2080-ti/

[96] NVIDIA, "NVIDIA GeForce RTX 3080 Ti Graphics Card." [Online]. Available: https://www.nvidia.com/en-us/geforce/graphics-cards/30-series/rtx-3080-3080ti/

[97] G. F. Oliveira, J. Gómez-Luna, L. Orosa, S. Ghose, N. Vijaykumar, I. Fernandez, M. Sadrosadati, and O. Mutlu, "DAMOV: A New Methodology And Benchmark Suite For Evaluating Data Movement Bottlenecks," *arXiv:2105.03725 [cs.AR]*, 2021.

[98] D. Pandiyan and C. Wu, "Quantifying The Energy Cost Of Data Movement For Emerging Smart Phone Workloads On Mobile Platforms," in *IISWC*, Oct. 2014.

[99] F. Parveen, S. Angizi, Z. He, and D. Fan, "Low Power In-Memory Computing Based On Dual-Mode SOT-MRAM," in *ISLPED*, 2017.

[100] F. Parveen, S. Angizi, Z. He, and D. Fan, "Imcs2: Novel Device-To-Architecture Co-Design For Low-Power In-Memory Computing Platform Using Coterminous Spin Switch," in *IEEE Trans. Magn.*, 2018.

[101] F. Parveen, Z. He, S. Angizi, and D. Fan, "Hybrid Polymorphic Logic Gate With 5-Terminal Magnetic Domain Wall Motion Device," in *ISVLSI*, 2017.

[102] F. Parveen, Z. He, S. Angizi, and D. Fan, "Hieim: Highly Flexible In-Memory Computing Using Stt Mram," in *ASP-DAC*, 2018.

[103] V. Patel, "DRAM CAM Memory," Aug. 8 2006, US Patent 7,088,603.

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

[105] I. Paul, W. Huang, M. Arora, and S. Yalamanchili, "Harmonia: Balancing Compute And Memory Power In High-Performance GPUs," in *ISCA*, 2015.

[106] J. T. Pawlowski, "Hybrid Memory Cube (Hmc)," in *HCS*, 2011.

[107] Y. Peng, B. W. Ku, Y. Park, K.-I. Park, S.-J. Jang, J. S. Choi, and S. K. Lim, "Design, Packaging, and Architectural Policy Co-optimization for DC Power Integrity in 3D DRAM," in *DAC*, 2015.

[108] A. S. Rakin, S. Angizi, Z. He, and D. Fan, "PIM-TGAN: A Processing-In-Memory Accelerator For Ternary Generative Adversarial Networks," in *ICCD*, 2018.

[109] A. K. Ramanathan, G. S. Kalsi, S. Srinivasa, T. M. Chandran, K. R. Pillai, O. J. Omer, V. Narayanan, and S. Subramoney, "Look-Up Table Based Energy Efficient Processing In Cache Support For Neural Network Acceleration," in *MICRO*, 2020.

[110] S. H. S. Rezaei, M. Modarressi, R. Ausavarungnirun, M. Sadrosadati, O. Mutlu, and M. Daneshtalab, "NoM: Network-On-Memory For Inter-Bank Data Transfer In Highly-Banked Memories," in *CAL*, 2020.

[111] P. Rosenfeld, "Performance Exploration Of The Hybrid Memory Cube," Ph.D. dissertation, rosenfeld2014performance, 2014.

[112] V. Seshadri, K. Hsieh, A. Boroumand, D. Lee, M. A. Kozuch†, O. Mutlu, P. B. Gibbons, and T. C. Mowry, "Fast Bulk Bitwise And And Or In DRAM," in *CAL*, 2015.

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

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

[115] V. Seshadri, T. Mullins, A. Boroumand, O. Mutlu, P. B. Gibbons, M. A. Kozuch, and T. C. Mowry, "Gather-Scatter DRAM: In-DRAM Address Translation To Improve The Spatial Locality Of Non-Unit Strided Accesses," in *MICRO*, 2015.

[116] V. Seshadri and O. Mutlu, "Simple Operations In Memory To Reduce Data Movement," in *Adv. Comput.*, 2017.

[117] V. Seshadri and O. Mutlu, "In-DRAM Bulk Bitwise Execution Engine," in *arXiv preprint arXiv:1905.09822*, 2019.

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

[119] P. Siegl, R. Buchty, and M. Berekovic, "Data-Centric Computing Frontiers: A Survey On Processing-In-Memory," in *MEMSYS*, 2016.

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

[121] P. R. Sutradhar, M. Connolly, S. Bavikadi, S. M. P. Dinakarrao, M. A. Indovina, and A. Ganguly, "Ppim: A Programmable Processor-In-Memory Architecture With Precision-Scaling For Deep Learning," in *CAL*, 2020.

[122] Y. Tian, T. Wang, Q. Zhang, and Q. Xu, "ApproxLUT: A Novel Approximate Lookup Table-Based Accelerator," in *ICCAD*, 2017.

[123] T. Vogelsang, "Understanding The Energy Consumption Of Dynamic Random Access Memories," in *MICRO*, 2010.

[124] K. Wang, K. Angstadt, C. Bo, N. Brunelle, E. Sadredini, T. Tracy, J. Wadden, M. Stan, and K. Skadron, "An overview of Micron's Automata Processor," in *IEEE/ACM/IFIP*, 2016.

[125] Q. Wang, X. Zhang, Y. Zhang, and Q. Yi, "Augem: Automatically Generate High Performance Dense Linear Algebra Kernels On X86 CPUs," in *SC*, 2013.

[126] Y. Wang, L. Orosa, X. Peng, Y. Guo, S. Ghose, M. Patel, J. S. Kim, J. G. Luna, M. Sadrosadati, N. M. Ghiasi *et al.*, "Figaro: Improving System Performance Via Fine-Grained In-DRAM Data Relocation And Caching," in *MICRO*, 2020.

[127] M. Ware, K. Rajamani, M. Floyd, B. Brock, J. C. Rubio, F. Rawson, and J. B. Carter, "Architecting For Power Management: The IBM®Power7™Approach," in *HPCA*, 2010.

[128] H. S. Warren, *Hacker's Delight*. Addison-Wesley, 2013.

[129] D. Weber, A. Thies, U. Kahler, M. Lepper, and R. Schutz, "Current and Future Challenges of DRAM Metallization," in *IITC*, 2005.

[130] W. A. Wulf and S. A. McKee, "Hitting The Memory Wall: Implications Of The Obvious," in *ACM SIGARCH Computer Architecture News*, 1995.

[131] S. Xiao and W.-c. Feng, "Inter-Block GPU Communication Via Fast Barrier Synchronization," in *IPDPS*. IEEE, 2010.

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

[133] I. Xilinx, "Zcu102 Evaluation Board: User Guide," https://www.xilinx.com/support/documentation/boards_and_kits/zcu102/ug1182-zcu102-eval-bd.pdf, 2019.

[134] I. Xilinx, "Vitis 2020.1," https://www.xilinx.com/products/design-tools/vitis.html, 2020.

[135] I. Xilinx, "Vivado 2020.1," https://www.xilinx.com/products/design-tools/vivado.html, 2020.

[136] X. Xin, Y. Zhang, and J. Yang, "ROC: DRAM-Based Processing With Reduced Operation Cycles," in *DAC*, 2019.

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

[138] L. Yang, S. Angizi, and D. Fan, "A Flexible Processing-In-Memory Accelerator For Dynamic Channel-Adaptive Deep Neural Networks," in *ASP-DAC*, 2020.

[139] J. Yu, H. A. Du Nguyen, L. Xie, M. Taouil, and S. Hamdioui, "Memristive Devices For Computation-In-Memory," in *DATE*, 2018.

[140] J. T. Zawodny and G. E. Hush, "Apparatuses And Methods To Reverse Data Stored In Memory," 2018, US Patent 9,959,923.

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

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

[143] H. Zhao, A. Goda, K. K. Parat, A. G. Mauri, H. Liu, T. Tanzawa, S. Yamada, and K. Sakui, "Apparatuses And Methods To Control Body Potential In Memory Operations," 2017, US Patent 9,536,618.

[144] B. Zoltak, "VMPC One-Way Function And Stream Cipher," in *FSE*, 2004.