2022 55th IEEE/ACM International Symposium on Microarchitecture (MICRO) | 978-1-6654-6272-3/22/$31.00 ©2022 IEEE | DOI: 10.1109/MICRO56248.2022.00060

CORUSCANT: Fast Efficient Processing-in-Racetrack Memories

Sebastien Ollivier[†], Stephen Longofono[†], Prayash Dutta[‡], Jingtong Hu[†], Sanjukta Bhanja[‡], and Alex K. Jones[†]
[†]Electrical and Computer Engineering
University of Pittsburgh
Pittsburgh, PA USA
{sbo15,stl77,jthu,akjones}@pitt.edu

[‡]Electrical and Computer Engineering
University of South Florida
Tampa, FL USA
{prayash,bhanja}@usf.edu

***Abstract*—The growth in data needs of modern applications has created significant challenges for modern systems leading to a "memory wall." Spintronic Domain-Wall Memory (DWM), provides near-SRAM read/write performance, energy savings and non-volatility, potential for extremely high storage density, and does not have significant endurance limitations. However, DWM's benefits cannot directly address data access latency and throughput limitations of memory bus bandwidth. Processing-in-memory (PIM) is a popular solution to reduce the demands of memory-to-processor communication by offloading computation directly to the memory. PIM has been proposed in multiple technologies including DRAM, Phase-change memory (PCM), resistive memory (ReRAM), and Spin-Transfer Torque Memory (STT-MRAM). DRAM PIM provides solutions for a restricted set of two operand bulk-bitwise operations. PIM in PCM and ReRAM raise concerns about their effective endurance and PIM in STT-MRAM has insufficient density for main-memory applications.**

We propose CORUSCANT, a DWM-based in-memory computing solution that leverages the properties of DWM nanowires and allows them to serve as polymorphic gates. While normally DWM is accessed by applying spin polarized currents orthogonal to the nanowire at access points to read individual bits, *transverse* access along the DWM nanowire allows the differentiation of the aggregate resistance of multiple bits in the nanowire, akin to a multi-level cell. CORUSCANT leverages this transverse reading to directly provide *multi-operand bulk-bitwise logic*. Leveraging this multi-operand concept enabled by transverse access, CORUSCANT provides techniques to conduct *multi-operand addition* and *two operand multiplication* much more efficiently than prior digital PIM solutions. CORUSCANT provides a 1.6× speedup compared to the leading DRAM PIM technique for query applications that leverage bulk bitwise operations. Compared to the leading PIM technique for DWM, CORUSCANT improves performance by 6.9×, 2.3× and energy by 5.5×, 3.4× for 8-bit addition and multiplication, respectively. For arithmetic heavy benchmarks, CORUSCANT reduces access latency by 2.1×, while decreasing energy consumption by 25.2× for a 10% area overhead versus non-PIM DWM.

***Index Terms*—Processing-in-memory, Domain-Wall Memory, Novel Memories, Machine Learning**

I. Introduction

Rising data demands of increasingly popular and ubiquitous applications, ranging from real-time searching to machine learning, have created significant data movement challenges for traditional von Neumann systems. While considerable effort has been undertaken to improve memory storage density and energy consumption—through deeply scaled memories and tiered memories that include non-volatile memory—the fundamental data access latency and throughput have not kept pace with application needs. This is commonly referred to as the "memory wall" [1], [2], as it limits potential performance of memory bound applications due to the limited bandwidth of the bus between memory and processor. Additionally, moving data on this bus has been proven to consume a disproportionately large amount of energy, especially for programs which require large working sets. For example, adding two 32-bit words in the Intel Xeon X5670 consumes 11× less energy than transferring a single byte from the memory to the processor [3].

Processing-in-memory (PIM) [4], [5], [6], [7], [8], [9], [10] and near data processing (NDP) [11], [12] solutions promise to reduce the demands on the memory bus and can be a solution to efficiently realizing the benefit of increasingly dense memory from deep scaling and tiered memory solutions. However, leading solutions for bulk-bitwise PIM in DRAM [4], [5] are limited to two operand operations. Multi-operand bulk-bitwise PIM has been suggested for Non-Volatile Memories (NVMs) but only experimentally explored for two-operands [6].

Unfortunately, Phase Change Memory (PCM), the leading commercial candidate in the tiered memory space, has endurance challenges (circa 10^8 writes [13]) and relatively high write energy (up to 29.7pJ per bit [14]) that raise concerns about its effectiveness for PIM. Resistive memory (ReRAM) has a similar concern. STT-MRAM, which has also proposed for PIM [15], is a worthy cache candidate and does not suffer from the same endurance challenges as PCM and ReRAM. However, STT-MRAM has insufficient density, *i.e.,* 28-32F^2, to be deployed at the main memory level of the hierarchy.

Recently, it has become popular to use the analog characteristics of, particularly memristor-based, crossbar arrays to accelerate neural networks [16], [17] but these techniques can also lead to endurance as well as fidelity concerns.

Spintronic Domain-Wall Memory (DWM), also known as Racetrack Memory [18], improves over other suggested candidates as PCM and ReRAM for tiered memory. It has the necessary density, *i.e.,* between 1-4F^2 per cell, while not suffering from endurance concerns. It also has a low energy consumption of circa 0.1pJ [7] per write and a low access latency of circa 1ns, which has led to several researchers' proposals to use DWM as main memory [19], [20], [21], [22], [23].

Recognizing the potential of DWM-based main memory [24]

978-1-6654-6272-3/22/$31.00 ©2022 IEEE
DOI 10.1109/MICRO56248.2022.00060

and the need for PIM to accelerate next-generation data movement constrained applications, this paper proposes CORUSCANT, or *Computing Optimized Racetracks Using Specialized Clusters Accessing Nanowires Transversely*. CORUSCANT leverages a special property of DWM that allows a *transverse* read (TR) [25], or a method to *access* the *nanowire* and count the number of ones across multiple domains. Our approach leverages TR to treat the DWM nanowire as a **polymorphic gate** to directly implement *Racetracks* that are *optimized* for *computing* arbitrary logic functions, sum, and carry logic output. CORUSCANT implements *multi-operand* bulk-bitwise as well as multi-operand addition PIM operations, which can outperform recently proposed main-memory PIM architectures. A carry-save inspired multi-operand addition is proposed to efficiently implement multiplication and provide savings for large reductions over addition.

CORUSCANT uses these building blocks to create *specialized* PIM-enabled domain-block *clusters* (DBCs). These DBCs are interleaved throughout memory tiles to create the facility for massively parallel PIM in the CORUSCANT main memory. We show that the combined speedup of our multiplication procedure and the ability to process multiple operands significantly mitigates the memory wall and enables more sophisticated general PIM than prior work. Specifically, the contributions of this paper are:

- We present a novel technique to utilize a segment of DWM nanowire as a polymorphic gate, including the required sensing and logic circuitry.
- We present the first technique, to our knowledge, to perform multi-operand logic and addition operations in DWM using this polymorphic gate. We further extend this with shifting to implement efficient multiplication.
- We describe a PIM-enabled domain-block cluster architecture built from the PIM-enabled DWM.
- We propose a technique called Transverse Write (TW) to write and shift a segment of the nanowire in a single operation.
- We provide a detailed analysis of CORUSCANT compared to state-of-the-art PIM approaches in terms of energy, performance, and area.

CORUSCANT is effective for myriad applications such as database searching that requires multi-operand bulk bitwise computation and convolution-based machine learning that leverages arithmetic operations.

The remainder of this paper is organized as follows. In Section II, the necessary background on Racetrack memory, its architecture, previous PIM techniques, and TR are related. Next, Section III describes the basic concepts of CORUSCANT, alongside our modified Sense Amplifier (SA) and supporting circuitry. Furthermore, this section discusses several approaches to perform smart multiplication with a concrete example. We describe a case study of using CORUSCANT PIM to implement deep learning convolutional network machine learning algorithms in Section IV. In Section V, experimental results compare the improvements of CORUSCANT with state of the art DWM PIM architectures, including basic DWM architectures for addition and multiplication workloads as well as bulk-bitwise operations in DRAM with Ambit and ELP^{2}IM. Finally, conclusions are reached in Section VI.

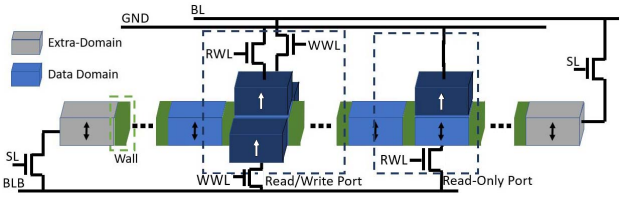


Fig. 1: Anatomy of a DWM nanowire.

II. Background and Related Work

In this section, we first introduce the fundamentals of DWM, how it functions, and our memory architecture designed for maximal compatibility with prior main memory organizations and controllers. We then discuss previous PIM work including proposals for PIM in DWM and propose appropriate comparison points for CORUSCANT. Finally, we describe details of TR and how it enables our PIM approach.

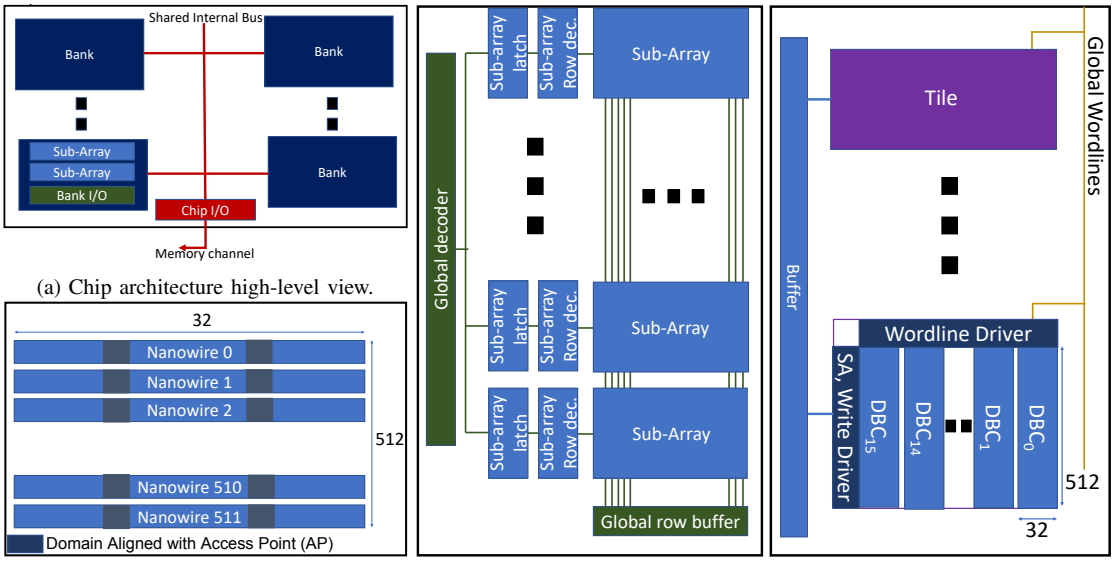
A. Domain-wall Memory Fundamentals

DWM is a spintronic non-volatile memory made of ferromagnetic nanowires. DWM nanowires consist of magnetic domains separated by domain walls (DWs) as shown in Fig. 1. Each domain has its own magnetization direction based on either perpendicular (+Z/-Z), as shown in the figure, or in-plane (+X/-X) magnetic anisotropy, (*i.e.,* magnetization direction).

Binary values are represented by the magnetization direction of each domain, either parallel/antiparallel to a fixed reference. In a nanowire, several domains share one/few access point(s) for read and write operations [26]. DW motion is controlled by applying a short current pulse laterally along the nanowire governed by `SL`. As storage elements and access devices do not correspond one-to-one, a random access requires two steps to complete: ① *shift* the target domain to *align* it with an access port and ② apply an appropriate voltage/current like in STT-MRAM to *read* or *write* the target bit.

The blue domains are dedicated for the actual data stored in memory. The grey domains are extra-domains used to prevent data loss while shifting data. The dark blue elements are the read or read/write ports. Fig. 1 contains a read-only port that has a fixed magnetic layer, indicated in dark blue, which can be read using `RWL`. The read/write port is shown using shift-based writing [27] where `WWL` is opened and the direction of current flows between `BL` and $\overline{\texttt{BL}}$, and reading conducted from $\overline{\texttt{BL}}$ through the fin and up through `RWL` to `GND`.

To align a domain with an access port, a current must be sent from one extremity of the nanowire, shifting each domain. This inherent behavior of DWM can be imprecise, generating what is known as a "shifting fault" in the literature. Several solutions have been proposed to mitigate shifting faults [28], [29], [30], [31], which are orthogonal to and compatible with CORUSCANT. After alignment, DWM accesses inherit the



(a) Chip architecture high-level view.

(d) DWM Domain Block Cluster (DBC) (b) DWM Bank (c) DWM subarray

Fig. 2: DWM Architecture

same reliability benefits, challenges, and solutions as STT-MRAM [32].

B. Memory Designs with DWM

To employ DWM as the sole main memory and to minimize changes to the memory controller design by maintaining the same I/O interface as used for DRAM, we adopt the same high-level architecture shown in Fig. 2(a); this approach has been previously proposed for emerging technologies [33] as well as extended for DWM [23]. The architecture preserves the bank organization into subarrays as shown in Fig. 2(b). Moreover, it maintains the same tile size [34] as used in traditional DRAM-based main memory. This also allows for data movement within the memory such as inter-bank copying that is described in previous work [35]. Fig. 2(c) shows the DWM subarray breakdown into tiles, which share the same global wordlines. To facilitate DWM integration we divide tiles into domain block clusters as also shown in Fig. 2(c). Each DBC shares the same local sensing circuitry and write driver as described in prior DWM memory proposals [23].

Each DBC, as detailed in Fig. 2(d), consists of X parallel racetracks composed of Y data domains. X represents the number of bits that can be accessed simultaneously. We show the example where $X = 512$ for a typical 512×512 tile. Y represents the distinct row addresses contained within the DBC. Y is determined based on the data length possible in a DWM nanowire. We show a conservative example of $Y = 32$, however, examples of longer nanowires are used in other DWM memory proposals [23], [29]. This architecture can easily be scaled such that $32 \leq Y \leq 512$, allowing for longer nanowires.

While, a single access point (AP) is necessary for each nanowire in the DBC, adding APs can reduce the delay by reducing the shift distance between accesses [36]. We show two APs in the example, which would traditionally divide the nanowire length into equal sections to minimize shift latency and reduce the number of overhead domains required. In addition, a second AP can also enable the polymorphic gate capability of the nanowire if placed sufficiently close to allow transverse access. We discuss this further in Section II-D. First, we review prior work in PIM in the next section.

C. Processing in Memory

In this sub-section, we first present the state of the art bulk-bitwise operations in DRAM followed by PIM in emerging technologies while specifically highlighting prior efforts for PIM in DWM.

1) PIM with bulk-bitwise operations in DRAM: There have been two major proposals to conduct bulk-bitwise processing directly in DRAM [4], [5] with some operations having been demonstrated in commercial devices [37]. Bulk-bitwise logic combines two rows "bitwise" with the same logic operation such that $c_i = a_i$ `OP` b_i. Ambit proposed to open three DRAM rows simultaneously and compare the combined voltage to the sensing threshold, *i.e.,* $\frac{V_{DD}}{2}$ [5]. A majority of '1's results in $\geq \frac{2V_{DD}}{3}$ which would drive the senseamp (SA) to V_{DD}. A minority of '1's results in $\leq \frac{V_{DD}}{3}$ driving the SA to V_{CC}. Computing `AND` requires a third control row set to '0' so that both data rows must contain '1's for the result to be '1.' `OR` is computed setting the control row to '1' requiring only one data row to be '1.' This process is destructive, as all three rows now contain the result of the logical operation.

Ambit builds on RowClone [35], which copies the source row by waiting for the SA to refresh the row and then opens the destination row which is overridden by the SA. Thus, operands are duplicated in a safe location to conduct the logic operation without destroying the original data. To create a complete logic set, a dual-contact cell (DCC) concept is employed allowing a cell to be read as the inverted value through $\overline{\texttt{BL}}$. A DCC row

requires the same overhead as two regular rows. To execute A XOR B requires using DCC rows to invert both operands, first computing $k = A$ AND $\overline{B}$, followed by $k' = \overline{A}$ AND B. The final answer comes from k OR k'

ELP^{2}IM improves on Ambit by directly performing logic operations without moving the data. Instead, the technique changes the pseudo-precharge state of the SA to replace the control row [4]. The process requires multiple comparisons to ultimately determine the final logic value, but avoids the need for cloning rows. ELP^{2}IM demonstrates a 3.2× performance improvement over Ambit and a near-data processing approach [11] for bitmap and table scan applications.

Ambit and ELP^{2}IM, insomuch as they are complete logic sets, are capable of computing more complex arithmetic operations such as addition. More complex logic requires sequential steps to determine the result similar to the XOR example described for Ambit. Next we discuss PIM for NVMs.

2) PIM in emerging memory technologies: Pinatubo is a PIM concept that resembles aspects of Ambit and ELP^{2}IM. Like Ambit, it opens the two rows for comparison simultaneously, and like ELP^{2}IM it adjusts the sensing circuitry to conduct different operations [6]. For example, changing the V_{TH} to $< \frac{V_{DD}}{2}$ allows an OR operation and $> \frac{V_{DD}}{2}$ allows an AND operation. Pinatubo conceptually applies to NVMs that distinguish data based on resistive sense margins including PCM, ReRAM, and STT-MRAM. Pinatubo mentions multi-operand operations in a qualitative scalability discussion.

MAGIC [10] proposes a novel memristive memory crossbar *transpose memory* allowing interesting operation flexibility over rows or columns. It demonstrates addition but is subject to endurance limitations, operates on one bit per row/column, and is admitted by the authors to be complicated to program with limited applications. CRAM implements bulk bitwise operations and uses them as building blocks to implement addition and multiplication in STT-MRAM [9]. It does this by extending the magneto-tunnel junction (MTJ) with a second transistor, which unfortunately further decreases the effective density of the already insufficiently dense memory.

Two techniques have been proposed to augment DWM with PIM capabilities. DW-NN creates a PIM processing element with dedicated circuitry to support current passing through two stacked domains at once. This allows measurement of the aggregate giant magnetoresistance (GMR) across the stacked domains [7]. This computes XOR which is '0' if the data is parallel and '1' if the data is anti-parallel. To conduct addition, operand bits are stored in consecutive bits within a single nanowire. The XOR operation is used in combination with a pre-charge sensing amplifier (PCSA) that can compute a function of data from three nanowires' access port—sum S is the result of two consecutive XORs, and C_{OUT} is the result of the comparison of PCSA$(A,B,C_{IN}) >$ PCSA$(\overline{A},\overline{B},\overline{C_{IN}})$. Both operations are bitwise serial since they must be shifted into alignment with the GMR/MTJs. Because operands are stored within a single nanowire, multiplication is possible using addition of shifted versions of one operand. Compared to accessing data and using a general purpose processor for computation, DW-NN claims

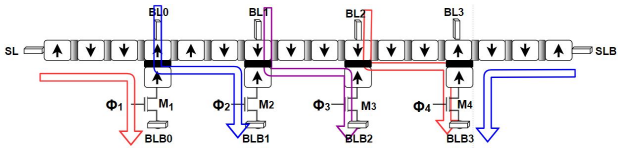


Fig. 3: Transverse read example for a full nanowire

an energy improvement of 92× and a throughput improvement of 11.6× for an image processing application.

SPIM extends DWM storage with dedicated skyrmion-based computing units [8]. Within these units, custom ferromagnetic domains are physically linked together with channels that support OR and AND operations. By permanently merging many such domains and channels, full adder circuits are formed to perform addition and multiplication.

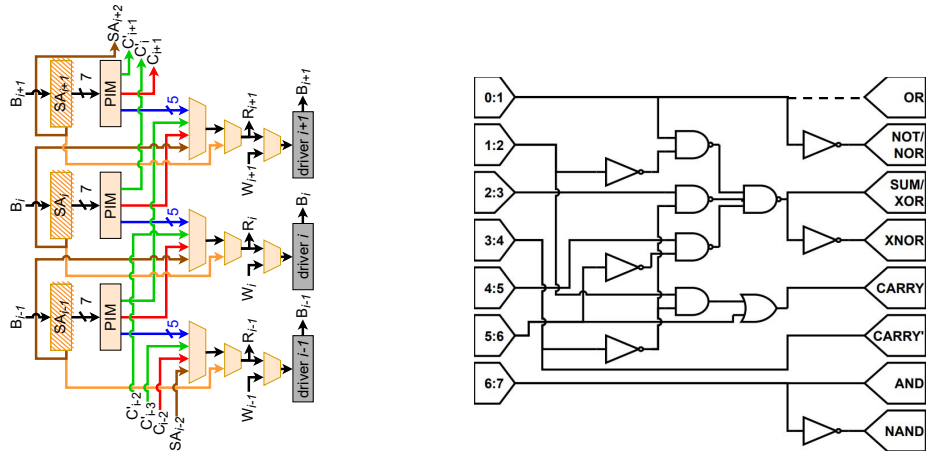
In contrast, CORUSCANT provides a new DWM PIM method that can achieve a superset of all the instructions proposed among these various methods while increasing the parallelism and efficiency. Thus, to demonstrate the value of CORUSCANT, we draw direct comparisons to Ambit, ELP^{2}IM as the most near-commercial PIM approaches in DRAM as well as DW-NN and SPIM as the technologically most close proposals in our experiments (Section V). CORUSCANT leverages the transverse read operation discussed in the next section to conduct its PIM functionality.

D. Transverse read

TR was recently proposed [25] and leveraged to improve reliability via detection and correction of over/under-shifting faults through codes that count the number of ones in overhead bits to check position [28], [30], [31]. TR is conceptually akin to using a portion of a DWM nanowire as a multi-level STT-MRAM cell. Specifically, TR is an aggregate function of several domains at once along the nanowire. The output of a TR provides the number of ones stored between two heads or one head and an extremity, but without information about their positions. As the number of domains in the TR increases, the minimum sense margin decreases which creates a limit on the number of domains that can be included in a TR. We refer to this as the *maximum TR distance* or simply TRD.

Fig. 3 presents a segmented TR that can be used to query the full nanowire in the case that distance from the extremity to the access point is larger than the TRD. Each colored arrow represents the path taken by the current used to perform the TR. For instance, to perform a TR on the middle four domains (purple arrow), the transistors M1, M2 and M4 are open and M3 is closed, thus when a current is sent from BL1, it has to go through the four middle domains and exit through M3.

The two red and blue arrows indicate TR over regions with the same color can occur simultaneously. For the red arrows, M2, M3 and SLB transistors are open while M1 and M4 are closed. The current sent from SL and BL2 will flow to M1 and M4, respectively, reading two and one '1's as output, respectively. Due to the larger nanowire resistivity between BL0 and BL2, the leakage current is small enough that the TR can safely be parallelized.



(a) PIM DBC sense and driver circuit.

(b) PIM block logic.

Fig. 4: Overview of the PIM enabled DBC extensions.

The number of '1's in a segment of a DWM nanowire may also be measurable through the Anomalous Hall Effect (AHE) [38], [39]. Recently, a "multi-domain" MTJ was proposed as a scalable alternative to TR for MDR [40]. The multi-domain MTJ creates an access port across multiple domains such that, when a read current is applied, each of the domains function as parallel resistors. Like TR, this provides different resistance levels based on the number of parallel and anti-parallel domains. While micromagnetic simulation shows that TR has a relatively low fault rate, the multi-domain MTJ demonstrates resilience to process variation and scalability to seven domains [40].

In the next section, we present how the TR can be used to perform logical and arithmetic operations, forming the foundation of CORUSCANT.

III. CORUSCANT

In this section, we discuss the CORUSCANT architecture including (i) modifications to the sensing circuit leveraging TR to realize multi-operand PIM and (ii) the algorithms to achieve multi-operand addition and two-operand multiplication.

A. CORUSCANT Architecture

We propose to add PIM capability to a portion of the DBCs in the memory architecture, see Fig. 2(c). The number of PIM enabled tiles/DBCs can be tuned based on the overhead and the desired PIM parallelism. The detailed DBC shown in Fig. 2(d) shows two access points. In CORUSCANT, these access points are spaced according to the TRD. TR has been demonstrated for a conservative TRD $= 4$ and was stated to scale beyond that number [25] and presumed to scale TRD $= 32$ in prior work [28]. We describe CORUSCANT for a TRD $= 7$, which is supported by recent work [40], while in our quantitative evaluation, we also conduct experiments for TRD $\in \{3,5,7\}$ as a sensitivity study.

Assuming $Y = 32$ distinct row addresses are contained within each DB, see Section II-B, with a single access point, each nanowire would require 63 domains $(2Y-1)$ to permit shifting data at the extremities to the access point. Normally, adding a second access point would place ports at positions 9 and 25, reducing the number of overhead domains from 31 to 16. To enable TR with a TRD $= 7$, the ports would move to positions 14 and 20 and the overhead domains would only reduce from 31 to 25. Adding ports in this way provides some reduction of average shift distance while allowing for the TR operation. For two ports to remain at their optimal shift reduction position would require a TRD $= 14$, which we presume to be infeasible.

For tiles/DBCs with the additional access port to conduct TR, we modify the sensing circuitry as shown in Fig. 4(a) where the tan blocks show the added elements. To enable performing TR requires each SA_i to output seven level bit values such that $SA_i[j]$ is '1' if there are $\geq j$ '1's in the TR and $j \in 1..7$. The extension with additional sensing circuitry is represented by a hashed tan block. These SA outputs become the seven inputs for the PIM unit described in Fig. 4(b). The PIM logic output is selected by a multiplexer. Note, the ith multiplexer selects between values provided by the local PIM block as well as two inputs from the $(i-1)$st and $(i-2)$nd PIM blocks, respectively. We will explain the purpose for the color coding and these connections in the following sections.

There is a direct read from the SA shown in orange that bypasses the PIM logic and the selector to a single two-way mux that feeds the read port. Thus, either a direct read or result of PIM logic can be directly forwarded via the hierarchical row-buffer structure to the memory controller and returned to the processor. The capability is also added that PIM output can be written back to the memory block so an additional selector multiplexes the PIM logic with the write port input. As in previous work [5], [6], [35], given the hierarchical row buffer in the memory, the shared row buffer in the subarray or across subarrays can be used to move data from non-PIM DBCs to PIM-enabled DBCs.

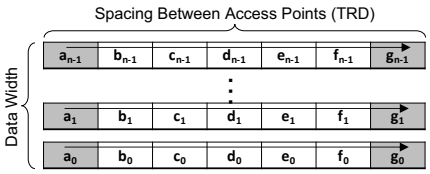


Fig. 5: TR for Bulk-Bitwise Operations

B. CORUSCANT Multi-operand Bulk-bitwise Operations

The TR operation described in Section II-D allows direct implementation of bulk-bitwise operations coupled with additional sensing and logic shown in Fig. 4(b). If the TR level is above one, the OR operation is '1' and similarly the inversion of this reports NOR. If a single data value is stored and the remaining rows are zero padded, this output also reports NOT. AND and NAND are obtained in a similar fashion but with the highest TR level. XOR reports exclusively the odd TR levels computed by relatively simple NAND/NAND implementation. To save area, XNOR is the inverted value of XOR. While, at first glance, this might appear like a significant overhead, it is important to keep in mind that this logic will compute these operations for seven operands in parallel. To support addition, the PIM block also contains a carry C computation which is a function of TR levels above two and not above four or above six. A super carry C' is computed from TR level above 4 and sum S is equivalent to XOR. Details on the energy consumption and area overhead are discussed in Section V.

We show an example in Fig. 5, for the portion of the DBC between the two access points, denoted by shaded domains where a or g could be directly read, and the rest of the nanowire is abstracted away for convenience of display. Using a TR, a multi-operand operation can be directly obtained for bulk-bitwise OR, NOR, AND, NAND, XOR, or XNOR. Because OR uses the same sensing circuit as a traditional read, but the read is conducted using a TR, it is made available through the orange path; the remaining five operations are denoted by the blue output of the PIM block [Fig. 4(a)]. Comparing fewer than seven operands can be accomplished by padding the unused locations in the scope of the TR. The result can be written over one of the original operands (either a or g) or written into a separate DBC. To facilitate efficient smaller cardinality operations, some DBC locations can be preloaded with zeros or ones. In order to minimize the energy and area overhead, CORUSCANT applies this PIM extension to a subset of the tiles, *e.g.,* one tile per subarray.

C. CORUSCANT Multi-operand Addition

Based on the bulk-bitwise operations from the previous section, we show an example addition operation for five operands in Fig. 6. In step ①, referencing Fig. 4(a), a TR of dwm_0 (first nanowire) is conducted, evaluating bit_0 of all operands. S_0, which is XOR of $a_0 ... e_0$, computed by the PIM block and is among the five blue bits.

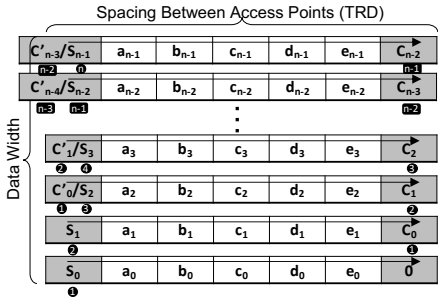


Fig. 6: Addition using TR

Simultaneously, carry, C_0, is computed and sent to the right to the driver for dwm_1 shown in red and super carry, C'_0 is sent to the driver for dwm_2, shown in green in Fig. 4(a) and computed using the logic functions shown in Fig. 4(b). S_0, C_0, C'_0 are written into the left access port ($port_L$) of dwm_0, the right access port ($port_R$) of dwm_1, and $port_L$ of dwm_2 simultaneously. In step ②, a similar set of steps occurs except the operations include C_0 in addition to $a_1 ... e_1$. Then in step ③ TR is conducted over $C'_0, a_2 ... e_2, C_1$, which is seven total elements. In the general case, for step $k+1$ (*i.e., dwm_k*), TR is conducted over $C'_{k-2}, a_k ... e_k, C_{k-1}$ with S_k written to $port_L$ of dwm_k, C_k written to $port_R$ of dwm_{k+1} and C'_k written to $port_L$ of dwm_{k+2}. The control for this operation is a simple counter circuit that provides the selectors the values for a window of three nanowires and activates the bit lines for $k ... k+2$.

Because the carry chain requires keeping the $port_L$ and $port_R$ clear to write C, C', for a TRD $= 7$ we can compute a maximum of five-operand addition. There are many cases when it may be desirable to efficiently add more than five operands, such as to conduct multiplication, which we discuss in detail next.

D. CORUSCANT Multiplication

A foundational method to compute $A * B$ is to sum A B times; *e.g.,* for $B = 3$, $A * 3$ can be computed as $A + A + A$. Thus, we can perform a multiplication by doing several additions. Even with a 5 operand add, this method can quickly require many steps. Consider $9A$, this can be computed by computing $5A$ in one step, and then computing $5A + A + A + A + A$ in a second step. This method could be improved by generating $5A$ in one addition step, then replicating $5A$ and summing to compute $25A$, and so on, but this clearly scales poorly. One method to accelerate this process is to shift the copies of A to more quickly achieve the precise partial products that, when summed, produce the desired product.

In Fig. 4(a), we show how data read from bit i is forwarded to bit $i+1$ using the brown lines. This connection allows a logical left shift, similarly proposed in prior work [41], which is equivalent to a multiply by 2 or A'. To logically shift by more than one position, we first write A' and then shift and write A'' or $A << 2$. It is important to distinguish between these **logical shifts** being discussed, which move bits *between* nanowires and **DW shifts**, which shift the nanowires to access different data locations.

Looking at Fig. 6, logical shifts occur in the Y direction and require the multiplexing logic from Fig. 4(a), DW shifts move in the X direction. So, to write $A << k$ requires k shifted read (brown arrows) and write operations. However to write $A << k$ next to $A << k+1$ requires k shifted read and followed write steps, an additional shifted read, a DW shift, followed by a write. Thus, to write y shifted copies with a max logical distance of x requires $x-1$ shifted read/write operations and y DW shifts. Based on the logical shifting capabilities we describe techniques to leverage CORUSCANT to conduct efficient multiplication through optimized multi-operand addition in different situations.

1) Constant Multiplication: When one of the operands of the multiplication operation is known, there are several ways to efficiently take advantage of CORUSCANT to complete multiplication, leveraging logical shifting. At compilation time, a method based on Booth multiplication is possible [42], [43] where numbers can be represented using `0`, `N`, and `P`, which represent 0, -1, and 1, respectively. For example, consider a constant multiplier 20061 $\rightarrow$ "100111001011101," this can be encoded as `P0P00N0P0N00N0P`. It can be decomposed using the pattern `P000000P0N`, which corresponds to 515, in positive and negative forms shifted by different amounts: `P000000P0N00000` $-$ `P000000P0N` $+$ `00P000000000000` $=$ `P0P00N0P0N00N0P`.

Thus, 20061 times A can be computed in two addition steps: ① $A << 9 + A << 1 + A \rightarrow 512A + 2A + A = 515A$, ② $515A << 5 - 515A + A << 12 \rightarrow 16480A - 515A + 4096A = 20061A$. Note $-515A$ can be computed by generating $\overline{515A}+1$ making the last step $515A << 5 + \overline{515A} + 1 + A << 12$ which is still one addition step. This is a significant improvement over adding 20061 copies of A.

2) Arbitrary Multiplication: A more generic method that can also work for arbitrary multiplications is to use the '1's in the multiplier to denote which shifted copies of the multiplier to be summed to create the product. In the 20061 example, there are 9 '1's in the binary form of 20061. Thus, the method to directly compute the product is to logically shift A n times where n is the bit-width of the multiplier B. When $b_i =$ '1' then we also shift the nanowire to retain that "partial product." When five partial products have been retained, we generate the sum. In this example in step ① $T = A + A << 2 + A << 3 + A << 4 + A << 6$, and in step ② product $P = T + A << 9 + A << 10 + A << 11 + A << 14$. Again, this takes only two addition steps. In the worst case, this takes $\frac{n}{\text{TRD}-2}n$ steps or $O(n^2)$ complexity where n is the bit-width of the operands.

3) Optimized Multiplication: Arbitrary multiplication based on partial products requires an efficient summation mechanism. In CORUSCANT we assume arbitrary partial products are generated by copying one operand, to a processing tile as in prior work [35]. The value is then logically shifted by one position and copied $n-1$ times. By then bringing B into the rowbuffer we shift back along the DBC and zero out the *ith* shifted version of A if that bit of B is '0.' This functions like a predicated copy so we have only the correct shifted copies of A to compute the multiply.

For large cardinality multiplication we are left with many partial products, typically $>(\text{TRD}-2)$. To sum them efficiently we can borrow from Carry Save Adders (CSAs). A CSA leverages the three inputs of a full adder A, B, C_{IN} to be used for three operands X, Y, Z instead of two. This creates an entirely parallel process to reduce three operands to two in the form of $S^{\dagger}, C^{\dagger}$. Then a traditional addition using a ripple carry adder can add $S^{\dagger} + C^{\dagger} \rightarrow S$. We can leverage our polymorphic gates in the same way but with more input operands. Seven rows of packed addition operands, or partial products, can be reduced with bulk-bitwise parallelism to three rows containing a S, C, C', as a $7 \rightarrow 3$ operand reduction function.

This method accomplishes two things. The need for the sequential carry logic of addition is not required for the reduction step and the technique can directly be performed on TRD instead of $\text{TRD}-2$ operands. Furthermore, the $7 \rightarrow 3$ reduction operation can be repeated on data, including prior output of this reduction function in a previous step. These reductions are continued until there are $\leq (\text{TRD}-2)$ operands remaining. The final result can then be computed with a single addition operation, where one output is generated from $\leq (\text{TRD}-2)$ inputs. These reductions make multiply an $O(n)$ operation and can also accelerate large cardinality additions found in many scientific and machine learning algorithms.

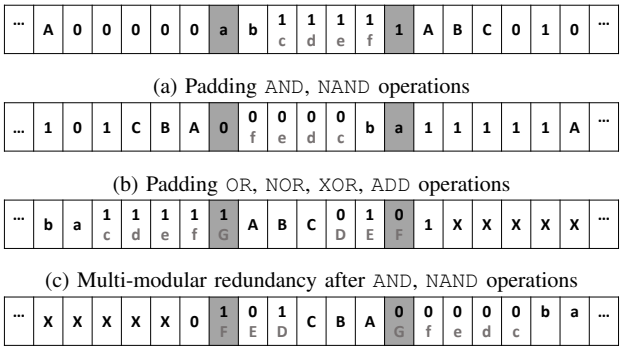
Next, we describe new instruction(s) to control the PIM operations through the memory controller used in CORUSCANT.

E. ISA Support

CORUSCANT presumes some portion of the virtual memory space is reserved for PIM operation and the operating system can manage this space when conducting virtual to physical address translation as is often the case with memory mapped I/O. Thus, the user can then schedule PIM operations in memory and align with tile and DBC boundaries. CORUSCANT also uses one (or more) new instructions that maps the operation to the memory controller to be issued by the memory controller [5]. This instruction:

```
cpim dst, src, op, blocksize
```

communicates to the memory controller to issue the appropriate commands to complete the operation requested. Each CORUSCANT PIM `cpim` instruction consists of a source address `src`, indicating which DBC and nanowire position to align to the leftmost access port. For data movement operations, the memory controller can use either AP in pim enabled tiles. For PIM operations, the operation must be done between the access points. Thus for a bulk bitwise operation of k operands where $k < \text{TRD}$ the user must pad the adjacent locations with data that does not impact the result. To minimize padding overhead, a DBC can be pre-populated as shown in Fig. 7. For example, '1's are added in Fig. 7(a) for `AND`, `NAND`, with operands written to the left access point in locations `a` and `b` ($k = 2$), where operands `c`–`f` ($k = 3$–6) can overwrite '1's as necessary. Similarly, '0's are added for Fig. 7(b) to conduct `OR`, `NOR`, `XOR`, and `ADD`.



(a) Padding AND, NAND operations

(b) Padding OR, NOR, XOR, ADD operations

(c) Multi-modular redundancy after AND, NAND operations

(d) Multi-modular redundancy after OR, NOR, XOR, ADD operations

Fig. 7: Configuration for padding and fault-tolerance.

The operation op coupled with the block size blocksize are used to program the multiplexers select bits of Fig. 4(a). Generally, it is presumed that bulk bitwise operations are done across the entire memory row and packed and padded as necessary by the user. The block size is most important for operations like add or max. The memory controller must issue commands that mask different bitlines to form the carry chain as discussed in Section III-C. The blocksize $\in \{8,16,32,64,128,256,512\}$ allows individual addition of 64 byte sized numbers packed into a row up to a full 512-bit addition.

These cpim operations can be generated by the compiler either automatically or through user directives. Furthermore, when standard memory accesses, *e.g.,* load/store instructions are issued to the memory, the memory controller sets the multiplexer selection bits to bypass the pim unit as represented by the orange arrow on Fig. 4(a).

F. N-Modular Redundancy Support

Since ECC encoding techniques are not homomorphic under PIM operations [5], the best proposed general solution for PIM fault tolerance is modular redundancy and voting. A common example is triple modular redundancy (TMR) where the same operation is computed three times and the final output is generated by voting between the three results. Thus, if there is a fault, the two non-faulty results will ensure the correct result is selected over the faulty one. However, PIM operations have considerably higher fault rates than those using CMOS functional units. This increases the changes that there are two faulty values. When two faulty results in the same bit position, in TMR the faulty result will be selected over the correct result, generating an uncorrectable error. Thus, CORUSCANT provides the capability to compute the N-modular redundancy for $N = \{3,5,7\}$, to offer several reliability thresholds. CORUSCANT directly uses the the majority function for voting, which is the same circuit for computing C' from Fig. 4(b).

Fig. 7(c) shows the DBC after computing a result three times and storing these results in A, B, and C, respectively. To compute a majority operation of our three values A, B, and C using the majority operation of seven rows C', requires padding with two rows of all '1's and two rows of all '0's. To accomplish this with minimized shifting we use the padding values preset with constants. For instance, following a padded AND operation, we use the last '1' row from the '1' padded AND aligned with the left access port and presume a preset set of rows with the values of '0's, '1's, and '0's, respectively as shown in Fig. 7(c). Thus, $2 \leq i \leq 3$ '1's from the inputs A, B, and C, results in $4 \leq m \leq 5$ '1's between the heads, where C' reports a '1' when $m \geq 4$. In contrast, $0 \leq i \leq 1$ '1's (or $2 \leq \bar{i} \leq 3$ '0's) results in $2 \leq m \leq 3$ '1's such that C' reports a '0'. The mirror is shown in Fig. 7(d) for TMR after a '0' padded operation, such as OR. Thus, for an error to occur would require faults in the same bit position of two of A, B, and C, *or* a fault in one of A, B, and C *and* a fault in sensing C'. Either way, this requires two faults in the same bit position for an uncorrectable error.

To extend triple- ($N = 3$) to quintuple- ($N = 5$) or septuple- ($N = 7$) modular redundancy, for protection against more faults, the optional values D–E or D–G can be written between the heads, respectively. In the case ($N = 5$), there will be one '1' and one '0' row, denoted as G and F in the padding bits at each of the access points. For quintuple-modular redundancy to have an uncorrectable error requires three faults in the same bit position among A–E and C'. The last case, ($N = 7$) does not require any padding bits and requires four faults in the same bit position.

Using N-modular redundancy for most operations (bulk-bitwise, $7 \rightarrow 3$ operand reductions, *et cetera*) is straightforward as the operation occurs in a single step. However, the addition operation has multiple steps. Voting during an add operation can either occur after each nanowire computes S, C, C' for a particular bit, or after the entire result is determined. Since the add operation is computed sequentially, this choice about fault tolerance creates a performance versus fault tolerance trade-off.

IV. Case Study: Implementing a CNN

To demonstrate the potential of CORUSCANT we explore the process of computing a convolutional neural network (CNN) using PIM. CNNs contain 3 main types of layers: convolution layers, pooling layers, and fully connected layers, each of which can be completed in CORUSCANT.

A. Convolution

The convolution layer is the process of taking a small kernel (or weight) matrix $\mathbf{K}$, and combining it in "windows" with a larger matrix $\mathbf{I}$ representing input features, at each step multiplying the overlapping positions and accumulating the results. As an example, for $\mathbf{I}$ and $\mathbf{K}$ of size $N \times N$ and 3×3, respectively, the convolution operation for the window at m, p is:

$$Conv(\mathbf{I},\mathbf{K})(m,p) = \sum_{j=0}^{2}\sum_{t=0}^{2} \mathbf{K}_{j,t} * \mathbf{I}_{m+j-1,p+t-1} \qquad (1)$$

In prior PIM work, convolution is completed by computing the multiplication and reduction additions in parallel [4], [8], [41]. Prior DWM work uses full precision form for convolution [8].

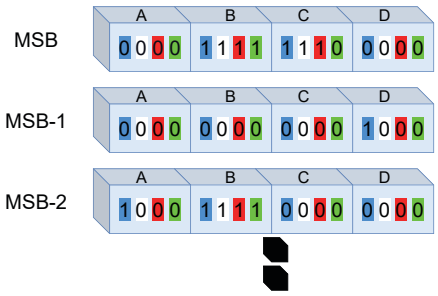


Fig. 8: Maximum function example

In full precision (8-bit) mode, to multiply two values requires adding eight partial products, which using parallel addition can be done in $O(n \log n)$ operations. CORUSCANT has a significant advantage using the CSA technique completing the multiplication in $O(n)$ operations.

The leading DRAM technique implements two approximations, DrAcc [4], [41], which uses ternary weight neural networks (TWNs), *i.e.,* $w_i \in \{-1, 0, 1\}$, and NID [4], [44] using binary weight neural networks (BWNs), *i.e.,* $w_i \in \{0, 1\}$, where w_i is the ith weight. These approximations, which reduce the point-wise multiplication operations to bulk bitwise operations, (*e.g.,* XNOR), essentially reduce the problem to be governed by the number of addition operations for reduction. The number of adds N_a is governed by:

$$N_a = O_s * ((K^2 - 1) * I_c + (I_c - 1)) \tag{2}$$

where O_s is the number of output values, K is the kernel size, and I_c is the number of input channels.

Prior work proposes to compute addition using a carry lookahead adder (CLA). In doing this they compute [41]:

$$\begin{gathered} 1.\ G_i = A_i \& B_i;\ 2.\ P_i = A_i \oplus B_i \\ 3.\ C_{i+1} = G_i || (P_i \& C_i);\ 4.\ S_i = P_i \oplus C_i \end{gathered} \tag{3}$$

which takes 40 cycles using ELP^{2}IM [4]. We can call this one *step*. The first reduction step of Alexnet requires 362 additions, or $\lceil \log_2 362 \rceil \rightarrow 9$ steps, where each step requires 40 cycles. Using CORUSCANT by using the CSA approach requires five $7 \rightarrow 3$ operand reduction steps each of $O(1)$ (4 cycles), followed by one addition that requires 16 cycles. This results in a circa $10\times$ speedup. For the largest convolution window requiring $4.5 \cdot 10^8$ adds, DRAM PIM requires 29 addition steps, while CORUSCANT requires 22 reduction steps and 1 addition step, achieving more than $11\times$ speedup. Smaller windows generate more moderate speedups for CORUSCANT. We evaluate the cumulative speedups in Section V-E.

B. Pooling

During pooling, the dimensionality of an input matrix is reduced by taking the average or maximum of all values in submatrices of a predefined size to generate the output matrix values. Using CORUSCANT, the max function can be realized via TR across all the candidates evaluating MSB to LSB sequentially. Each step compares the binary weight of the same bit position in each word, and the TR result determines the subsequent action via predicated execution with local information. This allows PIM instructions issued by the memory controller to work in parallel across many subarrays in parallel. First, the values upon which to compute the max are stored in adjacent positions between the access points. Then a TR is performed across the MSBs; If TR>0 the value under the right head is read and stored in the rowbuffer. If the MSB is '0' the rowbuffer is reset. This eliminates a value that is lower than the other values. Then the DBC is shifted right and the value of the rowbuffer is written to the left head.

All TRD words are processed in this manner. If TR $= 0$, then each word is read from the right and re-written to the left access point, while shifting in between. Essentially, the data remains unchanged. This is necessary because if all values are '0' in this position, it does not eliminate any values from being the maximum. From a PIM instruction execution perspective, the memory controller issued instructions are identical for all participating subarrays by making the rowbuffer reset command predicated on the TR and tested bit. We can use a DWM AND function in another DBC of the same tile/subarray to store and compute the logic value governing the predicated execution of the row-buffer reset.

The process is repeated for each bit position and the value is read using TR > 0, so the max vector is read, regardless of its location between the heads and if > 1 vectors equal the max value the TR value it is still accessed correctly. Fig. 8 depicts a concrete example for TRD $= 4$ of the state of words A, B, C, and D as they are processed by the maximum subroutine, in chronological order from left to right with a different color representing the values after a bit is processed. At the MSB pass starting with blue, TR> 0. Words A and D have '0' in their MSB, so they are overwritten by the zero vector and B and C are written back unchanged. The result after the first step is shown in white. For MSB-1, TR $= 0$. Thus all words are read and written back unchanged as shown in red. For MSB-2, TR > 0. Words A, C, and D all have '0' in that bit position, so they are overwritten by the zero vector and only B is written back unchanged as shown in green. Now the maximum value has been determined, and will be maintained as all the bits are traversed through the LSB.

The maximum function requires cycling through vectors many times which makes shifting the entire nanowire impractical. To address this concern, and to reduce delay, we propose the novel Transverse Write (TW) technique with *segmented shifting*. This is inspired by the shift-based writing [27] approach and

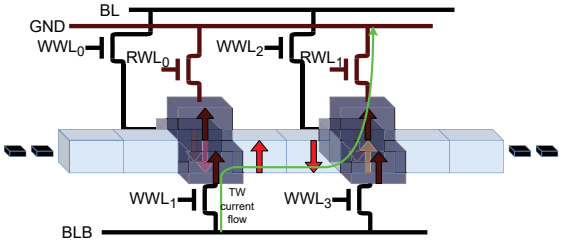


Fig. 9: Transverse write and segmented shift.

transverse access techniques [25] previously proposed. We illustrate this new concept in Fig. 9. Two write/read heads are represented in dark blue separated by data domains in light blue (TRD = 4 is shown instead of TRD = 7 to simplify the explanation). To perform a classic shift-based write under the left head, $\texttt{WWL}_1$ and $\texttt{RWL}_0$ are closed, thus the current flows from BLB to a fixed layer, to the domain, shifting the dark red upward orientation of the fixed layer to replace the pink downward orientation in the nanowire. This operation can be modified to shift the pink orientation along the nanowire rather than to ground. This TW operation closes $\texttt{WWL}_1$ and $\texttt{RWL}_1$. Thus, the current flows from BLB through the fixed layer and the four domains before exiting through the right head as indicated by the green arrow. By doing so, the fixed layer orientation at $\texttt{WWL}_1$ is written under the left head, and the pink orientation and those which follow it advance along the nanowire, forcing the yellow arrow to GND.

Applying this to the maximum function, in the context of Fig. 9, if TR > 0, CORUSCANT reads from the right head (yellow arrow). The predicated rowbuffer reset command is executed. Then the value of the word is written via TW from the left head to the right head. Thus, by reading from the right and conducting TW from the left the segmented shift ensures each updated operand is returned to its original position along the nanowires and the remaining locations are not disturbed. TW for TRD = 7 reduces maximum function cycles by 28.5%. TW can also reduce the cycles required for padding operations where the number of operands $<$ TRD and for TMR.

C. Fully Connected

The fully connected layer executes the following function:

$$ReLU(\mathbf{W}\mathbf{x}+\mathbf{b}) \tag{4}$$

where $\mathbf{W}$ is the weight matrix, $\mathbf{x}$ is the input vector and $\mathbf{b}$ is the bias vector. The *ReLU* function returns zero if $\sum_{i=0}^{i_0}\mathbf{W}_{ij}\mathbf{x}_i+\mathbf{b}_j \leq 0$ and $\sum_{i=0}^{i_0}\mathbf{W}_{ij}\mathbf{x}_i+\mathbf{b}_j$ otherwise. This function is implemented by computing $\sum_{i=0}^{i_0}\mathbf{W}_{ij}\mathbf{x}_i+\mathbf{b}_j$ using CORUSCANT addition and multiplication operations. Using a predicated row refresh based on the MSB, which is '1' for values < 0 and writing the value back the resulting value from the *ReLU* function to the array, repeating $\forall j$.

V. Experimental Results

To experimentally quantify the advantage of CORUSCANT, first, we compare CORUSCANT addition and multiplication characteristics against other computing units based on DWM. Next, we demonstrate the benefit of CORUSCANT PIM on addition/multiplication oriented benchmarks from polybench [45] versus computing them in the CPU. Third, we compare bitmap indices [46], a common component of database queries, against Ambit and ELP^{2}IM to show the benefit over state-of-the-art DRAM PIM. Finally, we implement two CNN applications, Lenet5 and Alexnet, using the method in Section IV and compare CORUSCANT to SPIM, Ambit, and ELP^{2}IM. To guide our experimental observations, we discuss device-level modeling assumptions for CORUSCANT in the next section.

A. CORUSCANT Experimental Assumptions

Based on the device level information provided in [7], [47], [48] and TR results for TRD = 4 with stated scalability to higher numbers of domains [25], we calculated the timing and energy of read, write and TR operations for DWM. We further note a new method to compute TR with improved reliability and scalability to TRD = 7 has been proposed [40]. We have designed CORUSCANT's 7 transistor sense circuits for TR and synthesized the PIM logic from Fig. 4(b) in 45nm technology using FreePDK45 [49] and the Cadence Encounter flow. We then scaled the design by setting F to 32nm as described in prior work [47], [50] to compare with the 32nm results reported in prior DWM PIM work [7]. To calculate the energy we used a modified version of NVSIM to report the DWM energy at 32nm and modified the sense amplifier energy using our custom sensing circuit designed in LTSPICE and scaled energy reported from ASIC synthesis for the PIM logic gates. We assume shifting fault tolerance [28], [29] that achieves a mean time to failure $>$10 years with $<$ 1% performance overhead sufficiently addresses misalignment and requires a negligible overhead in our simulations.

Table I shows the area overhead for extending the memory with PIM capability based on the circuit overheads, additional domains, and access points added to one tile (1-PIM) in each subarray of the memory. It results in a 10% overhead for including our full PIM ISA including multiplication (mult), five operand addition (add), and seven operand bulk-bitwise operations. By stripping the bulk-bitwise operations, this overhead reduces to about 9%, removing the multiplication also keeps us around 9%. Dropping from a five to two operand adder (*i.e.,* what is possible with TRD = 3) reduces the overhead to $<$ 4%.

For system-level simulation including PIM acceleration with CORUSCANT, we extended RTSIM [51] with a cycle-level simulation model of the full CORUSCANT PIM memory described in Fig. 2 using the DDR3-1600 standard to allow comparisons with DRAM and non-PIM enabled DWM. The system parameters are shown in Table II [3], [52], [53], [54].

B. Comparison with Prior DWM PIM

Prior work in DWM PIM presumed a custom memory design that does not conform to Fig. 2. In this case we directly compare DBCs [Fig. 2(d)] with the architectures proposed in prior work. DWM latency is obtained by calculating the number of operations, including computer and data movement, needed to perform an 8-bit add or multiply operation presuming a 1ns cycle speed, consistent with values reported by NVSIM and LLG for TR. CORUSCANT 8-bit addition shifts and writes the words between the two heads (10 cycles) and then writes after each TR (16 cycles), which yields to a total of 26 cycles. Table III reports the speed, energy, and area of CORUSCANT, DW-NN and SPIM for two operand addition (2op add), five operand addition optimized for area by conducting multiple additions in series (5op add area), five operand addition optimized for latency by replicating addition units (5op add latency), and two operand multiply (2op mult).

TABLE I: PIM area overhead vs. base DWM main memory.

Design	ADD2	ADD5	MUL+ADD5	MUL+ADD5+BBO
Area Overhead 1-PIM	3.7%	9.2%	9.4%	10.0%

TABLE II: DWM parameters

Memory size	1GB (8Gb)	Processor	Intel Xeon X5670
Bus Speed	1000 MHz	Memory Cycle	1.25ns
Number of Banks	32	PIM Mode	High Throughput
Subarrays per Bank	64	`add` (32 bits)	111 (pJ/op)
Tile per Subarray	16	`mult` (32-bits)	164 (pJ/op)
DBC per Tile	15 + 1-PIM	E_{trans}	1250 (pJ/Byte)
DRAM [cycles]	t_{RAS}-t_{RCD}-t_{RP}-t_{CAS}-t_{WR}		20-8-8-8-8
DWM [cycles]	t_{RAS}-t_{RCD}-t_{RP}-t_{CAS}-t_{WR}		9-4-*S*-4-4

CORUSCANT is 1.9×, 9.4×, 6.9× and 2.3× faster and 2.2×, 5.5×, 5.5× and 3.4× less energy than SPIM, the state-of-the-art technique, for 2op add, 5op add area optimized, 5op add latency optimized, and 2op multiplication, respectively. CORUSCANT is faster, even for 2op add because SPIM must compute sum and carry from a series of bit-wise operations in series, while CORUSCANT computes these operations in one step. For 5op add the advantage is from combining an $O(n \log m)$ operation into an $O(n)$ operation, while also being faster in the core $O(n)$ computation, where m is the number of operands and n is the number of bits. CORUSCANT is comparable in area to DW-NN and requires some area increase over SPIM for addition, but reduces the multiplication area by 3.7× and 3.3× compared to DW-NN and SPIM, respectively, while also providing additional processing capabilities such as bulk-bitwise operations.

C. Improvement to Memory Wall versus Non-PIM DWM

We tested CORUSCANT using the standard polyhedral polybench workloads that consists of 29 applications from different domains including linear algebra, data mining, and stencil kernels. From these 29 applications we selected the benchmarks most heavily focused on matrix addition and multiplication, from `2mm`, which is two matrix multiplication,

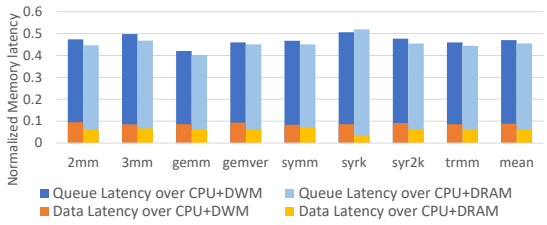


Fig. 10: Normalized DWM Latency

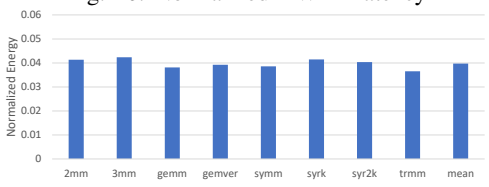


Fig. 11: Normalized Energy Reduction over Polybench benchmarks. Baseline without PIM is 1.

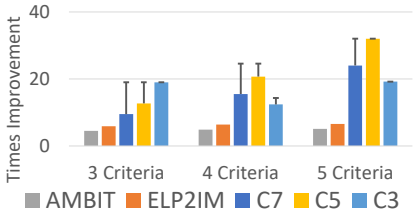


Fig. 12: Normalized performance, the error bars represent the latency improvement with padding

to `gemm` which is governed by $\mathbf{C} = \alpha\mathbf{AB} + \beta\mathbf{C}$. To map operations to PIM we extracted the traces using a pintool, then we examine the accesses and determine which accesses correspond to additions and multiplications. PIM latency, energy, and area overheads are computed using the methodology from Section V-B, with energies for relevant operations also recorded in the table. Instructions are dispatched to PIM-enabled tiles following the *high throughput mode* from prior DRAM PIM work [41], where each tile follows the timing requirement shown in Table II. In other words, to minimize the access latency, the instructions are sent to the different ranks consecutively, in a circular fashion.

The main difference compared to DRAM is "t_{RP}" which is the precharge time, however, DWM as a spintronic memory does not need to precharge, for this reason we replaced the precharge time by the shifting time "S" that is required for DWM. Shifting is dependent on the data placement. Comparing using CORUSCANT PIM to a CPU with DWM and DRAM memories shown in Fig. 10 demonstrates an average latency improvement of 2.07× and 2.20×, respectively. DRAM actually is slower than the DWM memory because, while DWM requires S shifts, the other aspects of its performance, including peripheral circuitry are faster.

We compare the energy to send the data from DWM to the CPU and back plus the energy required to perform a CPU operation versus the energy that CORUSCANT needs to perform the same operation using PIM. Fig. 11 reports the energy improvement from CORUSCANT of more than 25×, on average. This is principally from the data movement energy which is 30× the compute energy.

D. Bitmap Indices

While CORUSCANT has significant benefits over SPIM and DW-NN, neither of these schemes can perform bulk-bitwise logic. Thus, we compare the CORUSCANT to state-of-the-art techniques for bulk-bitwise operation in DRAM, ELP^{2}IM and Ambit for the bitmap indices database query [46] experiment from prior DRAM PIM work [4]. A query from 16 million users' data requested how many male users were active in the past w weeks, where $w \in \{2..4\}$. Fig. 12 shows the latency improvement of Ambit, ELP^{2}IM and CORUSCANT normalized to the latency of a standard DRAM CPU system. CORUSCANT benefits from its intrinsic multi-operand bulk bitwise operations rather than having to construct them from two operand versions as in prior work [4]. Thus for three, four, and five search

TABLE III: Operation Comparison

Scheme	CORUSCANT					DW-NN				SPIM			
Unit	2op Add (TR = 3)	2op Add (TR = 7)	5op Add (TR = 7)	Mult (TR = 3)	Mult (TR = 7)	2op Add	5op Add Area Opt.	5op Add Lat. Opt.	2op Mult.	2op Add	5op Add Area Opt.	5op Add Lat. Opt.	2op Mult.
Speed (cycles)	19	26	26	105	64	54	264	194	163	49	244	179	149
Energy (pJ)	10.15	22.14	22.14	92.01	57.39	40	169.6	169.6	308	28	121.6	121.6	196
Area (μm^2)	2.16	3.60	4.94	3.80	5.07	2.6	2.6	5.2	18.9	2	2	4	16.8

TABLE IV: CNN application comparison

Scheme	Alexnet (FPS)	Speedup× C3	C5	C7	Lenet5 (FPS)	Speedup× C3	C5	C7
Full Precision CNN Inference								
SPIM	32.1	2.2	2.6	2.8	59	2.2	2.6	2.8
CORUSCANT-3	71.1	1	1.2	1.3	131	1	1.2	1.3
CORUSCANT-5	84.0	∅	1	1.1	153	∅	1	1.1
CORUSCANT-7	90.5	∅	∅	1	163	∅	∅	1
ReRAM Crossbar CNN Inference								
ISAAC	34.0	10.5	13.2	14.4	2581	8.6	10.3	12.4
Binary Weight Neural Network (BWN) CNN Inference (NID)								
Ambit	227	1.6	2.0	2.2	7525	2.9	3.5	4.3
ELP^{2}IM	253	1.4	1.8	1.9	9959	2.2	2.7	3.2
Ternary Weight Neural Network (TWN) Inference (DrAcc)								
Ambit	84.8	4.2	5.3	5.8	7697	2.9	3.4	4.2
ELP^{2}IM	96.4	3.7	4.7	5.1	8330	2.6	3.2	3.9
CORUSCANT-3	358	1	1.3	1.4	22172	1	1.2	1.5
CORUSCANT-5	449	∅	1	1.1	26453	∅	1	1.2
CORUSCANT-7	490	∅	∅	1	32075	∅	∅	1

criteria, *i.e.,* male users for last two, three, and four weeks CORUSCANT maintains the same performance while DRAM PIM latency increases; specifically, CORUSCANT is 1.6×, 2.2×, and 3.4× query speedup, respectively, over the faster DRAM approach, ELP^{2}IM.

E. Convolutional Neural Networks

We implemented two CNN benchmarks: CNN Lenet-5 [55] and Alexnet [56], commonly used for image processing, machine learning training on handwritten digits and on RGB images, respectively. We have discussed the implementation of these benchmarks at full precision in Section IV. We compare CORUSCANT including a sensitivity study of CORUSCANT-TRD $\in \{3,5,7\}$ with SPIM in Table IV. CORUSCANT-3 provides a 2.2× improvement over both Alexnet and Lenet and this grows to 2.8× improvement for increasing the TRD $= 7$ due to the 27% performance improvement of CORUSCANT with the larger TRD.

Previous DRAM work, ELP^{2}IM, implements these CNNs using a TWN and BWN approximation. Binary convolution replaces multiplication with XNOR and ternary weights [57] also eliminates most multiplication. For direct comparison we implemented the TWN method modeled after DrAcc [41] using CORUSCANT for TRD $\in \{3,5,7\}$ and compared it with both Ambit and ELP^{2}IM implementations of DrAcc as well as the simpler binary implementation modeled after NID [44] shown in Table IV. CORUSCANT-3 (DrAcc) provides significant speedups of 3.7× and 4.2× for ternary implementations while providing 2.6× and 2.9× improvements over the simpler binary implementations of CNN inference for ELP^{2}IM and Ambit, respectively. This speedup grows to over 5× for Alexnet and approximately 4× for Lenet when using CORUSCANT-7. In general, increasing the TRD from $3 \rightarrow 5$ increases CORUSCANT performance 30-40%, and increasing from $5 \rightarrow 7$ increases performance by another 10-20%. CORUSCANT is considerably faster than the ISAAC ReRAM Crossbar [58], generally achieving an order of magnitude performance improvement. Interestingly CORUSCANT-5 at full precision is nearly identical to the ternary approximation using Ambit, and CORUSCANT-7 at full precision is competitive (within 6% performance) of ELP^{2}IM using the ternary approximation.

Presuming DDR3-1600 memory (Table II) CORUSCANT is capable of executing convolution at 26 Tera Ops Per Second (TOPS) with 108 Giga Ops Per Joule (GOPJ). In comparison, a dedicated similar precision FPGA CNN accelerator was able to achieve 0.34 TOPS with 12.5 GOPJ [59].

F. Reliability

Most PIM work does not address reliability under variation [5], [6], [7], [8]. TR reports a circa 3% change in resistance under process variation [25]. For completeness, we used the LLG Micromagnetic Simulator [60] to verify the TR sense margins [25] for Racetrack Memory 4.0 [61] and conducted an analysis using the total differential method based on output from LLG. Using this approach we define the maximum uncertainty of the actual critical read current in terms of current uncertainty and process variation, the latter based on widely reported 4% variation of spintronic MTJs [62]. We determined a probability of fault during a TR of circa 10^{-6} for 4 domains. Ambit shows a >1% fault rate at 5% variation [4] and ELP^{2}IM improves on Ambit, but does not provide sufficient visibility into fault rates at $< 10\%$ variation, instead reporting them indistinguishable from zero. The first non-zero data is circa 0.35% fault rate at 10% variation [4]. Extrapolating the reported error trend [4] estimates an ELP^{2}IM fault rate of 10^{-3} at 5% variation.

We report CORUSCANT fault rates presuming the intrinsic fault rate of 10^{-6} in Table V. Note a TR fault will cause the TR level to be read as one level higher or lower, a

TABLE V: Operation reliability

Error probability	C3	C5	C7
AND, OR, C'[1] (per bit)	3.3×10^{-7}	2.0×10^{-7}	1.4×10^{-7}
XOR (per bit)	1.0×10^{-6}	1.0×10^{-6}	1.0×10^{-6}
C (per bit)	3.3×10^{-7}	4.0×10^{-7}	4.3×10^{-7}
add (per 8-bits)	8.0×10^{-6}	8.0×10^{-6}	8.0×10^{-6}
multiply (per 8-bits)	4.1×10^{-4}	2.1×10^{-4}	7.6×10^{-5}
N-modulo redund.	$N = 3$ (C3,C5,C7)	$N = 5$ (C5,C7)	$N = 7$ (C7)
AND, OR, C'[1](8-bit)	$(9.6, 3.5, 1.8) \times 10^{-15}$	$(6.5, 2.4) \times 10^{-23}$	3.6×10^{-29}
XOR (8-bit)	$(8.7, 8.7, 8.7) \times 10^{-14}$	$(8.1, 8.1) \times 10^{-21}$	8.0×10^{-27}
C (8-bit)	$(1.0, 1.4, 1.6) \times 10^{-14}$	$(5.2, 6.4) \times 10^{-22}$	3.0×10^{-27}
add (8-bit)	$(5.6, 5.0, 4.8) \times 10^{-12}$	$(4.8, 4.6) \times 10^{-18}$	6.1×10^{-23}
multiply (8-bit)	$(6.2, 5.2, 4.9) \times 10^{-12}$	$(5.0, 4.7) \times 10^{-18}$	6.1×10^{-23}

[1] Only relevant for C5 and C7.

TABLE VI: CORUSCANT CNN with N-modulo redundancy

N-modulo redundancy	$N=3$ (C3,C5,C7)	$N=5$ (C5,C7)	$N=7$ (C7)
Alexnet Full Precision (FPS)	(17.7, 26.9, 29)	(16.2, 17.5)	12.5
Lenet Full Precision (FPS)	(32.5, 49.5, 53.0)	(29.8, 31.6)	22.6
Alexnet Ternary (FPS)	(90.2, 134.8, 155.8)	(81.1, 93.7)	67
Lenet Ternary (FPS)	(5907,8074,9862)	(4868, 5943)	4253

fault off by two or more levels is negligible. Thus, different functions have different error rates depending on the fault rate, due to which level transitions create an error. An 8-bit CORUSCANT `add` requires 8 TR's, one for each bit computing S,C,C', for which errors can accumulate if C,C' are changed. However, `add` has an error rate of 8×10^{-6} for at least one error. Using TMR, this fault rate can be improved to circa 5×10^{-12}. If TMR was done after each nanowire computation, this would provide circa 9×10^{-14}, a nearly two orders of magnitude lower fault rate. In constrast ELP^{2}IM requires 48 operations to compute 8-bit add, such that reliability with TMR is approximately 10^{-2}. CORUSCANT multiply has a large intrinsic error rate, particularly for $C<7$, but by voting between each reduction step it too can reach circa 5×10^{-12}. TMR provides a reasonable reliability, but to achieve >10 year error free runtime, we need $N=5$-modulo reduction which achieves $\leq 5\times10^{-18}$ probability of error.

N-modular redundancy creates a tradeoff in design choices. One option is to reduce the performance by repeating the operation and adding the voting operations computations. If we consider the Polybench results, this will increase the orange portion of the bars from Fig. 10, which are approximately 20% of the runtime with 80% of the runtime coming from queuing delay. While it may be possible to pipeline the results to hide some of the additional orange latency, another option is to increase the number of PIM-enabled tiles and subsequently the parallelism of the memory. While this would significantly increase the area and energy cost, Fig. 11 already shows an energy reduction of 95%, so reducing this savings for retaining performance with $N=\{3,5\}$-modular redundancy is a potentially reasonable tradeoff.

To enable the same level of reliability between ELP^{2}IM and CORUSCANT puts DRAM-based PIM at a disadvantage. Thus, for ISO-reliability operation, the performance improvements of CORUSCANT would only grow because the other PIM methods that report reliability intrinsically lag CORUSCANT by orders of magnitude. Moreover, as shown in Table VI, ISO-area CORUSCANT with TMR is still faster than both Ambit and ELP^{2}IM without any fault tolerance by 1.83×, 1.62× on ternary Alexnet, respectively. CORUSCANT with TMR is within 90% performance of SPIM without any fault tolerance, while remaining 4.8× faster than ISAAC for CNN acceleration. ISO-performance CORUSCANT can generally retain the performance improvements of Table IV through TMR parallelism, but will incur nominal overheads for the inserted voting instructions.

VI. Conclusion

CORUSCANT is a significant step forward for PIM in the promising DWM technology. Our technique takes advantage of the intrinsic proximity of bits in DWM nanowires and the advantages of TR to build a polymorphic gate that can support myriad PIM operations. CORUSCANT can perform bulk-bitwise or addition on multiple operands simultaneously, limited only by the TRD between access ports on DWM. Using carry-save adder inspired techniques these multi-operand operations can be used to efficiently implement multiplication with minimal additional logic.

Our results show that CORUSCANT improves over the state-of-the-art DWM-based PIM by 6.9× and 2.3× in terms of speed and 5.5× and 3.4× in terms of energy for five operand addition (optimized for latency) and multiplication, respectively. Compared to a standard DWM memory without PIM, CORUSCANT improves memory latency by 2.1×, decreases energy by 25.2× versus sending the data to the CPU. CORUSCANT incurs an area overhead of 10% when PIM enabling one tile per subarray. Using a smaller TRD, this area can be cut in less than half and still provide impressive speedups over prior work. CORUSCANT bulk-bitwise capabilities are $\geq 1.6\times$ faster than DRAM PIM, 2.2–2.8× faster than SPIM and 4.2–5.8× faster than ELP^{2}IM, respectively, depending on TRD size, for CNN inference.

CORUSCANT provides a natural capability to add $\{3,5,7\}$-modular redundancy for fault tolerance and can achieve $<5\times10^{-12}$ error rate with TMR while remaining faster than state-of-the-art PIM without fault tolerance. In our future work, we plan to explore additional in-memory capabilities such as floating-point operations and other intrinsic operations required for accelerated on-line training and other compute intensive applications.

References

[1] O. Villa, D. R. Johnson, M. Oconnor, E. Bolotin, D. Nellans, J. Luitjens, N. Sakharnykh, P. Wang, P. Micikevicius, A. Scudiero *et al.*, "Scaling the power wall: a path to exascale," in *SC'14: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 2014, pp. 830–841.

[2] S. A. McKee, "Reflections on the memory wall," in *Proceedings of the 1st conference on Computing frontiers*, 2004, p. 162.

[3] D. Molka, D. Hackenberg, R. Schöne, and M. S. Müller, "Characterizing the energy consumption of data transfers and arithmetic operations on x86- 64 processors," in *International conference on green computing*. IEEE, 2010, pp. 123–133.

[4] 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.

[5] 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)*. IEEE, 2017, pp. 273–287.

[6] S. Li, C. Xu, Q. Zou, J. Zhao, Y. Lu, and Y. Xie, "Pinatubo: A processing-in-memory architecture for bulk bitwise operations in emerging non-volatile memories," in *Proceedings of the 53rd Annual Design Automation Conference*, 2016, pp. 1–6.

[7] H. Yu, Y. Wang, S. Chen, W. Fei, C. Weng, J. Zhao, and Z. Wei, "Energy efficient in-memory machine learning for data intensive image-processing by non-volatile domain-wall memory," in *2014 19th Asia and South Pacific Design Automation Conference (ASP-DAC)*, 2014, pp. 191–196.

[8] B. Liu, S. Gu, M. Chen, W. Kang, J. Hu, Q. Zhuge, and E. H.-M. Sha, "An efficient racetrack memory-based processing-in-memory architecture for convolutional neural networks," in *2017 IEEE International Symposium on Parallel and Distributed Processing with Applications and 2017 IEEE*

International Conference on Ubiquitous Computing and Communications (ISPA/IUCC). IEEE, 2017, pp. 383–390.

[9] M. Zabihi, Z. I. Chowdhur, Z. Zhao, U. R. Karpuzcu, J.-P. Wang, and S. S. Sapatnekar, "In-memory processing on the spintronic cram: From hardware design to application mapping," *IEEE Transactions on Computers*, vol. 68, no. 8, pp. 1159–1173, 2018.

[10] N. Talati, S. Gupta, P. Mane, and S. Kvatinsk, "Logic design within memristive memories using memristor-aided logic (magic)," *IEEE Transactions on Nanotechnology*, vol. 15, no. 4, pp. 635–650, 2016.

[11] S. Li, D. Niu, K. T. Malladi, H. Zheng, B. Brennan, and Y. Xie, "Drisa: A dram-based reconfigurable in-situ accelerator," in *2017 50th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2017, pp. 288–301.

[12] A. Subramaniyan and R. Das, "Parallel automata processor," in *Proceedings of the 44th Annual International Symposium on Computer Architecture*, ser. ISCA '17. New York, NY, USA: Association for Computing Machinery, 2017, p. 600–612. [Online]. Available: https://doi.org/10.1145/3079856.3080207

[13] H.-S. P. Wong, S. Raoux, S. Kim, J. Liang, J. P. Reifenberg, B. Rajendran, M. Asheghi, and K. E. Goodson, "Phase change memory," *Proceedings of the IEEE*, vol. 98, no. 12, pp. 2201–2227, 2010.

[14] H. Luo, Q. Liu, J. Hu, Q. Li, L. Shi, Q. Zhuge, and E. H.-M. Sha, "Write energy reduction for pcm via pumping efficiency improvement," *ACM Transactions on Storage (TOS)*, vol. 14, no. 3, pp. 1–21, 2018.

[15] W. Kang, H. Wang, Z. Wang, Y. Zhang, and W. Zhao, "In-memory processing paradigm for bitwise logic operations in stt–mram," *IEEE Transactions on Magnetics*, vol. 53, no. 11, pp. 1–4, 2017.

[16] L. Song, Y. Zhuo, X. Qian, H. Li, and Y. Chen, "Graphr: Accelerating graph processing using reram," in *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2018, pp. 531–543.

[17] P. Yao, H. Wu, B. Gao, J. Tang, Q. Zhang, W. Zhang, J. J. Yang, and H. Qian, "Fully hardware-implemented memristor convolutional neural network," *Nature*, vol. 577, no. 7792, pp. 641–646, 2020.

[18] S. S. P. Parkin, M. Haashi, and L. Thomas, "Magnetic domain-wall racetrack memory," *Science*, vol. 320, no. 5874, pp. 190–194, Apr. 2008.

[19] Q. Hu, G. Sun, J. Shu, and C. Zhang, "Exploring main memory design based on racetrack memory technology," in *Proceedings of the 26th edition on Great Lakes Symposium on VLSI*. ACM, 2016, pp. 397–402.

[20] D. Wang, L. Ma, M. Zhang, J. An, H. H. Li, and Y. Chen, "Shift-optimized energy-efficient racetrack-based main memory," *Journal of Circuits, Systems and Computers*, vol. 27, no. 05, p. 1850081, 2018. [Online]. Available: https://doi.org/10.1142/S0218126618500810

[21] H. A. Khouzani, P. Fotouhi, C. Yang, and G. R. Gao, "Leveraging access port positions to accelerate page table walk in dwm-based main memory," in *Design, Automation Test in Europe Conference Exhibition (DATE), 2017*, 2017, pp. 1450–1455.

[22] H. Zhang, C. Zhang, Q. Hu, C. Yang, and J. Shu, "Performance analysis on structure of racetrack memory," in *2018 23rd Asia and South Pacific Design Automation Conference (ASP-DAC)*, 2018, pp. 367–374.

[23] A. A. Khan, F. Hameed, R. Bläsing, S. S. P. Parkin, and J. Castrillon, "Shiftsreduce: Minimizing shifts in racetrack memory 4.0," *ACM Trans. Archit. Code Optim.*, vol. 16, no. 4, dec 2019. [Online]. Available: https://doi.org/10.1145/3372489

[24] R. Bläsing, A. A. Khan, P. C. Filippou, C. Garg, F. Hameed, J. Castrillon, and S. S. Parkin, "Magnetic racetrack memory: From physics to the cusp of applications within a decade," *Proceedings of the IEEE*, vol. 108, no. 8, pp. 1303–1321, 2020.

[25] K. Roxy, S. Ollivier, A. Hoque, S. Longofono, A. K. Jones, and S. Bhanja, "A novel transverse read technique for domain-wall "racetrack" memories," *IEEE Transactions on Nanotechnology*, vol. 19, pp. 648–652, 2020.

[26] u. Zhang, W. Zhao, D. Ravelosona, J.-O. Klein, J.-V. Kim, and C. Chappert, "Perpendicular-magnetic-anisotrop cofeb racetrack memory," *Journal of Applied Phsics*, vol. 111, no. 9, p. 093925, 2012.

[27] R. Venkatesan, M. Sharad, K. Roy, and A. Raghunathan, "Dwm-tapestri-an energy efficient all-spin cache using domain wall shift based writes," in *Proc. of DATE*, 2013, pp. 1825–1830.

[28] S. Ollivier, D. Kline Jr., R. Kawsher, R. Melhem, S. Banja, and A. K. Jones, "Leveraging transverse reads to correct alignment faults in domain wall memories," in *Proceedings of the IEEE/IFIP Dependable Systems and Networks Conference (DSN)*, Portland, OR, June 2019.

[29] C. Zhang, G. Sun, X. Zhang, W. Zhang, W. Zhao, T. Wang, u. Liang, o. Liu, u. Wang, and J. Shu, "Hi-fi plaback: Tolerating position errors in shift operations of racetrack memory," in *ACM SIGARCH Computer Architecture News*, vol. 43-3. ACM, 2015, pp. 694–706.

[30] Y. M. Chee, A. Vardy, V. K. Vu, and E. Yaakobi, "Coding for transverse-reads in domain wall memories," in *2021 IEEE International Symposium on Information Theory (ISIT)*, 2021, pp. 2924–2929.

[31] S. Ollivier, S. Longofono, P. Dutta, J. Hu, S. Bhanja, and A. K. Jones, "Toward comprehensive shifting fault tolerance for domain-wall memories with piett," *IEEE Transactions on Computers*, pp. 1–14, 2022.

[32] S. M. Seedzadeh, R. Maddah, A. Jones, and R. Melhem, "Leveraging ecc to mitigate read disturbance, false reads and write faults in stt-ram," in *2016 46th Annual IEEE/IFIP International Conference on Dependable Sstems and Networks (DSN)*. IEEE, 2016, pp. 215–226.

[33] J. Yue and Y. Zhu, "Exploiting subarrays inside a bank to improve phase change memory performance," in *2013 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, 2013, pp. 386–391.

[34] Y. Kim, V. Seshadri, D. Lee, J. Liu, and O. Mutlu, "A case for exploiting subarra-level parallelism (salp) in dram," in *2012 39th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2012, pp. 368–379.

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

[36] R. Venkatesan, V. Kozhikkottu, C. Augustine, A. Rachowdhur, K. Roy, and A. Raghunathan, "Tapecache: a high density, energy efficient cache based on domain wall memory," in *Proc. of ISLPED)*, 2012, pp. 185–190.

[37] 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. [Online]. Available: https://doi.org/10.1145/3352460.3358260

[38] G. Yu, P. Upadhyaya, Y. Fan, J. Alzate, W. Jiang, K. Wong, S. Takei, S. Bender, L. Chang, Y. Jiang, M. Lang, J. Tang, Y. Wang, Y. Tserkovnyak, P. Amiri, and K. Wang, "Switching of perpendicular magnetization by spin-orbit torques in the absence of external magnetic fields," *Nature Nanotechnology*, vol. 9, no. 7, pp. 548–554, Jul. 2014.

[39] A. Razavi, H. Wu, Q. Shao, C. Fang, B. Dai, K. Wong, X. Han, G. Yu, and K. L. Wang, "Deterministic spin–orbit torque switching by a light-metal insertion," *Nano letters*, vol. 20, no. 5, pp. 3703–3709, 2020.

[40] P. Dutta, A. Lee, K. L. Wang, A. K. Jones, and S. Bhanja, "A multi-domain magneto tunnel junction for racetrack nanowire strips," arXiv, 2022. [Online]. Available: https://arxiv.org/abs/2205.12494

[41] Q. Deng, L. Jiang, Y. Zhang, M. Zhang, and J. Yang, "Dracc: a dram based accelerator for accurate cnn inference," in *2018 55th ACM/ESDA/IEEE Design Automation Conference (DAC)*, 2018, pp. 1–6.

[42] V. Lefèvre, "Multiplication by an integer constant," INRIA, Tech. Rep. RR-4192, 2001, ffinria-00072430f.

[43] R. Bernstein, "Multiplication by integer constants," *Software: Practice and Experience*, vol. 16, no. 7, pp. 641–652, 1986. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/spe.4380160704

[44] J. Sim, H. Seol, and L.-S. Kim, "Nid: processing binar convolutional neural network in commodity dram," in *2018 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. IEEE, 2018, pp. 1–8.

[45] L.-N. Pouchet *et al.*, "Polybench: The polyhedral benchmark suite," *available online*, vol. 437, pp. 1–1, 2012, http://www.cs.ucla.edu/pouchet/software/polybench.

[46] C.-Y. Chan and Y. E. Ioannidis, "Bitmap index design and evaluation," in *Proceedings of the 1998 ACM SIGMOD international conference on Management of data*, 1998, pp. 355–366.

[47] X. Dong, C. Xu, Y. Xie, and N. P. Jouppi, "Nvsim: A circuit-level performance, energy, and area model for emerging nonvolatile memory," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 31, no. 7, pp. 994–1007, 2012.

[48] C. Zhang, G. Sun, W. Zhang, F. Mi, H. Li, and W. Zhao, "Quantitative modeling of racetrack memory, a tradeoff among area, performance, and power," in *Design Automation Conference (ASP-DAC), 2015 20th Asia and South Pacific*. IEEE, 2015, pp. 100–105.

[49] North Carolina State University, "Freepdk45." [Online]. Available: https://research.ece.ncsu.edu/eda/freepdk/freepdk45/

[50] E. Chen, D. Apalkov, Z. Diao, A. Driskill-Smith, D. Druist, D. Lottis, V. Nikitin, X. Tang, S. Watts, S. Wang *et al.*, "Advances and future prospects of spin-transfer torque random access memory," *IEEE Transactions on Magnetics*, vol. 46, no. 6, pp. 1873–1878, 2010.

[51] 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, 2019.

[52] A. Boroumand, S. Ghose, Y. Kim, R. Ausavarungnirun, E. Shiu, R. Thakur, D. Kim, A. Kuusela, A. Knies, P. Ranganathan, and O. Mutlu, "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*, ser. ASPLOS '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 316–331. [Online]. Available: https://doi.org/10.1145/3173162.3173177

[53] O. Naji, C. Weis, M. Jung, N. Wehn, and A. Hansson, "A High-level DRAM Timing, Power and Area Exploration Tool," in *2015 International Conference on Embedded Computer Sstems: Architectures, Modeling, and Simulation (SAMOS)*, Jul 2015, pp. 149–156.

[54] S. Mittal, R. Wang, and J. Vetter, "DESTINY: A Comprehensive Tool with 3D and Multi-Level Cell Memory Modeling Capability," *Journal of Low Power Electronics and Applications*, vol. 7, no. 3, 2017.

[55] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, 1998.

[56] A. Krizhevsk, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," *Advances in neural information processing sstems*, vol. 25, pp. 1097–1105, 2012.

[57] C. Zhu, S. Han, H. Mao, and W. J. Dally, "Trained ternary quantization," *arXiv preprint arXiv:1612.01064*, 2016.

[58] A. Shafiee, A. Nag, N. Muralimanohar, R. Balasubramonian, J. P. Strachan, M. Hu, R. S. Williams, and V. Srikumar, "Isaac: A convolutional neural network accelerator with in-situ analog arithmetic in crossbars," *ACM SIGARCH Computer Architecture News*, vol. 44, no. 3, pp. 14–26, 2016.

[59] W. Jiang, E. H.-M. Sha, X. Zhang, L. Yang, Q. Zhuge, Y. Shi, and J. Hu, "Achieving super-linear speedup across multi-fpga for real-time dnn inference," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 18, no. 5s, p. 67, 2019.

[60] M. Scheinfein and E. Price, "Llg micromagnetics simulator, software for micromagnetic simulations," *see http://llgmicro. home. mindspring. com. Google Scholar*, 1997.

[61] S. Parkin and S.-H. Yang, "Memory on the racetrack," *Nature Nanotech*, vol. 10, pp. 195–198, 2015. [Online]. Available: https://doi.org/10.1038/nnano.2015.41

[62] T. Na, S. H. Kang, and S.-O. Jung, "Stt-mram sensing: A review," *IEEE Transactions on Circuits and Systems II: Express Briefs*, vol. 68, no. 1, pp. 12–18, 2021.