A 28-nm Compute SRAM With Bit-Serial Logic/Arithmetic Operations for Programmable In-Memory Vector Computing

Jingcheng Wang, *Student Member, IEEE*, Xiaowei Wang, *Student Member, IEEE*, Charles Eckert, *Student Member, IEEE*, Arun Subramaniyan, *Student Member, IEEE*, Reetuparna Das, *Member, IEEE*, David Blaauw, *Fellow, IEEE*, and Dennis Sylvester, *Fellow, IEEE*

***Abstract*— This article proposes a general-purpose hybrid in-/near-memory compute SRAM (CRAM) that combines an 8T transposable bit cell with vector-based, bit-serial in-memory arithmetic to accommodate a wide range of bit-widths, from single to 32 or 64 bits, as well as a complete set of operation types, including integer and floating-point addition, multiplication, and division. This approach provides the flexibility and programmability necessary for evolving software algorithms ranging from neural networks to graph and signal processing. The proposed design was implemented in a small Internet of Things (IoT) processor in the 28-nm CMOS consisting of a Cortex-M0 CPU and 8 CRAM banks of 16 kB each (128 kB total). The system achieves 475-MHz operation at 1.1 V and, with all CRAMs active, produces 30 GOPS or 1.4 GFLOPS on 32-bit operands. It achieves an energy efficiency of 0.56 TOPS/W for 8-bit multiplication and 5.27 TOPS/W for 8-bit addition at 0.6 V and 114 MHz.**

***Index Terms*— 8T transposable bit cell, bit-serial arithmetic, flexible bit-width, in-memory computing (IMC), near-memory computing, memory, SRAM, single instruction multiple data (SIMD) architecture.**

I. Introduction

IN THE conventional von Neumann architecture, a clear gap lies between data storage and processing: memories store data, while processors compute on data. Owing to Moore's law, in the past few decades, the computing power of the integrated circuits has rapidly scaled as logic gates became faster and the number of processing cores increased steadily until we hit the "Memory Wall" [1]. The on-chip global interconnects' latency and energy cannot keep up with the scaling of logic gates. Thus, the computation throughput and energy have become dominated by the memory bandwidth and data movement energy. As shown in Fig. 1(a), the bandwidth at the I/Os of all SRAM banks inside a big memory macro such as a 20-MB L3 cache, which is over a hundred TB per second [2], [3], and is comparable to the theoretical maximum computation bandwidth of the state-of-the-art systolic processing array [4]. Hence, the bottleneck is the local data network inside the memory macro and the global data bus on chip. Furthermore, a large fraction of energy consumption today is spent on moving data back and forth between memory and compute units [5]. As shown in Fig. 1(b), it only takes sub-pico joules of energy to do a 32-bit addition while tens of pico joules are spent on retrieving data from far away memory banks.

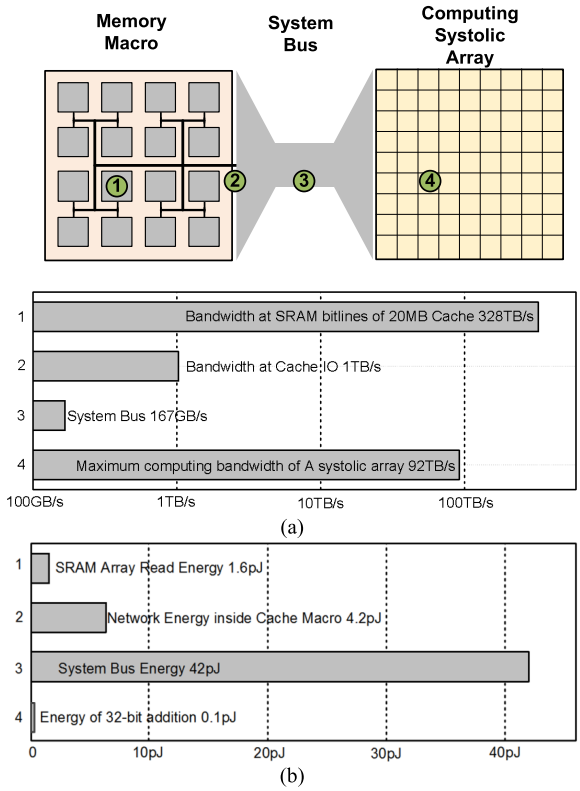


Fig. 1. Bottlenecks in the conventional von Neumann architecture. (a) Low on-chip network bandwidth. (b) High data movement energy.

Previously, people tried to overcome the "Memory Wall" by introducing more memory hierarchies, in an effort to bring the data closer to the computation. However, the memory problem

Manuscript received May 3, 2019; revised July 26, 2019 and August 30, 2019; accepted September 2, 2019. Date of publication September 23, 2019; date of current version December 27, 2019. This article was approved by Associate Editor Mingoo Seok. This work was supported by the Applications Driving Architectures (ADA) Joint University Microelectronics Program (JUMP) Center, a Semiconductor Research Corporation (SRC) Program sponsored by the Defense Advanced Research Projects Agency (DARPA). *(Corresponding author: Jingcheng Wang.)*

The authors are with the Department of Electrical Engineering and Computer Science, University of Michigan, Ann Arbor, MI 48109 USA (e-mail: jiwang@umich.edu).

Color versions of one or more of the figures in this article are available online at http://ieeexplore.ieee.org.

Digital Object Identifier 10.1109/JSSC.2019.2939682

is further exacerbated by the advent of data-intensive applications such as neural networks [6], [7], computer vision [8], and stream processing [9]. The shift from computation-centric to data-centric architecture has led to extensive research focused on in-/near-memory computing, which moves computation to where the data are located. Recently, we have seen many studies that try to bring computation to different levels of memory hierarchies, including DRAM [10] and non-volatile memories like STT-MRAM [11], ReRAM[12], and Flash [13]. This article focuses on designing computational SRAM banks. Most SRAM in today's chips is located in the caches of CPUs or GPUs. These large CPU and GPU SRAMs present an opportunity for extensive IMC and have, to date, remained largely untapped.

Many types of analog IMC architectures have been proposed. For example, some perform computation in the current domain. In this case, one operand is pre-stored in the SRAM array, while the other operand can be modulated into the word-line voltage level [14] or pulsewidth of the word-line enable signal [15], [16]. The multiplication result of the two operands is then represented by the various discharge currents of the bit cells. Often multiple word-lines are activated simultaneously, and the multiplication results are accumulated on the bitline as the total bitline discharge current is the sum of each individual bit-cell current. The final multiply–accumulate (MAC) result is naturally represented by the analog bitline voltage sensed by an analog-to-digital converter (ADC). Other in-memory approaches have proposed the use of time-domain computation [17], where the operands are modulated into the reference voltages to the voltage-controlled oscillator (VCO) and the MAC results are represented by various pulse widths sensed by time-to-digital converters (TDCs). These analog IMC approaches can usually achieve very high energy efficiency and throughput, but suffer from other problems. First, they usually require expensive analog-to-digital and digital-to-analog conversions at the array boundary. Second, the computation accuracy is highly susceptible to noise and process voltage temperature (PVT) variations, which limits the functionality to low precision addition or multiplication and algorithm to binary-weight networks (BWNs). Recently, charge-domain computing has been proposed to substantially improve the robustness and bit-precision scalability because modern very large scale integration (VLSI) processes have better control over the capacitances than the transistor parameters [18], [19]. However, the signal-to-quantization-noise ratio (SQNR) still limits precision in these approaches to <8 bits.

Although traditional computing architectures such as CPU and GPU show limitations in energy efficiency and memory bandwidth, their appeal lies in their general functionality and programmability. They can perform a wide range of operations from bit-wise logic operation to integer/floating-point arithmetic. Not only are these computations accurate and robust because the designs are fully digital, but also they are highly flexible and can implement many algorithms and neural network types and sizes. In this respect, most current in-memory approaches suffer from the same major limitation: they accelerate only one type of algorithm and are inherently restricted to a very specific application domain due to their limited bit-width precision and non-programmable architecture. On the other hand, software algorithms continue to evolve rapidly, especially in novel application domains such as neural networks, vision, and graph processing, which make rigid accelerators of limited use. This has led to at least one recent work to improve the programmability of IMC with instruction set architecture (ISA) and compiler design [20].

To address these limitations, we present a general-purpose hybrid in-/near-memory compute SRAM (CRAM) [21] that combines the efficiency of in-memory computation with the flexibility and programmability necessary for evolving software algorithms. It does part of the logic operations in SRAM bit-lines and most arithmetic operations in pitch-matched, near-memory peripherals at the end of each bitline. It can accommodate a wide range of bit-widths, from single to 32 or 64 bits, and operation types, including integer and floating-point addition, multiplication, and division, with a small amount of hardware overhead. Its high-throughput digital-domain computation is accurate and robust, and the design offers good energy efficiency. The CRAM tries to repurpose the large existing on-chip memory storage by augmenting a conventional SRAM bank in a cache with vector-based, bit-serial in-memory/near-memory arithmetic.

The remainder of this article is organized as follows. Section II generally introduces the bit-serial operation and the architecture of the proposed computational SRAM. Section III describes the 8T transposable bit cell and the computing peripheral in detail. Section IV presents the algorithm of multi-bit arithmetic operations. Section V discusses the measurement results of the proposed design, and finally, the conclusions are presented in Section VI.

II. Overview of Bit-Serial Arithmetic and CRAM Architecture

A. Bit-Serial Arithmetic

Several previous digital IMC works [22]–[24] supported some simple bit-parallel operations such as bit-wise logic and copy. However, these are carry-less operations that do not require interaction between bit-lines. To make IMC as general purpose as the ALU in a CPU, support is needed for more complex arithmetic operations such as addition, multiplication, and even floating-point operation. The main challenge in supporting these complex arithmetic operations is facilitating carry propagation between bit-lines. We propose a bit-serial implementation to address this challenge.

Since the 1980s, bit-serial computing architectures have been widely used for digital signal processing because it can usually provide the most area-efficient design in the presence of a massive bit-level parallelism [25], [26]. The key idea is to process 1 bit of multiple data elements every cycle. This model is particularly useful in scenarios where the same operation is applied to the same bit of all data elements in a vector, like in the single instruction multiple data (SIMD) architectures. For example, to compute the element-wise sum of two arrays with 512 32-bit elements, a conventional processor would take at least 512 cycles to get the operands element-by-element

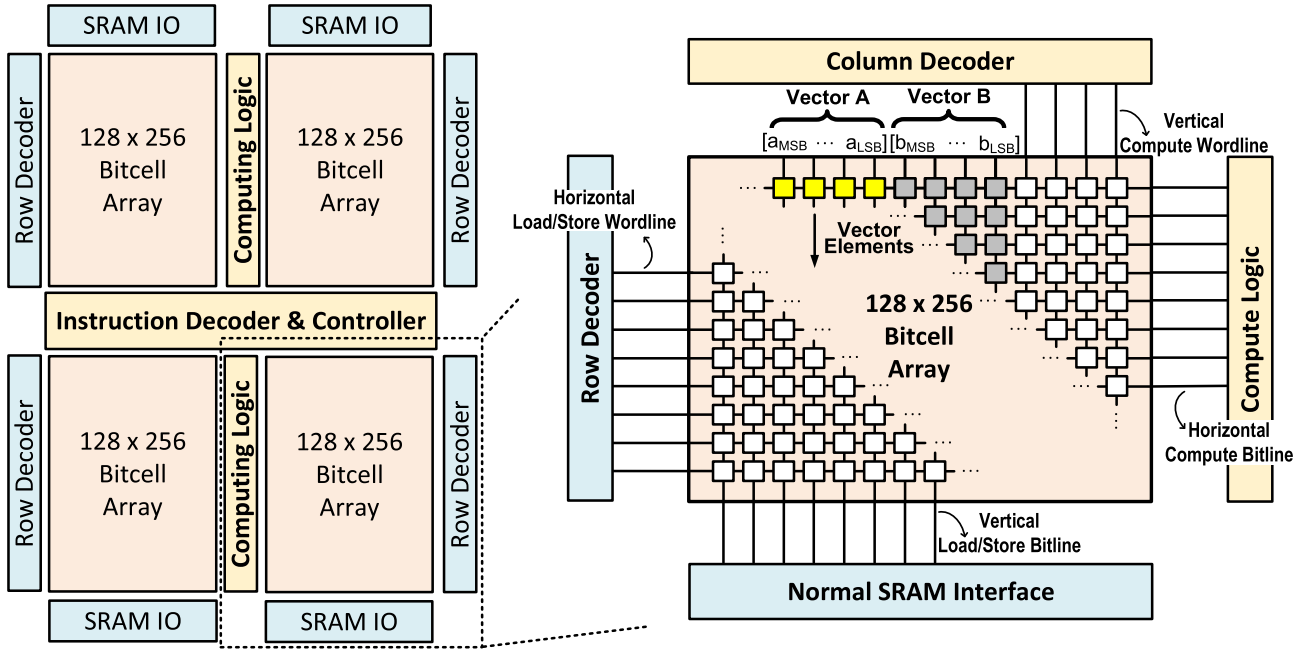


Fig. 2. Proposed CRAM architecture.

from the SRAM and then perform the operation. A bit-serial processor, on the other hand, would complete the operation in 32 steps as it processes the arrays bit-slice by bit-slice instead of element-by-element. Besides, bit-serial operation allows for flexible operand bit-width, which can especially be advantageous in DNN hardware designs where the required bit width can vary from layer to layer [27], [28].

Although some bit-parallel approaches [29] can perform addition/subtraction with the same throughput and energy efficiency as the bit-serial approach, they cannot support more complex arithmetic operations such as multiplication. However, the near-memory components in the CRAM are akin to a small reduced instruction set computer (RISC) machine. With a well-designed instruction set, the CRAM can support many complex arithmetic operations using only software. Therefore, a bit-serial approach provides the CRAM the advantages of greater programmability and versatility.

B. CRAM Architecture

Fig. 2 shows the overall architecture of one 16-KB CRAM bank. Each CRAM bank consists of four 128×256 arrays that load or store data conventionally using horizontal word-lines and vertical bit-lines. The normal SRAM peripherals, such as a row decoder, column mux, and sense amp, are shown in blue. In this diagram, the array has been preloaded with two vectors of data, vectors A and B. The data elements from the same vector are placed into different rows, while the various bits of the data elements are spread into different columns from the most significant bits (MSBs) to the least significant bits (LSBs). The corresponding elements from the two vectors that are going to be operated must be aligned on the same word-line. To perform bit-serial operation, we need to activate the same bit position from two vectors. Therefore, the column decoder and pitch-matched compute logic are added so that IMC can

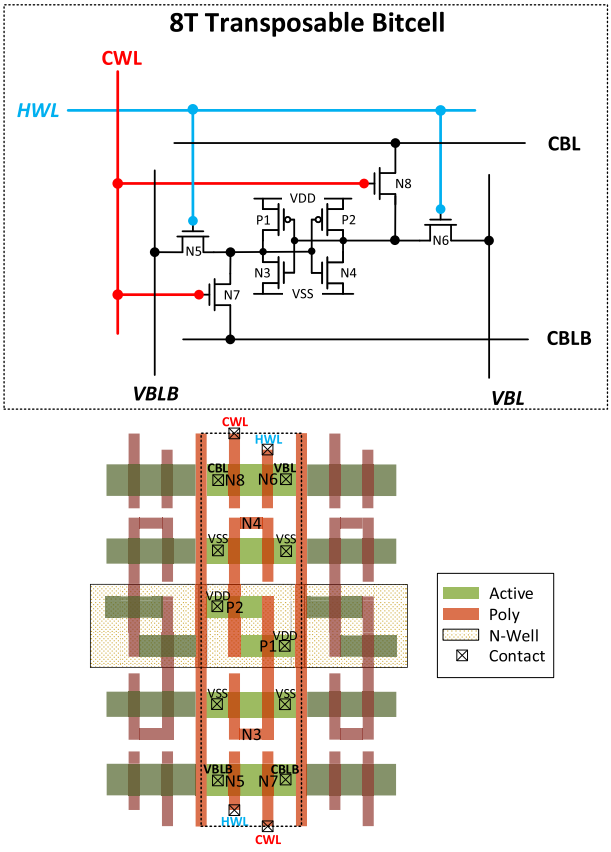


Fig. 3. Schematic and layout of 8T transposable bit cell.

be performed using vertical compute word-lines (CWLs) and horizontal compute bit-lines (CBLs). For example, in the first cycle, we simultaneously activate the vertical word-lines of the LSB from the two vectors. Then, the computation is performed in both horizontal bit-lines and the compute logics at the end of the bit-lines. Near the end of the cycle, the result is then stored

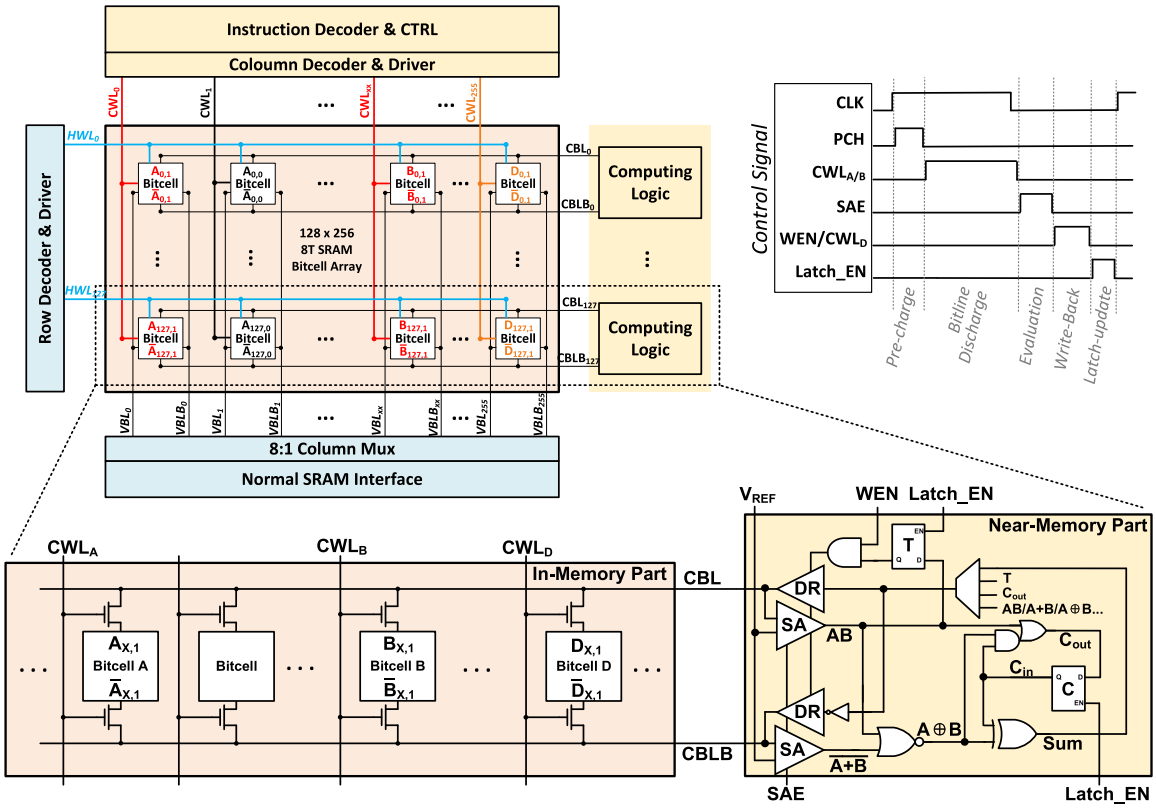


Fig. 4. CRAM array architecture (top-left), computation control signal timing diagram (top-right), and in-/near-memory computing peripherals (bottom).

back in the array at some destination bit location selected by a third vertical word-line. In the next cycle, other bits of each operand are activated to continue the computation. Again, the result is stored back at the designated position at the end of the cycle. By repeating single-bit operations cycle-by-cycle, we can perform any complex multi-bit arithmetic with carry-propagation. For example, a 32-bit adder will take 32 cycles to finish. Note that although bit-serial computation is expected to have high latency per operation, it gains significantly in terms of throughput. A 16-KB SRAM bank contains 256 horizontal CBLs in total, and a 35-MB last level cache (LLC) in the Haswell server processor can accommodate 2240 such 16-KB banks [2], which means a total of 573 440 bit-lines can do computations in parallel. In this case, the maximum throughput would be equivalent to 17 920 32-bit adders or 71 680 8-bit adders.

The computing logic is shared between the arrays on the left and right and takes 4.5% of the CRAM bank area. The instruction decoder and controller in the middle of the bank, shared by all four arrays, take 32-bit instruction and generate control signals for the computing logic. They occupy 5.2% of the bank area. The details of the controller instructions will be presented in Section III.

III. CRAM Circuitry

A. 8T Transposable Bit Cell

Many previous IMC works [16], [24] choose to store each word unconventionally by spreading bits into different rows of the same vertical bitline. This approach makes the computation much easier and can directly use 6T bit cell for minimizing area. However, the conventional SRAM read/write operations become much more complicated and incompatible with modern computer architectures where bits of a word are spread into bit cells on the same row. To maintain compatibility with the mainstream CPU/GPU architecture, the CRAM writes/reads operands conventionally with horizontal word-lines and vertical bit-lines, which is made possible by the 8T transposable bit cell. Fig. 3 shows the schematic and the layout of the bit cell [30]. Four of the transistors form the cross-coupled inverter pair to hold the data, and there are two pairs of access transistors for read/write. The structure is similar to the conventional 8T dual-port SRAM bit cell except that it provides bidirectional access: the bit cell can be read or written from either vertical bitline or horizontal bitline. Therefore, the CRAM can operate directly on the stored operands in memory by enabling the same bit position from two vector elements with vertical word-lines and perform the computation on horizontal bit-lines. Furthermore, it can also directly read a complete word by enabling the horizontal word-line and sense the result from vertical bit-lines. With the logic rule transistor in the 28-nm CMOS, the bit cell area is 0.782 μm^2 (0.405 μm $\times$ 1.93 μm), which is 638F^2 when normalized to technology node feature size (F). If we are allowed to use a push-rule transistor, as is commonly done in the foundry bit cell, the transposable 8T bit cell area would be 0.543 μm^2, which is 40% larger than a foundry-provided dual-port 8T cell (0.389 μm^2) and 3.5$\times$ larger than a standard 6T cell (0.157 μm^2). The extra area overhead is mainly due to the non-shared source/drain and poly contacts between the adjacent cells.

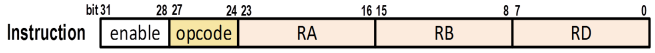
If the area density is of primary concern, such as in a very large (multiple MB) cache, we have proposed another solution [31] that uses a standard push-rule 6T cell with the data words stored and computed in the vertical data format. The bit-serial peripherals remain the same. This so-called neural cache design uses only a few transpose memory units (TMUs), which are built out of arrays of 8T transposable bit cells, at the gateway of the cache to serve as a translator between the conventional bit-parallel word layout and the transpose vertical layout.

B. Computing Peripherals

Fig. 4 shows a detailed view of one row in the bit cell array. The logic operations are performed on the bitline (in-memory), while small additional in-row logic (near-memory) enables carry-propagation between successive bit-serial calculations. An example of 1-bit addition will be used to illustrate the CRAM single-cycle operation and computing peripherals. $A_{x,y}$ stands for the yth bit of the xth elements of the vector A. Here, we add the second bit from vector A ($A_{X,1}$) and vector B ($B_{X,1}$) with carry-in (C_{in}) from the previous cycle and store the sum back to the second bit of vector D ($D_{X,1}$), and latch the carry-out (C_{out}) for the next cycle. First, the CRAM instruction decoder receives the ADD instruction with the three column addresses for bits $A_{X,1}$, $B_{X,1}$, and $D_{X,1}$. After pre-charging (PCH) the CBL and CBL bar (CBLB), we activate the vertical CWLs of $A_{X,1}$ and $B_{X,1}$ simultaneously to generate "A AND B" on CBL and "() AND ()" on CBLB. To prevent potential read disturbance issues caused by reading simultaneously from two bit cells, we have a separate supply voltage rail for the driver of $CWL_{A/B}$, so that we can lower the word-line voltage when necessary. In addition, the pseudo-differential sense amplifiers are used at the end of CBL and CBLB, allowing for early sensing of results at a much smaller bitline voltage swing. This is the in-memory part of the computation. Next, after the dual sense amps are enabled, the in-memory logic operation results propagate into the near-memory region located at the end of each CBL. The NOR gate generates "A XOR B," which combined with C_{in} from the C latch produces sum and C_{out}. Then CWL_D is activated, and the sum is written back to destination bit D_1. Finally, near the end of the cycle, C_{out} updates the C latch, which provides C_{in} for the next cycle.

When we activate the CWL, all 256 CBLs in the 16-KB CRAM banks are performing the same single-bit instruction in an SIMD fashion. To support complex multibit arithmetic, the CRAM has to be able to execute instructions only on certain selected CBLs; and therefore, we add the Tag (T) latch to enable conditional operation. The Tag latch is used as the enable signal of the write-back driver. Therefore, for the CBL whose Tag latch stores 0, the computation result will not be written back to the memory, as if the instruction is not executed at all. The content of the Tag latch can be loaded from or written into the memory array. In addition to the logics introduced before, we also add a multiplexer to allow for the write-back of the signals besides the sum, such as A AND B, A OR B, C_{out}, or Tag.

TABLE I
CRAM INSTRUCTION SET



Single-Cycle Primitives					
Type	**Opcode**	**RA**	**RB**	**RD**	**Comments**
Logic	AND/OR/XOR/ NAND/NOR/XNOR	✓	✓	✓	Perform logic operation on RA and RB, and store the result back to RD
Arithmetic	ADD	✓	✓	✓	Add RA and RB, write back to RD
Shift	Copy	✓		✓	Copy RA to RD
	INV	✓		✓	INV RA and write back to RD
Comparison	Equal	✓			Write "RA == AddrRB[0]" to Tag latch
Utility	LOAD T	✓			Load RA to Tag latch
	STORE C/T			✓	Write Carry/Tag latch back to RD
	Set C				Set Carry Latch to 1
	Reset C				Reset Carry Latch to 0
	C to T				Write Carry Latch to Tag Latch

TABLE II
SAMPLE OF SUPPORTED OPERATIONS AND CYCLE COUNTS

Sample Multi-Cycle Operations		
Type	Operation	**# Cycles**
Logic	AND	N
	NOR	N
	XOR	N
	NAND	N
	OR	N
	XNOR	N
Integer	Add	N+1
	Sub	2N+1
	Mult	$N^2 + 5N - 2$
	UDiv	$1.5N^2 + 5.5N$
32-bit Float Point	Add/Sub	4978
	Mult	679
	Div	697
Comparison	Equal	2N+1
	Greater/Less	2N+1
	Search	N
N is the bit-width of data		

With the computing peripherals as shown in Fig. 4, the CRAM controller can support up to 16 single-cycle instructions, given in Table I. Besides the logic and add operation, it includes copy, inversion, load/store of carry or tag, comparison, and set/reset carry. The CRAM controller takes 32-bit instruction. Four bits ([31:28]) are used for various enable signals for different features. Four bits ([27:24]) are used for the opcode for the 16 instructions. Eight bits are used for the address because every memory array contains 256 CWLs. Bits [23:16], [15:8], and [7:0] represent the bit address of operand A (RA), operand B (RB), and the destination location D (RD), respectively. Using these single-cycle micro instructions, we can build complex multi-cycle macro instructions, including search, multiplication, division, and floating-point arithmetic.

IV. MULTI-CYCLE ARITHMETIC

The users can program the CRAM to achieve many complex computations. Table II shows a sample list of the supported multi-cycle operations and the number of single-cycle instructions each takes. Next, we will introduce some commonly used arithmetic operations and the way to program them in the CRAM.

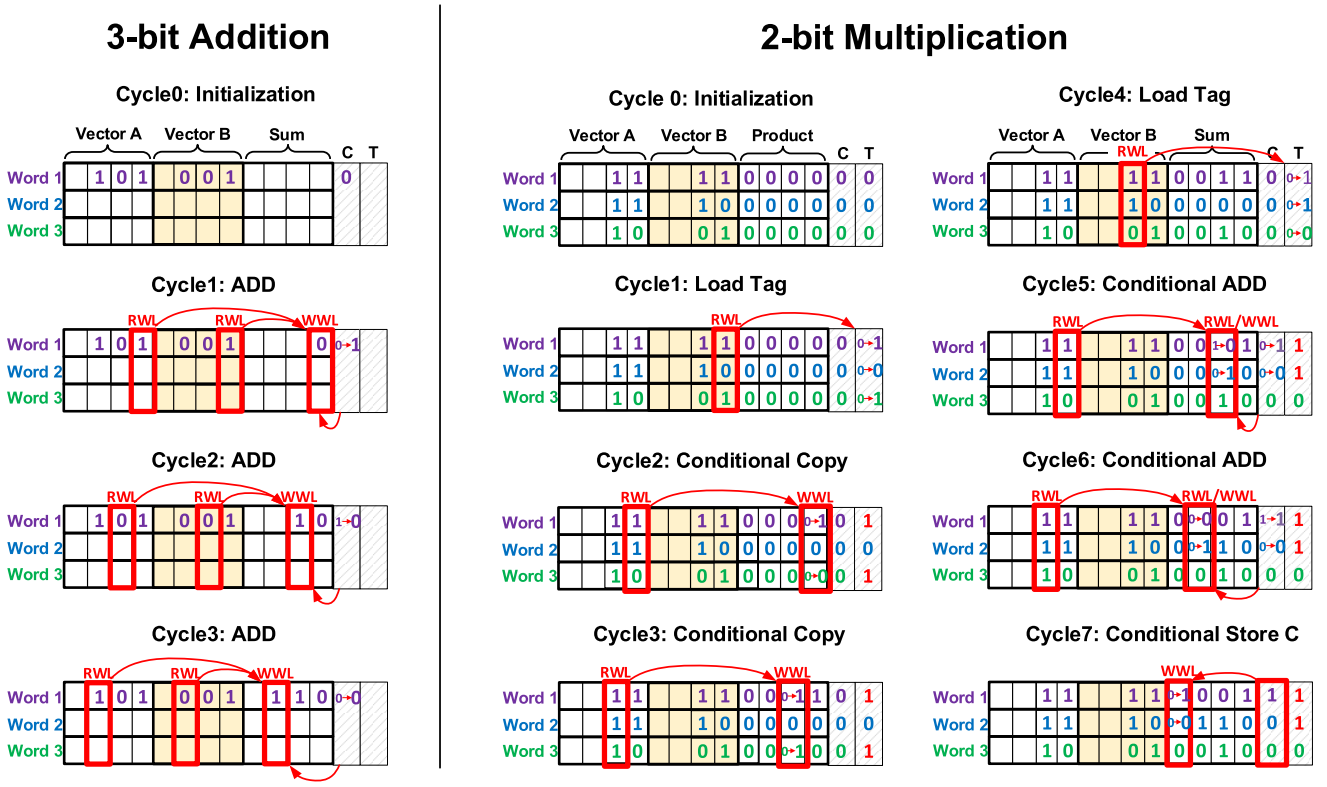


Fig. 5. 3-bit addition cycle-by-cycle demonstration (left) and 2-bit multiplication cycle-by-cycle demonstration (right).

A. Integer Addition and Subtraction

We use the addition of two vectors of 3-bit numbers (A and B) to explain how the addition algorithm is carried out bit-by-bit starting from the LSB (Fig. 5). The two vectors, each occupying three columns, need to be placed in the same array with their corresponding elements aligned on the same row but not necessarily abutted. In cycle 0, we first initialize the entire carry latch to 0 using instruction "Reset C." In cycle 1, we apply instruction "ADD" and provide the column address of the LSBs for RA and RB. We can either write the sum to an empty column of the array or one of the operand LSBs can be directly overwritten by the result depending on the destination address, RD, we give in the instruction. The carry latch is automatically updated with C_{out} at the end of the cycle. In cycles 2 and 3, we add the second and third bits' location the same way as we did in cycle 1. Thus, an N-bit addition takes $N+1$ cycles. Subtraction can be performed by first inverting vector B and then adding to A with the carry latch initialized to 1.

B. Unsigned Integer Multiplication

One way to perform multiplication is using shift and add. It requires the conditional copy and addition instruction enabled by the tag latch. As explained in Section III-B, if we enable the conditional execution feature, the tag latch becomes the local write bitline enable signal of the row, and the result of any instruction will only be written back into the destination bit RD if the tag latch stores 1. Fig. 5 demonstrates the example of a 2-bit multiplication. Suppose that vector A is the multiplicand and vector B is the multiplier. Initially, four columns in the array are reserved for the product and initialized to zero by setting all carry latches to 0 first using "Reset C" and then writing the carry latch back to the product columns in four cycles using "Store C." In the first computing cycle, the LSB of the multiplier is loaded to the tag latch using "Load T" instruction. In cycles 2 and 3, the multiplicands are copied to the product columns only if the tag latch in that row equals 1. In cycle 4, the second bit of the multiplier is loaded to the tag latch. In the next two cycles, for rows with tag equal to 1, the multiplicands are added to the second and third bits of the product, shifting the multiplicands by 1 to account for the multiplier bit position. Finally, we store C_{out} in the MSB of the product to complete the multiplication. Note that partial products are implicitly shifted as they are added using appropriate bit addressing in the bit-serial operation, and no explicit shift is performed.

C. Unsigned Integer Division

Division is conducted similarly by implicit shifting and subtracting from a partial result. The pseudo-code for the CRAM is given in Table III. The quotient is computed starting from the MSB. First, we copy the MSB of the dividend to the partial result (remainder). Then, we subtract the divisor from the partial result, put the result into a temporary location, and check whether the result is positive or negative by looking at the overflow bit C_{out} in the carry latch. A positive result from subtraction means the partial result is greater than the divisor, and the tag latch of that row will be set to 1. We conditionally

TABLE III
PSEUDO-CODE: UNSIGNED INTEGER DIVISION

Input: Divisor A[N-1:0], Dividend B[N-1:0] **Output: Quotient Q[N-1:0], Remainder R[N-1:0]**
[Note: extra N columns (TEMP) is used for temporary result]
0: initialize Q and R to 0
1: **for** i = 0 to N-1 **do**
2: copy B[N-1-i] → R[N-1-i]
3: R[N-1:N-1-i] - A[N-1:0] → {C_{out}, TEMP[N-1:0]}
4: if {C_{out}, TEMP[N-1:0]} is positive, update Tag to 1
5: (if Tag = 1) write 1 into bit Q[N-1-i]
6: (if Tag = 1) copy TEMP[i:0] to R[N-1:N-1-i]
7: **end for**
8: **return** Q, R

update the corresponding bit in the quotient and the remainder if the tag is 1. We repeat the previous steps N times until all the bits of the quotient are computed.

D. Comparison and Search

The arithmetic comparison between two operands in memory like "greater/less than" or "equal to" can be performed using subtraction or XOR logic operation. The CRAM also provides a multi-bit search operation between the operands stored in memory and the pattern given in the instruction, as in a content addressable memory (CAM). The "Search" operation is achieved by repeatedly using the CRAM single-cycle instruction "Equal," which compares all the bits in the column specified by bit-address RA with the 8th bit of the CRAM instruction (the LSB of bit-address RB field) and writes the result into the Tag latch. An N-bit pattern is compared bit-by-bit with N CRAM instructions, and therefore requires N cycles to finish the Search operation. The Search operation is frequently used in floating-point addition and subtraction.

E. Floating-Point Arithmetic

Taking 32-bit IEEE-754 floating point as an example, we will demonstrate one way to implement floating-point arithmetic on the CRAM using repeated conditional integer addition, subtraction, multiplication, division, and search operation. A 32-bit floating number is represented by 1 sign bit in the MSB followed by an 8-bit exponent and a 23-bit mantissa. During computation, we always use one extra memory column of all 1s to represent the implicit 24th bit of the mantissa. Floating-point multiplication and division is relatively simple. First, the result sign bit can be determined by XOR the operand sign bits. Then, an 8-bit addition between the two exponents is performed if it is multiplication or 8-bit subtraction if it is division. Then, a 24-bit multiplication or division between the mantissa is performed. However, floating-point addition and subtraction is much more complicated. Table IV shows the pseudo-code for floating-point addition. First, we equalize the exponents of the operands by shifting the one of the mantissa. If the operand A has a larger exponent, we right-shift the mantissa of operand B by the difference of the two exponents. Because the mantissa has at most 24 bits, we shift at most 24 times. Next, we add the mantissa if the signs of A and B

TABLE IV
PSEUDO-CODE: FLOATING-POINT ADDITION

Input: A[31:0], B[31:0] **Output: S[31:0]**
[Note: extra 32 columns (TEMP) is used for temporary result]
[Note: $S_{A/B/S}$: sign bit, $E_{A/B/S}$: exponent, $M_{A/B/S}$: Mantissa]
I. Equalize exponent: (first consider the case $E_A \geqslant E_B$)
0: $E_A - E_B \rightarrow E_{TEMP}$
1: compare & if($E_A \geqslant E_B$) copy $E_A \rightarrow E_S$
2: **for** i = 1 to 24 **do**
3: Search for row with E_{TEMP} = i, right shift M_B by i → M_{TEMP}
4: **end for**
5: compare & if($E_{TEMP} \geqslant 24$), clear M_{TEMP} to all 0
II: Add Mantissa
6: XOR S_A, $S_B \rightarrow$ Tag
7: if(Tag = 0) $M_A + M_{TEMP} \rightarrow M_S$, $S_A \rightarrow S_S$
8: if(Tag = 1 & $M_A > M_{TEMP}$) $M_A - M_{TEMP} \rightarrow M_S$, $S_A \rightarrow S_S$
9: if(Tag = 1 & $M_A < M_{TEMP}$) $M_{TEMP} - M_A \rightarrow M_S$, $S_B \rightarrow S_S$
III: Normalize result
10: **for** i = 1 to 24 **do**
11: Search M_S with #i leading 0, left shift M_S by i, E_S - i → E_S
12: **end for**
(Repeat previous steps again for $E_B \geqslant E_A$ case)
13: $E_B - E_A \rightarrow E_{TEMP}$
14: …

are the same. Otherwise, we subtract B from A if A has a larger mantissa or subtract A from B if mantissa B is larger. Finally, we need to normalize the result by left-shifting the result until the 24th bit of the mantissa is 1.

V. TEST CHIP AND MEASUREMENT RESULTS

To test the proposed in-/near-memory concept, we incorporate the CRAM into an Internet of Things (IoT) processor. The chip consists of an ARM Cortex-M0 CPU [32], a separate CRAM control bus, and eight 16-KB CRAM banks (in total 128-kB memory with 2048 computing rows). These memories can function either as traditional or compute memories. The ARM core can load or store data using standard memory I/O or perform computation in memory by directly generating and sending 32-bit CRAM instructions to the CRAM controller in each bank using memory mapped I/O. The CRAM control bus servers as a direct memory access (DMA) controller. Complex multi-cycle computing instructions can be first stored in one of the CRAM banks and the ARM M0 core can then program the DMA controller to stream instructions from one bank to one or multiple other selected banks, while M0 simultaneously performs other processing with the remaining memory banks.

Fig. 6 shows the layout of the CRAM bank and die photograph of the prototype chip fabricated in the 28-nm CMOS. A single memory bank is 245 × 625 μm with 70% array efficiency. The chip size is 1.5 by 1.7 mm. Fig. 7 shows the measured frequency and energy efficiency of 8-bit addition and multiplication across the supply voltage. The energy efficiency ratio between 8-bit addition and multiplication is constant over voltage because of the constant CRAM cycle counts' ratio between the two operations. Therefore, their energy efficiency curves are overlapped in Fig. 7. During testing, we found that the low-swing sensing technique was sufficient to prevent read disturbance issues, and therefore

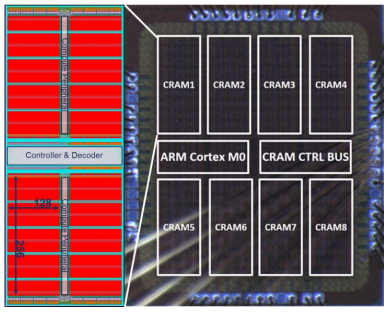


Fig. 6. Layout of the CRAM bank and die photograph.

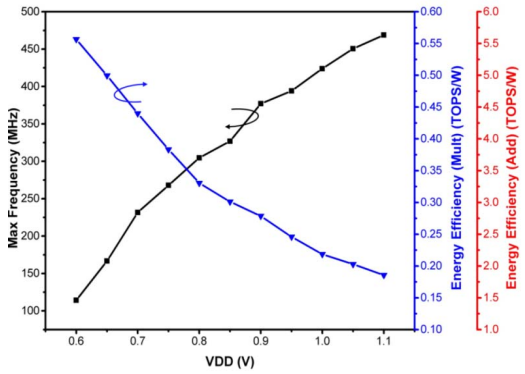


Fig. 7. Frequency and energy efficiency of 8-bit multiplication and addition at different VDD.

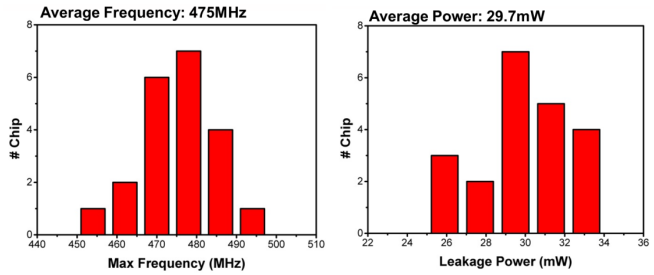


Fig. 8. Maximum frequency and leakage power distribution of 21 dies at 1.1 V.

word-line voltage underdrive was not used, which would have otherwise slowed down the CRAM. The best energy efficiency is achieved at 0.6 V and 114 MHz, resulting in 0.56 TOPS/W for 8-bit multiplication and 5.27 TOPS/W for 8-bit addition. At 1.1 V, the average frequency of 475 MHz results in 122 GOPS for 8-bit addition and 9.4 GOPS for 8-bit multiplication. If the memory size is scaled to 35 MB, which has a similar capacity of an L3 cache in a modern server-class processor, the CRAM is estimated to provide 34.2 TOPS of 8-bit additions while consuming 51.2 W. Fig. 8 gives the measured frequency and leakage power distributions for 21 measured dies. The performance of different multi-cycle operations is summarized in Table V.

Fig. 9 shows the performance of the test chip for diverse computationally intensive tasks ranging from neural networks to graph and signal processing. The total latency in terms of micro seconds is compared with a baseline operation where the

TABLE V
PERFORMANCE OF TEST CHIP AT 475 MHz

Type	Operation	32-bit Performance[1]	8-bit Performance[1]
Logic	AND	30.4GOPS	122 GOPS
	NOR		
	XOR		
	NAND		
	OR		
	XNOR		
Integer	Add	30.4 GOPS	122 GOPS
	Sub	15.2 GOPS	60.8 GOPS
	Mult	0.83 GOPS	9.40 GOPS
	UDiv	0.57 GOPS	6.97 GOPS
32-bit Float Point	Add/Sub	0.20 GFLOPS	/
	Mult	1.43 GFLOPS	/
	Div	1.40 GFLOPS	/
Comparison	Equal	14.9 GOPS	57 GOPS
	Greater/Less		
	Search	30.4 GOPS	122 GOPS

1. Measured performance on 128KB CRAM test chip at 475MHz

CRAMs are only used as data memories and the computation is entirely performed on the ARM CPU. The baseline memory clock frequency is 1.2× faster than the CRAM, because it does not have the write-back and latch-update phases in each cycle. The first benchmark is the first convolutional layer from Cuda-convnet [33], and the second is the last fully connected layer from Alex-net [34]. Due to their size, these layers must be executed in multiple smaller sub-sections. The third application consists of 512 simultaneous 32-tap FIR filters, and the fourth application performs traversal of a directed graph represented by a 192 × 192 adjacency matrix. The type of applications that are suitable for the CRAM should always have part of the operands remain stationary in memory. For example, the filter weights of the convolutional neural networks are assumed to be preloaded in memory already. Before the computation, we only need to transfer the input feature maps from other banks, and after the in-memory computation partial outputs are read out to be aggregated in the CPU. The details of how a convolutional layer is mapped into the CRAM are given in [35]. Fig. 9 shows the percentage of time that each application spends on input loading, output storing, and in-memory computation. Speedup, compared with executing the same workload with the ARM Cortex-M0, varies from 6.04× to 87×. The largest performance gain is achieved when the application is compute-heavy and low on input–output movement.

Table VI compares the proposed approach with recent state-of-the-art in-memory accelerators and one non in-memory digital system-on-chip (SoC) [36]. We can see that the CRAM provides a good trade-off point among in-memory and non in-memory architectures. From the comparison table, the analog in-memory architectures show excellent (best reported) energy efficiency but suffer from poor precision (mostly 1-bit weight), scalability (small macro size), and limited functionality (add and multiply only). The conventional digital SoCs can perform general-purpose high-precision arithmetic operations, but have poor energy efficiency. The CRAM is a digital in-/near-memory solution that provides an intermediate design point between the analog in-memory and the conventional Von Neumann architectures. It offers better precision, scalability,

	CONV		FC		FIR		GRAPH	
Testbench	Cuda-Convnet 1st layer		AlexNet last layer		512 32-Tap Filter		Nearest Neighbor Traversal	
Input Size	24x24x3		24x1		32x10		192x192	
Parameter Size	5x5x3x64		1000x24		512x32		0	
Output Size	1x1x64		1000x1		512x10		192x192	
Bit Precision	8		8		4		1	
# Array used for compute	3		6		2		2	
# Wordline for computation	375		1000		512		192	
	time(us)	**percentage**	**time(us)**	**percentage**	**time(us)**	**percentage**	**time(us)**	**percentage**
Total Latency	83.4	100	70.4	100	529	100	3,311	100
Input loading	38.6	46.2	0.05	0.07	0.67	0.13	2.4	0.07
CRAM Compute	7.2	8.7	44.8	63.6	387	73.2	3,277	99.0
output readout	37.6	45.0	25.6	36.3	141	26.6	31.6	0.95
	time(us)	**speedup**	**time(us)**	**speedup**	**time(us)**	**speedup**	**time(us)**	**speedup**
Baseline*	504	6.04×	2,060	29.3×	14,246	26.9×	288,520	87.1×

*Baseline clock frequency is 1.2× faster than CRAM

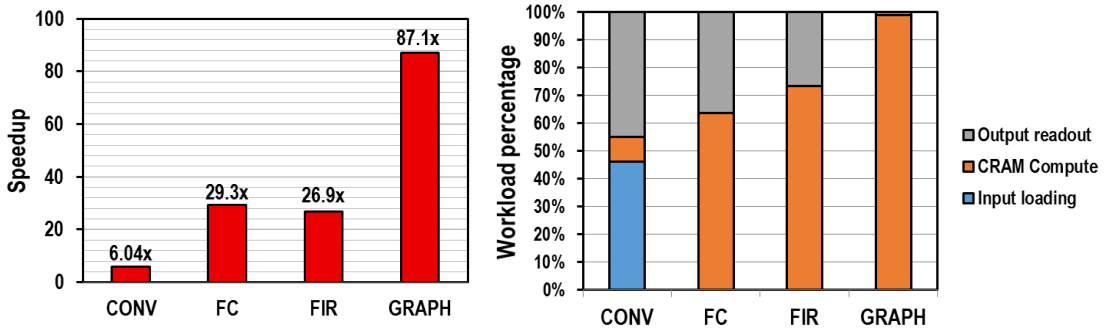


Fig. 9. Performance comparison between the CRAM and baseline scenario (top) and workload breakdown (bottom).

TABLE VI
COMPARISON WITH PREVIOUS WORK

	This Work	JSSC2019 [15]	ISSCC2019 [17]	JSSC2019 [19]	JSSC2018 [22]	JSSC2018 [36]
Technology	28nm	65nm	28nm	65nm	40nm	28nm
Supply Voltage	0.6~1.1V	0.8~1.2V	0.6~0.9V	0.68~1.2V	0.65~0.9V	0.6~1.1V
SRAM Macro size	16KB	2KB	18.75KB	4.8KB	8KB	-
SRAM bitcell	8T	10T	8T	8T	10T	-
Bitcell size (Ratio to foundry 6T)	0.78um^2(4.98x)	-	-	1.8um^2(3.16x)	0.84um^2(5.61x)	-
Method of Computing	In/Near-Memory Digital	In-Memory Analog(Current)	In-Memory Analog(Time)	In-Memory Analog(Charge)	In/Near-Memory Digital	Digital SoC
Type of Supported Functions	Logic/Add/Sub/ Mult/Div/FP	Add/Mult	Add/Mult	Add/Mult	Logic	General Purpose
Bit precision	Arbitrary	6b (input) 1b (weight)	8b (input) 1b (weight)	1b (input) 1b (weight)	Arbitrary	16b (input) 8/16b (weight)
Die Area (mm2)	2.7	0.063	0.22	12.6	1.28	5.76
Max Freqency (MHz)	475	5	400	100	90	1200
Normalized Performance (GOPS)*	32.7	0.75	-	147	-	49.8
Performance per unit Area (GOPS/mm^2)**	27.3	11.9	-	11.7	-	8.65
Normalized Energy Efficiency (TOPS/W) *	0.55 (mult) 5.27 (add)	4.81	5.83	10.3	-	2.45

*All normalized to 8-by-8 bit Multiply and Add Operation
** Using SRAM area only

and programmability than analog IMC and improved energy efficiency over conventional architectures.

VI. CONCLUSION

To summarize, we have presented an approach to general IMC that repurposes conventional SRAMs in caches to enable a wide range of flexible bit-width operations, enabling a host of rapidly evolving software algorithms such as neural networks. The CRAM is an area-efficient and low invasive technique that exploits vector-based, bit-serial in-/near-memory arithmetic. It achieves high throughputs by exploiting the massive bandwidth inside the SRAM banks and good energy efficiency by suppressing data movement energy.

ACKNOWLEDGMENT

The authors would like to thank TSMC University Shuttle Program for chip fabrication.

REFERENCES

[1] W. A. Wulf and S. A. McKee, "Hitting the memory wall: Implications of the obvious," *ACM SIGARCH Comput. Archit. News*, vol. 23, no. 1, pp. 20–24, Mar. 1995.

[2] M. Huang, M. Mehalel, R. Arvapalli, and S. He, "An energy efficient 32-nm 20-MB shared on-die L3 cache for intel Xeon processor E5 family," *IEEE J. Solid-State Circuits*, vol. 48, no. 8, pp. 1954–1962, Aug. 2013.

[3] D. G. Elliott, W. M. Snelgrove, and M. Stumm, "Computational ram: A memory-simd hybrid and its application to Dsp," in *Proc. IEEE Custom Integr. Circuits Conf.*, May 1992, pp. 30–36.

[4] N. P. Jouppi *et al.*, "In-datacenter performance analysis of a tensor processing unit," *ACM SIGARCH Comput. Archit. News*, vol. 45, no. 2, pp. 1–12, 2017.

[5] M. Horowitz, "Computing's energy problem (and what we can do about it)," in *IEEE ISSCC Dig. Tech. Papers*, Feb. 2014, pp. 10–14.

[6] Y. LeCun, Y. Bengio, and G. Hinton, "Deep learning," *Nature*, vol. 521, no. 7553, pp. 436–444, 2015.

[7] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit. (CVPR)*, Jun. 2016, pp. 770–778.

[8] Z. Li, J. Wang, D. Sylvester, D. Blaauw, and H. S. Kim, "A 1920 × 1080 25-frames/s 2.4-tops/W low-power 6-D vision processor for unified optical flow and stereo depth with semi-global matching," *IEEE J. Solid-State Circuits*, vol. 54, no. 4, pp. 1048–1058, Apr. 2019.

[9] S. Smets, T. Goedemé, A. Mittal, and M. Verhelst, "A 978GOPS/W flexible streaming processor for real-time image processing applications in 22 nm FDSOI," in *IEEE ISSCC Dig. Tech. Papers*, Feb. 2019, pp. 44–46.

[10] S. Li, D. Niu, K. T. Malladi, H. Zheng, B. Brennan, and Y. Xie, "DRISA: A dram-based reconfigurable In-Situ accelerator," in *Proc. 50th Annu. IEEE/ACM Int. Symp. Microarchitecture (MICRO)*, Oct. 2017, pp. 288–301.

[11] M. N. Bojnordi and E. Ipek, "Memristive Boltzmann machine: A hardware accelerator for combinatorial optimization and deep learning," in *Proc. IEEE Int. Symp. High Perform. Comput. Archit. (HPCA)*, Mar. 2016, pp. 1–13.

[12] W.-H. Chen *et al.*, "A 65 nm 1 Mb nonvolatile computing-in-memory ReRAM macro with sub-16 ns multiply-and-accumulate for binary DNN AI edge processors," in *IEEE Int. Solid-State Circuits Conf. (ISSCC) Dig. Tech. Papers*, Feb. 2018, pp. 494–496.

[13] L. Fick, D. Blaauw, D. Sylvester, S. Skrzyniarz, M. Parikh, and D. Fick, "Analog in-memory subthreshold deep neural network accelerator," in *Proc. IEEE Custom Integr. Circuits Conf. (CICC)*, Apr./May 2017, pp. 1–4.

[14] J. Zhang, Z. Wang, and N. Verma, "In-memory computation of a machine-learning classifier in a standard 6T SRAM array," *IEEE J. Solid-State Circuits*, vol. 52, no. 4, pp. 915–924, Apr. 2017.

[15] A. Biswas and A. P. Chandrakasan, "CONV-SRAM: An energy-efficient SRAM with in-memory dot-product computation for low-power convolutional neural networks," *IEEE J. Solid-State Circuits*, vol. 54, no. 1, pp. 217–230, Jan. 2019.

[16] M. Kang, S. K. Gonugondla, A. Patil, and N. R. Shanbhag, "A multi-functional in-memory inference processor using a standard 6T SRAM array," *IEEE J. Solid-State Circuits*, vol. 53, no. 2, pp. 642–655, Feb. 2018.

[17] J. Yang *et al.*, "24.4 sandwich-RAM: An energy-efficient in-memory BWN architecture with pulse-width modulation," in *IEEE Int. Solid-State Circuits Conf. (ISSCC) Dig. Tech. Papers*, San Francisco, CA, USA, Feb. 2019, pp. 394–396.

[18] H. Jia, Y. Tang, H. Valavi, J. Zhang, and N. Verma, "A microprocessor implemented in 65 nm CMOS with configurable and bit-scalable accelerator for programmable in-memory computing," 2018, *arXiv:1811.04047*. [Online]. Available: https://arxiv.org/abs/1811.04047

[19] H. Valavi, P. J. Ramadge, E. Nestler, and N. Verma, "A 64-Tile 2.4-Mb in-memory-computing CNN accelerator employing charge-domain compute," *IEEE J. Solid-State Circuits*, vol. 54, no. 6, pp. 1789–1799, Jun. 2019.

[20] P. Srivastava *et al.*, "PROMISE: An end-to-end design of a programmable mixed-signal accelerator for machine-learning algorithms," in *Proc. 5th Annu. Int. Symp. Comput. Archit.*, Los Angeles, CA, USA, 2018, pp. 43–56.

[21] J. Wang *et al.*, "A compute SRAM with bit-serial integer/floating-point operations for programmable in-memory vector acceleration," in *IEEE Int. Solid-State Circuits Conf. (ISSCC) Dig. Tech. Papers*, Feb. 2019, pp. 224–226.

[22] Y. Zhang, L. Xu, Q. Dong, J. Wang, D. Blaauw, and D. Sylvester, "Recryptor: A reconfigurable cryptographic cortex-M0 processor with in-memory and near-memory computing for IoT security," *IEEE J. Solid-State Circuits*, vol. 53, no. 4, pp. 995–1005, Apr. 2018.

[23] Q. Dong *et al.*, "A 4+2T SRAM for searching and in-memory computing with 0.3-V V_{DDmin}," *IEEE J. Solid-State Circuits*, vol. 53, no. 4, pp. 1006–1015, Apr. 2018.

[24] 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 J. Solid-State Circuits*, vol. 51, no. 4, pp. 1009–1021, Apr. 2016.

[25] K. E. Batcher, "Bit-serial parallel processing systems," *IEEE Trans. Comput.*, vol. C-31, no. 5, pp. 377–384, May 1982.

[26] P. Denyer and D. Renshaw, *VLSI Signal Processing A Bit-Serial Approach*. Reading, MA, USA: Addison-Wesley, 1985.

[27] J. Lee, C. Kim, S. Kang, D. Shin, S. Kim, and H.-J. Yoo, "UNPU: An energy-efficient deep neural network accelerator with fully variable weight bit precision," *IEEE J. Solid-State Circuits*, vol. 54, no. 1, pp. 173–185, Jan. 2019.

[28] K. Ueyoshi *et al.*, "QUEST: Multi-purpose log-quantized DNN inference engine stacked on 96-MB 3-D SRAM using inductive coupling technology in 40-nm CMOS," *IEEE J. Solid-State Circuits*, vol. 54, no. 1, pp. 186–196, Jan. 2019.

[29] K. C. Akyel *et al.*, "DRC^2: Dynamically reconfigurable computing circuit based on memory architecture," in *Proc. IEEE Int. Conf. Rebooting Comput. (ICRC)*, San Diego, CA, USA, Oct. 2016, pp. 1–8.

[30] J. Seo *et al.*, "A 45 nm CMOS neuromorphic chip with a scalable architecture for learning in networks of spiking neurons," in *Proc. IEEE Custom Integr. Circuits Conf. (CICC)*, Sep. 2011, pp. 1–4.

[31] C. Eckert *et al.*, "Neural Cache: Bit-serial in-cache acceleration of deep neural networks," in *Proc. ACM/IEEE 45th Annu. Int. Symp. Comput. Archit. (ISCA)*, Jun. 2018. vol. 39, no. 3, pp. 11–19.

[32] *ARM CORTEX-M Series*. Accessed: Oct. 2017. [Online]. Available: http://www.arm.com/products/processors/cortex-m

[33] *Cuda-Convnet*. Accessed: Sep. 2018. [Online]. Available: https://code.google.com/archive/p/cuda-convnet/

[34] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Proc. Adv. Neural Inf. Process. Syst. (NIPS)*, 2012, pp. 1097–1105.

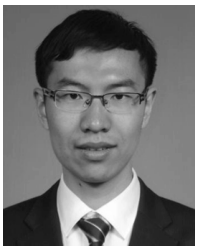
[35] C. Eckert *et al.*, "Neural cache: Bit-serial in-cache acceleration of deep neural networks," in *Proc. 45th Annu. Int. Symp. Comput. Archit.*, Los Angeles, CA, USA, 2018, pp. 383–396.

[36] P. N. Whatmough, S. K. Lee, D. Brooks, and G.-Y. Wei, "DNN engine: A 28-nm timing-error tolerant sparse deep neural network processor for IoT applications," *IEEE J. Solid-State Circuits*, vol. 53, no. 9, pp. 2722–2731, Sep. 2018.

Jingcheng Wang (S'15) received the B.S. degree in electrical engineering and computer science from the University of Michigan, Ann Arbor, MI, USA, and Shanghai Jiaotong University, Shanghai, China, in 2014, and the M.S. degree from the University of Michigan in 2017, where he is currently pursuing the Ph.D. degree in electrical engineering.

His current research interests include low-power system and memory design, near-/in-memory computing, and neural network accelerators.

Mr. Wang was a recipient of the Dwight F. Benton Fellowship in 2015 from the University of Michigan.



Xiaowei Wang (S'19) received the B.Eng. degree in electronic information science and technology from Tsinghua University, Beijing, China, in 2015, and the M.S. degree in computer science and engineering from the University of Michigan, Ann Arbor, MI, USA, in 2017, where he is currently pursuing the Ph.D. degree in computer science and engineering.

His current research interests include application-specific architectures based on in-memory computing and hardware/software co-design for machine learning workloads.

Charles Eckert (S'16) received the B.S. degree in computer engineering from SUNY Binghamton, Binghamton, NY, USA, in 2017, and the M.S.E. degree in computer science and engineering from the University of Michigan, Ann Arbor, MI, USA, in 2018, where he is currently pursuing the Ph.D. degree in computer science and engineering.

His research focuses on using in-memory computing to accelerate machine learning applications.

Mr. Eckert was a recipient of the NDSEG Fellowship along with NSF GFRP Honorable mention in 2019.

Arun Subramaniyan (S'16) received the B.E. degree (Hons.) in electrical and electronics from the Birla Institute of Technology and Science-Pilani, Pilani, India, in 2015, and the M.S. degree in computer science and engineering from the University of Michigan, Ann Arbor, MI, USA, in 2018, where he is currently pursuing the Ph.D. degree in computer science and engineering.

He is broadly interested in computer architecture, specifically in-memory computing and developing hardware accelerated systems for genomics.

Mr. Subramaniyan was a recipient of the Rackham International Student Fellowship in 2017 and the Precision Health Scholar Award in 2018 from the University of Michigan.

Reetuparna Das (M'13) received the Ph.D. degree in computer science and engineering from Pennsylvania State University, University Park, PA, USA.

She was a Research Scientist with Intel Labs and the Researcher-in-Residence with the Center for Future Architectures Research. She is currently an Assistant Professor with the University of Michigan, Ann Arbor, MI, USA. She also serves as the Co-Founder and CTO of a precision medicine startup, Sequal Inc. Some of her recent projects include in-memory architectures, custom computing for precision health and AI, fine-grain heterogeneous core architectures for mobile systems, and low-power scalable interconnects for kilo-core processors. She has authored more than 45 articles. She holds seven patents.

Dr. Das was a recipient of the two IEEE Top Picks Awards, the NSF CAREER Award, the CRA-W's Borg Early Career Award, and the Sloan Foundation Fellowship. She has been inducted into IEEE/ACM MICRO and ISCA Hall of Fame. She has served on over 30 Technical Program Committees and is serving as the Program Co-Chair for MICRO-52.

David Blaauw (M'94–SM'07–F'12) received the B.S. degree in physics and computer science from Duke University, Durham, NC, USA, in 1986, and the Ph.D. degree in computer science from the University of Illinois at Urbana–Champaign, Champaign, IL, USA, in 1991.

In 2001, he was with Motorola, Inc., Austin, TX, USA, where he was the Manager of the High Performance Design Technology Group. Since August 2001, he has been with the faculty of the University of Michigan, Ann Arbor, MI, USA, where he is the Kensall D. Wise Collegiate Professor of electrical engineering and computer science (EECS). He is also the Director of the Michigan Integrated Circuits Laboratory. He has authored or coauthored more than 600 articles. He holds 65 patents. He has extensively researched ultralow-power computing using subthreshold computing and analog circuits for millimeter sensor systems, which was selected by the MIT Technology Review as one of the year's most significant innovations. For high-end servers, his research group introduced a so-called near-threshold computing, which has become a common concept in semiconductor design. Recently, he has pursued research in cognitive computing using analog, in-memory neural networks for edge-devices and genomics acceleration.

Dr. Blaauw was a recipient of the Motorola Innovation Award at Motorola, Inc., the 2016 SIA-SRC Faculty Award for lifetime research contributions to the U.S. semiconductor industry, and the numerous Best Paper Awards and nominations. He was the General Chair of the IEEE International Symposium on Low Power, the Technical Program Chair for the ACM/IEEE Design Automation Conference, and serves on the IEEE International Solid-State Circuits Conference's Technical Program Committee.

Dennis Sylvester (S'95–M'00–SM'04–F'11) received the Ph.D. degree in electrical engineering from the University of California at Berkeley (UC Berkeley), Berkeley, CA, USA, where his dissertation was recognized with the David J. Sakrison Memorial Prize as the most outstanding research in the electrical engineering and computer science (EECS) Department.

He was the Founding Director of the Michigan Integrated Circuits Laboratory (MICL), University of Michigan, Ann Arbor, MI, USA, a group of 10 faculty and 70+ graduate students. He was a Research Staff with the Advanced Technology Group of Synopsys, Mountain View, CA, USA, and also with Hewlett-Packard Laboratories, Palo Alto, CA, USA, and a Visiting Professor with the National University of Singapore, Singapore, and Nanyang Technological University, Singapore. He co-founded Ambiq Micro, Austin, TX, USA, a Fabless Semiconductor Company, developing ultralow-power mixed-signal solutions for compact wireless devices. He is currently a Professor with the Department of Electrical Engineering and Computer Science, University of Michigan. He has authored or coauthored more than 500 articles along with one book and several book chapters. He holds 48 U.S. patents. His research interests include the design of millimeter-scale computing systems and energy-efficient near-threshold computing.

Dr. Sylvester was a recipient of the NSF CAREER Award, the Beatrice Winner Award at ISSCC, the IBM Faculty Award, the SRC Inventor Recognition Award, the University of Michigan Henry Russel Award for Distinguished Scholarship, and ten Best Paper Awards and nominations. He was named one of the Top Contributing Authors at ISSCC and most prolific author at IEEE Symposium on VLSI Circuits. He serves on the Technical Program Committee for the IEEE International Solid-State Circuits Conference and on the Administrative Committee for the IEEE Solid-State Circuits Society. He continues to serve as an Associate Editor for the IEEE JOURNAL OF SOLID-STATE CIRCUITS, the IEEE TRANSACTIONS ON CAD, and the IEEE TRANSACTIONS ON VERY LARGE SCALE INTEGRATION (VLSI) SYSTEMS, and was an IEEE Solid-State Circuits Society Distinguished Lecturer.