In-Memory Processing on the Spintronic CRAM: From Hardware Design to Application Mapping

Masoud Zabihi, Zamshed Chowdhury, Zhengyang Zhao,
Ulya R. Karpuzu, Jian-Ping Wang, and Sachin S. Sapatnekar

Abstract—The Computational Random Access Memory (CRAM) is a platform that makes a small modification to a standard spintronics-based memory array to organically enable logic operations within the array. CRAM provides a true in-memory computational platform that can perform computations within the memory array, as against other methods that send computational tasks to a separate processor module or a near-memory module at the periphery of the memory array. This paper describes how the CRAM structure can be built and utilized, accounting for considerations at the device, gate, and functional levels. Techniques for constructing fundamental gates are first overviewed, accounting for electrical and noise margin considerations. Next, these logic operations are composed to schedule operations in the array that implement basic arithmetic operations such as addition and multiplication. These methods are then demonstrated on 2D convolution with multibit data, and a binary neural inference engine. The performance of the CRAM is analyzed on near-term and longer-term spintronic device technologies. Significant improvements in energy and execution time for the CRAM-based implementation over a near-memory processing system are demonstrated, and can be attributed to the ability of CRAM to overcome the memory access bottleneck, and to provide high levels of parallelism to the computation.

Index Terms—Spintronics, In-memory computing, Memory bottleneck, STT-MRAM, Neuromorphic Computing, Nonvolatile memory

1 Introduction

Today's computational engines are inadequately equipped to handle the demands of future big-data applications. With the size of data sets growing exponentially with time [1], the computational demands for data analytics applications are becoming even more forbidding, and the mismatch between application requirements and evolutionary improvements in hardware is projected to become more extreme. Current hardware paradigms have moved towards greater specialization to handle this challenge, and specialized units that enable memory-centric computing are a vital ingredient of any future solution.

Technology Node	40nm	10nm HP	10nm LP
Energy of 64-bit data communication versus computation	1.55×	5.75×	5.77×

Table 1: Communication vs. computation energy, adapted from [2].

However, key bottlenecks for large-scale data analytics applications, even in state-of-the-art technology solutions, are the memory capacity, communication bandwidth, and performance requirements within a tight power budget. Technology scaling to date has improved the efficiency of logic more than communication, and communication energy dominates computation energy [3], [4]. Table 1 compares the cost of computation (a double-precision fused multiply add) with communication (a 64-bit read from an on-chip SRAM). The ratio of communication energy to computation energy increases from 1.55× at 40nm to approximately 6× at 10nm, for both the high performance (HP) and low power (LP) variants. Even worse, transferring the same quantity of data off-chip, to main memory, requires more than 50× computation energy even at 40nm [2]. Such off-chip accesses become increasingly necessary as data sets grow larger, and even the cleverest latency-hiding techniques cannot conceal their overhead. At the same time, the inevitable trend of higher degrees of parallel processing hurts data locality and results in increased execution time, power, and energy for data communication [3].

Moreover, general-purpose processors are often inefficient in computing for emerging applications: this has motivated a trend towards specialized accelerator units, tailored to specific classes of applications that they can efficiently execute. The trend of increasing data set sizes, coupled with the large cost (in terms of both energy and latency) of transporting data to the processor, have prompted the need for a significant departure from the traditional model of CPU-centric computing. An effective way to overcome the memory bottleneck and maintain the locality of computing operations is to embed compute capability into the main memory. In recent years, two classes of approaches have been proposed:

- *near-memory computing* places computational units at the periphery of memory for fast data access.
- *true in-memory computing* uses the memory array to perform computations through simple reconfigurations.

In this paper, we present an approach based on the spintronics-based computational random access memory (CRAM) paradigm. The CRAM concept [5] uses a small modification to the MTJ-based memory cell that enhances its versatility and enables logic operations through reconfiguration that enables the use of current-steered logic. Unlike many other approaches, this method fits the description of *true in-memory computing*, where computations are performed natively within the memory array and massive parallelism is possible, e.g., with each row of the memory performing an independent computation. The CRAM-based approach is digital, unlike prior analog-like in-memory/near-memory solutions [6], [7], which provides

The authors are with the Department of Electrical and Computer Engineering, University of Minnesota, Minneapolis, MN, USA. This work was supported in part by NSF SPX Award CCF-1725420, and by C-SPIN, one of the six SRC STARnet Centers, sponsored by MARCO and DARPA.

more robustness to variations due to process drifts, particularly in immature technologies than analog schemes. This sensitivity implies that digital implementations can achieve superior recognition accuracy over analog solutions when the full impact of errors and variations are factored in.

Our solution is based on spintronics technology, which is attractive because of its robustness, high endurance, and its trajectory towards fast improvement [8], [9]. The outline of the CRAM approach was first proposed in [5], operating primarily at the technology level with some expositions at the circuit level. The work was developed further to show system-level applications and performance estimations in [10]. In this work, we bridge the two to provide an explicit link between CRAM technology, circuit implementations, and operation scheduling. We present technology alternatives and methods for building gates and arithmetic units, study scheduling and data placement issues, and show how this approach can be used to implement a sample application, which is chosen to be a neuromorphic inference engine for digit recognition.

The rest of the paper is organized as follows. Section 2 discusses CRAM architecture. In Section 3, we present an approach for designing arithmetic function at the device, gate, and functional levels. Given this design, a more detailed elaboration on scheduling CRAM operations is discussed in Section 4. We elaborate on two example applications, corresponding to implementations of 2D convolution and a neural inference engine, in Section 5. We discuss the evaluation and results in Section 6, related work in Section 7, and then conclude the paper in Section 8.

2 CRAM Architecture

2.1 MTJ Devices

The unit storage cell used in a typical STT-MRAM is an MTJ, which is composed of two ferromagnetic layers – a fixed polarizing layer and a free layer – separated by an ultrathin nonconductive MgO barrier [11]. We consider perpendicular MTJ (PMTJ) technology, where both the free layer and the fixed layer are magnetized perpendicular to the plane of the junction. When the magnetization orientations of the two layers are parallel to each other (referred to as the P state), applying a voltage across the MTJ causes electrons to tunnel through the ultrathin nonconductive layer without being strongly scattered, as a result of which we have high current flow and relatively low resistance, R_P [12]; when the magnetization orientations of two layers are anti-parallel to each other (referred to as the AP state), the MTJ has a higher resistance, R_{AP}. In this way, an MTJ can store logic 1 and 0 depending on its resistance state, and we define logic 1 and 0 for the AP and P states, respectively [13]. A critical attribute of an MTJ is the tunneling magnetoresistance ratio (TMR), defined as

$$\text{TMR} = \frac{R_{AP} - R_P}{R_P} \tag{1}$$

With an electrical current flowing through the MTJ, the magnetization direction of the free layer can be reversed due to the spin-transfer-torque (STT) effect, and thus the MTJ can be switched between P state and AP state. To flip the magnetization direction of the free layer, the current density should be larger than a threshold switching current density, J_c, which is technology-dependent.

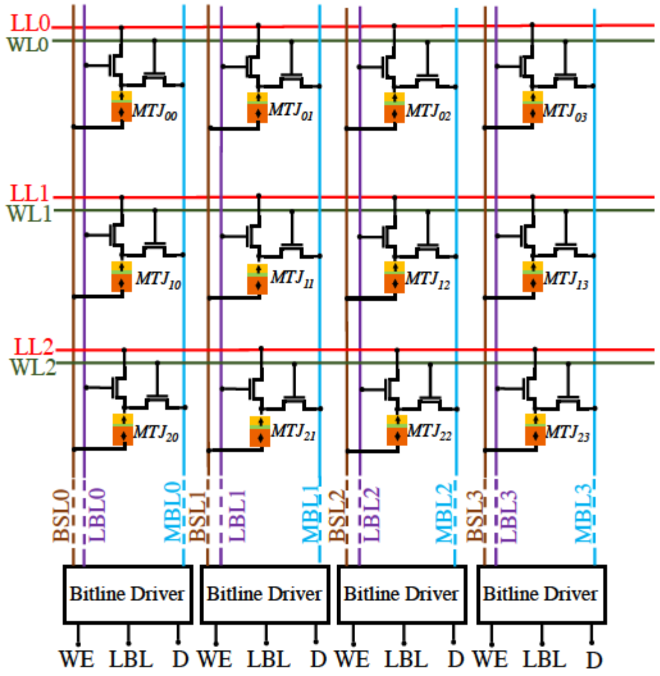


Figure 1: Overall structure of the CRAM.

2.2 The CRAM Array

The general structure of the spintronic CRAM is illustrated in Fig. 1. The overall configuration of the CRAM array is very similar to the standard 1-transistor 1-MTJ (1T1MTJ) STT-MRAM, except that the CRAM uses a 2T1MTJ bit-cell, with one additional transistor. Like the standard STT-MRAM memory array, the MTJ in each bit-cell is addressed using the memory word line (WL). The second transistor in the bit-cell, which enables logic operations, is enabled by selecting the logic bit line (LBL) for the transistor while turning off WL. The array can operate in two modes:

Memory mode: When the WL is high, it turns on an access transistor in each column and enables data to be read from or written into the MTJ through the memory bit line (MBL). The second transistor is turned off during this mode by holding down LBL, and the configuration is effectively identical to a bit cell in a memory array.

Logic mode: Turning on the LBL allows the MTJ to be connected to a logic line (LL) in each row. In the logic mode, several MTJs in a row are connected to the logic line. To realize a logic function with multiple inputs and one output, an appropriate voltage is applied to the bitlines. Since the states of the input MTJs are expressed in terms of their resistance, the current through the output MTJ depends on the input MTJ resistances, and if it exceeds the critical switching current, I_c, the output MTJ state is altered.

To understand the logic mode more clearly, consider the scenario where three MTJ devices are connected to the logic line, as shown in Fig. 2. The state variables are expressed in terms of the MTJ resistance, where resistances R_1 and R_2 correspond to the states of the two inputs, and R_o is the output MTJ resistance. Before the computation starts, the output MTJ state is set to a preset value. The bit select lines (BSLs) of the input MTJs are connected to a pulse voltage, while that of the output MTJ is grounded. This configuration corresponds to the application of a voltage V_{BSL} across a

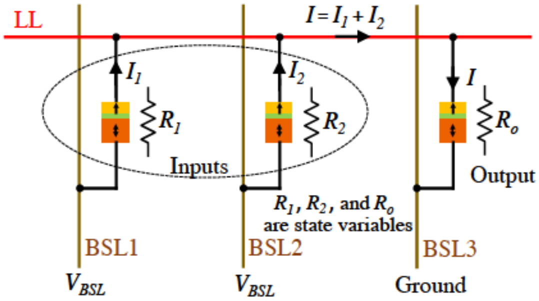


Figure 2: Performing a logic operation in a row of the CRAM array.

resistance of $(R_1 \mid\mid R_2)$ in series with R_o. As a result, a current I flows through the logic line:

$$I = V_{BSL} \Big/ \left[\left(\frac{R_1 R_2}{R_1 + R_2}\right) + R_o\right] \tag{2}$$

If $I > I_c$, where I_c is the critical threshold current required to switch the output MTJ from it preset value, the output state changes; otherwise it remains the same.

2.3 Performing Logic Operations Across Rows

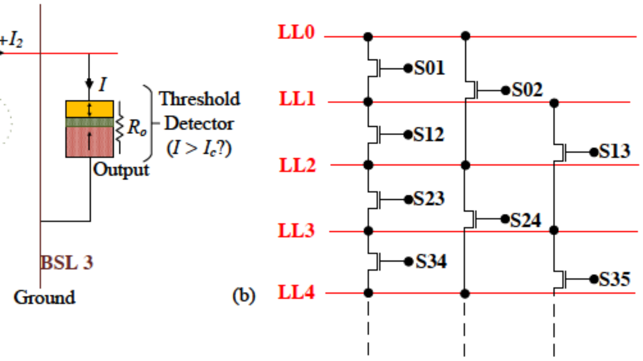


Figure 3: Switches between rows for inter-row transfers.

The scheme in Fig. 2 shows how logic operations can be carried out between MTJs in the same row. However, it is important at times to perform logic operations that traverse multiple rows. To enable inter-row communication, we augment the array of Fig. 2 by inserting switches between logic lines which, when turned on, allow MTJs in different rows to communicate. It is unrealistic to allow every row to communicate with every other row, and nor is this required in typical computations.

To maintain the simplicity of the architecture, an LL in row i is connected through switches to the LLs in its two nearest adjacent rows, i.e., as illustrated in Fig. 3, the LL in row $i-2, i-1, i+1$, and $i+2$, if they exist. In performing in-memory computations, it is important to ensure that an internal data traffic bottleneck is not introduced: in fact, the best solutions will perform very local data movement. This is the reason why our CRAM architecture only allows direct movement to the two nearest adjacent rows. Data movement to more distant rows is not prohibited, but must proceed in sequential hops of one or two rows. In principle, it is possible to connect every row to every other row. However, for n rows, such a scheme would add $C(n,2) = O(n^2)$ transistors, and would also introduce significant routing overheads. In contrast, our scheme adds $2n$ transistors, and these local connections can be made quite easily. To illustrate the use of this structure, let us consider a very common operation where the output of an operation in row N must be moved to row M for the next operation. Each such operation requires the implementation of a BUFFER gate, which copies a value from one row to another. To move from row N to row M, the data can "jump" two rows at a time to reach its destination, except when the destination is one row away, where it "jumps" one row. For example, to copy a value from row 0 to row 7, for example, one could move first to row 2, then row 4, then row 6, and then finally to row 7. It is easy to see that in general the number of steps required to transfer one bit from row M to row N is $\left\lceil \frac{|M-N|}{2} \right\rceil$.

Other interconnection schemes may also be possible and could reduce the communication overhead, depending on application characteristics. The scheme described above is built on the assumption that energy considerations dictate that most inter-row communication must be local.

2.4 Peripheral Circuitry

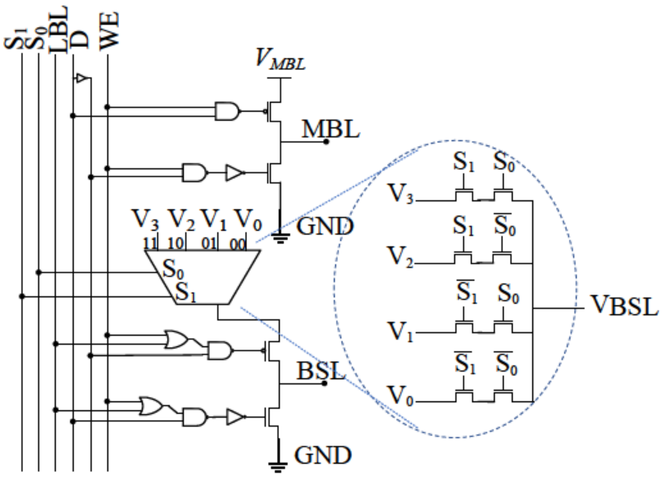


Figure 4: Bitline driver circuitry.

The voltages on BSL, LBL, and MBL are set by the bitline drivers. While LBL takes on normal V_{dd} values, required to turn on the access transistor, the chosen voltage on BSL depends on the logic function being implemented. As we will show in Section 3.2, this voltage is on the order of 100s of mV in today's technologies and 10s of mV in advanced technologies. The bitline drivers are illustrated in Fig. 4: the inputs are WE, D, and LBL, and the outputs BSL and MBL are generated using the circuitry shown in Fig. 4.

The generation of the MBL and BSL signals in Fig. 2 is illustrated in Fig. 4. In *memory mode*, LBL is grounded and line D is used to control the direction of the current. In case of a write operation, WE is set to V_{dd}, and if D is also at V_{dd}, then current is injected from MBL to BSL; otherwise, if D is grounded, the current direction is reversed. For a read operation, WE is grounded and both drivers are off; in this case, MBL is separately driven and connected to the sense amplifier. In *logic mode*, WL and WE are grounded, and LBL is at V_{dd}. If D is also at V_{dd}, then the driver connects BSL to ground, while if D is grounded, then BSL is connected to V_{BSL}.

3 Designing Arithmetic Functions

3.1 Device-level Models

In evaluating the performance of the CRAM, we consider two sets of device-level models whose parameters are listed in Table 2: (i) Today's MTJ technology, corresponding to a mainstream process today, and (ii) Advanced MTJ technology, corresponding to a realistic future process. The value of the latter point is that due to the rapid evolution of MTJ technology, using only today's node is likely to be pessimistic. Moreover, by using technology projections, the evaluation on an advanced MTJ technology provides a clear picture of design issues and bottlenecks for this method. For each technology, the table provides specifics of the MTJ materials and dimensions, the TMR, the resistance-area (RA) product, the critical switching current density, J_c, the critical switching current, I_c, the write time, t_{wr}, as well as the MTJ resistance in each state.

Table 2: MTJ Specifications

Parameters	Today's MTJ	Advanced MTJ
MTJ type	Interfacial PMTJ	Interfacial PMTJ
Material system	CoFeB/MgO/ CoFeB	CoFeB (SAF)/MgO/ CoFeB
MTJ diameter	45nm	10nm
TMR	133% [14]	500%
RA product	$5\Omega\mu\text{m}^2$	$1\Omega\mu\text{m}^2$ [15]
J_c	$3.1 \times 10^6\text{A/cm}^2$	10^6A/cm^2
I_c	$50\mu\text{A}$	$0.79\mu\text{A}$
t_{wr}	3ns [16]	1ns [14]
R_P	$3.15\text{K}\Omega$	$12.73\text{K}\Omega$
R_{AP}	$7.34\text{K}\Omega$	$76.39\text{K}\Omega$

In general, for the CRAM application, a higher TMR is beneficial since this helps differentiate between the 0 and 1 states more effectively. To select the parameters for the Advanced MTJ technology, we consider various roadmaps and projections, as well as consultations with technology experts. Today, the largest demonstrated TMR is 604% at room temperature for an MTJ built using a material stack of CoFeB/MgO/CoFeB [9], [17]. However, this MTJ uses a thick MgO layer, which results in a large RA product; moreover, it uses in-plane magnetization, which requires larger area due to its need for shape anisotropy in the plane, and is less preferred over perpendicular MTJ magnetization. While the best TMR for perpendicular MTJs in the lab today is about 208% [18], there are pathways to much higher TMRs. Accordingly, we set the TMR for the Advanced MTJ to 500%. This is further supported by predictions that show that a TMR of 1000% at room temperature will be attainable by 2024 [9]. The RA product can be tuned using new tunneling barrier materials (e.g., MgAlO), or reducing the MgO thickness while maintaining crystallinity.

3.2 Gate-level Design

When MTJs are connected together in logic mode, the type of gate functionality that results from the connection can be controlled by two factors: (i) the voltage applied on the BSL lines, which appears across the connected MTJ devices, and (ii) the logic value to which the output MTJ is preset. The corresponding bias voltage range and the preset value to implement each gate are summarized in Table 3. The output preset for each gate is unique and depends on the gate type rather than on MTJ technology parameters.

Consider the case where the configuration in Fig. 2 is used to implement a NAND gate. Since R_{AP} = (TMR+1)R_P, and a logic 0 corresponds to R_P, from Eq. (2), we have:

$$\begin{aligned} I_{00} &= V_{BSL} \Big/ \left(\frac{R_P}{2} + R_o \right) \\ I_{01} = I_{10} &= V_{BSL} \Big/ \left(\left(\frac{\text{TMR}+1}{\text{TMR}+2} \right) R_P + R_o \right) \\ I_{11} &= V_{BSL} \Big/ \left(\frac{(\text{TMR}+1)R_P}{2} + R_o \right) \end{aligned} \tag{3}$$

The requirements for the NAND gate is that the first two cases should result in logic 1 at the output MTJ, but the last case should keep the output value at logic 0. Using the fact that TMR > 0, it is easy to verify that the current monotonically decreases as $I_{00} > I_{01} = I_{10} > I_{11}$. Therefore, if the output is preset to logic 0, then by an appropriate choice of V_{BSL}, the first two cases can result in a current that exceeds I_c, thus switching the output while the last can result in a current below I_c, keeping the output at logic 0. A similar argument can be made to show that if the output is preset to 1, the gate will not function correctly because it requires the first two cases (higher currents) not to induce switching, while the last case (lowest current) must induce switching. The same arguments can be used to argue that an AND implementation should be preset to logic 1, allowing switching in the 00 and 01/10 cases, but not the 11 case.

It can further be seen that an XOR cannot be naturally implemented in a single gate under this scheme: depending on the preset value, it requires switching for the 00 and 11 cases but not 01/10, or vice versa. Neither case follows the trends by which the current I increases. Therefore, like CMOS, an XOR must be implemented using multiple stages of logic.

For the NAND gate, for a preset output value of 0, $R_o = R_P$. Therefore the results for the three cases are:

$$\begin{aligned} I_{00} &= \frac{V_{BSL}}{R_P} \left(\frac{2}{3} \right) \\ I_{10} = I_{01} &= \frac{V_{BSL}}{R_P} \left(\frac{\text{TMR}+2}{2\text{TMR}+3} \right) \\ I_{11} &= \frac{V_{BSL}}{R_P} \left(\frac{2}{\text{TMR}+3} \right) \end{aligned} \tag{4}$$

The requirements for the NAND gate is that the first two cases should induce switching to logic 1, but the last case should keep the output value at logic 0. Therefore,

$$\begin{aligned} &\left(\frac{\text{TMR}+2}{2\text{TMR}+3} \right) \frac{V_{BSL}}{R_P} > I_c > \left(\frac{2}{2\text{TMR}+3} \right) \frac{V_{BSL}}{R_P}, \\ \text{i.e., } &\left(\frac{2\text{TMR}+3}{\text{TMR}+2} \right) I_c R_p < V_{BSL} < \left(\frac{2\text{TMR}+3}{\text{TMR}+2} \right) I_c R_P. \end{aligned} \tag{5}$$

From the values of R_P, TMR, and I_c provided in Table 2 and the requirement that the 00 and 10/11 cases should switch, while the 11 case should not, we can obtain the values in Table 3. For NAND gate, 270.0mV $< V_{BSL} <$ 354.5mV for today's MTJs, and 18.6mV $< V_{BSL} <$ 40.2mV for advanced MTJs. Similar methods are used for other gates. From the table, it can be seen that the voltage V_{BSL} required to implement each gate type using today's MTJ

Table 3: Bias Voltage Ranges and Output Preset Values

Gate	Bias voltage range		Output preset
	Today's MTJ	Advanced MTJ	
NOT	315.0 – 551.5mV	20.1 – 70.4mV	0
BUFFER	551.5 – 788.0mV	70.4 – 120.6mV	1
AND	506.5 – 591.0mV	68.9 – 90.5mV	1
NAND	270.0 – 354.5mV	18.6 – 40.2mV	0
OR	472.7 – 506.5mV	65.3 – 68.9mV	1
NOR	236.2 – 270.0mV	15.0 – 18.6mV	0
MAJ3	459.6 – 481.5mV	64.9 – 67.8mV	1
$\overline{\text{MAJ3}}$	223.1 – 245.0mV	14.6 – 17.5mV	0
MAJ5	435.4 – 443.2mV	63.3 – 64.3mV	1
$\overline{\text{MAJ5}}$	198.9 – 206.7mV	13.0 – 14.0mV	0

technology is higher than that for the advanced MT nology.

For a NAND gate, the lower end and upper of t voltage (V_{BSL}) is shown in Eq. (5). For each gate typ denote the lower and upper ends of the bias voltag as V_{min} and V_{max}, respectively, then we can define th margin, NM, as:

$$NM = (V_{max} - V_{min})/V_{mid}$$
$$\text{where } V_{mid} = (V_{max} + V_{min})/2$$

This metric provides a measure of the range of the swings, normalized to the mean.

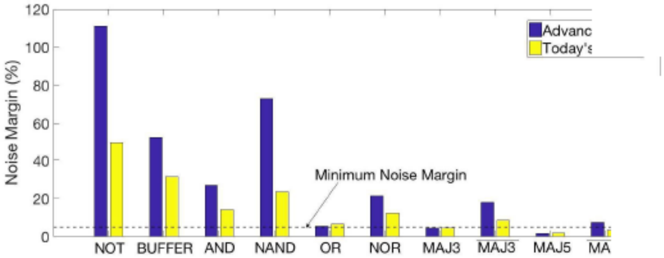
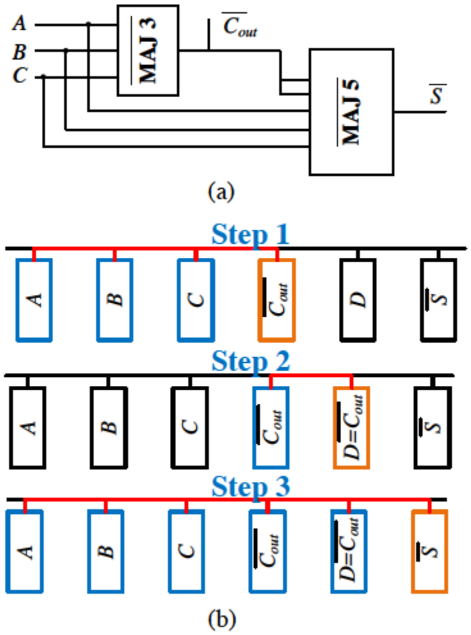


Figure 5: A comparison of the noise margin for various types, implemented in a CRAM today's MTJs and adva MTJs, as defined in Table 2.

Fig. 5 shows the noise margin for various gate t It can be seen that for Advanced MTJs, the noise mar most cases is about 2X larger than those of today's MTJ. noise margin is intended to capture the level of resilie each gate type to shifts in parameter values due to pr variations, supply voltage variations, thermal noise While it is difficult to know the level of such drifts, these technologies are still evolving and are in a high of flux. In this work, we choose a threshold for the mini acceptable noise margin in this work as $NM = 5\%$. the figure, it can be seen that the MAJ3, MAJ5, an gates for both technologies and the $\overline{\text{MAJ5}}$ gate for to technology fall below this threshold, and are not used

3.3 Functional-level Design

In [5], NAND based logic was applied to implement a full adder (FA) using CRAM. A NAND-based implementation of FA requires 9 stages of logic. As shown in [5], Carry and Sum can, respectively, be generated after 9 and 8 CRAM computation steps. This large number of sequenced operations for an addition can can incur high delays. FAs can be implemented more efficiently using majority logic [19] instead of NAND gates. While such implementations can reduce the number of steps required to implement a FA in the CRAM, MAJ3 and MAJ5 have low NM values (Fig. 5), but $\overline{\text{MAJ3}}$ and $\overline{\text{MAJ5}}$ gates have sufficient NM for advanced MTJs. Therefore, we adapt the MAJ-based FA designs to use complementary MAJ logic for advanced MTJs, and stay with NAND gates for today's MTJs.



(a)

(b)

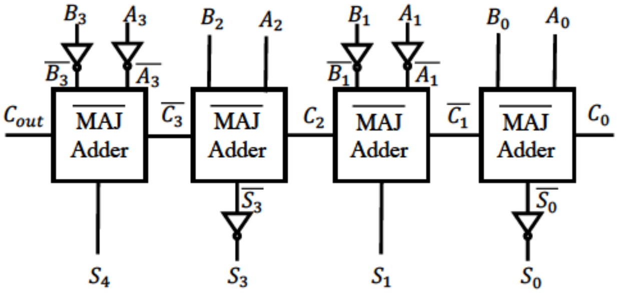


Figure 7: 4-bit ripple carry adder using bubble-pushing.

We propose a modification of the MAJ-based FA using the $\overline{\text{MAJ}}$-based logic structure shown in Fig. 6(a) to implement the complement of a FA. It can easily be verified that this correctly produces the outputs $\overline{S}$ and $\overline{C_{out}}$ based on input bits A, B, and C. We will defer the precise set of scheduling operations to our discussion in Section 4.

To demonstrate how this complemented FA can be used to build an n-bit adder, we show a 4-bit ripple carry adder in Fig. 7. The LSB (zeroth bit) uses the logic in Fig. 6 to generate the complemented output carry, which is the complemented input carry $\overline{C_1}$ of the first bit, and to generate the complemented sum bit $\overline{S_0}$. The latter is taken through an inverter to generate S0. Instead of inverting $\overline{C_1}$, we use "bubble-pushing" to implement the first bit, based on the observation that:

$$C_{out} = \overline{\text{MAJ3}}(\overline{A}, \overline{B}, \overline{C}) \quad (8)$$

$$S = \overline{\text{MAJ5}}(\overline{A}, \overline{B}, \overline{C}, C_{out}, C_{out}) \quad (9)$$

Thus, we invert A_1 and B_1, which are not on the critical path, instead of inverting $\overline{C_1}$, to generate S_1 and C_2, and so on. In general, for an n-bit adder, alternate bits use true and complemented inputs to the $\overline{\text{MAJ}}$-based FA. In this proposed scheme inversions are not required for any C_{out} bits (except for the MSB for an n-bit adder where n is odd – but it is unusual for n to be odd in real applications). Explicit inversions (i.e., NOT gates) are only required for the Sum outputs of alternate FAs in the n-bit adder.

4 Scheduling CRAM Operations

Scheduling an n-bit addition on the CRAM: We begin with the implementation of single-bit addition in the CRAM and then move to multibit additions. The FA structure involves multiple steps that implement $\overline{\text{MAJ3}}$, $\overline{\text{MAJ5}}$, NOT, and BUFFER, and these computational steps are shown in Fig. 6(b). For each step, it is assumed that initializations (output presets) are performed before the shown computational steps.

Step 1 For the FAs corresponding to odd-numbered bits in n-bit addition, the input is not complemented. In Step 1, we initialize the Cout cell to 0, and then compute $\overline{C_{out}} \leftarrow \overline{\text{MAJ3}}(A, B, C)$ by activating the BLL transistor, after initializing the Cout cell to 0.

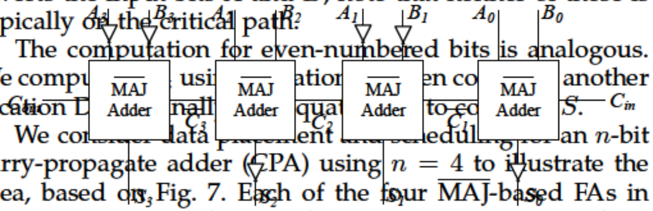
Step 2 We copy the computed $\overline{C_{out}}$ using $D \leftarrow \text{BUFFER}(\overline{C_{out}})$. The register D is used to store the value of $\overline{C_{out}}$, as two $\overline{C_{out}}$ operands are required for the next step.

Step 3 We compute $\overline{S} \leftarrow \overline{\text{MAJ5}}(A, B, C, \overline{C_{out}}, \overline{C_{out}})$.

In principle, this would have to be followed by Steps 4 and 5 (not shown in the figure), which use the NOT function to obtain the uncomplemented S and C_{out} outputs. However, bubble-pushing makes it unnecessary to invert a rippled carry output, and alternate output bits need no inversion on the sum bits, but need input inversions. However, for odd-numbered FAs, we require a Step 4 to invert the Sum output, and for even-numbered bits, we add a "Step 0" that inverts the input bits A and B; note that neither of these is typically on the critical path.

The computation for even-numbered bits is analogous. We compu[illegible] usi[illegible]ation[illegible]en co[illegible] another location D[illegible]nall[illegible]quat[illegible] to co[illegible] S.

We consider data placement and scheduling for an n-bit carry-propagate adder (CPA) using $n = 4$ to illustrate the idea, based on Fig. 7. Each of the four $\overline{\text{MAJ}}$-based FAs in this structure is implemented within a separate row of the CRAM, and the computation in each row is performed in separate steps that capture data dependencies.



A_3
B_3
A_2
B_2
A_1
B_1
A_0
B_0
$\overline{\text{MAJ}}$ Adder
$\overline{\text{MAJ}}$ Adder
$\overline{\text{MAJ}}$ Adder
$\overline{\text{MAJ}}$ Adder
C_{out}
C_3
C_2
C_1
C_{in}
S_3
S_2
S_1
S_0

Figure 5: 4-bit ripple carry adder using the bubble-pushing technique. In each $\overline{MAJ}$ FA complement of sum and carry are achieved after 3 steps.

Time	1	2	3	4	5	6	7	8	9
Row 0	$\overline{C_1}$	–	D_0	$\overline{S_0}$	S_0				
Row 1	–	$\overline{C_1}$	C_2	–	D_1	S_1			
Row 2	–	–	–	C_2	$\overline{C_3}$		D_2	$\overline{S_2}$	S_2
Row 3	–	–	–	–	–	$\overline{C_3}$	C_{out}	D_3	S_3

Table III: Scheduling table for the 4-bit CPA from $t = 1$ to 9.

Figure 8: Scheduling table for the 4-bit CPA from $t = 1$ to 9.

The scheduling table of the 4-bit ripple carry adder is shown in Fig. 8, where the i^{th} bit-slice maps to CRAM row i. Once a carry in row i is generated, it is transferred to row $i+1$. Thus, at $t = 1$, $\overline{C_1}$ is generated and is transferred to row 1 at $t = 2$; at $t = 3$, C2 is generated in row 1 and transferred to row 2 at $t = 4$, and at $t = 5$, $\overline{C_3}$ is generated and transferred to row 3 at $t = 6$. Now that all inputs to the MSB are available, using the schedule described in Fig. 6(b), three time units later, at $t = 9$, the computation is complete.

The computational steps are shown in Fig. 4(b). For odd-numbered bits, where the input is not complemented, in Step 1, we compute $\overline{C_{out}} \leftarrow \overline{\text{MAJ3}}(A, B, C)$ by activating the BLL transistor, after initializing the Cout cell to 0. Next, in Step 2, we copy this computed value to its adjacent cell by performing $D \leftarrow \text{BUFFER}(\overline{C_{out}})$.

Multiplication: The dot notation is a useful tool to represent arithmetic operations [20]. The notation is illustrated in Fig. 9 for the addition and multiplication of two 4-bit binary digits. Each dot represents a place significance, and dots representing each input number correspond to its four bits, with weights of 1, 2, 4, and 8, from right to left. Fig. 9(a) shows that the sum of these two 4-bit numbers is a 5-bit number. The multiplication of two 4-bit numbers (Fig. 9(b)), generates a set of four shifted partial products that are added to generate the 8-bit product.

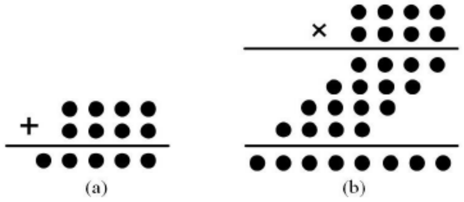
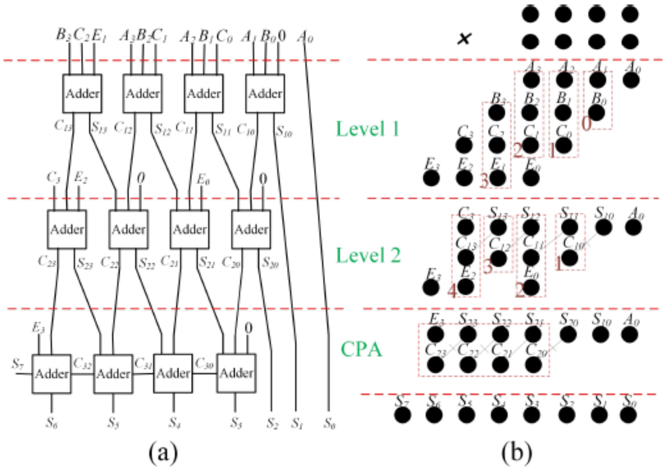


Figure 9: Dot notation representation [20]: (a) Addition of two 4-bit digits, (b) Multiplication of two 4-bit digits.



$B_3 C_2 E_1$
$A_3 B_2 C_1$
$A_2 B_1 C_0$
$A_1 B_0 0$
A_0
Adder
C_{13} S_{13} C_{12} S_{12} C_{11} S_{11} C_{10} S_{10}
C_3 E_2 0 E_0 0
C_{23} S_{23} C_{22} S_{22} C_{21} S_{21} C_{20} S_{20}
E_3
S_7
C_{32} C_{31} C_{30}
0
S_6 S_5 S_4 S_3 S_2 S_1 S_0
Level 1
Level 2
CPA
×
A_3 A_2 A_1 A_0
B_2 B_1 B_0
C_3 C_2 C_1 C_0
E_3 E_2 E_1 E_0
0
1
2
3
S_{13} S_{12} S_{11} S_{10} A_0
C_{13} C_{12} C_{11} C_{10}
4
E_3 S_{23} S_{22} S_{21} S_{20} S_{10} A_0
C_{23} C_{22} C_{21} C_{20}
S_7 S_6 S_5 S_4 S_3 S_2 S_1 S_0
(a)
(b)

Figure 10: (a) Schematic and (b) dot notation representation for a 4×4 Wallace tree multiplier.

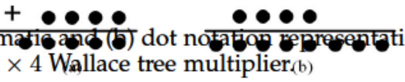
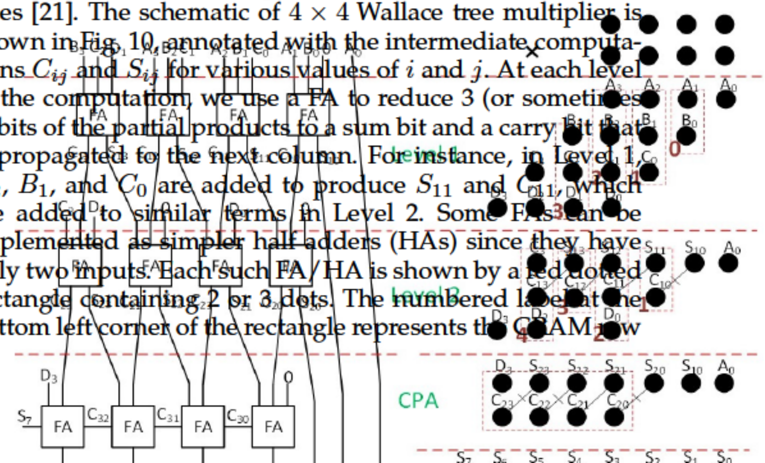


Figure 6: Dot notation representation [16]: (a) Addition of two 4-bit digits, (b) Multiplication of two 4-bit digits.

Breaking down the product computation further by mapping it to FA operations, a fast method for adding the partial products of a multiplication is to use Wallace/Dadda trees [21]. The schematic of 4×4 Wallace tree multiplier is shown in Fig. 10, annotated with the intermediate computations C_{ij} and S_{ij} for various values of i and j. At each level of the computation, we use a FA to reduce 3 (or sometimes 2) bits of the partial products to a sum bit and a carry bit that is propagated to the next column. For instance, in Level 1, A_2, B_1, and C_0 are added to produce S_{11} and C_{11}, which are added to similar terms in Level 2. Some FAs can be implemented as simpler half adders (HAs) since they have only two inputs. Each such FA/HA is shown by a red dotted rectangle containing 2 or 3 dots. The numbered label at the bottom left corner of the rectangle represents the CRAM row



FA
Level 1
Level 2
CPA
×
A_3 A_2 A_1 A_0
B_1 B_0
C_0
0
1
2
3
4
D_3 D_2 D_0
S_{11} S_{10} A_0
C_{13} C_{12} C_{11} C_{10}
D_3 S_{23} S_{22} S_{21} S_{20} S_{10} A_0
C_{23} C_{22} C_{21} C_{20}
S_7
C_{32} C_{31} C_{30}
0
S_7 S_6 S_5 S_4 S_3 S_2 S_1 S_0

is transferred to the next higher row).

Another view of the scheduling of these computations is presented in Table IV. The implementation of each level requires 5 steps, and computations related to FAs within each level are performed in parallel. As in the case of the ripple carry adder, the first three steps for each FA at Level 1 involve computing the complement of the output carry, cloning the computed carry complement to another cell, and then computing the complement of the sum at $t = 1, 2, 3$, respectively. As before, bubble-pushing is used so that the inverted

number that implements the FA operation. It can be seen that each column of the computation, which corresponds to a place significance, maps to the same CRAM row in each Level (e.g., the bits that compute carry in Level 1 that adds A_2, B_1, and C maps to CRAM row 2. The resultant sum S_{11} remains in that row and is added with other operands in Level 2, while the carry-out goes to the next row).

	Level 1			Transfer		Level 2			CPA
Time	1	2	3	4	5	6	7	8	
Row 0	$\overline{C_{10}}$	D_1	$\overline{S_{10}}$						CPA
Row 1	$\overline{C_{11}}$	D_2	$\overline{S_{11}}$	$\overline{C_{10}}$		C_{20}	D_5	S_{20}	
Row 2	$\overline{C_{12}}$	D_3	$\overline{S_{12}}$		$\overline{C_{11}}$	C_{21}	D_6	S_{21}	
Row 3	$\overline{C_{13}}$	D_4	$\overline{S_{13}}$	$\overline{C_{12}}$		C_{22}	D_7	S_{22}	
Row 4					$\overline{C_{13}}$	C_{23}	D_9	S_{23}	

Table IV: Scheduling table for the Wallace tree adder.

Figure 11: Scheduling table for the Wallace tree adder.

Another view of the scheduling of these computations is presented in Fig. 11. The implementation of each level requires 5 steps, and computations related to FAs within each level are performed in parallel. As in the case of the ripple carry adder, the first three steps for each FA at Level 1 involve computing the complement of the output carry, cloning the computed carry complement to another cell, and then computing the complement of the sum at $t = 1, 2, 3$, respectively. As before, bubble-pushing allows the inverted sum and carry outputs to be used directly in the next bit slice.

To begin the computations at Level 2, the computed carry values in row i must be sent to row $i + 1$, and this is accomplished at $t = 4, 5$. Note that due to the structure of the CRAM, this must be performed in two steps: when row i is connected to $i + 1$, we cannot simultaneously connect row $i + 1$ to $i + 2$, otherwise we create an inadvertent path from i to $i + 2$. Therefore, transfers from all even-numbered rows to the next row occur in one time slot, and transfers from all odd-numbered rows in another. Three more steps are required to perform the FA computation at Level 2, which completes at $t = 8$, and the results then go to a CPA, implemented as in Section 4.

5 CRAM Applications

In this section, we present two applications of the CRAM: a two-dimensional (2D) convolution operation for image filtering using images and filters represented by multiple bits, and a binary neuromorphic inference engine for digit recognition.

5.1 2D Convolution for Image Filtering

Convolution is a building block of many image processing applications, such as image filtering for sharpening and blurring. Fig. 12(a) shows an input image with 512×512 pixels that is convolved by a 3×3 filter to yield an output image with the same number of pixels, shown in Fig. 12(b). The output pixel in location (i, j) is computed as follows, as illustrated in Fig. 13(a):

$$O_{i,j} = \sum_{k=1}^{3} \sum_{l=1}^{3} f_{k,l} \cdot I_{i-k+2,j-l+2} \qquad (10)$$

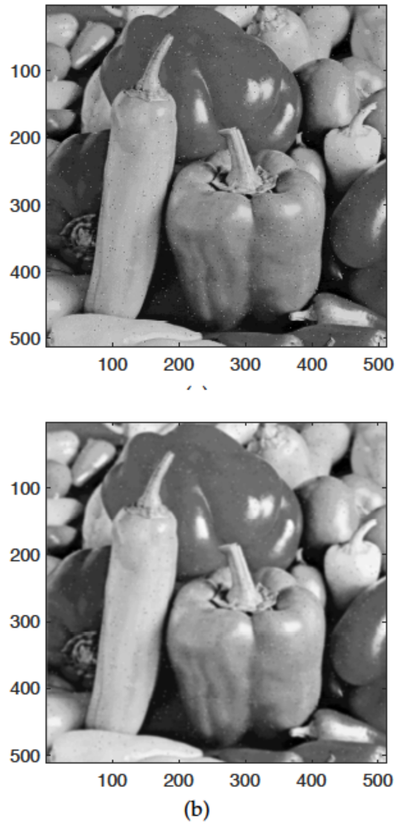


Figure 12: Using the average (mean) filter to denoise an image: (a) the noisy image with numerous specks, and (b) the denoised version.

where f represents a matrix associated with the 3×3 filter, and I is the matrix of input pixels. The image I is represented using 4 bits, and the filter f uses two bits. Thus, each partial product $(f_{k,l} \cdot I_{i-k+2,j-l+2})$ has six bits.

To compute $O(i, j)$, the result of the dot product representing a pixel of the output image, nine six-bit partial products are added together using a tree adder, as shown in Fig. 13(b). The tree adder has four levels, and uses a six-bit ripple carry adder at the final stage. As illustrated in the figure, the total number of rows required for the implementation of one dot product is 19. Similar to the adder and multiplier, it is easy to build a scheduling table for the implementation of the dot product: for conciseness, it is not shown here. The total number of steps for the implementation includes all steps for multiplication, additions within the rows, inter-row transfers, and the final CPA. The convolution for each output pixel can be computed in parallel.

5.2 A Neural Inference Engine

Using the building blocks described above, we show how the CRAM can be used to implement a neuromorphic inference engine for handwritten digit recognition using data from the MNIST database [22]. The neural network architecture from [23] (Fig. 14) is used to implement the recognition scheme. Each of the MNIST images is scaled to 11×11 as in [23], a transformation that maintains 91% recognition accuracy and reduces computation. Note that using the full image, or using a more complex neural engine with higher recognition accuracy, does not fundamentally change

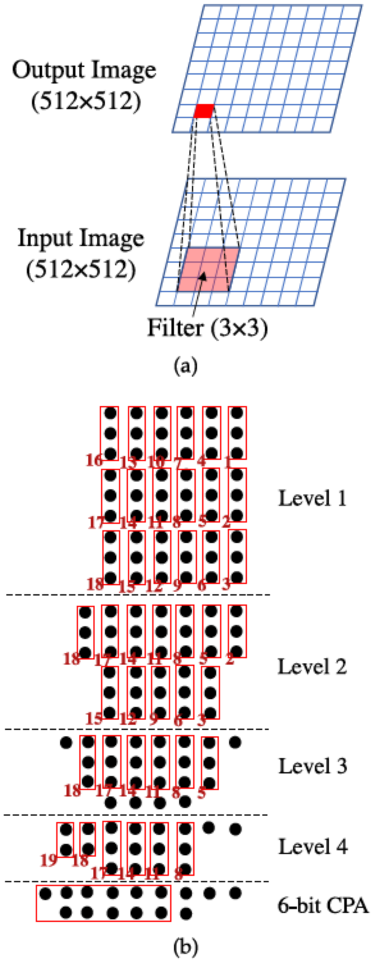


Figure 13: The implementation of convolution using CRAM: (a) a 512×512 image, with 4 bit per pixel, is filtered using a 3×3 filter, with 2 bits per word, and (b) the addition of nine six-bit partial products to compute the dot product that evaluates the output image pixel using a 4-level tree adder.

our approach or conclusions. This data is provided to the neural net with a set of one-bit inputs, $X_i, 1 \leq i \leq 121$, corresponding to each image bit, and fires one of 10 outputs, corresponding to the digits 0 through 9. In principle, this network can also be implemented using a larger number of bits, as in the previous application, rather than as a binary neural network; the unit operations for n bits have been described in Section 4. As in [23], the synaptic weights W_{ij} have three bits and are trained using supervised learning.

The outputs of the neural network are computed as the inner product:

$$Y_i = \sum_{j=1}^{121} W_{i,j} X_j \tag{11}$$

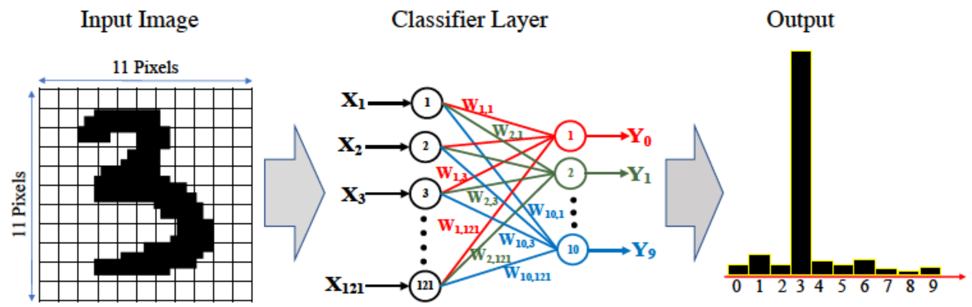
The inner product Y_i is computed as a sum of 121 partial products. Fig 15(a) uses the dot notation to represent the implementation of Y_i. Each partial product is a bitwise multiplication of a three-bit $W_{i,j}$ with a one-bit X_j, and this can be implemented using a bitwise AND operation. The resulting three-bit partial products, shown in each row at Level 1 in Fig. 15(a), are added together using a Wallace tree adder. Note that one can also use a Dadda tree adder or any other similar tree adder without changing the overall delay significantly. This is because the overall delay does not depend on the number of FAs in a level, as all FAs within a level act in parallel. As long as the number of tree levels (and the length of final ripple carry adder) is the same, the overall delay is quite similar.

Recall that each group of three dots in the figure is a FA computation performed in a row, after which the sum and carry are propagated to the next level of computation. In this case, the computation requires 9 levels. The choice of row assignments for the FA computations is critical in maintaining latency. In principle, it is possible to assign rows sequentially by column, e.g., the dots in the last column are assigned to rows 1 through $\lceil 121/3 \rceil$, then the second last column gets row $\lceil 121/3 \rceil + 1$ onwards, and so on. However, this would mean large communication latencies (e.g., the carry from row 1 may have to be sent to row 42 or higher). We use an approach that performs a zigzag assignment of rows, as shown in Fig. 15(a). After the computation in each row, the sum output is sent to the median row and the carry output to the largest row number. For example, the first three FAs in the right column are in rows 1, 2, and 4, according to the diagonal pattern. Their three sums are sent to the median row, row 2, and their carries are sent to the maximum row number, row 4. At Level 2, the same process repeats: the FAs in the first three groups, now in rows 2, 10, and 19, send their three sums to row 10 and three carrys to row 19, and so on. The Wallace tree has 9 levels in all, followed by a CPA stage.

Fig. 15(b) shows the footprint of computations, where the y-axis is the row number and the x-axis is time. Each colored block is an adder computation, which takes three steps if we use majority complementary logic (for advanced MTJ technology) or nine steps for NAND-based logic (using today's MTJ technology). As described above, once a computation in one level is complete, we transfer the sums to the median row number (as shown by blue arrows) and carrys to the largest row number (as shown by red arrows). For example, after Level 1, the Sum outputs for Rows 1 and 4 are transferred to Row 2, to set up the Level 2 computation that adds these to the Sum output produced in Row 2. Such inter-row transfers correspond to BUFFER operations that are carried out by activating the switches described in Section 2.3.

The span of rows involved in the computation shrinks from level to level. Fig. 16 shows the number of FA computations at each level of the Wallace tree and the number of inter-row data transfers involved before the beginning of the full adder computation at each level. Due to the scheme for moving sums and carrys across rows, as the computation proceeds, the span of rows that contain active data shrinks. For example, Level 1 involves all 120 rows, but fewer rows are involved at Level 2, starting from Row 2; at Level 3, the number reduces further, and the first active row is Row 5.

Our approach is shown on a binary neural network, a family of structures that has recently attracted great attention for low-energy deep learning on embedded platforms. However, the general concept and procedure for implementing our design can be applied to other neural inference engines, including multibit neural networks. In general, when the number of bits per pixel (for the same application) increases, the computation will employ unit operations with a greater number of bits (e.g., a tree adder with more levels and more FAs). This increases the number of steps and the size of the CRAM array for implementation. The fundamental operation in many neural computation models is a convolution of the form Equation (10) or a dot product of



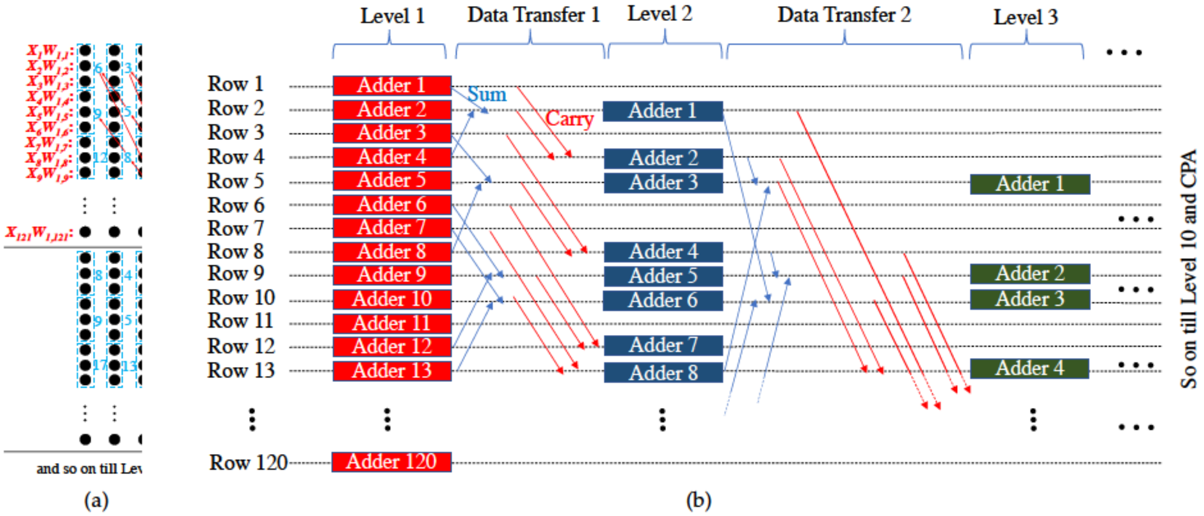


Figure 15: The implementation of each of the ten outputs, Y_i, of the inference engine, illustrating the (a) zigzag scheme for assigning addition operations to CRAM rows, and (b) the inter-row data transfers and the computation footprint of Y_i along the rows of the CRAM.

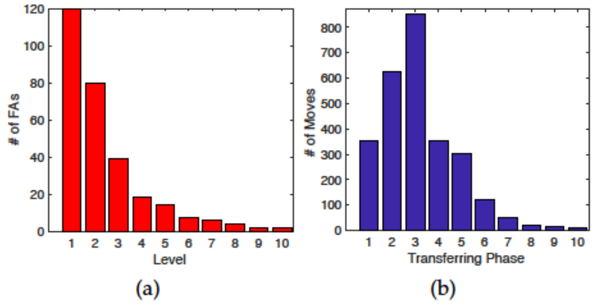


Figure 16: (a) The distributions of the number of FAs in each level of Wallace tree. (b) The distribution of the total number of moves required in the data transfer phases.

the form of Equation (11). As shown in Section 3, the CRAM architecture can perform the unit operations (addition and multiplication) for either. For example, the convolution layer in a convolutional neural network (CNN) involves dot product operations, and then a summation over the results of these dot products. Computations in other CNN layers, such as pooling and ReLU, also require simple arithmetic or Boolean operations that can be implemented on the CRAM substrate.

6 Evaluation and Results

We evaluate the performance of the CRAM for two applications: (a) performing 2D convolution to filter a 512×512 image, and (b) digit recognition, used to analyze 10,000 handwritten digit images from the MNIST database. In both applications, the execution time and energy of the CRAM are compared with those of a near-memory processing (NMP) system, where a processor is available at the edge of the memory array. We do not explicitly show comparisons between NMP and processor-based computing, where the data is taken from memory to a processor or coprocessor for computation, and the results are transported: it is well-documented [2], [4], [24] that this method is vastly inferior to the NMP approach due to the communication bottleneck described in Section 1. For example, [24] reports a 6.5× improvement through the use of NMP, as compared to processor-based computing. Note that this communication overhead limits the effectiveness of any processor or coprocessor that requires communication to and from memory, including specialized accelerator cores (e.g., neuromorphic units or GPUs).

The organization of the CRAM array is shown in Fig. 17. For the 2D convolution application, a 256Mb [512Mb] CRAM array is enough to compute all output pixels of a 512×512 image with 4 bits per pixel in parallel using the advanced [today's] MTJ device. For the digit recognition application, we require a 1Gb memory, where each image can be processed in parallel within a subarray. The overall

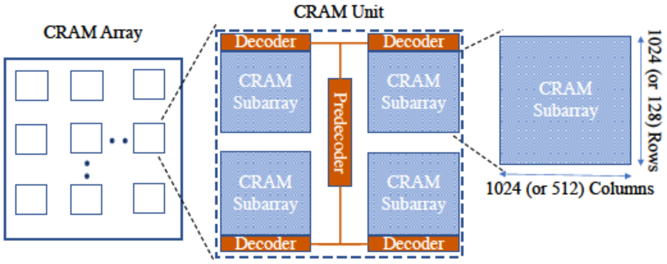


Figure 17: Each CRAM unit includes four CRAM subarrays and one predecoder. A predecoder block is at the center of the CRAM unit, and fans out to four CRAM column decoders.

array is divided into subarrays as shown in the figure. The operations in the CRAM array are scheduled by activating the appropriate LBL and BSL lines. In memory mode, the predecoder and decoder modules drive the selection of WL (see Fig. 1), while in logic mode, they drive the selection of LBL. The predecoder at the center of the CRAM unit fans out to a set of decoders in our evaluations: here, we show four decoders, but if a larger number of subarrays is used, this number can be different.

To calculate the energy and delay of the CRAM system, we considered the energy and delay components in both peripheral circuitry and the CRAM array. To determine the impact of the size of CRAM on execution time and energy, we considered two cases for the size of CRAM subarrays: 1024 rows × 1024 columns, and 128 rows × 512 columns.

6.1 Execution Time

CRAM: We assume that the data is placed in the appropriate CRAM array location. The execution time, t_{CRAM}, is:

$$t_{CRAM} = t_{MTJ} + t_{Dr}, \tag{12}$$

where t_{MTJ} and t_{Dr} are delay related to computations in the MTJ array and in the bitline drivers of Fig. 4, respectively. The total array delay is dominated by the MTJ delay,

$$t_{MTJ} = N_{step} t_{wr}, \tag{13}$$

where N_{step} and t_{wr} are, respectively, the number of computation steps and the MTJ write time per computation. Here,

$$N_{step} = N_{Mul} + N_L N_{FA} + \textstyle\sum_{i=1}^{N_L} I_{i \longrightarrow i+1} + t_{CPA} \tag{14}$$

where N_{Mul} is the number of steps required to generate partial products for the first level of the tree adder. The second term indicates total number of intrarow computation steps required for the implementation of the neural network, where N_L the number of levels in the implementation tree adder, and N_{FA} the number of steps for the implementation of a FA. The third term corresponds to the total number of steps for transferring data between rows of the CRAM array: at each level i of the tree, the number of such transfers is denoted by $I_{i \longrightarrow i+1}$. Finally, t_{CPA} is the time required for the carry propagation addition step at the end of the Wallace tree computations. The preset adds no execution time overhead and can be performed either during the write operation when CRAM is in the memory mode, or online during the computation when CRAM is in the logic mode. In the latter case, the output MTJs are preset in parallel with the logic computation of the previous step, adding no overhead to the compute time. During the logic operation LBLs, and BSLs are engaged in computation, and current flows through LLs (see Fig. 1 and Fig. 2). Simultaneously, one can also write the preset value for the next step, as only MBL and BSL (of another column) are involved in the writing operation, and there is no overlap between current path related to computation and that to output preset.

For the 2D convolution application, we have:

- From Section 5.1, $N_{Mul} = 9$ and $N_L = 4$.
- Based on Section 3.3, $N_{FA} = 3$ using the $\overline{\text{MAJ}}$ gates, with bubble-pushing, in advanced MTJ technologies, and $N_{FA} = 9$ using NAND-based logic in today's technology (where $\overline{\text{MAJ}}$ gates do not provide sufficient noise margin, as shown in Section 3.2).
- We count all number of steps in the inter-row communication phases, and find that $\sum_{i=1}^{N_L} I_{i \longrightarrow i+1} = 14$.
- Extending the argument from the four-bit adder in Section 3.3, $t_{CPA} = 13$ for the six-bit adder.

From (14), we obtain $N_{step} = 48$ (for the advanced CRAM with $\overline{\text{MAJ3}}$ based logic) and $N_{step} = 72$ columns (for today's CRAM with NAND-based logic). Thus, the computation for each pixel of the output image requires an array of 19 rows (Section 5.1) and N_{step} columns. By rounding the column counts to the nearest power of 2, and considering all 512×512 pixels of the output image, a CRAM array size of 256Mb is required for the computation on the advanced MTJ; the corresponding number for today's MTJ is 512Mb.

For the digit recognition application:

- From Section 5, $N_{Mul} = 6$, and $N_L = 10$.
- As before, $N_{FA} = 3$ using the advanced CRAM, and $N_{FA} = 9$ using today's technology.
- The number of steps in the inter-row communication phases is determined to be $\sum_{i=1}^{N_L} I_{i \longrightarrow i+1} = 247$.
- From Section 3.3, $t_{CPA} = 9$ for the four-bit adder.

Therefore, $N_{step} = 292$ for the advanced MTJ technology (using $\overline{\text{MAJ}}$ logic), and $N_{step} = 352$ using today's MTJs (using NAND logic). The computation of each image requires an array of 121 rows (corresponding to the partial products) times 10 outputs, and 292 or 352 columns (corresponding to the steps in the computation), depending on the type of MTJ used. Therefore, rounding 292 (or 352) to the nearest higher power of 2, in a 1024 × 1024 memory subarray, we can fit 18 images (9 images along the rows and 2 images along the columns). The entire set of 10,000 images thus requires 10,000/18 = 556 such arrays; rounding this up to 1024, we see that we require 1024 such subarrays, providing a total memory size of 1Gb, as listed earlier.

To incorporate the delay of bitline drivers in the CRAM, an overhead delay estimated in each step by considering the delay components of a DRAM array with the same physical size. We use the parameters and models in the NVSim memory simulator [25] at 10nm and 45nm to consider a subarray of this size and use the underlying parameters to obtain these delays. We tailor the NVSIM models to the specifics of the CRAM computation. Specifically, in logic mode, each computation step requires LBLs to be driven, similar to driving wordlines, but does not require bitline or sense amplifier circuitry. The load seen by the column drivers in logic mode can be modeled in the same way as the load seen by row drivers in memory mode: instead of driving a wordline transistor as in memory mode, the LBL drives the access transistor that connects a cell to the LL. For each step of computation, we calculate the sum of the delay of the decoder and predecoder using similar models as NVSim, and this value is multiplied by N_{step} to find the total overhead corresponding to t_{Dr}. The size of the bit-cell is also altered to reflect the increased size of the CRAM bit cell over that for an STT-MRAM cell.

NMP System: The near-memory processing (NMP) system takes data from the memory to a processor and performs its computation outside the memory system. We assume that the operation is based on a DRAM structure, with better performance characteristics than a spintronic memory. For the digit recognition application, to process all 10K images of the MNIST database, 10,000 images, each of size 121 bits, must be fetched from DRAM, for computations in the near-memory processor. The delay for this scenario is estimated using CACTI [26]. A similar approach is used for the 2D convolution application.

Computing one of the outputs, Y_i, of the neural net requires 121 MAC operations. To find the processing time of each block of data, it is assumed that the processor uses instruction pipelining technique and that it can perform the multiply-accumulator (MAC) operation in one clock cycle. In case multiple processing units are available on the processor, we appropriately scale the execution time by the number of processors. We pessimistically assume maximum parallelism, where each fetched image is processed in parallel, and the level of parallelism is only limited by the data rate from memory. The clock frequency of the processor is 1GHz, but due to the assumption above, the precise computing speed of the processor does not affect the evaluation of NMP execution time.

6.2 Energy

Analogous to delay, the CRAM energy is computed as:

$$E_{CRAM} = E_{MTJ} + E_{Dr} \tag{15}$$

where E_{MTJ} and E_{Dr} are the energy related to computations in the MTJ array and for the bitline drivers, respectively. The energy in the MTJ array is given by

$$E_{MTJ} = E_{Preset} + E_{Mul} + E_{row} + E_{transfer} + E_{CPA} \tag{16}$$

in which E_{Preset} is the preset energy before the logic operation starts; E_{Mul} is the energy for multiplication to produce partial products in Level 1 of the tree adders; E_{row} is the energy for intrarow computation, and can be obtained by enumerating all FAs working in parallel in the 9 levels of the implementation trees; $E_{transfer}$ is the energy for transferring data across rows between various levels of computation, and can be obtained by enumerating all inter-row moves and multiplying the count by the energy of BUFFER gate; E_{CPA} is the energy for the implementation of the final ripple carry adders. For the advanced MTJ technology using $\overline{\text{MAJ3}}$ gates, Eq. (16) can be rewritten as follows (a similar equation can be derived for today's MTJ):

$$\begin{aligned} E_{MTJ} =& N_{pre}E_{pre} + N_{NOT}E_{NOT} + N_{BUF}E_{BUF} + \\ & N_{\overline{MAJ3}}E_{\overline{MAJ3}} + N_{\overline{MAJ5}}E_{\overline{MAJ5}} \end{aligned} \tag{17}$$

Here, E_{pre} is the energy for preset the output of one gate; E_g and N_g are the energy for the implementation of a single gate g and the number of gates of type g, $g \in$ {NOT, BUFFER, $\overline{\text{MAJ3}}$, $\overline{\text{MAJ5}}$}. Note that N_{pre} is equal to the sum of counts of all gates, as we need to preset the outputs of all gates. colorredAs an example, the energy values and counts for gates and the output preset for the digit recognition application using advanced CRAM are listed in Table 4.

The value of driver energy, E_{Dr}, for the CRAM is estimated using NVSim, using analogous analysis techniques as for the delay computation. Since multiple columns may be driven in each step, we multiply the energy cost of driving each column by N_{eff}, the average number of columns driven in any part of the computation. The energy within each CRAM unit is the sum of energy of four CRAM subarrays and one decoder. This value is multiplied by N_{step} to obtain the total overhead corresponding to E_{Dr}.

For the near memory processing system, the energy consists of two components: (i) memory access energy and (ii) computation energy. The estimated cost for accessing 256 bits of the operand from memory is estimated using [2], normalized to CACTI.

Table 4: The energy cost for various CRAM gate types and preset operations under the Advanced MTJ technology.

Gate	NOT	BUFFER	$\overline{\text{MAJ3}}$	$\overline{\text{MAJ5}}$	Preset
Energy/gate(aJ)	30.7	73.8	7.6	6.3	26.1
Count($\times 10^5$)	365	3017	657	294	4333

6.3 Comparison between CRAM and NMP

The results for execution time and energy for CRAM (at 10nm and 45nm) and NMP (at 16nm and 45nm) are evaluated for both applications (10nm data for CMOS/NMP was not available).

The evaluation result for the 2D convolution application is listed in Table 5. Based on the result, today's CRAM is 620× faster, and 23× more energy efficient than the NMP system. The advanced CRAM is 1500× faster, and 750× more efficient than a NMP system.

Table 5: Comparison between the execution time, t, and energy, E, in CRAM and NMP based computations for the 2D convolution application. The size of the CRAM subarrays in this evaluation is 128×128.

	CRAM		NMP	
	10nm	45nm	16nm	45nm
t	54.0ns	231.2ns	84.3μs	144.4μs
E	252.1nJ	16.5μJ	189.2μJ	388.6μmJ

For the digit recognition application, the results for execution time and energy for CRAM (at 10nm and 45nm) and NMP (at 16nm and 45nm) are shown in Table 6 (10nm data for CMOS/NMP was not available). The value of E_{MTJ} is 53.8μJ for today's MTJ technology, and 35.4nJ for advanced MTJs, three orders of magnitude lower. While the driver energy also reduces from 45nm to 10nm, the reduction is more modest. As a result, the energy for advanced MTJs is dominated by the driver delay.

The improvements shown in the table can be attributed to (a) high locality of the operations and (b) large amounts of parallelism as each row computes in parallel. We see that

- For the 1024×1024 subarray, the CRAM energy is about 40× better than NMP at 45nm, and improves to over 2500× lower at 10nm. The execution time is 1400× better at 45nm, and about 1700× better at 10nm.
- The execution time [energy] for the 45nm CRAM are, respectively, over 500× [20×] better than 16nm NMP.
- The 10nm CRAM execution time [energy] is over 3× [80×] better than the 45nm CRAM.
- Further improvements are seen using the smaller subarray. The energy overhead associated with smaller subarrays is small at 45nm, but is magnified at 10nm, where the driver energy dominates the subarray energy.

Table 6: Comparison between the execution time, t, and energy, E, in CRAM and NMP based computations for neuromorphic digit recognition.

	CRAM				NMP	
	1024 × 1024		128 × 512			
	10nm	45nm	10nm	45nm	16nm	45nm
t	434ns	1381ns	338ns	1105ns	0.74ms	1.96ms
E	0.49μJ	60.3μJ	0.75μJ	63.8μJ	1.27mJ	2.57 mJ

The distributions of energy and delay for the CR/ both using today's MTJs and advanced MTJs, with sul rays of 1024 rows × 1024 columns, and 128 rows × columns, are shown in Fig. 18 and Fig. 19, respectively. the 1024 × 1024 case, under today's technology, the M array in the CRAM consumes a dominant component the energy. However, for advanced MTJs, due to the grea improved energy of future MTJs, the energy bottleneck w be in the driver circuitry. By decreasing the size of subarray to 128 rows×512 columns, the total execution t decreases due to a reduction in the driver circuitry delay. a result, the execution time is dominated more strongly the MTJ array. However, the driver circuitry plays a sligl more prominent role in determining the energy than for larger subarray in Fig. 18. Thus, tradeoffs between energy and delay can be obtained by altering subarray sizes.

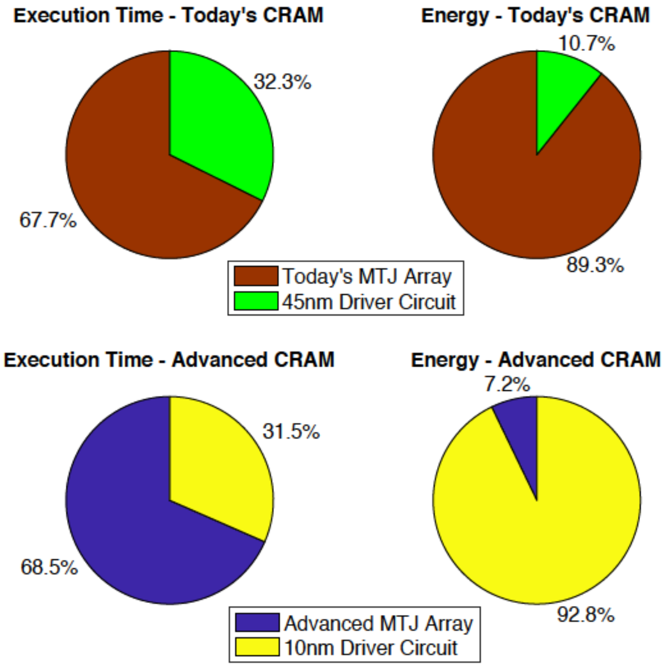


Figure 18: Distribution of energy and delay of the driver and CRAM array for CRAM with the subarray size of 1024 × 1024.

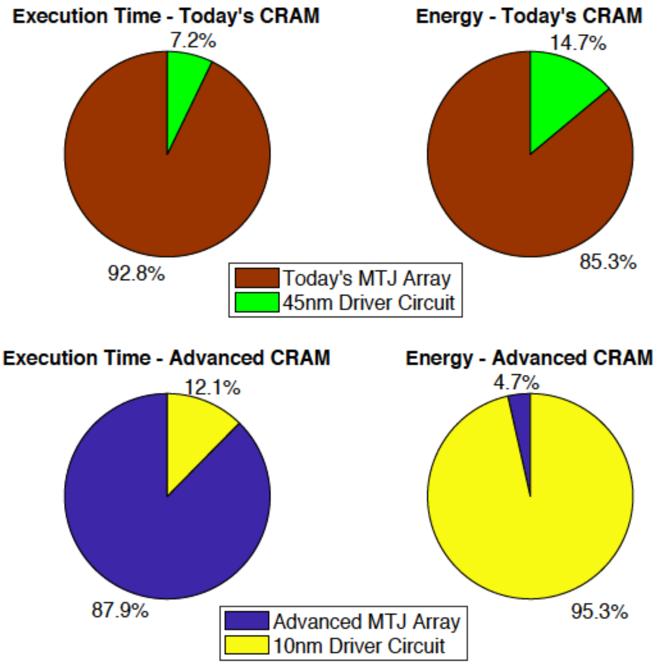


Figure 19: Distribution of energy and delay of the driver and CRAM array for CRAM with subarray size of 128 × 512.

These result clearly shows that for both applications, CRAM outperforms the NMP system in both energy and execution time. In the NMP system, it is necessary to fetch the data from memory and process it in processor units. Even with the maximum level of parallelism in NMP by using multiple processor units, and exploiting hidden latency techniques, the delay overhead of fetching data to the NMP at the edge of the memory is a major bottleneck. In contrast, the CRAM does not face this delay penalty. Moreover, the CRAM computation model enables a very high degree of parallelism as each row can perform its computations independently.

For example, in the 2D convolution application, all dot products generating output pixels can be computed in parallel. In contrast, the NMP system faces a serial bottleneck in the way that data is fetched from the memory. Moreover, the energy cost of the cost of data transfers cannot be hidden in the NMP system as data must be taken along long lines to the edge of memory. In contrast, all communication in the CRAM is inherently local within the subarray, providing large energy savings.

7 Related Work

Methods for addressing the communication bottleneck through distributed processing of data at the source have been proposed in [27], [28]. Such techniques feature a rich design space, which spans full-fledged processors [28], [29] and co-processors residing in memory [30], [31]. However, until recently, the promise of these approaches could not be translated to designs due to the incompatibility of the state-of-the-art logic and memory technologies.

This changed somewhat with the emergence of 3D-stacked architectures [32], [33], where a processor is placed next to the memory stack, has enabled the emergence of several approaches for near-memory computing [34]–[36]. However, building true in-memory computing has been difficult. In CMOS-based technologies: the computing engine is overconstrained as it must use the same technology as memory, and typically, methods that are efficient for computation may not be so for memory. As a result, techniques that attempt in-memory computation must necessarily draw the data out to the periphery of the memory array, e.g., to a sense amplifier or auxiliary computational unit, to perform the computation and then write the result back to memory as needed. There are several examples of such platforms. The work in [37] performs search operations for content-addressable memory functionalities, which need no write-back but are less general than full in-memory computing;

methods in [38] place a computational unit at the edge of memory; logic functionalities in [39] perform bitwise operations through the sense amplifier.

Post-CMOS technologies open the door to new architectures. The method in [40] presents a logic-in-memory platform that combines magnetic tunneling junctions (MTJs) with MOS transistors, and embeds computing elements within a memory array. However, this breaks the regularity of the array so that while it is efficient for computation, it may not be ideal for use as a memory module. SPINDLE [41], a spintronics-based deep learning engine proposes a tiered architecture of processing elements with a neural computing core and memory scratchpad at the edge, communicating with off-chip memory. The Pinatubo [42] processing-in-memory architecture performs bulk bitwise operations through redesigned read circuitry that performs computations at the periphery of a phase change memory array. A spintronics-based solution in [43] proposes a spin-transfer torque magnetic random access memory (STT-MRAM) approach that also performs bitwise computation at the periphery of the array by modifying the peripheral circuitry in a standard STT-MRAM module. Unlike CRAM, these methods perform computation at the edge of the memory array. Another architecture [44] builds a four-terminal domain wall device based on the spin-Hall effect, but incurs a significant area overhead. A memristor-based approach [45] shows the ability to perform logic functions in an array, but does not show larger applications.

8 Conclusion

This paper presents a detailed view of how the CRAM in-memory computation platform can be designed, optimized, and utilized. As opposed to many of the approaches proposed so far to solve the memory bottleneck by bringing processing closer to memory, CRAM implements a true in-memory computing paradigm that performs logic operations within the memory array. Methods for implementing specific logic functions have been presented and have been used to perform basic arithmetic operations, namely, adders and multipliers. At the application level, the problems of 2D convolution on multibit numbers, and an inference engine for binary neuromorphic digit recognition, have been mapped to the CRAM. An evaluation of these methods shows that for the task of evaluating the entire MNIST benchmark suite, the CRAM achieves improvements of over three orders of magnitude in the execution time. For today's MTJ technology, improvements of about 40× in the energy are seen, a figure that improves to $> 2500\times$ for future generations of MTJs.

References

[1] A. McAfee, *et al.*, "Big data: The management revolution," *Harvard Business Review*, Oct. 2012.

[2] S. W. Keckler, *et al.*, "GPUs and the future of parallel computing," *IEEE Micro*, vol. 31, pp. 7–17, 11 2011.

[3] K. Bergman, *et al.*, "Exascale computing study: Technology challenges in achieving exascale systems," *DARPA Information Processing Techniques Office (IPTO) sponsored study*, 2008. www.cse.nd.edu/Reports/2008/TR-2008-13.pdf.

[4] M. Horowitz, "Computing's energy problem (and what we can do about it)," in *Proceedings of the IEEE International Solid-State Circuits Conference*, pp. 10–14, Feb. 2014.

[5] J. P. Wang and J. D. Harms, "General structure for computational random access memory (CRAM)," Dec. 29 2015. US Patent 9,224,447 B2.

[6] X. Liu, *et al.*, "Reno: A high-efficient reconfigurable neuromorphic computing accelerator design," in *Proceedings of the ACM/ESDA/IEEE Design Automation Conference*, 2015.

[7] L. Fick, *et al.*, "Analog in-memory subthreshold deep neural network accelerator," in *Proceedings of the IEEE Custom Integrated Circuits Conference*, 2017.

[8] J. P. Wang, *et al.*, "A pathway to enable exponential scaling for the beyond-CMOS era," in *Proceedings of the ACM/ESDA/IEEE Design Automation Conference*, 2017.

[9] A. Hirohata, *et al.*, "Roadmap for emerging materials for spintronic device applications," *IEEE Transactions on Magnetics*, vol. 51, pp. 1–11, Oct. 2015.

[10] Z. Chowdhury, *et al.*, "Efficient in-memory processing using spintronics," *IEEE Computer Architecture Letters*, 2017.

[11] J. Kim, *et al.*, "Spin-based computing: Device concepts, current status, and a case study on a high-performance microprocessor," *Proceedings of the IEEE*, vol. 103, no. 1, pp. 106–130, 2015.

[12] F. Ren and D. Markovic, "True energy-performance analysis of the MTJ-based logic-in-memory architecture (1-bit full adder)," *IEEE Transactions on Electron Devices*, 2010.

[13] A. Lyle, *et al.*, "Direct communication between magnetic tunnel junctions for nonvolatile logic fanout architecture," *Applied Physics Letters*, vol. 97, no. 152504, 2010.

[14] G. Jan, *et al.*, "Demonstration of fully functional 8Mb perpendicular STT-MRAM chips with sub-5ns writing for non-volatile embedded memories," in *Proceedings of the IEEE International Symposium on VLSI Technology*, 2014.

[15] H. Maehara, *et al.*, "Tunnel magnetoresistance above 170% and resistance-area product of $1\Omega(\mu\text{m})^2$ attained by in situ annealing of ultra-thin MgO tunnel barrier," *Applied Physics Express*, vol. 4(03300), 2011.

[16] H. Noguchi, *et al.*, "3.3ns-access- time 71.2μW/MHz 1Mb embedded STT-MRAM using physically eliminated read-disturb scheme and normally-off memory architecture," in *Proceedings of the IEEE International Solid-State Circuits Conference*, 2015.

[17] S. Ikeda, *et al.*, "Tunnel magnetoresistance of 604% at 300K by suppression of Ta diffusion in CoFeB / MgO/ CoFeB pseudo-spin-valves annealed at high temperature," *Applied Physics Letters*, vol. 93, no. 8(082508), 2008.

[18] H. Almasi, *et al.*, "Effect of Mo insertion layers on the magnetoresistance and perpendicular magnetic anisotropy in Ta/CoFeB/MgO junctions," *Applied Physics Letters*, vol. 109, no. 3(032401), 2016.

[19] H. M. Martin, "Threshold logic for integrated full adder and the like," 1971. US Patent 3,609,329.

[20] L. Dadda, "Some schemes for parallel multipliers," *Alta Frequenza*, vol. 34, pp. 349–356, Mar. 1965.

[21] E. E. Swartzlander, "Recent results in merged arithmetic," in *SPIE Proceedings*, vol. 3461, pp. 576–583, 1998.

[22] Y. LeCun, "The MNIST database of handwritten digits." http://yann.lecun.com/exdb/mnist/.

[23] M. Liu, *et al.*, "A sable time-based integrate-and-fire neuromorphic core with brain-inspired leak and local lateral inhibition capabilities," in *Proceedings of the IEEE Custom Integrated Circuits Conference*, 2017.

[24] J. Jeddeloh and B. Keeth, "Hybrid memory cube new DRAM architecture increases density and performance," in *Proceedings of the IEEE International Symposium on VLSI Technology*, pp. 87–88, June 2012.

[25] X. Dong, *et al.*, "NVSim: A circuit-level performance, energy, and area model for emerging nonvolatile memory," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 31, pp. 994–1007, July 2012.

[26] N. Muralimanohar, *et al.*, "CACTI 6.0: A tool to model large caches," Tech. Rep. HPL-2009-85, HP Laboratories, 2009.

[27] H. S. Stone, "A logic-in-memory computer," *IEEE Transactions on Computers*, vol. C-19, pp. 73–78, Jan. 1970.

[28] D. Patterson, *et al.*, "A case for intelligent RAM," *IEEE Micro*, vol. 17, no. 2, pp. 34–44, 1997.

[29] S. Rixner, *et al.*, "A bandwidth-efficient architecture for media processing," in *IEEE International Symposium on Microarchitecture*, pp. 3–13, Dec. 1998.

[30] Y. Kang, *et al.*, "FlexRAM: Toward an advanced intelligent memory system," in *Proceedings of the IEEE International Conference on Computer Design*, pp. 192–201, 1999.

[31] J. B. Brockman, *et al.*, "Microservers: a new memory semantics for massively parallel computing," in *Proceedings of the 16th International Conference on Supercomputing*, 1999.

[32] J. T. Pawlowski, "Hybrid memory cube (HMC)," in *Proceedings of the IEEE HotChips Symposium*, 2011.

[33] J. Macri, "AMD's next generation GPU and high bandwidth memory architecture: FURY," in *Proceedings of the IEEE HotChips Symposium*, 2015.

[34] R. Nair, *et al.*, "Active memory cube: A processing-in-memory architecture for exascale systems," *IBM Journal of Research and Development*, vol. 59, no. 2/3, pp. 17:1–17:14, 2015.

[35] D. Zhang, *et al.*, "TOP-PIM: Throughput-oriented programmable processing in memory," in *Proceedings of the International Symposium on High-performance Parallel and Distributed Computing*, pp. 85–98, 2014.

[36] J. Ahn, *et al.*, "PIM-enabled instructions: A low-overhead, locality-aware processing-in-memory architecture," in *Proceedings of the ACM International Symposium on Computer Architecture*, pp. 336–348, June 2015.

[37] S. Jeloka, *et al.*, "A 28 nm configurable memory (TCAM/BCAM/SRAM) using push-rule 6T bit cell enabling logic-in-memory," *IEEE Journal of Solid-State Circuits*, vol. 51, pp. 1009–1021, Apr. 2016.

[38] J. Draper, *et al.*, "The architecture of the DIVA processing-in-memory chip," in *Proceedings of the 16th International Conference on Supercomputing*, pp. 14–25, 2002.

[39] V. Seshadri, *et al.*, "Ambit: In-memory accelerator for bulk bitwise operations using commodity dram technology," in *IEEE International Symposium on Microarchitecture*, 2017.

[40] A. Matsunaga, *et al.*, "MTJ-based nonvolatile logic-in-memory circuit, future prospects and issues," in *Proceedings of Design, Automation & Test in Europe*, 2009.

[41] S. G. Ramasubramanian, *et al.*, "SPINDLE: SPINtronic Deep Learning Engine for large-scale neuromorphic computing," in *Proceedings of the ACM International Symposium on Low Power Electronics and Design*, 2014.

[42] S. Li, *et al.*, "Pinatubo: a processing-in-memory architecture for bulk bitwise operations in emerging non-volatile memories," in *Proceedings of the ACM/ESDA/IEEE Design Automation Conference*, 2016.

[43] W. Kang, *et al.*, "In-memory processing paradigm for bitwise logic operations in STT-MRAM," *IEEE Transactions on Magnetics*, 2017.

[44] S. Angizi, *et al.*, "Design and evaluation of a spintronic in-memory processing platform for non-volatile data encryption," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2017. (in press).

[45] N. Talati, *et al.*, "Logic design within memristive memories using Memristor Aided loGIC (MAGIC)," *IEEE Transactions on Nanotechnology*, vol. 15, pp. 635–650, July 2016.

Masoud Zabihi received his B.Sc. and M.S. degrees in Electrical Engineering and Electronics from University of Tabriz in 2010, and Sharif University of Technology in 2013, respectively. He is currently pursuing the Ph.D. degree in Electrical Engineering at the University of Minnesota. His research interests include spintronics, emerging memory technologies, in-memory computing, and VLSI design automation.

Zamshed Iqbal Chowdhury received his B.Sc. and M.S. degrees in Applied Physics, Electronics and Communication Engineering from University of Dhaka, Bangladesh. He is a faculty member (on leave) at Jahangirnagar University, Bangladesh and currently pursuing his PhD at the Dept. of Electrical and Computer Engineering, University of Minnesota, Twin Cities, USA. His primary research interests include emerging non-volatile memory technologies, application specific hardware design, and computer performance analysis. He is a member of IEEE.

Zhengyang Zhao is currently pursuing the Ph.D. degree in Electrical and Computer Engineering at the University of Minnesota, Minneapolis, MN. He received the B.S. degree in Electrical Engineering from Xian Jiaotong University, China. His research focuses on the development of novel spintronic devices to implement energy-efficient memory cells and logic applications. His recent work includes studying current-induced magnet reversal using spin-orbit torque (SOT), as well as voltage-induced magnet reversal using piezoelectric strain or VCMA effect. More specific work includes the stack design, MTJ cell nanofabrication, advanced device characterization and physics study.

Ulya R. Karpuzcu is an associate professor at the Department of Electrical and Computer Engineering of University of Minnesota. She received the Ph.D. and M.S. degrees in Computer Engineering from University of Illinois, Urbana-Champaign. Her research interests span the impact of technology on computing, energy-efficient computing, application domain specialized architectures, approximate computing, and computing at ultra-low voltages.

Jian-Ping Wang received the Ph.D. degree from the Institute of Physics, Chinese Academy of Sciences, where he performed research on nanomagnetism, Beijing, China, in 1995. He was a Post-Doctoral Researcher with the National University of Singapore, Singapore, from 1995 to 1996. He established and managed the Magnetic Media and Materials Program with the Data Storage Institute, Singapore, from 1998 to 2002. He joined the faculty of the Electrical and Computer Engineering Department with the University of Minnesota, Minneapolis, MN, USA, in 2002 and was promoted to Full Professor in 2009. He is the Robert F. Hartmann Chair and a Distinguished McKnight University Professor of Electrical and Computer Engineering and a member of the Graduate Faculty in Physics and Chemical Engineering and Materials Science at the University of Minnesota. He is the Director of the Center for Spintronic Materials, Interfaces and Novel Architectures (C-SPIN), one of six STARnet program centers. Dr. Wang received the Information Storage Industry Consortium Technical Award in 2006 for his pioneering experimental work in exchange coupled composite magnetic media and the Outstanding Professor Award for his contribution to undergraduate teaching in 2010. His group is also known for several important experimental demonstrations and conceptual proposals, including the perpendicular spin transfer torque device, the magnetic tunnel junction-based logic device and random number generator, ultrafast switching of thermally stable MTJs, topological insulator spin pumping at room temperature, and a computation architecture in random access memory.

Sachin S. Sapatnekar (S'86, M'93, F'03) received the B. Tech. degree from the Indian Institute of Technology, Bombay, the M.S. degree from Syracuse University, and the Ph.D. degree from the University of Illinois. He taught at Iowa State University from 1992 to 1997 and has been at the University of Minnesota since 1997, where he holds the Distinguished McKnight University Professorship and the Robert and Marjorie Henle Chair in the Department of Electrical and Computer Engineering. He has received seven conference Best Paper awards, a Best Poster Award, two ICCAD 10-year Retrospective Most Influential Paper Awards, the SRC Technical Excellence award and the SIA University Research Award. He is a Fellow of the ACM and the IEEE.