Fulcrum: a Simplified Control and Access Mechanism toward Flexible and Practical In-situ Accelerators

Marzieh Lenjani*, Patricia Gonzalez*, Elaheh Sadredini*, Shuangchen Li**, Yuan Xie**,
Ameen Akel***, Sean Eilert***, Mircea R. Stan*, Kevin Skadron*
*University of Virginia, **University of California, Santa Barbara, ***Micron Technology, Inc.
{ml2au, lg4er, es9bt}@virginia.edu, {shuangchenli, yuanxie}@ece.ucsb.edu,
{aakel, sseilert}@micron.com, {mrs8n, skadron}@virginia.edu

ABSTRACT

In-situ approaches process data very close to the memory cells, in the row buffer of each subarray. This minimizes data movement costs and affords parallelism across subarrays. However, current in-situ approaches are limited to only row-wide bitwise (or few-bit) operations applied uniformly across the row buffer. They impose a significant overhead of multiple row activations for emulating 32-bit addition and multiplications using bitwise operations and cannot support operations with data dependencies or based on predicates. Moreover, with current peripheral logic, communication among subarrays is inefficient, and with typical data layouts, bits in a word are not physically adjacent.

The key insight of this work is that in-situ, single-word ALUs outperform in-situ, parallel, row-wide, bitwise ALUs by reducing the number of row activations and enabling new operations and optimizations. Our proposed lightweight access and control mechanism, Fulcrum, sequentially feeds data into the single-word ALU and enables operations with data dependencies and operations based on a predicate. For algorithms that require communication among subarrays, we augment the peripheral logic with broadcasting capabilities and a previously-proposed method for low-cost inter-subarray data movement. The sequential processor also enables overlapping of broadcasting and computation, and reuniting bits that are physically adjacent. In order to realize true subarray-level parallelism, we introduce a lightweight column-selection mechanism through shifting one-hot encoded values. This technique enables independent column selection in each subarray. We integrate Fulcrum with Compute Express Link (CXL), a new interconnect standard.

Fulcrum with one memory stack delivers on average (up to) 23.4 (76)× speedup over a server-class GPU, NVIDIA P100, with three stacks of HBM2 memory, (ii) 70 (228)× speedup per memory stack over the GPU, and (iii) 19 (178.9)× speedup per memory stack over an ideal model of the GPU, which only accounts for the overhead of data movement.

Keywords

processing in memory (PIM), in-situ computing, DRAM

1. INTRODUCTION

The energy consumption and execution time of applications with low computational intensity (low computation per datum) is mainly due to the high cost of data movement [1,2]. This is illustrated by a prior work that shows the energy cost of fetching a 32-bit word of data from off-chip DRAM is 6400× higher than an ADD operation in 45 nm technology [3]. We refer to applications with low computational intensity as memory-intensive applications. In the Big Data era, many applications, such as data analytics, scientific computing, graph processing, and machine learning, are memory-intensive.

To minimize the cost of data movement, recent studies have explored the possibility of processing data near or even inside the memory (PIM), for example, TOP-PIM [4], DRISA [5], Ambit [6], SCOPE [7], PRIME [8], ISAAC [9], and TOM [10].

Memory is designed in a multi-level hierarchy, consisting of vaults (in 3D stacked memories), banks, subarrays, etc. (more details in Section 2). PIM designs vary based on the level of the memory hierarchy at which the computation is performed. In-situ [5, 6] computing is one of the most aggressive forms of PIM that processes data in the row buffer, very close to the subarrays in which the data are stored. The subarray level provides a high throughput for two reasons: (i) the low latency of each access at the subarray level, and (ii) high subarray-level parallelism (e.g., Hybrid Memory Cube (HMC) has 512 banks [11], and assuming 32 subarrays per bank, the HMC will have 16384 subarrays.).

In-memory processing architectures are not limited to conventional DRAM configurations and interfaces (e.g., DIMM or HBM). Some prior in-situ approaches [5, 7, 8, 9, 12] target their design as a peripheral-attached accelerator instead of as a host memory to avoid the tight area and cost constraints for commodity DRAM. Indeed, these designs can be thought of as high-throughput accelerator architectures that happen to use DRAM as the most effective technology, based on parallelism, proximity to data, and overall capacity.

DRAM, compared to SRAM/eDRAM, provides a higher capacity for the accelerator. Compared to NVM, DRAM provides a lower latency. DRAM is also more tolerant to frequent writing of partial results. Therefore, in this paper, we focus on DRAM-based in-situ accelerators. Although computation in DRAM technology is inefficient [13] compared to a traditional logic process, the low computation requirement of the memory-intensive applications justifies the slower computations in DRAM-based in-situ accelerators.

In this project, we identified four problems that limit the benefit of current in-situ computing approaches.

The first limitation is the lack of flexibility for supporting a wide range of applications, which in turn, limits the market for the product. Prior in-situ approaches [5, 6] perform the same operation in all subarrays and on all bytes (e.g., 256 bytes) of the row buffer (row-wide operations). As a result, they cannot efficiently support operations with data dependencies along the row buffer or operations with a condition or predicate. For example, prior works cannot efficiently support reduction and scans (which are more efficient with serial operations along the row buffer), database operations such as Sort, FilterByKey, and FilterByPredicate (which require filtering based on predicate), or sparse operations (which require multiplications and accumulations only for nonzero values with matched indexes). Unfortunately, enabling operations with data dependency or operations based on a predicate, at a subarray level, using the traditional control and access mechanism is not practical (the area of a simple in-order core, such as ARM Cortex-A35 with 8KB of cache, excluding the SIMD units and scaled to 22 nm, is 25× larger than the area of a subarray).

The second issue is the capacity of the accelerator, which can be limited by the hardware overhead of the computing elements. At the subarray level, we read an entire row at once and store it in the row buffer. A processing unit with the capability of performing bitwise operations, addition, and multiplication (integer and single-precision floating point) on all bytes of the row buffer is at least 52× larger than the subarray. Thus, in-situ approaches only employ bitwise ALUs. It means any other operation should be emulated using multiple bitwise operations, requiring multiple subarray accesses and hundreds of cycles. We will show that the cost of emulating complex operations using bitwise ALU is higher than the cost of moving data out of the vaults. Some in-situ approaches use the analog computation capability of memory cells of ReRAM-based non-volatile memories (NVM) for multiplication and addition, without requiring any adder or multiplier [8, 9, 12]. However, this type of in-situ computation requires analog-to-digital converters (ADC), which also incur a significant hardware overhead. For example, the area overhead of ADCs in [12] is 45× higher than the area of the subarray. More importantly, analog computing introduces a high error rate and therefore, these accelerators suit applications that are highly tolerant of error, such as deep learning applications [8, 9]

The third problem is the physical layout of words in DRAM. The subarray itself is typically divided into smaller units, called mats. Due to circuit design constraints (which are explained in Section 3.4), in current memory designs, every four bits of a 32-bit word is stored in a separate mat. We refer to this as mat interleaving. Accordingly, prior in-situ approaches [5, 6] perform computation only on 1-bit, 2-bit, and 4-bit values.

The fourth limitation is inefficient peripheral logic. Many applications require sharing values among subarrays. For example, in matrix-vector multiplication, we can map each row of the matrix to a subarray. To perform matrix-vector multiplication, all subarrays need the values of the vector. A variety of applications require data movement among subarrays. For example, for the Sort application, the data is partitioned among subarrays. Then, all sorted partitions (which reside in different subarrays) should be merged. The merge requires inter-subarray data movement. Finally, many applications require parallel and independent column selection for each subarray. For example, in FilterByPredicate, in every subarray, we check a condition on each column and store the data in the row buffer, only if the condition is met. Therefore, different subarrays may write into different columns of the row buffer. Current peripheral logic serializes movement of shared values, inter-subarray movement, and column selection in different subarrays (more details in Section 3.3).

To resolve the aforementioned problems, we propose *Fulcrum*, where we rethink the design of in-situ accelerators to increase flexibility, practicality, and efficiency.

First, we propose a new lightweight access and control unit that processes data sequentially and determines the next operation based on the outcome of the previous operation (if necessary).

Second, we accommodate a processing unit capable of 32-bit addition, subtraction, and multiplication (in addition to bitwise operations) in the subarray level and limit the hardware overhead by performing operations on only a word of the row buffer at a time. Although this processing unit processes only a fraction (1/64) of the row buffer per cycle, it can provide a significant performance improvement overall. Our evaluation shows that performing complex operations on a subset of the row buffer outperforms prior in-situ approaches that perform computation on the whole row buffer but emulate complex operations by multiple bitwise operations.

Third, we slightly modified the mat-interleaving circuits to transfer all bits of a word to the side of a subarray (reuniting interleaved bits) so that all bits are physically close to each other. Since we only process one word at each cycle, the circuits for reuniting one word do not impose significant hardware overhead.

Fourth, we optimized peripheral logic for computation. To satisfy the data-sharing requirement, we enable broadcasting. For applications with inter-subarray data movement requirement, we employ a prior work [14] (Low-cost Interlinked SubArrays (LISA)) that transfers a whole row from one subarray to another at once, reducing the overhead of inter-subarray data movement. For applications with independent column access requirements, we enable a light-weight independent column selection mechanism through storing one-hot-encoded values of the column address in latches.

Our processing unit in each subarray has four parts: (i) Walkers, that provide sequential access with our lightweight column-selection mechanism. Walkers store the input operands of the computation (which are read from the memory array) or the output of the computation (to be written in the memory subarray), (ii) a small programmable instruction buffer, where we store the pre-decoded signals for the computation, (iii) a simple controller that determines the next operation and direction of the sequential access, and (iv) a single-word ALU. We show the flexibility of our design by mapping important kernels from different domains such as linear algebra, machine learning, deep learning, database management system (DBMS), as well as sparse matrix-vector multiplication and sparse matrix-matrix multiplication that appears in scientific computing, graph processing, and deep learning.

We integrate Fulcrum with CXL [15], a new class of interconnect that extends PCIe to significantly reduce the latency of access to the data in peripheral devices. Therefore, CXL provides high bandwidth and high power delivery of PCIe while supporting memory-like access to the data stored in Fulcrum. The low access latency enables communication through a single memory model. As a result, CXL can connect multiple in-memory accelerators, such as Fulcrum, and create a pool of disaggregated in-memory accelerators, providing scalability. The memory-like access of CXL also benefits workloads with phase behavior, where some phase is suitable for in-situ computing while other phases are not. In these cases, the host or other devices can access the output of Fulcrum with the latency in the order of latency of accessing normal memory.

Our paper makes the following contributions:

- Broadening the range of supported applications by in-situ accelerators
- Optimizing peripheral logic
- Optimizing mapping of important memory-intensive kernels and applications to Fulcrum.
- Releasing the source code of three artifacts: (i) MoveProf, a tool that uses profiled performance metrics for evaluating the cost of data movement, (ii) InSituBench, an in-situ computing benchmark suite, and (iii) the RTL code of our proposed method.
- Integrating Fulcrum with CXL to increase the power budget and enable scalability and disaggregation.

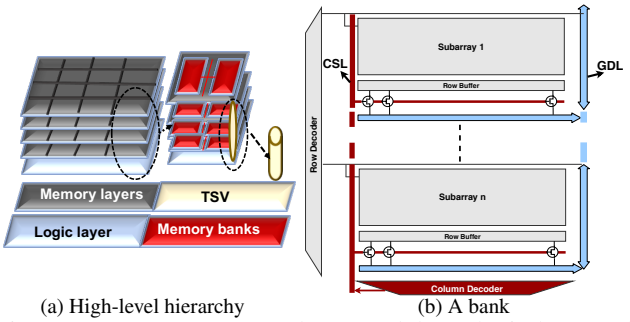
We evaluated our method against three approaches: (i) a prior work on DRAM-based in-situ computing (DRISA [5]), (ii) a server-class GPU that has three stacks of high-bandwidth memory (NVIDIA P100 with HBM2), and (iii) an ideal model of a GPU, where we take into account only the cost of data movement between GPU and the memory, assuming zero overhead for on-chip data movement and computation. Our evaluation shows that, for memory-intensive applications, our method delivers, on average (up to), 70 (228)× speedup per memory stack over the GPU. Our area evaluation shows that an 8GB integer Fulcrum (which supports bitwise operations and integer addition and multiplication) requires 51.74mm^2 (8 layers) and single-precision float Fulcrum (integer and bitwise functionality, plus floating-point addition and multiplication) require 55.26mm^2. The GPU die size of the NVIDIA P100 is 601mm^2. Accordingly, Fulcrum provides up to 839× higher throughput per area than the NVIDIA P100.

2. BACKGROUND

In this section, we introduce the structure of 3D stacked DRAM memories and their specific components that we have employed in our proposed method.

3D stacked DRAM is organized in a multi-level hierarchy. Figure 1 (a) illustrates the hierarchy of 3D stacked memories. Each 3D stacked memory has one logic layer and multiple (e.g., 4 or 8) memory layers. Each stack is divided into multiple vaults [11]] (e.g., 16 or 32). Layers are connected via TSVs (through-silicon via). The TSV acts as a shared bus. This shared bus is a bottleneck, as it serializes access to all subarrays in a vault. It is possible to employ a segmented TSV [16, 17], where every two layers are connected through one separate TSV, and data from upper layers are buffered in each layer before being sent to the lower layers. Although segmented TSVs increase the access bandwidth, they increase the latency and the energy consumption of data movement (due to the extra cost of buffering and arbitration) [16, 17].

In each layer of each vault, there are a few banks (e.g., 2 or 4). Figure 1 (b) shows the structure of a memory bank. A bank comprises several subarrays. Each subarray has multiple rows. Each row contains multiple columns of data. The column width varies in different DRAM configurations. The most common column widths are 32, 64, 128, and 256 bits. To access one column of the data from a bank, a row address is sent to all row decoders in all subarrays through the address bus in the bank. The row decoder selects the corresponding subarray and the corresponding row. The whole row that contains multiple columns is read out at once and will be stored in a row-wide buffer in the subarray, which is called the row buffer. To select one column from the row buffer, the column address should be sent to the column decoder, at the edge of the bank. The column decoder decodes the column address and sends the decoded bits to each subarray through column selection lines (CSL). The pass transistors/multiplexers in each subarray receive the decoded column address on CSLs and select the requested column and send it to the local data line (LDL), in each subarray. The data on LDL is sent to the logic-layer via global data lines (GDL). Two components make 3D stacked memories more suitable for Fulcrum: (i) the logic layer, and (ii) shared TSVs. Section 4 and 5 explain how we employ the shared TSVs for broadcasting and discuss the role of the logic layer in broadcasting data and reducing the partial results.



(a) High-level hierarchy (b) A bank

Figure 1: 3D DRAM: (a) each 3D stack has a logic-layer and a few memory layers containing multiple memory banks, (b) each bank comprises several subarrays

3. PROBLEM STATEMENT

In order to reduce the cost of data movement for tasks that are data-intensive and have locality within a row or subarray, prior works have proposed in-situ computing, where we perform computation on the row buffer, and therefore, there is no need to move data out of the subarray. We identified four problems that have hindered adoption and realization of in-situ approaches.

3.1 Lack of flexibility

Recent in-situ approaches employed non-flexible row-wide operations. As a result, they cannot support operations with

any form of dependency along the row buffer. For example, in operations such as Scan, the value of each partial sum depends on the value of the previous partial sum. They also cannot support algorithms that check a condition on a value and perform a different operation based on the outcome of the condition. For example, radix sort is an algorithm that sorts data by grouping values by the individual digits, which share the same significant position. Each iteration of this sort algorithm packs values into two different buffers. The target buffer for the value is determined by the digit that is being processed at that iteration. Another example is sparse matrix-vector multiplication, where we often store the non-zero values next to their indices (instead of wasting the capacity by storing the whole matrix with mostly zero values). Consequently, we only perform multiplication and accumulation on non-zero values, whose index matches.

3.2 Lack of support for complex operations

Prior works have evaluated a spectrum of in-situ computing. Seshadri et al. [6], evaluated row-wide bitwise operations using computation capability of bitlines without adding any extra gate, realized by activating two rows at the cost of destroying the values in both rows, requiring extra copies before each operation. Li et al. [5], evaluated row-wide bitwise ALUs, shifters, and latches (the latches eliminate the extra copies), emulating 4-bit addition and 2-bit multiplication using bitwise ALUs. They also evaluated adding row-wide 4-bit adders to the row buffer and reported that this increases the area by 100%. Unfortunately, emulating complex operations such as addition or multiplication using bitwise ALUs requires reading and writing multiple rows. Since row activation is very costly, the energy consumption of row activation for emulating complex operations by bitwise operations surpasses the energy consumption of sending data to the logic layer, as shown in Figure 2.

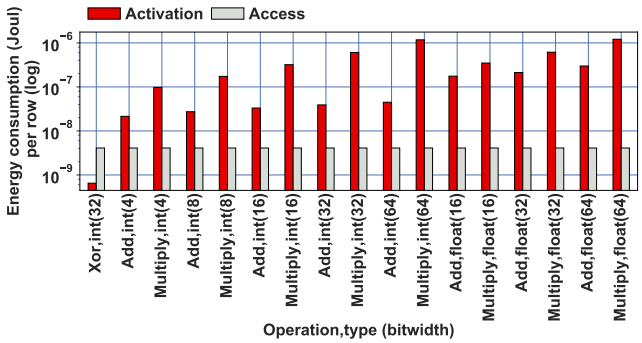


Figure 2: Energy consumption of accessing a row in the logic layer vs. energy consumption of multiple row activations, required for emulating complex operations by bitwise operations

3.3 Inefficient peripheral logic

Currently, there are three shared resources in the design of the memory: (i) the TSVs, shared among all layers in a vault (as explained in Section 2 and shown in Figure 1), (ii) the shared CSLs, and (iii) the GDLs, shared among all subarrays in a bank. With shared CSL, we can only select one column from a whole bank at a time and with shared GDLs we can transfer only one column at a time. As a result, the current peripheral logic and interconnect limit the performance of in-situ approaches in three ways: (i) although they act as a shared bus, they are not capable of broadcasting values for efficient data sharing, (ii) narrow GDLs are the only means for movement of an entire row from one subarray to another, (ii) the peripheral logic for column access (the column decoder) is shared among all subarrays (Figure 1 (b)). This shared peripheral logic limits the flexibility and parallelism of any potential in-situ approach that requires independent and parallel column access to the row buffer of individual subarrays.

3.4 Interleaving

In current commercial DRAMs, we have two types of interleaving: (i) mat interleaving, and (ii) subarray interleaving. Mat interleaving is shown in figure 3, where each subarray is divided into multiple mats [18, 19]. The GDLs are distributed among mats, and each mat has 4 bits of the GDLs. Therefore, for selecting the same column from all mats, CSLs are repeated for each mat. Pass transistors (PTs) receive CSLs, select a column, and place it on LDLs. This design is called mat interleaving and is efficient for random column access, as it reduces the LDLs's latency (LDLs in Figure 3 are shorter than LDLs in Figure 1 (b)). Without mat interleaving, LDLs become wide and long, where the latency of the last column is much longer than the latency of the first column.

The second type of interleaving is subarray interleaving or open-bitline architecture [14, 20]. Since the size of a sense amplifier is larger than a cell [20], modern DRAM designs accommodate only as many as sense amplifiers in a row to sense half a row of cells. To sense the entire row of cells, each subarray has bitlines that connect two rows of sense amplifiers, one above and one below the subarray.

As a side benefit, mat interleaving and subarray interleaving make the memory more robust against multiple-bit upset, where soft errors change the value of adjacent cells. In fact, when bits in a column are not physically close to each other, multiple-bit upset only changes one bit from a column and then error detection mechanisms (which can detect one error) can detect the error. Therefore, keeping the current interleaving and not changing the layout is desirable.

However, with interleaving, row-wide computation on more than 4-bit values is impractical, as the result of an addition and multiplication in each 4 bits of the output depends on the values in other mats. With row-wide operations, the circuits for reuniting the interleaved bits impose a significant hardware overhead as many wires cross each other.

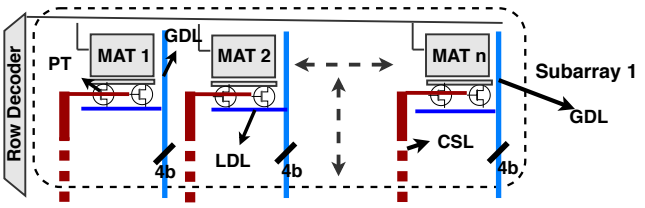


Figure 3: Interleaving a word among mats

4. KEY IDEAS

The key insight of this paper is that a narrow ALU with adder and multiplier, placed at the edge of each pair of subarrays, can outperform row-wide bitwise ALUs, because bit-

wise ALUs suffer from the drawbacks previously described. We show that these capabilities are important for a variety of computational kernels. Figure 4 (a) shows the architecture of our in-situ processing unit, which has two parts: (i) Walkers and (ii) AddressLess Processing Unit (ALPU). The Walkers are implemented by three rows of latches that are connected through a bus similar to LDLs (implementation details in 5).

The ALPU itself comprises four components: (i) a controller, (ii) three temp registers, (iii) an ALU, and (iv) an instruction buffer.

In this section, we explain the role of these components in resolving prior in-situ approaches' limitations and elaborate on implementation detail in Section 5.

4.1 A simplified control and access mechanism

Since a narrow ALU only processes a word of row buffer, we need a sequential access mechanism for selecting consecutive words. Due to the significant hardware overhead, employing a core with traditional access and control mechanism, for sequentially selecting a word from a row buffer at the subarray level, is impractical. We could give up subarray-level parallelism and employ only one traditional core per bank to limit the hardware overhead. Unfortunately, other than losing subarray-level parallelism, this solution imposes a significant overhead for control and access. Since logic in DRAM layers is slow, and the cost of data movement is low, unlike far-memory processors, this overhead comprises a significant portion of total energy consumption and execution time. Figure 5 (top) illustrates the control and access overhead of a traditional core at the bank-level for adding two vectors and storing the result in a third vector. In this example, the core generates an address for each element of the three vectors, imposing access overhead. Since the core is placed at the edge of each bank, the decoded addresses should be sent to the subarray through CSLs (Section 2 and Figure 1). Data read from the subarray also should move toward the core through GDLs, imposing data movement overhead. The decoding of all these instructions, branch prediction, and checking for data dependency in the pipeline of traditional cores impose control overheads. Despite the significant overhead, such a control and access mechanism provides full flexibility.

Our proposed method provides a tradeoff between flexibility and the overhead of control and access. In fact, while we enable operations with data dependency and operation based on predicates, we avoid the overhead of sophisticated control mechanisms and the overhead of accessing data by address. We observed that for almost any memory technology, an entire row is read/activated at once and stored in a buffer. We introduced Walkers, where each Walker either captures a row of input operands (read from the subarray) or stores a row of target variables (before being written to the subarray). We read/write to/from these rows sequentially and implement the sequential accesses using shifting of a one-hot-encoded value that determines which column of the row should be selected to be placed on the bus. Our simple controller determines the direction of shifts in each Walker and also determines the next operation based on the outcome of the previous operation, providing flexibility. Accordingly, for example, our simplified access and control mechanism performs an addition of two vectors by iteration of an instruction over the row buffer, similar to the instruction shown in Figure 5 (down) (Section 5 presents the exact format of Fulcrum's instructions and discuses how our hardware modules control loading new rows of each vector to Walkers).

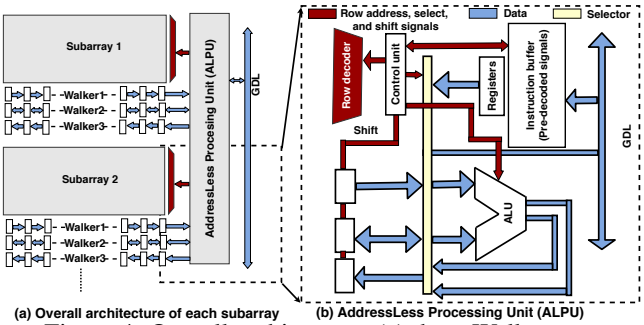


Figure 4: Overall architecture: (a) three Walkers per subarray and one ALPU per every two subarrays, (b) ALPU comprises a controller, an ALU, an instruction buffer, and few temp registers

(a) Traditional control and access mechanisms

1. RA=RA+1 // address generation for vector a[i] + control
2. RB=RB+1 // address generation for vector b[i] + control
3. RC=RC+1 // address generation for vector c[i] + control
4. Read RA, X1 // decoded address movement on CSLs
   // data movement on GDL + control
5. Read RB, X2 // data movement on GDL + control
6. X3=X2+X1 // the actual computation + control
7. Store X3, RC //decoded address movement on CSLs
8. i=i+1 // iteration counting + control
9. If (i< 1000000) Jump 1 // iteration checking+ control

(b) Fulcrum

Shift_Read a[i], shift_Read b[i], Shift_Write c[i] , c[i]=a[i]+b[i]

Figure 5: A traditional control and access mechanism vs. Fulcrum

4.2 Narrow and simple ALU

Our sequential access to Walkers enables processing only one word (one column) at a time, and consequently we do not need row-wide ALUs. We observed that addition, comparison, multiplication, and bitwise operations are the most common operations that appear in modern memory-intensive applications. Therefore, we included a single-word ALU, which supports these common operations. The input operands of the ALU can come from one of the four resources: (i) the value sequentially accessed from one of the Walkers, (ii) temp registers, (iii) the GDLs, or (iv) one of the outputs of the ALU (our controller supports two operations in one cycle). Section 6 explains that although our ALU is narrow, it outperforms row-wide bitwise operations and supports modern memory-intensive applications.

4.3 Efficient peripheral logic

We have introduced minor modifications in the peripheral logic to increase flexibility, parallelism, and efficiency of data movement for our method. First, we added a broadcasting command, by which every processing unit receives and captures the data on shared buses. Second, we build upon

Low-cost Inter-linked SubArrays (LISA) [14] to transfer an entire row at once to any other subarray in the same bank (otherwise we had to transfer the entire row, column by column, through the narrow GDL bus). Third, for independent column access in each subarray, instead of using column decoder and column address buses, which are shared among subarrays, we employ column selection latches in each subarray, where we store a one-hot-encoded value that determines the selected column. In each cycle, based on the outcome of the previous operation, the controller decides in which direction the on-hot-encoded value should be shifted.

4.4 Reuniting interleaved bits

Unlike reuniting the whole row, reuniting one word is possible through a slight modification of the current mat interleaving circuits. Therefore, we can transfer and reunite interleaved bits of a word at the side of the subarray to perform arithmetic operations.

To resolve subarray interleaving, we simply use only one processing unit per two subarrays. Figure 4 shows that per every two subarrays, we only have one ALPU.

To resolve the mat interleaving, we propose two solutions. Our first solution is to change the layout and completely remove the mat interleaving (if the target application does not need efficient random column access or is resilient against soft errors). As a side benefit of eliminating mat-interleaving, we will save the area overhead of CLSs and columns selection logics (repeated for each mat). The second solution is

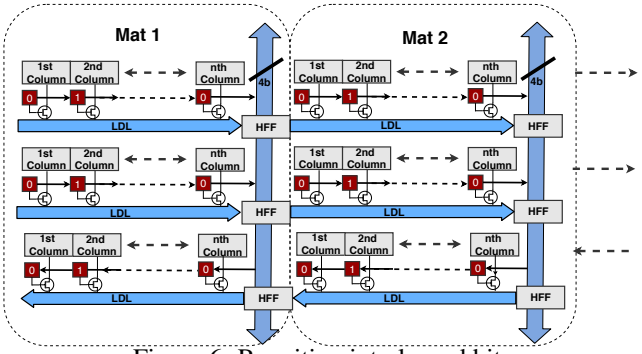


Figure 6: Reuniting interleaved bits

to keep the mat interleaving. In the traditional design of memory, each segment of the LDL corresponding to each mat is connected to four flip-flops (called helper flip flops (HFF)). We connect these flip flops to the segment of the LDL of the adjacent mat and form a pipeline so that we can transfer all values to the side of the subarray in a pipeline fashion. In a memory structure with four mats per subarray, this pipeline requires four cycles to transfer 32 bits (16 bits from the upper Walker and 16 bits from the lower Walker). If we use the TSV for sending the clock signal, the clock cycle should be at least twice as long as the latency of the TSV (according to CACTI-3DD, the latency of TSV with eight memory layers and one logic layer is more than 4.4 ns). This will significantly degrades our throughput. To resolve this issue, we employed segmented TSV [16, 17] (explained in Section 2) for the clock signal (according to CACTI-3DD, the latency of segmented TSV is 0.3 ns).

5. IMPLEMENTATION DETAIL

This section elaborates on hardware and software implementation.

5.1 Hardware

This section provides more implementation details about six hardware components: (i) Walkers, (ii) the controller, (iii) the instruction buffer, (iv) our in-logic layer components, (v) interconnect, and (vi) the walker renamer.

5.1.1 The Walkers

Walkers provide sequential accesses. We have two options for implementing the Walkers. The first option is changing the layout, employing shift registers (or shift latches, implemented by ping-pong shifting [21]), and accessing the row sequentially by shifting the values. The second solution is keeping the interleaved layout, employing the structure of traditional row buffers and local buses (similar to LDLs, explained in Section 2) along with a column-selection mechanism that selects a column to be placed on each Walker's LDL. In addition to keeping the interleaved format, the second solution has two side advantages: (i) enabling sequential read and write in both directions (with the first option, we can only read by shifting to the right and write by shifting to the left (in Figure 4)), and (ii) consuming less energy (with first option, per each shift, there will be value transition in all latches, whereas with the second solution, only the value of the bus and only one latch changes).

The only difference between the structure of the traditional row buffer and our Walkers is the column-selection mechanism. As we explained in Section 2 and Section 3, traditional memories share the peripheral logic for column selections. Figure 1 (b) shows that the CLSs (on which the decoded column address is placed), are shared among subarrays. Figure 3 shows that CLSs are repeated for each mat. To support operations based on predicates, we need independent column access for all Walkers and subarrays. Hence, we introduced column-selection latches where we store the one-hot-encoded value of a column and shift the value to access the next column, without requiring one column decoder per Walker, per subarray, and per mat.

5.1.2 The controller

Our controller employs a few counters: (i) a 6-bit counter (6 = log2 of the number of words in a row (64)) per Walker for detecting a fully-accessed Walker (fully read or written). Each Walker has a 2-bit latch that determines to which Walker we should switch and rename when the Walker is fully accessed (elaborated in Section 5.1.6), (ii) a 4-bit counter (4 = ceil of log2 of the number of wait cycles (9)) for counting the wait time for a new row to be read from the subarray and be stored in a Walker, or for a Walker to be written to the subarray, and (iii) three 11-bit (11 = log 2 of the number of rows in each pair of subarrays (2048)) row counters which are initialized to the row address of the beginning of the data and will be compared against the end of the data in the subarray.

5.1.3 The instruction buffer

Figure 7 illustrates the format of each entry of the instruction buffer. This format allows two operations at the same

time and has the following fields: (i) NextPc1 and NextPc2 that determine the program counter of the next instruction, (ii) NextPc_Cond determines the condition under which the controller switches to instruction determined by NextPc1 (otherwise, it switches to NextPc2). When NextPc1 equals NextPc2, the NextPc_Cond is used for determining which comparison flag should be the input bit of the bitwise shift operation, (iii) opCode1 and opCode2 are the operation codes of each operation, (iv) Src1Op1, Src2Op1, Src1Op2, and Src2Op2 select a source for each input of the operation, (v) ShiftCon1, ShiftCon2, and ShiftCon3 specify the condition under which the corresponding Walker should be shifted, (vi) ShiftDir1, ShiftDir2, and ShiftDir3 determine the direction of shifts in each row, (vii) repeat filed is the number of repeat before shifting when any of the shift conditions are IF_REPEAT_ENDS_SHIFT", and (vii) OutSrc selects the value shifted to the destination row among the two operation outputs and the two outputs of the first two Walkers. Although for the evaluated benchmarks an instruction buffer with four entries is enough, we also evaluated our area overhead with an instruction buffer that has eight entries.

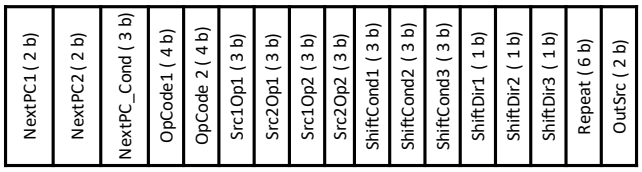


Figure 7: The format of instructions

5.1.4 *In-Logic layer components*

For in-logic layer operations, we use an ARM Cortex-A35 which is used in prior works [22] as the processing unit in logic layer (because it has low power consumption), along with a 128 KB buffer for buffering shared values.

5.1.5 *Interconnect*

Fulcrum can be integrated into a system in two ways. The first option is to integrate Fulcrum with CPU/GPU through current 3D interfaces. For this option, we have to lower the power budget to 10 Watt [4, 22] (we evaluate Fulcrum under different power budgets in Section 6). The second option is to use CXL [15] interface (similar to GPU and FPGA), which allows a higher power budget. This option allows a memory-like access to the data stored in Fulcrum. As a result, for workloads that have phase behavior, where some phases are suitable for in-situ while other phases are not, the host, GPU, FPGA, or any other device can access the output of Fulcrum (output of in-situ phases).

5.1.6 *The Walker renamer*

Fulcrum exploits broadcasting for reducing the cost of data movement. To overlap computation and broadcasting, all subarrays should work in lockstep so that the broadcasted value is used for computation and can be discarded in the next cycle, eliminating the need for storing broadcasted values. However, in some applications, the processing time for each Walker in each subarray might vary, hindering the lockstep computation. We propose to exploit Walker renaming to solve this problem. As an example, we explain the role of Walker renaming in SPMV.

To represent sparse matrices (where most of the values are zero), we can employ a few formats. One of the most popular formats is the compressed sparse row (CSR) format, which represents a matrix M by three arrays containing: (i) nonzero values, (ii) the positions of the start of the rows, and (iii) column indices. A naive implementation lays out the three vectors in three different rows and uses all Walkers. Since the values of the vector are being broadcasted, when a controller detects a fully accessed Walker in any of the subarrays, the process, in all subarrays, should wait until a new row is read into the Walker. To avoid this overhead, we place each non-zero value and its corresponding column index subsequently in the same array. This way, we only need one Walker for computation. Therefore, for example, while Walker A is being processed, another row can be captured in Walker B. When Walker A is fully accessed, the computation can continue by processing Walker B. However, the ALPU's instruction buffer is programmed to process Walker A. Here Walker renaming can help. As explained in Section 5.1.2, when Walker A is fully accessed, Walker B will be renamed to Walker A so that computation can continue with the same ALPU instruction.

5.2 Software

In this section, we discuss programming, data placement, and high-level programming.

5.2.1 *Programming*

Fulcrum's programs comprise two parts: (i) the in-logic layer portion, and (ii) the ALPUs' portion.

Our in-logic layer programs interact with vault controllers for generating commands for setting ALPU's registers and instruction buffer, sending broadcast values, and collecting the partial results. It also reduces the partial results or performs specific functions on intermediate results. Our in-logic layer core is an ARM Cortex-A35 and can be programmed by high-level programming languages such as C++.

To program the ALPU, we used the low-level instructions explained in Section 5.1.3 (Figure 7). Our online repository [23] contains ALPU programs for the evaluated applications. A non-expert programmer can easily use ALPU libraries, written by experts (similar to machine-learning users with no CUDA knowledge that are using NVIDIA libraries).

Therefore, a Fulcrum kernel call first loads the in-logic layer program. The in-logic layer program generates commands for the vault controller to load the ALPU programs and other ALPU settings and start the computation.

5.2.2 *Data placement*

The layout of data highly affects the performance benefit of Fulcrum. The data should be partitioned and laid out carefully to enable exploiting subarray-level parallelism, broadcasting, and the light-weight sequential access mechanism. For example, in matrix-vector multiplication, we use a row-oriented layout for the matrix and map each row of a matrix to one pair of subarrays. In each cycle, we broadcast one element of the vector to all ALPUs, and each ALPU multiplies the broadcasted value by the corresponding element of the row of the matrix. To choose the best strategy for placing data in the desirable layout, we categorize data as either: (i) long-term

and (ii) temporary resident data. The first category resides in Fulcrum for a long time, but the second group is the input of the application or the intermediate results that reside in Fulcrum temporarily. For example, DNN algorithms are composed of several layers. The core of computation in each layer is a matrix-vector multiplication, where a matrix of weights are multiplied by a vector of activations (for batch size of one). The output of each layer is a vector, which is the activation vector for the next layer. The matrix of weights can reside in Fulcrum for a long time. However, the activation vectors, which are the output of each layer, only reside in Fulcrum for a short time. Machine learning models such as reference points in KNN or database tables are other examples of long-term resident data. The query points in KNN are examples of temporary resident data.

For long-term data, we assume an offload paradigm for both the 3D and CXL deployments. So an API similar to CUDA's API (cudaMemcpy()) manages the data transfer. Mapping the address space to DRAM rows, banks, and subarrays is configurable in the memory controller [11]. Therefore, by copying data in a specific address, the programmer can place data in the desirable layout. For this group, we can ignore the overhead of laying out the data as this is a one-time cost.

The temporary resident data itself can be categorized into two groups. The first group are the data that we broadcast and do not need to be stored, such as the activation vectors or the query points in KNN. We store these values in our in-logic layer buffer. The space in this broadcasting buffer has also a memory address. If these data are generated outside Fulcrum (for example, in GPU, in the previous phase of the applications which are not suitable for in-situ computing), an API copies these data to the in-logic layer buffer. If these data are generated inside Fulcrum, our in-logic layer core's program collects these data from the memory and stores it in the in-logic layer buffer. The second group are the data that cannot be broadcast. If these data are generated outside Fulcrum, we propose to employ on-the-fly layout optimization methods [24] that are proposed for GPUs. If these data are generated inside Fulcrum but need to be laid out differently for the next phase, our in-logic layer core's program collects and changes the layout for the next phase.

5.2.3 High-level programming

We realize that a non-expert programmer will not write an assembly program for ALPUs. Our future work will develop a high-level programming language and a software stack. We hypothesize that a programming model similar to TensorFlow suits Fulcrum. Accordingly, we envision a software stack composed of two steps. The first step is to implement the important kernels of most commonly used libraries, such as cuBLAS [25], cuSPARS [26], and Thrust [27] and any other useful primitive such as Reduction, Scan, Sort, and Filter that are amenable for in-situ computing. The second step is to develop a programming model similar to TensorFlow([12]). The TensorFlow programs are Data-Flow Graphs (DFG) where each operator node can have multi-dimensional vectors, or tensors, as operands. The compiler transforms the input DFG into a collection of primitives and kernels which are implemented in step1. A similar approach is used for TPU and prior in-situ accelerators [7, 12].

6. EVALUATION

In this section, we first describe our evaluation methodology. Second, we compare the performance of our method against three approaches: (i) a server-class GPU, (ii) a prior work on in-situ computing, and (iii) an ideal model of the GPU, where we only incorporate the cost of data movement. Third, we discuss applications' characteristics that affect the Fulcrum's performance and energy benefit. Fourth, we evaluate the effect of each problem. Finally, we present area, energy, and power evaluation results, as well as performance evaluation under specific power budgets.

6.1 Methodology

We evaluated performance, area, and energy consumption of Fulcrum. For performance evaluation of Fulcrum, we divided applications into multiple phases, where each phase could either be an ALPU processing phase or an in-logic layer processing phase. We evaluated ALPU processing by modeling the ALPU's computation time, including both the row activation time and processing time. We also modeled the in-logic layer processing time, including data movement and partial-result calculations.

We evaluated Fulcrum with both integer and floating-point configurations: Integer Fulcrum is capable of integer addition and multiplication, as well as bitwise operations, whereas float Fulcrum adds single-precision floating-point addition and multiplication.

The major benefit of Fulcrum is reducing the cost of data movement. Accordingly, to abstract away from architectural details of GPUs, we also evaluated against an ideal model of the GPU, where we only incorporate the cost of data movement to and from the GPU's global memory. To this end, we measured the data that are read or written to the GPU's DRAM (using NVProf [28]) and divided the measured data movement by the raw bandwidth of the memory stack to obtain the performance cost of the data movement.

Our RTL and CACTI-3DD [19] evaluations show that Fulcrum can work at a frequency of 199 MHZ, in 22nm technology. We added slack of 21.5% (to incorporate the delay penalty of logic in DRAM technology [13]) and evaluated Fulcrum with 164 MHZ.

Table 2 lists the configuration of our evaluated systems, and Table 1 introduces our in-situ benchmark suite, InSituBench [29] (a combination of memory-intensive kernels, suitable for in-situ computing, from different domains). We selected Sort, Scan, Reduction, GEMM (matrix-matrix multiplication), and GEMV (matrix-vector multiplication) from the NVIDIA SDK benchmark [30]. We also included sparse matrix-vector (SMPV) and sparse matrix-matrix (SPMM), LSTM (a deep learning application), K-nearest neighbor (KNN) [31, 32] (a classical machine learning application), Scale, and AXPY (representatives of simple kernels). We also added FilterByKey, FilterByPredicate, Bitmap (from DBMS domain [22, 33, 34, 35]), and Xor (representative of bitwise kernels, used in bitmap indexing [36] and bitmap-based graph processing [5, 6, 37, 38]).

For area evaluation, we designed the ALPU in RTL and synthesized the modules using an industry-standard 1xFin-

Table 1: The evaluated applications

Application	Implementation	Operation	DRISA ?	input options
AXPY	cuBLAS [25]	add and multiply	Yes	–Num=1000000000
Bitmap	Thrust [27]	compare and bitwise shift	No	–Num=1000000000
FilterByKey	Thrust [27]	compare	No	–Num=1000000000
FilterByPredicate	Thrust [27]	compare	No	–Num=1000000000
GEMM	cuBLAS [25]	add and multiply	Yes	–numRowA=25600 numColA=19200 –numRowB=19200 –numColB=12800
GEMV	cuBLAS [25]	add and multiply	Yes	–numRow=25600 –numCol=19200 –lsVector=1
KNN	Fast KNN [31, 32]	add and multiply	NO	–global_mem –ref_nb=100000 –query_nb=1
LSTM	cuDNN [39]	add and multiply	Yes	–seqLength=100 –numLayers=4 –hiddenSize=4096 –miniBatch=1
Reduction	NVIDIA SDK [30]	add	Yes	–Num=16777216
SPMM	cuSPARSE [26]	compare, add and multiply	No	–NumRowA=8192 –NumColA=100000 –NumColB=8192 –percentage=0.2
SPMV	cuSPARSE [26]	compare, add and multiply	No	–NumRow=8192 –NumCol=100000 –percentage=0.2
Scale	Thrust [27]	multiply	Yes	–Num=1000000000
Scan	NVIDIA SDK [30]	add	No	–Num=1073741824
Sort	NVIDIA SDK [30]	compare	No	–Num=10000000
Xor	Thrust [27]	bitwise	Yes	–Num=1000000000

FET technology with foundry models (in modern technologies the node number does not refer to any feature in the process, and foundries use slightly different conventions. We use 1x to denote the 14/16nm FinFET nodes offered by the foundry.). Then we scaled the area estimation (both pessimistic and optimistic) to 22 nm technology. We modeled the area of Walkers in CACTI-3DD [19].

To evaluate the energy consumption, we extracted the energy consumption of ALPU by RTL simulation, and used the energy consumption modeling of CACTI3DD [19] for Walkers. To evaluate the breakdown of energy consumption for the GPU, we developed MoveProf [40] that integrates NVIDIA's NVProf [28] with GPUWattch [41]. GPUWattch [41] uses RTL models for processing elements and CACTI [19] for memory elements. Therefore our evaluation of Fulcrum is comparable to GPUWattch [41].

Table 2: Configuration details for evaluated architectures

Component	Parameters
GPU	Tesla P100 [42], 12 GB memory 3 HBM2 memory stacks at 549 GB/s (183 GB/s per stack)
Fulcrum	technology:22 nm, 32 vaults 32 subarray, open-bitline structure 4 mats per subarray, 256 bytes per row 64 banks per layer, 8 memory layers, row cycle:50 ns, frequency:164 MHz In-logic layer: 128 KB SRAM-based FIFO ARM Cortex-A35
Ideal machine	HBM2, bandwidth:183 GB/s

6.2 Performance improvement over GPU

Figure 8 illustrates the throughput of Fulcrum over a server-class GPU, NVIDIA P100. This figure shows that Fulcrum outperforms the GPU, on average by 23.4×, and up to 76× (achieved for Bitmap). For applications such as Sort, with lower memory-intensity, the speedup is around one order of magnitude (8.8×). Applications such as GEMM, which can employ blocking to significantly increase locality, gain lower speedup (1.5×). NVIDIA P100 has three memory stacks and Fulcrum has one memory stack. Therefore, Fulcrum delivers, on average (up to), 70 (228)× speedup per memory stack over the GPU.

6.3 Comparison against an ideal model and bitwise row-wide ALUs

The most beneficial aspect of in-situ computing is reducing the cost of data movement. Therefore we evaluated our method against an ideal model of GPU, where we only incorporated the cost of data movement between DRAM and

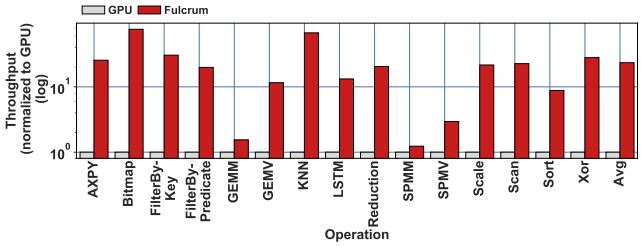


Figure 8: Throughput comparison against NVIDIA P100

GPU. We also evaluated our method against DRISA [5] for applications that do not have branches, where the complex operations can be emulated using bitwise operations. Figure 9 illustrates the throughput per memory stack of the ideal model, DRISA [5], GPU, and Fulcrum. This figure shows that Fulcrum outperforms the ideal model, on average, by 19× and up to 178.9×. However, it can not outperform the ideal model for GEMM, which has a higher locality. This figure also shows that Fulcrum can outperform DRISA [5] (the last column of Table 1 indicates applications that DRISA can support), often by more than two orders of magnitude. However, Fulcrum is 3.5× slower than DRISA for Xor–a bitwise task ideally suited for DRISA.

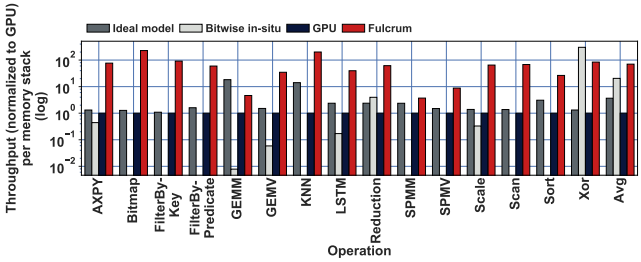


Figure 9: Throughput per stack comparison against the ideal model, DRISA [5], and GPU

Table 3: Metrics used in Figure 10

Metric	Definition
dram_read_bytes	Total bytes read from DRAM to L2 cache
dram_write_bytes	Total bytes written from L2 cache to DRAM.
inst_integer	Number of integer instructions executed by nonpredicated threads
inst_fp_32	Number of single-precision floating-point instructions executed by non-predicated threads (arithmetic, compare, etc.)
stall_memory_dependency	Percentage of stalls occurring because a memory operation cannot be performed due to the required resources not being available or fully utilized, or because too many requests of a given type are outstanding
sm_efficiency	The percentage of time at least one warp is active on a multiprocessor averaged over all multiprocessors on the GPU
sm_inefficiency	1-sm_efficiency
memory_read_per_computation	dram_read_bytes/(inst_integer+ inst_fp_32)
memory_write_per_computation	dram_write_bytes/(inst_integer+ inst_fp_32)

6.4 The effect of application characteristics

Figure 10 shows detailed performance metrics, collected by the NVIDIA profiler [28]. Tables 3 presents the definition of the metrics used in this Figure. In this Figure, the y-axis shows the normalized value of the metric, so that the y-axis value for the application with the highest metric value is one. This Figure demonstrates that applications such as Bitmap that have very high memory_read_per_computation benefit more from Fulcrum. Since reading data from DRAM is in the critical path, memory_read_per_ computation is more important than memory_write_per_computation. KNN, which gains a high speedup, has a high value of sm_inefficiency.

It also shows that SPMV on GPU has the highest stall_memory_dependency, which is the result of indirect memory accesses such as x[col[j]] [43]. To implement SPMV on Fulcrum, we partition rows of the matrix among subarrays and store column indexes and non-zero values consecutively. We broadcast the index of the vector elements, followed by the corresponding value. In each subarray, the ALPU checks the column index of the non-zero value with the broadcasted index and perform multiplication and addition for matched indexes. While our implementation does not require indirect memory accesses, for highly sparse vectors, we waste many cycles for broadcasting values that will not be matched in any subarray. However, the simplified control and access mechanism and in-situ computing (which reduces the energy consumption of data movement) still provide energy benefits. Figure 11 illustrates that the higher the density, the lower normalized energy-delay product (EDP) (the lower, the better). Therefore we can conclude that Fulcrum benefits applications with the density of 3-100%. Prior works have shown that many problems in statistics and sparse neural networks have such density [44].

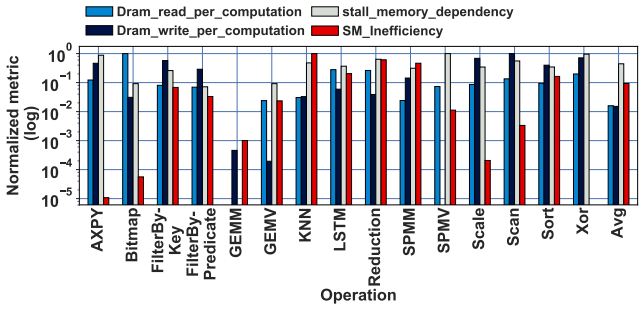


Figure 10: Performance metrics that affect Fulcrum's speedup

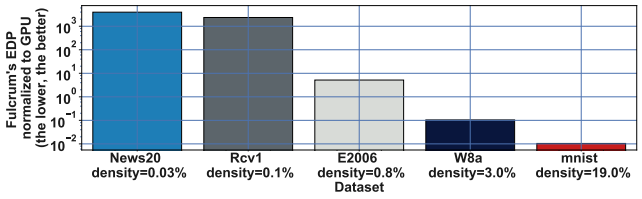


Figure 11: The effect of density on the EDP benefit

6.5 The impact of each problem

Section 3 lists four problems for prior in-situ approaches. In this Section, we discuss the impact of each problem.

1. Lack of flexibility: The fourth column of Table 1 demonstrates that DRISA [5], a prior in-situ approach, can not efficiently support applications such as Sort, Scan, FilterByKey, and FilterByPredicate. The hardware overhead of solutions such as a simple in-order core is 25× larger than the size of each subarray.

2. No support for complex operations: Figure 2 illustrates that the energy consumption of several row activations, required for emulating complex operations using bitwise operations, is higher than the energy consumption of accessing the same data in the logic layer, nullifying the advantages of in-situ computing over in-logic layer computing.

3. Interleaving: Without reuniting interleaved bits, computation on more than 4-bit values is impractical, as bits are not physically adjacent (Section 4).

4. Inefficient interconnections: Figure 12 illustrates the overhead of copying shared values. In this figure, the y-axis shows the percentage of Fulcrum's execution time that would be spent on copying shared values in all subarrays vs. the percentage of Fulcrum's time that is spent on broadcasting. Since broadcasting is often overlapped with computation, it imposes negligible performance overhead.

Figure 13 shows the performance overhead of inter-subarray data movement through GDLs. This figure shows that LISA data movement can alleviate the inter-subarray data movement overhead and improve the performance of applications with inter-subarray data movement requirement such as Scan, Sort, and KNN.

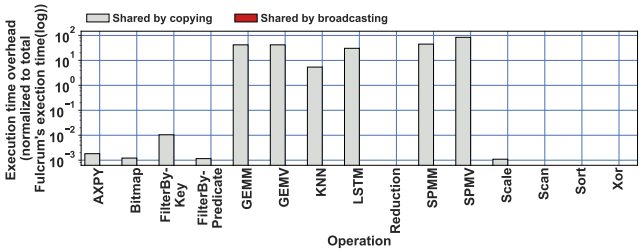


Figure 12: The performance overhead of copying shared values vs. the performance overhead of broadcasting. Since broadcasting and computation are often overlapped, broadcasting imposes zero/negligible overhead.

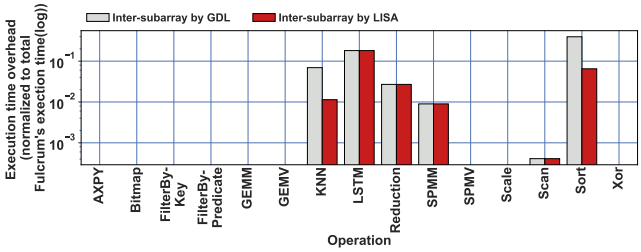


Figure 13: The performance overhead of inter-subarray data movement through GDL vs. LISA [14] for applications with inter-subarray data movement requirement.

Table 4: Area evaluation of Fulcrum

Component	Area mm^2			
	Per two subarrays		Per layer	
	Optimized	Pessimistic	Optimized	Pessimistic
Original DRAM	–	–	–	34.95
Walkers	–	0.011	–	11.26
Integer+Bitwise ALPU (4 entries)	0.0054	0.0076	5.53	7.87
Integer+Bitwise ALPU (8 entries)	0.0059	0.0083	6.09	8.51
Integer+Bitwise+Float ALPU(4 entries)	0.0088	0.0166	9.05	17.09
Integer+Bitwise+Float ALPU(8 entries)	0.0093	0.0173	9.52	17.73

6.6 Area evaluation

A high capacity accelerator is desirable as it can support applications with large footprints. Therefore we targeted an 8GB accelerator with eight layers (1GB per layer). Fulcrum has two major types of components added to the commodity DRAM: *(i)* Walkers and *(ii)* ALPUs. Table 4 lists the optimistic and pessimistic area evaluation of these components with different configurations. Our evaluation shows that an 8GB integer Fulcrum, with 4 entries of the instruction buffer, is achievable by eight layers, where the area of each layer optimistically (pessimistically) is 51.74mm^2 (54mm^2).

A 4-entry, 8GB float Fulcrum is achievable by eight layers, where the area of each layer optimistically (pessimistically) is 55.26mm^2 (63.3mm^2).

6.7 Energy consumption

Figure 14 compares the energy consumption of two configurations of Fulcrum: (i) integer Fulcrum, and (ii) float Fulcrum to GPU. This Figure illustrates the energy consumption spent (both dynamic and static) on three parts: (i) data movement (on on-chip and off-chip memory elements and interconnections), (ii) control (instruction fetch units and instruction schedulers), and (iii) computation (ALUs and FPUs). Fulcrum reduces the total energy consumption, compared to GPU, on average by 96%. Our evaluation shows that float Fulcrum reduces the energy consumption of movement, control, and computation by 97%, 73%, and 24%, respectively. Unlike the energy reduction in control and data movement (which are expected), the energy reduction in computation is unexpected as Fulcrum uses larger technology size than GPU for computation. Our evaluation shows that the dominant factor in energy reduction, for memory-intensive applications (where the computation units are often waiting for the data), is the reduced execution time, which reduces the static power consumption. For computation-intensive applications such as GEMM, Fulcrum increases the computation energy by 509%, as expected.

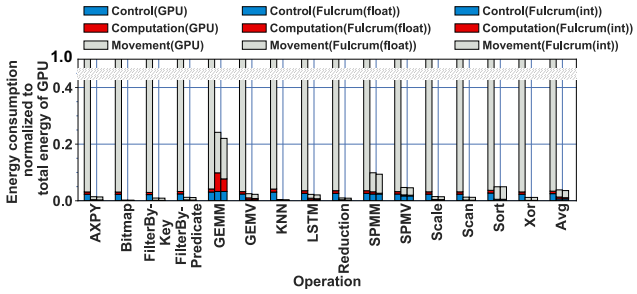


Figure 14: Breakdown of energy consumption

6.8 Power evaluation

Figure 15 compares the power consumption of GPU, float Fulcrum, and integer Fulcrum. This figure illustrates that, on average, float Fulcrum (integer Fulcrum) decreases the power consumption by 72.3% (73.8%). Our detailed evaluation shows that 33.2% percent of the power consumption of float Fulcrum is spent on row activation, 34.71% is spent on moving data to the side of the subarray, 13.1% is spent on computation, and 14.6% is spent on control. The rest is spent other forms of data movement such as broadcasting, LISA movement, and collecting the partial results.

6.9 Performance under power budget

The power budget of any accelerator directly affects the choice of interface and cooling system. Prior works [4, 22] suggest that a power budget of 10 Watts is practical through current 3D-stacked memory interfaces. With a higher power budget, deployment as a PCIe/CXL peripheral is required to deliver the required power for the accelerator and more complex and, consequently more expensive cooling system is required. To change the power consumption of Fulcrum,

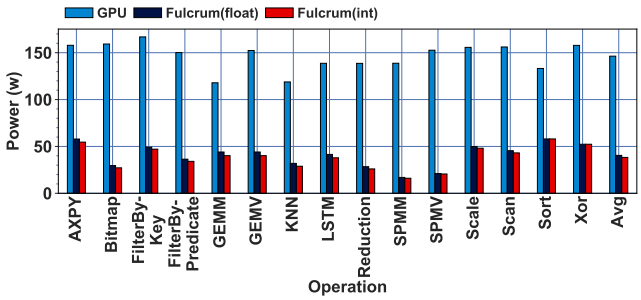


Figure 15: Power consumption of integer Fulcrum and float Fulcrum vs. GPU

we can simply change the frequency (however, we do not increase the frequency beyond 164 MHz, which we treat as our maximum frequency). Figure 16 illustrates the throughput of Fulcrum (normalized to GPU) under three power budgets: 10, 40, and 60 Watts. This Figure shows that even with the power budget of 10 watts, Fulcrum (one stack) outperforms a high-performance GPU (with three stacks of memory) by 6× on average, and up to 25×. However, under this power budget, Fulcrum slows down GEMM and SPMM.

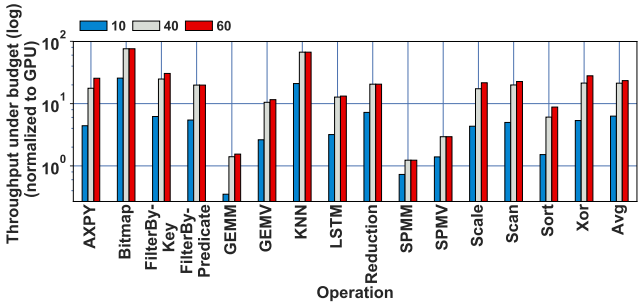


Figure 16: Performance under different power budgets

7. RELATED WORK

Traditional solutions for reducing the cost of data movement have limited benefits. Prefetching techniques can not alleviate the energy cost of data movements. Forwarding on-chip blocks techniques [45] are only applicable when cores share values. New techniques such as quantized memory hierarchies [46] mostly benefit applications that can tolerate errors. Therefore, in this section, we discuss only prior works with processing units near memory elements. We categorize these works into three groups. The first group only supports bitwise operations. The second group uses the analog computing capability of memory cells of some of the NVM technologies such as multi-bit memristors. Finally, the third group places flexible cores in the DRAM layer. This Section discusses how Fulcrum differs from such approaches.

In-situ computing with bitwise operations

SRAM-based [47] and DRAM-based [5, 6, 21, 48] in-situ accelerators often support only bitwise operations. An NVM-based in-situ accelerator, Pinatubo [38], also proposes to change the sense amplifiers to enable bitwise operation for NVMs. A limited number of applications can benefit from bitwise operation. Recent works [5, 49] implemented binarized and 2-bit quantized deep neural networks using in-situ accelerators. In most memory technologies, employing row-wide complex ALUs imposes a significant hardware overhead. Li

et al., [5] evaluated the overhead of adding a 4-bit adder per every four bits of the row buffer and concluded that it imposes more than 100% area overhead. Our proposed method has two advantages over these works. First, we enable 32-bit complex operations such as multiplication and addition without imposing a substantial hardware overhead. Second, our proposed method increases the flexibility of accelerators to support a wider range of applications.

Complex operations using analog computation capability of memory cells

Several prior works [8, 9, 50] employed the analog computation capability of ReRAM technology to perform matrix-vector multiplication. Our proposed method differs from such methods in three aspects. First, these technologies are memory-technology dependent and often use multi-bit memristor devices, which are unreliable. These techniques are neither applicable to SRAM/DRAM, nor commercialized NVM memories [51] such as 3DXpoint [52]. Second, they require ADC/DAC blocks that impose significant hardware overhead (98% of the total area) and power overhead (89% of the total power consumption) [51]). Third, performing multiplication and addition operations by approximately measuring the current, introduces potential imprecision. A recent work, FloatPIM [51] designs a CNN accelerator for training by enabling floating-point matrix-vector multiplication in memory blocks, without requiring ADC/DAC. To this end, this approach copies the vector (which is the shared value), not only in each subarray but per each row of the matrix, to enable parallel multiplication and addition, imposing capacity and energy overhead. Furthermore, it implements addition and multiplication using multiple bitwise operations and depends on the computation capability of memristors. More importantly, they change the entire memory architecture and interconnection, and as a result, the memory cannot be efficiently be accessed as a normal memory.

Flexible cores in DRAM layers

In the first round of research on processing in memory, in the 1990s, a variety of works [53, 54, 55, 56, 57, 58]) proposed to add flexible cores per each bank or per entire DRAM chip. Some of these proposals [59] add one processor per entire DRAM, plus multiple bank-level buffers. In these proposals, each column of these buffers moves toward the single processor. This imposes a high cost of data movement, is not scalable for modern DRAMs with many banks and subarrays, and limits parallelism. Our buffers enable sequential word-level access with subarray-level parallelism. Recently UPMEM [60] has described a product with a complex core per each bank of 64 MB. Fulcrum instead adds a simple processing unit per 1 MB and consequently provides higher parallelism and imposes less overhead for data movement (Our evaluations show that the energy consumption of accessing data at the edge of a bank through GDLs is at least 3× as much as that of a floating-point addition.). More importantly, as Section 4.1 explains, traditional cores impose a significant overhead of control and access for sequential operations. This overhead might be acceptable for far-memory cores for two reasons. First, for these cores, the ratio of this overhead compared to the overhead of data movement is negligible. Second, these cores use the small technology size and have a high frequency, whereas cores in DRAM layers have to use large technology size and have a low frequency. Consequently, the overhead of the extra cycles for such control and access mechanism becomes significant for in-memory cores. Fulcrum accelerates sequential operations and reduces the overhead of access and control.

8. CONCLUSION AND FUTURE WORKS

For memory-intensive tasks, data movement dominates computation. Keeping computations close to the subarray's row buffer avoids these data-movement overheads, while simultaneously enabling high throughput, thanks to subarray-level parallelism. Fulcrum overcomes key limitations of prior in-situ architectures, by placing a scalar, full-word processing unit at the edge of each pair of subarrays. We show that sequentially processing a row (instead of bit-parallel processing of the entire row buffer) with full-word computation ability allows a much wider range of tasks to leverage in-situ processing, such as a full range of arithmetic operations, key sparse and dense linear algebra tasks, operations with data dependencies, operations based on a predicate, scans and reductions, and so forth. This significantly broadens the market for an accelerator for memory-intensive processing. Leveraging DRAM technology as the basis for in-situ processing also enables high total data capacity and high total parallelism, thanks to the large number of subarrays. Our proposed method provides, one average (up to), 70 (228)× speedup per memory stack over a server-class GPU.

So far we discussed the challenges regarding implementing our method for DRAM. However we believe the same simplified control and access mechanisem can be employed for SRAM-based and NVM-based accelerators.

Fulcrum with SRAM-based memory technologies can benefit from the high frequency and more efficient logic of SRAM technologies. Recent works have utilized SRAM for in-situ pattern matching and have shown at least two orders of magnitude higher throughput per unit area than a prior in-DRAM solution [61, 62, 63]. More importantly, due to the flexibility of SRAM-based designs, we can include memory elements with efficient random access, which can support a broader range of applications. However, due to the lower capacity of SRAM, the number of subarrays are lower than DRAM, limiting parallelism. Furthermore, the current structure of SRAM-based memories is less suitable for Fulcrum as the row buffers are shorter. Therefore, a future work can optimize SRAM structures for employing Fulcrum.

NVMs have a higher capacity and higher number of subarrays and, hence, it can provide higher parallelism. Current NVM-based in-situ approaches use the computation capability of memory cells. This type of in-situ computing has three problems: (i) the ADC and DAC impose a significant hardware overhead, (ii) it is not flexible, and (iii) it introduces error, and the error depends on the number of rows in each subarray, limiting the size of each subarray and limiting the capacity. A future work can evaluate Fulcrum for NVM and investigate the challenges such as low endurance.

Acknowledgments

We would like to thank the anonymous reviewers for their valuable feedback and suggestions. This work was supported

in part by CRISP, one of six centers in JUMP, a Semiconductor Research Corporation (SRC) program, sponsored by MARCO and DARPA.

9. REFERENCES

[1] M. Ferdman, A. Adileh, O. Kocberber, S. Volos, M. Alisafaee, D. Jevdjic, C. Kaynak, A. D. Popescu, A. Ailamaki, and B. Falsafi, "Clearing the clouds: A study of emerging scale-out workloads on modern hardware," in *Proceedings of the 17th International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 37–48, 2012.

[2] S. Kanev, J. P. Darago, K. Hazelwood, P. Ranganathan, T. Moseley, G.-Y. Wei, and D. Brooks, "Profiling a warehouse-scale computer," in *Proceedings of the 42Nd Annual International Symposium on Computer Architecture*, pp. 158–169, 2016.

[3] S. Han, X. Liu, H. Mao, J. Pu, A. Pedram, M. A. Horowitz, and W. J. Dally, "EIE: Efficient inference engine on compressed deep neural network," in *Proceedings of the 43rd International Symposium on Computer Architecture*, pp. 243–254, 2016.

[4] D. Zhang, N. Jayasena, A. Lyashevsky, J. L. Greathouse, L. Xu, and M. Ignatowski, "TOP-PIM: Throughput-oriented programmable processing in memory," in *Proceedings of the 23rd International Symposium on High-performance Parallel and Distributed Computing*, pp. 85–98, 2014.

[5] S. Li, D. Niu, K. T. Malladi, H. Zheng, B. Brennan, and Y. Xie, "DRISA: A dram-based reconfigurable in-situ accelerator," in *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 288–301, 2017.

[6] 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 *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 273–287, 2017.

[7] 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 *Proceedings of the 51st Annual IEEE/ACM International Symposium on Microarchitecture* , pp. 696–709, 2018.

[8] 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 *Proceedings of the 43rd International Symposium on Computer Architecture*, pp. 27–39, 2016.

[9] 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 *Proceedings of the 43rd International Symposium on Computer Architecture*, pp. 14–26, 2016.

[10] 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 *Proceedings of the 43rd International Symposium on Computer Architecture*, 2016.

[11] R. Hadidi, B. Asgari, B. A. Mudassar, S. Mukhopadhyay, S. Yalamanchili, and H. Kim, "Demystifying the characteristics of 3d-stacked memories: A case study for hybrid memory cube," in *Proceedings of the IEEE International Symposium on Workload Characterization* , pp. 66–75, 2017.

[12] D. Fujiki, S. Mahlke, and R. Das, "In-Memory Data Parallel Processor," in *Proceedings of the 23rd International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 1–14, 2018.

[13] Y.-B. Kim and T. W. Chen, "Assessing merged dram/logic technology," *Integration*, vol. 27, no. 2, pp. 179–194, 1999.

[14] 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 *Proceedings of the IEEE International Symposium on High Performance Computer Architecture* , pp. 568–580, 2016.

[15] "Compute Express Link." `https://www.computeexpresslink.org/`.

[16] A. Farmahini-Farahani, S. Gurumurthi, G. Loh, and M. Ignatowski, "Challenges of high-capacity dram stacks and potential directions," in *Proceedings of the Workshop on Memory Centric High Performance Computing*, pp. 4–13, 2018.

[17] D. Lee, S. Ghose, G. Pekhimenko, S. Khan, and O. Mutlu, "Simultaneous multi-layer access: Improving 3d-stacked memory bandwidth at low cost," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 12, no. 4, pp. 1–29, 2016.

[18] T. Zhang, K. Chen, C. Xu, G. Sun, T. Wang, and Y. Xie, "Half-DRAM: a high-bandwidth and low-power DRAM architecture from the rethinking of fine-grained activation," in *Proceedings of the ACM/IEEE 41st International Symposium on Computer Architecture* , pp. 349–360, 2014.

[19] "CACTI-3DD: Architecture-level modeling for 3d die-stacked dram main memory,"

[20] D. Lee, Y. Kim, V. Seshadri, J. Liu, L. Subramanian, and O. Mutlu, "Tiered-latency DRAM: A low latency and low cost DRAM architecture," in *Proceedings of the IEEE 19th International Symposium on High Performance Computer Architecture* , pp. 615–626, 2013.

[21] V. Seshadri, Y. Kim, C. Fallin, D. Lee, R. Ausavarungnirun, G. Pekhimenko, Y. Luo, O. Mutlu, P. B. Gibbons, M. A. Kozuch, *et al.*, "RowClone: Fast and energy-efficient in-DRAM bulk data copy and initialization," in *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 185–197, 2013.

[22] M. Drumond, A. Daglis, N. Mirzadeh, D. Ustiugov, J. Picorel, B. Falsafi, B. Grot, and D. Pnevmatikatos, "The mondrian data engine," in *Proceedings of the ACM/IEEE 44th Annual International Symposium on Computer Architecture* , 2017.

[23] "FULCRUM ALPU." `https://github.com/MarziehLenjani/FULCRUM_ALPU`.

[24] S. Che, J. W. Sheaffer, and K. Skadron, "Dymaxion: Optimizing memory access patterns for heterogeneous systems," in *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, p. 13, ACM, 2011.

[25] "cuBLAS." `https://docs.nvidia.com/cuda/cublas/index.html`.

[26] "cuSPARSE." `https://docs.nvidia.com/cuda/cusparse/index.html`.

[27] "Thrust." `https://docs.nvidia.com/cuda/thrust/index.html`.

[28] "Profiler User's Guide." `https://docs.nvidia.com/cuda/profiler-users-guide/index.html`.

[29] "A benchmark suit for In-situ computing." `https://github.com/MarziehLenjani/InSituBench`.

[30] "CUDA Code Samples." `https://developer.nvidia.com/cuda-code-samples`.

[31] "knn." `https://github.com/vincentfpgarcia/kNN-CUDA`.

[32] V. Garcia, E. Debreuve, and M. Barlaud, "Fast k nearest neighbor search using GPU," in *2008 IEEE Computer Society Conference on Computer Vision and Pattern Recognition Workshops*, pp. 1–6, 2008.

[33] "DBMS Bitmap Indexing." `https://www.geeksforgeeks.org/dbms-bitmap-indexing/`.

[34] "Bitmap Indexes." `https://docs.oracle.com/cd/A87860_01/doc/server.817/a76994/indexes.htm`.

[35] "Bitmap Index vs. B-tree Index: Which and When?." `https://www.oracle.com/technetwork/articles/sharma-indexes-093638.html`.

[36] K. Wu, "FastBit: An efficient indexing technology for accelerating data-intensive science," in *Journal of Physics: Conference Series*, p. 556, 2005.

[37] S. Beamer, K. Asanović, and D. Patterson, "Direction-optimizing breadth-first search," *Scientific Programming*, pp. 137–148, 2013.

[38] 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 *Proceedings of the 53rd Annual Design Automation Conference*, p. 173, 2016.

[39] J. Appleyard, "Optimizing recurrent neural networks in cudnn 5." https://devblogs.nvidia.com/optimizing-recurrent-neural-networks-cudnn-5/.

[40] "MoveProf: integrating NVProf and GPUWattch for extracting the energy cost of data movement." https://github.com/MarziehLenjani/MoveProf.

[41] J. Leng, T. Hetherington, A. ElTantawy, S. Gilani, N. S. Kim, T. M. Aamodt, and V. J. Reddi, "GPUWattch: Enabling energy optimizations in GPGPUs," in *Proceedings of the 40th Annual International Symposium on Computer Architecture*, pp. 487–498, 2013.

[42] "Data Sheet: Tesla P100." https://images.nvidia.com/content/tesla/pdf/nvidia-tesla-p100-PCIe-datasheet.pdf.

[43] M. S. Mohammadi, T. Yuki, K. Cheshmi, E. C. Davis, M. Hall, M. M. Dehnavi, P. Nandy, C. Olschanowsky, A. Venkat, and M. M. Strout, "Sparse computation data dependence simplification for efficient compiler-generated inspectors," in *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2019.

[44] K. Hegde, H. Asghari-Moghaddam, M. Pellauer, N. Crago, A. Jaleel, E. Solomonik, J. Emer, and C. W. Fletcher, "Extensor: An accelerator for sparse tensor algebra," in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, ACM, 2019.

[45] M. Lenjani and M. R. Hashemi, "Tree-based scheme for reducing shared cache miss rate leveraging regional, statistical and temporal similarities," *IET Computers & Digital Techniques*, vol. 8, no. 1, pp. 30–48, 2014.

[46] M. Lenjani, P. Gonzalez, E. Sadredini, M. A. Rahman, and M. R. Stan, "An Overflow-free Quantized Memory Hierarchy in General-purpose Processors," in *Proceedings of the IEEE International Symposium on Workload Characterization*, 2019.

[47] S. Aga, S. Jeloka, A. Subramaniyan, S. Narayanasamy, D. Blaauw, and R. Das, "Compute caches," in *Proceedings of the International Symposium on High Performance Computer Architecture*, pp. 481–492, 2017.

[48] V. Seshadri, K. Hsieh, A. Boroum, D. Lee, M. A. Kozuch, O. Mutlu, P. B. Gibbons, and T. C. Mowry, "Fast bulk bitwise AND and OR in DRAM," *IEEE Computer Architecture Letters*, pp. 127–131, 2015.

[49] Q. Deng, L. Jiang, Y. Zhang, M. Zhang, and J. Yang, "DrAcc: a DRAM based accelerator for accurate CNN inference," in *Proceedings of the 55th Annual Design Automation Conference*, p. 168, 2018.

[50] M. N. Bojnordi and E. Ipek, "Memristive boltzmann machine: A hardware accelerator for combinatorial optimization and deep learning," in *Proceedings of the IEEE International Symposium on High Performance Computer Architecture*, pp. 1–13, 2016.

[51] M. Imani, S. Gupta, Y. Kim, and T. Rosing, "FloatPIM: In-memory acceleration of deep neural network training with high precision," in *Proceedings of the 46th International Symposium on Computer Architecture*, 2019.

[52] "3D XPoint Technology." https://www.micron.com/products/advanced-solutions/3d-xpoint-technology.

[53] D. Patterson, K. Asanovic, A. Brown, R. Fromm, J. Golbus, B. Gribstad, K. Keeton, C. Kozyrakis, D. Martin, S. Perissakis, *et al.*, "Intelligent ram (IRAM): the industrial setting, applications, and architectures," in *Proceedings International Conference on Computer Design VLSI in Computers and Processors*, pp. 2–7, 1997.

[54] D. Patterson, T. Anderson, N. Cardwell, R. Fromm, K. Keeton, C. Kozyrakis, R. Thomas, and K. Yelick, "A case for intelligent RAM," *IEEE MICRO*, pp. 34–44, 1997.

[55] J. Hensley, M. Oskin, D. Keen, L.-V. Lita, and F. T. Chong, "Active page architectures for media processing," in *First Workshop on Media Processors and DSPs, 32nd Annual Symposium on Microarchitecture*, 1999.

[56] M. Oskin, D. Keen, J. Hensley, L.-V. Lita, and F. T. Chong, "Reducing cost and tolerating defects in page-based intelligent memory," in *Proceedings 2000 International Conference on Computer Design*, pp. 276–284, 2000.

[57] C. E. Kozyrakis and D. A. Patterson, "Scalable, vector processors for embedded systems," *IEEE Micro*, pp. 36–45, 2003.

[58] N. Bowman, N. Cardwell, C. Kozyrakis, C. Romer, and H. Wang, "Evaluation of existing architectures in IRAM systems," in *Workshop on Mixing Logic and DRAM, 24th International Symposium on Computer Architecture*, p. 23, 1997.

[59] A. Nowatzyk, F. Pong, and A. Saulsbury, "Missing the memory wall: The case for processor/memory integration," in *23rd Annual International Symposium on Computer Architecture (ISCA'96)*, pp. 90–90, IEEE, 1996.

[60] "UPMEM." https://www.upmem.com/.

[61] E. Sadredini, R. Rahimi, V. Verma, M. Stan, and K. Skadron, "eAP: A Scalable and Efficient In-Memory Accelerator for Automata Processing," in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 87–99, ACM, 2019.

[62] E. Sadredini, R. Rahimi, M. Lenjani, M. Stan, and K. Skadron, "Impala: Algorithm/architecture co-design for in-memory multi-stride pattern matching," in *Proceedings of The 26th IEEE International Symposium on High-Performance Computer Architecture*, IEEE, 2020.

[63] E. Sadredini, R. Rahimi, M. Lenjani, M. Stan, and K. Skadron, "FlexAmata: A universal and efficient adaption of applications to spatial automata processing accelerators," in *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, ACM, 2020.