

Compute-In-Memory on FPGAs for Deep Learning: A Review

Aman Arora
Arizona State University
aman.kbm@asu.edu

Abstract—Field-Programmable Gate Arrays (FPGAs) are increasingly recognized as an efficient platform for accelerating Deep Learning (DL) applications. This is due to their hardware-level configurability, which allows for tailored datapaths and low-precision inference capabilities. As data-intensive applications such as DL continue to grow, there is a renewed interest in Compute-In-Memory (CIM) architectures to address the bottleneck caused by moving large volumes of data between memory blocks (off-chip or on-chip) and compute units. This paper delves deep into CIM architectures designed for FPGAs, focusing on making FPGAs better accelerators of DL applications. This paper provides an overview of CIM and FPGA architecture, motivating the need to design custom FPGA architectures with CIM capability. Detailed insights into CIM operation on FPGAs and the implementation of CIM on FPGAs are presented, focusing on modifications to the memory blocks on FPGAs. Existing FPGA CIM proposals are categorized and the nuances of architecting CIM blocks for FPGAs are explained. Lastly, it outlines the current challenges that must be addressed to facilitate broader adoption of CIM architectures on FPGAs.

I. INTRODUCTION

A. Data Movement Bottlenecks in Computing

The dominant form of computer architecture is the conventional von Neumann model, which generally comprises a memory unit for storing data and instructions, as well as a processing unit where data is transferred for computation. In recent decades, applications have grown more data-intensive, making data movement a primary performance bottleneck [1]. In modern systems, the energy expended in moving data exceeds that required for computation [2]. For instance, Boroumand et al. observe that in Google’s consumer applications, including the Chrome web browser, TensorFlow Mobile, and VP9 video playback and capture engines used by YouTube, data movement accounts for an average of 62.7% of the total system energy [3]. DL applications, with billions of parameters, are particularly memory intensive.

B. Compute-In-Memory Architectures

These observations have spurred renewed interest in CIM architectures, a concept first proposed over three decades ago [4, 5]. The key idea of CIM is to integrate computational structures within or close to where data is stored, such as inside memory arrays, on memory chips, in the logic layer of 3D-stacked memories, or within memory controllers. This setup aims to minimize or eliminate data movement between where computations occur and where data is stored.

Various proposals for CIM (used interchangeably with Processing-In-Memory (PIM)) architectures exist, categorized based on the type of memory utilized. Some proposals

leverage non-volatile memories (NVM) such as Resistive RAM (ReRAM), Spin Transfer Torque Magnetic RAM (STT-MRAM) or Magnetic Tunnel Junctions (MTJs) and Ferroelectric Field-Effect Transistors (FeFETs) for PIM implementation [6–11]. For volatile memories, two primary types of PIM architectures are common: Dynamic RAM (DRAM) based [12–14] and Static RAM (SRAM) based [15–17]. The objective of DRAM-PIM is to minimize data transfers between the main memory and the processor, thus reducing energy consumption. However, DRAM-PIM architectures are less suitable for applications requiring multiple reads of a memory location due to the complexity of DRAM read operations, involving copying entire rows into row buffers. In contrast, SRAMs provide faster read times and, as on-chip memory, enable efficient reuse of data. These architectures not only aim to reduce data movement but also to enhance compute throughput of the processing chip.

C. FPGA Architecture and Challenges

FPGAs typically consist of configurable logic blocks (LBs) composed of Look-Up Tables (LUTs) and Flip-Flops (FFs). In addition to LBs, FPGAs often include hardened blocks such as Digital Signal Processors (DSPs) which are optimized for performing arithmetic operations and Block RAMs (BRAMs) for on-chip memory storage. BRAMs offer fast access times and are commonly used to store frequently accessed data. FPGAs have very flexible global routing interconnect which facilitates communication between various components. This interconnect comprises switch blocks, which are a set of switches connecting different routing tracks, and connection blocks, which connect the input and output of blocks such as LBs, DSPs, and BRAMs to the routing tracks.

FPGA architectures have historically been influenced by the applications they are built for. They are gaining significant traction as a platform for accelerating DL applications owing to their hardware-level configurability, which allows for customized datapaths and numerical bit widths, making them well-suited for low-precision inference tasks. BRAMs on an FPGA play a vital role in DL acceleration by storing operands and results on-chip, feeding the compute units with data at a very high bandwidth. However, the separation of compute units (LBs and DSPs) from storage units (BRAMs) implies data movement to feed the compute units with input data and to store the outputs back to the storage units. This significantly stresses FPGA routing resources and leads to increased power consumption. Furthermore, the reconfigurability of FPGAs, the essential ingredient that makes FPGAs unique and useful,

has a significant overhead in terms of area. For example, the area spent on routing and configuration logic is a significant contributor to the area of modern FPGAs [18]. This reduces the compute density of FPGAs.

D. Compute-Enabled Block RAMs in FPGAs

To address these challenges, adding compute capabilities in FPGA BRAMs has been proposed. Block RAMs are modified to introduce processing elements (PEs) inside them enabling computations within the memory block. Adding compute capabilities to BRAMs provides the following advantages:

- Increases the compute throughput of the FPGA, because a larger portion of the FPGA can now perform computation, thereby speeding up applications.
- Reduces data movement through global routing because more processing can happen inside the RAMs, thereby saving energy.
- Provides massive compute parallelism as the large number of BRAM bitlines can operate as SIMD lanes computing on all the bits of a memory wordline.

By introducing compute capabilities, a BRAM that could only store data now becomes a SIMD compute unit with local data storage. This paper explores the design principles for compute-enabled BRAMs in FPGAs and compares current proposals. When architecting compute-enabled BRAMs for FPGAs, the following objectives should be considered:

- The normal “memory” mode of BRAM operation should be preserved to enable backward compatibility.
- Changes to the BRAM routing interface should be minimized to avoid re-architecting the global routing.
- The area and delay overhead of adding compute capabilities should be minimized.
- The compute throughput obtained from the BRAM should be maximized.

An important consideration when architecting FPGAs is the generality-specificity trade-off. An FPGA with generic components has wide applicability, but has reduced performance when deployed for specific applications. An FPGA with application-specific components captures fewer usecases, but has better performance for those few applications. A common approach to balance generality and specificity is to architect FPGAs keeping large swaths of application domains in mind. With DL being such a large application domain, making DL-optimized FPGAs is an attractive proposition. So, when architecting compute-enabled BRAMs for DL, supporting commonly used DL computations is essential.

E. Paper Overview

This paper employs a tutorial-style approach to facilitate the understanding of recent developments in CIM-enabled FPGA architectures for DL. It provides enough depth for experts in the field to grasp the nuances of architecting new FPGA architectures for enabling CIM operations. Simultaneously, it delivers adequate breadth for novices to comprehend the foundational concepts of CIM-enabled FPGA architectures

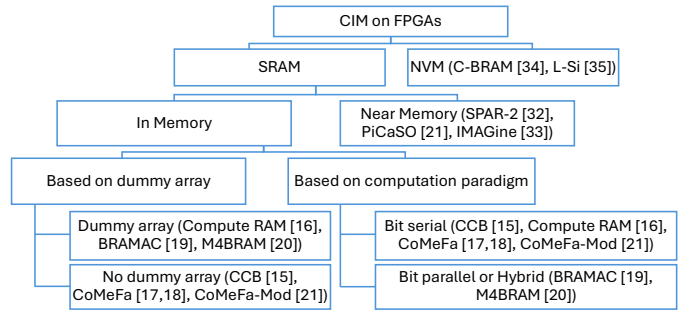


Fig. 1: Taxonomy of CIM on FPGAs

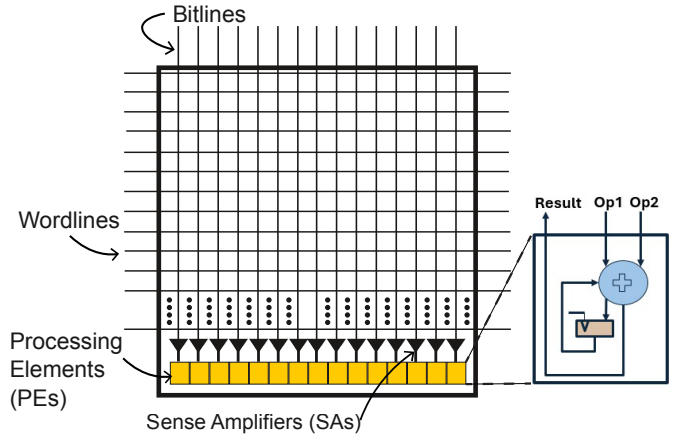


Fig. 2: Generic modified BRAM featuring PEs

while understanding the associated performance and programmability challenges. Section II provides an overview of the operating principles behind compute-enabled BRAMs. In Section III, the functioning of a compute-enabled BRAM is demonstrated using the CoMeFa RAM as an example. Section IV explores the various options available in the design space to enable CIM on FPGAs. Section V examines the trade-offs associated with the design variations discussed in Section IV. Section VI introduces alternative CIM architectures. Section VII anticipates future developments in this field and Section VIII concludes the paper.

F. Taxonomy

In this paper, we review several proposals on CIM on FPGAs. A simple high-level taxonomy is presented in Figure 1. There are two main categories - using SRAMs and using NVMs. Within SRAMs, there are proposals that are true in-memory and that are near-memory. Some approaches use a dummy array in addition to the main array in the BRAM, whereas some approaches differ in the type of processing element (PE) used (serial or parallel). In most of this paper, we discuss SRAM-based in-memory processing techniques on FPGAs. NVM-based CIM and near-memory techniques will be briefly discussed in Section VI.

II. OPERATING PRINCIPLES

At a high level, converting BRAMs to compute-enabled BRAMs requires connecting PEs to the output of the sense amplifiers inside the BRAM (as shown in Figure 2). The

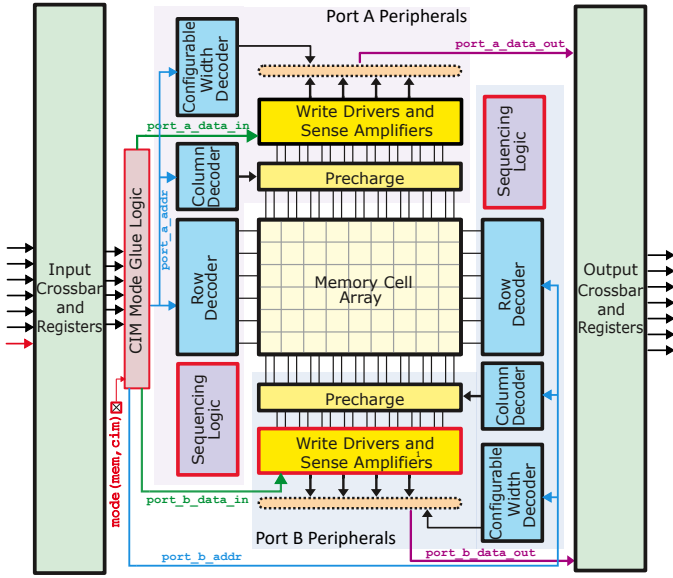


Fig. 3: Architecture of a BRAM with modifications done for CIM shown in red

vertical lines in the figure represent the bitlines used for reading and writing operations to transfer data from and to the SRAM cells. The horizontal lines represent wordlines, which, when activated, allow all cells in the respective row to connect to their corresponding bitlines for data transfer. At the end of each bitline is a sense amplifier which detects and amplifies the small voltage differences on the bitlines during read operation. PEs are added at the output of the sense amplifiers to perform computations on the read data. The architecture of a PE can vary depending on the type of computations that are being targeted. In this figure, the PE is a 1-bit adder with a flip-flop. The data to be processed (operands) is stored in the BRAM. When a computation is required to be performed, the BRAM is instructed to read the rows (wordlines) that contain the operands. The PEs receive the operands and perform the computation. Then, the results from the PEs are written back into some rows of the BRAM. All of this happens inside the RAM in an extended clock cycle. Additional control logic may be required inside the RAM to sequence these steps. The specific rows to read, the computation to perform in the PEs, and the specific row to write are provided to the BRAM through its interface - address, data, and other pins. The collection of bits that tell the RAM what to do in each cycle constitutes an instruction.

III. IMPLEMENTATION DETAILS

To illustrate the architecture and functioning of compute-enabled BRAMs, we discuss a simple implementation. The baseline BRAM considered here has a size of 20 Kbits as in modern Intel FPGAs [19], with support for single-port, simple dual-port, and true dual-port modes, with 512x40 being the shallowest and widest configuration. This BRAM has a physical geometry of 128 rows x 160 columns with a column multiplexing factor of 4. This implies that there is a set of 40 sense amplifiers and write drivers in the BRAM, for each port.

A. Modifications to the BRAM

Figure 3 shows a top-level diagram of an FPGA BRAM with blocks modified/added to incorporate compute capabilities shown with a red outline. The cells in the memory array remain unmodified. PEs are connected to the sense amplifiers and write drivers. Additional sense amplifiers and write drivers are added, enabling reading and writing a row (wordline) in all columns (bitlines) together. This results in 160 PEs in the BRAM and 160 sense amplifiers and write drivers on each port of the BRAM. This provides a parallelism 160 operations done in 1 clock cycle, whose duration is slightly longer than the baseline BRAM's clock period. Sequencing logic that sequences the events of the read/write operations (wordline activation, precharge, sense amp enable, etc.) in the memory is modified. This is done to support reading, computing, and writing in one cycle. CIM mode glue logic is also added. This logic contains the mode configuration bit, which selects whether the BRAM works in Memory mode or CIM mode. An extra interface pin (which can be multiplexed with an unused pin) is added. When this pin is asserted, the data and address are treated as an instruction. The glue logic decodes the instruction and forwards the right signals to the address, data, and control signals to the RAM peripherals. In Memory mode, the BRAM behaves as a conventional BRAM with no change in functionality. In CIM mode, the BRAM can be used for both compute and storage. In this mode, the RAM is automatically configured to its maximum width (512x40).

B. Computation Operation

Figure 4a illustrates a computation operation performed by the compute-enabled BRAM. Consider that the PEs are single-bit bit-serial adders, and the operation required to be done is to add two 4-bit arrays (A and B) of 160 elements. Each array element is stored in a column, 1 bit in 1 row. This is called a transposed layout. Elements of array 1 are stored in rows i , $i+1$, $i+2$, and $i+3$. Elements of array 2 are stored in rows j , $j+1$, $j+2$, and $j+3$. A total of 8 rows and 160 columns are required to store both input arrays. A single clock cycle has 3 phases during computation as shown in Figure 4b. In the first phase, rows i and j are read, one on each port, by precharging the bitline, activating the Read Wordline (RWL) signal for the rows i and j and finally enabling the Sense Amplifier (SA) at the end of each bitline. In the second phase, each PE computes the sum of the two bits (one from row i and one from row j) and the carry from the previous cycle (stored in the flip-flop in the PE). In the third phase, carry-out is stored in the PE, and the sum is written to row k using one port, by asserting the Write Wordline (WWL) signal for row k and enabling the Write Driver (WD) for the port. This process is repeated 4 times with increasing row addresses, taking 4 cycles. In the fifth cycle, the carry bits in each PE are written to row $k+4$ using the second port. So, after 5 cycles, the final result (array C) is available in rows k , $k+1$, $k+2$, $k+3$, and $k+4$. This is how a BRAM can be used to compute simple operations. Complex operations can be performed by repeating simpler operations and also by changing the architecture of the PE.

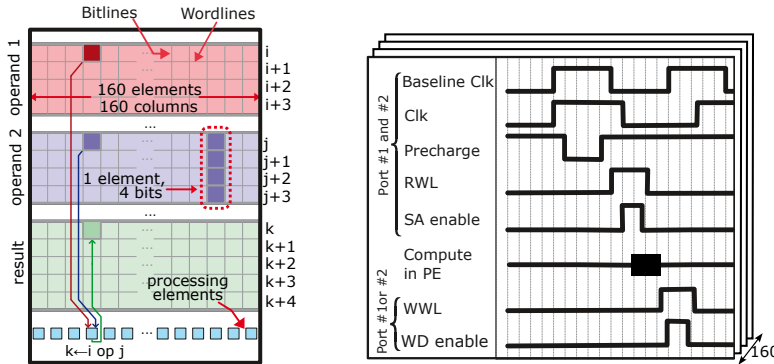


Fig. 4(a): Operation of a compute-enabled BRAM

Fig. 4(b): Sequence of operations in one clock cycle

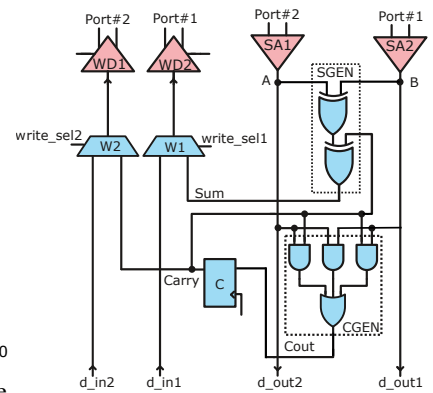


Fig. 4(c): Architecture of a PE

C. Processing Element Architecture

Figure 4c shows the architecture of this PE used in this simple implementation. On the read path, **A** and **B** are the bits of the two operands read from the memory at sense amplifiers **SA1** and **SA2** of the two ports. The two XOR gates (**SGEN**) generate the sum (**Sum**) of the input bits and the previous cycle's carry. Gates to generate the carry (**CGEN**) are also present. The carry output is stored in the carry latch (**C**) and can be used in the computation during the next cycle. The read outputs **A** and **B** are also sent to **d_out1** and **d_out2**, which is the normal read path. On the write path, 2-input multiplexers **W1** and **W2** are added before the write drivers **WD1** and **WD2** of the two ports. These multiplexers determine the sources for the write operation. **W1** can select between the **Sum** and the input data port **d_in1** (normal write operation). **W2** can select between the **Carry** and the input data port **d_in2** (normal write operation). The mux select lines **write_sel1** and **write_sel2** are driven by the CIM mode glue logic, depending on the mode setting and the instruction.

IV. IMPLEMENTATION VARIATIONS

There are several aspects of the compute-enabled BRAM architecture that can be varied to develop different architectures. Current proposals include CCB [20], Compute RAM [21], CoMeFa [22, 23], BRAMAC [24], M4BRAM [25], CoMeFa-Mod [26]. Each implementation varies some aspect of the design space discussed below.

A. Operations and precisions supported

The architecture of the PE governs the operations that can be performed in a compute-enabled BRAM. With MAC (multiply-accumulate) operations being at the heart of DL, PEs should efficiently support them. CCB and Compute RAM use a bit-serial adder-based PE, with masking and predication features. CoMeFa adds more flexibility to the PE by adding a dynamic LUT (lookup table) and allowing for shifting between neighboring PEs. Both CCB and CoMeFa support any precision, along with floating-point, although the number of cycles consumed for floating-point operations can be very high. BRAMAC uses a hybrid bit-serial and bit-parallel approach and has a precision-configurable adder as the

PE. It supports 2's complement 2-bit, 4-bit, and 8-bit MAC operations. M4BRAM enhances the BRAMAC PE by adding a duplication shuffler to allow for duplicating and shuffling one operand, enabling reuse. Mixed-precision mode is supported where one operand (weights of a DNN) can be 2, 4, or 8 bits, while the other operand (activations) can vary from 2 to 8 bits. Multiplications are performed by repeated additions in all PEs.

B. Computation paradigm

Different implementations can support different computation paradigms - Bit-Parallel, Bit-Serial, or hybrid. Bit-Parallel computing is the conventional paradigm in which multiple bits of one data element are processed every cycle. Bit-parallel PEs such as 16-bit fixed point adders or floating point multipliers can be added to RAMs [27]. However, this means that the precisions supported by the PE have to be predetermined at chip design time, thereby reducing the flexibility. Additionally, using bit-parallel PEs means restricting the location of data to be aligned to certain bitlines. Bit growth during addition and multiplication operations can cause additional challenges.

Bit serial computing, commonly used in DSP applications [28] [29], processes one bit of multiple data elements in each cycle. Adding bit-serial PEs in the RAM makes it a more generic computing unit. The PEs are agnostic to precision, which is useful for evolving applications such as DL. The main disadvantage of the bit-serial paradigm is that each operation takes many cycles, implying higher latency. However, this latency can be hidden or overlapped with other operations in data-parallel applications like DL. CCB, Compute RAM, CoMeFa, and CoMeFa-Mod use a bit-serial computing paradigm.

A hybrid approach combining Bit-Serial and Bit-Parallel paradigms is used by BRAMAC and M4BRAM. The additions are done in a bit-parallel manner using a precision-configurable adder that can take inputs from some or all bitlines. The multiplications are performed by successively/serially adding results from addition. This reduces latency significantly but has the disadvantage that the supported precisions have to be baked in at the time of designing the FPGA chip.

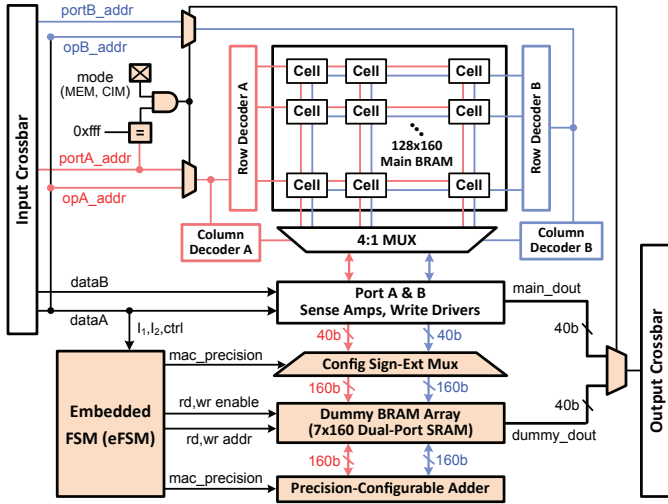


Fig. 5: BRAMAC features new circuit blocks, such as a dummy array shaded in orange (reproduced from [24]).

C. Data layout

For the bit-serial paradigm, the data must be laid out in a transposed manner (bits of an operand located in a bitline, instead of a wordline) to feed an operand into the PE one bit at a time. However, transposing the data can be costly. CoMeFa provides a swizzle unit that can be used to transpose data. This unit is implemented in soft logic and uses a significant amount of resources. 8-transistor SRAM cells can be used in BRAM to avoid transpose [30], but they have a significantly larger area than standard 6-transistor SRAM cells. This approach is also referred to as Transpose Memory Units (TMU) in [16]. Static data such as pre-trained weights of a DNN can be transposed offline, avoiding the use costly transpose units on the FPGA. During DNN evaluation, CCB and CoMeFa store pre-transposed weights in BRAMs. The bit-parallel paradigm does not require the data to be stored in a transposed format. In the hybrid computing paradigm, one operand is required to be transposed, whereas the other is not. BRAMAC and M4BRAM store pretransposed DNN weights in the BRAM (first operand), and the activations (second operand) are streamed into the BRAM untransposed.

D. Using RAM ports when computation is happening

CCB activates two wordlines at the same time on one of the BRAM's ports during computation. This means that the second port is free and can be used to read/write data from/to the RAM. This allows for overlapping the computation and data loading/unloading, thereby improving throughput. CoMeFa, on the other hand, uses both ports for computation, leaving no port available for data loading/unloading. BRAMAC and M4BRAM add an additional dummy memory array within the BRAM as shown in Figure 5. This array has the same number of bitlines as the main array, but has a few wordlines. The operands are copied to this dummy array and then computation is performed in the dummy array, leaving the main array's read and write ports available for use. BRAMAC proposes two variants - BRAMAC-2SA with two synchronous dummy

arrays, and BRAMAC-1DA with one double-pumped dummy array. M4BRAM proposes two variants - M4BRAM-S and M4BRAM-L with small and large dummy arrays.

E. Number of processing elements and sense amplifiers

The number of PEs and sense amplifiers is an architectural design choice that involves area delay trade-offs. A larger number of PEs in a RAM implies greater parallelism. One PE could be added for each bitline in the BRAM (similar to the example in Section II). This would require one sense amplifier and one write driver per bitline. This means adding additional sense amplifiers and write drivers, because BRAMs typically employ column multiplexing [31] [19] to improve the detection and correction of transient errors in memory cells, and also to reduce the number of wires to global routing in FPGAs. This method is used by CCB and CoMeFa-D (a delay-optimized variation of CoMeFa). Another alternative is to have one PE for each multiplexed column (i.e. # PEs = total number of bitlines divided by the column multiplexing factor). This does not require adding additional sense amplifiers and write drivers, but reduces parallelism. To recover parallelism, the sense amplifiers and write drivers can be used in a time-multiplexed manner, which can lead to a longer clock period. This method is used by CoMeFa-A (an area-optimized variation of CoMeFa). In BRAMAC and M4BRAM, the precision-configurable adder (the PE) is connected to the dummy memory array which does not have column multiplexing. So, each 1-bit adder receives operand bits from each bitline in the dummy array.

F. Source and method of obtaining operands

For true in-memory compute, both operands are stored in the memory array. To perform the computation, the PEs need to be provided with two operands in each cycle. There are multiple ways to achieve this. The first method, used in Computational RAM [32], involves adding flip-flops in the PE. The row containing the first operand's bits is read, and the bits are stored in the flip-flops in the PEs. The row containing the second operand's bits is read in the next cycle, and the computation is then performed. This increases the PE's area and leads to a multicycle operation, reducing performance.

Another method is based on Logic-In-Memory [33]. In this method, two wordlines containing bits of the two operands are activated at the same time. This method is used by CCB and Compute RAM. To avoid data corruption because of multirow access, the wordline voltage (and hence the frequency of operation) has to be lowered significantly. This requires changing the memory array (changing wordline voltage and sense amplifiers), requires adding an extra voltage source, has robustness issues, and is not very practical on a large scale.

CoMeFa, BRAMAC, and M4BRAM utilize the dual-portedness of BRAMs. Two bits, one from each operand, are read by the sense amplifiers on two ports and fed to the PE. Although in the logical diagram of Figure 3, the peripheral circuitry of the two ports of the RAM (decoders, write drivers, sense amplifiers, etc.) are shown in diagrammatically opposite

parts of the figure, in a typical physical layout of a RAM block, they are adjacent to each other. This ensures the practicality of adding a set of PEs fed by both sets of sense amplifiers.

CoMeFa also proposes a mode (called One Operand Outside RAM operations), where one operand is outside the memory. This can save cycles in some cases. Similarly, in BRAMAC and M4BRAM, one operand (the activations) is streamed from outside. This approach lowers CIM’s energy efficiency because a part of the data is now moving through the global routing of the FPGA. However, it is practical, because DNN activations typically have a streaming data pattern.

G. Intermediate results

When performing computations such as a dot product, intermediate results (e.g., partial sums) may need to be stored temporarily. CCB, Compute RAM, and CoMeFa use wordlines in the BRAM to store intermediate results. This reduces the effective capacity of the RAM to store the application’s data. However, BRAMAC and M4BRAM have a dummy array. Since only MAC operation is supported, specific rows of the dummy array are assigned to store specific values (including partial sums). The main array is fully available to store the application’s data.

CoMeFa-Mod adopts features from PiCaSo (detailed in Section VI) to the CoMeFa implementation to eliminate the need for scratchpad storage to copy operands for accumulation. PiCaSO’s network module can overlap computation across different PIM blocks with data movement to hide data transfer latencies. The operand-multiplexer (OpMux) provides a data path for reduction operations between the PEs without copying the operands between bitlines, thus saving both the cycles and memory during accumulation. This improves memory utilization by 6.2%.

H. Programming the BRAM

A compute-enabled BRAM needs to receive instructions to perform operations on the data. One method of doing this is to use a finite state machine (FSM) implemented in soft logic to generate instructions. This method leads to a flexible implementation, because the FSM can be customized by the user to (or hardcoded for) specific requirements of an application. However, this method is tedious because designing an FSM to generate instructions for bit-serial operations is not easy. Additionally, this leads to higher power consumption because the instructions have to be sent over global routing to the BRAM. CCB and CoMeFa use this method. Multiple BRAMs can share instruction generation logic to amortize its cost. However, doing so can increase fanout and reduce frequency. Since the FSM is outside the BRAM, some BRAM port signals are consumed to send instructions to the BRAM. This keeps BRAM port signals busy.

Another approach is to use a stored-program method. A program can be written using high-level macro-instructions and stored in another BRAM on the FPGA (like in CoMeFa) or in a small instruction memory inside the BRAM (like in Compute RAM). An instruction decoder can be designed either

in soft logic (like in CoMeFa) or hardened into the BRAM (like in Compute RAM) to convert macro-instructions to instructions. This approach makes it easy to program BRAMs.

BRAMAC and M4BRAM deploy a third approach of embedding an FSM inside the BRAM. The FSM’s functionality is fixed at the time of designing the FPGA chip. This is feasible because only one operation (MAC) is supported. The FSM is configurable to support different precisions, but otherwise it is fairly simple. Each BRAM has its own copy of the FSM in it, that is, it is not shared across BRAMs. Using a fixed FSM embedded in the BRAM frees up the port signals of the BRAM enabling it to be accessed by other parts of the application while it is computing.

I. Signalling instructions to the RAM

To send instructions to the BRAM, reusing existing ports of the BRAM is desirable to minimize the impact on the BRAM routing interface. A method needs to be devised to indicate that the data and address ports contain an instruction instead of regular data and address. CCB, CoMeFa, and BRAMAC achieve this by defining the last memory address as special. Data written to this address is treated as an instruction. This method has the limitation that the special address cannot hold data anymore. An alternative is to use a signal on the BRAM interface that when asserted indicates that the data written into the RAM be treated as an instruction (as shown in Figure 3). This signal can be one of the unused signals on the BRAM in the CIM mode (e.g., unused address bit in CoMeFa and unused write enable for a port in M4BRAM).

J. Where to store results

The result obtained from the PE can either be written back into the RAM array (like in CCB, Compute RAM, and CoMeFa), or sent out as an output of the RAM (like in BRAMAC and M4BRAM). Storing the result back into RAM enables the use of this result in the next computation in the same BRAM. However, sending it out allows it to be easily combined with the results from other BRAMs or used by the downstream blocks in the application.

V. BENEFITS AND COSTS

Adding compute capabilities to BRAMs leads to improved FPGA compute throughput, leading to significant speed-ups in DL applications. Most prior work uses an Intel Arria 10 FPGA as the baseline device and evaluates the speedups for multiple DL benchmarks on a modified FPGA where the BRAMs have been converted to compute-enabled BRAMs. CCB designs a new accelerator called RIMA (Reconfigurable In-Memory Accelerator) for DL inference on the modified FPGA. RIMA achieves $1.25\times$ and $3\times$ higher performance compared to Microsoft Brainwave [34] for 8-bit integer and block floating-point precisions across RNN, GRU, and LSTM benchmarks. CoMeFa shows a speedup of $2.55\times$ (CoMeFa-D) and $1.85\times$ (CoMeFa-A) across multiple microbenchmarks (GEMM, GEMV, Conv, etc.). A speedup of $2.5\times$ across DNNs (MLP, LSTM, GRU, Tiny Darknet, and Resnet-50)

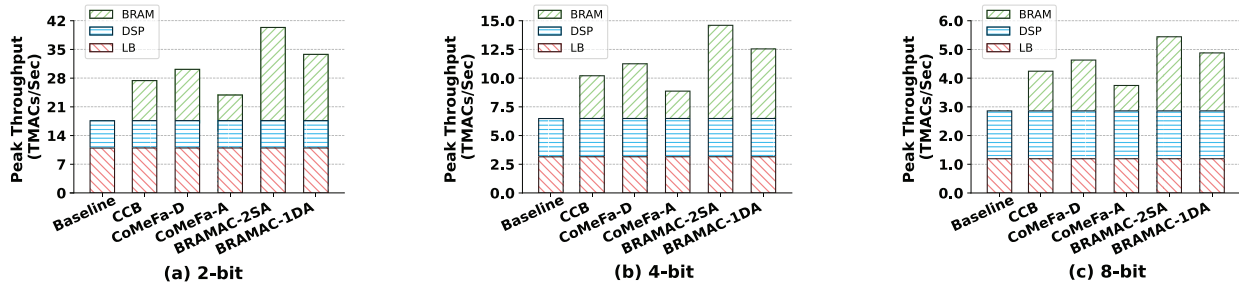


Fig. 6: Peak MAC throughput of different architectures for various MAC precisions (adopted from [24])

is also shown using a Microsoft Brainwave-like accelerator. BRAMAC shows a performance improvement of $2.04\times$ and $1.52\times$ for AlexNet and Resnet-34 using an Intel DLA-based accelerator [35]. M4BRAM outperforms BRAMAC by $1.43\times$ on average, across various DNNs. The applications demonstrated are large modern DNNs, showing the wide applicability and scalability of these architectures.

Figure 6 shows the peak MAC throughput values for CCB, CoMeFa-A, CoMeFa-D, BRAMAC-2SA, BRAMAC-1DA implementations. Although CCB and CoMeFa can compute MACs on 160 PEs parallelly, their throughput is lower than BRAMAC variants because each operation takes multiple cycles (bit-serial). The highest throughput improvement is seen on BRAMAC-2SA which improves upon the baseline Arria-10 device throughput by $2.6\times$, $2.3\times$, and $1.9\times$ for 2-bit, 4-bit, and 8-bit MAC, respectively. CoMeFa-Mod improves the MAC throughput by 19.5% compared to CoMeFa.

CoMeFa quantifies the energy reduction for various microbenchmarks as shown in Figure 7. For 8-bit integer precision, for compute-intensive benchmarks such as GEMM and GEMV, no energy reduction is observed, due to an increased power consumption resulting from the high resource usage to obtain the speedup. With a lower precision of 4-bit integer, an energy reduction of 24% is observed compared to the baseline.

The speedup and energy benefits come with some overhead. Adding compute capabilities to BRAMs adds area to the BRAM and the FPGA. The presented architectures use COFFE [36] to evaluate area and delay at the block level, and VTR [37] to evaluate area and delay at the FPGA level. The overhead in area of the BRAM ranges from 8.1% for the CoMeFa-A variation to 33.8% in the BRAMAC-2SA variation. Additionally, in most cases, the frequency of operation of the BRAM is impacted, primarily because in one cycle, read, compute, and write operations are performed. The clock period overhead ranges from 10% in BRAMAC-2SA to 150% in CoMeFa-Mod, with CoMeFa-A at 125%.

VI. ALTERNATIVE APPROACHES FOR CIM ON FPGAS

All the current proposals we have examined (CCB, Compute RAM, CoMeFa, BRAMAC, M4BRAM) focus on modifying the FPGA's BRAM architecture to integrate computation units. In contrast, SPAR-2 [38] presents a CIM overlay architecture designed to operate on commodity FPGAs. Rather than altering the BRAM, this proposal concurrently streams 16-bit wide bit-serial data from the BRAM to 16 Arithmetic

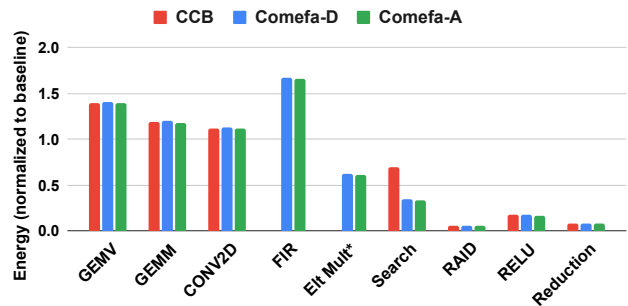


Fig. 7: Energy consumption for various microbenchmarks [23]. An asterisk (*) implies no DRAM bandwidth limitation.

Logic Units (ALUs) implemented in soft logic. SPAR-2 CIM blocks consist of a 4×4 PE grid, equipped with North-East-West-South (NEWS) network capabilities, thereby creating a two-dimensional array of SIMD bit-serial processors at the system level. SPAR-2 allows different ML networks to be quickly compiled and run on the overlay without resynthesis. SPAR-2 was deployed on Virtex-7 and Virtex UltraScale FPGAs, utilizing 10,000 PEs to enhance the performance of DNN applications like MLP, LSTM, GRU, and CNN. It delivered speed enhancements of up to 34.2 times and 3.5 times over other custom HLS-based and RTL-based accelerators, respectively. PiCaSO [26] extends the capabilities of the SPAR-2 PIM processor array by introducing support for fast reduction operations among PEs. This enhancement is achieved through the integration of an operand multiplexer (Op-Mux). PiCaSO achieves up to 80% of peak throughput and demonstrates improved memory utilization ranging from 25% to 43% compared to custom design approaches discussed earlier. IMAGine [39] builds on top of PiCaSO to build PiCaSO-IM which has a simpler data movement network and block-ID-based selection logic. They achieve high frequency and high utilization, outperforming prior work by upto $3.2\times$.

Although these alternatives offer configurability, code portability, and improved programmability, they still fall short in terms of energy efficiency. This is because the data must be transferred out of the BRAM onto the global routing to reach the PEs implemented in soft logic. This data movement process incurs energy overhead, limiting the overall energy efficiency of these solutions. Therefore, although described as CIM, these approaches do not constitute true CIM; they are computing-near-memory instead.

Property	CCB [20]	Compute RAM [21]	CoMeFa-D [23]	CoMeFa-A [23]	BRAMAC-IDA [24]	BRAMAC-2SA [24]	M4BRAM-S [25]	M4BRAM-L [25]	CoMeFa-Mod [26]
Precision supported	Arbitrary	Arbitrary	Arbitrary	Arbitrary	2,4,8	2,4,8	2,4,8	2,4,8	Arbitrary
PE flexibility (operations supported)	Medium	High	High	High	Low	Low	Low	Low	High
Compute paradigm	Bit-serial	Bit-serial	Bit-serial	Bit-serial	Hybrid	Hybrid	Hybrid	Hybrid	Bit-serial
Use transposed data layout	Both operands	Both operands	Both operands	Both operands	One operand	One operand	One operand	One operand	Both operands
# of occupied BRAM ports	One	Two	Two	Two	Two	Two	One	One	Two
Allow compute-load/unload overlap	Yes	No	No	No	No	No	Yes	Yes	No
Uses dummy array (in addition to main array)	No	Yes (for instructions)	No	No	Yes (1)	Yes (2)	Yes (4)	Yes (4)	No
Requires additional sense amps	Yes	No	Yes	No	Yes	Yes	Yes	Yes	No
Activate two wordlines at the same time on one port	Yes	Yes	No	No	No	No	No	No	No
Use dual ported'ness to read operands	No	No	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Final results stored back in	Main RAM	Main RAM	Main RAM	Main RAM	Dummy RAM	Dummy RAM	Dummy RAM	Dummy RAM	Main or Neighboring RAM
Intermediate results stored in main BRAM	Yes	Yes	Yes	Yes	No	No	No	No	Yes
Forward intermediate results to other PEs	No	No	No	No	No	No	No	No	Yes
Programming the BRAM	Soft logic FSM	Hard controller	Soft logic FSM, SP Special address, or reuse EA signal	Soft logic FSM, SP Special address, or reuse EA signal	Hardened FSM	Hardened FSM	Hardened FSM	Hardened FSM	Soft logic FSM, SP Special address, or reuse EA signal
Signalling instructions	Special address	Additional pins	Additional address, or reuse EA signal	Additional address, or reuse EA signal	Special address	Special address	Reuse WE signal	Reuse WE signal	Special address, or reuse EA signal
Area overhead	16.8%	33%	25.4%	8.1%	16.9%	33.8%	19.6%	33.4%	-
Clock overhead	60%	25%	25%	125%	46%	10%	26%	26%	150%

TABLE I: Comparison of the features of various compute-enabled BRAM implementations (SP=Stored Program, EA=Extra Address, WE=Write Enable)

All the CIM implementations seen so far use SRAMs as on-chip memory. However, there have been some proposals utilizing emerging non-volatile memory (NVM) technologies such as ReRAM as BRAMs. This approach offers significant advantages such as near-zero leakage power and higher density compared to SRAMs. [40] presents a computational BRAM (C-BRAM) architecture using ReRAM and a computational density-aware operation allocation methodology, showcasing a 68% improvement in computational density over SRAMs. [41] goes a step further by introducing a reconfigurable ReRAM-based computing fabric named Liquid Silicon Monona (L-Si). L-Si employs a 2D array of identical tiles configurable into one or a combination of four modes: heavy-weight compute, light-weight compute, interconnect, and memory. Compared to SRAM-based FPGAs, L-Si demonstrates a 52.3x speedup, 81% area reduction, and 113.9x energy consumption reduction. Nevertheless, non-volatile memories face challenges with lifetime issues and are far away from productization.

VII. LOOKING INTO THE FUTURE

While these new compute-enabled BRAMs have shown significant promise in improving the acceleration of DL applications on FPGAs, there is still more work to be done. Current results are based on simulation models, but proof-of-concept via chip fabrication is essential for wider adoption. Cost and effort can be concerns in doing this. However, fabrication cost could be reduced by using multi-project wafer shuttles from

companies such as eFabless [42], and the effort in FPGA design can be alleviated by using tools such as OpenFPGA [43]. Additionally, programming these RAMs is not a simple task. Current methods either use FSMs, which are tedious to design for bit-serial operations, or employ the stored program method, which consumes additional space in BRAMs. Future efforts should focus on improving the programmability of these RAMs, possibly by developing tools that can enable programming of these RAMs directly from DL frameworks. Moreover, future FPGAs will have 3D-stacked layers of logic and memory. Studying CIM in the context of such FPGAs is an open research question.

VIII. SUMMARY AND CONCLUSION

This paper has provided a comprehensive overview of the current landscape of CIM architectures on FPGAs for DL applications. We have explored various techniques aimed at improving compute throughput and reducing data movement. Table I provides a summary of these CIM architectures. Different architectures exploit tradeoffs in metrics such as performance vs. area, generality vs. specificity, hardening features vs. soft functionality, and practical vs. cutting-edge implementation. Although significant progress has been made in accelerating DL workloads on FPGAs using CIM architectures, there are still challenges to address, such as programmability, proof-of-concept via chip fabrication, and integration with advanced memory technologies like 3D-stacked memory.

REFERENCES

- [1] Onur Mutlu, Saugata Ghose, Juan Gómez-Luna, and Rachata Ausavarungnirun. *A Modern Primer on Processing in Memory*, pages 171–243. Springer Nature Singapore, Singapore, 2023.
- [2] Mark Horowitz. Computing’s energy problem (and what we can do about it). In *2014 IEEE International Solid-State Circuits Conference Digest of Technical Papers (ISSCC)*, pages 10–14, 2014.
- [3] Amirali Boroumand, Saugata Ghose, Youngsok Kim, Rachata Ausavarungnirun, Eric Shiu, Rahul Thakur, Daehyun Kim, Aki Kuusela, Allan Knies, Parthasarathy Ranganathan, et al. Google Workloads For Consumer Devices: Mitigating Data Movement Bottlenecks. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 316–331, 2018.
- [4] Zoe Butler, Alan Murray, and Anthony Smith. *VLSI Bit-Serial Neural Networks*, page 201–208. The Kluwer International Series in Engineering and Computer Science. Springer US, Boston, MA, 1989.
- [5] Alan Murray, Anthony Smith, and Zoe Butler. Bit-Serial Neural Networks. In D. Anderson, editor, *Neural information processing systems*, volume 0. American Institute of Physics, 1987.
- [6] Ping Chi, Shuangchen Li, Cong Xu, Tao Zhang, Jishen Zhao, Yongpan Liu, Yu Wang, and Yuan Xie. PRIME: A Novel Processing-in-Memory Architecture for Neural Network Computation in ReRAM-Based Main Memory. In *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*, pages 27–39, 2016.
- [7] Mohsen Imani, Saransh Gupta, Yeseong Kim, and Tajana Rosing. FloatPIM: In-Memory Acceleration of Deep Neural Network Training with High Precision. In *Proceedings of the 46th International Symposium on Computer Architecture*, page 802–815, 2019.
- [8] Shubham Jain, Ashish Ranjan, Kaushik Roy, and Anand Raghunathan. Computing in Memory With Spin-Transfer Torque Magnetic RAM. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 26(3):470–483, 2018.
- [9] Zamshed Chowdhury, Jonathan D Harms, S Karen Khatamifard, Masoud Zabihi, Yang Lv, Andrew P Lyle, Sachin S Sapatnekar, Ulya R Karpuzcu, and Jian-Ping Wang. Efficient In-Memory Processing Using Spintronics. *IEEE Computer Architecture Letters*, 17(1):42–46, 2017.
- [10] Xunzhao Yin, Xiaoming Chen, Michael Niemier, and Xiaobo Sharon Hu. Ferroelectric FETs-Based Nonvolatile Logic-In-Memory Circuits. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 27(1):159–172, 2018.
- [11] Dayane Reis, Michael Niemier, and X. Sharon Hu. Computing in memory with FeFETs. In *Proceedings of the International Symposium on Low Power Electronics and Design, ISLPED ’18*, New York, NY, USA, 2018. Association for Computing Machinery.
- [12] Vivek Seshadri, Donghyuk Lee, Thomas Mullins, Hasan Hassan, Amirali Boroumand, Jeremie Kim, Michael A Kozuch, Onur Mutlu, Phillip B Gibbons, and Todd 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*, pages 273–287, 2017.
- [13] Shuangchen Li, Dimin Niu, Krishna T. Malladi, Hongzhong Zheng, Bob Brennan, and Yuan Xie. DRISA: A DRAM-based Reconfigurable In-Situ Accelerator. In *2017 50th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 288–301, 2017.
- [14] Nastaran Hajinazar, Geraldo F. Oliveira, Sven Gregorio, João Dinis Ferreira, Nika Mansouri Ghiasi, Minesh Patel, Mohammed Alser, Saugata Ghose, Juan Gómez-Luna, and Onur Mutlu. SIMDRAM: a framework for bit-serial SIMD processing using DRAM. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS ’21*, page 329–345, New York, NY, USA, Apr 2021. Association for Computing Machinery.
- [15] Shaizeen Aga, Supreet Jeloka, Arun Subramaniyan, Satish Narayanasamy, David Blaauw, and Reetuparna Das. Compute Caches. In *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 481–492, 2017.
- [16] Charles Eckert et al. Neural Cache: Bit-Serial In-Cache Acceleration of Deep Neural Networks. In *Proceedings of the 45th Annual International Symposium on Computer Architecture, ISCA ’18*, page 383–396. IEEE Press, 2018.
- [17] Daichi Fujiki, Scott Mahlke, and Reetuparna Das. Duality Cache for Data Parallel Acceleration. In *Proceedings of the 46th International Symposium on Computer Architecture, ISCA ’19*, page 397–410, New York, NY, USA, 2019. Association for Computing Machinery.
- [18] Xifan Tang, Edouard Giacomin, Giovanni De Micheli, and Pierre-Emmanuel Gaillardon. FPGA-SPICE: A Simulation-Based Architecture Evaluation Framework for FPGAs. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 27(3):637–650, March 2019.
- [19] Jeffrey Tyhach et al. Arria 10 Device Architecture. In *2015 IEEE Custom Integrated Circuits Conference (CICC)*, pages 1–8, 2015.
- [20] Xiaowei Wang, Vidushi Goyal, Jiecao Yu, Valeria Bertacco, Andrew Boutros, Eriko Nurvitadhi, Charles Augustine, Ravi Iyer, and Reetuparna Das. Compute-Capable Block RAMs for Efficient Deep Learning Acceleration on FPGAs. In *2021 IEEE 29th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 88–96, 2021.
- [21] Aman Arora, Bagus Hanindhito, and Lizy K. John. Compute RAMs: Adaptable Compute and Storage Blocks

- for DL-Optimized FPGAs. In *2021 55th Asilomar Conference on Signals, Systems, and Computers*, page 1156–1163, Oct 2021.
- [22] Aman Arora, Tanmay Anand, Aatman Borda, Rishabh Sehgal, Bagus Hanindhito, Jaydeep Kulkarni, and Lizy K. John. CoMeFa: Compute-in-Memory Blocks for FPGAs. In *2022 IEEE 30th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, page 1–9, May 2022.
- [23] Aman Arora, Atharva Bhamburkar, Aatman Borda, Tanmay Anand, Rishabh Sehgal, Bagus Hanindhito, Pierre-Emmanuel Gaillardon, Jaydeep Kulkarni, and Lizy K. John. CoMeFa: Deploying Compute-in-Memory on FPGAs for Deep Learning Acceleration. *ACM Transactions on Reconfigurable Technology and Systems*, 16(3):50:1–50:34, July 2023.
- [24] Yuzong Chen and Mohamed S. Abdelfattah. BRA-MAC: Compute-in-BRAM Architectures for Multiply-Accumulate on FPGAs. In *2023 IEEE 31st Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, page 52–62, May 2023.
- [25] Yuzong Chen, Jordan Dotzel, and Mohamed S. Abdelfattah. M4BRAM: Mixed-Precision Matrix-Matrix Multiplication in FPGA Block RAMs. In *2023 International Conference on Field Programmable Technology (ICFPT)*, page 69–78, December 2023.
- [26] Md Arafat Kabir, Ehsan Kabir, Joshua Hollis, Eli Levy-Mackay, Atiyehsadat Panahi, Jason Bakos, Miaoling Huang, and David Andrews. FPGA Processor In Memory Architectures (PIMs): Overlay or Overhaul? In *2023 33rd International Conference on Field-Programmable Logic and Applications (FPL)*, page 109–115, Gothenburg, Sweden, September 2023. IEEE.
- [27] R. Gauchi, V. Egloff, M. Kooli, J.-P. Noel, B. Giraud, P. Vivet, S. Mitra, and H.-P. Charles. Reconfigurable Tiles Of Computing-In-Memory SRAM Architecture For Scalable Vectorization. In *Proceedings of the ACM/IEEE International Symposium on Low Power Electronics and Design*, page 121–126, Boston Massachusetts, Aug 2020. ACM.
- [28] Aaron Landy and Greg Stitt. Serial Arithmetic Strategies for Improving FPGA Throughput. *ACM Trans. Embed. Comput. Syst.*, 16(3), jul 2017.
- [29] Aaron Landy and Greg Stitt. Revisiting Serial Arithmetic: A Performance and Tradeoff Analysis for Parallel Applications on Modern FPGAs. In *2015 IEEE 23rd Annual International Symposium on Field-Programmable Custom Computing Machines*, pages 9–16, 2015.
- [30] Jingcheng Wang, Xiaowei Wang, Charles Eckert, Arun Subramanian, Reetuparna Das, David Blaauw, and Dennis Sylvester. A 28-nm Compute SRAM With Bit-Serial Logic/Arithmetic Operations for Programmable In-Memory Vector Computing. *IEEE Journal of Solid-State Circuits*, 55(1):76–86, Jan 2020.
- [31] David Lewis, David Cashman, Mark Chan, Jeffery Chromczak, Gary Lai, Andy Lee, Tim Vanderhoek, and Haiming Yu. Architectural Enhancements in Stratix V. In *Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays, FPGA '13*, page 147–156, New York, NY, USA, 2013. Association for Computing Machinery.
- [32] D.G. Elliott, M. Stumm, W.M. Snelgrove, C. Cojocar, and R. Mckenzie. Computational RAM: Implementing Processors In Memory. *IEEE Design Test of Computers*, 16(1):32–41, 1999.
- [33] Supreet Jeloka et al. A 28 nm Configurable Memory (TCAM/BCAM/SRAM) Using Push-Rule 6T Bit Cell Enabling Logic-in-Memory. *IEEE Journal of Solid-State Circuits*, 51(4):1009–1021, 2016.
- [34] Jeremy Fowers, Kalin Ovtcharov, Michael Papamichael, Todd Massengill, Ming Liu, Daniel Lo, Shlomi Alkalay, Michael Haselman, Logan Adams, Mahdi Ghandi, Stephen Heil, Prerak Patel, Adam Sapek, Gabriel Weisz, Lisa Woods, Sitaram Lanka, Steven K. Reinhardt, Adrian M. Caulfield, Eric S. Chung, and Doug Burger. A Configurable Cloud-Scale DNN Processor for Real-Time AI. In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, page 1–14, Los Angeles, CA, June 2018. IEEE.
- [35] Mohamed S. Abdelfattah, David Han, Andrew Bitar, Roberto DiCecco, Shane O’Connell, Nitika Shanker, Joseph Chu, Ian Prins, Joshua Fender, Andrew C. Ling, and Gordon R. Chiu. DLA: Compiler and FPGA Overlay for Neural Network Inference Acceleration. In *2018 28th International Conference on Field Programmable Logic and Applications (FPL)*, page 411–4117, August 2018.
- [36] Sadegh Yazdanshenas and Vaughn Betz. COFFE 2: Automatic Modelling and Optimization of Complex and Heterogeneous FPGA Architectures. *ACM Transactions on Reconfigurable Technology and Systems*, 12(1):1–27, March 2019.
- [37] Kevin E. Murray, Oleg Petelin, Sheng Zhong, Jia Min Wang, Mohamed Eldafrawy, Jean-Philippe Legault, Eugene Sha, Aaron G. Graham, Jean Wu, Matthew J. P. Walker, Hanqing Zeng, Panagiotis Patros, Jason Luu, Kenneth B. Kent, and Vaughn Betz. VTR 8: High-performance CAD and Customizable FPGA Architecture Modelling. *ACM Transactions on Reconfigurable Technology and Systems*, 13(2):1–55, June 2020.
- [38] Suhail Basalama, Atiyehsadat Panahi, Ange-Thierry Ishimwe, and David Andrews. SPAR-2: A SIMD Processor Array for Machine Learning in IoT Devices. In *2020 3rd International Conference on Data Intelligence and Security (ICDIS)*, pages 141–147. IEEE, 2020.
- [39] Md Arafat Kabir, Tendayi Kamucheka, Nathaniel Fredricks, Joel Mandebi, Jason Bakos, Miaoling Huang, and David Andrews. IMAGine: An In-Memory Accelerated GEMV Engine Overlay. In *2024 34th International Conference on Field-Programmable Logic and Applications (FPL)*, pages 220–226, Torino, Italy, September 2024. IEEE.
- [40] Hao Zhang, Mengying Zhao, Huichuan Zheng, Yuqing

Xiong, Yuhao Zhang, and Zhaoyan Shen. Towards High-throughput Neural Network Inference with Computational BRAM on Nonvolatile FPGAs. In *2024 Design, Automation and Test in Europe Conference and Exhibition (DATE)*, 2024.

- [41] Yue Zha and Jing Li. Liquid Silicon-Monona: A Reconfigurable Memory-Oriented Computing Fabric With Scalable Multi-Context Support. *ACM SIGPLAN Notices*, 53(2):214–228, 2018.
- [42] eFabless. ChipIgnite: Rapid and Affordable Prototyping. <https://efabless.com/prototyping>. Accessed: 2025-01-10.
- [43] Xifan Tang, Edouard Giacomini, Aurélien Alacchi, Baudouin Chauviere, and Pierre-Emmanuel Gaillardon. OpenFPGA: An Opensource Framework Enabling Rapid Prototyping of Customizable FPGAs. In *2019 29th International Conference on Field Programmable Logic and Applications (FPL)*, pages 367–374, September 2019.