# Count2Multiply: Reliable In-Memory High-Radix Counting

João Paulo C. de Lima[†§], Ben Morris III[*], Asif Ali Khan[†], Jeronimo Castrillon[†§], Alex K. Jones[‡]
TU Dresden[†], ScaDS.AI[§], Duke University[*], Syracuse University[‡]
joao.lima@tu-dresden.de, ben.morris@duke.edu, asif_ali.khan@tu-dresden.de, jeronimo.castrillon@tu-dresden.de, akj@syr.edu

***Abstract*—Computing-in-memory (CIM) has been demonstrated across various memory technologies, from memristive crossbars for analog dot-products to large-scale digital bitwise operations in commodity DRAM and other non-volatile memory technologies. However, current CIM solutions face challenges related to latency and reliability. CIM fidelity lags considerably behind standard memory access fidelity. Furthermore, bulk-bitwise CIM, though highly parallel, requires long latency for multiplication and addition due to bit-serial execution. This paper presents Count2Multiply, a digital CIM framework that performs multiplication, addition, and other operations using high-radix, massively parallel counting enabled by bulk-bitwise in-memory operations. Designed to meet fault tolerance requirements, Count2Multiply integrates traditional row-wise error correction codes, such as Hamming and BCH, to address the high error rates in existing CIM designs. We demonstrate *Count2Multiply* in commodity DRAMs, achieving on average 1.5× speedup, 4.6× higher energy efficiency (GOPS/Watt), and 4.4× better area efficiency (GOPS/mm$^2$) over NVIDIA A100; and 9.7×, 8.1×, and 12.9× improvements, respectively, over SIMDRAM.**

## I. Introduction

Multiply-accumulate (MAC) is a fundamental computational primitive in many data-intensive application domains, including high-performance computing, machine learning, and bioinformatics. GPUs, TPUs, FPGAs, and other accelerators address these applications' needs with parallel execution units and/or integrated specialized MAC units. Despite delivering PetaFLOPs-scale performance, these architectures have substantial energy requirements and remain memory-bound due to their compute-centric nature [1]. Consequently, there is a growing trend towards compute-in-memory (CIM) solutions [2]–[4]. CIM has gained particular attention because emerging workloads often require only low-precision integer-integer ($\leq$ 8 bits) operations for sufficient accuracy [5]–[9].

CIM solutions fully exploit the internal bandwidth and parallelism offered by memory arrays. These solutions are generally divided into two types. **Analog CIM** exploits current- or charge-sharing within memory arrays to compute the weighted sum between an input voltage vector and a column of memory cells. This enables vector-matrix multiplication in constant time [10]. However, it is mainly suitable for applications that can tolerate some loss in accuracy. Conversely, **digital CIM**, *the focus of this paper*, performs computations with greater precision, making it useful for a wider range of applications. In digital CIM, bulk-bitwise operations are commonly used across various memory technologies such as DRAM, SRAM, and emerging non-volatile memories (NVMs). These operations enable efficient execution of basic logic gates like AND, OR, and NOT, and serve as building blocks for more complex functions such as addition and multiplication. Early proposals for bulk-bitwise logic emerged about a decade ago [11]–[13]. However, it is only in recent years that experimental studies have shown real CIM capabilities in DRAM [14]–[16] and emerging NVMs [17]–[20]. Nonetheless, two fundamental challenges continue to hinder CIM adoption:

Challenge 1. Bulk-bitwise operations are carried out in a bit-serial, word-/row-parallel fashion. This offers higher throughput compared to traditional CPU and GPU systems [21], [22] and also significantly reduces energy consumption. However, the per-operation latency increases. Even with massive SIMD-style parallelism, the control flow of bit-serial operations relies on *sequential ripple propagation*, as in ripple-carry adders and multipliers [21]–[23]. For instance, in Ambit [24], each operation takes 49 ns compared to <1 ns in CMOS. This generally increases compute latency due to each operation's latency and the bit-serial steps in a full MAC operation.

Challenge 2. Fault rates for DRAM-based CIM range from $10^{-1}$ (experimental demonstration) [15] to $10^{-6}$ (simulations) [25]. This higher fault rate arises from reduced sense margin coupled with the impact of process variations [20], [24]. Existing error correction codes (ECCs) efficiently protect memory accesses but cannot be directly used for CIM, as they are not homomorphic over most Boolean operations. Hence, ECCs for CIM operations remain unsolved.

We make three observations to address these challenges:

Observation 1. Given that the number of CIM operations for bit-serial arithmetic depends on *the precision of the operands*, tuning each MAC operation to the *bit-width of each data element* reduces the latency of CIM arithmetic.

Observation 2. Given that the latency of bit-serial arithmetic is based on *the number of carry operations*, using *larger radices* to reduce the number of carry propagation can reduce the latency of CIM arithmetic.

Observation 3. For any Boolean operation, there exists a sequence of additional operations that results in X(N)OR. As traditional memory ECCs are homomorphic over X(N)OR, these sequences can be checked via traditional syndrome checks.

Drawing from these observations, we introduce Count2Multiply, a digital CIM framework for integer-vector/matrix ($\mathbb{X}$) and binary-matrix ($\mathbb{Z}$) computations.

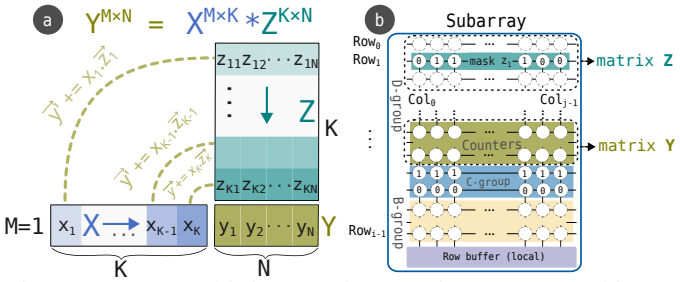



Fig. 1: Count2Multiply overview (a) integer-vector–binary-matrix multiplication example (b) DRAM subarray with counters and masks mapping, and Ambit's rows groups, i.e., computing (*B-group*), control (*C-group*) and data (*D-group*).

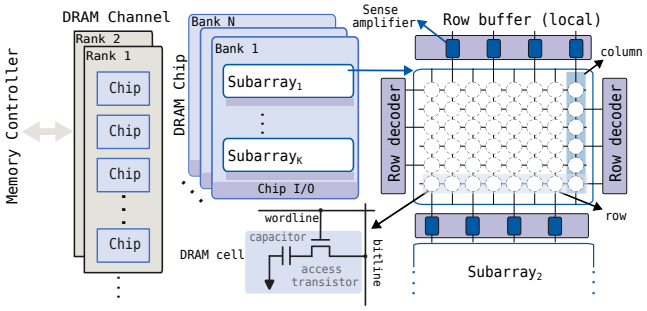
As shown in Fig. 1a, Count2Multiply departs from prior approaches [21], [26] that store both tensor operands in memory. Instead, it stores $\mathbb{Z}$ in the memory array, where it serves as counting masks. The elements of the integer-vector $\mathbb{X}$, $X_i$, are converted into memory commands and broadcast by the memory controller to update the high-radix counters organized column-wise in the memory array. Each column counter increments by $X_i$ only if the bit from $\mathbb{Z}$ (e.g., $Z_{i,j}$) is '1'. Count2Multiply naturally extends to perform integer-vector integer-matrix multiplication by applying bit slicing to $\mathbb{Z}$.

Count2Multiply leverages Observation 1 by tuning the number of CIM commands to the value $X_i$: *no commands are issued* for zero-valued digits in $X_i$, enabling efficient handling of sparse and narrow-width operands [27]–[32]. It also takes advantage of Observation 2 by using high-radix counters to minimize the number of carry operations required. When combined with an *early termination* of carry propagation, these optimizations significantly reduce the number of CIM operations and lead to substantial improvements in latency and energy compared to existing CIM addition methods.

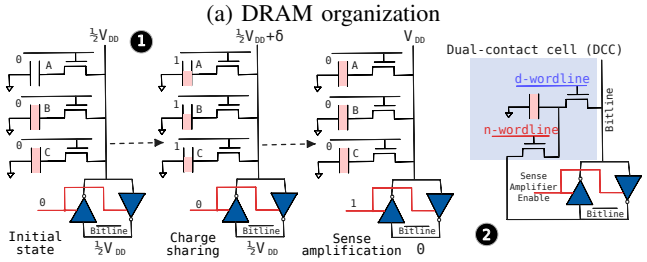
Count2Multiply also considers *reliability as a first-class optimization metric*. We revisit classic circuit concepts to implement counters with Johnson encoding, which also offer advantages in minimizing transition errors [33], [34]. Using Observation 3, we propose a set of CIM operations that increment Johnson counters and show how they can still protect against CIM faults—even if they occur during protection. Most current CIM designs lack built-in fault tolerance, relying mainly on replication and voting schemes [16], [35]. Count2Multiply reintroduces fault-tolerance with traditional ECCs to improve performance, storage overhead and fault tolerance over triple modular redundancy (TMR).

Building on Ambit [24], Count2Multiply implements fundamental tensor operations such as GEMV, GEMM, shift-left, ReLU, and addition. Our concrete contributions are as follows:

- We propose a novel high-radix in-memory counting methodology with optimizations that significantly enhance the performance of accumulation in DRAM (Sec. V).
- We illustrate how our in-memory counting mechanism can be used to execute massively parallel integer-vector binary-matrix multiplication and also extend it to integer matrix-matrix multiplications through bit-slicing, addition, and other operations (Sec. VI).
- We demonstrate how traditional row-level ECC can be leveraged to protect CIM operations in Count2Multiply, while still protecting row-level accesses (Sec. VII).
- We evaluate Count2Multiply on multiple applications from the bioinformatics and machine learning domains and compare performance to state-of-the-art in-DRAM designs and high-end GPUs (Sec. VIII).




(a) DRAM organization




(b) Ambit's triple-row activation for AND/OR and DCC for NOT

Fig. 2: Overview of DRAM and Ambit scheme for CIM.

Count2Multiply requires no hardware modifications, and compared to prior in-DRAM designs [21], it achieves up to 9.7× improvement in execution time, while delivering an average 8.1× higher GOPS/Watt and 12.9× higher GOPS/area.

## II. Background and Related Work

### A. DRAM Organization and Operation

Fig. 2a illustrates the hierarchical organization of a modern DRAM system. A CPU manages memory through multiple memory controllers (one per channel), each handling memory read, write, and refresh operations. Each channel connects to one or more DRAM modules, which contain ranks composed of multiple DRAM chips working in lockstep. A DRAM chip contains multiple banks that share an internal bus, connecting them to the chip's I/O circuitry. Each bank consists of several subarrays (e.g., 16-32), where each subarray consists of a two-dimensional grid of DRAM cells–the fundamental storage elements–each consisting a capacitor and an access transistor. These cells are organized into rows and columns and are supported by dedicated peripheral circuitry, including row decoders, sense amplifiers (SAs) also called row buffer, and local wordline drivers, for data manipulation. Within a subarray, each row is connected to a wordline, which is activated by the row decoder to enable read or write operations.

Accessing a DRAM row is a three-step process. First, the row is *activated* (ACT), bringing cell data to the row buffer. Second,

the *read/write* (RD/WR) operation transfers the data from/to the row buffer to/from the bus. Third, the row is *precharged* (PRE), restoring bitlines to a stable state for the next operation. The memory controller (MC) schedules commands regulated by a set of timing parameters that ensure sufficient delays between commands to correctly retrieve and retain data in DRAM cells. For example, tRAS (Row Active Time) defines the minimum time a row must remain active before it can be precharged, ensuring data is fully accessible.

### B. Compute-In-DRAM

Previous research has shown that certain functions can be performed directly *in memory* by carefully modifying the standard sequence of DRAM operations [14], [24], [36]. For instance, RowClone (RC) [36] copies src row to dst row within the same subarray using back-to-back ACT commands followed by a PRE command, known as *activate-activate-precharge* (AAP). The AAP sequence operates by first activating the src row to drive its contents onto the bitlines. Activating the dst row then transfers these values to overdrive its capacitors [36]. Finally, a precharge command resets the subarray for the next operation. In addition to RC, DRAM implements logic operations using *multirow activation* (MRA), wherein multiple rows in a subarray are activated *simultaneously*, followed by a PRE command, known as *activate-precharge* (AP). In-DRAM CIM designs achieve simultaneous MRA either through a custom row decoder [21], [24] or by violating memory timings to issue consecutive ACT commands [14], [15], [25].

As an example, consider the Ambit [24] approach in Fig. 2b that uses triple-row activation (TRA, ❶) to perform a bitwise majority (MAJ3) function. The state of the bitline reflects the majority state of the three activated cells. Initially, the bitline is precharged to a known voltage level ($\frac{1}{2}$VDD). During a TRA, the charge stored in the capacitors (i.e., 0, 1, 1) is shared onto the bitline, causing the bitline voltage to deviate ($\frac{1}{2}$VDD $+\delta$). Next, during the sense amplification, the cross-coupled inverters detect the small voltage difference and amplify it, pulling the bitline high (VDD). As a result, all involved cells are overwritten with a logical '1'.

For functional completeness, Ambit uses dual-contact cells (DCCs) to implement NOT, connecting a capacitor to either the bitline or $\overline{\text{bitline}}$ via two wordlines (d- and n-wordlines) (❷ in Fig. 2b). The MC uses reserved row addresses to access these wordlines. To compute NOT A $\rightarrow$ R, it issues: ACT A, ACT DCC's n-wordline, PRE, then copies from the d-wordline to row $R$. To simplify row decoding, Ambit divides the space of row addresses in each subarray into three groups, as shown in Fig. 1b: (i) B-group, eight rows for bulk bitwise MAJ3/NOT, (ii) C-group, two rows storing '0' (C0) and '1' (C1), and (iii) D-group, the remaining $r-10$ rows for data storage, where $r$ is the total row count per subarray. Though the B-group contains only 8 rows, it is responsible for 16 unique addresses – these addresses map to different combinations of 1, 2, or 3 rows – enabling the row activations required for access, copying and computing MAJ3, respectively.

### C. Fault Modes and Fault Tolerance for CIM

Violating DRAM timing parameters facilitates CIM but increases bit error rate [15], [16], [24], [25]. In simulations, the interaction between activated rows and bitlines to compute MAJ works reliably under an idealized scenario, assuming DRAM cells have rather small variability (<6% in [24]) and transistors and bitlines operate without deviation. However, in practical implementations, process variation induces non-uniform electrical characteristics across cells, resulting in instability in multi-row activation leading to faults.

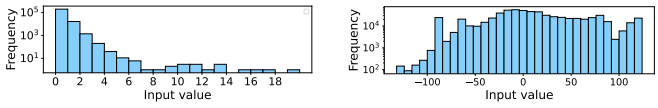
Presently, only minimal work beyond TMR exists to protect against CIM faults but they are for memories other than DRAM. For instance, for CIM based on Spintronic Racetracks, CIRM-ECC protects transverse read-based logic operations [37]. For RRAM, a recent work constructs the parity bits in specialized peripheral circuitry following each logic level of the CIM computation but this requires additional area overhead, and increases critical path latency [38]. In Sec. VII, we propose a reliability scheme that leverages *existing* ECC circuitry to protect memory access and CIM operations in Count2Multiply.

### D. Johnson Counters

Johnson counters (JC) [39], also known as twisted ring counters, are cyclic shift register-based sequential circuits with single-bit transitions between consecutive states. This intrinsic feature minimizes *transition errors* [33], [34]. For example, a 5-bit JC with the least significant to most significant bits moving left to right progresses through states as: 10000(1) $\rightarrow$ 11000(2) ... $\rightarrow$ 11111(5) $\rightarrow$ 01111(6) ... $\rightarrow$ 00001(9) $\rightarrow$ 00000(0), maintaining the cyclic property where decimal 9 rolls over to 0. This structure allows an $n$-bit counter to represent $2n$ distinct states. A JC counter increment shifts the register's content from the least to most significant bit position (*forward shift*) while inverting the most significant bit as a feedback bit to the new least significant bit (*inverted feedback*). Decrements follow shifting in the reverse direction.

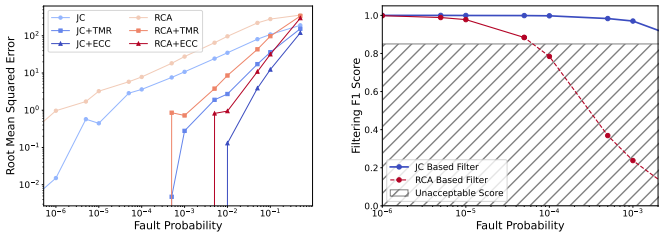
## III. Motivation

Several recent works on low-precision matrix multiplication have attempted to reduce the cost of MAC by using bitslicing [23], [35], [40], [41] or using Ternary Weight Networks (TWNs) [7], [42], [43]. With TWNs, multiplications are replaced by *masked additions* for more efficient CIM implementations [23], [26]. However, CIM arithmetic generally requires bit-serial addition in the form of ripple-carry adders (RCAs). To maximize the performance, this approach leverages element-parallel (e.g., vector-style) computing to add many values simultaneously [21], [26], [44]. Yet, additions in this computation style suffer from long carry propagation chains, even when many smaller values are added to a larger sum. Worse, managing these long carry chains can be made entirely *unnecessary* (Observation 1). Many memory-bound applications with low arithmetic intensity accumulate narrow-width values, as demonstrated in Fig. 3. For example, for the DNA pre-alignment filtering and BERT use-cases (see Sec. VIII-A), the accumulated values typically fall within a




(a) Short-read input tokens (b) 8-bit input embeddings

Fig. 3: Input distribution in DNA pre-alignment filtering and BERT language model. Values are small (circa 4–8 bits)




(a) Accumulated error of adds (b) Fault impact on DNA filtering

Fig. 4: CIM fault rate impact on accuracy

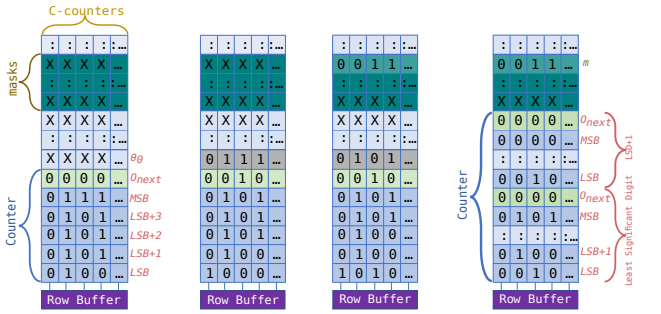
compact 4–8 bit range, as shown in Figs. 3a and 3b, respectively. For every new addition of these smaller values, RCA-based accumulation must fully process the carry propagation of larger values (circa 32 bits) due to a large accumulated total.

The additional CIM operations from carry chain management not only increase the latency but also substantially impact reliability, as more faulty CIM operations have the potential to perturb higher-order bits of the accumulated value. Two key reasons motivate the use of JC encoding for accumulation: (i) JCs represent high-radix numbers without requiring carry propagation (Observation 2), and (ii) the cyclic, self-checkable nature of the encoding minimizes transition errors and allows fault detection via invalid states. We compare the fault rate and impact of accumulated faults for RCA-based and radix-10 JC implementations to implement DNA pre-alignment filtering in Fig. 4. Comparing Root Mean Squared Error (RMSE) (4a), RCA shows substantial error with a CIM fault rate of $10^{-6}$, while JC can tolerate fault rates up to $10^{-5}$ to achieve the same error rate. The JC-based implementation sustains high accuracy at much higher fault rates compared to the RCA-based filter. This is critical for sensitive applications such as pre-alignment filtering, where the impact of faults on the application accuracy is severe. Fig. 4b highlights how the F1 score for RCA-based filters degrades sharply when CIM faults are introduced.

We also note that redundancy and voting are inherently inefficient: TMR has a circa $4\times$ overhead in operation count (three repeated operations and the voting operation) for both encodings (RCA+TMR and JC+TMR). In addition, TMR has a higher error rate than single ECC (RCA+ECC and JC+ECC). This motivates our approach that leverages JC accumulation and introduces integrated error correction based on traditional ECCs (Observation 3) rather than replication and voting.

## IV. Count2Multiply Overview

The Count2Multiply framework adopts the Ambit architecture *as-is*, including both the memory subarray and the MC proposals. It implements counting—a fundamental operation that is then leveraged to perform in-memory vector addition,




(a) initial (b) all count (c) masked (d) multi-digit

Fig. 5: $C$, 5-bit JCs in memory: (a) before counting; (b) all counters count; (c) masked counting; (d) multi-digit counting.

vector-matrix and matrix-matrix multiplication, as well as other tensor operations.

To illustrate in-memory counting, consider the example in Fig. 5a, where $C$ single-digit JCs are stored in a memory array. All counters are $n$-bit and can count from 0 to $2n-1$, i.e., radix-$2n$. In the figure, $n=5$ corresponds to a base-10 counter, from 0 ("00000") to 9 ("00001"). Our $n$-bit JC requires $n+1$ memory rows (one row for overflow, $O_{next}$). Bit positions are annotated on the right, ordered from the most significant bit (MSB) to the least significant bit (LSB). To increment these counters *by one*, each bit is *forward shifted* one position toward the MSB. Then, the LSB is updated using an *inverted feedback* of the old MSB, i.e., LSB $\leftarrow \overline{\text{MSB}}$.

These operations are implemented using CIM primitives as follows. First, the MSB row is copied into a temporary row (e.g., $\theta_0$) using RC. Next, each bit row, except the MSB, is shifted one position toward higher significance using successive RC operations. We call this the *forward shift* step. Finally, $\theta_0$ is inverted and copied into the LSB row using a NOT operation (see Sec. II-A). We call this the *inverted feedback* step. Incrementing 5-bit JCs (as shown in the Figure) thus requires four *forward shift* steps and one *inverted feedback* step. The sequence of memory commands used for incrementing is referred to as a *$\mu$Program*, as illustrated in its optimized form in Fig. 6b. For our running example, the result after an *increment by one* across $C$ 5-bit JCs are shown in Fig. 5b.

Selectively incrementing counters is supported by *predicating* the operations using masks, as illustrated in Fig. 5c and detailed in Sec. V-A. Similarly, increments by an arbitrary constant $9 \geq k > 1$ can be executed using the *same number of memory commands* as a unit increment, as described in Sec. V-D. For multi-digit counters (2-digit JC in Fig. 5d), each digit is incremented by its corresponding value in the multi-digit constant $(k_1, k_0)$, with overflow from a lower digit propagating an increment to the next higher digit.

In Count2Multiply, $\mu$Programs are generated at compile time, incurring no runtime overhead. During execution, the host CPU reads an increment value, populates the corresponding $\mu$Program, and broadcasts the associated memory commands to all memory arrays containing the counters, as demonstrated in Fig. 10. Section VI builds on this basic in-memory counting primitive to enable MAC and related operations. The key idea is

to store one operand in memory in a bit-sliced format (used as masks), broadcast the other operand as memory commands to all arrays, and accumulate the latter in the counters. This MAC, realized in *broadcast-and-accumulate* fashion, enables vector-matrix multiplication. We improve performance by *skipping zero-valued inputs*, reducing the number of counting operations. Our approach benefits workloads with sparse inputs [45]–[47]. When combined with our early-termination optimization in Sec. V-D, this results in *value-dependent* counting latencies, accelerating execution on inputs with narrow-width or bit-level sparsity [27]–[32].

In Section VII, we extend $\mu$Programs to enable protection. Importantly, like the counting logic, this requires *no additional hardware*; rather, we rely on the existing ECC modules in DRAM. We accomplish this by leveraging the data-dependent nature of faults in Ambit and other types of CIM.

## V. In-Memory High-Radix Counters

High-radix counters are constructed using multi-bit "digits" based on a specified radix and can consist as many digits as required to represent the maximum value for a given application. To implement these counters in memory, we allocate dedicated memory rows as shown in Fig. 5a, such that all bits of a counter reside in the same column. For counter digits, we adopt Johnson encoding (see Sec. II-D) as it requires fewer operations than an RCA for a single increment and provides fault-tolerance properties. A natural alternative to JC would be the trivial binary encoding. However, it inherently supports only power-of-two radices, reducing it to a special case of RCA-based accumulation, with the same drawbacks in terms of carry propagation and fault sensitivity.

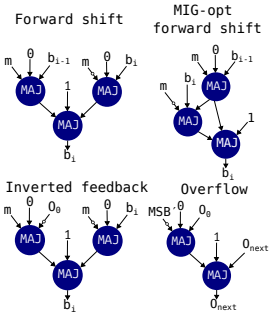
### A. Single-Digit Masked Unit Increment

Some applications, particularly those involving tensor operations, require selective increments or *masked counting*, where only a subset of counters is updated based on a stored mask $m$. This mask is stored in a row within the subarray containing the counters (highlighted row in Fig. 5c). The masked *forward shift* and *inverted feedback* are implemented using the following logical expressions:

$$b_i = (b_i \wedge \overline{m}) \vee (b_{i-1} \wedge m), \quad \text{where } i \in \{n, n-1, \ldots, 2\}$$
$$b_1 = (b_1 \wedge \overline{m}) \vee (\overline{b_n} \wedge m)$$

where $b_i$ represents the counter bit at index $i \in \{\text{LSB}...\text{MSB}\}$.

For every bit position ($b_i$) in the forward shift, the increment process requires two AND, one OR, and one NOT operation. The inverted feedback requires an additional NOT operation to invert the MSB. Fig. 6 shows the Majority-Inverter Graphs (MIGs) [48] and $\mu$Program for both forward shift and inverted feedback using Ambit's [24] primitives. As illustrated, we construct AND and OR from the MAJ3 function. We first synthesize this expression into a MIG (Fig. 6a), and subsequently employ MIG-based optimizations, similar to prior works [48], [49]. This minimizes the number of RCs in both the forward and backward shifts by scheduling the MAJ3 operations and allocating the rows in the B-group to maximize data reuse and




(a) MIG-based circuits

```
0: AAP b_n, O_0          // O0 ← MSB; Setup
1: ForwardShift:
2:   AAP     m, B8       // T0 ← m; DCC0 ← m̄;
3:   AAP    C0, B9       // T1 ← 0; DCC1 ← 1;
4:   AAP   b_i-1, B2     // T2 ← bi-1
5:   AP        B12       // T0,T1 ← T2 & T0
6:   AAP    b_i, B2      // T2 ← bi
7:   AAP   B14, B3       // T3 ← T2 & DCC0
8:   AAP   B15, b_i      // bi ← T0 | T3
9: InvertedFeedback:
10:  AAP    b_1, B2      // T2 ← LSB
11:  AAP     m, B8       // T0 ← m; DCC0 ← m̄;
12:  AAP    C0, B9       // T1 ← 0; DCC1 ← 1;
13:  AAP   B14, B3       // T3 ← T2 & DCC0
14:  AAP    O_0, B5      // DCC0 ← O0
15:  AP        B11       // T0 ← DCC0 & T0
16:  AAP   B15, b_1      // LSB ← T0 | T3
```

(b) $\mu$Program for Ambit [24]

Fig. 6: Majority-based in-DRAM operations for counting and overflow detection.[1]

reduce the number of row initialize operations, i.e., cloning constant '0' or '1' control rows for AND or OR to a compute row.

Fig. 6b presents the optimized sequence of seven AP/AAP memory commands for each step of the counter increment. Using AAP operations, the mask row $m$, constant zero C0, and the counter bit $b_{i-1}$ are cloned to three rows in the B-group (Line 2–4 in Fig. 6b), followed by MAJ3 to implement AND (Lines 5,7) and OR (Line 8) operations. The NOT operation is inherently handled by a RC operation with special DCC rows [24], so obtaining $\overline{m}$ incurs no additional overhead. This process repeats for all $n$ bits in the counter ($n-1$ forward shift and 1 inverted feedback step), for a total of $7n$ operations.

### B. Overflow Detection in Single-Digit Counters

For single-digit counters (Fig. 5a-5c), when a counter rolls over (*overflows*), this information is retained in a dedicated row ($O_{next}$). An overflow is detected when the JC's MSB transitions from '1' to '0'. In masked counting, each increment can potentially cause an overflow in one or more of the $C$ counters. The *overflow detection* is expressed as: $O_{next} \leftarrow O_{next}$ OR $\theta_0$ AND $\overline{\text{MSB}'}$, where MSB$'$ is the new MSB after increment (*forward shifts* and *inverted feedback*). Calculating $O_{next}$, as illustrated in Fig. 6a, requires a total of six AAP operations (4 RC and 2 MAJ3 operations).

### C. Multi-Digit Increment

For $D$-digit counters (Fig. 5d), all digits belonging to a counter are stored in the same column, requiring $D \cdot (n+1)$ rows. Starting from the least significant *digit* (LSD), the counter digits are updated sequentially using the single-digit counting mechanism. After $2n$ increments (*forward shift*, *inverted feedback* and *overflow detection*) to the $LSD$, $O_{next}$ is used to increment the LSD$+1$ and subsequent digits through $O_{next}$ rippling. Note, so far we have only discussed single-digit inputs. However, in practice, inputs can also have multiple digits, with each digit ranging between 0 and $2n-1$. In such cases, for counter accumulation, i.e., adding the *same multi-digit input* to all *multi-digit counters in memory*, we align digits with equal significance and update all digits sequentially

[1]We modified Ambit's B-group mapping [24] so address B11 activates T0, T1, and DCC0. This does not affect prior operations as B11 was unused.

from the LSD, resolving carries as we move to the most significant digit (MSD). To add an input $x$ to a counter, the unit increment process must be repeated $D + \sum_{i=1}^{D} d_i$ times, where $d_i$ is the value of the $i$-th digit of $x$ (represented in base $2n$). The summation represents the total number of *unit increments* triggered by all digits of the input, while the outer $D$ term accounts for the unit increments required for preventing overflow (*carry rippling*). This sequential digit increment remains costly, motivating a series of optimizations in Sec. V-D to reduce the number of CIM operations.

**Overflow Detection and Carry Rippling.** In multi-digit counters, an overflow in the current digit triggers an increment in the next higher digit. A detected overflow in the LSD generates a carry ($O_{next}$) to the next significant *digit*, i.e., LSD $\rightarrow$ LSD + 1, which potentially ripples through to the MSD, depending on the counters state. A naïve implementation of *carry rippling* would fully resolve detected overflows in the LSD, propagating carries through all digits before proceeding to the next increment.[2] This approach is inefficient since an overflow check after every increment is unnecessary, particularly for unit and small-value increments. To optimize this critical path operation, we use one dedicated $O_{next}$ row per digit to retain the pending overflow until it is unavoidable to resolve it (see Sec. V-D2).

**Decrements.** For negative inputs, the counter is decremented through *backward shifts* and *inverted feed-forward*. The *underflow* detection mechanism is similar to overflow except the MSB transitions from zero to one and the $O_{next}$ bit is used to decrement the next significant digit accordingly. Outstanding overflows or underflows must be resolved before switching from increment to decrement and vice versa, or a row representing a sign-bit of overflow must be allocated, $O_{sign}$.

### D. Optimized Counter Design

Counters with *unit increments* and *digit-wise carry rippling* are more costly than directly implementing ripple-carry addition (Fig. 8a). In this section we propose scheduling optimizations to significantly improve all counters' performance.

*1) Variable-Step (k-ary) Increment:* This section presents an optimization that enables increments by $k$, where $1 \leq k \leq 2n-1$, with the same number of memory operations as a unit increment; increment by one and increment by $k$ have the same latency. The cyclic property is preserved, meaning the JC state rolls back to 0 when the sum exceeds the counter capacity ($2n-1$). For instance, a 5-bit JC can transition directly from $\texttt{10000}(1) \rightarrow \texttt{00111}(7)$ and $\texttt{00111}(7) \rightarrow \texttt{11100}(3)$ when incrementing $k = 6$. Overflow detection indicates whether an increment results in an overflow (Sec. V-B).

For any $k$-ary increment, while the number of fundamental steps (*forward shift* and *inverted feedback*) remain the same as in unit increment, the shifting patterns are different. Fig. 7 illustrates all patterns for achieving a direct state transition from any radix-10 JC's value to its value incremented by $k$. For

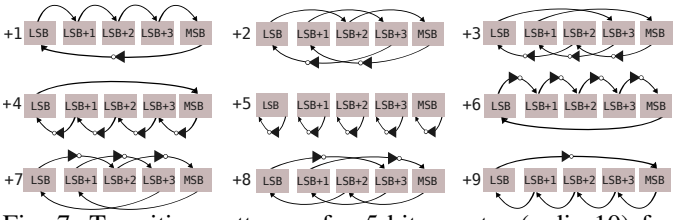
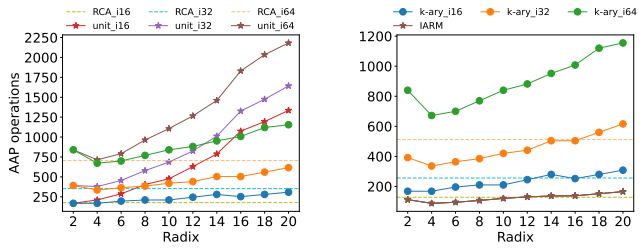



Fig. 7: Transition patterns of a 5-bit counter (radix-10) for incrementing any value between 1 and 9.




(a) Unit vs. $k$-ary increment (b) $k$-ary only vs. IARM

Fig. 8: Masked addition performance for unit, k-ary, carry rippling minization (IARM), and ripple-carry adder (RCA)

instance, an increment by +2, equivalent to applying the unit increment (+1) twice, is simplified into three *forward shifts* and two *inverted feedback* steps. The same principle applies to all patterns in Fig. 7. These transition patterns correspond to $k$ distinct $\mu$Programs, yet all require the same number of CIM (i.e., AAP) commands.

Algo. 1 generalizes the transition pattern generation, illustrating how each counter bit is updated depending on the value of $k$ with respect to $n$. The bit manipulation depends on whether the increment amount is less than or greater than $n$. If less than or equal to $n$, Lines 2-7 conditionally update the counter state using a mask $m$ and calculate the digit overflow accordingly. This process is illustrated by the increments from 1 to 5 in Fig. 7. Line 3 corresponds to the upper arrows, while Line 5 represents the lower arrows with a NOT operation. Similarly, for increments from 6 to 9, Line 12 corresponds to the upper arrows, and Line 10 represents the arrows with a NOT operation. With $k$-ary transitions, counting requires $2 \cdot (7n+7)$ operations per *input digit*, as each $k$-ary increment may propagate a carry, leading to an additional *carry rippling* command sequence. This occurs because $k$-ary transitions do not take a carry input, making carry propagation a separate operation. Next, we present a method to minimize the cost of this cascading effect in counters.

Fig. 8a compares the average number of AAP operations required for different bases when accumulating (uniform distribution) of 8-bit inputs on counters with capacities equivalent to 16, 32, and 64-bit ints.[3] $k$-ary counting provides a 2–6$\times$ reduction in CIM operations over unary counting for varying-radix counters.

*2) Input-Aware Rippling Minimization:* Optimizing the $k$-ary increments within a digit boosts performance, but the

[2] *Digit-wise carry ripple:* unit increment to the next higher digit using $O_{next}$ as a mask.

[3] A $D$-digit counter with $n$ bits per digit has a capacity of $(2n)^D$. Counters are sized to meet or exceed the capacity of binary integers by adding digits.

**Algorithm 1:** Variable-step increment for $n$-bit JC

**Input:** Johnson counter $C \leftarrow [b_0, \ldots, b_{n-1}]$, mask $m$, increment amount $k$

1 **if** $k \leq n$ **then**
2  **for** $i \leftarrow n-1$ **downto** $k$ **do**
3  $b'_i \leftarrow (\overline{m} \wedge b_i) \vee (m \wedge b_{i-k})$; // Forward Shift
4  **for** $i \leftarrow 0$ **upto** $k-1$ **do**
5  $b'_i \leftarrow (\overline{m} \wedge b_i) \vee (m \wedge \overline{b_{n-k+i}})$; // Inverted Feedback
6  $O'_{\text{next}} \leftarrow O_{\text{next}} \vee (b_{n-1} \wedge \overline{b'_{n-1}})$; // Overflow Checking
7 **else**
8  $k \leftarrow k-n$;
9  **for** $i \leftarrow n-1$ **downto** $k$ **do**
10  $b'_i \leftarrow (\overline{m} \wedge b_i) \vee (m \wedge \overline{b_{i-k}})$; // Inverted Feedback
11  **for** $i \leftarrow 0$ **upto** $k-1$ **do**
12  $b'_i \leftarrow (\overline{m} \wedge b_i) \vee (m \wedge b_{n-k+i})$; // Forward Shift
13  $O'_{\text{next}} \leftarrow O_{\text{next}} \vee (b_{n-1} \vee \overline{b'_{n-1}}) \wedge m$; // Overflow Checking

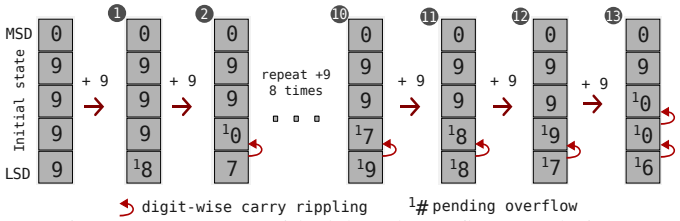



Fig. 9: Increments with delayed overflow resolution.

full carry propagation between digits remains a performance bottleneck. This section introduces Input-Aware Rippling Minimization (IARM), a run-time mechanism that postpones carry propagation to higher-order digits. Recall, each digit is augmented with an additional flag bit, $O_{next}$. Because this signals when there is a carry to propagate to the next digit, it increases the effective range of a digit from $2n-1$ to $4n-1$ unique values. Thus, even when a value exceeds $2n-1$, we do not trigger carry handling and delay it until a subsequent increment causes the counter to exceed $4n-1$. Furthermore, when a carry operation *is* performed, the digits may only store up to $2n-1$. Meaning, a minimum of $2n+1$ further increment is required before it is *possible* to exceed the $4n-1$ total capacity.

Based on this observation, IARM selectively triggers *carry rippling* based on increment history. Because IARM is oblivious of the masks stored in memory, it must presume each increment is applied to some counter. In other words, IARM must consider the worst-case increment behavior across all counters to maintain computational correctness. IARM implements a *virtual counter*, which increments *all* input values, to track the worst-case state of any counter. This virtual counter controls how long carry rippling can be delayed and issues commands just before one of its digit exceeds $4n-1$.

In Fig. 9 we explain an example of how IARM avoids overflow ripple operations where the counter has already been initialized to 9999. Presuming a radix-10 ($n=5$) system with $\geq 5$ digit counters, with traditional rippling, even an increment of 1 would cause a ripple effect from the lowest digit to the fifth digit. The baseline approach must presume this type of ripple is possible in some counter at any time requiring ripple propagation in all digits for every increment. In contrast, consider a worst-case series of increments (by 9 for radix-10) in IARM. Digits represented as $^1$# indicate digits storing values between $2n$ and $4n-1$ or between $^1$0 and $^1$9 for $n=5$. In step ❶ no carry resolution is necessary because the lowest digit stores $^1$8 and the entire counter is 999$^1$8. A second increment by 9 to step ❷ does cause a ripple increment to the next digit, but then stops there resulting in 99$^1$07. If we continue to add values of 9, we will eventually get to 99$^1$7$^1$9 in step ❿, 99$^1$8$^1$8 in step ⓫ and 99$^1$9$^1$7 in step ⓬ prior to rippling beyond the tens digit in step ⓭ to 9$^1$0$^1$0$^1$6. In our target applications, the increment's digit range tends to be much smaller than the counters' range, thus benefiting from IARM and reducing rippling.

Fig. 8b demonstrates the performance improvements from IARM over $k$-ary increment (without early termination). As our mechanism is solely input-dependent (determined by the number of non-zero digits in one operand), varying counter capacity does not affect its operation count, as reflected by the single IARM curve, invariant of counter capacity. IARM provides the fewest operation implementation over all other approaches, particularly for radices 4–8. [4]

## VI. Count2Multiply Implementation

### A. Execution Model and System Integration

Recall the Matrix-vector multiplication example from Fig. 1. Count2Multiply employs a *broadcast and accumulate* execution model, where a *host-side program* converts an input stream into AAP/AP command sequences representing *increments*, as illustrated in Fig. 10 and formalized in Algo. 2. To generate these memory commands, i.e., ❶..❸ in Fig. 10, the host-side program first reads elements of $\mathbb{X}$ from the memory (Lines 3–7). After converting $X_i$ from binary to the counter radix (Lines 8–12), ❷ an AAP/AP sequence for each digit of $X_i$—based on the appropriate $\mu$Program (Line 18; Fig. 6b)—is selected from preconstructed assembly macros. The optimized CIM sequence is generated offline using Majority synthesis [48]. Next ❸, this $\mu$Program is issued by the MC to one or many memory subarrays for parallel operation. An example of this process from Fig. 6b: ❶ "0b00101101" is read from memory and unpacked into the digits "45" (radix-10). ❷ For each digit, we use a $k$-ary increment $\mu$Program with the row addresses (counter) of the corresponding digit position—adding "5" to the ten's place and "4" to the hundred's place. Essentially, the input digit value determines the composition of *forward shifts* and *inverted feedback* (Sec. V-D1) as well as the row addresses for the corresponding counter-digit.

❸ These $\mu$Programs are then converted into a memory command sequence (i.e., ACT/PRE), which the MC broadcasts to the memory chips. This enables selective incrementing of memory columns at destination addresses ($\mathbb{Y}$ values stored as *counters* in memory) based on bit masks ($\mathbb{Z}$). In doing so, Count2Multiply updates only the necessary digits of $\mathbb{Y}$ to maintain computational accuracy, ultimately implementing the *early termination* of carry propagation among digits, consequently avoiding useless operations on high-order digits.

**IARM Integration.** The IARM mechanism (Sec. V-D2) is also implemented on the host side. Each $X_i$ value is first accumulated in a software-emulated *virtual counter* before

[4] Note that radix-4 incurs no storage overhead over binary; the $O_{next}$ signals are only relevant during counting and otherwise have no effect.

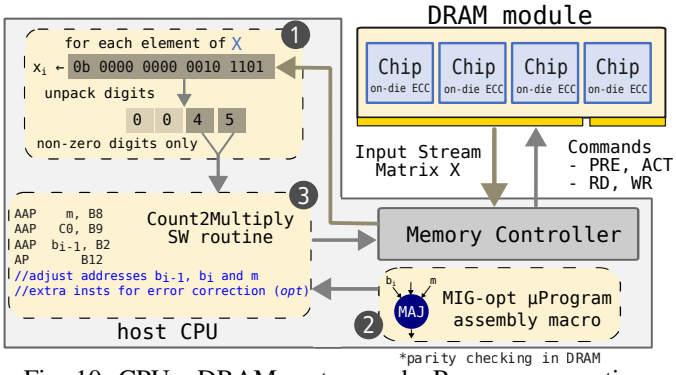



Fig. 10: CPU - DRAM system and $\mu$Program generation.

**Algorithm 2:** COUNT2MULTIPLY: host-side SW routine.

```
1  Inputs: tensor 𝕏, digit radix r, address of counters JC_addr, address of masks m_addr
2  virtual_counter ← INIT(0); //Only one multi-digit counter in the host to accumulate all values of x
3  for all x ∈ 𝕏 do
4      x ← BINARY2BASE(x, r); //Optional. This is called only if input 𝕏 is binary encoded
5      for all i = 0 to |x| − 1 do
6          if x[i] ≠ 0 then
7              INCREMENTDIGIT(virtual_counter, x[i], i, r); //It only increments non-zero digits
8  function BINARY2BASE(n, r):
9      digits ← empty list;
10     while n > 0 do
11         append (n mod r) to digits;   n ← ⌊n/r⌋;
12     return digits;
13 procedure INCREMENTDIGIT(VC, digit_incr, digit_pos, r):
14     incr ← digit_incr;  j ← digit_pos;
15     while j < |VC| do
16         sum = VC[j].digit + incr;  VC[j].digit ← sum mod r; //Update virtual counter
17         ISSUE_CIM(μProgram[incr], JC_addr[j], m_addr); //Populate and issue μProgram
18         if sum ≥ r then
19             if VC[j].O_next then
20                 VC[j].O_next ← False;
21                 j ← j + 1;  incr ← 2;  continue; //Increment +2 if it overflows twice
22             VC[j].O_next ← True;
23         break;
```

broadcasting the $k$-ary *increments* to memory, as illustrated in Algo. 2. If the next increment command could possibly cause a double overflow in any of the counters, an overflow instruction is prepended to the guilty increment command (Lines 21–22). The guilty increment command proceeds only after the current overflow bit has been cleared, preventing double overflow and ensuring correctness. Because the IARM mechanism runs on the host side (during conversion of input to commands), it is transparent to the CIM side, which only see a stream of deterministic instructions from the host.

**System Integration.** Fig. 10 illustrates a system where the CPU executes Count2Multiply's routine, unpacking inputs and populating $\mu$Programs with designated rows to trigger counting directly in memory. However, Count2Multiply is also compatible with non-CPU-based systems. For example, in FCDRAM [15], an FPGA can serve as the MC, orchestrating $\mu$Program execution. Alternatively, a specialized control unit integrated into the MC, like in SIMDRAM [21], is also suitable for implementing the masked matrix accumulation program, thus independently populating $\mu$Programs for DRAM execution. The overhead of dynamically building $\mu$Programs is negligible: (i) it involves only a few arithmetic instructions per input $\mathbb{X}$ (shown in Algo. 2), and (ii) the `AAP`/`AP` processing rate of the DRAM module is generally much lower than the $\mu$Programs generation on the host side, even when considering a single-core processor (see Tab. II). Note that a *conversion step* from Johnson to binary encoding is unnecessary when the matrix multiplication is followed by a ReLU, and the output is directly reused as input to the next matrix multiplication (Algo. 2, Line 4). Moreover, for integration with accelerator designs based on LUT-based operators (e.g., softmax [50]), the Johnson encoding can be seamlessly incorporated into LUTs, thereby avoiding explicit format conversion. A detailed exploration of this integration is left to future work.

### B. Kernels Accelerated by Count2Multiply

**Integer-Vector Binary-Matrix Operations:** Since counters increment a single input at a time, vector-matrix multiplications is reinterpreted as *masked matrix accumulations*. In Fig. 1a, we illustrate how vector-matrix multiplication $\vec{Y} = \vec{X}.\mathbb{Z}$ is formed by computing $\vec{Y} = \sum_{i=1}^{K} X_i.\vec{Z}_i$, wherein elements $(X_i)$ of the integer-vector $\vec{X}$ are accumulated into the vector $\vec{Y}$, as predicated by the mask rows $(\vec{Z}_i)$ in $\mathbb{Z}$.[5] Here, $\vec{Y}$ is stored in memory as high-radix *counters*, $\mathbb{Z}$ is stored in memory as binary masks, while $\vec{X}$ is an external input processed by the host CPU and issued by the MC (Sec. VI-A). This approach draws inspiration from the sum of outer products to ultimately implement an *integer-binary* matrix multiplication through *broadcast* and *accumulation*.

**Integer-Matrix Binary-Matrix Operations:** Vector-matrix multiplication naturally generalizes to matrix-matrix multiplication when $M > 1$,[5] as shown in Fig. 1a. Each row $\vec{Y}_o$ of the output matrix $\mathbb{Y}$ is computed independently as $\vec{Y}_o = \sum_{i=1}^{K} X_{oi} \cdot \vec{Z}_i$, for $o = 1, \ldots, M$, where the matrix $\mathbb{Z}$ is reused. As the rows of $\mathbb{Y}$ are computed sequentially, each computed *matrix* row can either be moved to a different subarray or used immediately. Copying the matrix row requires copying the memory rows dedicated to the high-radix counters to another subarray. Recall these are memory rows to store the counters from the D-group in Fig. 1b. These memory rows can be reused for accumulation of the new matrix row for $\mathbb{Y}$. This eliminates the need for dedicated counters for a specific row of $\mathbb{Y}$ within a single subarray and avoids the higher cost of copying the many more rows storing mask data, i.e., matrix $\mathbb{Z}$.

**Integer-Integer Matrix Operations:** Integer-binary matrix multiplication can be extended to *integer-integer* computation through *bit-slicing* matrix $\mathbb{Z}$. To support low-precision $p$-bit `int` and `uint` matrices, each value is decomposed into canonical signed digit (CSD) form [51], [52], requiring $2(p-1)$ or $p$ power-of-two-weighted binary masks, respectively. Each bit slice, representing a specific power-of-two significance and sign, maps to a row address in the memory subarray. For example, converting an `INT2` value ($p = 2$) in CSD form maps to two bit-sliced rows representing $+2^0$ and $-2^0$. The positive row encodes $0, +1$, and the negative row $0, -1$; when both are '1', they cancel out to zero. During accumulation, inputs from $\mathbb{X}$ are sequentially incremented for negative rows or decremented for positive ones. For larger bitwidths, the accumulation still uses counter array $\mathbb{Y}$ but uses power-of-two-scaled inputs based on the bit slice value. We can think of each row of $\mathbb{Z}$ requiring multiple bitsliced memory rows

[5] In the general case, matrix $\mathbb{X}$ has shape $[M \times K]$, $\mathbb{Z}$ $[K \times N]$, and $\mathbb{Y}$ $[M \times N]$.

$(+2^{p-2}, ..., +2^0$ and $-2^0, ..., -2^{p-2})$. The host-side routine scales the inputs based on the bit slice row address that indicates the power-of-two. The host can also use shifting for scaling as the bitsliced masks represent only powers-of-two, avoiding the need for or use of a CPU multiplier to generate $\mu$Programs. The host-side scaling allows accumulations for different bit slices to operate directly on a single counter row.

**Additional Tensor Style Operations:** Counting can also be leveraged to perform *shift-left*, *ReLU*, and *vector addition* of counters. For $c << i$ the counter value can be added to itself $i$ times. ReLU checks whether a counter is non-negative which is possible by checking $O_{sign}$. Adding two vectors of counters ($C_1$ and $C_2$) is done with both counters in memory. This approach involves using one of the $n$-bit counters (e.g., $C_1$) as masks for unit incrementing a second counter ($C_2$) in place, i.e., $C_2 \leftarrow C_1 + C_2$.

## VII. Fault Tolerance

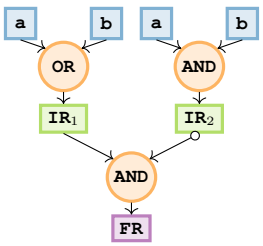
CIM operations that involve multi-row activations are more prone to faults than standard memory reads (Sec. II-C). Our goal is to detect these faults using the same ECC hardware already present in DRAM, rather than adding custom logic. To do this, we express each CIM operation in a form that is homomorphic with respect to ECC parity generation.

### A. System Overview

We consider an on-die ECC setup, wherein each DRAM die locally contains both parity bits and checking hardware. The only constraint is that the on-die setup must be able to communicate detection of errors to the memory controller. Prior works [53] propose exactly this in order to improve the overall system reliability. For example, XED [53] utilizes a unique "catch word" to communicate detected errors, but a single-bit signal can also be used. In our case, detected errors are sent to the memory controller and host so that CIM instruction flow can be reset to the most recent safe checkpoint. For our CIM fault protection scheme to be useful two conditions must be satisfied: 1) the on-die ECC must be triggered when a fault occurs during the computation of a result, 2) the on-die ECC must *not* be triggered when a fault *does not* occur during the computation of a result. In the next section, we go on to show how the property of *homomorphism* satisfies the latter condition and how *fault propagation* satisfies the former.

### B. Fault Protection Scheme Setup

Conventional ECCs such as Hamming, BCH, and Reed-Solomon codes are homomorphic over XOR, i.e., $\texttt{ECC}(a \oplus b) = \texttt{ECC}(a) \oplus \texttt{ECC}(b)$, but they are not homomorphic over AND or OR, e.g., $\texttt{ECC}(a \wedge b) \neq \texttt{ECC}(a) \wedge \texttt{ECC}(b)$. Therefore, if a CIM operation performs AND or OR directly, its output cannot be verified by ECC parity bits. To overcome this, we *embed* each CIM operation in XOR so that the final result can still be validated through ECC. The synthesis of an XOR function using AND, OR, and NOT gates is a two-step process, as shown in Fig 11a. The target operation to be protected–either OR or AND–produces $\texttt{IR}_1$ or $\texttt{IR}_2$, respectively. By additionally computing the complementary IR and combining both IRs to generate the final XOR result, i.e., $\texttt{FR} = \texttt{IR}_1 \wedge \texttt{IR}_2$, we perform ECC checks on FR to detect faults that may have occurred when computing $\texttt{IR}_1$, $\texttt{IR}_2$, and/or FR itself.




(a) XOR synthesis

| a | b | $\texttt{IR}_1$ | $\texttt{IR}_2$ | FR | XOR |
|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 1 | 1 | 0 | 1 | 1 |
| 1 | 0 | 1 | 0 | 1 | 1 |
| 1 | 1 | 1 | 1 | 0 | 0 |
| 0 | 0 | 1* | 0 | 1 | 0 |
| 1 | 1 | 0* | 0 | 0 | 0 |
| 0 | 1 | 0* | 1* | 0 | 1 |
| 0 | 1 | 0* | 1* | 1* | 1 |

(b) Fault examples

Fig. 11: Fault detection in OR/AND via XOR-based parity check.

Two key properties emerge from these operations: (i) most faults from an intermediate gate flip FR, invalidating the parity bits for FR (Fig. 11b); and (ii) faults that remain undetectable occur only for rare, data-dependent patterns with rates comparable to standard DRAM accesses–which we hereafter refer to as *unlikely*. The latter comes from the following observations: for the inputs (0,0,0) and (1,1,1), MAJ3 is reliable, since all cells pull the bitline uniformly, enabling safe sensing [16]. Recall, $\texttt{AND}(a,b) = \texttt{MAJ}(a,b,0)$ and $\texttt{OR}(a,b) = \texttt{MAJ}(a,b,1)$. Similarly, the NOT operation is functionally equivalent to RC, which is known to be *unlikely* to fault [15]. Even for a more fault-prone NOT implementation, we can express NOT as XOR with '1', so it can also be directly protected by ECC. We leverage this data-dependent fault pattern to ensure fault protection against all *likely* CIM faults.

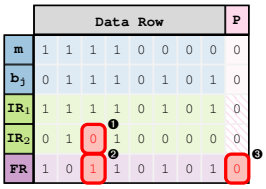
By exhaustively analyzing all possible inputs and combinations of *likely* and *unlikely* faults, we observe three possible fault modes: ① a single fault could occur in any result ($\texttt{IR}_1$,$\texttt{IR}_2$, or FR); ② two faults could occur–with both in $\texttt{IR}_1$ and $\texttt{IR}_2$ or with one in an IR and one in FR; ③ a fault occurs in all three computations. Figure 11b illustrates selected examples of each case through a single-bit comparison between FR and the expected XOR result, with faults marked in red*. The two possible outcomes of case ① are shown in the fifth and sixth rows of Fig. 11b. A fault occurs flipping $\texttt{IR}_1$ to '1'. The fault propagates to FR and is detected by parity check (i.e., $\texttt{FR} \neq \texttt{XOR}$). However, if a single *unlikely* fault occurs, our scheme cannot detect it as illustrated in the sixth row (fault in $\texttt{IR}_1$ but $\texttt{FR} = \texttt{XOR}$) making it a silent error. In case ②, shown in the seventh row of Fig. 11b, a fault occurs in both $\texttt{IR}_1$ and $\texttt{IR}_2$. Because neither of these faults were *unlikely*, the fault is propagated to FR and $\neq$ XOR, making it detected. In case ③, the scheme will not protect against faults in all three operations, shown in the eighth row of Fig. 11b. The only *likely* undetectable faults occur with one or both IRs being faulty *and* faulty computing of FR. Recomputing FR correctly reveals the fault.

### C. Fault Tolerant In-Memory Counting

In our $k$-ary counting method, we compute each new row by combining two rows ($b_i$ and $b_{i-k}$) masked with $m$ and

```
1 //Protected masking
2 AND bj,m,ir2 //ir2=bj&m
3 OR  bj,m,ir1 //ir1=bj|m
4 AND ir1,!ir2,fr //fr=ir1&!ir2
5 RC  ir2, t2 //t2=ir2
6 AND bi,!m,ir2 //ir2=bi&!m
7 OR bi,!m,ir1 //ir1=bi|!m
8 AND ir1,!ir2,fr //fr=ir1&!ir2
9 //Update row
10 OR t2,ir2,bi //bi=tj|ir2
```

(a) Generic protected *forward shift* $\mu$Program




(b) Fault propagation and detection in masking operation

Fig. 12: ECC scheme for in-memory counting.

$\overline{m}$ (Sec. V-A). Similarly, $\texttt{XOR}(a,b) = (a \wedge \overline{b}) \vee (\overline{a} \wedge b)$. Both functions share the following characteristics: (i) similar structure, being an OR of two AND terms, (ii) both represent a conditional and can be seen as multiplexers, with XOR being a specific case of 2-to-1 MUX with $b_{i-k} = \overline{a}$. This observation is useful because, in both functions, OR operates solely on mutually exclusive masked rows. Consequently, only the ANDs responsible for masking in the $k$-ary counting must be protected by embedding the mask in XOR as described in Sec. VII-B.

Fig. 12a details the generic increment $\mu$Program when using our protection scheme, omitting some RCs for the *forward shift* seen in Fig. 6b. In line 2, we compute one of the masking results to be protected: $b_j \wedge m$. In lines 3 and 4, the additional operations required to complete the XOR–and thus protect the FR–are performed. Line 5 copies the masking result to a temporary destination so that $\texttt{IR}_1$ and $\texttt{IR}_2$ can be reused in this example. Lines 6–8 compute and protect the masking operation $b_i \wedge \overline{m}$. Finally, line 10 updates the row with our protected results (line 10). The listing demonstrates the operation overhead to enable traditional ECC checks to detect errors in CIM results. Specifically, lines 3, 4, 7, 8 are the overhead required for protection when compared to the unprotected *forward shift* program in Fig. 6b.

Fig. 12b shows fault protection for an example row during in-memory masking. Initially, mask $m$ and JC bit $b_j$ rows are stored with a traditional parity bit, P. A fault occurs during the masking step which generates $\texttt{IR}_2$. This fault flips the bit stored in the position highlighted and labeled ❶. For the error detection scheme we compute $\texttt{IR}_1$ and FR, noting the flip from ❶ propagates to ❷. FR fails the parity check ❸ performed in ECC hardware. This requires repeating the computation. In Fig. 12a, this implies that if an ECC check fails after line 4 it is necessary to restart from line 2.

TABLE I: Fault tolerance varying recompute and fault rate.

| FR checks | 2 | | | 4 | | | 6 | | |
|---|---|---|---|---|---|---|---|---|---|
| Fault rate | $10^{-1}$ | $10^{-2}$ | $10^{-4}$ | $10^{-1}$ | $10^{-2}$ | $10^{-4}$ | $10^{-1}$ | $10^{-2}$ | $10^{-4}$ |
| Error rate | 1.4E-3 | 1.5E-6 | 1.5E-12 | 1.4E-5 | 1.5E-10 | *1.0E-20* | 1.4E-7 | 1.5E-14 | *1.0E-20* |
| Detect rate | 3.1E-1 | 3.5E-2 | 3.5E-4 | 4.4E-1 | 5.4E-2 | 5.5E-4 | 5.5E-1 | 7.3E-2 | 7.5E-4 |
| Ambit | $13n+16$ | | | $23n+26$ | | | $33n+36$ | | |

### D. Optimizations and Extensions

While performing the increment step with inversion $\overline{b_i} \wedge m$, the protection can be combined with that of $b_i \wedge \overline{m}$ by using De Morgan's Laws (they produce valid $\texttt{IR}_1$, $\texttt{IR}_2$ for XOR synthesis as in Fig. 11a). Because we can protect two masking steps through computing XOR when performing inverted feedback and the inverted feedback step accounts for half of all increment steps (Fig. 7), the net protection overhead can be reduced by 25%. Further, in Ambit [24] the special address groupings can be used to perform the NOT simultaneously with other operations. However, Ambit suffers seriously from the requirement that CIM operations be performed within the limited set of CIM-enabled addresses. Additionally, DRAM CIM solutions in general [15], [24] are destructive, requiring operand copying before each operation. For all schemes, we carefully consider potential optimizations and challenges in order to make a fair comparison. Tab. I shows operation counts under different protection levels.

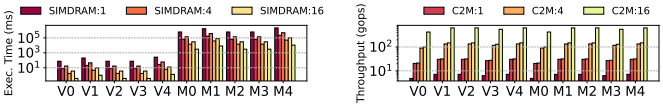



Fig. 13: Comparison of DRAM designs on ternary GEMV/GEMM from LLAMA and LLAMA-2 [7], [57], [58]

While the single error detection scheme can detect all possible single CIM faults, this may not suffice for higher fault rates, i.e., $10^{-1}$ – $10^{-4}$. Our protection scheme can be extended to two error detection and beyond. Additionally, the number of times we repeat the FR is configurable. Since all cases of *likely* faults going undetected require one of the faults to be in the FR (Sec. VII-B), we can simply repeat the FR computation to improve our fault tolerance. Tab. I compares protection schemes with varying repeats (`FR checks`) applied to different CIM fault rates. The *error* and *detect* rates report the per-bit probability of undetectable and detectable errors respectively. In our analysis, we assume that the *unlikely* CIM faults will exhibit a fault rate similar to that of a typical memory read operation (conservatively estimated at $10^{-20}$ for DRAM [54]), giving an upper bound on our fault tolerance level. The error rates bounded by DRAM fault rates in Tab. I are italicized.

TABLE II: Memory organization and architectural parameters

| | |
|---|---|
| DRAM | DDR5-4400, 1 channel, 1 rank, 8 devices, on-die ECC;<br>4Gb DRAM chip, 32 banks, 1 kB row size, 1024 rows per subarray;<br>Timing (ns): $t_{RC}$ = 46, $t_{RAS}$ = 32, $t_{RP}$ = 14.5, $t_{FAW}$ = 14.5 |
| | HBM2e, 16 channels, 8 Gb/channel, 8-Hi TSV stack;<br>32 banks/channel, 256-bit DQ width, 1 kB row size;<br>Timing (ns): $t_{RC}$ = 10.8, $t_{RAS}$ = 9.7, $t_{RP}$ = 4, $t_{FAW}$ = 8.6 |
| host CPU | in-order RISC-V @1.45 GHz, 64KB D-cache, 16KB I-cache [55], [56]<br>Memory Controller 8 kB row size, FR-FCFS scheduling |

## VIII. Experimental Setup and Results

To quantitatively evaluate the Count2Multiply approach, Ambit-style DRAM CIM is simulated by extending NVMain/RTSim [59], [60] with a cycle-level CIM simulation model. Our implementation of Ambit and SIMDRAM was rigorously validated against the results reported in [21], [24] and by MIMDRAM's simulator [61]. The architectural parameters of our memory organization are listed in Tab. II. Our setup follows commercial DRAM organization and timings.

### A. Configurations and Workloads

We consider the following in-DRAM designs and GPUs:

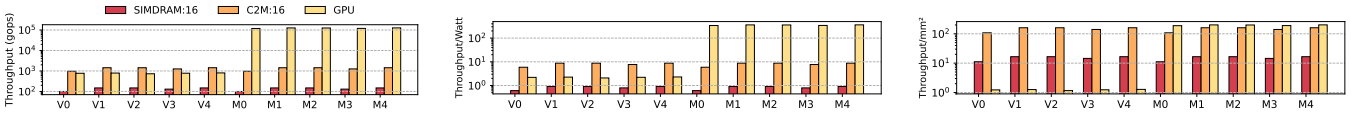



Fig. 14: Performance comparison for real ternary GEMM and GEMV [7], [57], [58]. DDR5-based SIMDRAM and C2M.

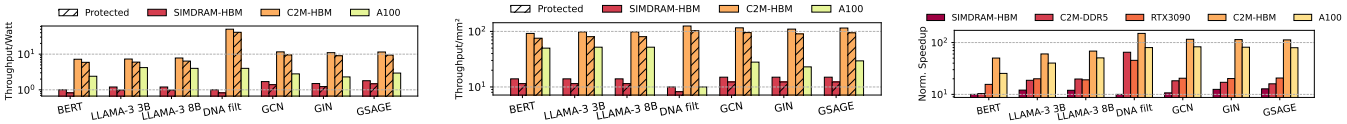



Fig. 15: Performance comparison of real-world workloads, including the protection scheme overhead.

TABLE III: GEMV and GEMM dimensions from [57], [58]

| Model | ID | M | N | K | ID | M | N | K |
|---|---|---|---|---|---|---|---|---|
| LLaMA | V0 | 1 | 22016 | 8192 | M0 | 8192 | 22016 | 8192 |
| LLaMA | V1 | 1 | 8192 | 22016 | M1 | 8192 | 8192 | 22016 |
| LLaMA-2 | V2 | 1 | 8192 | 8192 | M2 | 8192 | 8192 | 8192 |
| LLaMA-2 | V3 | 1 | 28672 | 8192 | M3 | 8192 | 28672 | 8192 |
| LLaMA-2 | V4 | 1 | 8192 | 28672 | M4 | 8192 | 8192 | 28672 |

- *SIMDRAM:X* RCA-based CIM design [21] using X banks.
- *GPU:* NVIDIA RTX 3090 Ti and A100. Each data point averages ten runs with a warm-up phase to avoid cold cache effects. GPU kernel performance and power reported with cudaEvents API and nvidia-smi, excluding data transfer. Die area is 628 $mm^2$ [62] for RTX and 826 $mm^2$ [63] for A100. For a fair comparison, BitBLAS [64] is employed in GEMV and GEMM implementations.
- *C2M:X* is DRAM-based Count2Multiply using X banks.

We evaluate Count2Multiply on the following workloads:

- *GEMV and GEMM*: We adopt the GEMM and GEMV shapes $(M, N, K)$ from Tab. III, based on [57], [58], assuming INT8-INT2 data types, where $\mathbb{X}$ is integer, $\mathbb{Z}$ is ternary, and $\mathbb{Y}$ is (high-radix) integer. These shapes represent the key computational loads in the models and serve as performance proxies.
- *Transformer inference* faces memory-bound challenges due to its large-scale data requirements. We evaluate all GEMV/GEMM operations in the attention layer of BERT and LLama-3 models. INT4 parameters [42] and sparse activations [45] are considered.
- *Pre-alignment filtering* is a memory-intensive step in DNA analysis, where the input genome is compared to a reference genome stored as bitvectors in memory [65]. Nucleotide repetitions in the *reads* of input genomes are represented as integers. For our evaluation, we use a human genome and a similar setup to prior work [66].
- *Graph Neural Networks* operate on graph-structured data (highly sparse) by aggregating and transforming features from a node's local neighborhood using the graph's connectivity. We evaluate three models (GCN, GIN, and GraphSAGE) on a node classification task using the Amazon dataset [67], [68]. All models are quantized to INT4.

### B. Design Space Exploration: LLM Kernels

In this section we evaluate Count2Multiply on tensor kernels from large language models LLaMA and LLaMA-2 for different metrics.

**Impact of DRAM Parallel Execution:** Fig. 13 presents a comparative analysis of the performance, i.e., latency and throughput, of only in-DRAM implementations with different subarray CIM parallelism for integer-ternary GEMV and GEMM workloads described in Tab. III. The evaluation uses an 8-bit signed integer input and radix-4 counters. All configurations assume an accumulation capacity of 64-bit integers to ensure computational precision.

Due to the sequential nature of RCA in *SIMDRAM*, *C2M* consistently outperforms *SIMDRAM* on all workloads and all system configurations. On average (geomean), *C2M* is 2× faster and delivers 1.15× higher throughput and throughput per Watt. This finding is in agreement with previous results obtained for a single addition (Fig. 8) and confirms that *C2M* maintains its performance gains in more complex kernels such as GEMV and GEMM.

We vary the number of banks from 1 to 16. With a single bank, latency is high, allowing one AAP every $t_{AAP} + t_{RRD}$. With 4 banks, commands can overlap across banks, though the fifth activation must still wait for the first to finish—bounded by $t_{AAP} + t_{RRD}$. At 16 banks, latency improves as the fifth activation is now constrained by $t_{FAW}$, which is shorter than $t_{AAP}$ ($t_{RAS}$ + $t_{RP}$ + 4).

**Comparison with GPU:** Fig. 14 presents throughput and throughput per Watt and area of *SIMDRAM*, *C2M*, and the GPU (RTX) baseline. As expected, with GPUs and BLAS routines being particularly designed and hand-optimized for GEMM, the CIM accelerators exhibit lower throughput than the GPU. Note that all results for in-DRAM designs use a single rank with one subarray per bank doing the computations. The results scale linearly with increasing the number of CIM subarrays and ranks. Further, we are using conservative estimates with a $t_{FAW}$ of 14.5 ns. All-bank activation, as suggested in prior work in the CIM domain [69], leads to superior throughput compared to powerful GPUs, as discussed in Sec. VIII-C.

**Impact of Sparsity on Performance:** Count2Multiply skips zero-value inputs (and zero digits from non-zero-value inputs), making it an ideal fit for sparse operations, which are common across various domains, including graph-based workloads, scientific computing, and deep learning, with sparsity ranging from 90% to 99.5% [70]–[72].

Fig. 16 compares latency and throughput of the GPU (RTX), SIMDRAM, and our 16-bank C2M configuration across varying sparsity levels (0% to 99.9%) for V0 and M0

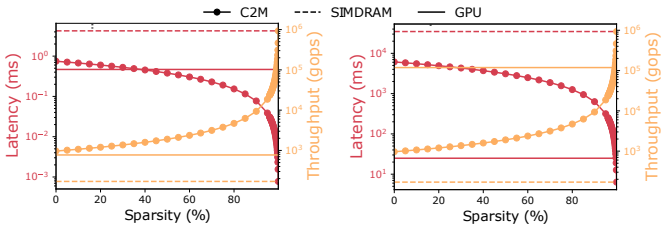
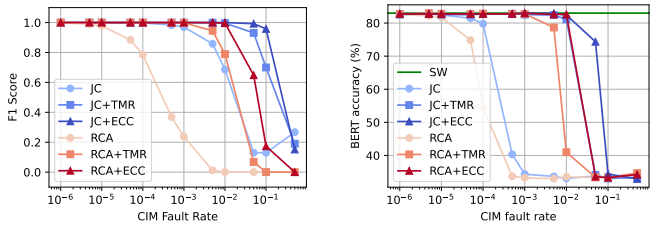



Fig. 16: Performance using sparse inputs: (left) Vector-Matrix Multiply (V0), (right) Matrix-Matrix Multiply (M0).




(a) DNA Filtering (b) BERT

Fig. 17: Accuracy comparison under CIM fault probability.

workloads (Tab. III). C2M outperforms SIMDRAM by orders of magnitude and matches or exceeds GPU latency beyond 40% sparsity in GEMV and 99.6% in GEMM. It also achieves higher throughput than the GPU in GEMV even for dense inputs, and in GEMM beyond 99.1% sparsity, despite the GPU leveraging optimized tensor cores and cuBLAS.

### C. Benchmark Analysis

This section analyses the impact of faults on applications' accuracy and the impact of fault tolerance on various metrics.

**Fault Tolerance Impact on Accuracy:** Count2Multiply, requiring significantly fewer CIM operations is less susceptible to accuracy degradation. Fig. 17 illustrates this in two applications – DNA filtering and a BERT model – using a generic MAJ-based RCA as a proxy for MAJ-based addition in CIM designs using DRAM [20]. As shown, Count2Multiply (*JC*) consistently achieves higher accuracy than RCA-based implementations across all fault probabilities and remains reliable under more severe fault conditions.

*JC* with protected schemes (+*ECC* or +*TMR*) outperforms RCA due to its lower fault susceptibility from early carry termination. The MAJ-based *ECC* method (Sec. VII) also applies to *RCA* designs and renders more reliable results than using *TMR* for both applications. Notably, *TMR*, the fault-tolerance approach used in SOTA [16], performs worse than *ECC* with a single repetition.

In DNA filtering, performance degrades more gradually, while BERT exhibits a sharp accuracy drop in task classification on the GLUE dataset [73] due to its many layers and greater error propagation. For DNA filtering, an F1 score over 0.9 with a fault rate of 10% is remarkable and acceptable for less sensitive downstream tasks such as phylogenetic analysis and genome assembly. For the BERT classification task (i.e., MNLI) >70% accuracy is considered acceptable [73] and is easily achieved for fault rates up to 5%.

**Fault Tolerance Impact on Performance:** Fig. 15 compares *C2M* with and without protection with the GPU and SIMDRAM. Protection increases latency by increasing operations for fault *detection* ($7n + 7 \rightarrow 13n + 16$), and by requiring *recomputation* upon fault detection (see Tab.I). *TMR* avoids recomputation but triplicates and votes over CIM operations ($\times 4$ operations) and suffers from high error rates (Sec. VIII-C). Bars labeled *Protected* in Fig.15 reflect the protection overhead in terms of GOPS/W and GOPS/$\mathrm{mm}^2$.

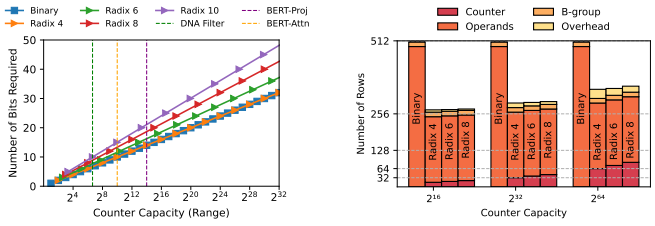
Estimating protection overhead requires the memory's fault rate and the number of repetitions to be performed. In Fig. 15, we consider an inherent fault rate of $10^{-4}$ and 1 round of `FC` computation (repeats=1). This translates to a detected fault rate of $3.5 \times 10^{-4}$ per bit (Tab. I) and 0.16 per 512-bit row. As shown in Fig. 15, the correction overhead in DRAM designs is 19.6%. This could be improved if a more fine-grained CIM control could repeat only columns with potential errors, decoupling a counter's need to recompute from its distant neighbors—similar to differential write for NVMs [74].

Fig. 15 also shows normalized speedup over SIMDRAM-DDR5. Across all workloads, C2M significantly outperforms SIMDRAM and both GPUs when implemented in HBM. On average, C2M-HBM achieves up to 2.0× speedup over A100, while offering the best energy efficiency, with GOPS/Watt ranging from 1.7× to 12.5×. On average, C2M delivers a 4.4× higher GOPS/$\mathrm{mm}^2$ than A100.

### D. Overhead Analysis

**Storage Capacity Analysis:** While binary (radix-2) encoding offers optimal storage efficiency, higher-radix counters deliver better performance with only moderate storage overhead (Fig. 8). Fig.18a compares the bit requirements for various counters to meet application-specific capacity needs. Notably, the absolute bit difference does not prohibit higher-radix use. For instance, DNA filtering needs a capacity of 100, achievable with 7 bits in binary or 10 bits in radix-10. BERT's projection and attention layers require capacities to accumulate 64 and 792 ternary products, respectively. Our chosen radix-4 counters offer the same bit-level storage density as binary. Fig. 18b shows row usage for GEMV in SIMDRAM (binary) versus C2M (radix-4/6/8). SIMDRAM must co-locate operands and output within the same subarray, whereas C2M stores only the mask (one operand) and counter. Despite moderate overhead from additional rows, higher radix enables denser operand placement and reduces costly inter-subarray operand transfers, improving row utilization.

**Runtime and Code Footprint:** We validate the suitability of the host CPU to execute the Count2Multiply SW under worst-case workloads–including JC conversion to non-power-of-two bases and dense inputs–on gem5. This evaluation ensures that the speed of $\mu$Program generation is sufficient to sustain parallel execution of matrix tiles across multiple memory banks. By interleaving Count2Multiply's operations, the host CPU can generate and issue one `AAP` command every 8 ns, which is sufficient to fully exploit bank-level parallelism in Count2Multiply. This throughput is achieved




(a) JC capacity in workloads (b) Array utilization

Fig. 18: JC storage capacity and subarray row usage for SIMDRAM (binary) and C2M (radix-4/6/8)

because the fixed cost of Algo. 1 for a single input is amortized over multiple AAP commands for *forward shifts*. Also, the code footprint of $\mu$Programs and the Count2Multiply SW is negligible: each $\mu$Program comprises 20 AAP/AP commands (42 with protection), each command implemented as a single instruction, the size of which depends on the host's ISA. Storing 4 $\mu$Programs (i.e., for radix-4) in the host cache requires at most 2KB, assuming 64-bit instructions. In total, Algo. 2 code requires ∼4KB. In terms of energy, the designated CPU's power consumption is as low as ∼60 m W [55]. This corresponds to only about 1.2% of the power drawn by a DDR5 module (∼5 W per DIMM under typical workloads), indicating that the energy overhead of host side is negligible.

**Johnson-binary Conversions:** As discussed in Sec. VI-A, no encoding conversion is needed when matrix multiplications are followed by ReLU or LUT-based operations. When conversion is necessary, the binary-to-base and vice-versa conversions for power-of-two radices incur a modest overhead of 30–60 CPU cycles (i.e., x86) per 8–16-bit integer.

## IX. Conclusions

We present Count2Multiply, a digital-CIM framework for integer-binary and integer-integer matrix multiplications using high-radix, parallel counting with bitwise logic operations. Count2Multiply considers reliability a first-class metric and presents a fault tolerance method that is compatible with existing ECCs. Unlike prior in-DRAM designs, Count2Multiply accelerates sparse matrix operations by skipping zeros, with substantial improvements over prior DRAM CIM and GPU baselines while providing dramatic energy improvements and better performance per chip area.

## Acknowledgment

The authors acknowledge the financial support by the Federal Ministry of Research, Technology and Space of Germany and by Sächsische Staatsministerium für Wissenschaft, Kultur und Tourismus in the programme Center of Excellence for AI-research "Center for Scalable Data Analytics and Artificial Intelligence Dresden/Leipzig", project identification number: ScaDS.AI, and funded by the German Research Council (DFG) through the HetCIM project (502388442), the CRC/TRR 404-Active 3D (528378584), and NSF Awards 2534062, 2450596, 2511445, 2328972.

## References

[1] A. Boroumand, S. Ghose, Y. Kim, R. Ausavarungnirun, E. Shiu, R. Thakur, D. Kim, A. Kuusela, A. Knies, and P. s. Ranganathan, "Google workloads for consumer devices: Mitigating data movement bottlenecks," in *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*, 2018, pp. 316–331.

[2] F. Aguirre, A. Sebastian, M. Le Gallo, W. Song, T. Wang, J. J. Yang, W. Lu, M.-F. Chang, D. Ielmini, and Y. Yang, "Hardware implementation of memristor-based artificial neural networks," *Nature Communications*, vol. 15, no. 1, p. 1974, 2024.

[3] H. Ahmed, P. C. Santos, J. P. Lima, R. F. Moura, M. A. Alves, A. C. Beck, and L. Carro, "A compiler for automatic selection of suitable processing-in-memory instructions," in *2019 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 2019, pp. 564–569.

[4] H. Farzaneh, J. P. C. de Lima, M. Li, A. A. Khan, X. S. Hu, and J. Castrillon, "C4cam: A compiler for cam-based in-memory accelerators," in *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'24), Volume 3*, ser. ASPLOS '24. New York, NY, USA: Association for Computing Machinery, May 2024, pp. 164–177. [Online]. Available: https://arxiv.org/abs/2309.06418

[5] F. Li, B. Liu, X. Wang, B. Zhang, and J. Yan, "Ternary weight networks," *arXiv preprint arXiv:1605.04711*, 2016.

[6] Y. Li, X. Dong, S. Q. Zhang, H. Bai, Y. Chen, and W. Wang, "Rtn: Reparameterized ternary network," in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 34, no. 04, 2020, pp. 4780–4787.

[7] S. Ma, H. Wang, L. Ma, L. Wang, W. Wang, S. Huang, L. Dong, R. Wang, J. Xue, and F. Wei, "The era of 1-bit llms: All large language models are in 1.58 bits," *arXiv preprint arXiv:2402.17764*, 2024.

[8] J. Wang, Y. Wang, Z. Yang, L. Yang, and Y. Guo, "Bi-gcn: Binary graph convolutional network," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2021, pp. 1561–1570.

[9] J. P. C. de Lima and L. Carro, "Quantization-aware in-situ training for reliable and accurate edge ai," in *2022 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 2022, pp. 1497–1502.

[10] A. A. Khan, J. P. C. De Lima, H. Farzaneh, and J. Castrillon, "The landscape of compute-near-memory and compute-in-memory: A research and commercial overview," *arXiv preprint arXiv:2401.14428*, 2024.

[11] S. Kvatinsky, D. Belousov, S. Liman, G. Satat, N. Wald, E. G. Friedman, A. Kolodny, and U. C. Weiser, "Magic—memristor-aided logic," *IEEE Transactions on Circuits and Systems II: Express Briefs*, vol. 61, no. 11, pp. 895–899, 2014.

[12] S. Hamdioui, L. Xie, H. A. Du Nguyen, M. Taouil, K. Bertels, H. Corporaal, H. Jiao, F. Catthoor, D. Wouters, and L. s. Eike, "Memristor based computation-in-memory architecture for data-intensive applications," in *2015 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 2015, pp. 1718–1725.

[13] V. Seshadri, K. Hsieh, A. Boroum, D. Lee, M. A. Kozuch, O. Mutlu, P. B. Gibbons, and T. C. Mowry, "Fast bulk bitwise and and or in dram," *IEEE Computer Architecture Letters*, vol. 14, no. 2, pp. 127–131, 2015.

[14] F. Gao, G. Tziantzioulis, and D. Wentzlaff, "Computedram: In-memory compute using off-the-shelf drams," in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO '52. New York, NY, USA: Association for Computing Machinery, 2019, p. 100–113.

[15] I. E. Yuksel, Y. C. Tugrul, A. Olgun, F. N. Bostancı, A. G. Yağlıkçı, G. F. Oliveira, H. Luo, J. Gómez-Luna, M. Sadrosadati, and O. Mutlu, "Functionally-complete boolean logic in real dram chips: Experimental characterization and analysis," in *2024 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, 2024, pp. 280–296.

[16] I. E. Yuksel, Y. C. Tugrul, F. Bostanci, A. G. Yaglikci, A. Olgun, G. F. Oliveira, M. Soysal, H. Luo, J. G. Luna, and M. s. Sadrosadati, "Pulsar: Simultaneous many-row activation for reliable and high-performance computing in off-the-shelf dram chips," *arXiv preprint arXiv:2312.02880*, 2023.

[17] Y. Lv, B. R. Zink, R. P. Bloom, H. Cılasun, P. Khanal, S. Resch, Z. Chowdhury, A. Habiboglu, W. Wang, and S. S. s. Sapatnekar, "Experimental demonstration of magnetic tunnel junction-based computational random-access memory," *npj Unconventional Computing*, vol. 1, no. 1, p. 3, 2024.

[18] B. Hoffer, V. Rana, S. Menzel, R. Waser, and S. Kvatinsky, "Experimental demonstration of memristor-aided logic (magic) using valence change memory (vcm)," *IEEE Transactions on Electron Devices*, vol. 67, no. 8, pp. 3115–3122, 2020.

[19] H. Padberg, A. Regev, G. Piccolboni, A. Bricalli, G. Molas, J. F. Nodin, and S. Kvatinsky, "Experimental demonstration of non-stateful in-memory logic with 1t1r oxram valence change mechanism memristors," *IEEE Transactions on Circuits and Systems II: Express Briefs*, vol. 71, no. 1, pp. 395–399, 2023.

[20] L. Brackmann, T. Ziegler, D. J. Wouters, and S. Menzel, "Experimental verification and evaluation of non-stateful logic gates in resistive ram," *IEEE Transactions on Circuits and Systems I: Regular Papers*, 2024.

[21] N. Hajinazar, G. F. Oliveira, S. Gregorio, J. a. D. Ferreira, N. M. Ghiasi, M. Patel, M. Alser, S. Ghose, J. Gómez-Luna, and O. Mutlu, "Simdram: a framework for bit-serial simd processing using dram," in *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '21. New York, NY, USA: Association for Computing Machinery, 2021, p. 329–345.

[22] G. F. Oliveira, M. Kabra, Y. Guo, K. Chen, A. G. Yağlıkçı, M. Soysal, M. Sadrosadati, J. O. Bueno, S. Ghose, and J. s. Gómez-Luna, "Proteus: Achieving high-performance processing-using-dram via dynamic precision bit-serial arithmetic," *arXiv preprint arXiv:2501.17466*, 2025.

[23] Q. Deng, L. Jiang, Y. Zhang, M. Zhang, and J. Yang, "Dracc: A dram based accelerator for accurate cnn inference," in *Proceedings of the 55th annual design automation conference*, 2018, pp. 1–6.

[24] V. Seshadri, D. Lee, T. Mullins, H. Hassan, A. Boroumand, J. Kim, M. A. Kozuch, O. Mutlu, P. B. Gibbons, and T. C. Mowry, "Ambit: In-memory accelerator for bulk bitwise operations using commodity dram technology," in *2017 50th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2017, pp. 273–287.

[25] X. Xin, Y. Zhang, and J. Yang, "Elp2im: Efficient and low power bitwise operation processing in dram," in *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2020, pp. 303–314.

[26] S. Roy, M. Ali, and A. Raghunathan, "Pim-dram: Accelerating machine learning workloads using processing in commodity dram," *IEEE Journal on Emerging and Selected Topics in Circuits and Systems*, vol. 11, no. 4, pp. 701–710, 2021.

[27] M. M. R. Islam and P. Stenström, "Characterization and exploitation of narrow-width loads: The narrow-width cache approach," in *Proceedings of the International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES)*. ACM, 2010, pp. 167–176.

[28] O. Ergin, O. Unsal, X. Vera, and A. Gonzalez, "Exploiting narrow values for soft error tolerance," *Computer Architecture Letters*, vol. 5, pp. 8–8, 2006.

[29] D. Brooks and M. Martonosi, "Dynamically exploiting narrow width operands to improve processor power and performance," in *Proceedings of the 5th International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 1999, pp. 13–22.

[30] O. Ergin, D. Balkan, G. Pekhimenko, and K. Ghose, "Register packing: Exploiting narrow-width operands for reducing register file pressure," in *Proceedings of the 37th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2004, pp. 304–315.

[31] M. Budiu, M. Sakr, and S. C. Goldstein, "Bitvalue inference: Detecting and exploiting narrow bitwidth computations," in *Euro-Par 2000 Parallel Processing*, ser. Lecture Notes in Computer Science, vol. 1900. Springer, 2000, pp. 969–979.

[32] P. R. Wilson, S. F. Kaplan, and Y. Smaragdakis, "The case for compressed caching in virtual memory systems," in *Proceedings of the USENIX Annual Technical Conference (USENIX ATC)*. USENIX Association, 1999, pp. 93–108.

[33] K. Chakrabarty, B. T. Murray, and V. Iyengar, "Built-in test pattern generation for high-performance circuits using twisted-ring counters," in *Proceedings 17th IEEE VLSI Test Symposium (Cat. No. PR00146)*. IEEE, 1999, pp. 22–27.

[34] W.-C. Lien, K.-J. Lee, T.-Y. Hsieh, and W.-L. Ang, "An efficient on-chip test generation scheme based on programmable and multiple twisted-ring counters," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 32, no. 8, pp. 1254–1264, 2013.

[35] S. Ollivier, S. Longofono, P. Dutta, J. Hu, S. Bhanja, and A. K. Jones, "CORUSCANT: Fast efficient processing-in-racetrack memories," in *55th IEEE/ACM International Symposium on Microarchitecture (MICRO)*, October 2022.

[36] V. Seshadri, Y. Kim, C. Fallin, D. Lee, R. Ausavarungnirun, G. Pekhimenko, Y. Luo, O. Mutlu, P. B. Gibbons, and M. A. s. Kozuch, "Rowclone: Fast and energy-efficient in-dram bulk data copy and initialization," in *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*, 2013, pp. 185–197.

[37] P. Brazzle, B. F. M. III, E. McKinney, P. Zhou, J. Hu, A. A. Khan, and A. K. Jones, "Towards error correction for computing in racetrack memory," 2024. [Online]. Available: https://arxiv.org/abs/2407.21661

[38] H. Cılasun, S. Resch, Z. I. Chowdhury, M. Zabihi, Y. Lv, B. Zink, J.-P. Wang, S. S. Sapatnekar, and U. R. Karpuzcu, "On error correction for nonvolatile processing-in-memory," in *2024 ACM/IEEE 51st Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2024, pp. 678–692.

[39] R. R. Johnson, "Electronic counter," US Patent US3 030 581A, 1958. [Online]. Available: https://patents.google.com/patent/US3030581A/en

[40] H. You, X. Chen, Y. Zhang, C. Li, S. Li, Z. Liu, Z. Wang, and Y. Lin, "Shiftaddnet: A hardware-inspired deep network," *Advances in Neural Information Processing Systems*, vol. 33, pp. 2771–2783, 2020.

[41] X. Peng, Y. Wang, and M.-C. Yang, "Chopper: A compiler infrastructure for programmable bit-serial simd processing using memory in dram," in *2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 2023, pp. 1275–1288.

[42] J. Chee, Y. Cai, V. Kuleshov, and C. M. De Sa, "Quip: 2-bit quantization of large language models with guarantees," *Advances in Neural Information Processing Systems*, vol. 36, 2024.

[43] J. P. C. de Lima, A. A. Khan, L. Carro, and J. Castrillon, "Full-stack optimization for cam-only dnn inference," in *Proceedings of the 2024 Design, Automation and Test in Europe Conference (DATE)*, ser. DATE'24. IEEE, Mar. 2024, pp. 1–6.

[44] O. Leitersdorf, D. Leitersdorf, J. Gal, M. Dahan, R. Ronen, and S. Kvatinsky, "Aritpim: High-throughput in-memory arithmetic," *IEEE Transactions on Emerging Topics in Computing*, vol. 11, no. 3, pp. 720–735, 2023.

[45] J. Liu, P. Ponnusamy, T. Cai, H. Guo, Y. Kim, and B. Athiwaratkun, "Training-free activation sparsity in large language models," *arXiv preprint arXiv:2408.14690*, 2024, submitted to ICLR 2025.

[46] H. Wang, S. Ma, R. Wang, and F. Wei, "QSparse: All Large Language Models can be Fully SparselyActivated," *arXiv preprint arXiv:2407.10969*, 2024, submitted to ICLR 2025.

[47] Y. Gao, Z. Zeng, D. Du, S. Cao, P. Zhou, J. Qi, J. Lai, H. K. So, T. Cao, F. Yang, and M. Yang, "SeerAttention: Learning Intrinsic Sparse Attention in Your LLMs," *arXiv preprint arXiv:2410.13276*, 2024, version updated February 2025; submitted to ICLR 2025.

[48] L. Amarú, P.-E. Gaillardon, and G. De Micheli, "Majority-inverter graph: A novel data-structure and algorithms for efficient logic optimization," in *Proceedings of the 51st Annual Design Automation Conference*, 2014, pp. 1–6.

[49] M. Soeken, H. Riener, W. Haaswijk, E. Testa, B. Schmitt, G. Meuli, F. Mozafari, S.-Y. Lee, A. Tempia Calvino, and G. Marakkalage, Dewmini Sudara De Micheli, "The EPFL logic synthesis libraries," Jun. 2022, arXiv:1805.05121v3.

[50] S. Liu, G. Tao, Y. Zou, D. Chow, Z. Fan, K. Lei, B. Pan, D. Sylvester, G. Kielian, and M. Saligane, "Consmax: Hardware-friendly alternative softmax with learnable parameters, 2024," *URL https://arxiv.org/abs/2402.10930*.

[51] A. Avizienis, "Signed-digit number representations for fast parallel arithmetic," *IRE Transactions on Electronic Computers*, vol. EC-10, no. 3, pp. 389–400, 1961.

[52] H. L. Garner, "Number systems and arithmetic," ser. Advances in Computers, F. L. Alt and M. Rubinoff, Eds. Elsevier, 1966, vol. 6, pp. 131–194.

[53] P. J. Nair, V. Sridharan, and M. K. Qureshi, "Xed: exposing on-die error detection information for strong memory reliability," *SIGARCH Comput. Archit. News*, vol. 44, no. 3, p. 341–353, Jun. 2016.

[54] B. Schroeder, E. Pinheiro, and W.-D. Weber, "Dram errors in the wild: a large-scale field study," *Commun. ACM*, vol. 54, no. 2, p. 100–107, Feb. 2011.

[55] D. Barbier, "Risc-v core ip products: An introduction to sifive risc-v core ip," https://cdn2.hubspot.net/hubfs/3020607/SiFive-RISCVCoreIP.pdf, Sep. 2017, accessed: 2025-10-21.

[56] J. Lowe-Power, A. M. Ahmad, A. Akram, M. Alian, R. Amslinger, M. Andreozzi, A. Armejach, N. Asmussen, B. Beckmann, S. Bharadwaj *et al.*, "The gem5 simulator: Version 20.0+," *arXiv preprint arXiv:2007.03152*, 2020.

[57] H. Touvron, T. Lavril, G. Izacard, X. Martinet, M.-A. Lachaux, T. Lacroix, B. Rozière, N. Goyal, E. Hambro, and F. s. Azhar, "Llama: Open and efficient foundation language models," *arXiv preprint arXiv:2302.13971*, 2023.

[58] H. Touvron, L. Martin, K. Stone, P. Albert, A. Almahairi, Y. Babaei, N. Bashlykov, S. Batra, P. Bhargava, and S. s. Bhosale, "Llama 2: Open foundation and fine-tuned chat models," *arXiv preprint arXiv:2307.09288*, 2023.

[59] M. Poremba, T. Zhang, and Y. Xie, "Nvmain 2.0: A user-friendly memory simulator to model (non-)volatile memory systems," *IEEE Computer Architecture Letters*, vol. 14, no. 2, pp. 140–143, July 2015.

[60] A. A. Khan, F. Hameed, R. Bläsing, S. Parkin, and J. Castrillon, "RTSim: A Cycle-Accurate Simulator for Racetrack Memories," *IEEE Computer Architecture Letters*, vol. 18, no. 1, pp. 43–46, Jan 2019.

[61] G. F. Oliveira, A. Olgun, A. G. Yağlıkçı, F. N. Bostancı, J. Gómez-Luna, S. Ghose, and O. Mutlu, "Mimdram: An end-to-end processing-using-dram system for high-throughput, energy-efficient and programmer-transparent multiple-instruction multiple-data computing," in *2024 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 2024, pp. 186–203.

[62] N. Corporation, "Nvidia geforce rtx 3090 whitepaper," https://www.nvidia.com/content/PDF/nvidia-ampere-ga-102-gpu-architecture-whitepaper-v2.pdf, 2020, accessed: 2024-08-01.

[63] ——, "Nvidia a100 tensor core gpu architecture," https://resources.nvidia.com/en-us-tensor-core/nvidia-ampere-architecture-whitepaper, 2020, accessed: 2025-08-01.

[64] L. Wang, L. Ma, S. Cao, Q. Zhang, J. Xue, Y. Shi, N. Zheng, Z. Miao, F. Yang, T. Cao, Y. Yang, and M. Yang, "Ladder: Enabling efficient low-precision deep learning computing through hardware-aware tensor transformation," in *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*, 2024.

[65] J. Kim, D. Senol Cali, H. Xin, D. Lee, S. Ghose, M. Alser, H. Hassan, O. Ergin, C. Alkan, and O. Mutlu, "GRIM-Filter: Fast Seed Location Filtering in DNA Read Mapping using Processing-in-memory Technologies," *BMC Genomics*, vol. 19, no. 2, 2018.

[66] F. Hameed, A. Khan, and J. Castrillon, "ALPHA: A Novel Algorithm-Hardware Co-design for Accelerating DNA Seed Location Filtering," *IEEE Transactions on Emerging Topics in Computing*, pp. 1–1, 2021.

[67] M. Fey and J. E. Lenssen, "Fast graph representation learning with PyTorch Geometric," in *Proceedings of the ICLR Workshop on Representation Learning on Graphs and Manifolds*, 2019.

[68] J.-A. Chen, H.-H. Sung, X. Shen, S. Choudhury, and A. Li, "Bitgnn: Unleashing the performance potential of binary graph neural networks on gpus," in *Proceedings of the 37th International Conference on Supercomputing*, 2023, pp. 264–276.

[69] Y. Paik, C. H. Kim, W. J. Lee, and S. W. Kim, "Achieving the performance of all-bank in-dram pim with standard memory interface: Memory-computation decoupling," *IEEE Access*, vol. 10, pp. 93 256–93 272, 2022.

[70] J. Gao, W. Ji, F. Chang, S. Han, B. Wei, Z. Liu, and Y. Wang, "A systematic survey of general sparse matrix-matrix multiplication," *ACM Computing Surveys*, vol. 55, no. 12, pp. 1–36, 2023.

[71] T. Hoefler, D. Alistarh, T. Ben-Nun, N. Dryden, and A. Peste, "Sparsity in deep learning: Pruning and growth for efficient inference and training in neural networks," *Journal of Machine Learning Research*, vol. 22, no. 241, pp. 1–124, 2021.

[72] S. Qiu, L. You, and Z. Wang, "Optimizing sparse matrix multiplications for graph neural networks," in *International Workshop on Languages and Compilers for Parallel Computing*. Springer, 2021, pp. 101–117.

[73] A. Wang, A. Singh, J. Michael, F. Hill, O. Levy, and S. R. Bowman, "Glue: A multi-task benchmark and analysis platform for natural language understanding," *arXiv preprint arXiv:1804.07461*, 2018.

[74] P. Zhou, B. Zhao, J. Yang, and Y. Zhang, "A durable and energy efficient main memory using phase change memory technology," *SIGARCH Comput. Archit. News*, vol. 37, no. 3, p. 14–23, Jun. 2009.