FAT: An In-Memory Accelerator With Fast Addition for Ternary Weight Neural Networks

Shien Zhu, Luan H. K. Duong, Hui Chen, and Weichen Liu, Member, IEEE

Abstract—Convolutional neural networks (CNNs) demonstrate excellent performance in various applications but have high computational complexity. Quantization is applied to reduce the latency and storage cost of CNNs. Among the quantization methods, binary and ternary weight networks (BWNs and TWNs) have a unique advantage over 8- and 4-bit quantization. They replace the multiplication operations in CNNs with additions, which are favored on in-memory-computing (IMC) devices. IMC acceleration for BWNs has been widely studied. However, though TWNs have higher accuracy and better sparsity than BWNs, IMC acceleration for TWNs has limited research. TWNs on the existing IMC devices are inefficient because the sparsity is not well utilized, and the addition operation is not efficient. In this article, we propose FAT as a novel IMC accelerator for TWNs. First, we propose a sparse addition control unit, which utilizes the sparsity of TWNs to skip the null operations on zero weights. Second, we propose a fast addition scheme based on the memory sense amplifier (SA) to avoid the time overhead of both carry propagation and writing back the carry to memory cells. Third, we further propose a combined-stationary data mapping to reduce the data movement of activations and weights and increase the parallelism across memory columns. Simulation results show that for addition operations at the SA level, FAT achieves 2.00× speedup, 1.22× power efficiency, and 1.22× area efficiency compared with a state-of-the-art IMC accelerator ParaPIM. FAT achieves 10.02× speedup and 12.19× energy efficiency compared with ParaPIM on networks with 80% average sparsity.

Index Terms—AI accelerators, convolutional neural networks (CNNs), in-memory computing (IMC), spintronics.

I. INTRODUCTION

DEEP convolutional neural networks (CNNs) have been widely adopted in computer vision [1], natural language processing [2], robotics [3], and many other fields. Deep CNNs have numerous parameters, and the convolution layers are both memory intensive and computation intensive, resulting in high storage cost and long latency when deployed on the edge.

Manuscript received 23 October 2021; revised 14 March 2022 and 9 May 2022; accepted 1 June 2022. Date of publication 17 June 2022; date of current version 20 February 2023. This work was supported in part by the Ministry of Education, Singapore, through Academic Research Fund Tier 2 under Grant MOE2019-T2-1-071 and Tier 1 under Grant MOE2019-T1-001-072, and in part by the Nanyang Technological University, Singapore, through NAP under Grant M4082282 and SUG under Grant M4082087. This article was recommended by Associate Editor C. Bolchini. (Corresponding author: Weichen Liu.)

Shien Zhu, Luan H. K. Duong, Hui Chen, and Weichen Liu are with the School of Computer Science and Engineering, Nanyang Technological University, Singapore (e-mail: shien001@e.ntu.edu.sg; lhkduong@ntu.edu.sg; hui.chen@ntu.edu.sg; liu@ntu.edu.sg).

Di Liu is with the HP-NTU Digital Manufacturing Corporate Laboratory, Nanyang Technological University, Singapore (e-mail: liu-di@ntu.edu.sg).

Digital Object Identifier 10.1109/TCAD.2022.3184276

© 2022 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See https://www.ieee.org/publications/rights/index.html for more information.

IEEE TRANSACTIONS ON COMPUTER-AIDED DESIGN OF INTEGRATED CIRCUITS AND SYSTEMS, VOL. 42, NO. 3, MARCH 2023 781
TWNs. As deep CNNs are both memory-intensive and computation-intensive, the memory access on hardware is an essential factor in CNN performance besides the computation. Unfortunately, traditional von Neumann architecture separates the memory from the processing unit, which results in limited memory bandwidth, long memory access latency, and high energy costs in transferring data between the memory and processing units. These challenges are known as the memory wall. IMC is a promising way to relieve the memory wall, utilizing resistant memories, such as spin-transfer torque magnetic random-access-memory (STT-MRAM) to compute inside the memory [19]. Thanks to the massive data parallelism, ultra-high internal memory bandwidth, and reduced data movement, IMC brings high performance and excellent energy efficiency. In addition, state-of-the-art (SOTA) IMC devices can conduct Boolean functions and addition operations, making them very suitable for accelerating addition-centric CNNs [20], [21], including BWNs and TWNs.

However, though TWNs have higher accuracy and better sparsity than BWNs, current IMC accelerators have not been optimized to fit the characteristics of TWNs. First, the sparsity of TWNs is not well utilized. The sparse weights of TWNs bring the need for compressed sparse formats to save storage [22], [23]. However, the compression and decompression modules will increase the complexity of the accelerator design, area cost, and power. Moreover, the compressed sparse format has reduced storage benefit on TWNs because it stores the indices of the 2-bit nonzero weights in longer bitwidth (e.g., 8-bit index numbers). It is a challenge to deal with the sparsity and the sparse data format. As a result, the sparsity of TWNs has not been exploited to skip the operations of neural networks on the existing IMC accelerators. For example, ternary-in-memory (TiM)-DNN [24] and XNOR-SRAM [25] all process the ternary neural networks without considering the sparsity.

Also, addition operations have replaced the multiplications for higher performance in TWNs, but the addition operations in the existing IMC devices are not efficient. The mainstream STT-CIM series [26]–[28], ParaPIM series [29], [30], and GraphS series [31]–[33] designs are excellent traditional IMC architectures or application-specific accelerators, but their addition schemes are not efficient. They either need to wait for the carry propagating to the last bit or write back the carry to the memory cells and read out the carry back and forth. Applying the same addition schemes directly to processing TWNs will lead to long latency and high power consumption.

In this article, we proposed FAT as an STT-MRAM-based IMC accelerator for high-performance and energy-efficient TWN inference. First, we propose a sparse addition control unit (SACU) at the architecture level, which utilizes the sparsity of TWNs to speed up the convolution and fully connected layers. We combine the 2-bit weights with the control signals of the memory arrays to skip the addition operations where the weights are zeros. This design also benefits FAT with the 16× storage reduction without compressed sparse formats.

Second, we propose a fast addition scheme at the circuit level based on the memory sense amplifier (SA). We store the intermediate carry in a latch and compute the summation of two operands in only one step. This addition scheme avoids the time overhead of both waiting for the propagation of the carry signal and writing the carry back to the memory.

Third, we propose a combined-stationary (CS) data mapping scheme to fit the new computation pattern in FAT and fully utilize the computational capacity. The image-to-column (Img2Col)-based flexible mapping can reduce the data movement of activations and weights for lower energy, achieve near 100% column parallelism in memory arrays for higher performance, and balance the endurance of STT-MRAM cells for a longer lifetime.

Our proposed T WN accelerator FAT combines the speedup from fast addition and sparsity together, as Fig. 1 shows. Taking the SOTA IMC BWN accelerator ParaPIM [29] as the baseline, the proposed fast addition scheme based on the SA provides a 2.00× fast and 1.22× power-efficient addition operator. Our SACU provides another 5.00× speedup with 80% layer sparsity. As a result, FAT is up to 10.02× faster and 12.19× more energy-efficient than ParaPIM on quantized CNNs with 80% average sparsity.

II. RELATED WORKS AND MOTIVATION
A. Nonvolatile Memory and In-Memory-Computing

Memory access is one of the leading performance and energy bottlenecks in traditional computing systems. Because moving the data from memory to processing units for computation and sending back the result to the memory lead to long latency and massive power consumption. Thus, placing the logic near or even inside the memory has been proposed to mitigate this problem, namely, near-memory computing (NMC) [34], [35] and IMC [18], [19]. NMC places the logic closer to the memory, while IMC utilizes the memory arrays to do computation.

IMC can be realized using static random-access memory (SRAM) [36], dynamic RAM (DRAM) [37], and nonvolatile memories (NVMs), including resistive RAM (ReRAM) [38], phase change memory (PCM) [39], STT-MRAM [19], spin-orbit torque MRAM (SOT-MRAM) [29], and other emerging memory devices [18]. NVMs outperform SRAM and DRAM with near-zero leakage power and the nonvolatile property of the stored data [40]. STT-MRAM and SOT-MRAM outperform ReRAM and PCM with ≈ 10^15 memory cell write endurance and shorter write latency [41], [42]. STT-MRAM also has excellent density, read speed, and energy efficiency. ReRAM and PCM crossbars are widely adopted for matrix-vector multiplications, while STT/SOT-MRAM arrays perform parallel Boolean and addition operations. Therefore, this article chooses STT-MRAM to accelerate addition-centric TWNs.
Fig. 2. STT-MRAM memory cells and the abstracted circuit. (a) Standard 1T-1MTJ STT-MRAM memory cell. (b) Equivalent circuit of one activated memory cell. (c) Activating two rows of memory cells simultaneously. (d) Equivalent circuit of two activated memory cells.

### Table II: Related IMC Accelerators on Binary and Ternary Networks

| CNN  | Target Ops          | NVM Based                                      | Tradi. Mem. Based   |
|------|---------------------|------------------------------------------------|---------------------|
| BNN  | XNOR + Popcnt       | RRAM-BNN [38]                                  | XNOR-SRAM [25]     |
|      |                     | PIMBALL [44]                                   | SRAM-CIM [36]      |
|      |                     | XNOR-BNN [40]                                  | XNOR-BNN [40]      |
|      |                     | 2T2R-TCAM [45]                                 | TM-DNN [24]        |
|      |                     | VR-XNOR [46]                                   | XNOR-SRAM [25]     |
| TNN  | Ternary Mul., or    | 4T2R-IM-DP [47]                                | IMC-CD-TNN [50]    |
|      | Gated-XNOR +       | SpinLiM [48]                                   |                     |
|      | Popcnt              | TeC-Cell [49]                                  |                     |
|      |                     | Ter-LiM [51]                                   |                     |
| BWN  | Dense Addition      | ParaPIM [29]                                   |                     |
|      |                     | MRIMA [30]                                     |                     |
| TWN  | Sparse Addition     | This Work                                      |                     |

The standard STT-MRAM memory cell contains one magnetic-tunnel-junction (MTJ) and one access transistor connected to the bit-line (BL), the word-line (WL), and the source-line (SL) as Fig. 2(a) shows. The MTJ has a pinned layer as the reference and a free layer whose magnetic orientation direction can be switched by memory writes. A tunneling oxide isolates the pinned layer and the free layer. The MTJ has a low resistance in the “parallel state” where the free layer has the same magnetic orientation direction as the pinned layer, and it has a high resistance in the other “anti-parallel state”. Applying the same current to the STT-MRAM memory cell, the SA receives a voltage $V_{\text{sense}}$ as shown in Fig. 2(b). The anti-parallel state MTJ gives a high sensed voltage (can be identified as “1”), while the parallel state MTJ provides a low sensed voltage (identified as “0”). Similarly, IMC takes advantage of the resistance of the memory cells to do computation. For example, activating two rows of STT-MRAM memory cells simultaneously as Fig. 2(c) and (d) shows, the sensed voltage can be high (the two cells storing “11”), middle (storing “01” or “10”) or low (storing “00”). Thus, the SA can perform computations like Boolean functions between the operands in the activated cells by identifying the sensed voltage, and this is the basic idea of IMC.

### B. In-Memory-Computing Accelerators for Quantized CNNs

We review the related works on IMC accelerators for quantized neural networks in Table II, where traditional architecture-based accelerators, such as Bit-Tactical [43], are not listed. As IMC architectures are able to compute Boolean functions and the addition operation, IMC-based CNN accelerators mainly focus on ternary and binary networks: TNNs and BNNs, which quantize both the weights and activations to 2-bit/1-bit values and the main operations are Boolean operations, and TNNs and BNNs that only quantize the weights and the primary operation is addition. The accelerators can also be categorized into traditional memory-based (SRAM and DRAM) and NVM-based (including ReRAM, PCM, STT-MRAM, and SOT-MRAM) according to the memory type.

BNNs replace the 1-bit multiplication with XNOR and the 1-bit accumulation with popcnt (count the number of 1s in a binary value) in the binary dot product. Thus, the BNN accelerators target XNOR and popcnt for high speedup. RRAM-BNN [38], XNOR-BNN [40], and PIMBALL [44] utilize the XNOR operations of the ReRAM, SRAM, and STT-MRAM memory arrays to process the BNNs. Some related works build special memory cells to accelerate BNNs. For example, 2T2R-TCAM [45] creates a 2-transistor-2-ReRAM (2T2R) ternary content addressable memory (TCAM) that supports in-memory logic and XNOR/XOR-based binary dot product. Voltage-resistance XNOR (VR-XNOR) [46] proposes a memristor-based VR-XNOR cell with a filter bank for binary convolution acceleration. Some papers also use traditional memories, especially SRAM, as processing elements for BNNs. SRAM computing-in-memory (SRAM-CIM) [36] proposes an SRAM-CIM unit-macro for binarized fully connected neural networks. XNOR-SRAM [25] presents an SRAM macro supporting ternary-XNOR-and-accumulate operations for both BNNs and TNNs.

TNN acceleration is another active research area besides BNN acceleration. The dominant operation in TNNs is ternary multiplication, which is equivalent to a Gated-XNOR followed by a popcnt. Thus, most TNN accelerators build dedicated processing units for 2-bit ternary multiplications. 4T2R-IM-DP [47] provides a ReRAM-based 1C 4T2R bit cell to enable in-memory dot product (IM-DP) for TNNs. Ternary logic-in-memory (Ter-LiM) [51] implements a Ter-LiM scheme based on the memristive dual-crossbar structure with multilevel memristor cells. SpinLiM [48] utilizes two 2T-2MTJ SOT-MRAM cells to realize the ternary multiplication. Ternary compute-enabled memory cell (TeC-Cell) [49] proposes a nonvolatile TeC-Cell based on ferroelectric transistors as well as TeC-Arrays to perform parallel signed ternary multiplications. The SRAM-based TNN accelerators in related works also introduce special processing units for the ternary multiplications. For example, TiM-DNN [24] proposes TiM tiles for parallel signed ternary vector–matrix multiplications based on CMOS ternary processing cells with memory and ternary multiplication functions. IMC-CD-TNN [50] realizes the ternary multiplications with switch-capacitor ternary neurons and the wide vector summation in the charge domain to accelerate TNNs.

BWNs transform the multiplications into additions rather than the Boolean operations, such as BNNs and TNNs, so BWN accelerators utilize the addition and the memory-array-level parallelism in IMC architectures. ParaPIM [29] takes advantage of the parallel addition in SOT-MRAM memory arrays for efficient in-memory addition and convolution. Similarly, MRIMA [30] leverages the massive parallelism
as Fig. 3(a) shows. The upper two arrays stand for the operands $A$ and analyze the addition schemes in these related works and has $N$ series, and GraphS series, as Fig. 3 shows. We will review our knowledge, this work is the first sparse IMC accelerator across the STT-MRAM memory banks, matrices, and arrays to accelerate BWNs. As a result, ParaPIM and MRIMA both achieve higher performance and energy efficiency than DRAM-based, ReRAM-based, and ASIC-based accelerators.

TWNs are different from BWNs due to the 2-bit weights, the higher accuracy and the network sparsity. To the best of our knowledge, this work is the first sparse IMC accelerator for TWNs. This work builds more efficient addition operations than ParaPIM and MRIMA. This article also takes the 2-bit representation and the network sparsity into account to enable sparse additions for lower latency and higher energy efficiency.

C. In-Memory-Computing Addition Schemes

Existing IMC devices are able to perform Boolean functions and addition operations. The mainstream IMC addition schemes can be categorized into STT-CiM series, ParaPIM series, and GraphS series, as Fig. 3 shows. We will review and analyze the addition schemes in these related works and propose our efficient addition scheme in this section.

STT-CiM series designs [26]–[28] store the values in rows, as Fig. 3(a) shows. The upper two arrays stand for the operands $A = (1, 1, 1)$ and $B = (3, 3, 3)$. First, the SA of STT-CiM reads two operands together and analyzes the total current to calculate the summation result $Y = A + B$. Second, the SA outputs the addition result $Y = (4, 4, 4)$, and the result can be stored in the memory array at the bottom. This addition scheme is logically straightforward and efficient for scalar addition. However, it has two shortcomings: first, it needs to wait for the carry to propagate to the last bit, so the scalar latency $t_{\text{STT}}$ is long. As (1) shows, $t_{\text{STT}}$ increases linearly with the operand bitwidth $N$ due to the carry calculation and propagation latency $t_{\text{Carry}}$. Second, the vector addition $t_{\text{STT}}$ has $N$ times latency as the scalar addition, as (2) shows

\[ t_{\text{STT}} = t_{\text{Read}} + (N - 1) \cdot t_{\text{Carry}} + t_{\text{SUM}} + t_{\text{Write}} \]  
\[ t_{\text{VSTT}} = t_{\text{STT}} \times N \]  
\[ t_{\text{VFAT}} = (t_{\text{Read}} + t_{\text{SUM}} + t_{\text{Write}}) \times N. \]

ParaPIM [29] and MRIMA [30] are IMC accelerators for BWNs based on SOT-MRAM and STT-MRAM, respectively. ParaPIM series accelerators store the values in columns, as Fig. 3(b) shows. For example, operand 1 encoded as "001" is stored in a column rather than a row in the first memory array $A$. ParaPIM series accelerators perform the addition bit by bit rather than finish the scalar addition in one step. This addition scheme has the advantage that the vector addition has the same latency as the scalar addition as long as the vector length does not exceed the number of memory columns. However, it also has two weaknesses: first, it computes the Sum and Carry-out sequentially, which brings long latency. Second, this addition scheme needs to write the carry-out back to the memory to utilize it as the carry-in for the next bit, which results in extra latency and energy cost.

GraphS [31], ParaPIM-SA-II [32], and CA-DNN-PIM [33] adopt a new SA design to overcome the first weakness of ParaPIM series addition while keeping a similar workflow. GraphS series designs compute the sum and carry-out in one step as Fig. 3(c) shows and achieve $1.3 \times$ performance/area as ParaPIM. However, GraphS series designs still have two problems: first, GraphS only has $0.8 \times$ energy efficiency/area as ParaPIM due to its complex 3-operand logic and the increased area. Second, GraphS series accelerators still store back and read in the intermediate carry results to/from the memory array following ParaPIM, bringing unnecessary latency and energy.

To overcome the shortcomings of existing addition schemes, we propose an efficient addition scheme at the algorithm level along with a novel SA at the circuit level. Our FAT style fast addition operation does not need to wait for the propagation of the carry nor write the carry back to the memory, as illustrated in Fig. 3(d). First, similar to ParaPIM and GraphS, we store the operands in columns and compute the summation bit by bit. Second, instead of storing the intermediate carry of the previous bit back to the memory array, we keep it into a D-latch inside the SA to make it more energy efficient. Third, we compute the summation by reading in two operands in only one step with the carry-in (same as the carry-out from the previous bit) stored inside the SA, which brings a shorter

![Fig. 3. Addition schemes in (a) STT-CiM series [26]–[28], (b) ParaPIM series [29], [30], (c) GraphS series [31]–[33] IMC designs, and (d) our proposed FAT.](image-url)
critical path (CP) for addition operations. As the carry is used in the next bit, storing the carry in a latch hides the carry calculation and propagation latency. As (3) shows, FAT’s vector addition latency \( t_{PA} \) equals \( N \times 1 \)-bit addition, shorter than STT-CiM when \( N > 1 \). Also, we only need 2-operand logic instead of the 3-operand logic in ParaPIM and GraphS, reducing the power and increasing the sensing reliability.

III. ACCELERATOR FAT

We present our accelerator FAT in this section, including the architecture overview, the memory arrays with computational capacity, and the data mapping method. The memory array features two components: 1) the SACU, which provides a sparse vector dot product and 2) the SA, which implements the proposed fast addition scheme. In addition, the CS data mapping method reduces the data movement and increases the parallelism of memory arrays.

A. Overview

Our accelerator, FAT, is an STT-MRAM-based IMC accelerator for ternary weight neural network (TWN) inference. FAT also supports binary weight neural network (BWN) inference with few configurations. We will explain the accelerator networks, the accelerator architecture, and the basic workflow in this section.

1) Target Applications: The typical basic block of CNNs comprises a convolutional or fully connected layer followed by an activation function and a batch normalization layer. The convolutional and fully connected layers are computation intensive, while the activation and batch normalization layers only have a few floating-point operations. For example, the convolution of activation \( X \) and weight \( W \) needs \( C \cdot KH \cdot KW \) multiplications for every output point, as (4) shows, where \( KN, C, KH, \) and \( KW \) are the number of filters, channel, kernel height, and kernel width. In comparison, the popular activation function ReLU() and the batch normalization BN() are much simpler than convolution, as (5) and (6) show

\[
Y_{kn,h,w} = \sum_{i=1}^{C} \sum_{j=1}^{K} \sum_{c=1}^{N} X_{c,h+i,w+j} \cdot W_{kn,c,i,j} \quad (4)
\]

\[
\text{ReLU}(x) = \begin{cases} 
  x, & x > 0 \\
  0, & x \leq 0
\end{cases} \quad (5)
\]

\[
\text{BN}(Y) = \frac{Y - \mu[Y]}{\sqrt{\text{Var}[Y]} + \epsilon}. \quad (6)
\]

The weights in TWNs are quantized to ternary values to reduce the computation complexity of convolution and fully connected layers. The weights are ternarized to \( \{+1, 0, -1\} \) by comparing with trained or given thresholds, as (7) shows, where \( w \) and \( w' \) are the original and ternarized weight values, and \( \text{TH}_{\text{low}} \) and \( \text{TH}_{\text{high}} \) are the thresholds. Modern TWNs train the weights to be ternary values [11]–[13]; thus, the weights of target neural networks are already quantized into 2-bit numbers when the training finishes

\[
w' = \begin{cases} 
  +1, & w > \text{TH}_{\text{high}} \\
  -1, & \text{TH}_{\text{low}} < w < \text{TH}_{\text{high}} \\
  0, & \text{otherwise}
\end{cases} \quad (7)
\]

\[
y = \bar{x} \cdot \bar{w}' = \sum_{i=1}^{N} \bar{x}_i \cdot \bar{w}'_i, \quad \bar{w}' \in \{+1, 0, -1\}^N. \quad (8)
\]

The convolution and matrix multiplication in convolution and fully connected layers can be decomposed as the inner dot product between the activation vector \( \bar{x} \) and the weight vector \( \bar{w}' \), as (8) shows, where \( N \) is the vector length. Furthermore, the ternarized weights in TWNs simplify the dot product into addition/subtraction and null operations (multiplying zero is nonsense). Therefore, we build FAT to accelerate TWNs with two main features: 1) the low-level fast addition and 2) the high-level addition-based sparse dot product that skips the null operations. FAT is also able to infer BWNs because the weights of BWNs only contain \( \{+1, -1\} \), but BWN inference has no performance benefit from the sparsity.

2) Accelerator Architecture: Our accelerator FAT consists of computing memory arrays (CMAs) and a CMOS data processing unit (DPU) as Fig. 4(a) shows. The CMAs are in charge of the addition operations in convolutional and fully connected layers, equipped with the proposed SACU and SA for sparse dot products and efficient addition operations. FAT contains 4096 CMAs with 64-MiB total memory capacity and 128K 2-bit weight registers.

The DPU takes care of the batch normalization and activation layers. We keep almost the same DPU architecture as ParaPIM [29] and MRIMA [30] except that our DPU has no hardware quantizer. As the weights of modern TWNs and BWNs are trained to be 2-bit ternary values or 1-bit binary values, which means the weights are already quantized before inference, the weight quantizer is unnecessary. Excluding the quantizer also reduces the chip area, power, and inference time, thanks to no procedure on quantization. In summary, the CMAs with sparse dot product and efficient addition work together with the DPU containing activation functions and batch normalization to finish the TWN inference.

3) Computation Workflow: FAT conducts the inference block by block, and the workflow of one example convolution layer is shown in Fig. 4(b). First, the activations of each convolution layer are loaded to the memory cells of CMAs. Second, the weights of each layer are loaded to the SACU inside the memory controller (MC). The loading of weights may repeat many times as there are many filters in a convolution layer. Then the MC performs parallel computation...
across memory columns. All the CMAs with loaded activations and weights can process the convolution layer in parallel. Fourth, the convolution results are sent to the DPU through the internal buses. Fifth, the DPU performs the activation function and batch normalization to generate the output feature maps. After all, the output feature maps are stored again from the DPU to CMAs to work as activations of the next layer. We repeat this process until we get the output of the whole neural network.

B. Computing Memory Array

The CMA consists of an MC, a memory row address decoder (MRAD) which controls the WLs, a memory column address decoder (MCAD)/Writer Driver that controls the BLs, a memory array, and SAs connected to the SLs, as Fig. 5 shows. Keeping the same array size as related works [29], [33], each CMA has 512 rows and 256 columns. FAT adopts the same column-major data storage format as ParaPIM series [29], [30] and GraphS series [31]–[33] designs, which stores different bits of one number in sequential rows in one column. As we store all the data in a column-major format, there is no need to convert the data between bit serial and bit parallel.

Our CMA works in three modes: 1) a standard memory device mode with basic Read/Write support; 2) a traditional IMC device mode that enables the Boolean and Addition functions; and 3) a TWN accelerator mode. Working as a standard memory device or a traditional IMC device, the MC receives instructions from a CPU and sends control signals to other components to perform read, write, Boolean, and addition operations. The MRAD/MCAD decodes the row/column addresses based on the signals from MC and activates the WLs/BLs to access target memory cells. Then, the SA receives the current from the memory array by the SL and produces the corresponding result of Read, Boolean functions or Addition. These two memory modes also utilize the nonvolatility of STT-MRAM by keeping the data after power-off and avoiding data reloading when powered on again. Working as a TWN accelerator, our CMA provides two main features. First, the SACU inside the MC enables the high-level sparse vector dot product. Second, the SA facilitates the low-level fast addition. We will introduce the SACU and the SA in the following two sections in detail.

1) Sparse Addition Control Unit: When processing a convolution layer or a fully connected layer in TWNs, the weights determine which operands of the activations should be added together. Therefore, we store the activations inside the memory array to conduct Addition operations. At the same time, we load the weights into the MC, to be exact, the inside SACU.

   a) Architecture: The architecture of the SACU is shown in Fig. 5(a). It contains weight registers and a reduction unit. The SACU uses weight registers to generate the control signals of the sparse dot product. The reduction unit can accumulate the summations in different columns when necessary. The weights of TWNs only have three values, namely, \{+1, 0, -1\}. Thus, we adopt the encoding of standard signed integer for the weights, as Table III shows. The 2-bit encoding containing a sign bit and a data bit determines what operation should be applied to the activations in corresponding rows. The sign bit determines whether this is an addition (sign bit = 0) or a subtraction (sign bit = 1). As +1(01) and 0(00) share the same sign bit “0,” we need the data bit to work as a mask. Therefore, we activate the corresponding rows only when data bit = 1 so that the rows corresponding to weight 0 (00) have no operation. In other words, the null operations are skipped during the sparse dot product.

   b) Sparse addition workflow: As mentioned in the previous paragraph, the data bit of the weight determines whether the operands in the corresponding row need operation and the sign bit indicates the operation type is addition or subtraction. This motivates us to conduct the addition-based sparse dot product following Fig. 5(d) workflow. The SACU generates the row activation signals based on the data in weight registers, and the MRAD enables the corresponding rows of operands for addition.
The addition-based sparse dot product has three stages. Taking Fig. 5(d) as an example, two vectors \(a\) and \(b\) in the memory array perform dot product with the weight vector \((0, +1, +1, -1, 0, -1)\). First, we add the operands corresponding to weight +1 and get a partial sum in each column. The SACU activates rows 2 and 3 simultaneously, adds them together, and produces partial sums \(Sa_1\) and \(Sa_2\). Second, we add the operands corresponding to weight -1 and get another partial sum in every column. The SACU adds rows 4 and 6 together and calculates partial sums \(Sb_2\) and \(Sb_2\) simultaneously. Third, a subtraction operation (\(\text{SUB} = \text{NOT} + \text{ADD}\)) between these two partial sums produces the summation result of the whole column. The SACU performs \(\text{Co}1 = Sa_1 - Sa_2\) and \(\text{Co}2 = Sb_1 - Sb_2\) in parallel.

Our sparse dot product has three features. First, it automatically skips the addition operations corresponding to zero weights. Because the SACU only activates the rows where the weights are +1 or -1. Second, it has outstanding memory-column-level parallelism. As the example shows above, the memory columns can be activated simultaneously to conduct parallel processing across the whole memory array. We will combine this column-level parallelism with suitable mapping and scheduling to boost the performance further in Section III-C. Third, it replaces the subtractions corresponding to weights -1 with additions to achieve higher performance and energy efficiency. The addition is more efficient than subtraction on FAT because subtraction is realized by \(\text{NOT}\) and addition. The proposed three-stage addition pipeline separates the operands corresponding to weight +1 and -1, and performs additions on all the activation operands and only one subtraction on the partial sums, which reduces the dot product latency and energy.

As mentioned in the accelerator overview, FAT could also work as a BWN accelerator with simple configurations. We only need to extend the 1-bit binary weight data (+1 and -1) to 2-bit signed integers according to Table III, then store the 2-bit binary values into weight registers in the SACU, and FAT will work as a BWN accelerator. However, the SACU will activate all the rows based on the weights in this case, so there will be no benefit from sparsity.

2) Sense Amplifier: The SA is the core component of an IMC device because it determines which kinds of operations the device support. Our motivation drives us to design an SA that supports faster and more efficient addition operations than related works. Furthermore, our SA is able to conduct common Boolean functions and subtraction not limited to the addition operation. In the following paragraphs, we will introduce the SA architecture, the supported operations, and the comparison with related works.

a) Architecture and workflow: Our SA’s architecture is presented in Fig. 5(c). The signal flow inside the SA has four stages: 1) sensing; 2) comparing; 3) combining; and 4) selecting. We will follow the signal flow to show how our SA works.

Sensing: In the sensing stage, the WL and the BL are activated, and the current flows through the target memory cells to the SL, as Fig. 5(b) shows. The equivalent circuit of the sensing state is illustrated in Fig. 6(a). The operational amplifier (OpAmp) in the SA gets the total current from the SL along with the voltage of SL \(V_{SL}\), as (9) shows, where \(I_{\text{ref}}\) is the reference activation current, and \(R_{\text{MTJ}}\) and \(R_T\) are the resistances of the MTJ and the access transistor in a memory cell

\[
V_{SL} = I_{\text{ref}} \cdot ((R_{\text{MTJ1}} + R_T) (R_{\text{MTJ2}} + R_T)) \quad (9)
\]

\[
V_{\text{ref}} = I_{\text{ref}} \cdot R_{\text{ref}}. \quad (10)
\]

Comparing: Getting the sensed voltage \(V_{SL}\), the OpAmp in the SA compares it with reference voltages \(V_{\text{ref}}\) [generated in (10)] to get the result of READ, AND, or OR, which can be further used to compute more complex functions. The sensed voltage \(V_{SL}\) of reading out a single memory cell can be either a lower voltage \(V_{p,0}\) when the MTJ is in a parallel state (storing a 0) or a higher voltage \(V_{ap,1}\) with an anti-parallel state MTJ (storing a 1). Thus, the reference voltage for reading \(V_{\text{READ}}\) lies between \(V_{p,0}\) and \(V_{ap,1}\), as Fig. 6(b) shows. Similarly, the sensed voltage \(V_{SL}\) of reading out two memory cells can be \(V_{p,00}, V_{ap,01}\), and \(V_{ap,11}\), as Fig. 6(c) shows. Thus, the reference voltage of AND \(V_{\text{AND}}\) lies between \(V_{p,00}\) and \(V_{ap,01}\), while the OR is between \(V_{p,00}\) and \(V_{ap,01}\).

\[
A \text{ XOR } B = [A \text{ AND } B] \text{ NOR } [A \text{ NOT } B] \quad (11)
\]

\[
\text{SUM} = [A \text{ XOR } B] \text{ XOR } Cin \quad (12)
\]

\[
\text{Cout} = ([A \text{ OR } B] \text{ AND } Cin) \text{ OR } [A \text{ AND } B]. \quad (13)
\]

Combining: Next, the other logic gates in the SA combine the AND, OR, and NOR signals generated by the OpAmps to compute more complex functions, e.g., XOR, SUM, and carry-out of the addition operation as presented in Fig. 5(c). For example, the XOR is calculated by \(\text{NOT}\) between the AND and NOR of operands \(A\) and \(B\), as (11) shows, where the underlined parts in the square brackets are signals generated by the OpAmp from the comparing stage. Similarly, the SUM and the carry-out \(\text{Cout}\) are calculated following (12), (13) where \(Cin\) is the carry-in from the previous bit stored in the D-Latch. Thus, our SA uses four logic gates (\(\text{NOT}, \text{XOR}, \text{OR}, \text{AND}\)) and one D-Latch in the combining stage, as (11)–(13) and Fig. 5(c) show.

Selecting: Finally, the selector selects the desired result based on the selecting signals \(\text{Sel1}\) and \(\text{Sel2}\), and sends it to the output port OUT.

b) Configuration and supported operations: Our SA performs READ, \(\text{NOT}\), AND, NAND, \(\text{OR}\), \(\text{XOR}\), and Addition (\(\text{ADD}\)) functions natively, and Subtraction (\(\text{SUB}\)) extensively. It relies on the enable signals and selector signals from the MC to give the desired result. We configure the enable signals of
The SA to perform different functions according to Table IV. Meanwhile, we select the results routed to the input ports of the selector as Table V shows, and then we get the desired result at the OUT port.

The SA supports eight functions but building eight selector ports for all the functions brings high complexity and large area cost. Thus, we optimize the SA on READ, NOT, NAND and SUB functionality to simplify the design to only four selector ports. First, the READ and OR operations share the same OR selector port because they use the same OpAmp. Second, as the NOT equals XOR with 1s as (14) shows, we read in the operand along with a row filled with 1s and produce the NOT result at the XOR selector port. RISC-V has adopted the same technique to simplify the instruction set by replacing NOT with XOR [52]. Third, we disable the enable signals of EN_OR and EN_READ at the second OpAmp in the SA when computing the NAND. Then the OpAmp NOR port produces “0’s for any $V_{th}$ higher than zero. The NAND result appears at the XOR port after a NOR between the AND and “0’s, as (15) shows. Last, as the SUB equals ADD the opposite number as shown in (16), we perform the SUB operation by one NOT followed by one ADD with the first carry-in to be 1

$$\text{NOT } A = A \oplus 111 \cdots 1$$  \hspace{1cm} (14)  

$$A \text{ NAND } B = (A \text{ AND } B) \text{ NOR } 000 \cdots 0$$  \hspace{1cm} (15)  

$$A - B = A + ((\text{NOT } B) + 1).$$  \hspace{1cm} (16)  

Therefore, our SA has the least number of enabling (EN) signals, selector signals and amplifiers among related works, as inferred from Table VI. Nevertheless, the XOR and addition operations cost one D-latch and four Boolean gates. The area of the SAs can also be inferred from Table VI. STT-CiM’s SA has smaller area than ours due to one less D-latch, but STT-CiM has four more control signals than FAT. The SA of FAT has a smaller area than ParaPIM’s SA, thanks to fewer control signals and a smaller output selector. OpAmps usually have a much larger area than the Boolean gates due to higher complexity and higher loading capacity. Thus, the SA of FAT will be smaller than GraphS’ SA, which has one more OpAmp and a larger selector. The exact area and speed comparison will be given in Section IV.

### Fast addition:
We propose the new SA to adopt the efficient addition scheme shown in Fig. 3(d). Temporarily our CMA works as a sequential 1-bit adder and computes the summation bit by bit to realize the N-bit addition. First, the MC initializes the D-Latch containing the carry in the SA as 0 before the addition operation. Next, the MCAD enables those columns containing the operands, and the MRAD activates two rows catering to two bits of the operands simultaneously. The reference current flows through the memory cells to the SA. Then the SA computes the summation (SUM) of these two bits and stores the carry-out of the current bit in the Carry Latch to use it as the carry-in of the next bit. This unique design avoids storing back the carry-out results to the memory and saves a considerable amount of time and energy compared with ParaPIM [29] and GraphS [31] series IMC accelerators. Thanks to the bit-by-bit addition, storing the carry in a latch also hides the latency of carry calculation and propagation in the STT-CiM [26] series designs. Compute-SRAM [53] is an SRAM-based IMC design that adopts similar one-step bit-serial addition and storing the carry inside a D-Latch as FAT. Except for memory specific differences, FAT distinguishes from Compute-SRAM on using OR instead of XOR to generate the carry and simplifying the output selection to use a 4-input selector rather than an 8-input one. FAT has faster carry signal than Compute-SRAM because OR is earlier than XOR in the SA as (11)–(13) show. FAT has faster SUM signal than Compute-SRAM because our XOR port has fewer loading circuits than Compute-SRAM’s and the 4-input selector has shorter latency than the 8-input selector.

Our addition operation is throughput driven rather than latency driven. FAT has longer latency than STT-CiM series IMC designs on single scalar addition because we compute the summation bit by bit. FAT’s advantage lies in the vector addition, where hundreds to thousands of pairs of operands need addition results. Thanks to the column-major data storage format shown in Fig. 3, the bit-by-bit addition in FAT, ParaPIM and GraphS has N times parallelism as STT-CiM series designs on N-bit vector addition. Supposing two 1024 × 32-bit vectors stored in 1024 columns and 32 rows conduct addition inside a memory array. FAT repeats the 1-bit addition 32 times to get the vector addition result because the operand vector is stored in 1024 columns vertically. In contrast, STT-CiM performs the 32-bit addition 32 times across the memory array because the operand vector is stored in 32 rows horizontally. Our FAT has shorter vector addition latency than STT-CiM because FAT conducts the 1-bit addition 32 times while STT-CiM repeats the 32-bit addition 32 times.

In summary, our addition scheme is faster than ParaPIM, GraphS, and Compute-SRAM on scalar and vector addition, and is faster than STT-CiM on vector addition. Thus, we refer to the proposed addition scheme as fast addition. The Evaluation section will provide a detailed performance comparison of these works. As the IMC addition operands are
C. Data Mapping

FAT has a new data mapping scenario, where the activations are mapped to the memory arrays, and the weights are mapped to the MC to control the sparse dot product. Suitable data mapping brings higher performance and better energy efficiency. However, the existing mapping methods for IMC platforms cannot directly apply to FAT. Peng et al. [54] targeted ReRAM crossbars where weights are stored in the memory cells and activations are mapped to WLs. Jain et al. [26] mapped both activations and weights to the memory arrays. FAT cannot adopt these two mapping schemes due to architecture differences. BWN accelerators ParaPIM [29] and MRIMA [30] only provide the mapping of computing one output feature at a time, which does not apply to FAT due to the lack of high-level data movement.

Therefore, we review the data mapping schemes and present the CS mapping to optimize memory utilization and parallelism, taking the convolution layer as an example.

1) Existing Convolution and Mapping Methods: Convolution is usually implemented by direct convolution or Img2Col + general matrix multiplication (GEMM). In direct convolution, the weights slide across the activations and perform vector dot product, as the first algorithm in Fig. 7 presents. For an activation tensor a in shape \([N, C, H, W]\) (Batch Size, Channel, Height, Width) and the weight tensor w in shape \([KN, C, KH, KW]\) (Number of Filters, Channel, Kernel Height, Kernel Width), the shape of output feature map tensor y is \([N, OC, OH, OW]\) (Batch Size, Output Channel, Output Height, Output Width). As the output channel OC equals the filter number KN, we use kn to access the second dimension of y in line 8 of Algo.1. Direct convolution is straightforward to implement and can reduce memory usage due to the data reuse across sliding windows. However, the convolution stride S can decrease its data reuse. Furthermore, direct convolution is a sequential algorithm inefficient for achieving high parallelism.

Therefore, Img2Col/GEMM-based convolution is introduced to reduce the data dependency and improve the parallelism. First, Img2Col transforms the activations into 2-D arrays taking the convolution stride into account, as Fig. 8 shows. Then, a GEMM between the unrolled weights and the transformed activations computes the convolution results. Img2Col only keeps the activations needed for convolution to maximize the data-level parallelism.

There are four main ways of mapping the convolution data to the hardware for execution, namely, weight-stationary (WS), input-stationary (IS), output-stationary (OS) and row-stationary (RS) [15], [55], [56]. WS is optimized to reuse the weights and reduce the movement of weights. Similarly, IS and OS are optimized to reuse the activations and the (intermediate) convolution results, respectively. While RS optimizes DRAMs’ memory access, accessing the DRAM data in consecutive rows has less latency and energy. Fig. 7 provides an example of OS mapping that keeps the output feature map tensor y in the inner loop unchanged as long as possible.

2) Img2Col-Based Combined-Stationary Mapping: We propose the mapping method for FAT based on the following considerations. First, the memory arrays are throughput driven with high parallelism across columns. Img2Col-based convolution is favored because it decouples the sliding window style direct convolution into a parallel GEMM problem. Second, The 2-bit weights of TWNs are loaded to the SRAM registers inside the MC, while the 8-bit activations are loaded to the STT-MRAM memory array. Reloading the 2-bit weights is faster and more energy-efficient than reloading the 8-bit activations. Third, SRAM has almost unlimited write endurance, but STT-MRAM has \(10^{15}\) write operations in its lifecycle. Therefore, reusing the activations brings more memory cell endurance and write energy benefits than reusing the weights, and keeping the activations in the memory arrays rather than moving them is essential.

Therefore, we propose a CS style mapping for FAT upon the Img2Col/GEMM-based convolution and IS mapping scheme, bringing high data-level parallelism and reducing the data movement of activations. Figs. 7 and 9 present the algorithm and graphic illustrations of CS mapping. The \(Ax[N][KN][I]\), \(Aw[KN][I]\), and \(Ay[N][I][J]\) in the second algorithm of Fig. 7 refer to the activation array, the weight array, and the convolution kernel array, respectively.
TABLE VII

| Mapping          | X/Aw|N|H|J| Loading | W/Av|K|N|E| Loading | Parallel Columns | Occupied CMAs | Computing Time |
|------------------|-----|---|---|---|---------|-----|---|---|---|---------|-----------------|---------------|-----------------|
| Direct-OS        | KN*N*MH*PW | [W/M]*[W/M]*[W/M]*[W/M]*[W/M] | KN*N*MH | [W/M]*[W/M]*[W/M]*[W/M]*[W/M] | min(MW/S, H/WS) | KN*N | [W/M]*[W/M]*[W/M]*[W/M]*[W/M] |               |
| Img2Col-OS       | KN*N*MH*PW | [W/M]*[W/M]*[W/M]*[W/M]*[W/M] | KN*N*MH | [W/M]*[W/M]*[W/M]*[W/M]*[W/M] | min(MW/S, H/WS) | KN*N | [W/M]*[W/M]*[W/M]*[W/M]*[W/M] |               |
| Img2Col-IS       | N*F* | 1 | N*F* | 1 | min(MW/S, H/WS) | KN*N | [W/M]*[W/M]*[W/M]*[W/M]*[W/M] |               |
| Img2Col-CS       | L*F* | 1 | L*F* | 1 | min(MW/S, H/WS) | KN*N | [W/M]*[W/M]*[W/M]*[W/M]*[W/M] |               |

TABLE VIII

| Mapping          | CMAs | X/Av Loading | W/Av Loading | Layer Level Performance | Loading | Computing | Max Single Cell Write |
|------------------|------|--------------|--------------|-------------------------|---------|-----------|----------------------|
| Direct-OS        | 4096 | 21668 | 3.29M | 12437 | 0.59K | 128756 | 76.56% | 71144 | 1.00x | 4.295 | 100.0% | 64x |
| Img2Col-OS       | 4096 | 48753 | 7.40M | 3105 | 1.34K | 196256 | 76.56% | 60883 | 1.17x | 7058 | 164.3% | 64x |
| Img2Col-IS       | 4096 | 2708 | 0.51M | 2323 | 1.09K | 256256 | 94.23% | 14622 | 4.88x | 2440 | 56.8% | 64x |
| Img2Col-WS       | 4096 | 48753 | 7.40M | 169 | 0.08K | 196256 | 76.56% | 60481 | 1.18x | 7057 | 164.3% | 64x |
| Img2Col-CS       | 4096 | 1354 | 0.51M | 1259 | 1.09K | 256256 | 47.11% | 10400 | 6.86x | 2449 | 57.8% | 1x |

Fig. 9. (a) Mapping of filters and activations to the CMAs. (b) Computation sequence on the example activation matrix with eight CMAs. (c) Computation sequence with three CMAs.

output feature map array after Img2Col. The output feature maps’ height and width are combined as one dimension I (I = OH*OW), which is mapped to the memory columns horizontally for parallel processing. The filters’ channel, height, and width are combined as one dimension J (J = C*KH*KW), which is mapped to the memory rows vertically for sequential addition. The 2-bit zero and nonzero weights are loaded to the MCs of corresponding memory arrays.

The CS mapping contains the following optimizations upon the IS mapping. First, we distribute J = C*KH*KW across memory arrays in aligned columns. As Fig. 9(a) shows, J is the vector length in the sparse dot product that caters to a whole output point. One CMA with 512 rows and 256 columns can store MH*MW = 64*256 8-bit operands. As the CMA height MH = 64 is usually smaller than J, storing the whole dimension J inside one CMA reduces the column parallelism by J/MH times, which also needs to load one filter in J/MH parts sequentially. The 2-bit zero and nonzero weights are loaded to the MCs of corresponding memory arrays.

The CS mapping contains the following optimizations upon the IS mapping. First, we distribute J = C*KH*KW across memory arrays in aligned columns. As Fig. 9(a) shows, J is the vector length in the sparse dot product that caters to a whole output point. One CMA with 512 rows and 256 columns can store MH*MW = 64*256 8-bit operands. As the CMA height MH = 64 is usually smaller than J, storing the whole dimension J inside one CMA reduces the column parallelism by J/MH times, which also needs to load one filter in J/MH parts sequentially. The 2-bit zero and nonzero weights are loaded to the MCs of corresponding memory arrays.

end
unrolling factor across KN. The execution time in Table VII corresponds to the number of occupied memory arrays in perfect conditions without scaling-up or scaling-down. Thus, we need to compare their execution time on the same convolution layer and IMC device.

We take layer 10 of ResNet-18 as an example to give a showcase of the actual performance of these mapping methods in Table VIII, where \((N, C, H, W) = (5, 128, 28, 28)\), \((KN, KH, KW) = (256, 3, 3)\), stride \(S = 2\), and \((MH, MW) = (64, 256)\). The time and energy are calculated referring to \([57]\). We scale up the mapping methods to use the same 4096 CMAs for fair comparison, as more CMAs provide higher performance.

First, Img2Col-IS mapping achieves the fewest memory write on activations, the highest 256 parallel columns (Para. Cols), and the highest memory utilization (94.23%) because it unrolls the activation across \(N*H\) to fill more CMAs and reuses activations. Second, Img2Col-WS mapping achieves the shortest weight loading time and fewest write times as it tries to keep the weights unmoved. Third, Img2Col-OS and Img2Col-WS have small speedup values due to long activation loading time than Direct-OS. As one CMA only need one column of 2-bit weights, the weight loading time and memory writes are much smaller than the activations. Thus, reusing the weights reduces less loading time than reusing the activations. Fourth, our Img2Col-CS mapping keeps the same memory writes on activations and weights as Img2Col-IS mapping. The reserved intervals inside the CMAs reduce the effective MH from 64 to 32. Thus, our Img2Col-CS mapping has shorter activation and weight loading time than Img2Col-IS. Smaller MH also reduces the sequential additions at each CMA but increases the immediate results 2\(\times\). As a result, Img2Col-CS achieves the highest 6.86\(\times\) speedup with 0.2% slightly higher energy than Img2Col-IS. Fifth, Img2Col-CS balances the maximum cell write by 64\(\times\) (or up to MH\(\times\)) compared with other mappings at the cost of half memory utilization of activations. The other half of the memory rows store immediate addition results, which significantly increases the lifetime of the CMAs.

IV. EVALUATION

We implement the SAs of FAT and related works using NCSU 45-nm FreePDK45 library \([58, 59]\). We adopt the comparator of STT-CiM \([26]\) and include the output selector for all the four SAs for a fair comparison. We build, verify, and evaluate the circuits in Cadence Virtuoso IC6.1.8 using Virtuoso ADEL, Spectre, and Layout Suite XL to obtain the corresponding latency, power, and area. We refer to \([60]\) for the write time of the 1-Transistor-1-Junction STT-MRAM memory array implemented in the same 45-nm process.

The performance gain of our accelerator comes from the improvements in the addition operation and the sparsity. Keeping the standard STT-MRAM array which accounts for around 85% area in a CMA \([26]\) unchanged, we only modify the SA and add a SACU to the MC. The first section evaluates the SAs on the performance and efficiency of IMC operations. Then, we evaluate the network-level performance with sparsity in the second section.
addition in one step, as mentioned in Fig. 3, while the other three methods perform the 8-bit addition in eight steps. Thus, STT-CiM has the shortest CP and latency when performing one scalar addition. The ParaPIM series, GraphS series, and proposed FAT style addition conduct the addition bit by bit. One 8-bit addition and adding two 8-bit vectors have the same eight steps in the bit-by-bit style addition, as long as the vector length does not exceed the memory array width. Thus, adding two scalars and two vectors share the same latency in these three methods. Our FAT outperforms ParaPIM and GraphS in the CP, single addition, and vector addition, thanks to the new SA and the proposed efficient addition scheme, which stores the carry in a latch instead of the memory array. FAT has shorter latency than STT-CiM on vector addition because STT-CiM has to repeat the addition for \(N\) times in the \(N\)-bit vector addition.

Our FAT builds the SA for the most efficient vector addition among related works. As neural networks are throughput driven and TWNs utilize vector additions in convolution layers, vector addition is more important than a single scalar addition in TWN accelerators. We take the 32-bit vector addition to analyze further the performance and efficiency of these SAs in Fig. 11, including the latency, performance/Watt, energy-delay-product (EDP), and power density (power/area).

First, FAT is \(1.12 \times 2.00 \times 1.98\) faster than STT-CiM, ParaPIM, and GraphS, respectively, in 32-bit vector addition. Taking the write overhead of the SUM and Carry into account, we find that ParaPIM and GraphS are much slower than STT-CiM due to the extra write of Carry. Though GraphS has less computation latency in the SA, writing the Carry to memory is so time consuming that it slows down the whole vector addition. Our FAT avoids the propagation of Carry and has a shorter SUM CP than STT-CiM. FAT also avoids the extra write of Carry in ParaPIM and GraphS. Thus, FAT has the shortest vector addition latency. Second, FAT has the highest performance/Watt and is \(1.01 \times 2.86\) as efficient as related works. Third, our FAT has the least EDP among related works and is \(1.14 \times 5.69\) as efficient as STT-CiM, ParaPIM, and GraphS. Fourth, FAT has a lower power density than STT-CiM and GraphS, which means our FAT is more balanced on the power distribution and may have a longer lifetime.

In summary, we realize a vector addition with high performance and high efficiency for TWNs by innovation from the addition scheme and the SA circuit.

3) Area and Reliability of Sense Amplifiers: We present the high-resolution layout figure (6800 × 1300 pixels) of our SA in Fig. 12 for reference, which can be zoomed in for more details. The area of our proposed FAT and related works are shown in Fig. 13. We normalize the area to FAT for easy comparison. FAT has 21% more area than STT-CiM because of the D-Latch, but it is \(1.22 \times 1.17\) area efficient than ParaPIM and GraphS. ParaPIM has seven output ports requiring an 8-input selector and a D-Latch. Thus, ParaPIM has a larger size than STT-CiM with a 4-input selector and no latch. GraphS has fewer logic gates but one more comparator than ParaPIM, resulting in a similar area as ParaPIM. Though FAT has the most logic gates, FAT has smaller area than ParaPIM and GraphS due to only two comparators and a 4-input selector. Also, our SA is more reliable than ParaPIM and GraphS. FAT’s SA only contains two-operand operations rather than three-operand operations like ParaPIM and GraphS. The sense margin of two-operand operations is \(2.4\) as high as that of three-operand operations \([29]–[32]\). As a larger sense margin brings less error rate, our SA is more reliable than ParaPIM and GraphS.

B. Network-Level Performance

Fig. 14 presents the network-level speedup and energy efficiency on models with similar sparsity across layers, which also reflects the single layer speedup and efficiency. We take ParaPIM as the baseline because only ParaPIM series designs are BWN IMC accelerators among the three series of related works. 

Fig. 11. Normalized latency and efficiency of 32-bit addition of our proposed FAT and related works (Baseline: FAT).

Fig. 12. Layout of our SA in virtuoso layout suite XL.

Fig. 13. Normalized area breakdown of the SAs in our proposed FAT and related works (Baseline: FAT).

Fig. 14. Speedup and energy efficiency of our FAT across network sparsity (Baseline: ParaPIM [29]).
works. The overall performance improvement is determined by both the fast addition and the sparsity. We achieve 2.00× speedup and 1.22× power efficiency compared with ParaPIM from the proposed addition operator. The SACU further brings speedups from the sparsity of the TNNs. Our accelerator can achieve 3.34×, 5.01×, and 10.02× network-level speedup and 4.06×, 6.09×, and 12.19× energy efficiency compared with ParaPIM when the average sparsity is 40%, 60%, and 80%. Since our mapping in Section III-C performs dense mapping, and then the proposed SACU exploits the fine-grained sparsity of the filters, the speedup and energy efficiency are independent of layer sizes and the model architectures.

V. CONCLUSION

In this article, we proposed FAT as an IMC accelerator with fast addition and sparse dot product for TNNs. We proposed an efficient in-memory addition scheme and a new SA that stores the carry inside the SA to avoid the carry propagation latency and carry writing back latency. Our SA is 2.00× faster, 1.22× power efficient, and 1.22× area efficient than ParaPIM and MRIMA on addition operations. We proposed a SACU which utilizes the sparsity of TNNs to skip the addition operations corresponding to zero weights. FAT with the sparse dot product achieves up to 10.02× speedup and 12.19× energy efficiency on networks with 80% sparsity compared with ParaPIM and MRIMA. We further presented the CS mapping to reach near 100% column parallelism across the whole memory array and balance the STT-MRAM cell write times for longer memory lifetime, which brings 6.86× speedup and reduces 43.0% energy compared with direct convolution-based OS mapping on ResNet-18 layer 10.

REFERENCES

[1] M. Tan, R. Pang, and Q. V. Le, “EfficientDet: Scalable and efficient object detection,” in Proc. IEEE/CVF Conf. Comput. Vis. Pattern Recognit., 2020, pp. 1078–10790.
[2] A. Baevski, H. Zhou, A. Mohamed, and M. Auli, “WarZeC 2.0: A framework for self-supervised learning of speech representations,” in Advances in Neural Information Processing Systems, vol. 33, Red Hook, NY, USA: Curran Assoc., 2020.
[3] E. Wijmans et al., “DD-PPO: Learning near-perfect pointgoal navigators from 2.5 billion frames,” in Proc. Int. Conf. Learn. Represent. ICLR, 2020, pp. 1–21.
[4] M. Rastegari, V. Ordonez, J. Redmon, and A. Farhadi, “XNOR-Net: ImageNet classification using binary convolutional neural networks,” in Proc. Eur. Conf. Comput. Vis., 2016, pp. 525–542.
[5] Z. Cai, X. He, J. Sun, and N. Vasconcelos, “Deep learning with low precision by half-wave Gaussian quantization,” in Proc. IEEE Comput. Vis. Pattern Recognit., 2017, pp. 5918–5926.
[6] H. Pouransari, Z. Tu, and O. Tuzel, “Least squares binary quantization of neural networks,” in Proc. IEEE/CVF Conf. Comput. Vis. Pattern Recognit. Workshops, 2020, pp. 698–699.
[7] F. Zhu et al., “Towards unified INT8 training for convolutional neural network,” in Proc. IEEE/CVF Conf. Comput. Vis. Pattern Recognit., 2020, pp. 1969–1979.
[8] K. Zhao et al., “Distribution adaptive INT8 quantization for training CNNs,” in Proc. AAAI Conf. Artif. Intell., 2021, pp. 1–9.
[9] R. Li, Y. Wang, F. Liang, H. Qin, J. Yan, and R. Fan, “Fully quantized network for object detection,” in Proc. IEEE/CVF Conf. Comput. Vis. Pattern Recognit., 2019, pp. 2810–2819.
[10] X. Sun et al., “Ultra-low precision 4-bit training of deep neural networks,” in Advances in Neural Information Processing Systems, vol. 33, Red Hook, NY, USA: Curran Assoc., 2020.
[11] C. Zhu, S. Han, H. Mao, and W. J. Dally, “Trained ternary quantization,” in Proc. 5th Int. Conf. Learn. Represent., 2017, pp. 1–10.
[12] Y. Li, X. Dong, S. Q. Zhang, H. Bai, Y. Chen, and W. Wang, “RTN: Reparameterized ternary network,” in Proc. AAAI Conf. Artif. Intell., vol. 34, 2020, pp. 4780–4787.
[13] Y. Li, W. Ding, C. Liu, B. Zhang, and G. Guo, “TRQ: Ternary neural networks with residual quantization,” in Proc. AAAI Conf. Artif. Intell., 2021, pp. 8538–8546.
[14] N. Parashar et al., “SCNN: An accelerator for compressed-sparse convolutional neural networks,” ACM SIGARCH Comput. Archit. News, vol. 45, no. 2, pp. 27–40, 2017.
[15] Y.-H. Chen, T.-J. Yang, J. Eimer, and V. Sze, “Eyeriis V2: A flexible accelerator for emerging deep neural networks on mobile devices,” IEEE J. Emerg. Sel. Topics Circuits Syst., vol. 9, no. 2, pp. 292–308, Jun. 2019.
[16] J. Choquette and W. Gandhi, “NVIDIA A100 GPU: Performance amp; innovation for GPU computing,” in Proc. IEEE Hot Chips (HCS), 2020, pp. 1–43.
[17] K. He, X. Zhang, S. Ren, and J. Sun, “Deep residual learning for image recognition,” in Proc. IEEE Conf. Comput. Vis. Pattern Recognit., 2016, pp. 770–778.
[18] A. Sebastian, M. Le Gallo, R. Khadham-Aljameh, and E. Eleftheriou, “Memory devices and applications for in-memory computing,” Nat. Nanotechnol., vol. 15, no. 7, pp. 529–544, 2020.
[19] S. Jain, S. Sapatnekar, J.-P. Wang, K. Roy, and A. Raghunathan, “Computing-in-memory with spintronics,” in Proc. Design Autom. Test Europe Conf. Exhibition (DATE), 2018, pp. 1640–1645.
[20] C.-J. Jiang, C.-X. Xue, J.-M. Hung, F.-C. Chang, and M.-F. Chang, “Challenges and trends of SRAM-based computing-in-memory for AI edge devices,” IEEE Trans. Circuits Syst. I, Reg. Papers, vol. 68, no. 5, pp. 1773–1786, May 2021.
[21] Y. Ma, Y. Du, L. Du, J. Lin, and Z. Wang, “In-memory computing: The next-generation AI computing paradigm,” in Proc. Great Lakes Symp. VLSI, 2020, pp. 265–270.
[22] D. Fujiki, N. Chatterjee, D. Lee, and M. O’Connor, “Near-memory data transformation for efficient sparse matrix multi-vector multiplication,” in Proc. Int. Conf. High Perform. Comput. Netw. Storage Anal., 2019, pp. 1–17.
[23] A. Mehrabi, D. Lee, N. Chatterjee, D. J. Sorin, B. C. Lee, and M. O’Connor, “Learning sparse matrix row permutations for efficient SPM on GPU architectures,” in Proc. IEEE Int. Symp. Perform. Anal. Syst. Softw. (ISPASS), 2021, pp. 48–58.
[24] S. Jain, S. K. Gupta, and A. Raghunathan, “TIM-DNN: Ternary in-memory accelerator for deep neural networks,” IEEE Trans. Very Large Scale Integr. (VLSI) Syst., vol. 28, no. 7, pp. 1567–1577, Jul. 2020.
[25] S. Y. Jin, Z. Jiang, J.-S. Seo, and M. Seok, “XNOR-SRAM: In-memory computing SRAM macro for binary/ternary deep neural networks,” IEEE J. Solid-State Circuits, vol. 55, no. 6, pp. 1733–1743, Jun. 2020.
[26] S. Jain, A. Ranjan, K. Roy, and A. Raghunathan, “Computing in memory with spin-transfer torque magnetic RAM,” IEEE Trans. Very Large Scale Integr. (VLSI) Syst., vol. 26, no. 3, pp. 470–483, Mar. 2018.
[27] S. K. Thirunala, S. Jain, A. Raghunathan, and S. K. Gupta, “Non-volatile memory utilizing reconfigurable ferroelectric transistors to enable differential read and energy-efficient in-memory computation,” in Proc. IEEE/ACM Int. Symp. Low Power Electron. Design (ISLPED), 2019, pp. 1–6.
[28] S. K. Thirunala et al., “Valley-coupled-spintronic non-volatile memories with compute-in-memory support,” IEEE Trans. Nanotechnol., vol. 19, pp. 635–647, Jul. 2020, doi: 10.1109/TNANO.2020.3012550. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/9151583
[29] S. Angizi, Z. He, and D. Fan, “ParaPIM: A parallel processing-in-memory accelerator for binary-weight deep neural networks,” in Proc. 24th Asia South Pacific Design Autom. Conf., 2019, pp. 127–132.
[30] S. Angizi, Z. He, A. Awad, and D. Fan, “MRIMA: An MRAM-based in-memory accelerator,” IEEE Trans. Comput.-Aided Design Integr. Circuits Syst., vol. 39, no. 5, pp. 1123–1136, May 2020.
[31] S. Angizi, J. Sun, W. Zhang, and D. Fan, “Graphs: A graph processing accelerator leveraging SOT-MRAM,” in Proc. Design Autom. Test Europe Conf. Exhibition (DATE), 2020, pp. 378–383.
[32] S. Angizi and D. Fan, “Deep neural network acceleration in non-volatile memory: A digital approach,” in Proc. IEEE/ACM Int. Symp. Nanoscale Archit. (NANOARCH), 2019, pp. 1–6.
[33] L. Yang, S. Angizi, and D. Fan, “A flexible processing-in-memory accelerator for dynamic channel-adaptive deep neural networks,” in Proc. 25th Asia South Pacific Design Autom. Conf. (APSAD), 2020, pp. 313–318.
[34] G. Singh et al., “A review of near-memory computing architectures: Opportunities and challenges,” in Proc. 21st Euromicro Conf. Digit. Syst. Design (DSD), 2018, pp. 608–617.
[35] G. Singh et al., “Near-memory computing: Past, present, and future,” Microprocess. Microsyst., vol. 71, Nov. 2019, Art. no. 102868.

[36] X. Si et al., “A dual-split 6T SRAM-based in-memory unit-macro with fully parallel product-sum operation for binarized DNN edge processors,” IEEE Trans. Circuits Syst. I, Reg. Papers, vol. 66, no. 11, pp. 4172–4185, Nov. 2019.

[37] M. F. Ali, A. Jaiswal, and K. Roy, “In-memory low-cost bit-serial addition using commodity DRAM technology,” IEEE Trans. Circuits Syst. I, Reg. Papers, vol. 67, no. 1, pp. 155–165, Jan. 2020.

[38] X. Sun, X. Peng, P.-Y. Chen, R. Liu, J.-S. Seo, and S. Yu, “Fully parallel rram synaptic array for implementing binary neural network with (+1, −1) weights and (+1, 0) neurons,” in Proc. 23rd Asia South Pacific Design Autom. Conf. (ASP-DAC), 2018, pp. 574–579.

[39] A. Antolini et al., “Characterization and programming algorithm of phase change memory cells for analog in-memory computing,” Materials, vol. 14, no. 7, p. 1621, 2021.

[40] X. Sun, R. Liu, X. Peng, and S. Yu, “Computing-in-memory with SRAM and RRAM for binary neural networks,” in Proc. 14th IEEE Int. Conf. Solid-State Integrated Circuit Technol. (ICSICT), 2018, pp. 1–4.

[41] S. H. Kang, “Embedded STT-MRAM for mobile applications: Enabling advanced chip architectures,” Non-Volatile Memories Workshop, 2010, pp. 1–23.

[42] S. Angizi et al., “Accelerating deep neural networks in processing-in-memory platforms: Analog or digital approach?” in Proc. IEEE Comput. Soc. Annu. Symp. VLSI (ISVLSI), 2019, pp. 197–202.

[43] A. D. Laszcorz et al., “Bit-tactical: A software/hardware approach to exploiting value and bit sparsity in neural networks,” in Proc. 24th Int. Conf. Archit. Support Program. Lang. Oper. Syst., 2019, pp. 749–763.

[44] S. Resch et al., “PIMBALL: Binary neural networks in spinnronic memory,” ACM Trans. Archit. Code Optim., vol. 16, no. 4, pp. 1–26, 2019.

[45] Y. Chen, L. Lu, B. Kim, and T. T.-H. Kim, “Reconfigurable 2T2R ReRAM with split word-lines for TCAM operation and in-memory computing,” in Proc. IEEE Int. Symp. Circuits Syst., 2020, pp. 1–5.

[46] Y. Halawani, B. Mohammad, M. Abu Lebdeh, M. Al-Qutayri, and S. F. Al-Sarawi, “ReRAM-based in-memory computing for search engines and neural network applications,” IEEE J. Emerg. Sel. Topics Circuits Syst., vol. 9, no. 2, pp. 388–397, Jun. 2019.

[47] Y. Chen, L. Lu, B. Kim, and T. T.-H. Kim, “A reconfigurable 4T2R ReRAM computing in-memory macro for efficient edge applications,” IEEE Open J. Circuits Syst., vol. 2, pp. 210–222, Jan. 2021. doi: 10.1109/OJCSA.2020.3042550. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/9335229.

[48] L. Luo, H. Zhang, J. Bai, Y. Zhang, W. Kang, and W. Zhao, “SpinLiM: Spin orbit torque memory for ternary neural networks based on the logic-in-memory architecture,” in Proc. Design Autom. Test Europe Conf. Exhibition (DATE), 2021, pp. 1865–1870.

[49] S. K. Thirumala, S. Jain, S. K. Gupta, and A. Raghunathan, “Ternary compute-enabled memory using ferroelectric transistors for accelerating deep neural networks,” in Proc. Design Autom. Test Europe Conf. Exhibition (DATE), 2020, pp. 31–36.

[50] X. Yang et al., “An in-memory-computing charge-domain ternary CNN classifier,” in Proc. IEEE Custom Integrate. Circuits Conf. (CICC), 2021, pp. 1–2.

[51] W. Liu, Y. Sun, W. He, and Q. Wang, “Design of ternary logic-in-memory based on memristive dual-crossbars,” in Proc. IEEE Int. Symp. Circuits Syst. (ISCAS), 2021, pp. 1–5.

[52] (RISC-V Int., Zürich, Switzerland). RISC-V Instruction Set Manual Volume 1: User-Level ISA (2019). [Online]. Available: github.com/riscv/riscv-isa-manual/releases/download/Ratified-IMAFDQC/riscv-spec-20191213.pdf.

[53] J. Wang et al., “A 28-nm compute SRAM with bit-serial logic/arithmetic operations for programmable in-memory vector computing,” IEEE J. Solid-State Circuits, vol. 55, no. 1, pp. 76–86, Jan. 2020.

[54] X. Peng, R. Liu, and S. Yu, “Optimizing weight mapping and data flow for convolutional neural networks on RRAM based processing-in-memory architecture,” in Proc. IEEE Int. Symp. Circuits Syst. (ISCAS), 2019, pp. 1–5.

[55] Y.-H. Chen, J. Emer, and V. Sze, “Eyeriss: A spatial architecture for energy-efficient dataflow for convolutional neural networks,” ACM SIGARCH Comput. Archit. News, vol. 44, no. 3, pp. 367–379, 2016.

[56] H. Kwon, P. Chatarasi, M. Pellauer, A. Parashar, V. Sarkar, and T. Kipf, “Understanding reuse, performance, and hardware cost of dnn dataflow: A data-centric approach,” in Proc. 52nd Annu. IEEE/ACM Int. Symp. Microarchit., 2019, pp. 754–768.

[57] D. Reis et al., “Modeling and benchmarking computing-in-memory for design space exploration,” in Proc. Great Lakes Symp. VLSI, 2020, pp. 39–44.

[58] “NCSU FreePDK45.” North Carolina State University. 2011. [Online]. Available: https://research.ece.ncsu.edu/eda/freepdk/freepdk45/

[59] J. E. Stine et al., “FreePDK V2.0: Transitioning VLSI education towards nanometer variation-aware designs,” in Proc. IEEE Int. Conf. Microelectron. Syst., Educ., 2009, pp. 100–103.

[60] L. K. Vemula, N. M. Hossain, and M. H. Chowdhury, “Emerging STT-MRAM circuit and architecture co-design in 45nm technology,” in Proc. IEEE 60th Int. Midwest Symp. Circuits Syst. (MWSCAS), 2017, pp. 719–722.

Shien Zhu received the bachelor’s degree from the University of Science and Technology of China, Hefei, China, in 2018. He is pursuing the Ph.D. degree with Nanyang Technological University, Singapore. His current research topics include accelerating quantized neural networks on the edge and designing in-memory computing accelerators for deep neural networks.

Luan H. K. Duong received the Ph.D. degree from the Hong Kong University of Science and Technology, Hong Kong, in 2018. He is currently a Postdoctoral Research Fellow with the School of Computer Science and Engineering, Nanyang Technological University, Singapore. His research interests include optical interconnect networks, optical computing, embedded systems, and many-core systems.

Hui Chen received the B.E. degree from the School of Computer Science and Technology, Zhejiang University, Hangzhou, China, in 2016, the M.Sc. degree from the School of Computing, National University of Singapore, Singapore, in 2018, and the Ph.D. degree from the School of Computer Science and Engineering, Nanyang Technological University, Singapore, in 2022. Her current research interests include embedded and real-time systems, multicore scheduling, and network-on-chip.

Di Liu (Member, IEEE) received the Ph.D. degree from Leiden University, Leiden, The Netherlands, in 2017. He is currently a Postdoctoral Researcher with the HP-NTU Digital Manufacturing Corporate Lab, Nanyang Technological University, Singapore. His research interest includes hardware-aware deep neural network design.

Weichen Liu (Member, IEEE) received the B.Eng. and M.Eng. degrees from the Harbin Institute of Technology, Harbin, China, in 2004 and 2006, respectively, and the Ph.D. degree from the Hong Kong University of Science and Technology, Hong Kong, in 2011. He is a Nanyang Assistant Professor with the School of Computer Science and Engineering, Nanyang Technological University, Singapore. He has authored and coauthored more than 100 research papers, peer-reviewed journals, conferences, and books. His research interests include embedded and real-time systems, multiprocessor systems, network-on-chip, and machine learning acceleration.