2021 IEEE 29th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM) | 978-1-6654-3555-0/20/$31.00 ©2021 IEEE | DOI: 10.1109/FCCM51124.2021.00018

Compute-Capable Block RAMs for Efficient Deep Learning Acceleration on FPGAs

Xiaowei Wang, Vidushi Goyal, Jiecao Yu, Valeria Bertacco, Andrew Boutros†,
Eriko Nurvitadhi†, Charles Augustine†, Ravi Iyer†, Reetuparna Das
University of Michigan †Intel Corporation
{xiaoweiw, vidushi, jiecaoyu, valeria, reetudas}@umich.edu,
{andrew.boutros, eriko.nurvitadhi, charles.augustine, ravishankar.iyer}@intel.com

***Abstract*—The density of FPGA on-chip memory has been continuously increasing with modern FPGAs having thousands of block RAMs (BRAMs) distributed across their reconfigurable fabric. These distributed BRAMs can provide a tremendous amount of on-chip bandwidth for efficient acceleration of data-intensive applications. In this work, we propose enhancing the ubiquitous FPGA BRAMs with in-memory compute-capabilities. As a result, BRAMs can act as normal storage units or their bitlines can be re-purposed as SIMD lanes executing bit-serial arithmetic operations. Our proposed architectural change results in 1.6× and 2.3× increase in the peak multiply-accumulate throughput of a large Stratix 10 FPGA, at a minimal cost of only 1.8% increase in the FPGA die size and no change to the BRAM's interface to the programmable routing. Then, we present RIMA, a reconfigurable in-memory accelerator architecture for deep learning (DL) inference. RIMA exploits the proposed compute-capable BRAMs and the FPGA's reconfigurability to achieve 1.25× and 3× higher performance compared to the state-of-the-art Brainwave DL soft processor for 8-bit integer and block floating-point precisions, respectively. In addition, RIMA implemented on a Stratix 10 FPGA enhanced with compute-capable BRAMs can achieve an order of magnitude higher performance compared to a same-generation GPU.**

I. Introduction

Over the past three decades, field-programmable gate arrays (FPGAs) have evolved from simple arrays of reconfigurable logic with routing capabilities into complex heterogeneous systems with several hard blocks [29]. On-chip memories are one of the most important components of modern FPGAs. In state-of-the-art FPGAs, users can flexibly configure thousands of block RAMs (BRAMs) to operate at different modes, widths, and depths depending on the their application needs. For example, the largest monolithic Intel Stratix 10 device has 11,721 BRAMs that collectively provide 229 Mb of distributed on-chip memory [15]. Future generations of FPGA devices are also expected to have higher density of on-chip BRAMs to satisfy the ever increasing demands of memory-intensive datacenter workloads [20].

The distributed nature of the FPGA on-chip memories, along with the flexibility of the reconfigurable fabric surrounding them, provides tens of TB/s of on-chip bandwidth between the BRAMs and the compute units implemented in the FPGA's logic or digital signal processing (DSP) blocks. With the prevalence of FPGAs in datacenters, these unique properties have positioned FPGAs as a fertile choice for accelerating many data-intensive workloads such as deep learning (DL). By keeping all the model weights persistent in the large on-chip memory and employing the compute-near-memory paradigm, FPGAs were able to achieve an order of magnitude higher performance than same-generation graphics processing units (GPUs) on memory-bound DL inference workloads [6].

In this work, we aim to bring the compute even closer to the data by proposing a new compute-capable BRAM architecture for FPGAs. Previous work has demonstrated the potential to accelerate a variety of applications with in-memory computation in the context of CPU caches [8], [11]. In FPGAs, the gains of compute-capable BRAMs are compelling for three main reasons: (1) The large number of distinct memory blocks in an FPGA enables a significantly higher degree of parallelism. The BRAMs in the largest monolithic Intel Stratix 10 FPGA can be re-purposed to over 1.5 million bit-serial SIMD compute lanes when enhanced with compute-capable peripherals. When not computing, BRAMs can still function as normal memory units. (2) The in-fabric distributed nature of the FPGA on-chip memory enables tighter integration of the compute-capable memories with the rest of the design logic. For operations that cannot be computed in-memory, the data from a compute-capable last-level cache (LLC) in a CPU would still need to go through the different levels of the memory hierarchy and the fixed processor pipeline to reach the compute units, which is not the case for compute-capable FPGA BRAMs. (3) Performing the compute in BRAMs eliminates the data movement between memory and compute units which reduces the competition over the fabric's precious routing resources. This can potentially alleviate congestion and increase the routability of FPGA designs.

Our evaluation shows that enhancing a modern Stratix 10 FPGA architecture with compute-capable BRAMs can result in 1.6× and 2.3× increase in the device's peak multiply-accumulate (MAC) throughput for 8-bit integer and block floating-point precision, respectively. These gains are achieved with no change to the modes of operation or routing interface of existing BRAMs and at a minimal cost of 1.8% increase in the FPGA die size. We then evaluate the performance gains of our proposed architectural modification when accelerating a variety of real-time DL workloads.

To the best of our knowledge, this work is the first attempt to:

- Evaluate the performance gains and area cost of enhancing modern FPGAs with in-BRAM compute capabilities,
- Implement a matrix-vector multiplication engine that combines and balances the use of in-BRAM and DSP compute units to maximize the overall throughput,
- Quantify the performance gains of FPGAs with compute-capable BRAMs on a variety of real-time DL workloads.

II. Background: In-SRAM Bit-Serial Compute

Prior work has investigated the use of SRAM memory arrays for performing bit-serial, in-memory computations. Neural Cache [8], which we build our work on, re-purposes the SRAM in the last-level cache of commodity processors, as bit-serial arithmetic units. In Neural Cache, the studied CPU model has 14 LLC slices, and there are 320×8 KB SRAM arrays in each slice. Each array contains 256 wordlines and 256 bitline pairs. To enable *in-situ* compute inside the SRAM array, an additional row-decoder is added to the memory array to enable the activation of two wordlines at the same time. The design of the sense amplifiers is also modified such that, by sensing the shared bitlines, it is possible to perform a bitwise Boolean operation, `and` or `nor`, between

978-1-6654-3555-0/21/$31.00 ©2021 IEEE
DOI 10.1109/FCCM51124.2021.00018

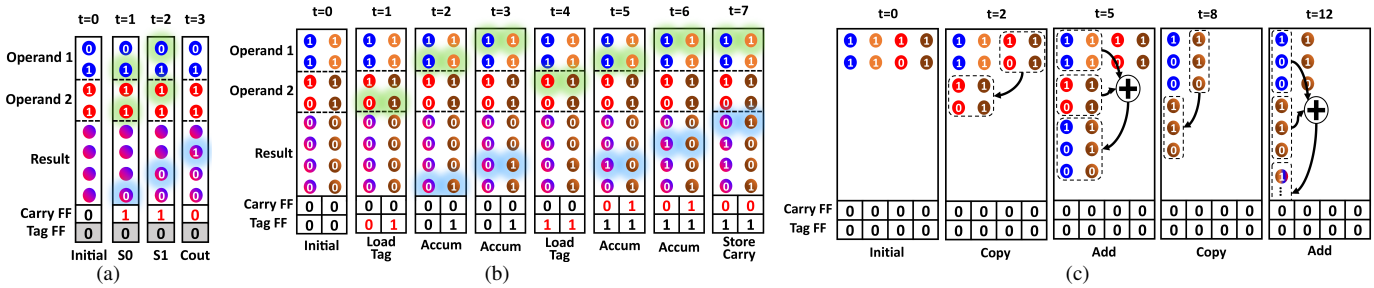


Fig. 1: Steps for different in-SRAM bit-serial (a) addition, (b) multiplication, and (c) reduction operations. Green and blue shades correspond to read and write activated wordlines at each time step. The bits from a single data word are marked with the same color.

TABLE I: Number of cycles for in-SRAM bit-serial operations where n is the operands' bitwidth and k is the number of reduced elements.

Operation	# Cycles
Addition	$n+1$
Multiplication	n^2+3n-2
Reduction	$(2n+\log_2 k)\cdot\log_2 k$

the two activated wordlines. For the activated cells attached to the same bitline pair (BL and $\overline{BL}$), sensing the BL performs an `and` operation (i.e. sensed result is high only if both cells store value 1) and sensing the $\overline{BL}$ performs a `nor` operation (i.e. sensed result is high only if both cells store value 0). With these modifications to the SRAM array peripherals, a 256-bit-wide logical (`and/nor`) operation can be performed every cycle.

To perform more complex computations with a high degree of parallelism, Neural Cache proposes a transposed data layout and bit-serial arithmetic algorithms for addition, multiplication and reduction operations. In the transposed layout, all the bits of one data word are stored vertically in cells attached to the same bitline, such that bit i of all the data words map to the same wordline. With this transposed format, a region with n wordlines and 256 bitlines in an SRAM array can hold 256 n-bit operands. Arithmetic operations are built out of logical operations in a bit-serial fashion, where each bitline acts as a SIMD lane. Full adder circuitry is added to the bitline peripherals to compute the sum and carry bits out of the `and` and `nor` results from the sense amplifiers. Fig. 1 and Table I summarize the steps of each arithmetic operation and the number of cycles it takes as explained below.

For addition, in each cycle, the two wordlines of the same bit position of the two operands are activated. Then, the bit-line peripheral computes the sum bit, updates the carry for the next cycle computation, and writes back the sum bit to the result location before proceeding to the next bit position in the next cycle. Thus, the addition for n-bit operands takes $n+1$ cycles. Bit-serial multiplication is based on iterative addition of partial results. In each iteration, one bit of the first operand is loaded as a *tag* value, and the second operand's bits are added to the partial sum only if the tag value is 1, as shown in Fig. 1b. In total, the multiplication takes n^2+3n-2 cycles. Finally, the reduction operation is implemented as a series of across-bitline copies and additions. In each iteration, the number of operands is halved, until the final result is generated. Note that other common INT/FP arithmetic operations as well as transcendental bit-serial in-array operations can also be implemented [11]. With the modified peripherals and the right control sequence, the 35 MB LLC in [8] operates as a massively parallel ALU with $1,146,880$ single-bit SIMD lanes.

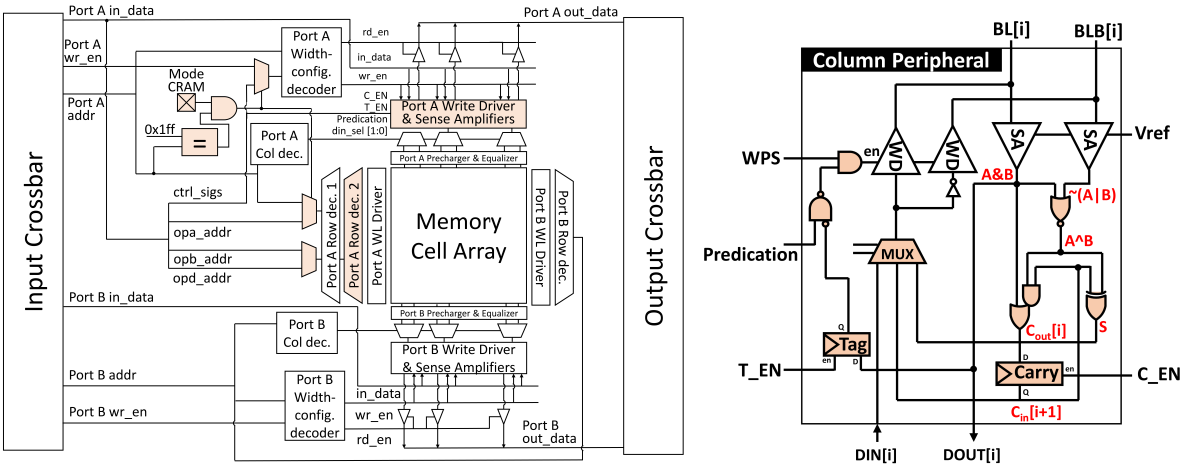
III. Compute-Capable Block RAMs for FPGAs

A. Enhanced BRAM Architecture

Fig. 2a shows the proposed BRAM architecture following the dual-port 1-bank SRAM array design from [30]. Blocks added or modified to enhance the FPGA BRAM with the in-memory compute capabilities are highlighted in a different color. One of our main design targets is to implement all the necessary changes while reusing the same existing BRAM interface (i.e. keep the same number of input/output ports) to minimize the area overhead of the proposed changes and avoid stressing the routing from/to the BRAMs. Therefore, as highlighted in Fig. 2a, all the changes in the proposed BRAM architecture are limited to the block's internals.

The two major changes are the addition of: (1) a second row decoder to one of the two BRAM ports to allow the simultaneous activation of two memory cells that share the same bitline and (2) the bit-serial compute logic to the peripherals of each bitline. Fig. 2b shows the circuitry of the modified bitline peripheral. On the read path, the outputs of the sense amplifiers (SA) go through a few logic gates to create the sum and carry bits for adding the two activated memory cells. The carry bit is stored in the carry flip-flop (FF) to be used in the following cycle's computation. The sum bit is routed to the write path to be stored back to the memory array in the location reserved for the results. The output of the SA can be passed directly to the output for a normal read operation, or stored in the tag FF used in the bit-serial multiplication operation. On the write path, a 3:1 multiplexer is added to choose between different sources for a memory cell write: input data bit (for a normal write operation), sum bit (for addition), or carry bit (for the final carry out value). We also add logic circuitry to allow enabling/disabling the write drivers (WD) based on the tag value, which is key for implementing the bit-serial multiplication algorithm described in Section II. Our SPICE simulations using 28 nm process technology at 0.9V supply voltage show that the added circuitry results in a 1.6× increase in the BRAM cycle time only when it is operating in the compute mode. In the memory mode, the 3:1 multiplexer in Fig. 2b is the only circuitry added on the normal write path. However, this added negligible delay and did not affect the BRAM speed in the conventional memory mode.

The proposed compute-capable BRAM can function in the conventional *memory mode* or the new *compute mode*. The mode of operation is determined by an additional configuration SRAM (CRAM) cell for each BRAM tile, which is configured during programming the FPGA. These added configuration bits represent a negligible 12 Kb increase to the 577 Mb compressed bitstream of the largest Stratix 10 device. When the compute-capable BRAM is configured in memory mode, its operation is exactly the same as that of a conventional BRAM and the designer can flexibly configure the number of ports and the width/depth of the BRAM. On the other hand, when configured in compute mode, the BRAM is automatically configured to its maximum width to maximize the read/write throughput for populating the memory array with input data and reading the final results in transposed format. In addition, a special address (`0x1ff`) is reserved; any data word written to the reserved address is treated as an in-memory compute instruction, while accesses to other addresses are processed normally. In this case, the row address, the SRAM array access control signals (including `wr_en`), and the bitline peripheral control signals are all packed into the incoming instruction. To support that, we insert several multiplexers (shown in Fig. 2a) to select the set of right



(a) (b)

Fig. 2: Block diagrams for: (a) internal architecture of the compute-capable dual-port single-bank BRAM, and (b) peripheral logic circuitry added for each bit-line of the memory cell array. The added/modified circuitry is highlighted in color.

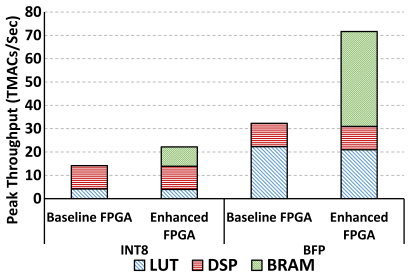


Fig. 3: Peak MAC throughput gains when enhancing a baseline Stratix 10 GX2800 FPGA with compute-capable BRAMs for 8-bit integer and block floating-point precisions.

control signals based on the operation mode, and a comparator to determine if the input address is equal to the reserved one. These instructions are generated by a control finite-state machine (FSM) implemented in the soft logic and supplied to the `in_data` port of the BRAM. The control FSM is lightweight; it utilizes less than ∼300 look-up tables (LUTs), which can be amortized by sharing the FSM across multiple BRAMs working in lockstep.

B. Peak MAC Throughput Gain

In this subsection, we evaluate the peak MAC throughput gains achieved by enhancing the largest Stratix 10 (GX 2800) FPGA architecture with our proposed compute-capable BRAMs. This study gives the workload-agnostic performance gains before evaluating the achievable gains with real benchmarks. We experiment with two numerical precisions: 8-bit integer (INT8) and block floating-point (BFP8) with 1 sign bit, 2 mantissa bits, and 5 exponent bits (1s.2m.5e), which is used in various DL workloads [10]. For the INT8 experiments, we assume 27-bit accumulation as in [19].

To calculate the peak throughput of the soft logic, we synthesize, place and route one MAC unit to LUTs to determine its operating frequency and resource utilization. We then calculate the total LUT throughput by optimistically assuming that we can fill all the available LUTs with MAC units at the same operating frequency. This is an unrealistic assumption as it does not consider the routability and frequency degradation as we fill the device, but serves the purpose of this peak throughput study. In addition, we run simple Quartus experiments to determine the the maximum operating frequencies for DSP blocks in the 2× integer MACs mode and BRAMs in the simple dual-port mode, which are found to be 866 MHz and 998 MHz, respectively. This means that the BRAMs would run at a maximum frequency of 624 MHz when configured in the compute mode (1.6× slower than memory mode). A single MAC operation implemented using the in-BRAM bit-serial algorithms takes 113 and 23 cycles in case of INT8 and BFP8, respectively.

Fig. 3 shows the peak throughput in TMACs/sec for the baseline and enhanced Stratix 10 FPGA architectures for the two studied precisions. It shows that compute-capable BRAMs can deliver an additional **8.3 INT8 TMACs/sec** and **40.7 BFP8 TMACs/sec**, enhancing the peak device throughput by a factor of **1.6**× and **2.3**× for INT8 and BFP8, respectively. These gains come with minor degradation to the contribution of LUTs to the peak device throughput since, with every 64 BRAMs sharing the same control FSM, all the compute-capable BRAMs in the device utilize ∼55k LUTs (less than 6% of the device resources).

C. Area Overhead and CAD Support

The area overhead due to the added row decoder and column peripherals for bitline computing is estimated to be 7.5% for a 64 Kb SRAM array in 28 nm process technology [8]. This was then verified by a fabricated prototype chip in [27]. Since the BRAM's interface to the programmable routing is not changed at all when adding the in-memory compute capabilities, we simply rely on the data produced by the COFFE automatic transistor sizing tool for FPGA circuitry [25] to quantify the FPGA BRAM tile area overhead. According to this data, a 64 Kb SRAM array in an FPGA BRAM tile is $11,016\ \mu m^2$ in a similar 22 nm process technology and therefore the area of added compute circuitry would be $826\ \mu m^2$ for its 256 wordlines and 256 bitlines. Since the added circuitry size scales linearly with the number of wordlines (the extra decoder) and bitlines (the column peripherals), then the overhead for an M20k block in a Stratix 10 architecture with 128 wordlines and 128 bitlines (excluding ECC bits) would be $413\ \mu m^2$. This represents a **7.4% increase in the BRAM tile area** according to the data generated by COFFE [25]. With BRAMs occupying ∼25% of the die size of modern FPGAs with traditional compositions [24], this overhead corresponds to only **1.8% increase in the FPGA die size**.

In addition, enhancing BRAMs with in-memory compute capabilities does not require any significant CAD support. In an FPGA design tool like Quartus, an RTL designer usually instantiates a memory block from the library of vendor-supplied IPs [16]. For compute-capable BRAMs, a new IP is added to the IP catalog that maps to one or more BRAMs. The designer only needs to instantiate the new IP wherever necessary in the design. Then, the synthesis engine can directly map the instantiated IP to BRAMs and implements its associated control FSM in soft logic.

IV. RIMA: DL Reconfigurable In-Memory Accelerator

In this section, we introduce our DL reconfigurable in-memory accelerator, or RIMA for short. RIMA utilizes our proposed compute-capable BRAMs to achieve higher DL inference throughput. It also exploits the FPGA reconfigurability by customizing the workload balance between compute-capable BRAMs and DSPs to further optimize performance for each workload.

A. Target DL Workloads

RIMA is targeted for accelerating recurrent neural networks (RNNs). These networks process sequence inputs such as speech samples or sentences and are typically used in natural language processing and machine translation. They consist of multiple matrix-vector multiplications followed by vector operations known as *gates*. Different variations of RNNs include vanilla RNNs, gated recurrent units (GRUs), and long short-term memories (LSTMs) with 2, 6, and 8 vector-matrix multiplications per time step. The multiple matrix-vector multiplications typically consume the majority of the compute time of an RNN. With no data dependencies between the matrix-vector multiplications of the same time step, they can be combined into a larger matrix-vector multiplication, in which the input is a vector of C elements, the weight matrix is of size $R \times C$, and the output is a vector of R elements.

B. Accelerator Architecture

Our accelerator adopts a similar architecture as that of the Brainwave neural processing unit (NPU), a state-of-the-art FPGA overlay for DL acceleration [10], [19]. However, RIMA has two key differences compared to the NPU: (1) it utilizes compute-capable BRAMs as massively parallel SIMD lanes in the matrix-vector multiplication engine, and (2) it customizes its architecture parameters and instruction sequences for each specific workload instead of being a *one-size-fits-all* software programmable overlay. In this work, we apply per-workload architecture customization to maximize the overall performance by balancing between in-BRAM and DSP compute. Fig. 4a illustrates the top-level organization of our accelerator. It consists of five pipeline stages: the matrix-vector multiplication unit (MVU), the external vector register file (eVRF) for skipping the MVU when necessary, two identical multi-function units (MFUs) for vector elementwise operations (e.g. activation, addition, subtraction), and finally the loader (LD) which writes back to any of the VRFs. RIMA uses the same eVRF, MFUs, and LD blocks as in [19], but re-designs the MVU (the key compute complex) to exploit the compute-capable BRAMs in our proposed enhanced FPGA architecture.

The MVU, as illustrated in Fig. 4a, consists of T tiles followed by an inter-tile global reduction tree to generate the final MVU output. As shown in Fig. 4b, each tile consists of dot product engines (DPEs) that compute dot product operations between a portion of the input vector and several rows of the weight matrix. There are two types of DPEs in each tile; the logic DPEs (L-DPEs) that implement an array of multipliers and an adder tree to accumulate the products using LUTs and DSP blocks, and memory DPEs (M-DPEs) that perform the same operation using compute-capable BRAMs. The two sets of DPEs work in tandem to fully exploit the computational resources in our proposed FPGA architecture with in-BRAM compute capabilities. To further optimize performance, the M-DPEs are designed such that they belong to a different clock domain than the rest of the architecture, since the M-DPEs can perform in-memory computations at a much faster clock speed due to their reduced routing utilization and simple control logic. The values of the weight matrix used by each L-DPE are pinned in a tightly coupled matrix register file (MRF) implemented using conventional memory-mode BRAMs. The M-DPEs do not need an MRF since the data is stored and processed in-place in the compute-capable BRAMs. Additionally, each tile has two VRFs (instead of one in the baseline NPU) that store the same part of the input vector in different layouts and supply inputs to the L-DPEs and M-DPEs at different rates due to their different compute styles.

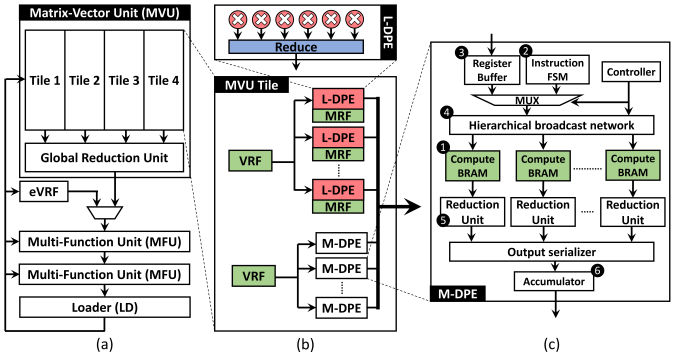


Fig. 4: Block diagrams showing: (a) overview of the RIMA architecture, (b) our proposed hybrid MVU tile, and (c) the architecture of an M-DPE, with compute-capable BRAMs.

C. Memory Dot-Product Engines

Fig. 4c depicts the internal architecture of an M-DPE. The core of the M-DPEs is an array of compute-capable BRAMs (❶). Unlike an L-DPE that computes a single dot product operation between part of the input vector and a single matrix row, multiple dot product operations with multiple matrix rows are mapped to an M-DPE to maximize the degree of parallelism on the BRAM bitlines. Each M-DPE has one Instruction FSM (❷) that sequentially generates the required bitline compute instructions. Instructions are broadcast to all the BRAMs in an M-DPE, as they all act as SIMD lanes executing the same instructions in lockstep. We implement a set of registers (❸) that buffer a complete data word coming from the VRF while the controller sequentially loads the appropriate portions of it to the input-reserved locations in the BRAMs. We use a pipelined tree interconnect network (❹) to broadcast instructions and input vector values to all BRAMs in an M-DPE without stressing the FPGA routing with such high fanout connections. This broadcast network is time-shared between the register buffer and instruction FSM, as the data loading and computation never occur simultaneously. The output of each BRAM is fed to a reduction unit (❺) where the results on multiple bitlines are added together as detailed later in Section IV-D. Finally, the results from all reduction units are serialized and passed to an accumulator (❻) that performs any necessary additions across different BRAMs to produce the final results of the M-DPE.

D. Transposed Integer Arithmetic

We discuss our implementation of the dot product operation of two vectors in compute-capable BRAMs for INT8 and BFP formats in this subsection and the next, respectively.

1) ***Data Mapping****:* In the transposed data mapping, each INT8 value is mapped to 8 memory cells attached to the same bitline (i.e. across 8 consecutive memory words), and different values in an operand vector are mapped to consecutive bitlines. Once all bitlines in the SRAM array are filled, the remaining values in the operand vector can be mapped to another set of 8 wordlines. The elements of the other operand vector are mapped similarly such that each two corresponding elements (i.e. elements in the same position) of the two vectors are mapped to memory cells on the same bitline. For example, for a memory cell array with 128 bitlines and a dot product between input and weight vectors of length 256 elements, the first 128 weights are mapped to all bitlines of wordlines 1-8 and weights 129-256 are mapped to wordlines 9-16. Similarly, the 256 inputs are mapped to word-lines 17-32.

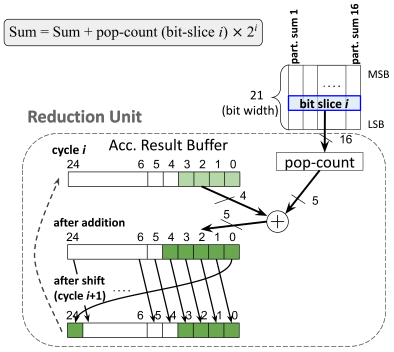


Fig. 5: External reduction unit for summing up 16 partial results. Every cycle a bit slice from all values is accumulated with pop-count, addition, and shift.

*2) **Operand Loading**:* Before any computation starts, the weight matrix and input vector need to be loaded to the memory array. For persistent-style DL, the weights remain unchanged in the on-chip memory during inference. Therefore, they can be transposed and loaded into the memory array offline in a preprocessing step. On the other hand, the input values are supplied by the user in the MVU VRFs and loaded to the compute-capable BRAMs from the register buffer (see Fig. 4c). In our implementation, the register buffer holds 32 consecutive values in the input vector (one VRF word) that, when loaded to the compute-capable BRAM, goes through an 8:1 bit-slice selector before broadcasting. Then, the selected bit slice of the 32 values maps to 32 cells on consecutive bitlines and the same wordline.

*3) **In-BRAM MAC & Reduction**:* After loading the operands into the BRAMs, the corresponding weights and inputs are multiplied with the bit-serial algorithm, and the products are accumulated in each bitline with bit-serial addition. After the MAC, the accumulated partial sums in all bitlines are iteratively reduced until there are 16 partial sums left. Performing further in-BRAM reduction significantly degrades the BRAM compute throughput as only a successively smaller portion of the BRAM bitlines are actively performing compute during each reduction iteration. We empirically choose to reduce down to 16 partial results as this design point achieves a good balance between the BRAM throughput and the resources of the external reduction unit.

*4) **External Reduction**:* Then, the 16 remaining partial results are read out from the BRAM, one bit slice at a time (starting from the least significant bit), and sent to the external reduction unit shown in Fig. 5 to obtain the final sum. The formula for accumulating with bit-slices is $\sum_{i=0}^{N} \text{pop-count}(\textit{bit-slice } i) \times 2^i$, where N is the bitwidth of the partial results. At each cycle, 16 bits from the same position of 16 different values first go through pop-count logic (counting the number of 1's). Then, the pop-count result is left-shifted by i bits and added to the accumulated sum. Practically, a barrel shifter (i.e. with variable offset) is expensive to implement on an FPGA, so we designed the external reduction unit using a circular shift register instead, as shown in Figure 5. In each cycle, the pop-count result (5 bits representing a value between 0 and 16) is added to the last 4 bits of the current accumulation value. After addition, the bits are circularly right-shifted by 1 bit to prepare for the next cycle addition. This addition-and-shift operation repeats until the most significant bit slice is processed. The external reduction unit is pipelined with the addition in the last iteration of the in-BRAM reduction to hide its latency. The external reduction unit reads out and operates on bit slice i, while the in-BRAM reduction generates bit slice $i+1$ at the same time since the FPGA BRAMs have separate read and write wordlines.

*5) **Bit Slicing**:* We further implement bit slicing, an optimization for achieving higher utilization of bitline SIMD lanes. First, each weight/input value is partitioned into multiple *slices* (e.g. an 8-bit value is split into higher 4 bits and lower 4 bits). Then, different slices are mapped to different compute-capable BRAMs to extract a higher degree of parallelism and reduce the bit-serial processing latency. Finally, the results from different slices are shifted to the appropriate bit position and summed up in the soft logic to produce the combined results.

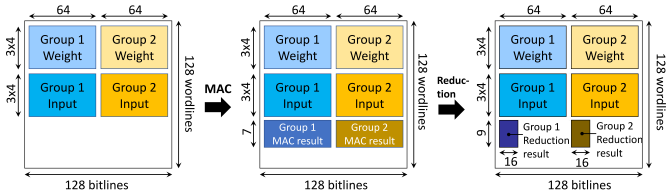


Fig. 6: In-memory operations for dot-product with block floating-point in a compute-capable BRAM. Vector has 512 elements and is partitioned into 2 exponent-sharing groups in BFP.

E. Transposed Block Floating-Point Arithmetic

The BFP data format is mainly used in DL acceleration due to its computational efficiency and adequate accuracy [10], [2], [7]. Unlike the traditional floating-point format, where each value has its own sign, mantissa and exponent, the BFP format shares the same exponent across a *block* of values. Therefore, to compute the dot product of two BFP vectors, only the mantissa bits need to be multiplied and accumulated as integer values, while the new shared exponent is calculated based on the shared exponents of operands.

As the mantissa of each BFP value has fewer bits, multiple weights and one input can be mapped to each bitline, so that the input can be reused for multiplying with different weights. In addition, each memory array is partitioned into groups of bitlines, and the elements with a shared exponent are mapped within a group for efficient accumulation as shown in Fig. 6. Bit-serial reduction is done in parallel for all the groups, and then the external reduction unit handles results from all groups sequentially. The reduction operation beyond the block size is performed using DSP blocks in floating-point mode, after the results of the external reduction unit are converted to the normal single-precision floating-point format.

V. Architecture Modeling & Customization

A. Load Partitioning and Execution

Fig. 7 illustrates how the matrix-vector multiplication is partitioned into sub-problems and mapped to the tiles and DPEs of the MVU in RIMA. The matrix and input vector are equally split (horizontally) into T *column blocks*, such that each tile is responsible for a matrix-vector multiplication sub-problem of dimensions $R \times (C/T)$. Within a sub-problem, each matrix column block is split (vertically) into several *row blocks*, such that each row block is mapped to a DPE. The number of matrix rows in a row block (i.e. row block size) differs depending on whether the row block is mapped to an L-DPE or an M-DPE. Within an M-DPE, the weights in one matrix row are mapped to one or more compute-capable BRAMs as described in Section IV-D. The number of L-DPEs and M-DPEs in a tile, as well as the portion of the matrix mapped to each type, are architectural parameters specified by the designer and can be different for each workload. All the DPEs in the same tile, regardless to their type, use the same portion of the input vector stored in the tile's VRFs. The outputs from all DPEs in a tile are concatenated to form an R-element partial result vector, then different partial results from different tiles are summed up in the global reduction unit (see Fig. 4) to produce the final output.

The execution of an RNN dataflow graph on RIMA proceeds as follows. The weights are loaded into the M-DPEs and the MRFs of L-DPEs offline as a pre-processing step. Different parts of the user-supplied input vector are first dispatched to their corresponding VRFs in different tiles. Then, all the DPEs in all

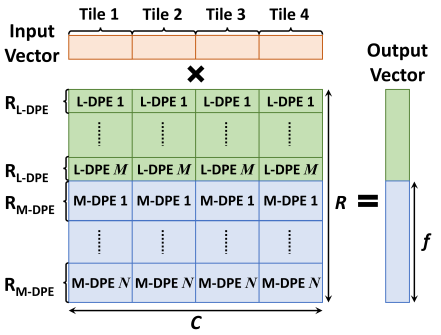


Fig. 7: Partitioning of a matrix-vector multiplication operation to an example RIMA architecture with 4 tiles, M L-DPEs and N M-DPEs.

tiles process their own sub-problem concurrently. The M-DPEs perform input loading, MAC, and reduction in parallel. Each M-DPE then performs across-BRAM accumulation, and sends out the results sequentially to reduce data transfer bandwidth. The L-DPEs load partial matrix rows and input vector from MRFs and VRF respectively, and perform the dot product computations in a pipelined fashion. Finally, after the complete output vector of the matrix-vector operation is produced, the subsequent MFUs consume it in chunks of 40 elements each to perform the vector elementwise operations depending on the workload dataflow graph. These steps are repeated again for the next time step of the RNN. For simplicity, these execution steps assume no overlap between different stages of the RIMA pipeline. For example, the first MFU does not start processing until the MVU is completely done. This is definitely sub-optimal and can be avoided by passing a part of the MVU result to the MFU while generating the next part. However, we leave this optimization for future work as we focus more on showcasing the gains of FPGA in-memory compute capabilities, rather than building the most efficient accelerator architecture.

B. Design Space Exploration & Architecture Customization

To achieve the optimal performance within the available FPGA resources, it is necessary to change the M-DPE configuration as well as the workload partitioning between the L-DPEs and M-DPEs for different problem sizes. Assigning a large portion of a large workload matrix to M-DPEs will cause all the model weights not to fit within the total chip BRAM capacity, since a portion of the BRAMs configured in compute mode must be reserved for the intermediate and final computation results. On the other hand, assigning only a small portion of the matrix to M-DPEs can cause significant underutilization of the BRAM compute throughput.

Since our RIMA architecture is very deterministic (e.g. with no pipeline overlaps or multi-threading), it is possible to estimate a given workload's performance using a detailed analytical model for a given architecture configuration. Therefore, we implement a design space exploration tool that uses an analytical model to find the optimal combination of architecture parameters that minimizes the processing latency for each specific workload. The analytical model includes the following configurable architecture parameters: the fraction of elements in the output vector that are computed by the M-DPEs (f), the number of MACs performed serially in each bitline (m), the operands' bitwidth after applying bit slicing (b), and the number of MVU tiles (T).

The optimal configuration of these parameters minimizes the MVU latency of a given workload with dimensions $R \times C$ under specific resource constraints. The overall MVU latency is the higher of L-DPE and M-DPE latencies, and the resource constraint is that the number of BRAMs used by L-DPEs (for MRFs) and M-DPEs, and the number of DSPs used by the L-DPEs do not exceed 85% of the total number of available BRAMs and DSP blocks in the target FPGA device. This leaves enough BRAMs and DSPs to implement the rest of the fixed RIMA pipeline. Although it is hard to capture the logic block utilization in the analytical model, it was never the design's bottleneck for all the RIMA instances that we experimented with in our study.

VI. Evaluation Methodology

A. Platforms and Benchmarks

To highlight the gains of using our enhanced FPGA architecture with compute-capable BRAMs for DL acceleration, we first evaluate the performance of the RIMA architecture in comparison to state-of-the-art FPGA-based accelerators for RNNs. To ensure a fair comparison, we compare the BFP and INT8 versions of RIMA to the Microsoft Brainwave architecture from [10] which uses the same (1s.2m.5e) BFP format (BW-BFP) and the INT8 Intel NPU architecture from [19] (NPU-INT8), respectively. Both BW-BFP and NPU-INT8 use the same Stratix 10 GX 2800 FPGA device, and RIMA assumes a similar device (in resource count) in which the BRAMs are enhanced with in-memory compute capabilities. We also compare RIMA's performance to that of the same-generation Nvidia Titan V GV100 on the same set of workloads using the official Nvidia persistent CuDNN kernels. Although this GPU can perform half-precision (FP16) computations, the official kernels for these workloads only support single-precision (FP32). All our experiments use the RNN, GRU and LSTM models from the DeepBench [3] benchmark suite.

B. FPGA Implementation and Validation

We implement the MVU and MFU of the RIMA architecture in SystemVerilog RTL. The MVU is the main compute complex of the architecture and the only pipeline stage of the original NPU architecture that we re-architect to use in-BRAM compute capabilities. According to the results in [19], the other NPU pipeline stages (i.e. eVRF, and LD) utilize less than 10% of the FPGA's LUTs and BRAMs, and they are never the bottleneck for the NPU's operating frequency. For the compute-capable BRAMs, we write an RTL module that maps to a normal BRAM block since the interface of compute-capable BRAMs to the programmable routing remains unchanged. We use Intel Quartus Prime Pro 17.1 to synthesize, place and route the RIMA architecture with four tiles and necessary number of M-DPEs/L-DPEs to obtain the frequency and resource utilization results for each instance customized for each specific benchmark. Then, we add in the resources utilized by the rest of the RIMA architecture components as previously mentioned. We also validate the functional correctness of our RIMA architecture by performing RTL cycle-accurate behavioral simulations using ModelSim. To simulate the compute-capabilities of our enhanced BRAMs, we write a black-box simulation model for the BRAM that supports both the memory and compute modes of operation. The evaluation on the RIMA architecture accounts for the overheads of routing and on-chip data movement, demonstrating the realistic effects of the improved computing throughput on FPGA.

C. Performance Modeling

As existing FPGAs do not have the needed circuitry for in-BRAM compute, we rely on cycle-accurate RTL simulation with our behavioral simulation model of the compute-capable BRAM to obtain the cycle count for the RIMA MVU, which constitutes the majority of processing cycles. Then we combine that with an analytical model latency estimation for the deterministic MFU vector element-wise operations, assuming no overlap between the MVU and MFU processing. We obtain the maximum operating frequencies of the M-DPEs and the rest of the design from the Quartus timing reports. The BRAM in compute mode has a 1.6× lower frequency upper bound than an unmodified BRAM; in memory mode, the BRAM maximum frequency remains unchanged. The overall performance of the each RIMA instance is then calculated based on the obtained frequencies and number of processing cycles for each benchmark.

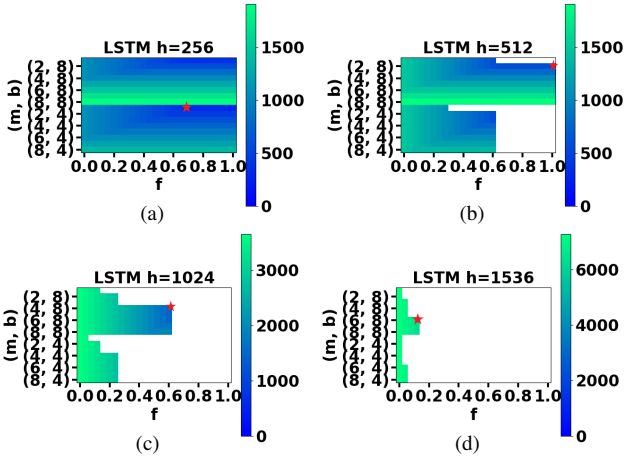


Fig. 8: Design space exploration for the RIMA architecture parameters with number of tiles and L-DPE lanes fixed at 4 and 40, respectively. Each pixel is one architecture configuration and the color represents the processing latency. White points are invalid configurations violating the resource constraints.

VII. Experimental Results

A. RIMA Architecture Design Space Exploration

The heatmaps in Fig. 8 show the latency results obtained by the analytical model during the design space exploration process for the RIMA-INT8 variation for four different LSTM benchmarks. In this experiment, we fix the number of MVU tiles and lanes of the L-DPEs to be 4 and 40 respectively, similar to that used in [19]. Each pixel in the heatmap represents one architecture configuration. The pixel color indicates the estimated latency for one time step in nanoseconds, a lighter color represents a higher latency and a darker color represents a lower latency (i.e. the darker the color is, the better). The white parts of the heatmaps indicate that the corresponding configuration violates the resource constraints of the analytical model. On the horizontal axis, the parameter f (fraction of compute mapped to M-DPEs) changes from 0% to 100%. The vertical axis represents 16 different combinations of the parameters (b, m), where b (bits per slice) takes the value 8 or 4, and m (MACs in serial on the same bit-line) is swept from 1 to 8.

With infinite resources, a larger f and smaller m would increase the degree of extracted parallelism and therefore lead to better performance. However, as shown in the figure, the larger the model size gets, the less fraction of compute can be mapped to the M-DPEs to ensure that the complete model still fits in the on-chip BRAMs. The optimal configurations for these models (highlighted by a red star on the heatmaps) are different, which highlights the importance of hardware customization. Our experiments show that the per-workload customization of the RIMA architecture offers an average 18% (up to 40%) performance improvement compared to a fixed RIMA instance for all workloads. For the largest LSTM model ($h = 1536$), the optimal architecture configuration has $m = 6$, $b = 8$, and $f = 0.15$. This model has 13.5 MB of weight data, which imposes tighter constraints on the fraction of computations that can be offloaded to the M-DPEs (only 15%). To offload more computation to the M-DPEs, six bit-serial MACs ($m = 6$) are performed sequentially on the same bitline, and no bit slicing is applied. On the other hand, for the smallest LSTM ($h = 256$), the optimal architecture configuration has $m = 1$, $b = 4$, and $f = 0.68$. This small model has a 512×1024 weight matrix, which enables us to map 68% of the compute to the M-DPEs, perform only 1 MAC operation per bitline (fully unrolled) and apply the bit slicing to extract a high degree of parallelism.

For RIMA-BFP, we found that for all the studied workloads, we were able to map all the computations to M-DPEs, which provides superior performance than any hybrid configurations. This highlights that in-BRAM compute on FPGAs achieves the best performance gains for lower precisions (3-bit operations in BFP8), which are becoming more widely adopted for DL inference.

TABLE II: FPGA implementation results of RIMA instances customized for each workload. The operating frequencies are for the M-DPE clock domain.

Benchmark	Precision	Logic	BRAMs	DSPs	Freq. (MHz)
RNN	**INT8**	87%	72%	50%	328
h=1152	**BFP**	59%	46%	44%	333
RNN	**INT8**	70%	65%	50%	417
h=1792	**BFP**	79%	64%	60%	250
LSTM	**INT8**	60%	55%	50%	455
h=256	**BFP**	49%	46%	40%	345
LSTM	**INT8**	74%	69%	50%	417
h=512	**BFP**	63%	42%	39%	323
LSTM	**INT8**	73%	69%	50%	313
h=1024	**BFP**	63%	42%	39%	323
LSTM	**INT8**	89%	93%	50%	278
h=1536	**BFP**	84%	57%	60%	303
GRU	**INT8**	74%	69%	50%	417
h=512	**BFP**	82%	57%	60%	303
GRU	**INT8**	70%	65%	50%	417
h=1024	**BFP**	82%	57%	60%	303
GRU	**INT8**	85%	89%	50%	296
h=1536	**BFP**	82%	57%	60%	303

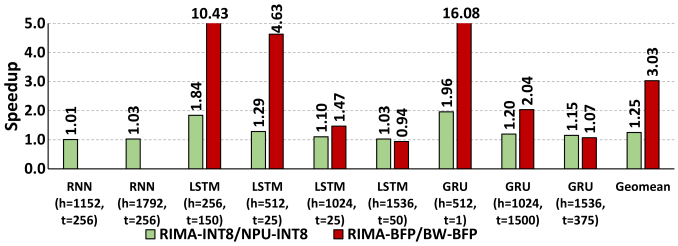


Fig. 9: Speedup achieved by RIMA INT8 and BFP compared to NPU-INT8 and BW-BFP, respectively.

B. RIMA FPGA Implementation Results

The FPGA resource utilization results of the per-workload customized RIMA instances are shown in Table II. The results show that the implemented RIMA instances achieve a high degree of the FPGA resource utilization in all cases. Models with a larger weight size generally have a higher BRAM usage, as more weights need to be pinned on chip. The M-DPE clock frequencies for different workloads also vary due to the effect of architecture parameter customization on the external reduction units and accumulators. In RIMA-BFP, the whole architecture operates at the same frequency reported in Table II, while in RIMA-INT8, the M-DPEs operate at a higher frequency (shown in the table) than the 278 MHz clock driving the rest of the architecture.

C. Performance Results

Fig. 9 presents the speedup achieved by RIMA-INT8 compared to the NPU-INT8, and RIMA-BFP compared to the BW-BFP. RIMA-INT8 and RIMA-BFP achieve average speedups of **1.25× and 3× over the baseline NPU-INT8 and BW-BFP** that do not use compute-capable BRAMs, respectively. As previously discussed, the gains of adding BRAM compute-capabilities are higher when used to accelerate narrower precisions. This can be attributed to two main reasons: (1) bit-serial multiplication latency grows quadratically with operands bitwidth as shown in Table I, and (2) narrower bitwidths allow sharing the same memory array bitline among different weight values that are all multiplied by the same input. Therefore, the RIMA architecture has the potential to accelerate other low-bit-width formats such as INT4; the detailed analysis for other precisions is left for future work. The figure also

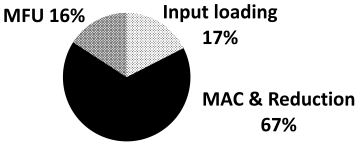


Fig. 10: Latency breakdown of RIMA-INT8 for the LSTM ($h = 1024$) benchmark.

TABLE III: Processing latency comparison between RIMA and Nvidia Titan V GV100 GPU. All numbers are in milliseconds.

Workload	RNN		LSTM			GRU	
h	1152	1792	256	512	1536	1024	1536
t	256	256	150	25	50	1500	375
GPU (FP32)	1.0	1.38	0.44	0.15	5.7	12.5	29.94
RIMA (INT8)	0.21	0.42	0.06	0.02	0.24	2.63	1.26
RIMA (BFP8)	0.19	0.39	0.04	0.02	0.15	1.86	0.89

shows higher speedups for smaller workloads that impose more relaxed constraints on the amount of workload computation that can be mapped to M-DPEs. The smaller the amount of weights that need to be kept persistent in the on-chip memories, the higher the degree of compute parallelism that can be extracted from the compute-capable BRAMs.

For some of the workloads in Fig. 9, the speedup achieved by RIMA exceeds the peak performance gains of compute-capable BRAMs (1.6× and 2.3× in Section III). This is because the baseline NPU-INT8 and BW-BFP architectures suffer from significant underutilization and padding overheads due to their fixed overlay architecture for all workloads. In contrast, RIMA best exploits the FPGA reconfigurability by customizing the architecture parameters to achieve the optimal performance for each given workload.

As an example, Fig. 10 shows the breakdown of the processing latency of RIMA-INT8 for the LSTM benchmark with 1024 hidden units ($h = 1024$). It shows that the matrix-vector multiplication in M-DPEs constitutes 84% of the processing time, where the majority of this time is spent in bit-serial MAC and reduction operations. The latency of the external reduction unit is fully overlapped with the in-BRAM computation. Loading the inputs to the compute-capable BRAMs takes 17% of the cycles, while the remaining time is spent on the vector operations in the MFUs.

Table III compares the processing latencies of different workloads running on the RIMA architecture and the Nvidia Titan GV100 GPU. The results show that the RIMA architecture **outperforms the GPU by 8.1× and 10.6×** when using the INT8 and BFP numerical precisions, respectively.

VIII. Related Work

FPGA Architecture Changes for DL: Prior work on FPGA architecture changes for more efficient DL acceleration have focused many on DSPs and logic blocks. Boutros et al. [4] and Rasoulinezhad et al. [21] re-architect the DSP blocks to support higher density of low precision MACs for DL inference. Arora et al. [1] proposes integrating in-fabric tensor units for high-efficient matrix-matrix multiplications used in many DL models. Boutros et al. [5] and Eldafrawy et al. [9] propose several ideas for logic block changes that can substantially increase the arithmetic density of the soft fabric. To the best of our knowledge, this work is the first attempt to enhance the FPGA's on-chip memories with massively parallel bit-serial compute capabilities.

FPGA DL Accelerators: Several works have proposed interesting techniques to accelerate DL inference on FPGAs. Han et al. [13], Samragh et al. [22], and Wang et al. [28] use weight pruning and compression techniques to reduce the model memory footprint for FPGA deployment. Li et al. [18] proposes a design methodology to determine the optimal parameters for compression techniques. Moreover, there are several works that propose methods to efficiently map the DL computations into available compute resources on FPGAs such as [23], [12], and [26]. Our work is orthogonal to all above works, we present a reconfigurable accelerator architecture that can flexibly balance between in-BRAM and DSP compute to achieve the highest performance. Both [10] and [19] introduce the NPU overlay, which we re-architect to showcase the performance of our proposed compute-capable BRAMs in accelerating DL workloads. Our proposal to enhance FPGA BRAMs with in-memory compute capabilities is a more fundamental change in the device architecture which can potentially benefit any accelerator architecture.

In-Memory DL Accelerators: Many interesting prior work have performed DNN acceleration with in-memory computing. DRISA [17] is a DRAM-based in-memory accelerator with Boolean logic operating on data from all bit-lines. Neural Cache [8] re-purposes a CPU's last level cache as bit-serial computing units, and is compared to our work in evaluation. FloatPIM [14] performs high precision floating point operation directly on digital signals. In contrast to the above works which target a fixed accelerator ASIC or fixed structure of CPU caches, our proposal is to leverage in-memory computing in FPGAs which can be optimally reconfigured for both different applications and different DL models. Zha et al. [31] proposes a ReRAM based reconfigurable fabric where each tile can be configured for compute, memory or interconnect mode. While the above work [31] builds a new architecture based on ReRAM for reconfigurable in-memory computing, our work leverages the existing BRAMs on FPGA for computation with minimal architecture modifications and area cost while maintaining the reconfigurability of FPGAs.

IX. Conclusion

The continuous increase in the capacity of FPGA on-chip memories offers a great opportunity to also enhance the device's compute throughput by supporting in-BRAM compute capabilities. This work is the first attempt to evaluate the gains and costs on in-memory compute on FPGAs. Our proposed architectural change can enhance the peak MAC throughput of a large Stratix 10 device by a factor of 1.6× and 2.3× for 8-bit integer and block floating-point precisions, respectively. This comes at a cost of 7.4% increase in the BRAM tile area, which corresponds to only 1.8% increase in the total FPGA die size. In addition, our proposed compute-capable BRAM does not change the existing BRAM modes or interface to the programmable routing, and requires minimal CAD support which further simplifies its adoption in commercial architectures. We also evaluate the effect of enhancing FPGAs with compute-capable BRAMs on deep learning inference performance. To do that, we implement a reconfigurable in-memory accelerator architecture, RIMA, which uses compute-capable BRAMs and exploits the FPGA reconfigurability to perform per-workload architecture customization. The RIMA architecture outperforms the state-of-the-art Brainwave overlay by 1.25× and 3× for 8-bit integer and block floating-point precisions respectively on a variety of real-time memory-bound RNN workloads. It also achieves an order of magnitude higher performance compared to same-generation GPUs. While in this work we focus on RNN workloads, compute-capable BRAMs may accelerate other compute intensive applications with the abundant bit-serial computing units created. This study shows promising results for incorporating compute-capable BRAMs in modern FPGA architectures, especially for narrower data precisions which are becoming widely adopted in deep learning inference tasks.

Acknowledgement

We thank the anonymous reviewers for their helpful feedback. This work was supported in part by the NSF CCF-1908601 award, the NSF CAREER-1652294 award, the Intel gift award, and the Applications Driving Architectures (ADA) Research Center, a JUMP Center co-sponsored by SRC and DARPA.

REFERENCES

[1] A. Arora *et al.*, "Hamamu: Specializing FPGAs for ML Applications by Adding Hard Matrix Multiplier Blocks," in *IEEE International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, 2020.

[2] U. Aydonat *et al.*, "An OpenCL Deep Learning Accelerator on Arria 10," in *ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA)*, 2017.

[3] Baidu Research, "Baidu DeepBench," https://github.com/baidu-research/DeepBench, accessed: 2021-01-18.

[4] A. Boutros *et al.*, "Embracing Diversity: Enhanced DSP Blocks for Low-Precision Deep Learning on FPGAs," in *IEEE International Conference on Field Programmable Logic and Applications (FPL)*, 2018.

[5] ——, "Math Doesn't Have to be Hard: Logic Block Architectures to Enhance Low-Precision Multiply-Accumulate on FPGAs," in *ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA)*, 2019.

[6] ——, "Beyond Peak Performance: Comparing the Real Performance of AI-Optimized FPGAs and GPUs," in *IEEE International Conference on Field Programmable Technology (FPT)*, 2020.

[7] M. Drumond *et al.*, "Training DNNs with Hybrid Block Floating Point," *Advances in Neural Information Processing Systems (NeurIPS)*, vol. 31, pp. 453–463, 2018.

[8] C. Eckert *et al.*, "Neural cache: Bit-Serial In-Cache Acceleration of Deep Neural Networks," in *ACM/IEEE International Symposium on Computer Architecture (ISCA)*, 2018.

[9] M. Eldafrawy *et al.*, "FPGA Logic Block Architectures for Efficient Deep Learning Inference," *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, vol. 13, no. 3, pp. 1–34, 2020.

[10] J. Fowers *et al.*, "A Configurable Cloud-Scale DNN Processor for Real-Time AI," in *ACM/IEEE International Symposium on Computer Architecture (ISCA)*, 2018.

[11] D. Fujiki *et al.*, "Duality Cache for Data Parallel Acceleration," in *ACM/IEEE International Symposium on Computer Architecture (ISCA)*, 2019.

[12] Y. Guan *et al.*, "FP-DNN: An Automated Framework for Mapping Deep Neural Networks onto FPGAs with RTL-HLS Hybrid Templates," in *IEEE International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, 2017.

[13] S. Han *et al.*, "ESE: Efficient Speech Recognition Engine with Sparse LSTM on FPGA," in *ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA)*, 2017.

[14] M. Imani *et al.*, "FloatPIM: In-Memory Acceleration of Deep Neural Network Training with High Precision," in *ACM/IEEE International Symposium on Computer Architecture (ISCA)*, 2019.

[15] *Intel Stratix 10 GX/SX Device Overview*, Intel Corporation, 2019.

[16] *Intel Quartus Prime Pro Edition User Guide: Design Recommendations*, Intel Corporation, 2020.

[17] S. Li *et al.*, "DRISA: A DRAM-based Reconfigurable In-Situ Accelerator," in *IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2017.

[18] Z. Li *et al.*, "E-RNN: Design Optimization for Efficient Recurrent Neural Networks in FPGAs," in *IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2019.

[19] E. Nurvitadhi *et al.*, "Why Compete When You Can Work Together: FPGA-ASIC Integration for Persistent RNNs," in *IEEE International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, 2019.

[20] A. Putnam *et al.*, "A Reconfigurable Fabric for Accelerating Large-Scale Datacenter Services," in *ACM/IEEE International Symposium on Computer Architecture (ISCA)*, 2014.

[21] S. Rasoulinezhad *et al.*, "PIR-DSP: An FPGA DSP Block Architecture for Multi-precision Deep Neural Networks," in *IEEE International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, 2019.

[22] M. Samragh *et al.*, "Customizing Neural Networks for Efficient FPGA Implementation," in *IEEE International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, 2017.

[23] Y. Shen *et al.*, "Maximizing CNN Accelerator Efficiency Through Resource Partitioning," in *ACM/IEEE International Symposium on Computer Architecture (ISCA)*, 2017.

[24] K. Tatsumura *et al.*, "High Density, Low Energy, Magnetic Tunnel Junction Based Block RAMs for Memory-Rich FPGAs," in *IEEE International Conference on Field-Programmable Technology (FPT)*, 2016.

[25] ——, "Enhancing FPGAs with Magnetic Tunnel Junction-Based Block RAMs," *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, vol. 11, no. 1, pp. 1–22, 2018.

[26] S. Venieris and C.-S. Bouganis, "fpgaConvNet: A Framework for Mapping Convolutional Neural Networks on FPGAs," in *IEEE International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, 2016.

[27] J. Wang *et al.*, "14.2 A Compute SRAM with Bit-Serial Integer/Floating-Point Operations for Programmable In-Memory Vector Acceleration," in *IEEE International Solid-State Circuits Conference (ISSCC)*, 2019.

[28] S. Wang *et al.*, "C-LSTM: Enabling Efficient LSTM Using Structured Compression Techniques on FPGAs," in *ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA)*, 2018.

[29] S. Yazdanshenas and V. Betz, "Automatic Circuit Design and Modelling for Heterogeneous FPGAs," in *IEEE International Conference on Field Programmable Technology (FPT)*, 2017.

[30] S. Yazdanshenas *et al.*, "Don't Forget the Memory: Automatic Block RAM Modelling, Optimization, and Architecture Exploration," in *ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA)*, 2017.

[31] Y. Zha and J. Li, "Liquid Silicon-Monona: A Reconfigurable Memory-Oriented Computing Fabric with Scalable Multi-Context Support," in *ACM SIGPLAN Notices*, vol. 53, no. 2, 2018, pp. 214–228.