CoMeFa: Deploying Compute-in-Memory on FPGAs for Deep Learning Acceleration

AMAN ARORA, University of Texas, USA
ATHARVA BHAMBURKAR, BITS Pilani Goa, India
AATMAN BORDA and TANMAY ANAND, BITS Pilani, India
RISHABH SEHGAL and BAGUS HANINDHITO, University of Texas, USA
PIERRE-EMMANUEL GAILLARDON, University of Utah, USA
JAYDEEP KULKARNI and LIZY K. JOHN, University of Texas, USA

Block random access memories (BRAMs) are the storage houses of FPGAs, providing extensive on-chip memory bandwidth to the compute units implemented using logic blocks and digital signal processing slices. We propose modifying BRAMs to convert them to CoMeFa (*Co*mpute-in-*Me*mory Blocks for *F*PGA*s*) random access memories (RAMs). These RAMs provide highly parallel compute-in-memory by combining computation and storage capabilities in one block. CoMeFa RAMs utilize the true dual-port nature of FPGA BRAMs and contain multiple configurable single-bit bit-serial processing elements. CoMeFa RAMs can be used to compute with any precision, which is extremely important for applications like deep learning (DL). Adding CoMeFa RAMs to FPGAs significantly increases their compute density while also reducing data movement. We explore and propose two architectures of these RAMs: CoMeFa-D (optimized for delay) and CoMeFa-A (optimized for area). Compared to existing proposals, CoMeFa RAMs do not require changing the underlying static RAM technology like simultaneously activating multiple wordlines on the same port, and are practical to implement. CoMeFa RAMs are especially suitable for parallel and compute-intensive applications like DL, but these versatile blocks find applications in diverse applications like signal processing and databases, among others. By augmenting an Intel Arria 10–like FPGA with CoMeFa-D (CoMeFa-A) RAMs at the cost of 3.8% (1.2%) area, and with algorithmic improvements and efficient mapping, we observe a geomean speedup of 2.55× (1.85×) across microbenchmarks from various applications and a geomean speedup of up to 2.5× across multiple deep neural networks. Replacing all or some BRAMs with CoMeFa RAMs in FPGAs can make them better accelerators of DL workloads.

CCS Concepts: • **Computer systems organization → Reconfigurable computing**; **Neural networks**; • **Hardware →** *Static memory*; **Hardware accelerators**; **Reconfigurable logic and FPGAs;**

Additional Key Words and Phrases: FPGA, Processing-In-Memory, Compute-In-Memory, Block RAM, Deep Learning, Machine Learning

This research was supported by National Science Foundation (NSF) grant 1763848 and the Intel Rising Star Faculty award.
Authors' addresses: A. Arora, R. Sehgal, B. Hanindhito, J. Kulkarni, and L. K. John, The University of Texas at Austin, Electrical and Computer Engineering, 2501 Speedway, C0803, Austin, TX 78712; emails: {aman.kbm, sehgal.rish}@utexas.edu, hanindhito@bagus.my.id, jaydeep@austin.utexas.edu, ljohn@ece.utexas.edu; A. Bhamburkar, Birla Institute of Technology & Science, Pilani K K Birla Goa Campus, Department of Electrical and Electronics Engineering, NH - 17B, Zuarinagar, Goa 403726, India; email: f20190456@goa.bits-pilani.ac.in; A. Borda and T. Anand, Birla Institute of Technology and Science, VidyaVihar Campus, Department of Electrical and Electronics Engineering, Pilani, Rajasthan 333031, India; emails: {borda.aatman, tanmay.anand29}@gmail.com; P.-E. Gaillardon, Department of Electrical and Computer Engineering, 50 S. Central Campus Drive, Rm. 2110 MEB, Salt Lake City, UT 84112; email: pierre-emmanuel.gaillardon@utah.edu.

1936-7406/2023/07-ART50 $15.00
https://doi.org/10.1145/3603504

ACM Reference format:
Aman Arora, Atharva Bhamburkar, Aatman Borda, Tanmay Anand, Rishabh Sehgal, Bagus Hanindhito, Pierre-Emmanuel Gaillardon, Jaydeep Kulkarni, and Lizy K. John. 2023. CoMeFa: Deploying Compute-in-Memory on FPGAs for Deep Learning Acceleration. *ACM Trans. Reconfig. Technol. Syst.* 16, 3, Article 50 (July 2023), 34 pages.
https://doi.org/10.1145/3603504

1 INTRODUCTION

Deep Learning (DL) applications are commonplace in today's world. The ever-increasing computational demands of DL workloads have resulted in an explosion of hardware acceleration alternatives, ranging from ASICs to GPUs to FPGAs. FPGAs are well suited to the evolving needs of DL applications because they provide customizable hardware with massive parallelism, low latency, and high energy efficiency.

FPGAs contain fine-grained programmable **Logic Blocks (LBs)**, fixed-function math units (**Digital Signal Processing (DSP)** slices), and **Block Random Access Memory (BRAM)** structures that are connected via a highly configurable routing/interconnection fabric. BRAMs play a vital role by storing operands and results on-chip, feeding the compute units with data at a very high bandwidth.

The current usage paradigms of BRAMs, LBs, and DSPs pose limitations to the acceleration that can be achieved using FPGAs. The separation of compute units (LBs and DSPs) from storage units (BRAMs) implies data movement using the routing/interconnect to feed the compute units with input data and to store the outputs back to the storage units. This significantly stresses the routing resources and leads to increased power consumption.

FPGAs provide the ability to develop hardware for different precisions. This is especially important for DL applications because the precision requirements change rapidly. DSP slices, however, support a limited set of precisions. FPGA programmers end up implementing low-precision math units on LBs instead of DSPs, reducing the number of LBs available for other purposes and leaving DSPs unused.

In many large FPGAs deployed in cloud applications, hundreds of megabits of data can be stored on-chip in BRAMs, enabling fully data-resident acceleration. However, in applications where on-chip storage requirements are low (e.g., where data is streamed to the FPGA), BRAMs may be left idle. Additionally, BRAMs on FPGAs support a limited set of heights and widths. This limits the bandwidth available to the compute units because the data needs to be read out from the interface of the BRAM to programmable routing (address and data buses). Typically, a larger number of bits can be sensed per cycle inside the BRAM than can be brought out to the interface.

In this article, we solve the limitations mentioned previously by proposing to convert BRAMs on an FPGA to CoMeFa **Random Access Memories (RAMs)**. A CoMeFa RAM block enables computation within the RAM array, without transferring the data in or out of it. One-bit bit-serial configurable **Processing Elements (PEs)** are added to the output of the sense amplifiers. This transforms the BRAM into a parallel **Single Instruction Multiple Data (SIMD)** computation unit. The availability of true dual-port mode in FPGA BRAMs [24, 55] is exploited to read operands.

Computation in any precision can be easily performed in CoMeFa RAMs without any explicit hardware because it uses bit-serial compute [14]. For performing a different operation or for using a different precision, a different instruction sequence needs to be generated and applied to the CoMeFa RAM. CoMeFa RAMs reduce the dependence on routing/interconnect and hence increases the routability of the FPGA. Data movement is reduced because the computation is done in the RAM itself, thereby saving power and reducing energy. Since the data is not moved in/out of the RAM block, routing interface limitations do not restrict the available bandwidth. Instead, the

internal physical geometry of the RAM, which is wider than the interface width, governs the effective bandwidth. The compute throughput and compute density ($GOPS/mm^2$) of the FPGA is increased significantly owing to the massive parallelism that is unlocked because of the existence of numerous RAM blocks on an FPGA. When not computing, CoMeFa RAMs can still function as normal BRAMs to store data.

Although we focus on DL in this article, CoMeFa RAMs are versatile blocks that can be used in many applications. The increased compute throughput of the FPGA by adding CoMeFa RAMs can be utilized by any application that suits a SIMD execution paradigm. CoMeFa RAMs can be used in diverse parallel applications like signal and image processing, databases, compression, encoding, decoding, and so forth. Because of the bit-serial nature of the compute, CoMeFa RAMs are particularly suited for throughput-oriented latency-tolerant workloads. Workloads with low-precision compute and bitwise operations are also well accelerated using CoMeFa RAMs.

Our contributions in this article are the following:

(1) We propose compute-enabled BRAM blocks called *CoMeFa RAMs* and describe their architecture and operation.
(2) We show the versatility of CoMeFa RAMs by mapping several applications with different workload characteristics, with a special focus on DL applications.
(3) We present novel processing-in-memory hardware concepts: a configurable PE and exploiting dual-portedness of RAMs to perform computation.
(4) We present novel processing-in-memory algorithmic concepts: **One Operand Outside RAM (OOOR)** operations.
(5) We quantify the performance and energy benefits of using CoMeFa RAMs for multiple microbenchmarks and **Deep Neural Networks (DNNs)**.

An earlier and less detailed version of this work appeared at the FCCM 2022 symposium [6]. We extend that work by describing the implementation options of CoMeFa RAMs in detail, including new microbenchmarks and experiments, increasing the focus on DL, including evaluation of end-to-end neural networks, and showcasing the integration of CoMeFa RAMs in an accelerator framework. We also provide a solution for a challenge identified in the previous work—using stored programs to enhance the programmability of CoMeFa RAMs.

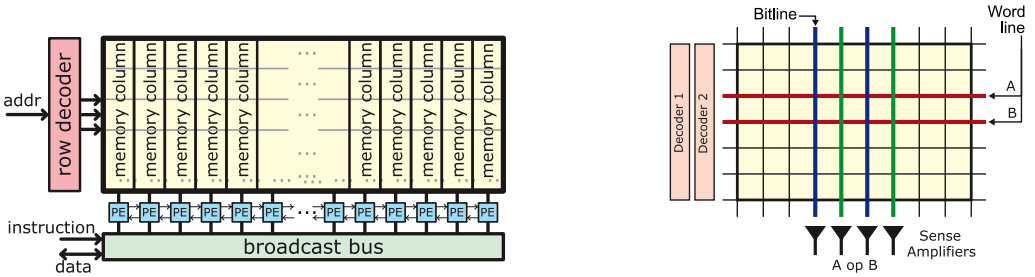
2 RELATED WORK

2.1 Compute-in-Memory

Compute-in-Memory or **Processing-in-Memory (PIM)** [20] is the paradigm of bringing computation closer to the data, instead of moving data to distant compute units. Many accelerators using PIM have been proposed and deployed: ReRAM based [13, 21, 44], **Dynamic Random Access Memory (DRAM)** based [18, 37, 43], and **Static Random Access Memory (SRAM)** based [3, 30, 31, 52].

Computational RAM (or C-RAM) [16] is an architecture where a row of PEs is added to a memory (DRAM or SRAM) to convert it into a SIMD processor, as shown in Figure 1(a). Each PE is pitch-matched with a memory column (bitline). An instruction is received by the memory from the host, operand rows (wordlines) are read and stored in the PEs, the operation is then performed, and the results are stored back into a row. All PEs in a memory execute the same instruction in a cycle. This is shown to achieve significant speedup for applications like image processing, databases, and computer-aided design, among others.

Jeloka et al. [28] created a logic-in-memory SRAM prototype shown in Figure 1(b), where multiple wordlines are activated simultaneously and the shared bitlines can be sensed, effectively performing logical AND and NOR operations on the data stored in the activated wordlines. This



(a) Architecture of Computational RAM [16]

(b) Logic-in-memory from Jeloka et al. [28]

Fig. 1. Related approaches for compute-in-memory.

technology is deployed on CPU caches to transform them into parallel processing engines [2], leading to speedups in many applications involving operations like word count, string match, and so forth. In Neural Cache, Eckert et al. [14] apply this technology to DL applications, adding PEs to the sense amplifiers and deploy bit-serial compute to perform DL operations. Wang et al. [53] proposed integrating the technology from Neural Cache into FPGA BRAMs to create **Compute Capable Block Random Access Memories (CCB)**. Speedup is shown for **Recurrent Neural Networks** (vanilla RNN, **Long Short-Term Memory (LSTM)**, and **Gated Recurrent Unit (GRN)**), for int8 and 8-bit block floating-point precisions. To avoid data corruption because of multi-row access, the wordline voltage (and hence the frequency of operation) has to be lowered significantly. An additional row decoder is required for multi-row access. Additionally, in this architecture, one sense amplifier for each pair of bitlines (BL/BLB) is replaced with two sense amplifiers (one with BL/Vref and another with BLB/Vref). The complexity associated with these changes to the memory array makes this architecture not very practical to implement on a large scale.

In this article, we propose CoMeFa RAMs, which are compute-capable SRAMs, specifically targeted for FPGAs. CoMeFa RAMs exploit the dual-portedness of FPGA BRAMs instead of activating multiple wordlines.

2.2 DL-Optimized FPGAs

The FPGA industry has deployed many DL-specific modifications to the FPGA architecture in recent years. Specialized vector processors for DL acceleration are integrated in the Xilinx Versal family of FPGAs [54]. Intel's Stratix 10 NX FPGAs have in-fabric AI tensor blocks [35]. Achronix Speedster7t FPGAs [1] have embedded machine learning processor blocks that have an array of multipliers, an adder tree, and accumulators.Recent FPGAs have also introduced native support for the `fp16` and `bfloat16` data formats in DSP slices.

Several academic research ideas to enhance FPGA architecture for DL have also been proposed. Eldafrawy et al. [15] proposed improvements to the LB architecture, including incorporating a shadow multiplier in LBs. Boutros et al. [11] suggest strengthening DSP blocks by efficiently supporting low-precision multiplications. DSP slice modifications such as including a register file for data reuse are proposed by Rasoulinezhad et al. [42]. New blocks called *Tensor slices* were proposed to be added to FPGAs by Arora et al. [5, 8].

In this article, we propose converting BRAMs on FPGAs to CoMeFa RAMs to enhance the compute throughput of FPGAs while reducing the dependency on programmable routing, to make them more efficient DL accelerators.

3 PROPOSAL: COMEFA RAMS

In this section, we describe the architecture and design of CoMeFa RAMs. We explain the changes made to BRAMs to convert them to CoMeFa RAMs. We consider a BRAM size of 20 kilobits as in

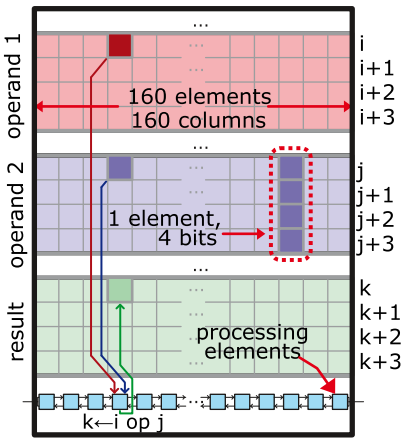


Fig. 2. High-level operation of CoMeFa RAM shown for 4-bit operands and a 4-bit result.

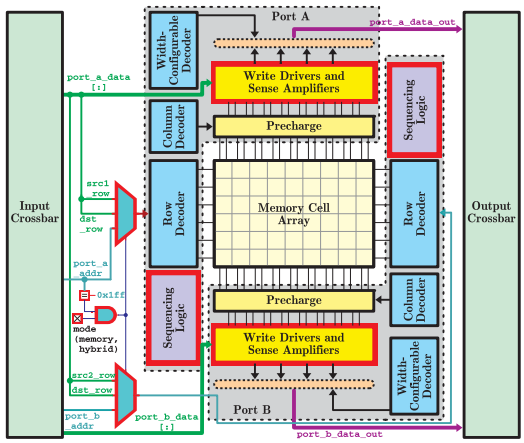


Fig. 3. Top-level logical diagram of an FPGA BRAM [57] with added/modified blocks for CoMeFa RAM highlighted in red.

the modern Intel FPGAs, with support for single-port, simple dual-port, and true dual-port modes, with 512×40 being the shallowest and widest configuration. This BRAM has a physical geometry of 128 rows $\times$ 160 columns with a column multiplexing factor of 4 [36, 49]. Even though we use BRAM sizes, geometries, and modes from Intel FPGAs, our proposal is not specific to a vendor. BRAMs from other vendors, such as Xilinx, may have slightly different sizes and geometries, but our proposal will work as long as the BRAMs support dual-port modes.

3.1 High-Level Operation

At a high level, converting BRAMs into CoMeFa RAMs requires adding PEs to the sense amplifiers inside the BRAM block, as shown in Figure 2. The architecture of a PE can vary depending on the type of computations being targeted. The PEs are fed operands by reading multiple wordlines. They perform the required computation, and the result is written back into another wordline.Note that each PE requires 2 bits (one of each operand) in one cycle. All computation is done in a bitwise manner, using a transposed data layout. Figure 2 shows how operands are stored, read, computed on, and the result stored back. Consider an example of bitwise ANDing of the elements of two arrays (array length = 160 and element width = 4 bits). Each element is stored in a column, 1 bit in one 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 eight rows and 160 columns are required to store both arrays. In one cycle, rows i and j are read, each PE computes the AND of 2 bits, and the result is stored in row k. This process is repeated four times with increasing row addresses, and the final result is available, after four cycles, in rows k, $k + 1$, $k + 2$, and $k + 3$. Note that 160 operations are done in parallel in each cycle.

3.2 Implementation Options and Changes to the BRAM

To achieve the high-level operation described previously, different aspects of the BRAM need to be modified. For each aspect, there are multiple design options. Table 1 lists the implementation options we consider. The following sections explain each aspect and our design decisions.

Our goal is to minimize the number of changes done to the BRAM to make the CoMeFa RAM design easily adoptable. Figure 3 shows a top-level diagram of an FPGA BRAM, with blocks modified/added for CoMeFa shown with a red outline. The sense amplifiers and write drivers are

Table 1. Implementation Options (the Option Used in This Work Is Presented in Bold Type)

Objective	Options
Processing paradigm	• Bit-parallel • **Bit-serial**
Obtaining two operands	• Activating one wordline and storing bits • Storing operands in separate banks • Activating two wordlines together • **Using dual-ported memory**
Number of PEs and SAs	• **# PEs = # SAs = Number of bitlines** • **# PEs = # SAs = Number of datalines** • Or something in between
Distinguishing between data and instructions	• **Write to a special address** • Add a new signal on the interface
Transposing the data	• **In soft logic** • In DRAM controller • Use RAM with transposable cells
Programming the CoMeFa RAM	• **Workload-specific state machine** • **Stored program**

SA, sense amplifier.

modified to add and connect the PEs. 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 and writing in one cycle. Some additional logic (comparator, `mode` configuration bit, multiplexers in front of row decoders) is also added. The memory array itself stays unmodified. The following sections explain each change in detail.

3.3 Processing Paradigm

There are two paradigms we can choose between to convert a BRAM into CoMeFa RAM: bit-parallel and bit-serial. Bit-parallel computing is the conventional paradigm in which multiple bits of one data element are processed every cycle. As an example, a conventional bit-parallel processor will take 128 steps to perform an elementwise sum of two arrays with 128 16-bit elements, using 16-bit PEs (adders). We could add bit-parallel PEs—for example, 16-bit fixed-point adders or floating-point multipliers—in the RAM [19]. However, this means that the precisions supported by the PE have to be pre-determined, thereby reducing the flexibility of the block. 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. This paradigm is low in utility because it will not be very different from directly connecting a BRAM and a DSP slice.

Bit-serial computing, however, is commonly used for DSP and has been used on FPGAs as well [33, 34]. The main idea is to process 1 bit of multiple data elements every cycle. For the preceding example, a bit-serial processor with 128 PEs would complete the operation of adding the two arrays in 16 steps as it processes the arrays *bit-by-bit* instead of *element-by-element*. Adding bit-serial PEs in the RAM makes the block a more generic computing unit. The PEs are agnostic to precision, which is useful for evolving applications like DL. The data has to 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 1 bit at a time. Adding bit-serial PEs to a BRAM converts the BRAM into a SIMD engine with a high vectorization width—up to 160 (in the case of Intel FPGA BRAMs that we consider) when one PE is added for each bitline. The main disadvantage of the bit-serial is that each operation takes many cycles, implying higher latency. However, this latency can be hidden/overlapped with other operations in data-parallel applications like DL. We add bit-serial PEs in CoMeFa RAMs.

3.4 Obtaining Two Operands

To perform computation, each bit-serial PE needs 1 bit from each operand. There are multiple ways to achieve this. The first method, based on Computational RAM [16], involves adding flip-flops in the PE (see Figure 1(a)). The row (wordline) 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. The results are stored back in another row in the third cycle. This increases the area of the PE, and also leads to a multi-cycle operation, reducing the speedup that can be obtained for applications.

In the second method, data can be stored such that each operand exists in a different bank of the RAM. The two banks can be accessed simultaneously. This requires the RAM to be implemented using two banks and also places a restriction on the data layout—that the two operands cannot be in the same bank.

The third method is based on logic-in-memory [28] (see Figure 1(b)). In this method, two wordlines containing bits of the two operands are activated at the same time. This needs changing the memory array, has robustness issues, and, as mentioned in Section 2, is not very practical on a large scale.

The fourth method that we propose in this work uses dual-ported RAMs. Two bits, one from each operand, are read by the two ports' sense amplifiers and fed to the two operands to the PE. This costs additional area for the second port, but FPGA BRAMs are already dual-ported, so this does not add any additional area in the case of FPGAs. 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.

3.5 Modes, Stages and Phases

As shown in Figure 3, a new configuration SRAM cell is added which decides the mode of operation of a CoMeFa RAM block. A CoMeFa RAM can operate in two modes:

— *Memory mode*: In this mode, CoMeFa RAM behaves as a conventional BRAM with no change in functionality. In this mode, the FPGA programmer can flexibly configure the number of ports and the width/depth of the BRAM.
— *Hybrid mode*: If this mode is enabled at configuration time, the CoMeFa RAM can be used for computation as well as storage. In this mode, the RAM is automatically configured to its maximum width (512×40) to maximize the read/write throughput for populating the memory array with input data and reading the results.

Operations on CoMeFa RAMs typically happen in three stages:

— *Data loading stage*: Input data is stored in a transposed format into the memory array in this stage.
— *Compute stage*: In this stage, the CoMeFa RAM is instructed to read source operand rows, perform computation in the PEs, and write the results to a destination row.
— *Data unloading stage*: Results can be read out in this stage by reading them from any address in the memory array.

A clock cycle during computation has three phases. In the first phase, two rows containing operand bits are read by activating the corresponding wordlines. In the second phase, the logic gates in the PE compute the result. In the third phase, the result is stored back by activating a wordline. This leads to a longer clock period, compared to typical BRAM.

3.6 Number of PEs and Sense Amplifiers

BRAMs typically employ column multiplexing [36, 49] for improving the detection and correction of transient errors in memory cells, and also to reduce the number of signals or wires to the programmable routing in FPGAs. The memory array layout is roughly square (the number of bitcells in the x-direction is similar to the number of bitcells in the y-direction). When reading, column multiplexers select a smaller number of cells from those along a wordline, based on the address input. When no column multiplexing is present, the number of sense amplifiers in the RAM is equal to the number of bitlines. When column multiplexing is present, the number of sense amplifiers in the RAM is equal to the number of datalines (i.e., number of data signals on the RAM interface). When adding compute-in-memory capabilities into RAMs, the number of PEs and sense amplifiers is an architectural design involving area-delay tradeoffs. For CoMeFa RAMs, we explore two architectures at the ends of the area-delay design space (evaluating other candidates in this space is future work) as discussed next.

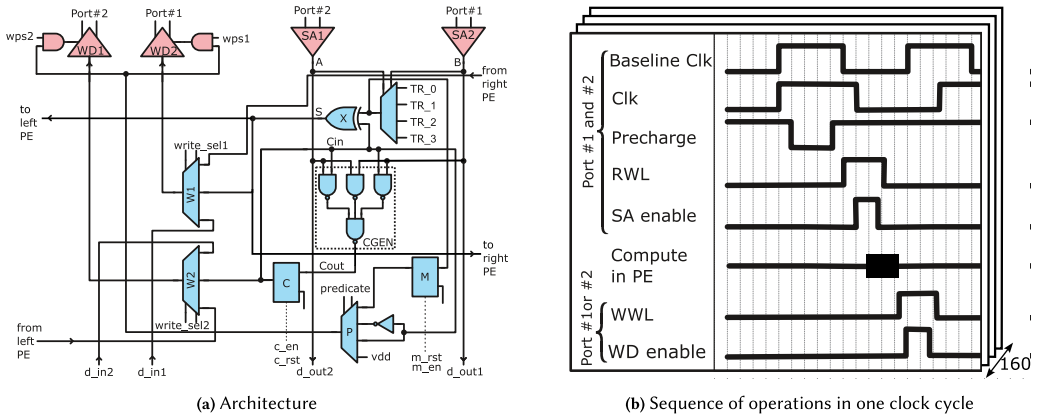
CoMeFa-D. In this architecture, we add additional sense amplifiers and write drivers to enable reading and writing a row in all columns (bitline pairs) together. A PE is added below each column. This is similar to the architecture used in other works [2, 14, 53]. During physical design/implementation, PEs should be laid out so that they pitch-match with the SRAM cells (and sense amplifiers and write drivers) for a bitline pair (BL and BLB), which can be challenging. There are 160 sense amplifiers and write drivers per port and 160 PEs. This provides a parallelism of 160 operations done in one clock cycle (slightly longer than the baseline BRAM's clock period) at the cost of high area overhead. This architecture is more practical than CCB because multiple wordlines are not activated simultaneously on a port and voltage reduction is not required for robustness.

CoMeFa-A. In this architecture, the number of sense amplifiers and write drivers stays the same as the baseline. A PE (different from the one in CoMeFa-D) is added below each multiplexed column (i.e., one PE for each dataline). An optimization technique called *sense amp cycling* [46] is employed to sequentially sense multiplexed column bits in an extended clock cycle. There are 40 sense amplifiers and write drivers per port, and 40 PEs in the RAM. This provides a parallelism of 160 operations done in one extended clock cycle, thereby trading off delay for area. This extended clock cycle is not a major concern though, since critical paths in most circuits usually include routing and LBs, and very rarely include BRAMs that can run at much higher frequencies. This architecture has the highest practicality among CCB and CoMeFa variations because it retains column multiplexing.

3.7 PE Architecture

The next aspect of converting BRAMs to CoMeFa RAMs is to identify the architecture of the PE. The PE can be a simple logic gate, in which case the CoMeFa RAM would be capable of only performing logical operations, and hence would not be very useful for DL applications. Instead, using a bit-serial adder (two inputs, 1-bit full-adder, one carry flip-flop, one output) as the PE enables arithmetic operations like addition and multiplication (by repeated addition). This is the core of the PE in the work of Eckert et al. [14] and Wang et al. [53]. Additional logic is provided for predication to enable cases where operations in some columns (or bitlines) need to be masked. The PE architecture used in CoMeFa RAM extends from this and adds additional features like configurability and BRAM-to-BRAM connections.

Figure 4(a) shows the structure of PE added to each column of the memory in CoMeFa-D. 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. Multiplexer TR evaluates a logical function of A and B, depending on the inputs TR0, TR1, TR2, TR3 (truth table). If a 2-bit addition is required, the truth table bits will correspond to that of an XOR gate. The TR mux is basically like a two-input dynamic LUT that



(a) Architecture (b) Sequence of operations in one clock cycle

Fig. 4. PE for CoMeFa-D. RWL, read wordline; SA, sense amplifier; WWL, write wordline; WD, write driver; Rd, read; Wr, write.

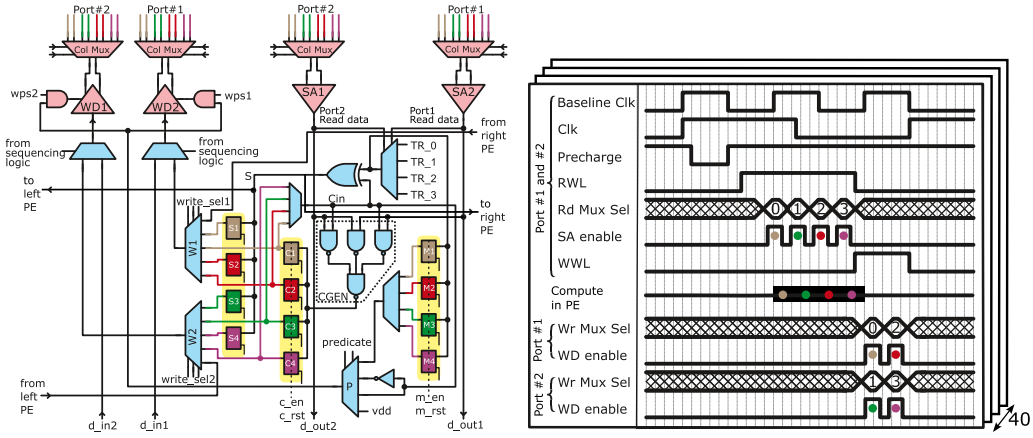
can be configured every cycle. The output of TR goes through another XOR gate (X) to generate the addition of the input bits (S), including the previous cycle's carry. Gates to generate the carry (CGEN) are also present. The carry is stored in the carry latch (C) and can be used in the following cycle's computation. If an addition operation is not required, the carry latch is reset with C_RST=1, which enables X to pass the output of TR transparently to the S wire. C_EN=0 disables the latch so it keeps the old value. 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, three-input multiplexers W1 and W2 are added before the write drivers of the two ports. These multiplexers determine the sources for the write operation. W1 can select between the S, the input data port d_in1 (normal write operation), and the value read from the right neighboring PE (used during left shift operation). W2 can select between the carry, the input data port d_in2 (normal write operation), and the value read from the left neighboring PE (used during the right shift operation).

Figure 4(b) shows a waveform view of the sequence of operations in one clock cycle for CoMeFa-D. After the bitlines are pre-charged, the read operation is performed by activating the read wordline and asserting sense amplifier enable. The PEs compute on the values read by the sense amplifier. The results are written by activating the write wordline and asserting write driver enable.

The output of multiplexer TR is also stored in a special latch called *M* and is called *mask*. Predication logic allows enabling/disabling the write drivers (WD1 and WD2). For this, a multiplexer (P) is added to select the signal that will enable/disable the write drivers. The mask, carry, not-carry, and VDD (logic 1; default) can be selected. This helps CoMeFa RAMs mask writing the results based on various conditions, like the value of the mask or the carry bit, to support multiplications and floating-point operations. The wps1/2 signals decide which port's write path is activated for a given cycle.

Figure 5(a) shows the structure of PE added to each multiplexed column of the memory in CoMeFa-A. All the labels have the same meaning as the PE described previously. The number of C and M latches changes to 4, and there are four additional latches for S. On each port, 4 column-multiplexed bits are read and two results are written back in an extended clock cycle. In the read phase of the cycle, the brown bitline pairs from each port are sensed first. The resulting S bit is stored in latch S1, carry bit C is stored in the latch C1, and mask bit M is stored in latch M1. This is repeated for red, green, and purple bitline pairs successively. All Sn, Cn, and Mn latches get updated in this process. Then, in the write phase of the cycle, results for the brown and red bitlines



(a) Architecture (b) Sequence of operations in one clock cycle

Fig. 5. PE for CoMeFa-A. RWL, read wordline; SA, sense amplifier; WWL, write wordline; WD, write driver; Rd, read; Wr, write.

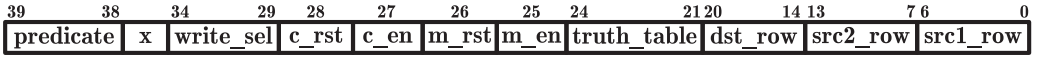


Fig. 6. Instruction format for CoMeFa RAMs.

are written using the write drivers of the two ports, followed by the green and purple ones. This is shown in Figure 5(b). Clocks in the PE are driven by signals derived from the sense amplifier enable pulses. The paths in the PE do not add any additional delay to the extended clock from sense amplifier cycling.

3.8 Instructions

An instruction is defined as a bit vector whose bits tell the CoMeFa RAM and the PE what to do in each cycle. The CoMeFa RAM instruction is 40 bits, and the format is shown in Figure 6. The field names in the instruction are self-explanatory. They directly drive the corresponding signals in the PE (e.g., `predicate` bits are applied to the select lines of the multiplexer P). The `src1_row`, `src2_row`, and `dst_row` bits are used for activating the first operand row on Port A, the second operand row on Port B, and the row at which results will be stored, respectively. These addresses are fed to the appropriate row decoders at the right time in the clock cycle by the sequencing logic in the CoMeFa RAM using the multiplexers shown in Figure 3. Instructions are generated using instruction generation logic and fed to the CoMeFa RAMs. Section 3.14 will discuss the various ways of achieving this.

3.9 Distinguishing Between Data and Instructions

As mentioned in Section 3.5, in Hybrid mode, the CoMeFa RAM can be used for computation as well as storage. So, in this mode, we need a way to distinguish between compute operations (sending instructions to the RAM) and storage operations. We consider two options for this. In the first option, a special address (0x1FF) is reserved to signal sending instructions to the RAM, similar to the work of Wang et al. [53]. Data written to this address is treated as an instruction. Accessing other addresses is done normally, used for storing operands and reading results. A comparator is added to Port A's address signal to check for this address (see Figure 3). One limitation of this

method is that the special address (0x1FF) cannot hold data anymore. This can be a problem especially when an application needs to store data on all addresses of the RAM. As a workaround, data can be written to another address first and then copied or moved to the special address internally in the BRAM using a compute instruction.

The second option we consider is to use a dedicated signal on the RAM interface that when asserted indicates that the data written into the RAM be treated as an instruction. This does not need adding an extra signal on the RAM interface because existing signals can be reused. In Hybrid mode, the BRAM is configured into its widest mode (512 × 40). So, 9 address bits are required. In other modes (like 1024 × 20 or 2048 × 10), there are more address bits (10 in 1024 × 20 and 11 in 2048 × 10), and those address bits are unused in the widest mode. We propose reusing one of these bits to denote that the data bus contains instructions.

Both methods do not have any impact on the performance of the CoMeFa RAM. The results from our evaluation are independent of which method is used.

3.10 RAM-to-RAM Chaining

CoMeFa RAMs provide the capability of performing left-shift and right-shift operations efficiently. Shifts are single operand operations. For a left (right) shift operation, the source operand row is read into the PEs, each PE's W1 (W2) mux is configured to select the bit read from the right (left) neighboring PE, and that bit is written into the destination row. For CoMeFa-A, shifting values from a bitline pair to another bitline pair within the same column multiplexer is also supported, by decoding the `write_sel` bits of the instructions and setting the select lines of W1 and W2 muxes appropriately. Direct links connecting top and bottom neighboring CoMeFa RAMs are provided to allow for shifting data between the corner PEs in each CoMeFa RAM. These connections can provide a much easier way to perform inter-CoMeFa RAM communication and obtain even more parallelism. Figure 7 shows the pins on a CoMeFa RAM required to provide these direct connections between CoMeFa RAMs, along with the details of the shift operation support inside each PE. These connections are similar to carry chains in LBs and cascade chains in DSP slices in modern FPGAs. Xilinx FPGAs have direct BRAM-to-BRAM interconnections as well. If unidirectional wiring is used, four additional pins would be required on the CoMeFa RAM to allow for shifting in both directions. To minimize this overhead, we choose to provide bidirectional wires controlled by tri-state switches, because at one time, shifting in only one direction is required. The tri-state switches are controlled by a signal decoded from the `write_sel*` and `wps*` bits, because they govern the shift direction.

3.11 Transposing the Data

As mentioned in Section 3.1, data has to be stored in a transposed layout in CoMeFa RAMs for computation. There are multiple methods that can be used for transposing data. The first method is to modify the memory array to use transposable bitcells, similar to that in the work of Wang et al. [51]. This approach is also referred to as Transpose Memory Units in the work of Eckert et al. [14]. This requires extensive modification to the RAM and also increases the area significantly, so we do not pursue this approach.

Another method is to design logic to transpose data and implement it in soft logic. We refer to this as a swizzle module (or swizzle logic) that can read data from DRAM, transpose it, and write it a CoMeFa RAM on-the-fly. The architecture of the swizzle logic, shown in Figure 8, employs a ping-pong buffer FIFO. Untransposed data read from DRAM is written in order into the ping part of the FIFO (depth = 40 elements). When the ping part is full, a transposed word (a bit slice from 40 elements) can be read and written into consecutive bitlines on the same wordline in a CoMeFa RAM, and new data from DRAM is written into the pong part. After the pong part is

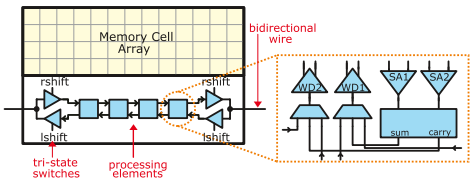


Fig. 7. CoMeFa RAM supports shifting within a block and across blocks using chaining.

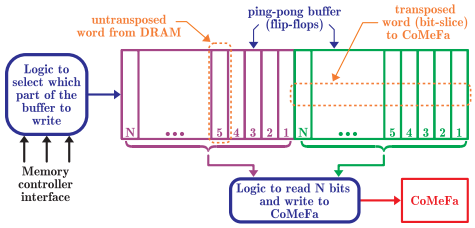


Fig. 8. Swizzle logic to load non-transposed data from DRAM directly into CoMeFa RAM in a transposed layout (N = 40).

full, transposed data is read from the pong part and written into a CoMeFa RAM, and new data from DRAM is now written into the ping part again. This process continues until the required data has been populated into CoMeFa RAMs. Similarly, transposed data can be read from CoMeFa RAMs and stored into DRAM in untransposed form using swizzle logic. We use this method in our evaluation.

The swizzle module is mapped to LBs. So, for an application that is already bound by LB usage, this can be a concern. There are multiple ways to reduce dependence on swizzle modules by avoiding the need for transpose, such as OOOR operations (Section 3.13), popcount-based addition [53], and storing pre-transposed values in CoMeFa RAMs for static data like weights of a neural network during inference.

A third method is based on the realization that transposing data is only needed when reading/writing from/to the DRAM. Modern FPGAs have hardened DRAM controllers integrated into them. So, transpose logic (like the swizzle module) could be hardened into a DRAM controller. We plan to explore this in the future. Our current baseline FPGA does not have a hard memory controller.

3.12 Variable Precision Support

Hardware in CoMeFa RAM PEs is not specific to any numerical precision. A different sequence of instructions is all that is required to compute in a different precision. The sequences for fixed-point addition, multiplication, and in-RAM reduction are the same as those in the work of Eckert et al. [14]. Addition for n-bit operands takes $n + 1$ cycles. Multiplication of n-bit operands takes $n^2 + 3n - 2$ cycles. CoMeFa RAMs can natively support floating-point precisions as well, as opposed to CCB [53]. We adopt the floating-point algorithms for addition and multiplication from FloatPIM [21]. The CoMeFa RAM PE can perform all the steps in the sequences because (1) carry and not-carry are used in the predication logic, (2) mask is populated from the output of the programmable multiplexer TR instead of just A or B, and (3) operations like XOR can be performed easily using TR and the `truth_table` fields in the instruction. The approximate number of cycles consumed for floating-point multiplication and addition are $M^2 + 7M + 3E + 5$ and $2ME + 9M + 7E + 12$, where M = number of mantissa bits and E = number of exponent bits.

3.13 OOOR Operations

In Section 3.8, two operands were stored inside the RAM. However, in many cases, an optimization can be applied—one of the operands can be outside the RAM (e.g., multiplying an array of numbers (stored in the RAM) with a scalar operand (outside the RAM)). We call these *OOOR operations*. This method saves space inside the RAM. Without OOOR, in the multiplication example, we would need to replicate the scalar operand in each column. This method allows easy inspection of outside

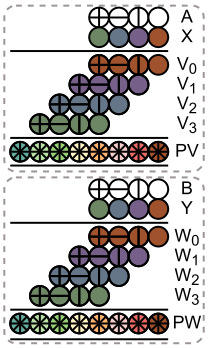


Fig. 9. Steps to perform A×X and B×Y.

ALGORITHM 1: Naive dot product

Input: A, X, B, Y
Output: Z
  $PV = 0$
  $PW = 0$
  for $i = 0$ to $precision$ **do**
    Calculate V_i
    $PW+ = (V_i << i)$
  end for
  for $i = 0$ to $precision$ **do**
    Calculate W_i
    $PW+ = (W_i << i)$
  end for
  $Z = PV + PW$

ALGORITHM 2: Intelligent dot product

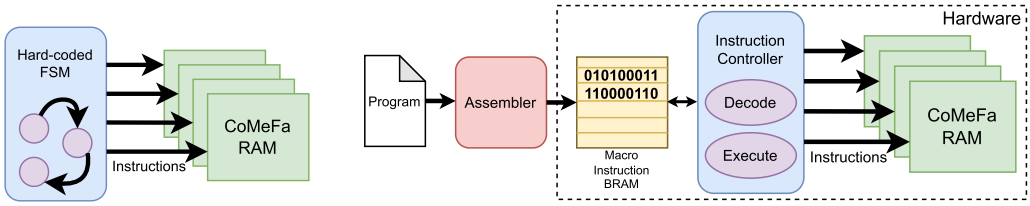
Input: A, X, B, Y
Output: Z
  Calculate $temp = A + B$
  for $i = 0$ to $precision$ **do**
    if $X_i, Y_i == 2'b11$ **then**
      $Z+ = (temp << i)$
    else if $X_i, Y_i == 2'b10$ **then**
      $Z+ = (A << i)$
    else if $X_i, Y_i == 2'b10$ **then**
      $Z+ = (B << i)$
    else $No\ Change$
    end if
  end for

operand's bits, thereby enabling efficient algorithms. For example, in the normal shift-and-add-based multiplication explained in the work of Eckert et al. [14], if a bit in the scalar operand is 0, cycles are still consumed, which can be avoided by using OOOR. In the average case, half of the bits will be 0, and therefore the number of cycles can be reduced by 50%. Efficient algorithms like Booth multiplication can also be deployed. A simple way to perform OOOR operations is to have a row of ones and a row of zeros in the RAM and use that as a proxy for bits in the operand outside the RAM. Alternatively, we can send appropriate truth table (TR) bits to the PE in CoMeFa RAMs to achieve the same goal. Overall, OOOR operations make the PEs more powerful by expressing two (or three) operand operations as one (or two) operand operations.

We apply OOOR to design an efficient algorithm for dot product operations where one of the vector's elements is common to all columns. This is useful in many applications like matrix vector multiplication and **Finite Impulse Response (FIR)** filter. Consider the case where a weight vector $[X, Y, Z, W]$ needs to be multiplied with multiple vectors $[A, B, C, D]$, $[E, F, G, H]$, ... and each vector is stored in a different column of the RAM. The weight vector does not need to be stored in the RAM but can be outside the RAM and inspected in the instruction generation logic to generate appropriate instructions. To simplify, let us see how $AX + BY$ can be calculated as the building block operation in one column. Figure 9 pictorially shows the evaluation of partial sums PV and PW. In a naive algorithm, PV will be calculated first and then PW. So, now we have both PV and PW in the same column in the RAM. They will be added to get the result. This algorithm is shown in Algorithm 1 and can be done using OOOR with X and Y being outside the RAM, and will provide a speedup of 2× on average assuming that half the bits are zeros.

In our intelligent algorithm (shown in Algorithm 2), we first calculate $A + B$ (and call it $temp$). So, now we have A, B, and $A + B$ (temp) in the column. When X and Y are outside the RAM, we can inspect bits $X[0]$ and $Y[0]$ together. If they are 11, we add $A+B$ (temp) to the result. If they are 10, we add X to the result. If they are 01, we add Y to the result. If they are 00, we do nothing. We do this successively for all bit locations of X and Y. When adding to the partial result each time, we add the correct rows to effectively do the shifts required during a normal multiplication. This algorithm provides a speedup of 2× compared to Algorithm 1, and up to 4× compared to the naive multiply-then-add algorithm.

For OOOR operations, the data outside the RAM does not need to be transposed, thereby saving some swizzle logic. There are some disadvantages to using OOOR operations as well. The instruction generation logic becomes more complex. A lesser reduction in energy consumption should be expected, because of additional control logic outside the RAM and because of higher dependency on programmable routing. Since the instruction generation logic takes different paths based on



(a) An FSM sending instructions to CoMeFa RAMs

(b) An instruction controller decoding macro-instructions and sending instructions to CoMeFa RAMs

Fig. 10. Programming CoMeFa RAMs.

the data, the opportunities of sharing it across many CoMeFa RAMs may drop depending on the application.

3.14 Programming CoMeFa RAMs

Programming a CoMeFa RAM means sending it a sequence of instructions to perform a given operation. We consider two methods for programming CoMeFa RAM. In both methods, multiple CoMeFa RAMs can share instruction generation logic to amortize its cost. However, doing so can increase the fanout and reduce frequency.

First, we use a **Finite State Machine (FSM)** implemented in soft logic to generate instructions, similar to Wang et al. [53]. This is shown in Figure 10(a). This method leads to an efficient implementation because the FSM can be customized 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.

To improve programmer productivity, we also consider a stored program method, inspired by Compute RAMs [9]. This is shown in Figure 10(b). For this method, we define macro-instructions for the various operations supported by the CoMeFa RAMs. Table 2 shows the list of macro-instructions (or opcodes) we support. The `ram[x]` notation refers to an operand stored in the CoMeFa RAM at row `x`. The `out[x]` notation refers to an operand outside the RAM. The controller currently supports selecting an operand from nine values outside the CoMeFa RAM using multiplexing logic. A macro-instruction ending in `_ooor` means that at least one of the operands is outside the RAM. Each operation is done for the precision specified in the macro-instruction (using a field ending in `_prec`). The address specified in the instruction refers to the row number of the least significant bit of the operand. The `dot_prod` operations performs $a * x + b * y$. Since there are only 40 bits in a macro-instruction, and we need more than 40 bits to express all operands of this instruction, we make an assumption that the x is laid out right after a and y is laid out right after b. This is why we have `ram[src3+prec]` and `ram[src1+prec]` in the instruction description.

An assembler (written in Python) converts a program written using these macro-instructions into a binary format. This binary data is loaded into a BRAM (either at configuration time or at runtime). An instruction controller (implemented in soft logic) fetches macro-instructions from the BRAM, decodes them, and sends instructions to the CoMeFa RAMs.

Not all applications need to use all the macro-instructions supported by CoMeFa RAMs. For example, an application that performs elementwise operations may only need the `add` and `mul` macro-instructions. To support this, we create an instruction controller generator (written in Python). A user can generate an instruction controller that only supports the macro-instructions they need. This keeps the instruction controller lean and lowers the overhead.

The hardware for executing some macro-instructions such as `reduce` can be complex. Providing support for such macro-instructions in the controller will make it complicated. Instead, for such

Table 2. Macro-Instructions Supported by Our Assembler

Instruction	Operands	Semantics
add	dst, dst_prec, src2, src2_prec, src1, src1_prec	ram[dst +: dst_prec] ← ram[src2 +: src2_prec] + ram[src1 +: src1_prec]
add_ooor	dst, dst_prec, src2, src2_prec, src1, src1_prec	ram[dst +: dst_prec] ← out[src2 +: src2_prec] + ram[src1 +: src1_prec]
mul	dst, dst_prec, src2, src2_prec, src1, src1_prec	ram[dst +: dst_prec] ← ram[src2 +: src2_prec] × ram[src1 +: src1_prec]
mul_ooor	dst_prec, src2, src2_prec, src1, src1_prec	ram[dst +: dst_prec] ← out[src2 +: src2_prec] × ram[src1 +: src1_prec]
logical	dst, src2, src1, prec, op	ram[dst +: dst_prec] ← ram[src2 +: src2_prec] *op* ram[src1 +: src1_prec]
logical_ooor	dst, src2, src1, prec, op	ram[dst +: dst_prec] ← out[src2 +: src2_prec] *op* ram[src1 +: src1_prec]
shift	dst, src, dir, shamt, prec	ram[dst +: prec] ← ram[src +: prec] left or right shifted by *shamt*
dot_prod	dst, dst_prec, src3, src3_prec, src1, src1_prec	ram[dst +: dst_prec] ← ram[src3+src3_prec +: src3_prec] × ram[src3 +: src3_prec] + ram[src1+src1_prec +: src1_prec] × ram[src1 +: src1_prec]
dot_prod_ooor	dst, dst_prec, src4, src3, src2, src1, src_prec	ram[dst +: dst_prec] ← out[src4 +: src_prec] × ram[src3 +: src_prec] + out[src2 +: src_prec] × ram[src1 +: src_prec]
reduce	dst, src, levels, prec	internally reduce operands (each of precision `prec`) located across CoMeFa RAM `levels` times
unload	src, count	unload data from ram[src] to ram[src+count] from the ram
init (set/reset)	dst, pattern, count	ram[dst+count] to ram[dst] ← pattern
set_mask	src	mask register in PE ← ram[src]
nop	count	No operation for `count` cycles

Note that +: has a meaning similar to Verilog's index part-select operator. For example, data[24 +: 8] is the same as data[31:24].

macro-instructions, we move the burden to the assembler. The assembler converts these complex macro-instructions into a sequence of simple macro-instructions such as `add` and `shift`. So, the controller only supports simple macro-instructions and stays lean.

In some applications, the program can be very small (a few macro-instructions). In such cases, using a BRAM to store a few instructions is wasteful. A user can map the binary to distributed RAM in LBs instead of a BRAM. However, in some cases, the program can be very long and may exceed the number of instructions that can be stored in a BRAM. In the applications we evaluated, this was never the case. But, in the future, to reduce the size of the program, we plan to support a macro-instruction (`repeat`) that will implement hardware loops. Support for floating-point operations in the controller is also planned.

To make adoption of CoMeFa RAMs even easier by users, a compiler could be developed that would translate a DNN application (or a part of it) written in Python or C into macro-instructions. Such a compiler would identify the best parallelism distribution across CoMeFa RAM blocks, perform data allocation in memory rows, keep track of the lifetime of each data, and eventually generate macro-instructions (`add`, `mul`, etc.). The assembler would then convert these macro-instructions to the binary format, which will be then be decoded and executed by the instruction controller. We leave the development of this compiler as future work.

4 EVALUATION

4.1 Tools and Methods Used

The **Verilog-to-Routing (VTR)** tool flow [38] is used to evaluate and compare FPGA architectures. VTR synthesizes, places, and routes a given benchmark design on the given FPGA architecture and generates resource usage, area, and timing reports. To obtain the area and delay values for the various components of the FPGA, including CoMeFa RAMs, to enter them in the FPGA architecture model for VTR experiments, we use COFFE [56]. When running COFFE, we used a cost factor of $area * delay^2$ as it reflects the greater emphasis on delay compared to area, which is typical of modern high-performance FPGAs. COFFE-based SPICE simulations use 22-nm libraries from the

Table 3. Properties of the Baseline FPGA Architecture (Intel Arria 10 GX 900 Like)

Resource	Count	Percentage Area
Logic blocks	33,962	66
DSP slices	1,518	18
Block RAMs	2,423	15
DRAM bandwidth	2,048 bits/clock	
Channel width	300	

predictive technology model [50]. We also perform SPICE simulations using FreePDK45 [40] for a circuit containing one bitline pair, two wordline circuits, and two memory cells, with other transistor and wire loads modeled. This is done to get more confidence that the read+compute+write operation of CoMeFa RAMs works, and to validate the numbers obtained from COFFE.

We develop a cycle-accurate behavioral model of CoMeFa RAM to use in functional simulations. This model is written in System Verilog and has the exact same interface (input and output signals) as a CoMeFa RAM hard block in the FPGA architecture model for VTR experiments. Both the storage and compute modes are modeled, by accurately capturing the functionality of the PE and the RAM array. Synopsys VCS and Xilinx Vivado's integrated simulator are used for functional verification of all designs (e.g., designs for benchmarks) used during the evaluation.

We use an analytical model to estimate energy consumption. We add transistor energy and wire energy. For transistor energy, we calculate the energy based on the number of transistors in each block (obtained from the area consumed by the block from VTR). For wire energy, we use wire energy numbers (fJ/mm) from Keckler et al. [32], scale them to 22-nm technology node using the work of Stillmaker and Baas [45], and multiply that with the total routing wirelength from VTR. We use an activity factor of 0.1.

4.2 Baseline vs. Proposed Architectures

We use an Intel Arria 10–like FPGA architecture as the baseline with the same resources as Arria 10 GX900 [25] (Table 3). Arria 10 FPGAs [24] use a technology node (20 nm) similar to our setup (22 nm). Arria 10 GX900 has 96 transceiver channels that support up to 17.4 Gbps [26]. We assume that a four-port full-width soft HMC (Hybrid Memory Cube) controller [22] is implemented on the FPGA to provide a DRAM bandwidth of 2,048 bits/clock. Resources consumed by the controller are not used to map the applications to the FPGA. FPGAs with a higher BRAM:DSP ratio will see even more benefits by converting BRAMs to CoMeFa RAMs.

We use the VTR FPGA architecture used in previous work [7] to make a baseline architecture model. We run COFFE simulations on an Arria 10–like DSP to identify its delay and area. We get delay and areas of a 20 Kb BRAM from COFFE (by interpolating between 16K and 32K). We scale these results based on the DSP and BRAM delays specified by Intel [23]. The DSP slice works at 630 MHz in fixed-point mode and 550 MHz in floating-point mode. The BRAM works at 735 MHz in single-port, simple dual-port, and true dual-port modes. The proposed FPGA architecture models (CoMeFa-D and CoMeFa-A) differ from the baseline in having CoMeFa RAMs instead of normal BRAMs.

4.3 Benchmarks

We create Verilog designs for several diverse applications to use as microbenchmarks (Table 4). These include DL (matrix vector multiplication (**General Matrix Vector Multiplication (GEMV)**), matrix matrix multiplication (**General Matrix Matrix Multiplication (GEMM)**), 2D convolution (Conv2D), reduction, elementwise multiplication (Elt Mul), **Rectified Linear Unit (ReLU)** activation), signal processing (FIR filter or 1D convolution), and bitwise applications

Table 4. List of Microbenchmarks Used for Evaluation

Microbenchmark	Domain	Scenario Created	Storage	Precision
GEMV	DL	CB	DRAM	8-bit
GEMM	DL	CB	DRAM	8-bit
Conv2D	DL	CB	DRAM	8-bit
FIR Filter	Signal Processing	CB	DRAM	16-bit
Elt Mult	DL	DBB	DRAM	HFP8
Bitwise-Search	Databases	OMB	BRAM	16-bit
Bitwise-RAID	Data Center	OMB	BRAM	20-bit
ReLU	DL	OMB	BRAM	16-bit
Reduction	DL	OMB	BRAM	Multiple

CB, compute bound; OMB, on-chip memory-bandwidth bound; DBB, DRAM bandwidth bound.

(database search and RAID array data recovery). We manually map the applications to CoMeFa RAMs and instantiate the CoMeFa RAM blocks in Verilog RTL. During functional verification, a simulation model of CoMeFa RAM is used. We create different scenarios (compute bound, DRAM bandwidth bound, and on-chip memory bound) in these applications. Additionally, we evaluate the impact of adding CoMeFa RAMs on the performance of real-world DNNs from three common types: fully connected networks (Multi-Level Perceptron (MLP)), RNNs (LSTM and GRU), and **Convolutional Neural Networks (CNNs)** (Tiny Darknet and ResNet).

GEMV and GEMM. GEMV and GEMM are fundamental operations in DL applications. They are used in MLPs, LSTMs, and many other DNNs. We consider a GEMV workload where a weight matrix of size 2048×512 is multiplied with an input vector of size 512×1, and a GEMM workload where a weight matrix of size 1536×512 is multiplied with an input matrix of size 512×32. These are sizes from actual layers in DeepBench benchmarks [39]. Eight-bit integer precision with 27-bit accumulation is used. On the baseline FPGA, compute units are implemented using efficient chaining of DSPs. On the proposed FPGA, compute units based on CoMeFa RAMs are additionally deployed, because many RAM blocks are available after mapping the baseline design on the proposed FPGA. The efficient OOOR-based dot product algorithm, described in Section 3.13, is used. Partial sums are read out from the CoMeFa blocks and accumulated using a pipelined bit-serial tree [34]. No online data transpose is required—the weight matrix is transposed offline and pinned into CoMeFa RAM blocks; the input is streamed and does not need to be transposed because it is outside the RAM. Since both DSP-based and CoMeFa-based compute units are used, a reduction in data movement is not expected.

Convolution. The convolution operation forms the backbone of CNNs. We consider a convolution layer with the following parameters: *Input*—Height = 72, Width = 72, Channels = 128; *Filters*—Height = 2, Width = 2, Number = 128; *Output*—Height = 71, Width = 71, Channels = 128. On the baseline FPGA, dot product units are designed using DSP slices to perform multiplications and additions along the channel dimension, then the results from the four filter locations are added. The filters are stored in BRAMs, and the inputs are streamed. A compute unit on the baseline FPGA is made of 64 DSPs and 8 BRAMs. On the proposed FPGAs, CoMeFa RAMs are additionally deployed. Filters are pre-transposed and stored in the CoMeFa RAMs. A compute unit formed by CoMeFa RAMs contains 128 CoMeFa RAMs, along with instruction generation logic. The columns of a CoMeFa RAM are used to store different filters (vectorization across the output channel dimension), whereas the RAMs in a unit are used for vectorization across the input channel dimension. OOOR operations are used to compute dot products. The input data is divided between the compute units formed by DSPs and those formed by CoMeFa RAMs.

FIR Filter. FIR filters are a common DSP application. We consider an FIR filter with 128 taps. Input operands are streamed onto the FPGA through the DRAM interface. The baseline FPGA uses an efficient implementation of the FIR filter using systolic DSP chaining [4]. The proposed FPGA uses CoMeFa RAMs for computation along with DSP chains. LBs were used for control logic. Operands are transposed on-the-fly and loaded into multiple CoMeFa RAMs in parallel. While some CoMeFa RAMs are computing, other CoMeFa RAMs are loaded in a pipelined manner to improve parallelism. When a CoMeFa RAM finishes computing, its results are unloaded and sent to DRAM, and the process starts again until all inputs are processed. We refer to this as the LCU (Load-Compute-Unload) pipeline. In this application, the CoMeFa RAM-to-CoMeFa RAM chaining (Section 3.10) feature is used to share inputs between neighboring blocks.

Elementwise Multiplication. Elementwise multiplications (Elt Mult) are commonly used in DL, for example, in normalization layers and Winograd-based convolution layers. We consider an application involving elementwise multiplication of two arrays of 100K elements. Floating-point data with a precision of HFP8 [47] is used. We showcase here that CoMeFa RAMs are adaptable to any custom precision. The operands are read from DRAM, and the results are written to DRAM. This is a DRAM bandwidth bound application because of low arithmetic intensity. We observed that the number of LBs used was significantly higher (25×) than in the baseline FPGA. This is because to saturate the DRAM bandwidth available on the chip, many swizzle logic instances are required. However, if the swizzle logic is hardened into a DRAM controller, as discussed in Section 3.11, then this overhead is entirely removed.

Bitwise Operations. Bitwise operations (AND, OR, XOR, XNOR, etc.) are commonly used in databases, encryption, DNA sequence alignment, and so forth. They are also used in binary neural networks. CoMeFa RAMs are very efficient at these massively parallel operations because of the presence of mux-based fully configurable PEs. The operands are assumed to be available in BRAMs in the right layout. The speedup seen in these applications is attributed to the effective increase in on-chip memory bandwidth because 160 bits can be operated upon in one cycle in a CoMeFa RAM, compared to only 40 bits from a BRAM in the baseline FPGA. We consider two applications in this category:

(1) *Database search*: In this application, records matching a key are searched. If a record matches the key, it is replaced with special marker data (like constant 0). Each operand is bitwise XORed with the key. Bitwise OR reduction is performed on the result. And then a bitwise ANDing operation is performed to zero out the operands that match the key. BRAMs are used to store operands. Each row of a BRAM has two 16-bit elements. On the proposed FPGA, elements are stored in 256 CoMeFa RAMs. Seven data elements are stored in each column, and temporary results consume 16 rows in a CoMeFa RAM. The key is outside the RAM.
(2) *RAID data recovery*: In RAID (Redundancy Array of Independent Disks) arrays, parity protection is used. If a drive in an array fails, the remaining data on other drives is combined with the parity data (using XOR) to reconstruct the missing data. These numerous parallel XOR operations with the parity data can be accelerated using an FPGA. Instead of storing operands in a transposed format (bits of one operand in multiple rows), we use an untransposed data layout where we store bits of one operand in one row and bits of the second operand in another row. This works for logical operations like bitwise XOR where there is no dependency/communication between consecutive bits and avoids the overhead of transposing data. Performing an XOR operation between operands stored on two rows takes one cycle. A total of 256 RAMs is used.

ReLU. ReLU is the most common activation function used in DNNs. Activations usually follow a GEMM or GEMV or CONV operation. The operation involves zeroing out any negative input,

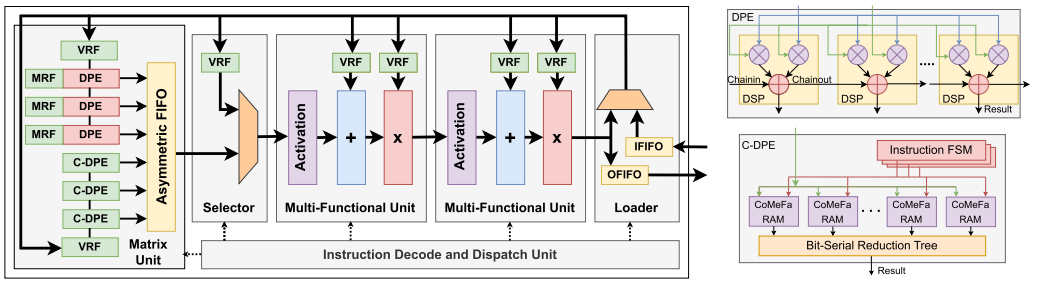
but any positive input stays unchanged. We consider that the input data is available in a RAM (e.g., computed by a prior kernel). The precision is 16-bit. In CoMeFa RAMs, the inverted most significant bit (sign bit) of each input is copied into the mask latches in the PEs. The value 0 is written to each row containing the input elements. In some columns, the operation is masked (because the sign bit was 0), implying the values stay unchanged. But in other columns, the values are zeroed out. In the baseline FPGA, values are read from the RAM, their most significant bit is inspected, and the output is generated using simple multiplexing logic and written back into the RAM.

Reduction. Reduction (or accumulation) is heavily used in DL and DSP applications. We design this application to create a scenario of an on-chip memory bandwidth limited application. Data is available in transposed format (e.g., computed in RAM by a prior kernel). The precision is varied from 4-bit to 20-bit (accumulator size = 32-bit). In the baseline, operands stored in BRAMs are read and successively accumulated using a pipelined adder tree (in LBs). On the proposed FPGA, CoMeFa RAMs store the operands. The reduction algorithm from Eckert et al. [14] is used to reduce the elements to 40 partial sums (1 partial sum in each multiplexed column of the RAM). These intermediate results from multiple CoMeFa RAMs are then read out and accumulated using a popcount-based adder [53] to obtain the result. A significantly smaller number of LBs (~2x–3.5x) is required on the proposed FPGA.

DNNs. To evaluate full neural networks, we create a Microsoft Brainwave-like accelerator [17] based on the work of Boutros et al. [10]. This accelerator consists of five pipeline stages: the **Matrix Unit (MU)** for matrix-vector multiplication operations, the selector unit for skipping the MU when necessary, two MFUs (multi-function units) for vector elementwise operations (e.g., activation, addition, multiplication), and the LD (loader), which interfaces with the DRAM to load and unload data. Register files (MRF and VRF) store the data locally. Similarly to CCB [53], we create two versions of this accelerator: one for the baseline FPGA and another for the proposed FPGA. For the baseline FPGA, the MU consists of **Dot Product Engines (DPEs)** that contain DSP slice cascade chains. Each DPE generates one result. For the proposed FPGA, the MU additionally contains DPEs that are mapped to CoMeFa RAMs (we call these *CoMeFa-DPEs* or *C-DPEs*). The CoMeFa RAMs in C-DPEs receive instructions from instruction generation FSM (duplicated to reduce fanout). A popcount-based bit-serial reduction tree [53] is used to combine the results from various CoMeFa RAMs. Each C-DPE generates 40 results. Figure 11 shows the architecture of the accelerator for the proposed FPGA.

We write an analytical model to explore the distribution of data and BRAMs between DPEs and C-DPEs. There are two main knobs in our analytical model: `f_data`, which decides the fraction of workload (in terms of rows of the matrix processed by the MU) processed by DPEs compared to C-DPEs, and `f_arch`, which decides the fraction of BRAMs allocated to DPEs compared to C-DPEs. Additionally, the analytical model also varies the number of DSPs per DPE and the number of BRAMs per C-DPE over pre-specified ranges. The analytical model iterates over each layer for each neural network and calculates the cycles consumed for each layer. Then we post-process the results from the analytical model using Pandas to find out the best knob (or parameter) settings for each neural network. This results in a different architecture for each neural network. So, instead of having a one-size-fits-all overlay, there is a customized overlay for each neural network. We write an RTL generator to generate the Verilog design for the accelerator with the best hardware parameters identified by the analytical model. Through simulation, we perform sanity verification of our Verilog design and also the analytical model's results.

The Brainwave-like accelerator does not directly support convolutions. So, for CNNs, convolution is expressed as matrix multiplication using the `im2col` operation. We assume that the `im2col` operation is performed in hardware. Although this can be optimized by designing an accelerator



(a) Top-level architecture

(b) Internals of the DPE and C-DPE units

Fig. 11. Microsoft Brainwave-like accelerator used to evaluate DNN performance.

specifically for convolution, our goal here is to showcase the gains from in-memory computation rather than designing the most efficient accelerator.

We consider five DNN benchmarks for this part of the evaluation from three common DNN types: fully connected networks (Multi-Level Perceptron (MLP)), RNNs (LSTM and GRU), CNNs (Tiny Darknet) and residual neural networks (ResNet)). The `mlp` network is a five-layer MLP with each hidden layer having 1,024 neurons, with 4M parameters. The `gru` network has a hidden size = 512, embedding size = 512, and timesteps = 50. It has 1.5M parameters. The `tdarknet` network is Tiny Darknet, a small image classification network for edge devices. It has 650K parameters. The `lstm` network is an LSTM with hidden size = 1,024, embedding size = 1,024, and timesteps = 50. It has 8.4M parameters. The `resnet` benchmark is the ResNet-50 variation of ResNet. It has 24M parameters.

We consider two precisions (int8 and int4) and two batch sizes (1 and 8). We also evaluate the speedup using the two dot product algorithms mentioned in Section 3.13. The FPGA used in our evaluation (Intel Arria 10) is a mid-sized FPGA (47 megabits capacity). Some of the DNNs used for evaluation have weights that do not fit on the FPGA. For int8, `lstm` and `resnet` do not fit. For int4, only `resnet` does not fit. For those cases, we consider the overhead in loading the weights onto the FPGA from DRAM as well.

4.4 Implementation Details

Area. Table 5 shows the area breakdown of both architectures of CoMeFa RAM. For CoMeFa-D, the area overhead is 1,546.78 um^2. This represents an increase of 25.4% in the BRAM tile area compared to the baseline. This overhead is mainly attributed to the addition of 160 PEs and the additional 120 sense amplifiers and write drivers. With BRAMs occupying 15% of the die size in our baseline FPGA, this overhead corresponds to only 3.8% increase in the FPGA chip area. The overhead for CoMeFa-A is 493.5 um^2. Compared to the baseline, this represents an increase of 8.1% in BRAM tile area and only 1.2% increase in FPGA chip area. This overhead is mainly attributed to the addition of 40 PEs.

Frequency. We use COFFE to obtain the overhead in the frequency of operation of a CoMeFa RAM in Hybrid mode, compared to a BRAM (735 MHz). For CoMeFa-D, the cycle duration increases to 1.25× (588 MHz). This is mainly attributed to performing read, compute (PE circuitry delay), and write in the same cycle. For CoMeFa-A, the cycle duration increases to 2.5× (294 MHz). This is because four reads and two writes are done successively as described in Section 3.7. A lower frequency of the CoMeFa RAM is not a concern because realistic FPGA designs typically are constrained by soft logic and routing delays, so designs do not achieve high frequencies like those of individual BRAMs (735 MHz in this case). In Memory mode, the delay overhead is negligible; there is only one additional mux in the write path, and the read path remains unchanged.

Table 5. Area Breakdown of Various RAM Blocks

Component	BRAM	CoMeFa-D	CoMeFa-A
Input and output crossbars	5.6	4.5	5.2
Decoders & wordline drivers	7.8	6.3	7.3
Write drivers & sense amps.	6.9	14.0	6.4
Memory cell array	53.4	43.0	49.6
Routing (conn. & switch)	26.0	20.9	24.1
Processing elements	0	11.1	7.1
Total (%)	**100**	**100**	**100**

Table 6. Differences Between CCB and CoMeFa

Property	CCB	CoMeFa-D	CoMeFa-A
Activate two wordlines at the same time on one port	Yes	No	No
Additional voltage source required	Yes	No	No
Additional row decoder required	Yes	No	No
Changes in sense amplifiers	Yes	No	No
Additional sense amplifiers	Yes	Yes	No
Sense amplifier cycling	No	No	Yes
Compute uses dual-port behavior	No	Yes	Yes
Generic/Flexible PE	No	Yes	Yes
Shift between RAM blocks	No	Yes	Yes
Floating-point support	No	Yes	Yes
Flip-flops in PE to store operands	No	No	Yes
Parallelism	128	160	160
Application(s) demonstrated	DL	Many	Many
Clock duration overhead	60%	25%	125%
Area overhead (block)	16.8%*	25.4%	8.1%
Area overhead (chip)	2.5%*	3.8%	1.2%
Column multiplexing	No	No	Yes
Practicality	**Low**	**Medium**	**High**

*Includes overhead of additional sense amplifiers and write drivers.

Routing. The interface of a CoMeFa RAM block to the programmable routing is not changed compared to that of a BRAM. The only change is the addition of two pins, which are used for direct connections between neighboring BRAMs. These do not impact the programmable interconnect directly but do increase the pin density.

CCB. The implementation of CCB [53] is based on a BRAM with 128 x 128 geometry. The area overhead for the CCB block evaluated in the work of Wang et al. [53] does not include the area of the additional sense amplifiers and write drivers. In our re-implementation of CCB, the total area overhead comes out to be 872.64 um^2, which is a 16.8% increase at the block level and 2.5% at the chip level in the Arria 10–like FPGA used in this study. The frequency of operation of the CCB evaluated by Wang et al. [53] is 1.6× (469 MHz) compared to the baseline BRAM. Table 6 shows the differences between CCB and CoMeFa.

5 RESULTS

5.1 BRAM+PE vs. CoMeFa RAMs

We perform a comparison of CoMeFa RAMs with a normal BRAM and single-bit bit-serial PEs implemented in soft logic. We call the latter the *BRAM+PE* architecture. Figure 12 shows the block

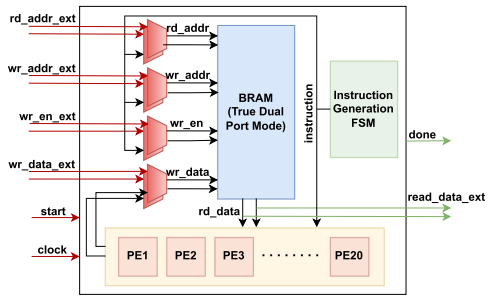


Fig. 12. Block diagram the BRAM+PE architecture.

Table 7. Comparison Between CoMeFa RAMs and BRAM+PE

Parameter	BRAM+PE	CoMeFa
LBs	78	32
RAMs	10	10
Frequency	337.9	536.5
Cycles	64	8
Time (us)	0.19	0.015

diagram of the BRAM+PE architecture. The BRAM is used in true dual-port mode (1024 × 20) so that each PE can be fed with two operands at the same time. Multiplexing logic is provided to allow data to be loaded into the BRAM before the operation, and the results to be unloaded after the operation has finished. When `start` is asserted, the instruction generation FSM (also implemented in soft logic) starts generating instructions. The instruction specifies the BRAM addresses to read to provide data to the PEs. It also specifies the operation to be performed by the PEs. The address to write the results back to is also included in the instruction. After the operation is complete, the done signal is asserted and results can be read out from the BRAM.

Qualitatively, the BRAM+PE architecture suffers from the following disadvantages compared to CoMeFa RAMs:

(1) More cycles are required because separate cycles are required to read each operand and then write the result, and also in each cycle only 40 bits can be read compared to 160 in CoMeFa RAMs.
(2) It uses programmable routing/interconnect to transfer data from BRAM to PEs, resulting in higher power consumption.
(3) It has low frequency of operation and higher area because of the PEs and complex control logic being implemented in soft logic.

Quantitatively, to compare the BRAM+PE and CoMeFa RAMs, we perform a simple elementwise addition operation on an array of two numbers (precision = 8 bits). Data is laid out inside the RAMs in a transposed manner in both the cases. The results are shown in Table 7. We observe that CoMeFa RAMs use approximately 60% less LBs than BRAM+PE. The BRAM+PE design has the PEs, the multiplexing logic for each RAM interface signal, and the instruction generation logic implemented in LBs. However, the CoMeFa RAM design does not have any PEs in LBs, and the multiplexing logic and the instruction generation logic is much simpler. The frequency of operation is higher in the CoMeFa RAM case for the same reasons. In the BRAM+PE case, the critical path included routing wires to implement the multiplexing logic, whereas in the CoMeFa RAM case, the critical path was inside an LB. There is a factor of 8 difference in the number of cycles between the two cases. This includes a factor of 4 from the difference in available bandwidth (160 in the CoMeFa RAM case vs. 40 in the BRAM+PE case) and a factor of 2 from the difference in the number of cycles for each operation (one cycle to read+compute+write in the CoMeFa RAM case vs. two cycles to read+compute+write in the BRAM+PE case). Overall, the time taken by CoMeFa RAMs is an order of magnitude less compared to the time taken in the BRAM+PE case.

5.2 Throughput Comparison

To evaluate the peak throughput, we consider the MAC (multiply-accumulate) operation, which is the most common operation in DSP and DL applications. We use common fixed-point precisions:

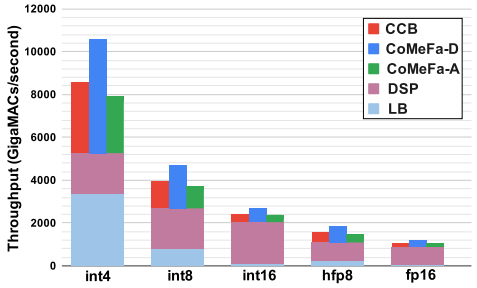


Fig. 13. Peak throughput for MAC operations for the whole FPGA for various precisions.

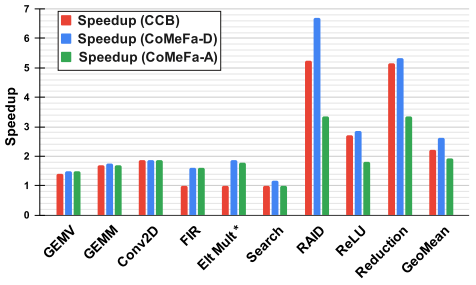


Fig. 14. Speedups obtained for different FPGA architectures for various benchmarks. An asterisk (*) implies no DRAM bandwidth limitation.

4 bit (accumulator = 16 bits), 8 bit (acc = 27 bits), and 16 bit (acc = 36 bits). Additionally, we use floating-point precisions: HFP8 ({exp = 4, frac = 3} and acc = {exp = 6, frac = 9}) [47] and IEEE FP16 (acc = IEEE FP32). We compare the throughput of CoMeFa RAMs to the traditional compute units (LBs and DSPs). For LBs, we synthesize, place, and route one MAC onto the FPGA and determine the operating frequency and resource utilization. We then calculate the throughput by optimistically assuming that we can fill the FPGA at the same operating frequency. This serves the purpose of evaluating peak throughput. For DSPs, MACs are created and taken through a similar process. The DSPs do not natively support FP16 and HFP8 precisions, so MACs for these precisions are designed using soft logic and DSPs. For CoMeFa RAMs, 160 MACs are implemented in parallel by instantiating one CoMeFa RAM and an instruction generation FSM.

Figure 13 shows the peak throughput for each precision obtained from each different computing resource in GigaMACs per second. We observe that the throughput of the FPGA increases by 2×, 1.7×, 1.3×, 1.7×, and 1.3× for int4, int8, int16, hfp8, and fp16, respectively, by adding CoMeFa-D RAMs. Similarly, the throughput of the FPGA increases by 1.5×, 1.36×, 1.16×, 1.36×, and 1.15× for int4, int8, int16, hfp8, and fp16, respectively, by adding CoMeFa-A RAMs. CoMeFa RAM throughput reduces as the precision increases, due to the bit-serial nature of computation in CoMeFa RAMs. CoMeFa RAMs can be used for computing in any precision, unlike DSPs. The frequency of operation of CoMeFa RAMs does not change significantly with changing precision, unlike LBs.

Note that the compute throughput enhancement evaluated here is for MAC operations only and does not use OOOR operations. The speedup we obtain for different benchmarks can vary from the peak throughput enhancement calculated here because (1) non-MAC operations like reductions may be needed, (2) clock frequency may be lower because of large designs, (3) cycles may be spent in loading and unloading data to/from CoMeFa RAMs, (4) DRAM reads and writes may bound certain parts of the application, (5) OOOR operations may be used to speed up the operation, and (6) LBs may only be used for control logic and not for computation.

5.3 Resource Usage and Frequency

Table 8 shows the resource usage and frequency of operation for the various compute-bound and DRAM-bound microbenchmarks obtained from the VTR flow (averaged over three seeds). The table shows the data for the baseline FPGA and the three FPGA variations with compute-enabled BRAMs (CCB, CoMeFa-D, and CoMeFa-A). The resource usage for each resource is in percentage of the total resources of that type on the FPGA. We observe that the LB usage increases significantly for all microbenchmarks. This is because of the control logic (instruction generation logic, data loading/unloading logic, etc.) required for using the compute-enabled BRAMs. The DSP usage

Table 8. Resource Usage (Percentage) and Frequency (F, in Megahertz) for Compute- and DRAM-Bound Microbenchmarks

	Baseline				FPGA with Compute-Enabled BRAMs					
Benchmark	**LB**	**DSP**	**BRAM**	**F**	**LB**	**DSP**	**BRAM**	**F(CCB)**	**F(CoMeFa-D)**	**F(CoMeFa-A)**
GEMV	1.6	90.1	43.4	253	27.9	90.1	91.8	231	242	242
GEMM	0.8	92.4	38.6	269	25.5	92.4	86.7	260	267	260
Conv2D	5.0	91.8	28.5	255	35.5	91.8	91.3	245	246	243
FIR	12.8	93.0	3.5	243	53.1	93.0	95.3	–	229	229
Elt Mult	25.8	49.8	38.1	300	21.7*	0	82.6	–	292	288

*Does not include LB usage from swizzle modules to capture the infinite DRAM bandwidth case.

Table 9. Resource Usage (Absolute Values) and Frequency (F, in Megahertz) for On-Chip Memory Bound Microbenchmarks

	Baseline				FPGA with Compute-Enabled BRAMs					
Benchmark	**LB**	**DSP**	**BRAM**	**F**	**LB**	**DSP**	**BRAM**	**F(CCB)**	**F(CoMeFa-D)**	**F(CoMeFa-A)**
Search	2,242	0	280	600	1,206	0	256	451	465	294
RAID	1,538	0	256	702	578	0	256	459	588	294
ReLU	560	0	256	616	301	0	256	445	465	294
Reduction	4,072	0	256	445	1,184	0	256	453	469	294

remains the same as baseline, and the usage of RAMs significantly increases. Note that RAMs are not used *in place of* DSPs but *additionally*, to maximize the usage of the FPGA to exploit the higher compute throughput to obtain speedup. The FIR benchmark uses chaining of RAMs, which is not supported by CCB. Similarly, CCB does not support floating-point operations. So, we do not implement the FIR and Elementwise Multiplication benchmarks on the CCB architecture and hence the frequency is marked with a dash (–). For the Elementwise Multiplication benchmark, we construct a design with no swizzle modules so that the design fits on the FPGA by maximizing the number of CoMeFa RAM for compute. This is done to obtain the theoretical speedup in the case with unlimited DRAM bandwidth (see Section 5.4).

Table 9 shows the resource usage and frequency of operation of the various on-chip memory bound microbenchmarks obtained from the VTR flow. The resource usage shown here is in absolute numbers. This is because for these benchmarks, we create a small design that uses similar FPGA RAM resources on the baseline FPGA and the FPGA with compute-enabled BRAMs. Bitwise-Search uses more BRAMs in the baseline because of under-utilization of the RAM due to the data layout. No DSPs are used for computation in these benchmarks on the baseline FPGA as well the FPGA with compute-enabled BRAMs. A significantly lesser number of LBs are used in the FPGAs with compute-enabled BRAMs because the computation is done internal to the BRAMs and LBs are not used for computation. The frequency of operation is very high on the baseline FPGA because the control logic is much simpler compared to the control logic (instruction generation logic) in the FPGAs with compute-enabled BRAMs. An interesting observation is that in the CoMeFa-A case, the frequency of operation is always limited by the frequency of operation of the CoMeFa-A RAM (294 MHz).

5.4 Speedup and Energy Benefits

Figure 14 shows the speedup obtained by using compute-enabled BRAMs across microbenchmarks. We see significant speedups by using CoMeFa RAMs in the compute-bound applications because of the augmented compute throughput provided by the FPGA. For the GEMV benchmark, speedups of 47.5% are seen in CoMeFa-D and CoMeFa-A. With CCB, the max speedup was 40%. For GEMM,

the speedup was 74.5% for CoMeFa-D and 69% for CoMeFa-A and CCB. A speedup of ~85% is seen in the Convolution benchmark for all three architectures because the frequency of operation was similar in all three, as seen in Table 8. A speedup of 59% is seen in the FIR benchmark for both CoMeFa-D and CoMeFa-A. The FIR benchmark uses chaining of RAMs, which is not supported by CCB. So, no speedup is considered compared to the baseline.

Since the Elementwise Multiplication benchmark is limited by DRAM bandwidth, no speedup is seen by using CoMeFa RAMs. CoMeFa RAMs are targeted to improve the compute throughput of the FPGA, not the DRAM bandwidth. If we remove the restriction of DRAM bandwidth and assume that all compute units (CoMeFa RAMs as well as DSPs/LBs) can be fed with data, then speedups of 86% and 79% can be seen on CoMeFa-D and CoMeFa-A FPGAs, respectively. Since CCB does not support floating-point operations, the speedup for this benchmark for CCB is shown as 0%.

The Search benchmark is sped up by 18% for CoMeFa-D. The design on baseline FPGA had the highest frequency of operation because of very simple operations done in soft logic, as seen in Table 9. No speedup is seen using CoMeFa-A RAMs because of the low frequency of operation. This application is not sped up by using CCB either. CCB takes ~2× cycles compared to CoMeFa RAM because of the inflexibility of the PEs that only support a few operations. For example, the AND operation can be done in two cycles in CCB, compared to one cycle in CoMeFa RAM. The RAID application is sped up by 6.7× in CoMeFa-D, 3.35× in CoMeFa-A, and 5.2× in CCB. The baseline frequencies were very high in this case also, but the difference in number of cycles enabled the significant speedups. In the ReLU benchmark, speedup of 2.7×, 2.85×, and 1.8× are seen in CCB, CoMeFa-D, and CoMeFa-A, respectively. The speedups for the Reduction benchmark (4-bit precision) were 5.3× in CoMeFa-D, 3.3× in CoMeFa-A, and 5.1× in CCB.

Results from our energy model are shown in Figure 15. We observe that the results are similar for the various architectures—CCB, CoMeFa-D, and CoMeFa-A. FIR and Elt Mult benchmarks are not run on the CCB architecture as mentioned earlier, so those results are omitted from the figure.

In compute-bound benchmarks (GEMV, GEMM, Conv2D, FIR), an increase in energy consumption is observed. This is because the resource usage in these benchmarks is significantly higher compared to the baseline, as seen in Section 5.3. For example, for GEMV, the baseline uses 1.6% LBs, 90.1% DSPs, and 43.4% BRAMs, but with CoMeFa RAMs, 27.9% LBs, 90.1% DSPs, and 91.8% CoMeFa RAMs are used. The additional LBs are required for control logic to program the CoMeFa RAMs and also for reduction of partial results obtained from CoMeFa RAMs. To reduce fanout from this logic to CoMeFa RAMs (to achieve high frequencies), this control logic had to be replicated multiple times, increasing the LB usage significantly. This increased resource usage leads to a high power consumption. The reduction in time for these benchmarks, compared to the baseline, is less than 2×, as seen earlier in this section from the Speedup results (see Figure 14). For example, for GEMV, the speedup is ~1.47. Since energy is evaluated by combining the power consumption and the time taken, the energy consumption is higher when running these benchmarks on an FPGA with CoMeFa RAMs. These results indicate that using lower precision (e.g., int4) in these benchmarks could lead to an overall energy reduction because of increased speedup. To confirm this, we implemented the GEMV benchmark for an FPGA with CoMeFa-A RAMs with the int4 precision. We observed a speedup of 2.83 compared to the baseline. The baseline used 1.45% LBs, 91% DSPs, and 23.8% BRAMs, but with CoMeFa RAMs, 28.8% LBs, 89.5% DSPs, and 89.7% CoMeFa RAMs were used. A energy reduction of 24% was observed.

In the memory-bound benchmark (Elt Mult), an energy reduction of 40% is seen in both CoMeFa-D and CoMeFa-A, but note that this excludes the impact from the LBs used for swizzle logic, to showcase the infinite DRAM bandwidth case.

In the on-chip memory bandwidth bound microbenchmarks (Search, RAID, ReLU, Reduction), up to 38% less LBs are used in CoMeFa compared to baseline. That is because no LBs are needed

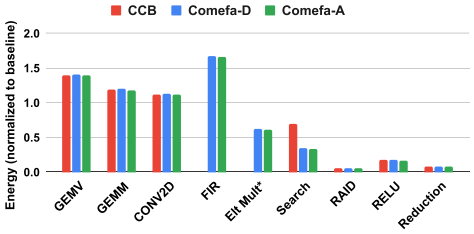


Fig. 15. Energy consumption for all microbenchmarks. An asterisk (*) implies no DRAM bandwidth limitation.

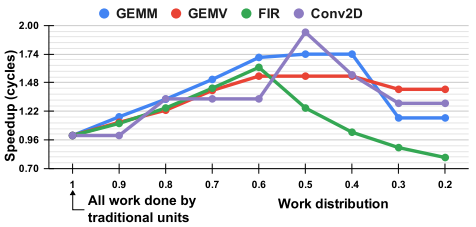


Fig. 16. Illustration of variation in speedup (based on cycles) by partitioning the application between DSPs and CoMeFa RAMs.

for computation when CoMeFa RAMs are used. Routing wirelength reduction of up to 68% is seen, which directly correlates to reduction in data movement. This reduces power consumption by up to 56% in CoMeFa-A and up to 52% in CoMeFa-D. With significant reduction in time obtained by using CoMeFa RAMs for these benchmarks as seen in the Speedup results (see Figure 14), the energy reduction of up to 95% can be seen.

5.5 Application Co-mapping

CoMeFa RAMs supplement DSPs and LBs as compute units, and enhance the FPGA's compute throughput. Appropriately dividing the data between CoMeFa RAMs and traditional compute units is key. For the compute-bound applications (GEMV, GEMM, FIR, and Conv2D), we analytically explore the effect of varying data distribution between CoMeFa RAMs and DSPs/LBs on the proposed FPGA. The results are shown in Figure 16. We see that as more work is given to CoMeFa RAMs, more speedup can be obtained up to a limit, after which the overheads (loading, unloading, serial compute) associated with CoMeFa RAMs can start dominating and reduce the overall speedup. This sweet spot is different for each application. In some cases, mapping a majority of the application onto CoMeFa RAMs can even cause an overall slowdown because of higher latency.

5.6 Adaptability to Precision

CoMeFa RAMs can be used for efficiently computing in any custom precision. Figure 17 shows the results of sweeping the precision from 4- to 20 bits in the Reduction benchmark. We see speedups ranging from 5.3× (3.3×) to 2.7× (1.7×) with CoMeFa-D (CoMeFa-A) as precision increases. CoMeFa-D is 3% better than CCB owing to the improved frequency achieved by the design. The baseline takes the same number of cycles for each precision because of the bit-parallel nature of compute. But the number of cycles taken increases as the precision increases when CoMeFa RAMs are used. This is because of bit-serial arithmetic and illustrates that applications using smaller precisions are better suited for CoMeFa RAMs. Note that the frequency of operation stays constant for CoMeFa RAMs because the hardware architecture stays the same. For the baseline, the frequency decreases slightly as the precision increases.

5.7 Using Stored Programs Instead of Hardcoded FSM

For each microbenchmark, we create a design that uses the stored program method discussed in Section 3.14. Table 10 shows the resource usage of benchmarks when the stored program method is used. Comparing this to the resource usage in Table 8, we observe that the LB usage increases significantly. This increase is attributed to the relatively more generic instruction controller logic, compared to the hardcoded FSM logic that can be highly specialized and optimized for a specific benchmark. The DSP usage remains exactly the same because the instruction controller logic does not use any DSPs. The BRAM usage remains almost the same because of how the experiment is

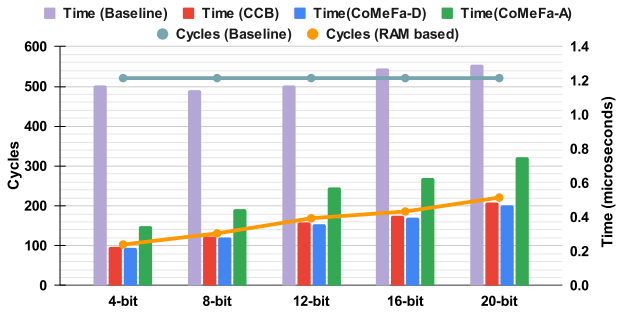


Fig. 17. Sweeping precision in the Reduction benchmark.

Table 10. Resource Usage When Instruction Generation Logic Is Implemented Using the Stored Program Method

Benchmark	LB	DSP	BRAM
GEMV	49.0	90.1	91.7
GEMM	53.5	92.4	86.7
Conv2D	55.2	91.8	91.4
FIR	63.0	93.0	95.7
Elt Mult	24.2	0	84.2

Table 11. Speedup Obtained When Instruction Generation Logic Is Implemented Using a Customized FSM and Using the Stored Program Method

Benchmark	Customized FSM	Stored Program
GEMV	1.47	1.07
GEMM	1.69	1.22
Conv2D	1.85	1.45
FIR	1.59	1.41
Elt Mult	1.79	1.21

designed. Some BRAMs used for computation are instead re-purposed to be used for storing instructions. The minor differences arise because number of CoMeFa RAMs removed from compute units (e.g., the dot product units in GEMM and GEMV microbenchmarks) in the benchmark and the number of instruction RAMs required by the remaining compute units may differ.

We compare the speedup obtained by using the stored program method with the speedup obtained using the hardcoded FSM-based method. The results are shown in Table 11. We observe a ~40% reduction in speedup on average. But there is a significant hard-to-quantify improvement in programmability of the CoMeFa RAMs by using the stored program method. There are two main reasons for reduction in the speedup. First, instruction storage in the stored program method can consume a significant number of BRAMs, reducing the BRAMs available for compute. The hardcoded FSM method does not use any BRAMs in the instruction generation logic. Second, when using the stored program based method, a reduction in frequency of operation of the design was observed. The critical path was in the instruction decoder of the controller.

In the future, we plan to improve the speedup obtained with the stored program method by (1) adding pipeline stages in the controller to improve frequency and (2) mapping macro-instructions to distributed RAMs in LBs to keep the number of BRAMs available for compute the same.

We only evaluate the reduction in speedup for compute-bound and DRAM-bound microbenchmarks here. For the on-chip memory bandwidth microbenchmarks, the experimental setup is such that a small number of BRAMs is used in both baseline and proposed cases. A few extra BRAMs can be used to store instructions and obtain the same speedup as the case with hardcoded FSM.

5.8 DNN Evaluation

Figure 18 shows the speedup obtained by using the accelerator shown in Figure 11, for five DNNs along with the geometric mean. The baseline uses an accelerator without C-DPEs, on an FPGA without CoMeFa RAMs. Three knobs or parameters are varied: precision, batch size, and dot product algorithm. The frequency of operation of the accelerator was the same for both cases—using CoMeFa-D and using CoMeFa-A—because the critical path of the design was not in the MU of the accelerator. So, these speedups apply to both cases.

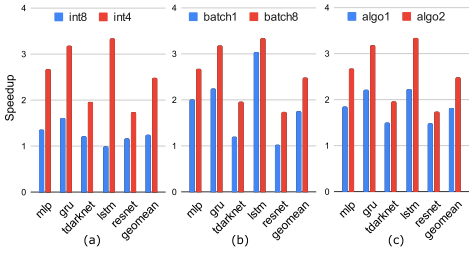


Fig. 18. Variation of speedup for DNNs with precision, batch size, and dot product algorithm.

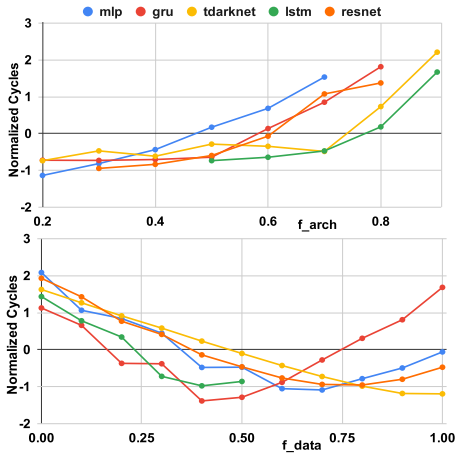


Fig. 19. Variation of cycles consumed for DNNs with changing f_arch and f_data.

In Figure 18(a), we see a geomean speedup of 1.26× with int8 precision, and that increases to 2.49× with int4 precision. Because of the bit-serial computation in CoMeFa RAMs, smaller precision exhibit low latencies and higher speedups compared to the baseline. These speedups are for a batch size of 8 using Algorithm 2. In Figure 18(b), we compare the speedup for batch size of 4 and batch size of 8, for int4 precision and Algorithm 2. The speedup increases with batch size because of improved utilization, higher reuse, and amortization of weight loading (when needed). In Figure 18(c), we compare the speedups obtained by using Algorithm 1 and Algorithm 2, for a batch size of 8 and precision of int4. Algorithm 1 is slower than Algorithm 2 because in Algorithm 2, we take advantage of inspecting 1 bit each from two operands outside the RAM, and reduce cycles by up to 2×.

In Figure 19, we observe the trends of varying the knobs `f_arch` and `f_data` (discussed in Section 4.3) in our analytical model. We plot the number of cycles consumed (normalized) for each DNN, along the y-axis. In the top chart, `f_data` is kept constant at 0.5, implying that 50% of the workload (matrix rows) is assigned to the DPEs and the remaining 50% is assigned to the C-DPEs. For higher values of `f_arch`, the number of cycles consumed is high because we do not have enough BRAMs available for the 50% of the workload assigned to C-DPEs. As we move left along the x-axis, `f_arch` reduces, and more BRAMs become mapped to C-DPEs, which in turn means the part of the workload assigned to C-DPEs can be executed efficiently, reducing the overall cycles consumed.

In the bottom chart in Figure 19, `f_arch` is kept constant at 0.5, implying that 50% of the BRAMs are assigned to DPEs and 50% are assigned to C-DPEs. When `f_data` is low (on the left of the x-axis), only a small amount of the workload is assigned to C-DPEs, so we do not get much speedup. But as we move right along the x-axis, more of the workload is assigned to C-DPEs, achieving a lower number of cycles. But as we move farther right, the cycles start to increase again because the latency of the C-DPEs starts to dominate. Medium values of `f_data` give the highest speedup.

6 DISCUSSION

6.1 Comparison with Other FPGA Blocks

CoMeFa RAMs are universal blocks and can be used for accelerating any application. CoMeFa RAMs are not replacements of DSPs or LBs but can work together and complement them. In some

ways, CoMeFa RAMs can be thought of as blocks that fuse together LBs and BRAMs. They provide a more structured way of computation compared to LBs, along with the storage capability of BRAMs. Compared to DSPs, CoMeFa RAMs provide infinite flexibility in terms of precision (because of their bit-serial nature) even after the chip has been designed. DSPs can support multiple precisions too, but the precisions have to be hardened at the time of designing the FPGA, and adding more precisions increases DSP area. CoMeFa RAMs support more operations than DSPs (which mainly just support multiplication and addition) because of the configurable PE in them. They also allow flexibility in which algorithm to use for multiplication and addition, through bit-serial and OOOR operations. With DSPs, a user is forced to use the multiplier or adder architecture that was designed into it.

6.2 Applications

Applications that are well suited for deploying CoMeFa RAMs include the following:

(1) Applications that have significant SIMD parallelism (e.g., DL and signal, image, and video processing).
(2) Applications that do not require a lot of communication between PEs (e.g., elementwise and bitwise operations).
(3) Applications that use reduced and/or custom numerical precisions (e.g., DL).

6.3 Organizing Data for Computation

When performing computation using CoMeFa RAMs, data is laid out in a transposed manner (i.e., bits of an element are stored along a bitline). In addition to operands and results, intermediate results need to be stored in the same column. If the intermediate results are not required at a later stage, they can be overwritten to improve RAM utilization. For example, consider the case where four operands a, b, c, d are stored in a column and the operation required is $e = a * b + c * d$. The intermediate results $a * b$ and $c * d$ are calculated bit-serially and then added. The final result can reuse the same rows as those storing the intermediate results by overwriting them.

In some cases, complex operations can be split over multiple columns and the final results can be obtained by reducing intermediate results from different columns. For example, in the example of calculating $e = a * b + c * d$, if the precision of operands is 18-bit fixed point, we would run out of rows (total available = 128) to store the operands, intermediate results, and the final result. Splitting independent operations over multiple columns exposes more parallelism and can achieve better speedups. However, reducing intermediate results stored in different columns involves serialization and can reduce the obtained speedup. Bit-slicing [53] is another method in which individual elements are split over multiple columns. For example, a 16-bit number can be sliced into two 8-bit chunks and stored in eight rows in two columns. Operations can be performed independently on the two bit-slices and then concatenated and reduced appropriately to get the final result.

There are tradeoffs in RAM utilization, data reuse, and latency. Having a smaller number of operands in one column (bitline) means reduced utilization but low latency, because it will take fewer cycles to perform the operation (because there is only one PE per column). However, if we store more operands in a column, the RAM will have higher utilization but the latency will be higher. Additionally, more operands in a column allow for more reuse. Consider an example where one operand needs to be multiplied by two other operands and then the partial products need to be added. Having all three operands in one column allows this operation to happen locally. Otherwise, two multiplications will need to be done in separate columns (or separate BRAMs) and then reduced. Smaller precisions allow for more elements to co-exist in a column.

Consider an operation where elements stored in all 160 columns of a CoMeFa RAM need to be reduced to get one final result. Performing in-CoMeFa RAM reductions will proceed in a tree

fashion, where after the first reduction step, 80 columns will have the new partial results. Then, these will be reduced further into 40 columns. At some stage, performing further in-CoMeFa RAM reductions can significantly degrade the compute throughput, as only a successively smaller portion of the CoMeFa RAM columns are actively performing compute during each reduction iteration. To avoid this, reduction can be performed in soft logic outside the CoMeFa RAMs. Forty partial results can be read out one bit-slice at a time (bit 0 of 40 partial results in one cycle, bit 1 of these partial results in the next cycle, and so on) and reduced externally. A popcount-based external reduction [53] can be used for this addition/accumulation.

6.4 Parallelism

The parallelism CoMeFa RAMs provide (SIMD) is different from the pipeline parallelism that is commonly used with LBs and DSP slices. Although SIMD parallelism can be achieved with LBs and DSP slices as well, CoMeFa RAMs provide it in a more efficient and compact form. In applications where SIMD parallelism from CoMeFa RAMs is used to obtain speedup (e.g., the on-chip memory boundary bound applications in Section 4), reduced data movement will typically be observed leading to a significant reduction in energy.

In addition to SIMD parallelism, data parallelism (splitting the data to be processed between traditional DSP and LB-based compute units, and CoMeFa RAM based compute units) is used to exploit the additional compute throughput provided by CoMeFa RAMs. Let us consider the case where time T was spent on processing D chunks of data on a baseline FPGA (using only traditional compute units like LBs and DSPs). However, if a part of the data (e.g., $D/3$ chunks) is processed by CoMeFa RAMs and if the rest of the data ($2D/3$) is processed by traditional compute units in parallel, then the total time taken would be less than T (say $2T/3$). This achieved speedup depends on the distribution of work between traditional units and CoMeFa RAM based units, and the best case would be when both types of units finish in an approximately equal amount of time. In applications where data parallelism is used to obtain speedup (e.g., compute-bound applications in Section 4), energy consumption may not reduce because more hardware is used to solve the problem.

Different applications may need different types of parallelism. Even parts of one application may be suited for different types of parallelism. So, adding CoMeFa RAMs to FPGAs opens the door to new ways to exploit parallelism efficiently.

6.5 Integration into an Open Acceleration Framework

To demonstrate using CoMeFa RAMs with already existing acceleration frameworks, we integrate a CoMeFa RAM based acceleration unit into CFU Playground [41] from Google and Harvard. Figure 20 shows the overall architecture of our system. CFU stands for Custom Functional Unit. CFU Playground is a collection of software and hardware to make it easy for everyone, including software engineers, to accelerate machine learning/DL inferencing. Overall, the system provides a soft RISC-V-based SoC that can be mapped to any FPGA, with a simple C-based programming interface, along with the capability to design a CFU that is easily hooked up to the CPU. In the original CFU Playground framework, the CFU is tightly coupled to the pipeline of the CPU. Both commands and data to the accelerator are sent from the CPU interface. This limits the acceleration that can be achieved because of the large amount of data to be transferred using a narrow interface. We enhance the framework by adding a direct memory access path from the accelerator unit shown using red arrows in the figure (currently only using a simulation model).

We write a C program to first populate the instruction RAM in the accelerator. Then the accelerator is triggered by writing into a control register. This trigger initiates the instruction controller in the accelerator to fetch data into the CoMeFa RAMs using the direct memory access path via the

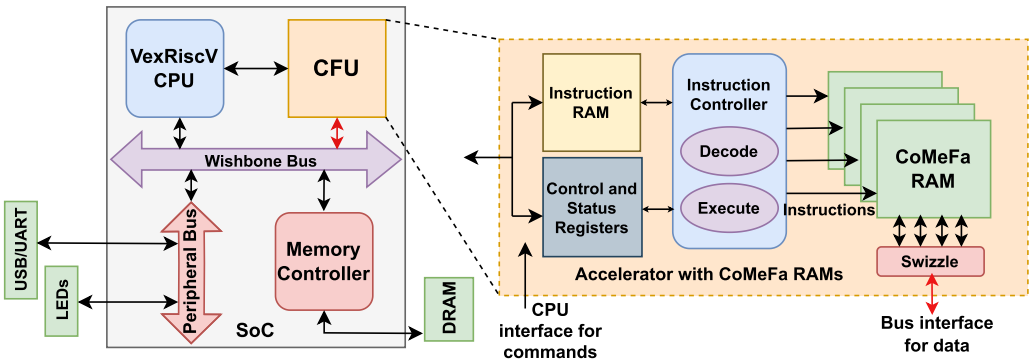


Fig. 20. Integrating a CoMeFa RAM based unit into an SoC using an open source accelerator framework (CFU Playground).

swizzle logic. A status register is read to ensure that the data transfer has been completed. Then another command is sent by writing to a control register in the accelerator. This command initiates the controller to fetch the instructions from the instruction RAM in the accelerator, and decode and execute them on CoMeFa RAMs. After the execution has finished, another status register is set. The CPU busy-waits on this status register until the execution is finished (there are no interrupts currently).

We deploy kernels to perform elementwise addition and multiplication on large arrays of data, using this framework. With the C-based interface of the RISC-V CPU and the instruction-based interface of the CoMeFa RAMs, it was very easy to develop and use this accelerator. We compare the cycles to perform the same kernels using DSP-based accelerators, and the speedup obtained was similar to that in Section 5.

7 CONCLUSION

In this article, we proposed augmenting the compute density of FPGAs by modifying BRAMs into new blocks called *CoMeFa RAMs*, which are ideal for enhancing applications with inherent parallelism like DL. To the best of our knowledge, this is the first work that (1) utilizes the dual-port nature of BRAMs to achieve in-BRAM compute, (2) deploys configurable 1-bit PEs inside an FPGA BRAM, (3) uses OOOR operations, and (4) applies in-BRAM compute to DL and non-DL applications on FPGAs. With improvement in compute density and reduction in energy consumption, converting some or all BRAMs on FPGAs to CoMeFa RAMs can be a significant step toward closing the performance gap between FPGAs and ASICs. Non-SRAM technologies like ReRAM or STT-MRAM have been proposed to be used for FPGA BRAMs instead of SRAMs [12, 29, 48]. Simultaneously, compute-in-memory has been explored with these technologies as well [21, 27]. In the future, we plan to work on adding compute-in-memory capabilities to FPGA BRAMs based on these technologies. Most of the concepts in CoMeFa RAMs are agnostic to the underlying technology.

REFERENCES

[1] Achronix. 2019. Speedster7t FPGAs. Retrieved June 9, 2023 from https://www.achronix.com/product/speedster7t-fpgas.

[2] Shaizeen Aga, Supreet Jeloka, Arun Subramaniyan, Satish Narayanasamy, David Blaauw, and Reetuparna Das. 2017. Compute caches. In *Proceedings of the 2017 IEEE International Symposium on High Performance Computer Architecture (HPCA'17)*. 481–492. https://doi.org/10.1109/HPCA.2017.21

[3] Amogh Agrawal, Akhilesh Jaiswal, Chankyu Lee, and Kaushik Roy. 2018. X-SRAM: Enabling in-memory Boolean computations in CMOS static random access memories. *IEEE Transactions on Circuits and Systems I: Regular Papers* 65, 12 (2018), 4219–4232.

[4] Altera. 2015. Designing Filters for High Performance. Retrieved June 9, 2023 from https://www.intel.cn/content/dam/www/programmable/us/en/pdfs/literature/wp/wp-01260-stratix10-designing-filters-for-high-performance.pdf.

[5] Aman Arora, Samidh Mehta, Vaughn Betz, and Lizy K. John. 2021. Tensor slices to the rescue: Supercharging ML acceleration on FPGAs. In *Proceedings of the International Symposium on Field-Programmable Gate Arrays (FPGA'21)*. 23–33.

[6] Aman Arora, Tanmay Anand, Aatman Borda, Rishabh Sehgal, Bagus Hanindhito, Jaydeep Kulkarni, and Lizy K. John. 2022. CoMeFa: Compute-in-memory blocks for FPGAs. In *Proceedings of the 2022 IEEE 30th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM'22)*. 1–9. https://doi.org/10.1109/FCCM53951.2022.9786179

[7] Aman Arora, Andrew Boutros, Daniel Rauch, Aishwarya Rajen, Aatman Borda, Seyed Alireza Damghani, Samidh Mehta, et al. 2021. Koios: A deep learning benchmark suite for FPGA architecture and CAD research. In *Proceedings of the 2021 31st International Conference on Field-Programmable Logic and Applications (FPL'21)*. https://doi.org/10.1109/FPL53798.2021.00068

[8] Aman Arora, Moinak Ghosh, Samidh Mehta, Vaughn Betz, and Lizy K. John. 2022. Tensor slices: FPGA building blocks for the deep learning era. *ACM Transactions on Reconfigurable Technology and Systems* 15, 4 (Dec. 2022), 1–34. https://doi.org/10.1145/3529650

[9] Aman Arora, Bagus Hanindhito, and Lizy K. John. 2021. Compute RAMs: Adaptable compute and storage blocks for DL-optimized FPGAs. In *Proceedings of the 2021 55th Asilomar Conference on Signals, Systems, and Computers*. 1156–1163. https://doi.org/10.1109/IEEECONF53345.2021.9723277

[10] Andrew Boutros, Eriko Nurvitadhi, Rui Ma, Sergey Gribok, Zhipeng Zhao, James C. Hoe, Vaughn Betz, and Martin Langhammer. 2020. Beyond peak performance: Comparing the real performance of AI-optimized FPGAs and GPUs. In *Proceedings of the International Conference on Field Programmable Technology (FPT'20)*.

[11] A. Boutros, S. Yazdanshenas, and V. Betz. 2018. Embracing diversity: Enhanced DSP blocks for low-precision deep learning on FPGAs. In *Proceedings of the 2018 28th International Conference on Field Programmable Logic and Applications (FPL'18)*. 35–357.

[12] Yi-Chung Chen, Wenhua Wang, Hai Li, and Wei Zhang. 2012. Non-volatile 3D stacking RRAM-based FPGA. In *Proceedings of the 22nd International Conference on Field Programmable Logic and Applications (FPL'12)*. 367–372. https://doi.org/10.1109/FPL.2012.6339206

[13] Ping Chi, Shuangchen Li, Cong Xu, Tao Zhang, Jishen Zhao, Yongpan Liu, Yu Wang, and Yuan Xie. 2016. PRIME: A novel processing-in-memory architecture for neural network computation in ReRAM-based main memory. In *Proceedings of the 2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA'16)*. 27–39. https://doi.org/10.1109/ISCA.2016.13

[14] Charles Eckert, Xiaowei Wang, Jingcheng Wang, Arun Subramaniyan, Ravi Iyer, Dennis Sylvester, David Blaauw, and Reetuparna Das. 2018. Neural cache: Bit-serial in-cache acceleration of deep neural networks. In *Proceedings of the 45th Annual International Symposium on Computer Architecture (ISCA'18)*. IEEE, Los Alamitos, CA, 383–396. https://doi.org/10.1109/ISCA.2018.00040

[15] Mohamed Eldafrawy, Andrew Boutros, Sadegh Yazdanshenas, and Vaughn Betz. 2020. FPGA logic block architectures for efficient deep learning inference. *ACM Transactions on Reconfigurable Technology and Systems* 13, 3 (June 2020), Article 12, 34 pages. https://doi.org/10.1145/3393668

[16] D. G. Elliott, M. Stumm, W. M. Snelgrove, C. Cojocaru, and R. Mckenzie. 1999. Computational RAM: Implementing processors in memory. *IEEE Design Test of Computers* 16, 1 (1999), 32–41. https://doi.org/10.1109/54.748803

[17] Jeremy Fowers, Kalin Ovtcharov, Michael Papamichael, Todd Massengill, Ming Liu, Daniel Lo, Shlomi Alkalay, et al. 2018. A configurable cloud-scale DNN processor for real-time AI. In *Proceedings of the 45th Annual International Symposium on Computer Architecture (ISCA'18)*. IEEE, Los Alamitos, CA, 1–14. https://doi.org/10.1109/ISCA.2018.00012

[18] Fei Gao, Georgios Tziantzioulis, and David Wentzlaff. 2019. ComputeDRAM: In-memory compute using off-the-shelf DRAMs. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'19)*. ACM, New York, NY, 100–113. https://doi.org/10.1145/3352460.3358260

[19] R. Gauchi, V. Egloff, M. Kooli, J.-P. Noel, B. Giraud, P. Vivet, S. Mitra, and H.-P. Charles. 2020. 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*. ACM, New York, NY, 121–126. https://doi.org/10.1145/3370748.3406550

[20] S. Ghose, A. Boroumand, J. S. Kim, J. Gómez-Luna, and O. Mutlu. 2019. Processing-in-memory: A workload-driven perspective. *IBM Journal of Research and Development* 63, 6 (2019), Article 3, 19 pages. https://doi.org/10.1147/JRD.2019.2934048

[21] Mohsen Imani, Saransh Gupta, Yeseong Kim, and Tajana Rosing. 2019. FloatPIM: In-memory acceleration of deep neural network training with high precision. In *Proceedings of the 46th International Symposium on Computer Architecture*. 802–815. https://doi.org/10.1145/3307650.3322237

[22] Intel. 2016. Hybrid Memory Cube Controller IP Core User Guide v16.0. Retrieved June 9, 2023 from https://www.intel.com/content/www/us/en/docs/programmable/683854/16-0/introduction.html.
[23] Intel. 2020. Intel Arria 10 Device Datasheet. Retrieved June 9, 2023 from https://www.intel.com.tw/content/dam/www/programmable/us/en/pdfs/literature/hb/arria-10/a10_datasheet.pdf.
[24] Intel. 2021. Intel Arria 10 Device Overview. Retrieved June 9, 2023 from https://www.intel.com/content/www/us/en/docs/programmable/683332/current/device-overview.html.
[25] Intel. 2021. Intel Arria 10 Product Table. Retrieved June 9, 2023 from. https://www.intel.cn/content/dam/www/programmable/us/en/pdfs/literature/pt/arria-10-product-table.pdf.
[26] Intel. 2021. Intel Arria 10 Transceiver PHY User Guide. Retrieved June 9, 2023 from https://www.intel.cn/content/dam/www/programmable/us/en/pdfs/literature/hb/arria-10/ug_arria10_xcvr_phy.pdf.
[27] Shubham Jain, Ashish Ranjan, Kaushik Roy, and Anand Raghunathan. 2018. Computing in memory with spin-transfer torque magnetic RAM. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 26, 3 (March 2018), 470–483. https://doi.org/10.1109/TVLSI.2017.2776954
[28] Supreet Jeloka, Naveen Bharathwaj Akesh, Dennis Sylvester, and David Blaauw. 2016. 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 (2016), 1009–1021. https://doi.org/10.1109/JSSC.2016.2515510
[29] Lei Ju, Xiaojin Sui, Shiqing Li, Mengying Zhao, Chun Jason Xue, Jingtong Hu, and Zhiping Jia. 2018. NVM-based FPGA block RAM with adaptive SLC-MLC conversion. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 37, 11 (Nov. 2018), 2661–2672. https://doi.org/10.1109/TCAD.2018.2857261
[30] Mingu Kang, Sujan K. Gonugondla, and Naresh R. Shanbhag. 2020. Deep in-memory architectures in SRAM: An analog approach to approximate computing. *Proceedings of the IEEE* 108, 12 (2020), 2251–2275. https://doi.org/10.1109/JPROC.2020.3034117
[31] Mingu Kang, Min-Sun Keel, Naresh R. Shanbhag, Sean Eilert, and Ken Curewitz. 2014. An energy-efficient VLSI architecture for pattern recognition via deep embedding of computation in SRAM. In *Proceedings of the 2014 IEEE International Conference on Acoustics, Speech, and Signal Processing (ICASSP'14)*. 8326–8330. https://doi.org/10.1109/ICASSP.2014.6855225
[32] Stephen W. Keckler, William J. Dally, Brucek Khailany, Michael Garland, and David Glasco. 2011. GPUs and the future of parallel computing. *IEEE Micro* 31, 5 (2011), 7–17. https://doi.org/10.1109/MM.2011.89
[33] Aaron Landy and Greg Stitt. 2015. Revisiting serial arithmetic: A performance and tradeoff analysis for parallel applications on modern FPGAs. In *Proceedings of the 2015 IEEE 23rd Annual International Symposium on Field-Programmable Custom Computing Machines*. 9–16. https://doi.org/10.1109/FCCM.2015.53
[34] Aaron Landy and Greg Stitt. 2017. Serial arithmetic strategies for improving FPGA throughput. *ACM Transactions on Embedded Computing Systems* 16, 3 (July 2017), Article 84, 25 pages. https://doi.org/10.1145/2996459
[35] Martin Langhammer, Eriko Nurvitadhi, Bogdan Pasca, and Sergey Gribok. 2021. Stratix 10 NX Architecture and Applications. In *Proceedings of the International Symposium on Field-Programmable Gate Arrays (FPGA'21)*. 57–67
[36] David Lewis, David Cashman, Mark Chan, Jeffery Chromczak, Gary Lai, Andy Lee, Tim Vanderhoek, and Haiming Yu. 2013. Architectural enhancements in Stratix V. In *Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays (FPGA'13)*. ACM, New York, NY, 147–156. https://doi.org/10.1145/2435264.2435292
[37] Shuangchen Li, Dimin Niu, Krishna T. Malladi, Hongzhong Zheng, Bob Brennan, and Yuan Xie. 2017. DRISA: A DRAM-based reconfigurable in-situ accelerator. In *Proceedings of the 2017 50th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'17)*. 288–301.
[38] Kevin E. Murray, Oleg Petelin, Sheng Zhong, Jai Min Wang, Mohamed ElDafrawy, Jean-Philippe Legault, Eugene Sha, et al. 2020. VTR 8: High performance CAD and customizable FPGA architecture modelling. *ACM Transactions on Reconfigurable Technology and Systems* 13, 2 (2020), Article 9, 55 pages.
[39] Sharan Narang. 2016. Baidu DeepBench. Retrieved June 9, 2023 from https://svail.github.io/DeepBench/.
[40] NCSU. 2018. FreePDK45. Retrieved June 9, 2023 from https://eda.ncsu.edu/freepdk45/.
[41] Shvetank Prakash, Tim Callahan, Joseph Bushagour, Colby Banbury, Alan V. Green, Pete Warden, Tim Ansell, and Vijay Janapa Reddi. 2022. CFU playground: Full-stack open-source framework for tiny machine learning (tinyML) acceleration on FPGAs. *arXiv:2201.01863 [cs]* (2022). http://arxiv.org/abs/2201.01863.
[42] S. Rasoulinezhad, H. Zhou, L. Wang, and P. H. W. Leong. 2019. PIR-DSP: An FPGA DSP block architecture for multi-precision deep neural networks. In *Proceedings of the 2019 IEEE 27th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM'19)*. 35–44.
[43] Vivek Seshadri, Donghyuk Lee, Thomas Mullins, Hasan Hassan, Amirali Boroumand, Jeremie Kim, Michael A. Kozuch, Onur Mutlu, Phillip B. Gibbons, and Todd C. Mowry. 2017. Ambit: In-memory accelerator for bulk bitwise operations using commodity DRAM technology. In *Proceedings of the 2017 50th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'17)*. 273–287.

[44] Ali Shafiee, Anirban Nag, Naveen Muralimanohar, Rajeev Balasubramonian, John Paul Strachan, Miao Hu, R. Stanley Williams, and Vivek Srikumar. 2016. ISAAC: A convolutional neural network accelerator with in-situ analog arithmetic in crossbars. In *Proceedings of the 2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA'16)*. 14–26. https://doi.org/10.1109/ISCA.2016.12

[45] A. Stillmaker and B. Baas. 2017. Scaling equations for the accurate prediction of CMOS device performance from 180 nm to 7 nm. *Integration, the VLSI Journal* 58 (2017), 74–81. http://vcl.ece.ucdavis.edu/pubs/2017.02.VLSIntegration.TechScale/.

[46] Arun Subramaniyan, Jingcheng Wang, Ezhil R. M. Balasubramanian, David Blaauw, Dennis Sylvester, and Reetuparna Das. 2017. Cache automaton. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'17)*. ACM, New York, NY, 259–272. https://doi.org/10.1145/3123939.3123986

[47] Xiao Sun, Jungwook Choi, Chia-Yu Chen, Naigang Wang, Swagath Venkataramani, Vijayalakshmi Srinivasan, Xiaodong Cui, Wei Zhang, and Kailash Gopalakrishnan. 2019. Hybrid 8-bit floating point (HFP8) training and inference for deep neural networks. In *Advances in Neural Information Processing Systems*, Vol. 32. Curran Associates. https://proceedings.neurips.cc/paper/2019/file/65fc9fb4897a89789352e211ca2d398f-Paper.pdf.

[48] Kosuke Tatsumura, Sadegh Yazdanshenas, and Vaughn Betz. 2018. Enhancing FPGAs with magnetic tunnel junction-based block RAMs. *ACM Transactions on Reconfigurable Technology and Systems* 11, 1 (March 2018), 1–22. https://doi.org/10.1145/3154425

[49] Jeffrey Tyhach, Mike Hutton, Sean Atsatt, Arifur Rahman, Brad Vest, David Lewis, Martin Langhammer, et al. 2015. Arria 10 Device Architecture. In *Proceedings of the 2015 IEEE Custom Integrated Circuits Conference (CICC'15)*. 1–8.

[50] Arizona State University. 2012. Predictive Technology Model. Retrieved June 9, 2023 from http://ptm.asu.edu/.

[51] Jingcheng Wang, Xiaowei Wang, Charles Eckert, Arun Subramaniyan, Reetuparna Das, David Blaauw, and Dennis Sylvester. 2020. 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 (Jan. 2020), 76–86. https://doi.org/10.1109/JSSC.2019.2939682

[52] Jingcheng Wang, Xiaowei Wang, Charles Eckert, Arun Subramaniyan, Reetuparna Das, David Blaauw, and Dennis Sylvester. 2020. 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 (2020), 76–86. https://doi.org/10.1109/JSSC.2019.2939682

[53] Xiaowei Wang, Vidushi Goyal, Jiecao Yu, Valeria Bertacco, Andrew Boutros, Eriko Nurvitadhi, Charles Augustine, Ravi Iyer, and Reetuparna Das. 2021. Compute-capable block RAMs for efficient deep learning acceleration on FPGAs. In *Proceedings of the 2021 IEEE 29th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM'21)*. 88–96. https://doi.org/10.1109/FCCM51124.2021.00018

[54] Xilinx. 2018. AI Engines and Their Applications. Retrieved June 9, 2023 from https://www.xilinx.com/support/documentation/white_papers/wp506-ai-engine.pdf.

[55] Xilinx. 2021. UltraScale Architecture Memory Resources. Retrieved June 9, 2023 from https://www.xilinx.com/support/documentation/user_guides/ug573-ultrascale-memory-resources.pdf.

[56] Sadegh Yazdanshenas and Vaughn Betz. 2019. COFFE2: Automatic modelling and optimization of complex and heterogeneous FPGA architectures. *ACM Transactions on Reconfigurable Technology and Systems* 12, 1 (Jan. 2019), Article 3, 27 pages.

[57] Sadegh Yazdanshenas, Kosuke Tatsumura, and Vaughn Betz. 2017. Don't forget the memory: Automatic block RAM modelling, optimization, and architecture exploration. In *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA'17)*. ACM, New York, NY, 115–124. https://doi.org/10.1145/3020078.3021731

Received 30 January 2023; revised 17 May 2023; accepted 1 June 2023