# Compute RAMs: Adaptable Compute and Storage Blocks for DL-Optimized FPGAs

Aman Arora*, Bagus Hanindhito†, Lizy K. John‡
*Department of Electrical and Computer Engineering*
*The University of Texas at Austin*
*aman.kbm@utexas.edu, †hanindhito@bagus.my.id, ‡ljohn@ece.utexas.edu

***Abstract*—The configurable building blocks of current FPGAs — Logic blocks (LBs), Digital Signal Processing (DSP) slices, and Block RAMs (BRAMs) — make them efficient hardware accelerators for the rapid-changing world of Deep Learning (DL). Communication between these blocks happens through an interconnect fabric consisting of switching elements spread throughout the FPGA. In this paper, a new block, Compute RAM, is proposed. Compute RAMs provide highly-parallel processing-in-memory (PIM) by combining computation and storage capabilities in one block. Compute RAMs can be integrated in the FPGA fabric just like the existing FPGA blocks and provide two modes of operation (storage or compute) that can be dynamically chosen. They reduce power consumption by reducing data movement, provide adaptable precision support, and increase the effective on-chip memory bandwidth. Compute RAMs also help increase the compute density of FPGAs. In our evaluation of addition, multiplication and dot-product operations across multiple data precisions (int4, int8 and bfloat16), we observe an average savings of 80% in energy consumption, and an improvement in execution time ranging from 20% to 80%. Adding Compute RAMs can benefit non-DL applications as well, and make FPGAs more efficient, flexible, and performant accelerators.**

## I. Introduction

Deep Learning (DL) has become ubiquitous in today's world. The ever-increasing computational demands of DL applications has triggered 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 and energy efficiency.

FPGAs contain fine-grained programmable logic (e.g., LBs), fixed-function math units (e.g., DSP slices), and embedded memory structures (e.g., BRAMs) that can be connected by a configurable interconnection fabric. These building blocks are very generic, making FPGAs a great solution to design various accelerator, but this flexibility, unfortunately, limits the performance we can achieve with FPGAs for DL applications. In recent years, DL-optimized FPGA architectures have been proposed and deployed, such as adding vector processors [1] and integrating DL-specific blocks [2] [3] on the FPGA chip. Most FPGA vendors have added support for smaller, DL-friendly precisions (e.g., 8-bit fixed-point (`int8`) and `bfloat16` [4]) in DSP slices.

Even so, there are still some limitations in current FPGA architectures. Separation of compute units (LBs and DSPs) from storage units (BRAMs) leads to a lot of data movement to feed the compute units with input data and to store the outputs back to the storage units. This is exacerbated for DL applications because of the math-intensive nature of operations involved in them. This data movement, although on-chip, is expensive in terms of power consumption because the movement happens through the FPGA interconnect which comprises of numerous switches instead of hard connected wires. This flexible but inefficient interconnect also leads to slower frequencies for designs on FPGAs (typically 3-4x lower than ASICs [5]).

BRAMs on FPGAs support a limited set of heights and widths. For example, BRAMs in Intel Stratix 10 [6] are 20 Kilobits in size and can be configured as 512x40, 1024x20 and 2048x10 bits, with only 1 or 2 read and write ports. Owing to the parallel nature of DL applications, it is common to process thousands of bits of data together. It is preferred for users to split the data over multiple BRAMs for higher bandwidth, with only a few rows of each block are utilized.

Another limitation is the limited number of precisions supported by the DSP slices. For example, DSP slices in Intel Agilex FPGAs [7] support multiplication and MAC (multiply-accumulate) operations in 9x9, 27x27, 18x19 fixed-point and 16-bit or 32-bit floating-point precisions. Although DSP slices have become more complex over the years to support more precisions, the precision requirements change rapidly, especially in the world of DL. Users have to implement math units on LBs instead, reducing the number of LBs available for other purposes and leaving DSPs unused.

In this paper, we propose adding a new type of block, called Compute RAM, to FPGAs. **A Compute RAM block enables computation within the RAM array, without transferring the data in or out of it.** The implementation of these blocks is based on an emerging Logic-in-Memory SRAM prototype by Jeloka et. al. [8]. Using this technology and its extensions (bit-line computing [9] and bit-serial arithmetic [10]), processing-in-memory engines can be designed. For Compute RAMs, we add components to such SRAMs to integrate them into the FPGA fabric and make them configurable. Compute RAMs can be programmed during FPGA configuration time or during run-time. The user to perform math in any precision using Compute RAMs. Every column of the memory performs the same operation simultaneously, resulting in massive parallelism and high throughput. An operation can be performed per column in the Compute RAM, thereby dramatically reducing

the bandwidth limitations. Since there is no need to move the data in and out of the block, the energy efficiency improves drastically. A reduced dependency on FPGA interconnect also means using Compute RAMs leads to faster frequencies. On the top of these advantages, the blocks can still be used as pure storage blocks as needed. A block diagram of an FPGA with Compute RAMs is shown in Figure 1.

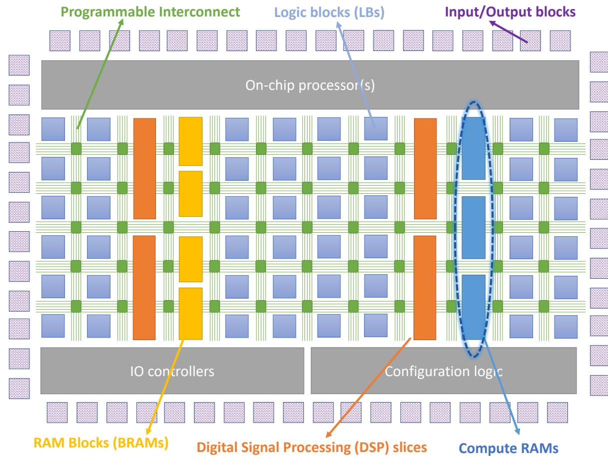


**Fig. 1:** An FPGA with the proposed Compute RAM blocks

Here is the summary of the contributions of this paper:

1) Propose adding blocks called Compute RAMs to FPGAs
2) Describe the architecture of Compute RAM blocks and their various features
3) Demonstrate the benefit of deploying Compute RAMs for common DL operations

The rest of this paper is organized as follows. The next section provides an overview of related work in the area of in-memory compute and DL-optimized FPGAs. In Section III, we present the proposed Compute RAMs and its various aspects in detail. Next, the methodology used to perform the evaluation is detailed in Section IV. We discuss the results for common DL operations in Section V. Finally, we conclude and briefly mention future work in Section VI.

## II. Related Work and Background

### A. DL-Optimized FPGAs

In recent years, many DL-specific modifications to FPGA architecture have been deployed by the industry. Xilinx Versal family [1] adds specialized vector processors for DL acceleration. Intel's Stratix 10 NX FPGAs integrate in-fabric AI tensor blocks [2]. Achronix Speedster7t FPGAs [11] have embedded machine learning processor (MLP) blocks that have an array of multipliers, an adder tree and accumulators. The FlexLogix nnMAX [12] inference IP also contains hard blocks to perform convolutions. Native support for `fp16` and `bfloat16` data types in DSP slices has also been added to recent FPGAs.

There have also been a number of academic research proposals to optimize FPGA architectures for DL. Eldafrawy et al. [13] proposed several enhancements to the LB architecture, including adding a shadow multiplier in them. In [14], the Boutros et al. propose enhancing DSP blocks by efficiently supporting low precision multiplications. Rasoulinezhad et al. [15] have proposed DSP slice modifications such as including a register file for data reuse and improvements to DSP-DSP interconnect. Arora et al. [3] also proposed adding Tensor slices in FPGAs.

To our knowledge, no existing work proposes adding processing-in-memory (PIM) blocks to the FPGA fabric.

### B. Processing-In-Memory

Proposals for Processing-In-Memory (PIM) [16] architectures have been around for decades, but the products which implement it can only be seen in the recent years, specifically for DL acceleration. Memristor-based PIM accelerators like ISAAC [17] and PRIME [18] were the early entrants in this field, but recently many digital solutions have been proposed as well, such as FloatPIM [19] which has support for floating point operations. The main limitation of these architectures is integrating them on the same Silicon die that uses a standard CMOS-based process. Only some vertical stacked architectures have been shown to work so far [20].

Samsung recently announced a DRAM product called HBM-PIM [21] which integrates compute units onto a High Bandwidth Memory chip. This paradigm is near-memory compute instead of in-memory compute. Mythic AI's Intelligent Processing Unit (IPU) contains tiles that have analog matrix multiplier which uses Flash memory transistors. Their design requires DACs and ADCs for operation.

Jeloka et. al. [8] showcased a Logic-in-Memory SRAM prototype. Multiple word lines are activated simultaneously and the shared bit-lines can be sensed, effectively performing logical AND and NOR operations on the data stored in the activated rows. By lowering the word-line voltage, data corruption due to multi-row access is prevented. Aga et al. [9] proposed deploying these bit-line computing enabled SRAMs as caches in CPUs to create massively parallel compute engines. They extend the technology to add the capability of operations like compare, NOT, XOR, copy, search, etc.

Eckert et al. [10] combine this capability with bit-serial arithmetic, which involves processing one bit of multiple data elements every cycle, instead of processing multiple bits of 1 data element every cycle. Data is stored in a transposed format in the array. That is, the bits of operands are stored in one column (i.e. in multiple word lines). Some logic gates are added near the sense amplifiers of each column (bit-line) to make performing arithmetic operations easier. In the first half of a clock cycle, two bits of operands are read, and logical or arithmetic operations are performed on them. The result is written back into the rows of the same column in the second half of the same clock cycle. Figure 2 shows how a dot product operation can be performed using this method.

In this paper, we propose Compute RAMs which extend and enhance this technology for FPGAs. Note that Compute RAMs are not limited to using this technology; any memory technology as that provides bit-level computing can be used to design Compute RAMs.

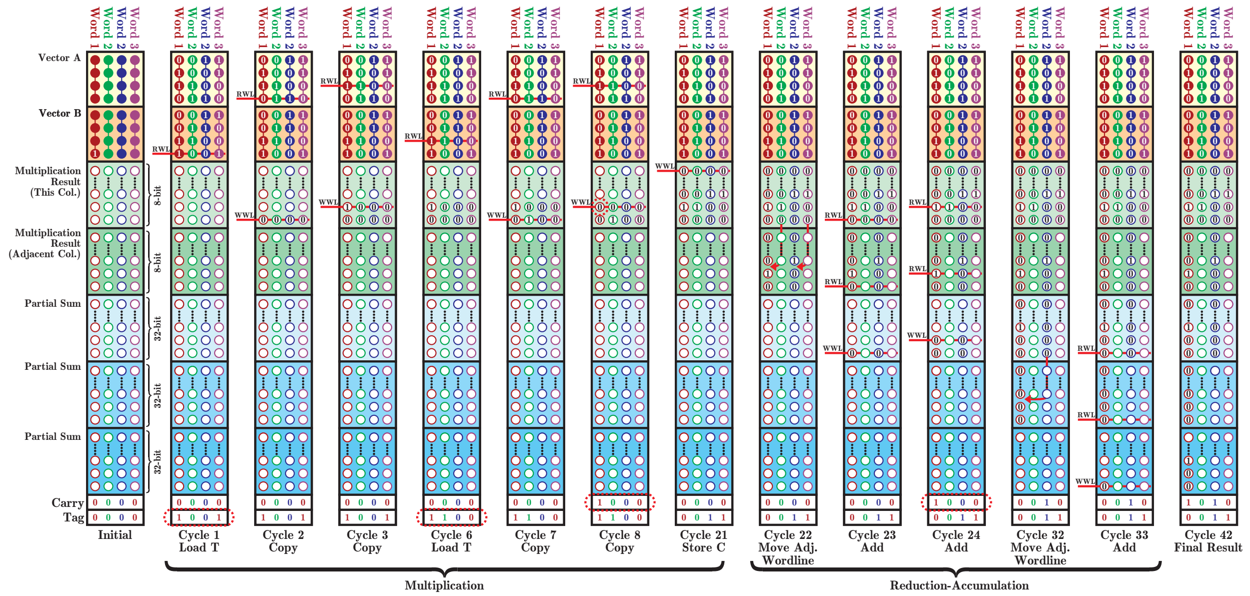


**Fig. 2:** Performing a dot product based on the architecture in [10] (WWL = Write Word Line, RWL = Read Word Line)

## III. Proposed Architecture: Compute RAM

Figure 3 shows the architecture of the proposed Compute RAM block. The heart of the Compute RAM block is an SRAM array (called the main array) that supports bit-line computing, as prototyped in [8]. The instruction memory is a small SRAM that contains the instructions for operation to be performed on the data. For example, it could contain the instruction sequence for performing `int4` (4-bit fixed-point) additions on the data in the array. This software-like mechanism enables users to perform computations with any precision while using Compute RAMs. A controller reads and decodes the instruction sequence stored in the instruction memory. Based on these instructions, it sends commands to the array to perform the computations. Logic peripherals, enhanced compared to [10] are present in bit-line. This reduces the length of instruction sequences required to perform more complex operations.

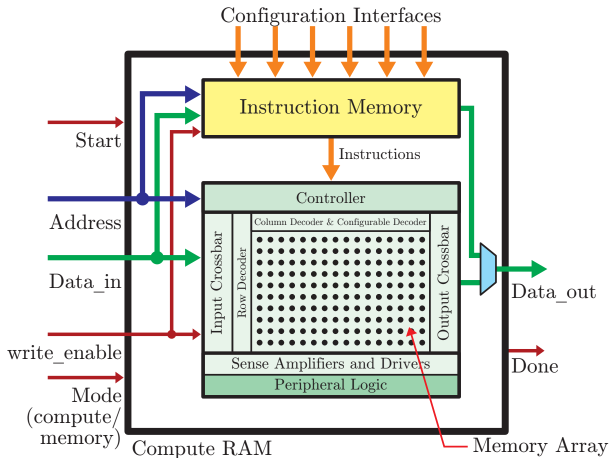


**Fig. 3:** Architecture of a Compute RAM block

### A. Details of Components

*1) Main Array:* The main array is an SRAM array that supports logic-in-memory or bit-line computing. It's operation is briefly described in Section II. This SRAM array is a drop-in replacement of a typical BRAM on an FPGA. FPGAs have additional circuitry to allow for configurable geometries (height and width) [22]. The same circuitry can be used here without changing the operation of the Compute RAM. Because the compute operations are done in parallel across columns, to obtain the most parallelism, it is best to configure the Compute RAM as wide as possible and as shallow as possible. We use the same size and geometries as BRAMs in Intel FPGAs. The main array in Compute RAM is 20 Kilobits in size and can be configured in 512x40, 1024x20 and 2048x10 geometries.

*2) Instruction Memory:* An SRAM is provided to store the sequence of instructions for the operations to be done on the data in the main array and can be loaded in two ways:

- **At FPGA configuration time.** To allow this, we provide connection for this memory to talk to the FPGA configuration interface. This method can be used when the operation executed by the block is not expected to be changed during execution time.
- **At execution time.** Sometimes, the operation that will be performed on the data inside the array needs to be changed dynamically. For e.g., when the instruction sequences are longer than the capacity of this memory, which can only hold 256 instructions. Writing into this memory dynamically is made possible by sharing address and data bus of the array.

To identify the size of this memory for our architecture, we wrote the sequences for common operations like fixed-point addition, multiplication and MAC, and floating-point addition, multiplication and MAC. We found that the none of the operations was more than 200 instructions. So, we provide

TABLE I: I/O interface of a Compute RAM block

| Signal | Direction | Function |
|---|---|---|
| mode | Input | Compute mode or storage mode |
| start | Input | Start executing instructions |
| address | Input | Read/write address |
| data_in | Input | Write data |
| write_en | Input | Read or write |
| data_out | Output | Read data |
| done | Output | Instruction execution finished |

space for 256 instructions. With each instruction being 16-bit wide, the size of this memory is 4 Kilobits.

*3) Controller:* The controller in the Compute RAM block is required to fetch, decode, and execute the instructions in the instruction memory. It is implemented as a simple pipelined processor. The main array serves as the data memory for this processor. The number of registers in the register file is 8. In the sequences of common operations we wrote, we never used more than 5 registers at a time. The register file is implemented using flip-flops instead of a RAM to save area and allow multiple registers to be accessed at the same time.

From the viewpoint of this controller, the instructions are of two types:

- Instructions executed by the controller's execution unit. For example, branch or add a value to a register.
- Instructions sent to the main array. For example, performing bitwise add operation on bits stored in two rows in the main array.

The controller has a very simple execution unit - it only has 1 adder, 1 comparator and 1 logical unit. It does not have complex blocks like multipliers. We note that common DL operations involve repetitive instructions requiring loops. To reduce the number of instructions for an operation, the controller employs zero-overhead branch processing using dedicated hardware loop control, like in conventional DSP processors [23].

*4) Logic Peripherals:* Logic peripherals are present for each bitline (each column) as in [10]. The single-ended sense amplifiers sense the result from two cells A and B, in the same bitline. BL gives $A.B$, while BLB gives $\bar{A}.\bar{B}$. To support floating point operations, we need instruction execution predicated on multiple conditions like the sign of a previous result. We add a 4-to-1 mux that selects the predication condition from among `Carry`, `NotCarry` and `Tag`.

### B. Interface and Operation

Table I presents the name and description of the ports on a Compute RAM block. Most of the ports are the same as a BRAM (`address`, `data_in`, `write_en`, `data_out`). We add some additional ports to the block to enable Compute RAM functionality (`start`, `mode`, `done`). Only 3 additional ports are added, minimizing the overhead of adding ports to an FPGA block.

The `mode` input specifies whether a user wants to use the Compute RAM block in compute mode or storage mode. In storage mode (`mode=0`), the block works exactly like a BRAM on an FPGA. The controller and logic peripherals as well as both `start` and `done` signals are not used in this mode. The instruction memory can be used as a regular BRAM by the application. The non-utilized structures has area overhead which is insignificant ($\sim$12%).

In a typical use case, logic external to the Compute RAM (eg. a state machine implemented in LBs) will configure the Compute RAM in storage mode first. The input data will then be loaded into the array (e.g. from external DRAM). Then, the mode will be changed to compute mode and the `start` signal will be asserted. Instructions in the instruction memory will execute in order. When the last instruction (signalled by the presence of `end` instruction) is executed, the `done` signal is asserted. The external logic will wait for assertion of `done` before reading the results.

### C. Advantages and Limitations

A computation performed using Compute RAM would have be done using the following on a baseline FPGA:

- A BRAM to store the input operands and results
- LBs implementing control logic to orchestrate the data transfer and computation
- DSP slices or LBs to perform the actual computation

Compute RAM, on the other hand, provides the storage, the computation capability and the control logic integrated into one block. There are many advantages of using Compute RAMs:

1) Because the computation happens inside the memory block, no wire and switching energy is spent in sending data to/from the compute units. Data movement between various blocks on the FPGA is significantly reduced. This leads to reduction in power consumption and an increase in energy efficiency. Another impact of the reduced dependence on the FPGA interconnect is that designs can now operate at higher frequencies of operation, thereby resulting in speeding up applications.
2) Any custom operation with any custom precision can be supported by a Compute RAM block. No hardware with hardcoded support for specific operations and fixed number of precisions is involved in a Compute RAM. For performing a different operation or for using a different precision, the instruction sequence needs to be modified. This can be done either at FPGA configuration time or at execution time. Changing the instruction sequence at execution time makes Compute RAMs programmable in a software-like manner.
3) Using a Compute RAM for compute reduces the limitations of the bandwidth available from a BRAM because of the limited geometries and number of ports. In a Compute RAM, there are as many operations in progress at a time as many columns. Users can avoid splitting data over multiple BRAMs to get more bandwidth and using only a few rows of each blocks. Now, the array can be fully utilized, and the total area of implementing a circuit is greatly reduced.
4) Using Compute RAMs leads to reduced area to implement a given circuit. In comparison to a BRAM, a

Compute RAM has an area overhead of the instruction memory, controller and peripheral logic. However, this area overhead is smaller than using a BRAM, a DSP slice and several LBs for realizing a computation on a baseline FPGA. This also leads to reduced power consumption. More importantly, this means larger circuits can now fit on the same FPGA chip. Adding Compute RAMs to FPGAs leads to an increase in the compute density of the FPGA ($GOPS/mm^2$).

There are some limitations of adding Compute RAMs to FPGAs. Adding a new block on an FPGA means more heterogeneity make mapping/synthesizing harder. But all BRAMs can be replaced with Compute RAMs, thereby preserving the heterogeneity that exists today. Also, for some operations like floating point operations, Compute RAMs utilize some rows to store temporary results, reducing the overall capacity of the array. But these rows can be reused across all computations in a column and can be repurposed dynamically. Adding Compute RAMs to FPGAs means that users have to adopt a different programming model (writing instruction sequences), but this can be made easy by designing compilers and/or creating libraries of common operation sequences.

## IV. Experimental Methodology

### A. Tools

We used the following tools to perform the experiments described in this paper:

- VTR 8.0 for FPGA architecture exploration [24]
- Synopsys VCS 2018 for Verilog simulations [25]
- Synopsys Design Compiler 2018 for ASIC synthesis [26]
- OpenRAM for estimating area and delays of SRAMs

VTR is an academic tool for exploration of FPGA architectures. It takes two inputs - an FPGA architecture description file and a Verilog design file. In the FPGA architecture description file, the information of FPGA's building blocks and interconnect resources is provided. The Verilog design file contains the circuit we intend to map onto the FPGA. VTR synthesizes and implements the benchmark Verilog design for a hypothetical FPGA with the given architecture, and generates area and timing reports.

### B. FPGA Architecture

For the experiments in this paper, we use an architecture similar to Intel Agilex [7] used by Arora et al. in [3] as the baseline FPGA. Some of the properties of this FPGA architecture are as follows:

- **Logic Block:** The logic block contains 10 basic logic elements. Each logic element consists of fracturable 6-input LUT, a flip-flop, and 2 bits of arithmetic. There are 60 inputs and 40 outputs on a logic block.
- **DSP Slice:** The DSP slice supports addition (floating point only), multiplication, and MAC operations, along with some complex modes like $a * b + c$ or $(a + b) * c$. The precisions supported are 8-bit, 18-bit and 27-bit fixed point, and 16-bit (IEEE half precision and bfloat) and 32-bit (IEEE full precision) floating point.
- **BRAM:** The BRAMs are 20 Kilobits in size and can be configured as 512x40, 1024x20 and 2048x10. Both single port and dual port modes are supported.
- **Interconnect:** The routing channel width is 320. Wire segments of length 4 and 16 are used. The switch block is a Wilton Switch box with a flexibility of 3.

To this baseline FPGA architecture, we add Compute RAM blocks to create the proposed FPGA architecture. We evaluate the area and delay parameters of a Compute RAM block and plug in the description of a Compute RAM block in the FPGA architecture file. To find the area, we start from the BRAM area from [3]. We evaluate the area and delay of a 4 Kilobit RAM (instruction memory) using OpenRAM [27]. For the controller, we develop a simple pipelined processor in Verilog. We also design a logic peripheral block in Verilog. We then use Synopsys DC to synthesize these units and add a 15% overhead of placement and routing [28]. Then, we add the area of a BRAM, instruction memory, controller and logic peripherals.

To evaluate the frequency of Compute RAM block, we first note that none of the additional components added to a BRAM impact it's critical path delay. From [8], we see that the logic mode of the logic-in-memory RAM runs at a 34% reduced frequency compared to the memory mode, owing to the reduced voltage requirement. We apply the same factor to the frequency of operation of the BRAM to obtain the frequency of operation of Compute RAM.

### C. Experimental Setup

The goal of our experiments is to evaluate the benefit of using Compute RAMs instead of baseline FPGA blocks (LBs, DSPs and BRAMs) for common operations like addition, multiplication and dot product. We compare various metrics: area consumed, energy and total time taken. We use the most widely used precisions in FPGA DL accelerators: int4, int8 and bfloat16. However, it should be noted that Compute RAMs are fully adaptable to any precision.

The Verilog designs used for the experiments include:

- Memory to store the inputs and outputs.
- Compute units for performing the computation (LBs in case of fixed-point addition, and DSPs in other cases).
- Control logic to coordinate movement of operands and results between compute units and memory.

In case of a baseline FPGA, we assume the design contains 1 BRAM (20 Kbits in 512x40 geometry) and that the data is laid out in the BRAM in the most optimal way to ensure maximum bandwidth usage. We also instantiate enough compute units to saturate the bandwidth from 1 BRAM. For example, for int4 addition operation, one row contains 3 input-output tuples (operand1, operand2, result), one row is read out in 1 cycle and the data is fed to 3 adders. For bfloat16 multiplication operation, three rows contain the operands and the results of 2 operations (row1 ->{operand1, operand2},

row2 ->{operand3, operand4}, row3 ->{result1, result2}). Only 1 bfloat16 adder is enough to saturate the bandwidth provided by the BRAM. This is the most optimal configuration and ensures a fair comparison.

In case of the FPGA with Compute RAMs, most of the design is absorbed in a Compute RAM block. A Compute RAM block with 20 Kilobits capacity in the main array, with a geometry of 512x40 is used. We assume that the data is laid out in transpose format in the main array.

We run VTR with these designs and the baseline and proposed architectures to observe the area, delay/frequency and routing wirelength metrics. We run VTR without a target frequency, which means it finds the fastest implementation possible. We disable any I/O to register and register to I/O paths from timing analysis. All the areas and delays in our results are based on the 22nm technology node. In some cases, because of unavailability of 22nm standard cell libraries, we used the 45nm GPDK library from Cadence, and scale the delays and areas based on equations present in [29].

On a baseline FPGA, the total cycles for an operation to complete include the time taken to read the inputs, perform the computation and the write the results. In the case of Compute RAM based FPGA, the total cycles for an operation to complete are the cycles to execute all the instructions in the instruction memory for a given operation.

For energy, we add transistor energy and wire energy. For transistor energy, we use an activity factor of 0.1 and calculate the energy based on the number of transistors in each block (obtained from the area consumed by the block). For wire energy, we use wire energy numbers (fJ/mm/bit) from [30], scale them to 22nm technology node and multiply them with the number of bits used for data transfer and the average net length obtained from VTR.

## V. Results

### A. Properties of Compute RAM

Table II compares the various properties of a Compute RAM block with a DSP slice, a BRAM and a Logic block. We observe that a Compute RAM has ∼33% more area compared to a BRAM. The additional overhead comes from the existence of components like the instruction memory, controller and peripheral logic. A DSP Slice has ∼12% more area than a Compute RAM block.

Compute RAMs are ∼37% slower than BRAMs because of the lower voltage requirement for logic mode operation. But they are ∼43% faster than DSPs in fixed-point mode and ∼67% faster than DSPs in floating-point mode. DSP slice is slow even though it is pipelined because it is a large block with many I/O ports and has a large input crossbar in it. Compute RAMs are smaller than DSP slices and have a smaller number of inputs compared to a DSP slice leading to a smaller local input crossbar and hence shorter delay. The path delay through the main array of the Compute RAM is shorter than the combinatorial delay through a DSP as well. The frequency of operation of a Logic block varies with the size and nature of the computation or logic mapped to it.

**TABLE II:** Comparison of Compute RAM, DSP, BRAM, and LB

| Metric | Compute RAM | DSP Slice | BRAM | Logic Block |
|---|---|---|---|---|
| Area ($\mu m^2$) | 11072.5 | 12433 | 8311 | 1938 |
| Frequency (MHz) | 609.1 | 391.8 (fixed)<br>336.4 (float) | 922.9 | Varies |
| Throughput (GOPS) (int4/int8/bfloat16) | 4.8/2.7/0.3 | 0.7/0.5/0.2 | 0 | 1.4/0.6/- |

The compute throughput (in giga operations per second (GOPS)) for different precisions can also be seen in the table. In baseline FPGA, fixed-point additions are mapped to LBs, whereas other computations are mapped to DSPs (mapping additions to DSPs is inefficient because of the lower frequency). However, Compute RAMs are efficient for all computations. The table shows the throughput value of addition or multiplication, whichever is larger. BRAMs are only used for storage, so their compute throughput is 0. We can see that Compute RAMs have the highest throughput values among all blocks.

### B. Addition

Figure 4 shows the results for addition operation. We compare various metrics for a baseline FPGA against an FPGA with Compute RAMs. The total number of addition operations in both cases are such that 20 Kilobits is required for storing all the operands and the results.

Area consumed is the total areas of all the blocks (LBs, DSPs, BRAMs, Compute RAMs) used by the circuit on the FPGA. We observe significant reduction in area for both precisions. This is because in a baseline FPGA, soft logic (multiple LBs) is used for designing the control logic, but in case of Compute RAMs, the controller is hardened.

The energy metric shows the dynamic energy consumed by the circuits mapped to the baseline FPGA and the FPGA with Compute RAMs. We see that energy consumed when using Compute RAMs is ∼20% of the energy consumed on baseline FPGA. This is because of the much lower dependence on FPGA interconnect fabric and also the reduced circuit area.

The overall time taken shown in Figure 4 is the product of cycles taken for the entire operation and the frequency of the circuit. The frequency of operation of the circuit is reported by VTR. The frequency of operation is 60-65% higher when using Compute RAMs. This is because when using a baseline FPGA, there are paths between multiple DSPs, LBs and BRAMs through the interconnect fabric. These paths tend to be long and circuitous. When using Compute RAM, a very few short timing paths exist outside the Compute RAM.

For int8 precision, we see a significant reduction in time taken, because the number of cycles taken by Compute RAM is lower than the cycles taken on the baseline FPGA. However, for `bfloat16`, the time taken is only 20% smaller. The number of cycles taken by Compute RAM is indeed larger in this case, because floating point addition requires a lot of steps. However, the overall time is still lower because the frequency of operation is much higher for Compute RAM.

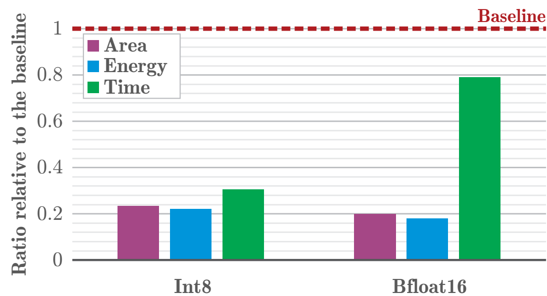


**Fig. 4:** Comparing a baseline FPGA with an FPGA with Compute RAMs for addition operation (RAM arrays are 512x40)

### *C. Multiplication*

Figure 5 shows the results for multiplication operation. The total number of multiplication operations is such that 20 Kilobits is required for storing all the operands and the results.

The area and energy results for multiplication are very similar to addition. The total time taken for multiplication operations is ∼12% shorter for Compute RAMs than the baseline FPGA. Because of the bit-serial nature of the computation done by Compute RAMs, the number of cycles taken for multiplications is quite high. However, the overall time is still lower because the frequency of operation is much higher for Compute RAM.

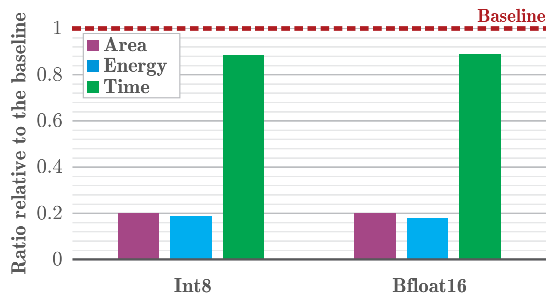


**Fig. 5:** Comparing a baseline FPGA with an FPGA with Compute RAMs for multiplication operation (RAM arrays are 512x40)

### *D. Dot Product*

Dot product operation is the building block of neural networks. It is used in matrix-matrix multiplication and matrix-vector multiplication, which form 80-90% of all computations in modern neural networks. Layers such as fully connected, convolution and LSTM are all based on these operations. Many FPGA-based hardware accelerators, ASIC chiplets and FPGAs have dot product engines in them [31] [32] [33]. Dot product operation involves MAC and reduction operations. Two vectors are multiplied element-wise and the products are added to produce a scalar output.

In this section, we show the results of a dot product operation using int4 precision. The accumulation is performed using 32-bits (typical for DL). We consider vector sizes that ensure maximum utilization of the Compute RAM and the BRAM on a normal FPGA. On a baseline FPGA, there are 5 multipliers and 4 adders for accumulation, to ensure bandwidth provided by the BRAM is fully utilized.

The area and energy results are similar to the addition and multiplication results. However, there is an interesting

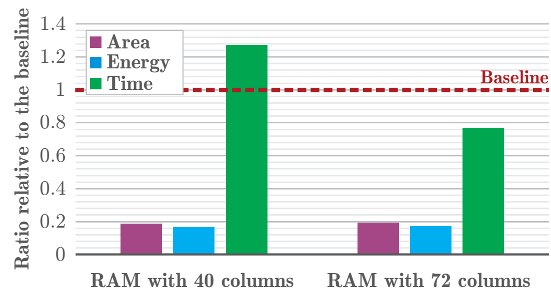


**Fig. 6:** Comparing a baseline FPGA with an FPGA with Compute RAMs for dot product operation in int4 precision

observation pertaining to the overall time taken (referring to the left half of Figure 6). Compute RAM takes more time, even with the frequency of operation being higher. This is because Compute RAM takes much larger number of cycles compared to the baseline FPGA (1470 vs 480). In the design implemented on the baseline FPGA, there is enough parallelism. There are 5 4-bit multipliers running in parallel and 4 int32 adders perfoming accumulation in a tree structure. On top of that, all these compute units are pipelined. In Compute RAM, however, the parallelism is limited to the number of columns (bit lines), which is 40. To ensure maximal data packing and utilization of the Compute RAM, we store multiple input data items in one column. But within a column, the operations are performed serially. And the number of serial operations is relatively high in this case.

To reduce the time and improve the performance of Compute RAMs, more parallelism is required. This implies that a shallower and wider memory array will be required. BRAMs in Xilinx FPGAs have wider configurations up to 72 columns [34]. We experiment with using 72 columns. We evaluate the impact of increasing the columns analytically and show it in the right half of Figure 6. We observe that there is minor impact on the area and energy metrics, but the total time is now ∼20% better than the baseline (because of almost 2x the parallelism). Even more parallelism and speedup can be achieved if we increase the number of columns even further (say 40 rows x 512 columns, instead of 512 rows x 40 columns), but such a memory array will be expensive because large number of I/O ports, leading to significant changes in the interconnect architecture of the FPGA. We leave further detailed investigation of this topic as future work.

## VI. Conclusion

This paper proposes adding blocks called Compute RAMs to FPGAs to improve their performance for DL applications. A Compute RAM block enables processing-in-memory by utilizing an emerging bit-line SRAM circuit technology coupled with bit-serial arithmetic. Each individual operation is performed serially, but multiple operations are done in parallel in the same block. This unlocks performance benefits for parallel compute-intensive operations which tend to involve a lot of on-chip data movement if implemented on current FPGAs using Logic Blocks, DSP slices and BRAMs.

We present the architecture of Compute RAM blocks, propose adding them to FPGAs, and describe how computations

can be orchestrated using them. We demonstrate the efficacy of these blocks for common DL math operations.

**We believe that Compute RAMs can replace BRAMs on existing FPGAs, and transform them into massively parallel computation units, while still performing the traditional role of acting as storage units.** In the future, we plan to evaluate the performance boost that can be obtained at the application level (neural networks) by using these blocks. We also plan to explore using shallower, wider RAMs to increase the amount of parallelism and speedup.

Note that Compute RAMs are not limited to using this technology; other memory technologies that support bit-level computing can also be used to design Compute RAMs.

## References


[1] Xilinx. (2018) Xilinx AI Engines and Their Applications. [Online]. Available: https://www.xilinx.com/support/documentation/white_papers/wp506-ai-engine.pdf

[2] M. Langhammer *et al.*, "Stratix 10 NX Architecture and Applications," in *International Symposium on Field-Programmable Gate Arrays (FPGA)*, 2021.

[3] A. Arora *et al.*, "Tensor Slices to the Rescue: Supercharging ML Acceleration on FPGAs," in *International Symposium on Field-Programmable Gate Arrays (FPGA)*, 2021.

[4] S. Wang and P. Kanwar. BFloat16: The Secret to High Performance on Cloud TPUs. https://cloud.google.com/blog/products/ai-machine-learning/bfloat16-the-secret-to-high-performance-on-cloud-tpus.

[5] I. Kuon and J. Rose, "Measuring the Gap Between FPGAs and ASICs," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 26, no. 2, pp. 203–215, Feb 2007.

[6] Intel. (2015) Stratix 10 FPGA Features. [Online]. Available: https://www.intel.com/content/www/us/en/products/programmable/fpga/stratix-10.html

[7] ——. (2019) Intel Agilex FPGAs and SOCs. [Online]. Available: https://www.intel.com/content/www/us/en/products/programmable/fpga/agilex.html

[8] S. Jeloka, N. B. Akesh, D. Sylvester, and D. Blaauw, "A 28 nm Configurable Memory (TCAM/BCAM/SRAM) Using Push-Rule 6T Bit Cell Enabling Logic-in-Memory," *IEEE Journal of Solid-State Circuits*, vol. 51, no. 4, pp. 1009–1021, 2016.

[9] S. Aga, S. Jeloka, A. Subramaniyan, S. Narayanasamy, D. Blaauw, and R. Das, "Compute Caches," in *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2017, pp. 481–492.

[10] C. Eckert, X. Wang, J. Wang, A. Subramaniyan, R. Iyer, D. Sylvester, D. Blaauw, and R. Das, "Neural Cache: Bit-Serial in-Cache Acceleration of Deep Neural Networks," in *Proceedings of the 45th Annual International Symposium on Computer Architecture*, ser. ISCA '18. IEEE Press, 2018, p. 383–396. [Online]. Available: https://doi.org/10.1109/ISCA.2018.00040

[11] Achronix. (2019) Speedster7t FPGAs. [Online]. Available: https://www.achronix.com/product/speedster7t/

[12] Flex-Logix. (2019) Flex-Logix nnMAX Inference Acceleration Architecture. [Online]. Available: https://flex-logix.com/wp-content/uploads/2019/09/2019-09-nnMAX-4-page-Overview.pdf

[13] M. Eldafrawy, A. Boutros, S. Yazdanshenas, and V. Betz, "FPGA Logic Block Architectures for Efficient Deep Learning Inference," *ACM Trans. Reconfigurable Technol. Syst.*, vol. 13, no. 3, Jun. 2020. [Online]. Available: https://doi.org/10.1145/3393668

[14] A. Boutros, S. Yazdanshenas, and V. Betz, "Embracing Diversity: Enhanced DSP Blocks for Low-Precision Deep Learning on FPGAs," in *2018 28th International Conference on Field Programmable Logic and Applications (FPL)*, 2018, pp. 35–357.

[15] S. Rasoulinezhad, H. Zhou, L. Wang, and P. H. W. Leong, "PIR-DSP: An FPGA DSP Block Architecture for Multi-precision Deep Neural Networks," in *2019 IEEE 27th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, 2019, pp. 35–44.

[16] S. Ghose, A. Boroumand, J. S. Kim, J. Gómez-Luna, and O. Mutlu, "Processing-In-Memory: A Workload-Driven Perspective," *IBM Journal of Research and Development*, vol. 63, no. 6, pp. 3:1–3:19, 2019.

[17] A. Shafiee, A. Nag, N. Muralimanohar, R. Balasubramonian, J. P. Strachan, M. Hu, R. S. Williams, and V. Srikumar, "ISAAC: A Convolutional Neural Network Accelerator with In-Situ Analog Arithmetic in Crossbars," in *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*, 2016, pp. 14–26.

[18] P. Chi, S. Li, C. Xu, T. Zhang, J. Zhao, Y. Liu, Y. Wang, and Y. Xie, "PRIME: A Novel Processing-in-Memory Architecture for Neural Network Computation in ReRAM-Based Main Memory," in *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*, 2016, pp. 27–39.

[19] M. Imani, S. Gupta, Y. Kim, and T. Rosing, "FloatPIM: In-Memory Acceleration of Deep Neural Network Training with High Precision," in *Proceedings of the 46th International Symposium on Computer Architecture*, ser. ISCA '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 802–815. [Online]. Available: https://doi.org/10.1145/3307650.3322237

[20] F. Cai, J. M. Correll, S. H. Lee, Y. Lim, V. Bothra, Z. Zhang, M. P. Flynn, and W. D. Lu, "A fully integrated reprogrammable memristor–CMOS system for efficient multiply–accumulate operations," *Nature Electronics*, 2019.

[21] K. Y.-C. et al., "25.4 a 20nm 6gb function-in-memory dram, based on hbm2 with a 1.2tflops programmable computing unit using bank-level parallelism, for machine learning applications," in *2021 IEEE International Solid- State Circuits Conference (ISSCC)*, vol. 64, 2021, pp. 350–352.

[22] K. Tatsumura, S. Yazdanshenas, and V. Betz, "High density, low energy, magnetic tunnel junction based block RAMs for memory-rich FPGAs," in *2016 International Conference on Field-Programmable Technology (FPT)*, 2016, pp. 4–11.

[23] P. L. et al., *DSP Processor Fundamentals : Architectures and Features*. New York: IEEE Press, 1997.

[24] K. E. Murray, O. Petelin, S. Zhong, J. M. Wang, M. ElDafrawy, J.-P. Legault, E. Sha, A. G. Graham, J. Wu, M. J. P. Walker, H. Zeng, P. Patros, J. Luu, K. B. Kent, and V. Betz, "VTR 8: High Performance CAD and Customizable FPGA Architecture Modelling," *ACM Trans. Reconfigurable Technol. Syst.*, 2020.

[25] Synopsys. (2018) Synopsys VCS. [Online]. Available: https://www.synopsys.com/verification/simulation/vcs.html

[26] ——. (2018) Synopsys Design Compiler. [Online]. Available: https://www.synopsys.com/implementation-and-signoff/rtl-synthesis-test/dc-ultra.html

[27] M. R. Guthaus, J. E. Stine, S. Ataei, Brian Chen, Bin Wu, and M. Sarwar, "OpenRAM: An open-source memory compiler," in *2016 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, 2016, pp. 1–6.

[28] C. Ho, C. Yu, P. Leong, W. Luk, and S. Wilton, "Domain-Specific Hybrid FPGA: Architecture and Floating Point Applications," 09 2007, pp. 196 – 201.

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

[30] S. W. Keckler, W. J. Dally, B. Khailany, M. Garland, and D. Glasco, "GPUs and the Future of Parallel Computing," *IEEE Micro*, vol. 31, no. 5, pp. 7–17, 2011.

[31] J. Fowers *et al.*, "A Configurable Cloud-scale DNN Processor for Real-time AI," in *Proceedings of the 45th Annual International Symposium on Computer Architecture*, ser. ISCA '18. Piscataway, NJ, USA: IEEE Press, 2018, pp. 1–14. [Online]. Available: https://doi.org/10.1109/ISCA.2018.00012

[32] A. Boutros *et al.*, "Beyond Peak Performance: Comparing the Real Performance of AI-Optimized FPGAs and GPUs," in *International Conference on Field Programmable Technology (FPT)*, 2020.

[33] E. Nurvitadhi, S. Shumarayev, A. Dasu, J. Cook, A. Mishra, D. Marr, K. Nealis, P. Colangelo, A. Ling, D. Capalija, and U. Aydonat, "In-Package Domain-Specific ASICs for Intel® Stratix® 10 FPGAs: A Case Study of Accelerating Deep Learning Using TensorTile ASIC," 02 2018, pp. 287–287.

[34] Xilinx. (2021) UltraScale Architecture Memory Resources. [Online]. Available: https://www.xilinx.com/support/documentation/user_guides/ug573-ultrascale-memory-resources.pdf