# Brain-inspired Cognition in Next Generation Racetrack Memories

ASIF ALI KHAN*, Center for Advancing Electronics Dresden (cfaed), TU Dresden, Germany
SÉBASTIEN OLLIVIER* and STEPHEN LONGOFONO, University of Pittsburgh, USA
GERALD HEMPEL and JERONIMO CASTRILLON, Center for Advancing Electronics Dresden (cfaed), TU Dresden, Germany
ALEX K. JONES, University of Pittsburgh, USA

*Hyperdimensional computing* (HDC) is an emerging computational framework inspired by the brain that operates on vectors with thousands of dimensions to emulate cognition. Unlike conventional computational frameworks that operate on numbers, HDC, like the brain, uses high dimensional random vectors and is capable of one-shot learning. HDC is based on a well-defined set of arithmetic operations and is highly error-resilient. The core operations of HDC manipulate HD vectors in bulk bit-wise fashion, offering many opportunities to leverage parallelism. Unfortunately, on conventional von Neumann architectures, the continuous movement of HD vectors among the processor and the memory can make the cognition task prohibitively slow and energy-intensive. Hardware accelerators only marginally improve related metrics. In contrast, even partial implementations of an HDC framework inside memory can provide considerable performance/energy gains as demonstrated in prior work using memristors. This paper presents an architecture based on *racetrack memory* (RTM) to conduct and accelerate the entire HDC framework within memory. The proposed solution requires minimal additional CMOS circuitry by leveraging a read operation across multiple domains in RTMs called *transverse read* (TR) to realize exclusive-or (XOR) and addition operations. To minimize the CMOS circuitry overhead, an RTM nanowire-based counting mechanism is proposed. Using language recognition as the example workload, the proposed RTM HDC system reduces the energy consumption by 8.6× compared to the state-of-the-art in-memory implementation. Compared to dedicated hardware design realized with an FPGA, RTM-based HDC processing demonstrates 7.8× and 5.3× improvements in the overall runtime and energy consumption, respectively.

CCS Concepts: • **Hardware → Memory and dense storage**; **Spintronics and magnetic technologies**; • **Computing methodologies → Neural networks**; • **Computer systems organization** → *Embedded hardware*; *Reconfigurable computing*.

Additional Key Words and Phrases: High Dimensional Computing, Hyper Dimensional Computing, Racetrack Memory, In-memory Computing, Language Recognition, Domain Wall Memory, Embedded Systems, Processing-in-Memory

*Authors contributed equally to the paper.

Authors' addresses: Asif Ali Khan, asif_ali.khan@tu-dresden.de, Center for Advancing Electronics Dresden (cfaed), TU Dresden, Helmholtzstrasse 18 , 01069, Dresden, Germany; Sébastien Ollivier, sbo15@pitt.edu; Stephen Longofono, stl77@pitt.edu, University of Pittsburgh, 1238 Benedum Hall, 3700 O'Hara Street, Pittsburgh, Pennsylvania, USA, 15260; Gerald Hempel, gerald.hempel@tu-dresden.de; Jeronimo Castrillon, jeronimo.castrillon@tu-dresden.de, Center for Advancing Electronics Dresden (cfaed), TU Dresden, Helmholtzstrasse 18, 01069, Dresden, Germany; Alex K. Jones, akjones@pitt.edu, University of Pittsburgh, 1238 Benedum Hall, 3700 O'Hara Street, Pittsburgh, Pennsylvania, USA, 15260.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
© 2022 Association for Computing Machinery.
1539-9087/2022/6-ART1 $15.00
https://doi.org/10.1145/1122445.1122456

**ACM Reference Format:**
Asif Ali Khan, Sébastien Ollivier, Stephen Longofono, Gerald Hempel, Jeronimo Castrillon, and Alex K. Jones. 2022. Brain-inspired Cognition in Next Generation Racetrack Memories. *ACM Trans. Embedd. Comput. Syst.* XX, Y, Article 1 (June 2022), 29 pages. https://doi.org/10.1145/1122445.1122456

## 1 INTRODUCTION

The success of machine learning has fueled the transformation of industry and society in recent decades. A key factor for the ubiquity of these learning algorithms is their use in mobile devices such as smartphones, tablets, or sensor networks. However, classic approaches such as deep learning require enormous computing and power resources [57]. For example, training of a single transformer-based deep learning model requires weeks on modern GPUs and produces carbon footprints (a proxy for energy consumption) $\approx 5\times$ more than the entire lifetime carbon footprint of a passenger car [56]. Unfortunately, these characteristics are at odds with the requirements of many IoT devices, namely limited bandwidth, memory and compute power, and battery capacity. Architectural innovations such as near-memory and in-memory computing, along with the alternate models for machine learning such as hyperdimensional computing, substantially reduce the area and energy consumption of cognitive-inspired computing systems without compromising accuracy [21].

The idea of *hyperdimensional computing* (HDC) is inspired by biological systems that generally combine sufficient accuracy with a very high energy efficiency. Compared to conventional machine learning models, HDC is more robust and error-resilient [19] as well as more compute and energy efficient [8, 53]. Moreover, HDC provides comparable accuracy to the highest fidelity ML models (cf. Table 2 in [10]). HDC frameworks mainly operate on binary or bi-polar hypervectors, typically having thousands of dimensions [19]. The *base* or *seed* hypervectors are randomly generated and describe input features. In HDC training, class hypervectors are generated by performing a set of basic algebraic operations (XOR, permutation, addition, thresholding, and multiplication) that combine several hypervectors and the properties of the desired class. In inference, the same encoding is applied to the input data to generate a query hypervector and reason about a given dataset. The query hypervector is then classified by performing a similarity match operation.

With conventional von Neumann machines, shuttling of hypervectors between the memory and the processor makes the overall classification process prohibitively slow. To overcome this, state-of-the-art proposals use accelerators and near-memory processing to achieve parallelism and energy efficiency [5, 48, 49]. Since the algebraic operations in most of the HDC frameworks are memory intensive and inherently parallel, they are particularly well-suited for in-memory computing. Furthermore, in most emerging memory technologies, the physical properties of the memory cells can be exploited to realize some, if not all, HDC operations in place [13, 50].

In one of the most recent works, an entire HDC framework is implemented on an integrated system using memristor crossbars with additional CMOS logic [21]. Specifically, the multiplication operation required for "binding" and "similarity search" operations is implemented using phase change memory (PCM) crossbars while the addition, permutation and thresholding operations are realized by additional near-memory CMOS logic. Although the in-PCM HDC system significantly reduces energy consumption (by more than $6\times$), it has three major limitations. First, the additional CMOS logic incurs large area and energy penalties. In the ideal case, the entire framework should be implemented using memory devices. Second, the write operation in resistive memories such as PCM is extremely expensive (in terms of latency and energy) and induces wear on the endurance-limited cells. Although the proposed solutions avoid repetitive programming of the memristive devices, the fundamental problem of expensive writes and finite endurance remains. Third, memristive devices compute values in the analog domain. Besides accuracy implications, which are not as severe due

to the inherent resilience of HDC, analog computation requires power hungry [55] back-and-forth conversion between the analog and digital domains (via ADC/DAC).

To overcome these challenges, we use another class of emerging nonvolatile memory technologies called *racetrack memory* (RTM) or *domain wall memory* (DWM) [2] to implement the entire HDC framework. An RTM cell consists of a magnetic nanowire that stores multiple data bits in magnetic *domains* and is associated with one or more access ports. RTM promises to realize the entire framework in the digital domain with relatively low additional logic and without compromising on accuracy.

We present *HyperDimensional Computing in Racetrack* (HDCR), a complete in-RTM HDC system where all HDC operations are implemented in RTM using the RTM device characteristics. Namely, a novel access mode called *transverse read* (TR) [51] is used to conduct processing within the RTM [38, 39]. By applying a sub-shift-threshold current across two access points along the nanowire, the resistance state of the nanowire can be used to count '1's at each bit position across multiple adjacent data words within the memory. HDCR leverages the TR operation and makes appropriate changes to the peripheral circuitry to realize the XOR operation, and efficient counters. Together with our design for in-memory majority operation, and "permutation," TR enables all necessary HDC processing operations to be performed in a highly parallel fashion within RTM.

Our experimental results show that for the well-known use case of language recognition, our HDC system is an order of magnitude faster than the state-of-the-art FPGA solution and consumes 5.3× and 8.6× less energy compared to the state-of-the-art FPGA and PCM-crossbar solutions, respectively.

The main contributions of this paper are as follows:

(1) We present a complete HDC system with precise control and datapaths based on nonvolatile racetrack memory.
(2) For the rotation operation, we make necessary changes to the RTM row buffer to enable rotation of HD vectors with a simple copy (read and write) operation.
(3) We propose a first RTM nanowires-based counter design to perform the majority operation and compute the Hamming weight.
(4) For binding, we implement the XOR logic by doing a transverse read operation and using the modified row buffer to infer the result.
(5) For bundling, we use RTM counters to find the majority output at each position in the hypervectors.
(6) For comparison with the class vectors, we compute the Hamming distance between the query vector and each class vector leveraging a TR-based XOR operation and the RTM counter.
(7) We evaluate our system on a standard benchmark and compare the runtime and energy consumption with state-of-the-art FPGA [48] and in-PCM implementations [21].

The remainder of this paper is organized as follows: Section 2 provides background information about HDC, language recognition, RTM and TR. Section 3 proposes the architectural modification needed to perform operations inside RTM and explains the implementation of our RTM counter. Section 4 explains different HDCR modules and their integration to perform HDC operations in RTM. Section 5 evaluates HDCR, demonstrating the energy and latency advantages of using RTM. Section 6 presents some of the most related work in the literature. Finally, Section 7 concludes the paper.

## 2 BACKGROUND

In this section, we introduce the fundamentals of HDC, its major operations, and main components. We then describe our use case and provide details on classes and input features/symbols. Finally, we provide background on RTM technology, its properties and organization, and the working principles of the transverse read operation.

### 2.1 Hyperdimensional Computing

Hyperdimensional computing, also referred to as brain-inspired computing, is based on the observation that neural activity patterns can be regarded as one fundamental component behind cognitive processes. These patterns can be modeled by leveraging the mathematical properties of hyperdimensional spaces. In conjunction with a well-defined algebra, they can be used to implement machine learning tasks with less computational effort than other approaches such as the support vector machine (SVM) algorithm [12]. Since the dimension $D$ of the hyperdimensional space is on the order of $10^4$, this approach is extremely robust to variation and errors within its hypervectors.

In HD computing, each *hypervector* (HV) describes a unique point in space and encodes either a feature, a group of features, or a class in the given machine learning problem. As shown in Fig. 1-I, the base or seed hypervectors describe input features, and are randomly generated. In HDC training, a set of algebraic operations—*i.e.*, binding, bundling, permutation, and similarity check—are performed on the seed hypervectors and their intermediate results are used to generate *class* hypervectors. Each class hypervector represents a class in the data set. In HDC inference, the same encoding is applied to the input data to generate a *query* hypervector. The query hypervector is then classified into one of the classes by performing a similarity check operation.

Various HDC frameworks exist that implement HDC in different ways such as (1) using different types of hypervectors (bipolar, binary, integer, etc.), (2) using a different distribution of elements in hypervectors (sparse and dense hypervectors), and (3) employing a different set of algebraic operations. A detailed comparison of these frameworks is presented in [48, 54]. Since we focus on a digital, in-memory implementation of HDC, we consider a binary HDC subset. Thus hypervectors consist of binary values and the framework leverages Boolean operations to implement the required algebraic operations. For the hypervectors, we consider the dimensionality of a hypervector $D = 8192$ and a probability of $P = 0.5$ for each component to be a one or a zero. This is because, for our selected use case, $D = 8192$ does not have any considerable impact on the accuracy (only reduces it from 97.8% to 97.7%) while still leaving the memory to be used by other general-purpose applications.

We use the Hamming distance $d_H(\vec{a}, \vec{b})$ metric to compare the hypervectors $\vec{a}$ and $\vec{b}$, resulting in the normalized number of dissimilar elements of both vectors. For large vector sizes, the Hamming distance between random vector pairs, in 98% of the cases, results in $d_H(\vec{a}, \vec{b}) = D/2$. In this context, we classify any two vectors as similar ($d_H < 0.5$) or dissimilar ($d_H \geq 0.5$). Since $d_H(\vec{a}, \vec{b}) \approx B(D, P = 1/2)$ with $B$ representing the binomial distribution, random, *i.e.*, unrelated, vectors are unlikely to deviate from $D/2$. Thus, HDC defines sufficiently dissimilar (*e.g.*, $d_H \geq 0.5$) vectors to be *orthogonal*[1].

In the context of HDC for binary hypervectors, relevant algebraic operations are:

- **Binding** is used for combining related hypervectors. This operation is implemented as an element-wise XOR operation between $N$ hypervectors *e.g.*, $\vec{c} = \vec{x_1} \oplus \vec{x_2} \ldots \vec{x_N}$ binds $\vec{x_i} : i = 1, 2, ..., N$ together.

[1] Mathematically, orthogonal vectors would have $d_H = 1$, HDC relaxes this definition to $d_H \geq 0.5$ because it is attempting to distinguish between *similar* and *dissimilar* vectors. HDC redefines vectors with $d_H = 1$ as *diametrically opposed.*

- **Permutation** is used to generate a new hypervector that is orthogonal to the original hypervector by performing a reversible operation. The permutation is a unary operation $\vec{x_p} = \rho(\vec{x})$ such that the resulting vector $\vec{x_p}$ is orthogonal to $\vec{x}$. In the context of this work, we use piece-wise circular shifts to perform this operation (see Section 4.3.1). Rotating a hypervector $n$ times is expressed as $\vec{x_p} = \rho^n(\vec{x})$.
- **Bundling** is used to generate a hypervector representing a set of hypervectors. This operation is implemented by performing the vector sum and element-wise thresholding, also referred to as the majority operation. For an even number of binary hypervectors, the tie is broken by a fixed random hypervector. The bundling operation generates a representative hypervector which is non-orthogonal to the operand hypervectors.
- **Similarity Check**: The similarity check operation compares the query hypervector to all class hypervectors to find the closest match. Different frameworks use a variety of similarity metrics. For this work, we use Hamming distance and compare the Hamming weights of the query and class hypervectors. The operation is implemented as an XOR followed by the population count operation (see Section 4.4.)

## 2.2 Use Case: Language Recognition

In the context of this work, we use the *language recognition* (LR) classification task, which has already been used as a benchmark by other HDC approaches in the literature [21, 48, 49]. With this example application, we demonstrate the scalability and efficiency of our architecture compared to the state-of-the-art FPGA [48] and in-memory [21] implementations. We use the language recognition code published on [47] that classifies an input text to one of 22 European languages. The input features consist of 26 letters of the Latin alphabet and the space character (represented by $\tau$). As a first step in building the hyperdimensional (HD) model, hypervectors are generated for all input letters and are stored in an *item memory* (IM) $\Theta = \{a \longrightarrow \vec{a}, b \longrightarrow \vec{b}, \ldots, z \longrightarrow \vec{z}, \tau \longrightarrow \vec{\tau}\}$ (see Fig. 1-I). The dimensionality of the hypervectors ($D = 8192$) is carefully chosen to ensure better utilization of the memory architecture.

After the IM is created, the training of the HD model is carried out using one text for each of the 22 languages. In order to model the probability distribution of individual letters in the respective language, the text is broken down into substrings of length $N$ called *N-grams*. In the binding

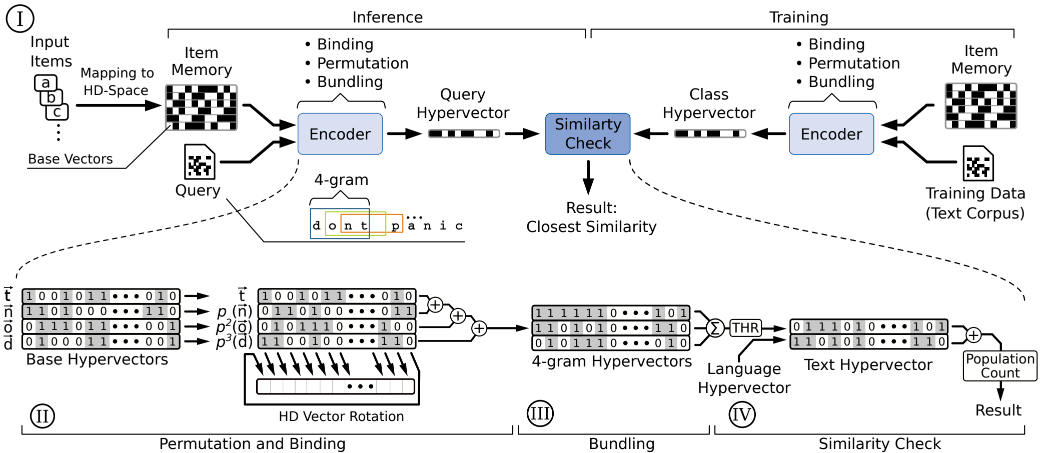



Fig. 1. An overview of the HDC operations

operation, a hypervector is generated for each N-gram of the input text, which is subsequently combined by the bundling operation into a single hypervector. This is in contrast to models which use dictionaries and banks of phrases, which increases the complexity of similarity checking without a commensurate advantage in accuracy or efficiency [58]. For example, the first N-gram of the phrase *"dont panic"* for $N = 4$ would be *"dont"*. This is encoded to a single N-gram vector, as shown in Fig. 1-II, by permuting and XORing the individual hypervectors from the IM ($\vec{\Theta}$) as follows: $\vec{\Phi}_{dont} = \rho^3(\vec{d}) \oplus \rho^2(\vec{o}) \oplus \rho(\vec{n}) \oplus \vec{t}$. Due to the properties of the selected encoding, all generated N-gram vectors $V_z = \{\vec{\Phi}_{dont}, \vec{\Phi}_{ont\tau}, \ldots, \vec{\Phi}_{anic}\}$ are orthogonal. Finally, the language vector $\vec{\mathcal{T}}$ is generated as follows: $\vec{\mathcal{T}} = \text{Majority}\ (\vec{\Phi}_{dont}, \vec{\Phi}_{ont\tau}, \ldots, \vec{\Phi}_{anic})$ (see Fig. 1-III). In the training phase, $\vec{\mathcal{T}}$ represents a (language) class hypervector $\vec{\mathcal{L}}$ and is stored in the associative memory. In the inference phase of HDC, $\vec{\mathcal{T}}$, the query hypervector, represents the input sentences or phrases and is generated with exactly the same operations.

After the query hypervector is generated, the distance between the query vector and the class vectors must be determined. As shown in Fig. 1-IV and mentioned in Section 1, this is done by calculating the Hamming distance between the input vector and each of the 22 class vectors $d_h(\vec{\mathcal{T}}, \vec{\mathcal{L}}) = cnt_p(\vec{\mathcal{T}} \oplus \vec{\mathcal{L}})$. The Hamming distance is computed by performing an element-wise XOR operation followed by a population count on the resultant vector. As a final step, $\vec{\mathcal{T}}$ is classified into $\vec{\mathcal{L}}_\xi$ where $\xi = \text{argmin}_{i \in \{1,\ldots,22\}} \left(d_H\left(\vec{\mathcal{T}}, \vec{\mathcal{L}}_i\right)\right)$.

This method is based on the fact that the language vectors lie in a linear space that is spanned by a unique N-gram distribution of the associated language. The class vector with the closest N-gram distribution has the smallest distance to the input vector and represents the resulting language.

### 2.3 Racetrack Memory

The basic storage unit in racetrack memory is a magnetic nanowire that can be grown vertically or horizontally on a silicon wafer, as shown in Fig. 2. The nanoscale magnetic wires, also referred to as tracks or racetracks, can be physically partitioned into tiny magnetic regions called *domains* that are delineated by *domain walls* (DWs) wherever the magnetization changes. This magnetization direction can be based on either in-plane (± X) or perpendicular (± Z) magnetic anisotropy. The state of any given domain exhibits a different resistance when it is parallel/antiparallel to a fixed reference domain, which can be interpreted as bits representing 1s and 0s [2]. Generally, each track in RTM has its associated *access ports* (APs) and can store $K$ bits delineated by $K - 1$ physical notches along the nanowire, where $K$ can be up to 128. The number of APs per nanowire is usually less than the number of domains due to the larger footprint of the APs [69]. This mismatch in the number of domains and APs leads to compulsory *shifts*, *i.e.*, each random access requires two steps to complete: ① *shift* the target domain and *align* it to an AP and ② apply an appropriate voltage/current to *read* or *write* the target bit.

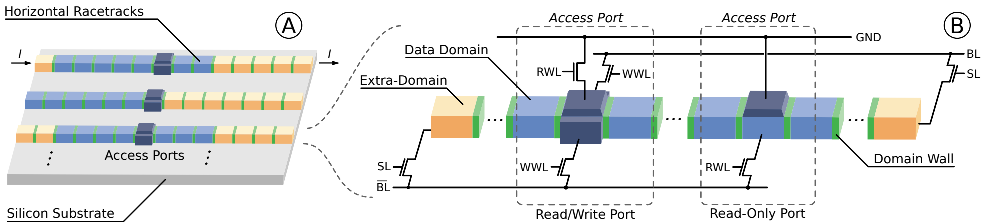



Fig. 2. RTM nanowire structure (A) and anatomy(B).

Shifting is conducted by passing spin-polarized current along the nanowire from either an access point or an endpoint to another access or endpoint; sufficient densities of spin-polarized current can overcome a potential well ("pinning") created at notches and in turn advance all the domain walls toward the next notch position. This inherent behavior of RTM can be imprecise, generating what is known as a "shifting fault" in the literature. Several solutions have been proposed to mitigate this fault mode [1, 38, 67]. Due to shifting, the access latency of RTM is limited by the velocity with which domains move within the nanowire as well as the amount of shifts. The maximum number of domains per track depends on device parameters, but considering the user/application requirements and the number of APs, the number of *addressable* domains per track varies to accommodate shifting each addressable domain to align with any port.

Fig. 2 depicts the major components of an RTM nanowire and its access circuitry. The blue domains represent the actual data stored in memory. The yellow domains are extra domains used to prevent data loss while shifting domain walls (and the data between them) along the nanowire. The dark blue elements and the connected access transistors form read-only or read-write ports. A read-only port 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 [61] where `WWL` is opened and the direction of current flows between `BL` and $\overline{\texttt{BL}}$. Reading is conducted from $\overline{\texttt{BL}}$ through the domain and `RWL` to `GND`.

Similar to contemporary memory technologies, RTM cells are grouped together to form a 2D memory array. To minimize the integration complexity, we deliberately conserve a DRAM hierarchical organization consisting of banks, subarrays, and tiles, as shown in Fig. 3. As illustrated, the basic building block of the RTM array is a group of $T$ nanowires and is referred to as a *domain wall block cluster* (DBC) [23, 59]. A DBC therefore can accommodate $K$ $T$-bit memory objects. Data in a DBC is distributed across nanowires, which facilitates parallel access of all bits belonging to the same data word. Access ports of all $T$ tracks of a DBC point to the same location and domains can be moved together in a lock-step fashion. For our proposed system, we use $K = 32$ and $T = 512$, the standard cache line size, as shown in Fig. 3. Note that for simplicity, we do not show the overhead domains in Fig. 3 and $K$ refers to only addressable domains in the nanowires. We assume 16 DBCs per tile, 16 tiles per subarray. Furthermore, we assume a single *compute in memory* (CIM) tile or *cim-tile* per subarray, capable of performing in-RTM computations (see Section 3).

**RTM strengths, challenges and developments:** Table 1 provides a direct comparison of RTMs to other memory technologies. RTM offers high-performance SRAM comparable latency with extremely low leakage power and higher write endurance compared to other non-volatile memory

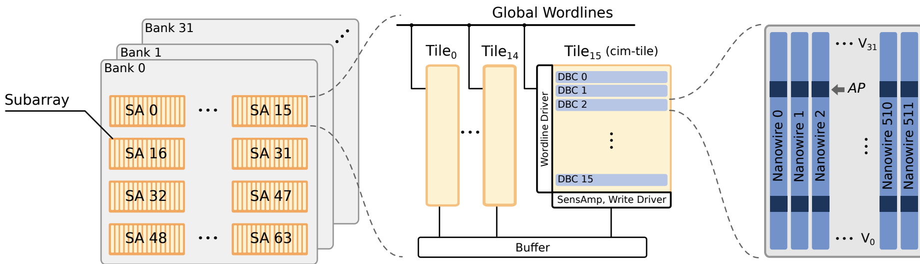



Fig. 3. RTM organization. SA stands for subarray, *domain wall block clusters* (DBCs) for domain wall block cluster, AP for access port, and SensAmp for sense amplifier.

Table 1. Comparison of RTMs with other memory technologies [2]

| | SRAM | DRAM | STT-MRAM | ReRAM | PCM | RaceTrack 4.0 |
|---|---|---|---|---|---|---|
| Cell Size ($F^2$) | 120-200 | 4-8 | 6-50 | 4-10 | 4-12 | ≤ 2 |
| Write Endurance | $\geq 10^{16}$ | $\geq 10^{16}$ | $4 \text{ X } 10^{12}$ | $10^{11}$ | $10^{9}$ | $\geq 10^{16}$ |
| Read Time (ns) | 1–100 | 30 | 3–15 | 10–20 | 5–20 | 3–250† |
| Write Time (ns) | 1–100 | 30 | 3–15 | 20 | >30 | 3–250† |
| Write Energy | Low | Medium | High | High | High | Low |
| Read Energy | Low | Medium | Medium | Medium | Medium | Low |
| Leakage Power | High | Medium | Low | Low | Low | Low |
| Retention Period | Voltage-dependent | 64–512ms | Variable | Years | Years | Years |

† including shift latency

technologies. However, due to the device's sequentiality, RTM access latency and energy consumption depend on the number of required shift operations. In the worst case, the RTM access latency can be 25.6× higher compared to an iso-capacity SRAM [60]. In addition, shifts can also incur position and alignment faults. A number of solutions have been proposed to optimize RTM performance through shift minimization [2]. Additionally, solutions have been proposed to detect and correct RTM misalignments [38, 67].

In recent years, RTMs have seen fundamental breakthroughs in device physics. In the earliest version of RTM [42], controlled movement of domain walls in the nanowires was not only challenging but also extremely slow. Later, the same authors demonstrated accurate movement of domain walls with up to 10× higher velocities [43]. More recently, the field-driving magnetic domain wall mobility has remarkably enhanced to 20 km/sT [25], more than 20× faster compared to the previous version or a two-order of magnitude improvement over the original prototypes. Similarly, moving domain walls in ferromagnetic materials with an exchange coupling torque [3] has shown promise to reduce the critical current density to reduce shift energy. The data access devices, *magnetic tunnel junction*s (MTJs), have also attracted significant interest and have observed considerable improvements in performance and thermal stability by employing different materials (*e.g.*, MgO as a tunneling barrier) and adopting different switching mechanisms (such as spin-orbit torque instead of spin-transfer torque). These newer MTJs allow for ultrafast magnetization switching, in sub-ns, with extremely low incident power [45].

**Transverse Read Operation in RTM:** The *transverse read* (TR) operation is an alternate access mode which conducts reads *along* the nanowire rather than across it [38]. By applying a sub-shift-threshold current at an AP, and performing a normal read at the next nearest AP (for example, between the two access ports in Fig. 2), it is possible to detect how many of the domains between the ports are in a particular magnetic orientation. The resultant magnitudes of the difference of resistances are small compared to the normal access mode, which limits how many domains can be accurately read in this manner without inadvertently shifting the domain walls. However, using a *transverse read distance* (TRD) of five domains can reliably produce a count of domains which are in either magnetic orientation [51].

Prior work used this count to detect misalignment when shifting nanowires [38], but this count can also be used to conduct bitwise logical operations on the data within the TRD [39]. Using a level-detecting sense amplifier, we can detect different voltage thresholds when $0, 1, ..., n$ bits are set, where exceeding any given threshold implies that all lower thresholds are also exceeded. For example, if a TR is conducted across four words at a specific bit position in a nanowire, we derive logical `OR` if any of the thresholds are exceeded, logical `AND` if the threshold for four bits is exceeded, and `XOR` $\iff$ the threshold exceeded $\in \{1, 3, 5\}$. For a fixed TR distance, these levels

can be be used to realize carry-sum operations which can be composed to realize addition and multiplication [39]. In the next section we show how a modified version of these level operations combined with handful of additional CMOS logic gates can be used to implement the fundamental HDC operations.

## 3 ENABLING COMPUTATION IN RACETRACK MEMORY

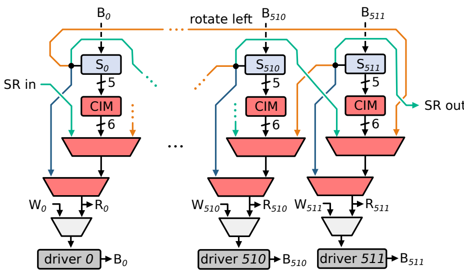
This section presents the extensions to the cim-tile circuitry that enable in-place logical operations and counting in RTMs.

### 3.1 Logical Operations in RTM

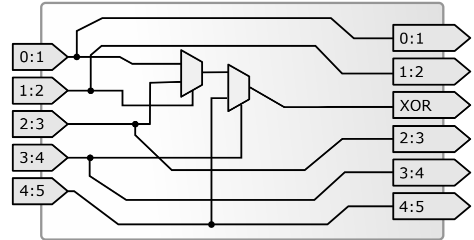
Similar to [48, 49], we use the binary spatter-coding [18] framework that has four primary operations, *i.e.*, `XOR` and circular shift operations for binding, the majority for bundling, and `XOR` for the similarity check as described in Section 2.1.

To implement these operations in RTM, HDCR exploits the nanowires' properties and modifies the peripheral circuitry in selected RTM tiles (see Fig. 3), referred to as compute-in-memory tiles. Concretely, one tile ($T_{15}$) in each subarray is designated as a cim-tile. Fig. 4a shows the necessary support circuitry similar to [39], with the logic required for compute-in-memory operation outlined in red. Sense amplifiers ($S_i$) shown in blue are aligned with access points at bitline $B_i$ to conduct either a normal read at that bit position, or a TR as described in Section 2. During a TR operation, the sense amplifier outputs five bits indicating the five possible reference thresholds corresponding to a particular count of 1s between the access port at $B_i$ and another access port at a $TRD = 5$ distance in the same nanowire. For example, 2:3 indicates that the voltage threshold between 2 and 3 ones was exceeded, indicating that at least 3 ones exist in the TR. To realize TR-based computations, we introduce the CIM block as shown in Fig. 4b. Based on the thresholds representing the count of ones in the TR, and `XOR` is high when only the threshold for 0:1 is high or 2:3 is high with 3:4 being low, or when 4:5 is high. The results of all operations are output simultaneously, to be selected using the multiplexer immediately below the CIM blocks.

During a normal read operation, each sense amplifier outputs the value of the single bit position directly beneath the access port. This output bypasses the CIM tile and feeds directly to the first row of multiplexers to enable a fast read path. This same read path for bit line $B_i$ is routed to the multiplexer for the prior bit line $B_{i-1}$ and the subsequent bit line $B_{i+1}$, shown with orange and




(a) Cim-tile architecture for in-place permutation and logical operations. Additional logic relative to non-cim tiles is shown in red. 'S' is sense amplifier and 'SR' represents shift right.




(b) CIM block gates for logical operations. Inputs i:j indicate that the reference voltages used to distinguish between i and j ones was exceeded.

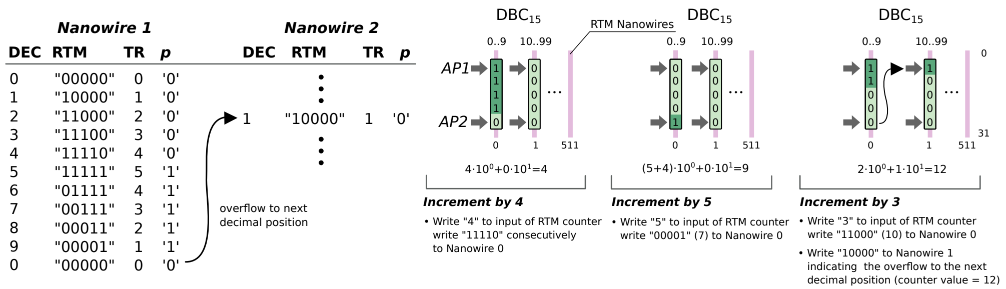
Fig. 4. Cim-tile architecture.

turquoise arrows, respectively. These paths enable circular shifting (permutation) of words by one bit position at a time. Together with the six outputs of the CIM block, the topmost row of multiplexers selects from eight operations on the input data. The second row of multiplexers from the top is added to select from the CIM/shifting data path or the direct read path. The final row of multiplexers and the writeback drivers are identical to the architecture of the non-cim tiles; data for writeback can be fed in from local row buffers $W_i$, or read from the current tile to move data or write back the result of a cim operation.

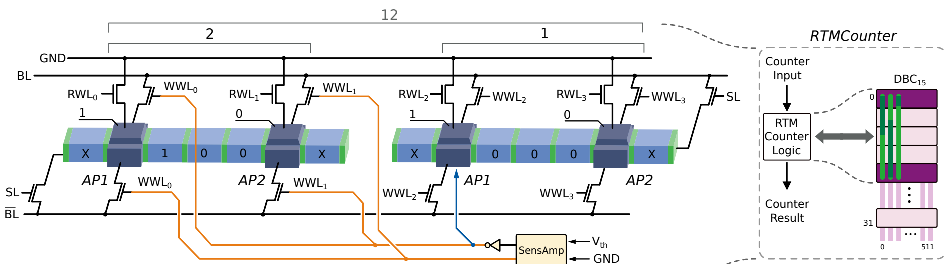
Operating this circuitry requires a new pseudo-instruction in the ISA called *cimop*. Each cimop instruction consists of a source address (*src*), indicating which data to align to the access ports, a *size*, indicating the number of nanowires to be included in the TR operation, and *op*, which selects the cim operation from the topmost row of multiplexers. Note that this psuedo-instruction entails some primitive operations to conduct the alignment and pad operands for sizes less than the TRD. We assume that these primitive operations are scheduled by the compiler and conducted by the memory controller.

## 3.2 Counting in RTM

Fig. 5a presents an overview of the proposed in-RTM counter. It combines the TR operation in the RTM nanowire with the basic read/write operations to realize counters. The RTM nanowires used for counters must be equipped with two read-write APs, necessary for the TR operation. For a base$_{2\cdot X}$ counter, the two access ports in the nanowires must be $X-2$ domains apart, *i.e.*, the TRD in the nanowire must be $X$.




(a) RTM counter overview.




(b) RTM counter implementation.

Fig. 5. RTM counter: overview and details.

In HDCR, we prefer decimal counters for the majority operation and the population count. As such, we use $X = 5$, delimited by APs in dark blue in Fig. 5b and with arrows in Fig. 5a. Note that each nanowire in the RTM counter only uses the domain between the access ports and the number of nanowires in the counter are defined by the counter size. For instance, in a decimal counter, *i.e.*, $X = 5$, a single nanowire can only count between 0 and 9 (see Fig. 5a). If we want to count from 0 and 99, the RTM counter requires at least two nanowires. In general, for a decimal counter having size $C$, an RTM counter requires at least $\lfloor \log_{10}(C) \rfloor + 1$ nanowires.

The RTM counter operates using the same principle as a Johnson counter. Let us assume a two-nanowire decimal counter that can count up to 99 and is initially set to 0 (see Fig. 5a). The counter value at any instant in time is determined by the number of 1s between the APs and the state of bit $P$, the bit under AP2, *i.e.*, the right AP in Fig. 5b. The bit $P$ determines if the counter is in the first or second half of counting, in this case between 0-4 or between 5-9. For the decimal value 0, the $X$ bits are all filled with 0s and hence the bit $P$ is zero. If we want to increment the counter by four, for instance, four 1s need to be shifted under AP1, as shown in Fig. 5a. To count beyond 5, *i.e.*, when all bits between APs including the $P$ bit are 1, 0s are shifted under AP1. The decision to shift a 1 or a 0 under AP1 is controlled by the $P$ bit position: when $P = 0$, we interpret the counter value as the count of ones between access points, and when $P = 1$, we interpret the counter value as ten minus the count of ones (or five plus the count of zeros) between access points. To realize this behavior, toggling the value of $P$ also toggles the value pushed into the nanowire when the counter is incremented, as shown for the decimal value 12 in Fig. 5a. The table of Fig. 5a represents all TR and $P$ combinations and their associated values.

The RTM counter requires nanowires in DBCs to be shifted independently. This drastically increases the shift controller complexity since each nanowire AP position needs to be stored and controlled independently instead of a single position per DBC (512 nanowires). In order to reduce this impact on the nanowire shifting logic, we also used the notion of *transverse write* (TW) [39]. Traditionally, to perform a shift based write under the left AP on Fig. 5b, $\mathtt{RWL}_0$ and one $\mathtt{WWL}_0$ would be closed, the current flows through the fixed layer, one domain and then go to the ground, writing a new value and erasing the previous value under the left AP. However, by closing one $\mathtt{WWL}_0$ and $\mathtt{RWL}_1$, while sending a higher current density, our design can perform a write operation and perform a partial shift along the nanowire rather than between the fixed layer and ground. We called it partial (*i.e.,* segmented) shift since only the bits between the heads are shifted. Thus, a TW from the leftmost AP writes a value under that AP, and shifts the remaining bits between the APs to the right, erasing the bit that was under the right AP.

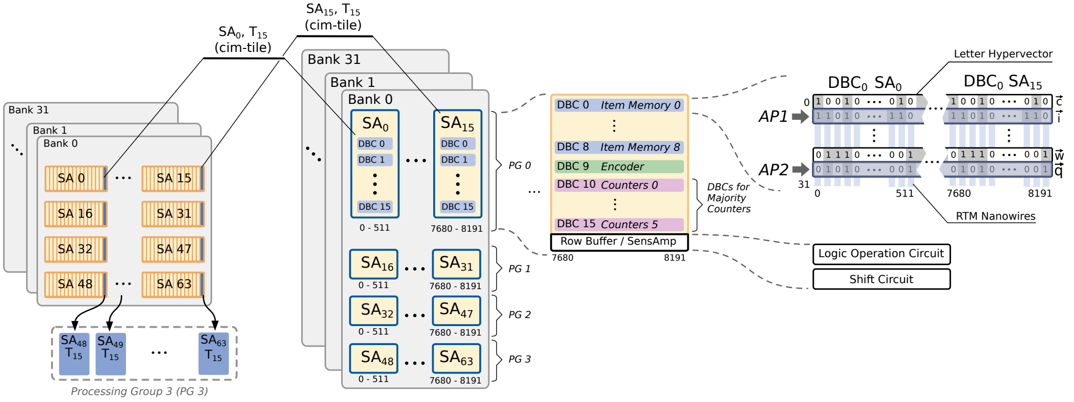
In the next section, we use these in-RTM compute-in-memory concepts and present our proposed architecture for HDCR. Further, we explain how the cim-tile operations implement each of the fundamental HDC operations.

## 4 HYPERDIMENSIONAL COMPUTING IN RACETRACK MEMORY

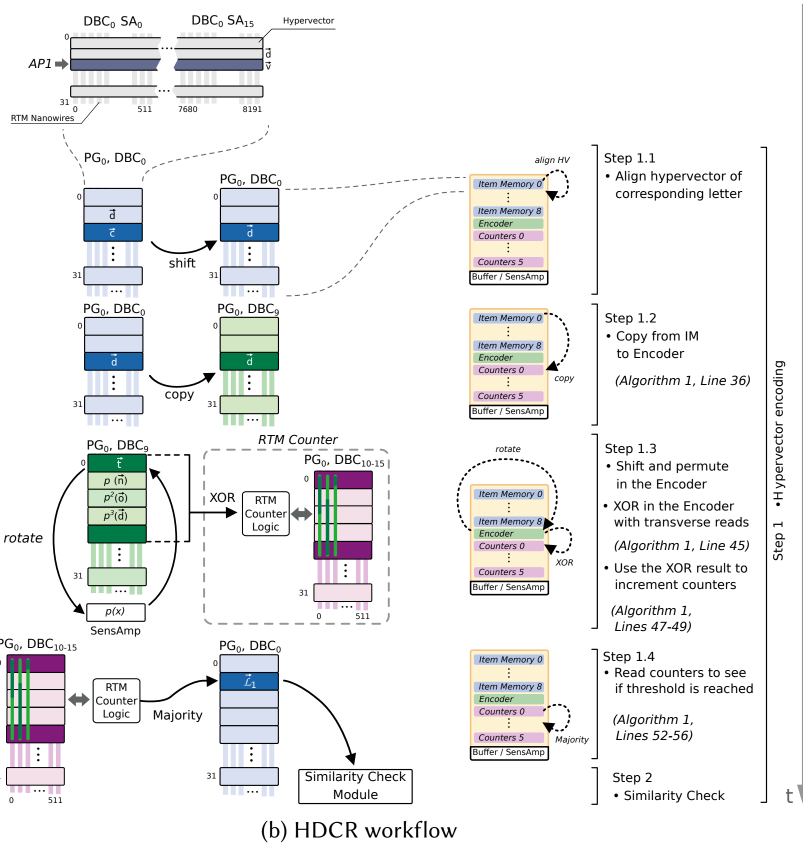
This section presents the implementation details of the proposed HDCR. It provides an overview of the overall system and explains the individual modules and their system integration.

### 4.1 Overview

Fig. 6 presents an overview of the proposed in-RTM HDC system. As explained in Section 2.1, the 27 hypervectors of the input letters are initially mapped to the item memory, 9 DBCs in each subarray as shown in Fig. 6a. Note that for simplicity, we only show the cim-tiles in the subarrays. For the encoding operation, the hypervectors in the item memory are loaded into the encoder module. This requires the hypervectors in the item memory to be shifted and aligned to the port positions in their respective DBCs (Step 1.1 in Fig. 6b). Subsequently, HDCR copies the hypervectors to the encoder




(a) HDCR overview




(b) HDCR workflow

Fig. 6. An overview of the HDCR. The figure shows hypervectors' mapping to cim-tiles and provides detail of the individual operations in HDCR. Note that all tiles shown in the figure are cim-tiles.

**Algorithm 1** HDC Procedures

1: **Global variables**: $V_z \leftarrow \varnothing, \theta, AM, THR$
2: ▷$\theta$ = item memory, $AM$ = Associative memory, (cf. Section 2.2)

3: **function** HDC_Train($LS, \theta$)
4: ▷LS: List of Lang strings for training
5: **for all** $L_i \in LS$ **do**
6: $\vec{\mathcal{L}_i} \leftarrow$ Encode($L_i$)
7: Store $\vec{\mathcal{L}_i}$ in AM
8: **return** $AM$
9: **function** HDC_Classify($L, \theta, AM$)
10: ▷$L$: Text string to be classified
11: $\vec{\mathcal{T}} \leftarrow$ Encode($L$)
12: LangLabel $\leftarrow$ Sim_Check($\vec{\mathcal{T}}$)
13: Display: $L$ is *LangLabel* language.

14: **function** $\rho(\vec{e})$
15: $\vec{\eta} \leftarrow [], \vec{\psi} \leftarrow []$
16: PG_size $\leftarrow \frac{dim(\vec{e})}{T}$ ▷$T = 512$
17: **for** $Itr \leftarrow 0$ to PG_size **do**
18: ▷Rotate left within each SA
19: $\vec{\eta} \leftarrow rol\ (\ \vec{e}_{[512 \cdot Itr]:[512 \cdot (Itr+1)]-1})$
20: ▷Concatenate rotated chunks
21: $\vec{\psi}_{[512 \cdot Itr]:[512 \cdot (Itr+1)]-1} \leftarrow \vec{\eta}$
22: **return** $\vec{\psi}$
23: **function** Sim_Check($\vec{\mathcal{T}}$)
24: **for all** $\vec{\mathcal{L}_i} \in AM$ **do**
25: ▷Implemented with TRs (cf. Sec 4.4)
26: $d_H(\vec{\mathcal{T}}, \vec{\mathcal{L}_i}) \leftarrow$ Hamdist($\vec{\mathcal{T}}, \vec{\mathcal{L}_i}$)
27: ▷Implemented at the MemControl level
28: $\xi = \text{argmin}_{i \in \{1,\ldots,22\}}(d_H(\vec{\mathcal{T}}, \vec{\mathcal{L}_i}))$
29: **return** Label of language class $\vec{\mathcal{L}_\xi}$

30: **function** Encode(String L)
31: $\vec{v_0} = \vec{v_1} = \vec{v_2} = \vec{v_3} \leftarrow 0$
32: $N \leftarrow 4, D \leftarrow 8192$
33: $charCount \leftarrow 0$
34: $counters \leftarrow 0$ ▷$D$ counters in total
35: **for all** $c_i \in L$ **do**
36: $\vec{c_i} \leftarrow \theta(c_i)$ ▷Read HV from IM
37: ▷Rotate HVs in the N-gram
38: $\vec{v_3} \leftarrow \rho(\vec{v_2})$
39: $\vec{v_2} \leftarrow \rho(\vec{v_1})$
40: $\vec{v_1} \leftarrow \rho(\vec{v_0})$
41: $\vec{v_0} \leftarrow \vec{c_i}$
42: $charCount \leftarrow charCount + 1$
43: **if** $charCount \geq N$ **then**
44: ▷XOR with a TR operation
45: $\vec{\phi} = \vec{v_0} \oplus \vec{v_1} \oplus \vec{v_2} \oplus \vec{v_3}$
46: ▷Push counters at all bit positions
47: **for** $Itr \leftarrow 0$ to $D$ **do**
48: **if** $\vec{\phi}_{Itr} == 1$ **then**
49: $\text{counters}_{Itr}$ ++
50:
51: ▷Check all counters' state against THR
52: **for** $Itr \leftarrow 0$ to $D$ **do**
53: **if** $\text{counters}_{Itr} > THR$ **then**
54: $\vec{\mathcal{T}}_{Itr} \leftarrow 1$
55: **else**
56: $\vec{\mathcal{T}}_{Itr} \leftarrow 0$
57: **return** $\vec{\mathcal{T}}$

module implemented in $DBC_9$ of the subarray (see Step 1.2 in Fig. 6b and Line 36 in Algorithm 1). HDCR then permutes the hypervectors in the encoder module (see Lines 38-40 in Algorithm 1) and performs the XOR operation to generate their N-gram hypervector (see Step 1.3 in Fig. 6b and Line 45 in Algorithm 1). Since the N-grams represent $N$ contiguous characters in the input text, the encoder module produces a new N-gram hypervector for each new character in the text. Thus for an input text of $S$ characters, the encoder module generates $S - N + 1$ hypervectors in total.

For each new N-gram hypervector, the counters for each bit position implemented in $DBCs_{10-15}$ are incremented based on the XOR result (see Step 1.4, Lines 47-49 in Algorithm 1). The counting module performs the majority operation on all N-gram hypervectors and generates a single hypervector based on the final counters' state (Step 1.4). In the training phase of the HDC this

generated hypervector represents a language class hypervector ($\vec{\mathcal{L}_i}$). This is stored in the *associative memory* (AM), and the process is repeated for all remaining languages. In contrast, during the inference phase, the resultant hypervector ($\vec{\mathcal{T}_i}$) represents the input text. After generating this hypervector, it is passed on to the similarity search module in Step 2 to classify it into one of the language classes, as shown in Fig. 6b. In the following sections, we provide the implementation details of the individual modules.

## 4.2 Item Memory

The HDC framework operates on $D = 8192$ bit wide binary vectors. Since our DBCs are only 512 bits wide, this requires dividing the hypervectors into 16 chunks of 512 bits each to store the complete 8192-bit hypervector. These chunks can be stored in DBC(s) of the same subarray, as we are doing in Section 4.4, or in the same DBC (*e.g.*, $DBC_i$) across 16 different subarrays. However, for the encoder module in HDCR, to enable performing the TR operation in parallel across all 8192 bit-positions, the HV chunks need to be distributed across different subarrays, as shown in Fig. 6a. This group of 16 subarrays sharing and manipulating chunks of the same hypervectors is referred to as a *processing group* (PG). A PG generates the output of a CIM operation on TRD hypervectors in a single cycle.

For the LR application, the item memory (IM) is composed of 27 hypervectors (HVs), one for each character of the Latin alphabet plus the space character $\tau$ (see Section 2.2). Since a DBC in our proposed system has 32 domains per nanowire, the 27 HVs can be stored in a single DBC (*e.g.*, $DBC_0$) across all subarrays in a PG. However, since each new character consumed from the input text accesses the IM to retrieve its corresponding HV, this tight packing of HVs in a single DBC can lead to a significant number of shift operations in RTM. In the worst case, access to the IM can incur $27 - TRD = 23$ shifts, which stalls the other modules in HDCR and substantially increases the overall runtime. To overcome this, HDCR dedicates 9 DBCs (see Fig. 6a) to the IM and distributes the HVs in the IM such that accessing an HV requires at most one RTM shift. That is, by placing each character HV directly at or adjacent to one of the two access ports, we can access the 18 HVs beneath the access ports without shifting, and the remaining 9 HVs by shifting by one position.

To efficiently map the character HVs into the IM, we profiled each language to rank the frequency of each character in our corpus. The most frequently occurring characters are then placed directly under the access ports, and the remaining characters are distributed among the bit positions adjacent to the access ports.

## 4.3 Encoding

The encoder module transforms the entire language into a representative vector (see Section 2.1). From the implementation perspective, the encoder module performs three major operations, *i.e.*, binding, permutation and bundling (see Fig. 6b). In the following sections, we explain how these operations are implemented.

#### 4.3.1 Binding and Permutation in RTM.

As explained in Section 2.1, the binding operation in HDC generates a new hypervector by `XOR`ing the permuted versions of the $N$ character hypervectors which form each N-gram in the input text.

Initially, all hypervectors of the respective N-gram are iteratively loaded into the encoder module *i.e.*, $DBC_9$ (see step 1.2 in Fig. 6b). Depending on the HVs position in the IM, this may require a shift operation in RTM, as demonstrated in Fig. 6b (step 1.1). In the next step, the hypervectors are rotated by $M$ times, where the value of $M$ for a particular hypervector depends upon its position in the N-gram. This rotation is functionally equivalent to a bitwise circular shift, where the $M$ most significant bits overwrite the $M$ least significant bits after shifting the remaining $512 - M$ bits

left by $M$ bit positions. Note that this shifting is different from the RTM nanowire shift operation. In this case, the HV bit positions along the nanowire do not change, rather the HV representing the character is shifted across all nanowires it spans, using the peripheral circuitry in Fig. 4a. For instance, for the first N-gram "dont" in the running example, the hypervector $\vec{d}$ of the first character 'd' is rotated by 3, the hypervector $\vec{o}$ is rotated by 2, the hypervector $\vec{n}$ is rotated by 1, and the hypervector $\vec{t}$ is taken unchanged. This is important for differentiating this permutation of these four characters from any other permutation.

To efficiently rotate a hypervector, which spans many DBCs, the *rotate* control signal is enabled and a read operation is performed on all subarrays in a PG. The resultant hypervector in the row-buffer is the rotated-by-one version of the original hypervector. A subsequent write command is issued to the RTM controller to update the new value in RTM. To perform a rotation by three, our RTM architecture will perform three rotated-by-one operations sequentially.

Note that rotating an entire 8192 bit HV in RTM requires considerable modifications to the RTM row buffer. The customization in Fig. 4a only allows rotating a 512-bit chunk of the HV, *i.e.*, rotation at the granularity of the subarray. HDCR performs chunk-wise permutation on all subarrays in a PG and concatenates the permuted chunks to generate the permuted HV, as demonstrated in Fig. 6b (Step 1.3) and Algorithm 1 (Lines 19-21). This chunk-wise rotation operation is reversible and the generated hypervector was empirically verified to not adversely impact the accuracy of the HDC framework.

Once the required $N$ hypervectors for a particular N-gram are loaded and $N-1$ (all but last) hypervectors are permuted, they are XORed together to generate the resultant N-gram hypervector ($\vec{\phi_i}$). As described in Section 2.3, a TR operation and sense amplifiers detect how many ones exist between the TR access ports. When exactly one, three, or five 1s are detected, the logic in Fig. 4b asserts the XOR output, representing the XOR of all TRD operands.

This binding operation is performed iteratively for all N-grams in the input text. As the input text is consumed, each character hypervector in each N-gram is used at least $N$ times in different permutations to generate $N$ N-gram vectors. For instance, the hypervector $\vec{t}$ is used as-is to generate the first N-gram vector in the running example. However, for the second N-gram ("ont$\tau$") vector, $\vec{t}$ is rotated by 1. Similarly, for third and fourth N-gram vectors, $\vec{t}$ is rotated by 2 and 3, respectively. Since the sequence of operations is known, we can reuse each permutation result in the next iteration to save execution cycles.

To accomplish this we leverage both upper and lower access points to align, read/shift into the row buffer, and then write back the rotated into the access points while minimizing alignment operations. The detailed approach is described in Algorithm 2 referencing DBC locations from Fig. 3 in the encoder $DBC_9$ shown in Fig. 6. Using the example, we first read $\vec{v_0}$ and rotate and then write it back to complete $p^1(\vec{n})$. We then align $\vec{v_1}$ with the lower access point to complete $p^2(\vec{o})$. We then align the outgoing $\vec{v_3}$ with the upper access point to reset it to zero. We then align $\vec{v_2}$ with the upper access point to complete $p^3(\vec{d})$ and then align the lower access point to write $\vec{t}$ from the IM.

As a result of the binding and permute operation, a new N-gram vector is generated and is consumed by the bundling unit, as explained in the next section. For the entire input text, a whole set of N-gram vectors is generated where each vector corresponds to an N-gram in the text. Recall, $V_z$ represents all N-gram vectors of the input text (see Section 2)[2]. The bundling operation combines all elements in $V_z$ by taking the bit-wise majority on each bit position, as explained in Section 2. In the next section, we discuss the implementation of bundling in HDCR.

[2] $V_z$ is distinct from $\mathsf{V}_{0..31}$, which represents logical locations in the DBC (see Fig. 3).

**Algorithm 2** Memory operations required for computing an N-gram HV

1: ▷$\vec{v}_i$, $i \in \{0,1,2,3,4\}$ represents HV stored in DBC locations 0,1,2,3,4, i.e., all five locations between APs (see Step 1.3 in Fig. 6b)
2: ▷At any time Shift (if necessary) to align $\vec{c}_i$ to AP in IM

3: Algorithm Step: $\vec{v}_1 \leftarrow \rho(\vec{v}_0)$ (see Line 40 in Algorithm 1)
4: Memory operations:
 (i) Read $\vec{v}_0$ (with rotate signal enabled)
 (ii) Write the row buffer contents to lower access point (old $V_0$, new $V_1$)
5: Algorithm Step: $\vec{v}_2 \leftarrow \rho(\vec{v}_1)$ (see Line 39 in Algorithm 1)
6: Memory operations:
 (i) Shift down one position to align $\vec{v}_1$ to lower AP
 (ii) Read $\vec{v}_1$ (with rotate signal enabled)
 (iii) Write the row buffer contents to lower access point (old $V_1$, new $V_2$)
7: Clear old $\vec{v}_3$:
8: Memory Operations
 (i) Shift up by three positions to align $\vec{v}_3$ to upper AP while resetting row buffer
 (ii) Write the row buffer contents to upper access point (old $V_3$, new $V_4$)
9: Algorithm Step: $\vec{v}_3 \leftarrow \rho(\vec{v}_2)$ (see Line 38 in Algorithm 1)
10: Memory operations:
 (i) Shift up by one position to align $\vec{v}_2$ to AP
 (ii) Read $\vec{v}_2$ (with rotate signal enabled)
 (iii) Shift to align DBC location three to AP
 (iv) Write the row buffer contents to the DBC upper access point (old $V_2$, new $V_3$)
11: Algorithm Step: $\vec{v}_0 \leftarrow \vec{c}_i$ (see Line 41 in Algorithm 1)
12: Memory operations:
 (i) Shift down by one position to align DBC new $V_0$ to lower AP and Read $\vec{c}_i$
 (ii) Write the row buffer contents to the DBC $V_0$

*4.3.2 Bundling Operation in RTM.* Bundling in the HDC framework is a conjunctive operation that forms a representative vector for the set of N-gram hypervectors $V_z$ (see Section 2.1). Concretely, it computes a new hypervector $\vec{\Gamma}$ by adding all hypervectors in $V_z$, *i.e.*, $\vec{\Gamma} = \sum_{\vec{\Phi} \in V_z} \vec{\Phi}$. Each component in $\vec{\Gamma}$ is then compared to a fixed threshold to make it binary , *i.e.*, $\forall i \in \{1, 2, \ldots 8192\}$, $\vec{\mathcal{T}_i} = \beta_i$, and

$$\beta_i = \begin{cases} 1, & \text{if } \vec{\Gamma}_i > \text{threshold} \\ 0, & \text{otherwise} \end{cases}$$

(see Algorithm 1, Lines 52-56). The threshold value for binary hypervectors is typically the greatest integer less than 0.5 times the number of elements in $V_z$. For instance, for $|V_z| = 55$, the threshold value will be $\lfloor 55 \times 0.5 \rfloor = 27$, which also means that the resultant hypervector $\vec{\mathcal{T}}$ is equivalent to the output of the *majority* function, *i.e.*, $\vec{\mathcal{T}} = \text{Majority}(\ \vec{\Phi},\ \forall \vec{\Phi} \in V_z)$.

HDCR uses RTM counters (see Section 3.2) for each bit position to implement the majority function for $|V_z| >$ TRD. As shown in Fig. 6b (step 1.2-1.4), each subarray dedicates $\text{DBCs}_{10-15}$ for RTM counters. At each bit position in a PG, the 6 nanowires in $\text{DBCs}_{10-15}$ are used to implement the counter for that particular position. With 6 nanowires, the RTM counters can count from 0 to $10^6 - 1$, far more than what is required for the LR use case. For each new N-gram hypervector, HDCR updates all counters simultaneously based on the XOR result. Once a particular counter hits the

threshold, it ignores subsequent incrementing. To simplify the thresholding, the memory controller can preset the state of the counter to $M - T$ where $M$ is the maximum value represented by the counter and $T$ is the desired threshold. Thus, the thresholding does not require any additional logic and can be represented by the status of the $P$ bit of the most-significant digit of the counter.

In our evaluated system, we have 128 PGs (see Section 5.1). To reduce the overall runtime, the input text is divided into 128 chunks, and each chunk is provided to a separate PG. Once the computation in all PGs is finished, the majority output of all PGs is combined to make a single final vector. In the training phase of the HDC framework, this final computed hypervector represents the language (class) hypervector and is written to the AM (same DBCs as for item memory, i.e., DBCs$_{0-9}$ but different positions). In inference, this hypervector is referred to as the query hypervector and is compared to all class hypervectors to infer the final result, as shown in Fig. 6b (step 2) and explained in the next section.

### 4.4 Inference

The inference phase of the HDC framework uses the same encoding module to generate a query hypervector for the input text. Since the language class hypervectors are pre-generated in the training phase and are stored in the cim-tiles, classification is conducted by computing the Hamming distance of the query vector with all class vectors to find the closest match (see Section 2.1).

This similarity search is encapsulated in a module which performs three main operations. First, the query hypervector is XORed with all class hypervectors for bit-wise matching. Subsequently, the Hamming weight is computed by performing a population count of set bits within each of the computed hypervectors. Finally, the language with the minimum Hamming weight is inferred as the output.

From the implementation perspective, HDCR uses one subarray per language hypervector. For the 22 language hypervectors, HDCR uses 22 subarrays (2 PGs). As shown in Fig. 7, the language vectors in subarrays are stored across different DBCs of the same subarray, unlike the encoding module which stores hypervectors across different subarrays. The query vector is then written to all 22 subarrays to compute the Hamming weights independently.

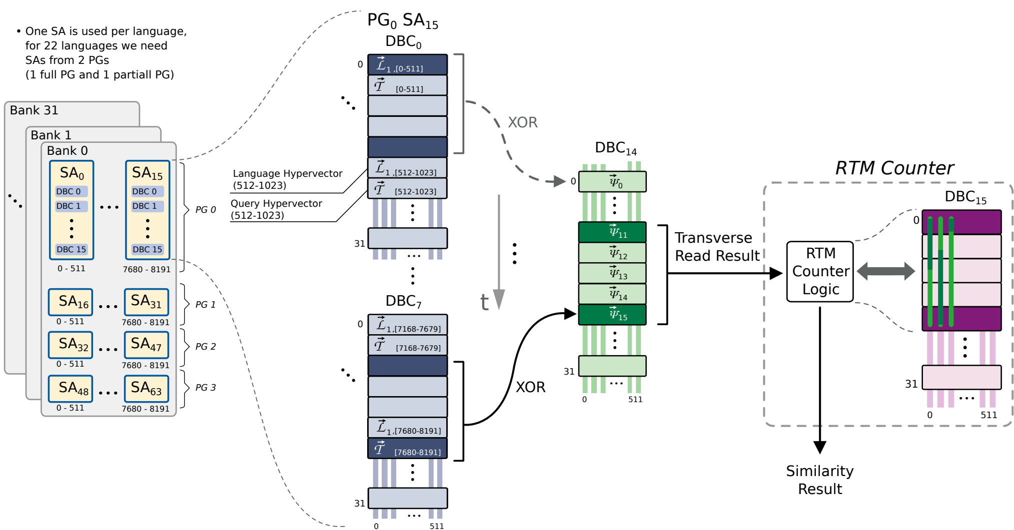



Fig. 7. Similarity search module

The XOR operation generates 16×512 bits for each language. In each subarray (for each language), the 16 chunks are processed sequentially, with each iteration producing one 512-bit chunk of the XOR operation in a single cycle, and then storing the results adjacent to one another in the same DBC ($DBC_{14}$ in Fig. 7) for the subsequent population count operation. For each of these 16 parallel 512 bit results, the TR operation sequentially performs the '1' counting in $DBC_{14}$. HDCR uses the TR result to shift bits in the RTM counter implemented in $DBC_{15}$, as shown in Fig. 7. Since the maximum count value in the similarity search module can be 8192, HDCR uses four nanowires for the RTM counter in this module. Note that, unlike the per-bit counting for the majority operation in the encoding module, the similarity search module uses a single RTM counter per DBC to find a single Hamming weight value per language. This necessitates the counters to be updated sequentially for all 512 TR outputs after each TR operation.

Once the counting operations of the inference is done, the TR and $P$ values for all counters packed like in Fig. 8 and sent sequentially to the memory controller for final input language selection.

| Index | 0..4 | 5 | 6 .. 10 | 11 | 12 .. 16 | 17 | 18 .. 22 | 23 | 24 .. 511 |
|---|---|---|---|---|---|---|---|---|---|
| Value | $TR_{00}$ .. $TR_{04}$ | $P_0$ | $TR_{10}$ .. $TR_{14}$ | $P_1$ | $TR_{20}$ .. $TR_{24}$ | $P_2$ | $TR_{30}$ .. $TR_{34}$ | $P_3$ | ∅ .. ∅ |

Fig. 8. Example of packing TR and $P$ values from the counters into local subarray rowbuffer.

## 5 EVALUATION

This section explains our experimental setup, provides details on the dataset, and compares our proposed system to state-of-the-art solutions for performance and energy consumption. Concretely, we evaluate and compare the following designs.

- *HDCR:* Our proposed in-RTM HDC system.
- *FPGA:* The FPGA based HDC system from [49].
- *PCM:* The in-PCM HDC implementation from [21].
- *CPU:* For the sake of completeness, we also compare to a software/CPU control.

### 5.1 Experimental Setup

As a target system, we consider an RTM-based 8GB main memory that consists of 32 banks, having 64 subarrays each. A subarray consists of 16 tiles composed of 16 DBCs, which are 512 bits wide and have 32 columns/data domains per racetrack. We assume two access ports per nanowire and an operating clock frequency of 1000 MHz. The cim-tiles utilize a high throughput mode proposed in prior PIM work [7]. The peripheral circuitry in cim-tiles does not affect the storage capability or otherwise prevent its use to store data beyond the marginal delay of a single multiplexer. The majority of the latency overhead results from the reducing the number of domains between the ports, from 16 to 5, which increases the average shift distance in the cim-tiles. While the target technologies may be subject to different types of faults, the experiments here presume fault free operation to ensure a fair comparison, particularly with respect to PCM which has limited endurance. However, HDCR is is compatible with previous reliability schemes proposed in the literature as DECC [38], or Hi-Fi [67] and by employing these techniques the major fault mode of shift misalignment the intrinsic fault rate of circa $10^{-5}$ can be reduced to circa $10^{-20}$ with negligible performance penalty [67]. For the LR use case, the entire training and test data sets fit in RTM. However, since the proposed solution is generic and use case independent, the data sets can also be partially loaded into RTM as needed to accommodate larger inputs with the same size working set. The energy and latency numbers of the memory subsystem are estimated using the CIM architecture presented in [39], the parameters from [66] and are shown in Table 2.

Table 2. RTM latency and energy parameters

| | |
|---|---|
| Domains per track | 32 |
| Tracks per DBC | 512 |
| Background power [mW] | 212 |
| rRead energy [pJ]/bit | 0.5 |
| Shift energy [pJ]/bit | 0.3 |
| Shift latency [Cycle] | 1 |
| Read latency [Cycle] | 1 |
| Write latency [Cycle] | 1 |

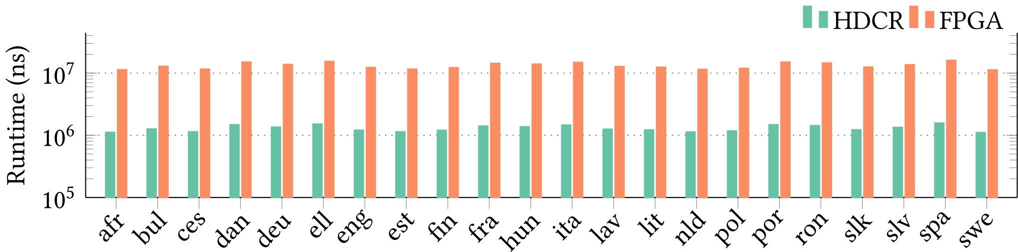



Fig. 9. Runtime of HDC training on different platforms.

**Baseline Systems:** For the FPGA design, we use the System Verilog implementation from [49]. We synthesize the design on a Xilinx Virtex 7 FPGA (7vx1140tflg1930) using Vivado 19.2. The maximum clock frequency was 80 MHz and the device utilization is 61% and 23%, for LUTs and flip flops, respectively. We get the throughput result from the post place & route simulation, which was also used to record the switching characteristics of the design. The switching activity file is fed to the Vivado power estimator to get the overall energy consumption.

For the CPU results, we use an Intel® Core(TM) i7-5650U CPU @ 2.20 GHz, with 8 GB RAM. We use the C libraries for the LR use case from [11]. For comparison with the PCM configuration, we used the numbers reported in [21].

## 5.2 Data Set

The language training data is taken from a corpora [46], which contains sample texts in 22 languages. For inference, an independent data set from [29] is used, which comprises 1000 sentences per language. The training, respectively the derivation of the language hypervectors, was carried out with the entire training data set, which contains a text of 120000-240000 words per language. The classification and thus the evaluation of the accuracy is carried out on multiple instances of one sentence per language. Concretely, 1000 tests with one sentence per test are performed for each language. We implement both the training and the inference phases of the HDC framework and report the results in the following sections.

## 5.3 Performance Comparison

The runtime comparison for training and inference in HDCR and FPGA designs is presented in Fig. 9 and Fig. 10, respectively. The runtime, and also the energy consumption in the next section, for the training and inference phases are computed and reported separately because training is

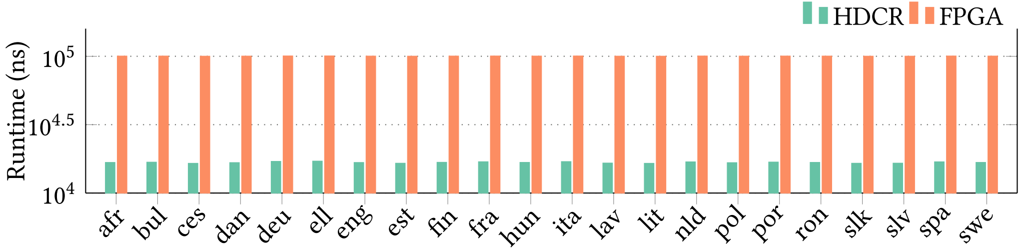



Fig. 10. Runtime of the HDC inference on different platforms. The results are generated on average length input text for all languages.

typically performed once and in advance. In contrast, the inference is performed more frequently in real-world applications. Therefore, the measured values for the inference should be regarded as having a higher relevance. Since the runtime depends on the number of letters in the input text, which varies for different languages, the evaluation is performed for each language.

On average (geomean), HDCR is an order of magnitude faster compared to the FPGA design. Note that the FPGA implementation we used for comparison is already optimized for a high degree of concurrency and parallelism. All hypervectors are stored in registers, and encoding an N-gram requires only a single clock cycle, *i.e.*, all $N$ HVs are simultaneously permuted, and the `XOR` operation is performed directly in the same combinational path. This results in long combinational paths, which leads to a lower clock frequency of 80 MHz. The massively parallel implementation of bit operations on the vectors also results in an enormous consumption of resources, limiting the given FPGA design to large devices, e.g., from the Virtex 7 series. Unlike the encoding operation, the similarity check module compares HVs sequentially and requires 8192 cycles to compare the query HV to a single class HV. This module is replicated 22 times to compare to all languages simultaneously.

In HDC training, only the encoding module is used to encode large training texts[3] into their respective class vectors. Despite the sequential rotation of hypervectors in HDCR, it outperforms the FPGA design by a geometric mean of $\approx 10.2\times$ (see Fig. 9). This is mainly attributed to the smaller clock period in HDCR 1 ns compared to 12.5 ns in the FPGA design.

In HDC inference, due to the smaller input text[4], the overall runtime of the FPGA design is largely dominated by the similarity checking module. We use an average sentence size per language generated from all 1000 test sentences per language in the test data set for this evaluation. Again, despite the sequentiality in population counting, HDCR on average (geomean) reduces the runtime by $\approx 6\times$ compared to the FPGA design (see Fig. 10). This is because the FPGA design performs the vector comparison sequentially while HDCR compares in 512-bit chunks, in parallel across languages.

We also synthesized the hardware to an ASIC 65 nm process using Cadence RTL Compiler to generate a performance comparison point consistent with the ASIC energy comparison point presented in the next section. The best achievable clock speed was 400 MHz which is approximately $5\times$ faster than the FPGA implementation. The silicon required an area of 4.37 mm$^2$, which is quite substantial for a single function accelerator. Given HDCR is more than $6\times$ faster for all operation modes and an ASIC implementation would be limited to the single task, HDCR provides a substantial benefit over custom silicon. For completeness of comparison, on the CPU machine, the training

[3]The number of characters for the training texts was between 100000 and 200000.
[4]The number of characters for the test sentences was between 100 and 200 for all languages.

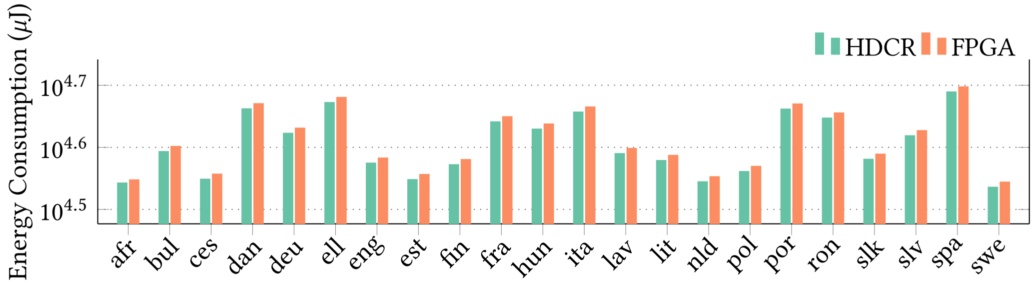



Fig. 11. Energy consumption of the HDC training.

and inference modules require $0.107 \times 10^3$ sec and $0.095 \times 10^3$ sec for all 22 languages, which are four and seven orders of magnitude slower, respectively, compared to HDCR. The reason for this significant performance gap is that CPU machine requires a huge amount of data shuffling between the memory and the core [9]. Additionally, the HDCR performance gain is partially attributed to its custom in-memory compute units, *e.g.,* the `XOR` implementation. Since the total energy consumption is dependent on the runtime, a similar trend is expected in the energy comparison of HDCR and the CPU machine.

### 5.4 Energy Consumption

In terms of energy consumption, HDCR is comparable to the FPGA design during the HDC training phase (see Fig. 11) and ≈ 5.3× better during the inference phase. In the similarity checking module alone, HDCR reduces the energy consumption by ≈ 95× (see Fig. 12). However, this is masked by the roughly equivalent energy consumption of the encoder module in both designs. The dominant impact on the energy consumption for the HDCR encoding phase is attributed to the parallel implementation of the majority operation with RTM counters. This requires 8192 counters which enable the required number of parallel bit-write operations. Since the energy consumption for RTM is proportional to such write operations, it is correspondingly large for the encoding step. The result presented in Fig. 11 shows the energy consumption during the training phase, which includes the encoder. While the results vary less than 1% different between FPGA and HDCR, this analysis does not consider I/O energy associated with moving data to and from the accelerator. In both cases, the input letters need to be transferred from the main memory to the computing unit. While HDCR only needs the input letter to be read and sent to the RTM memory controller, the FPGA system must also forward the data on the bus to the FPGA implementation. This omission makes our results more conservative, but independent of how the external system interfaces the implementation. Regardless, the reduced inference-time energy allows the HDCR implementation to immediately realize a net energy benefit over the FPGA implementation as presented in Fig. 12.

In the case of inference, the similarity checking in HDRC requires a single counter per language, and the operation is performed only once. As soon as the bitwise comparison with the XOR operation is performed, the 1s in the resultant vector are aggregated using the TR operation and the RTM counter while the FPGA synthesizes a direct 1s counting circuit.

To summarize, with regard to the overall energy efficiency, the HDCR implementation reduces the energy consumption by 5.3× (geomean).

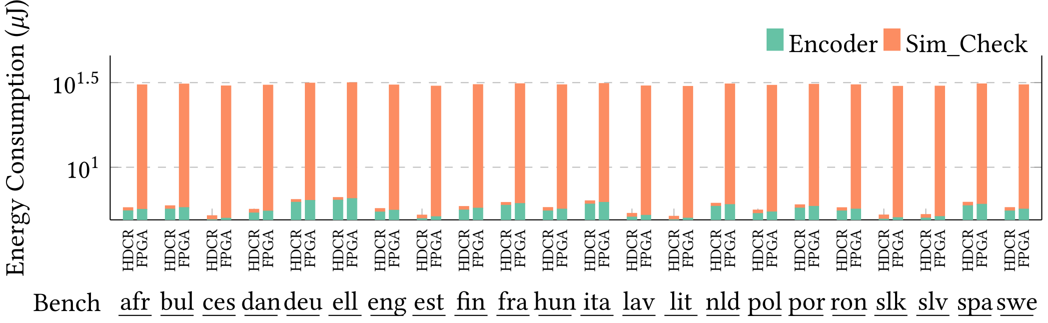



Fig. 12. Energy consumption of different modules in the HDC inference.

### 5.5 Comparison between HDCR and PCM

In the paper from Karunaratne et al. [21], they propose to use the novel PCM memory to implement HDC. This work does not report the latency of their implementation, thus here we only show the energy comparison. Table 3 compares the inference energy consumption of the HDCR and PCM designs for an average-sized input text. Overall, HDCR outperforms the PCM design by 10.1× in the encoding module and 1.08× in the similarity search module. Although the PCM design reports dramatic reduction in the energy consumption in the similarity checking module, largely due to parallel multiplications and current accumulation in the crossbar architecture, its overall energy consumption is still higher than HDCR. This is due to the higher write energy of the memristive devices compared to RTM. Comparing with the 65 nm CMOS-only design of the same reference, HDCR achieves a 51.6× improvement.

Table 3. Average energy per query

| | Encoder | Sim_Check | Total |
|---|---|---|---|
| all-CMOS [nJ] | 1474 | 1110 | 2584 |
| PCM [nJ] | 420.8 | 9.44 | 430.3 |
| HDCR [nJ] | 41.4 | 8.67 | 50.07 |
| Improvement (PCM / HDCR) | 10.1× | 1.08× | 8.59× |

## 6 RELATED WORK

Hyperdimensional computing has been used for learning and classification problems in many application domains. Among others, HDC has been used for analogy-based reasoning [20], language classification [48], hand gesture and activity recognition [37], text classification [14], and medical applications such as epileptic seizure detection [4]. Although compared to conventional learning algorithms, HDC is considered lightweight, the dimensionality of the hypervectors still makes HDC resource-intensive, particularly on embedded and IoT devices. To improve the performance and energy consumption of the HDC frameworks, they have been accelerated on various platforms. These include: FPGAs [52], conventional CPUs and GPUs [6], and domain-specific accelerators [15, 26, 36]. Since HDC is a memory-intensive application and is based on simple mathematical and logical

operations, the in-memory compute capabilities of emerging nonvolatile memory technologies can be exploited to accelerate it.

Many recently proposed architectures conduct near- or in-memory computation using emerging nonvolatile memory technologies [30], typically tuned to leverage the strength of the particular memory technology and the intended application. These works can be broadly categorized based on the underlying technology (phase-change memory (PCM), ReRAM, STT-MRAM), and further by how they conduct their processing (bitwise operations, arithmetic logic, vector multiplication).

Vector multiplication and arithmetic is a fundamental operation to many machine learning and neural network tasks. In HDC, the same is applied in the encoding and similarity search modules to compute the n-gram hypervector and similarity score. Karunaratne et al. [21] implement dot-product operations using PCM in a crossbar architecture. Using an on-chip network and DAC/ADC circuits, smaller multiply-accumulate subarrays are composed to realize larger dot-product results. Other recent work conducts 8-bit multiply accumulate logic for convolutional neural networks by converting the values from digital to analog and uses analog crossbar computation to obtain the results [22]. These, along with similar works leveraging ReRAM [31, 34, 63] provide acceleration and improved energy consumption relative to GPU/CPU implementations, but offer limited flexibility for input size, limited accuracy associated with computation in the analog domain, and require additional area to interpret and accumulate the analog results. This makes such approaches unscalable for our target application.

Besides PCM and ReRAM, STT-MRAM technology can also be used for in-memory computation. For instance, HieIM [44] and MLC-STT-CIM [40] exploit customized STT-MRAM memories to conduct bitwise operations on memory contents and build arithmetic operations by combining bitwise operations. These designs offer energy and area benefits for simple large matrix operations such as convolution. Still, they are less efficient than other general PIM proposals and require customized cell designs, which are difficult to fabricate. A more efficient design in STT-CIM [16] conducts computation by opening multiple rows and sensing the combined current on shared bitlines. Using modified reference voltages at the sense amplifiers allows `OR`, `AND`, and `XOR` operations, which are then composed to realize arithmetic operations. This is more efficient than prior designs since the additional circuitry is restricted to the sense amplifiers, and more realistic to fabricate since it does not modify the fundamental cell structures. Unfortunately, STT-MRAM designs require an access point and a fixed reference layer for every cell. While some of this area's cost is mitigated by the use of crossbar architecture, the density is limited to the feature size of the access network. A similar density limitation exists for computation using other non-volatile memories [32], often with the added complication of limited endurance in the underlying memory cells. In contrast, planar racetrack memories only need as many access points as the length of the DBC, and in turn can achieve superior densities.

RTM was initially proposed as a secondary storage device [42, 43]. However, due to its promising characteristics, particularly its best-case SRAM class latency and high energy efficiency, RTM has been considered for application at all levels — from register file and instruction memory to SSDs — in the memory stack. For instance, Mao and Wang *et al.* have proposed RTM-based GPU register files to combat the high leakage and scalability problems of conventional SRAM-based register files [35, 62]. Xu *et al.* evaluated RTM at lower cache levels and reported an energy reduction of 69% with comparable performance relative to an iso-capacity SRAM [64] and explored the impact of lightweight compression to allow independent shifting [65]. Venkatesan *et al.* demonstrated RTM at last-level cache and reported significant improvements in area (6.4×), energy (1.4×) and performance (25%) [59]. Park advocates the usage of RTM instead of SSD for graph storage which not only expedites graph processing but also reduces energy by up-to 90% [41]. Besides, RTMs have been proposed as scratchpad memories [24], content addressable memories [68], reconfigurable

memories [70], and even as network buffers [27, 28]. A recent review on RTMs covers more details on the latest developments in RTMs and provides an exhaustive list of references on the application of RTM in the memory subsystem [2].

There are relatively fewer instances of processing-in-memory applied to racetrack memories. The state-of-the-art offers three approaches: S-CNN [33], DW-NN [66], and PIRM [39]. SPIM adds a dedicated processing unit utilizing skyrmions that can compute logical `OR` and `AND` operations. Unfortunately, these operations require dedicated circuitry for a fixed number of operands, limiting the utility of the approach when more complex computation is required [39]. DW-NN uses dedicated racetrack pairs, which store data from either operand and compute logical functions by reading across the stacked magnetic domains. Simple XOR operations are computed directly, and in concert with an additional precharge sensing amplifier, can be used to compute a `SUM` and `CARRY` for addition. These results are then transferred to conventional racetracks, which can be shifted and summed to perform multiplication. Unfortunately, performance is bottlenecked in two places: first, the data must be read from the conventional racetracks to the paired racetracks one bit at a time. Second, each bit position in the paired nanowires must be shifted under the access port, serializing the computation. While the architecture offers an energy and throughput advantage compared to von Neumann, this serialization limits the utility of the approach. Finally, PIRM offers a more generalized computation framework, utilizing a more capable PIM-enabled tile to compute arbitrary logical operations, addition, and multiplication. PIRM accelerates computation by leveraging TR and multi-operands, and does not require specialized racetracks to do its work.

Our cim-tile uses the same philosophy as PIRM, but is tuned for the operations needed to compute HDC. Additionally, we explore new operations such as counters and majority determination. While prior work for HDC using in-memory PCM did not conduct a performance analysis, recent work for convolutional neural networks (CNNs) using a similar PIM approach did report 1.0 tera operations per second (TOPS) for 8-bit values. This is a significant improvement over FPGA capabilities which can achieve 0.34 TOPS [17]. However, compute-in-racetrack memory approaches can outperform this PCM result by an order of magnitude, for example with S-CNN achieving 9.3 TOPS. Given HDCR uses a similar mechanism to PIRM, and PIRM provides 26 TOPS for 8-bit CNN inference, we can expect similar order of magnitude speedups of HDCR over PCM for HDC applications.

## 7 CONCLUSIONS

The data dimensionality and mathematical properties of the HDC frameworks make them ideal fits for in-memory computations. Many conventional and emerging memory technologies allow (partial) implementation of the HDC framework in-memory. In this paper, we present a complete racetrack memory based HDC system, requiring near-negligible additional CMOS logic. Most of the HDC operations are implemented with the *TR* operation that reports the number of 1s in the nanowire, exploiting its properties and magnetic domain (and domain wall) arrangements. For the majority and the population count operations, we propose RTM nanowires-based counters that are scaleable and area and energy-efficient compared to their CMOS counterparts. The hypervectors are organized in RTM in a way that allows maximum possible parallelism and minimum possible data movement. For the in-RTM computations, we dedicate one tile per subarray – the cim-tile – and make minimal but necessary changes to its peripheral circuitry. For the logic operations, a few additional multiplexing/selection gates are added to the row buffer circuitry to infer the transverse results into different HDC operations. Our hardware customization and extensions are negligible compared to other memory technologies, *e.g.*, the power-hungry ADC/DAC converters, etc., in memristive devices. For the language recognition use case, our proposed system, on average, consumes 5.33× and 8.59× less energy compared to the state-of-the-art FPGA and PCM-crossbar designs, respectively.

## ACKNOWLEDGMENTS

This work was partially funded by the German Research Council (DFG) through the TraceSymm project (366764507) and the Co4RTM project (450944241), and by the NSF awards 1822085 and 2133267 and by the laboratory of physical sciences (LPS) and NSA.

## REFERENCES

[1] Samantha Archer, Georgios Mappouras, Robert Calderbank, and Daniel Sorin. 2020. Foosball coding: Correcting shift errors and bit flip errors in 3d racetrack memory. In *2020 50th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE, 331–342.

[2] Robin Bläsing, Asif Ali Khan, Panagiotis Ch. Filippou, Chirag Garg, Fazal Hameed, Jeronimo Castrillon, and Stuart S. P. Parkin. 2020. Magnetic Racetrack Memory: From Physics to the Cusp of Applications Within a Decade. *Proc. IEEE* 108, 8 (2020), 1303–1321. https://doi.org/10.1109/JPROC.2020.2975719

[3] Robin Bläsing, Tianping Ma, See-Hun Yang, Chirag Garg, Fasil Kidane Dejene, Alpha T N'Diaye, Gong Chen, Kai Liu, and Stuart SP Parkin. 2018. Exchange coupling torque in ferrimagnetic Co/Gd bilayer maximized near angular momentum compensation temperature. *Nature communications* 9, 1 (2018), 1–8.

[4] Alessio Burrello, Lukas Cavigelli, Kaspar Schindler, Luca Benini, and Abbas Rahimi. 2019. Laelaps: An Energy-Efficient Seizure Detection Algorithm from Long-term Human iEEG Recordings without False Alarms. In *2019 Design, Automation Test in Europe Conference Exhibition (DATE)*. 752–757. https://doi.org/10.23919/DATE.2019.8715186

[5] Sohum Datta, Ryan A. G. Antonio, Aldrin R. S. Ison, and Jan M. Rabaey. 2019. A Programmable Hyper-Dimensional Processor Architecture for Human-Centric IoT. *IEEE Journal on Emerging and Selected Topics in Circuits and Systems* 9, 3 (2019), 439–452. https://doi.org/10.1109/JETCAS.2019.2935464

[6] Sohum Datta, Ryan A. G. Antonio, Aldrin R. S. Ison, and Jan M. Rabaey. 2019. A Programmable Hyper-Dimensional Processor Architecture for Human-Centric IoT. *IEEE Journal on Emerging and Selected Topics in Circuits and Systems* 9, 3 (2019), 439–452. https://doi.org/10.1109/JETCAS.2019.2935464

[7] Quan Deng, Lei Jiang, Youtao Zhang, Minxuan Zhang, and Jun Yang. 2018. Dracc: a dram based accelerator for accurate cnn inference. In *Proceedings of the 55th Annual Design Automation Conference*. 1–6.

[8] Lulu Ge and Keshab K Parhi. 2020. Classification using hyperdimensional computing: A review. *IEEE Circuits and Systems Magazine* 20, 2 (2020), 30–47.

[9] Saransh Gupta, Justin Morris, Mohsen Imani, Ranganathan Ramkumar, Jeffrey Yu, Aniket Tiwari, Baris Aksanli, and Tajana Šimunić Rosing. 2020. Thrifty: Training with hyperdimensional computing across flash hierarchy. In *2020 IEEE/ACM International Conference On Computer Aided Design (ICCAD)*. IEEE, 1–9.

[10] Eman Hassan, Yasmin Halawani, Baker Mohammad, and Hani Saleh. 2021. Hyper-Dimensional Computing Challenges and Opportunities for AI Applications. *IEEE Access* (2021).

[11] Michael Hersche et al. [n.d.]. HDlib. https://github.com/skurella/hdlib. Accessed: 2022-02-22.

[12] Michael Hersche, José del R. Millán, Luca Benini, and Abbas Rahimi. 2018. Exploring Embedding Methods in Binary Hyperdimensional Computing: A Case Study for Motor-Imagery based Brain-Computer Interfaces. arXiv:1812.05705 [eess.SP]

[13] Daniele Ielmini and H-S Philip Wong. 2018. In-memory computing with resistive switching devices. *Nature Electronics* 1, 6 (2018), 333–343.

[14] Mohsen Imani, Deqian Kong, Abbas Rahimi, and Tajana Rosing. 2017. VoiceHD: Hyperdimensional Computing for Efficient Speech Recognition. In *2017 IEEE International Conference on Rebooting Computing (ICRC)*. 1–8. https://doi.org/10.1109/ICRC.2017.8123650

[15] Mohsen Imani, Zhuowen Zou, Samuel Bosch, Sanjay Anantha Rao, Sahand Salamat, Venkatesh Kumar, Yeseong Kim, and Tajana Rosing. 2021. Revisiting HyperDimensional Learning for FPGA and Low-Power Architectures. In *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. 221–234. https://doi.org/10.1109/HPCA51647.2021.00028

[16] Shubham Jain, Ashish Ranjan, Kaushik Roy, and Anand Raghunathan. 2018. Computing in Memory With Spin-Transfer Torque Magnetic RAM. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 26, 3 (2018), 470–483. https://doi.org/10.1109/TVLSI.2017.2776954

[17] Weiwen Jiang, Edwin H-M Sha, Xinyi Zhang, Lei Yang, Qingfeng Zhuge, Yiyu Shi, and Jingtong Hu. 2019. Achieving super-linear speedup across multi-fpga for real-time dnn inference. *ACM Transactions on Embedded Computing Systems (TECS)* 18, 5s (2019), 67.

[18] Pentti Kanerva. 1996. Binary spatter-coding of ordered K-tuples. In *Artificial Neural Networks — ICANN 96*. Springer Berlin Heidelberg, Berlin, Heidelberg, 869–873.

[19] Pentti Kanerva. 2009. Hyperdimensional Computing: An Introduction to Computing in Distributed Representation with High-Dimensional Random Vectors. *Cognitive Computation* 1 (06 2009), 139–159. https://doi.org/10.1007/s12559-

009-9009-8

[20] P. Kanerva. 2010. What We Mean When We Say "What's the Dollar of Mexico?": Prototypes and Mapping in Concept Space. In *AAAI Fall Symposium: Quantum Informatics for Cognitive, Social, and Semantic Processes.*

[21] Geethan Karunaratne, Manuel Gallo, Giovanni Cherubini, Luca Benini, Abbas Rahimi, and Abu Sebastian. 2020. In-memory hyperdimensional computing. *Nature Electronics* 3 (2020), 327–337. https://doi.org/10.1038/s41928-020-0410-3

[22] Riduan Khaddam-Aljameh, Milos Stanisavljevic, Jordi Fornt Mas, Geethan Karunaratne, Matthias Brändli, Feng Liu, Abhairaj Singh, Silvia M Müller, Urs Egger, Anastasios Petropoulos, et al. 2022. HERMES-Core–A 1.59-TOPS/mm$^2$ PCM on 14-nm CMOS In-Memory Compute Core Using 300-ps/LSB Linearized CCO-Based ADCs. *IEEE Journal of Solid-State Circuits* (2022).

[23] Asif Ali Khan, Fazal Hameed, Robin Bläsing, Stuart SP Parkin, and Jeronimo Castrillon. 2019. Shiftsreduce: Minimizing shifts in racetrack memory 4.0. *ACM Transactions on Architecture and Code Optimization (TACO)* 16, 4 (2019), 1–23.

[24] Asif Ali Khan, Norman A. Rink, Fazal Hameed, and Jeronimo Castrillon. 2019. Optimizing Tensor Contractions for Embedded Devices with Racetrack Memory Scratch-Pads. In *Proceedings of the 20th ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, and Tools for Embedded Systems* (Phoenix, AZ, USA) *(LCTES 2019).* Association for Computing Machinery, New York, NY, USA, 5–18. https://doi.org/10.1145/3316482.3326351

[25] Kab-Jin Kim, Se Kwon Kim, Yuushou Hirata, Se-Hyeok Oh, Takayuki Tono, Duck-Ho Kim, Takaya Okuno, Woo Seung Ham, Sanghoon Kim, Gyoungchoon Go, et al. 2017. Fast domain wall motion in the vicinity of the angular momentum compensation temperature of ferrimagnets. *Nature materials* 16, 12 (2017), 1187–1192.

[26] Yeseong Kim, Mohsen Imani, Niema Moshiri, and Tajana Rosing. 2020. Geniehd: Efficient dna pattern matching accelerator using hyperdimensional computing. In *2020 Design, Automation & Test in Europe Conference & Exhibition (DATE).* IEEE, 115–120.

[27] Donald Kline, Haifeng Xu, Rami Melhem, and Alex K. Jones. 2015. Domain-wall memory buffer for low-energy NoCs. In *2015 52nd ACM/EDAC/IEEE Design Automation Conference (DAC).* 1–6. https://doi.org/10.1145/2744769.2744826

[28] Donald Kline, Haifeng Xu, Rami Melhem, and Alex K Jones. 2018. Racetrack Queues for Extremely Low-Energy FIFOs. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 99 (2018), 1–14.

[29] Philipp Koehn. 2005. Europarl: A Parallel Corpus for Statistical Machine Translation.

[30] Bing Li, Bonan Yan, and Hai Li. 2019. An Overview of In-Memory Processing with Emerging Non-Volatile Memory for Data-Intensive Applications. In *Proceedings of the 2019 on Great Lakes Symposium on VLSI* (Tysons Corner, VA, USA) *(GLSVLSI '19).* Association for Computing Machinery, New York, NY, USA, 381–386. https://doi.org/10.1145/3299874.3319452

[31] Haitong Li, Tony F. Wu, Abbas Rahimi, Kai-Shin Li, Miles Rusch, Chang-Hsien Lin, Juo-Luen Hsu, Mohamed M. Sabry, S. Burc Eryilmaz, Joon Sohn, Wen-Cheng Chiu, Min-Cheng Chen, Tsung-Ta Wu, Jia-Min Shieh, Wen-Kuan Yeh, Jan M. Rabaey, Subhasish Mitra, and H.-S. Philip Wong. 2016. Hyperdimensional computing with 3D VRRAM in-memory kernels: Device-architecture co-design for energy-efficient, error-resilient language recognition. In *2016 IEEE International Electron Devices Meeting (IEDM).* 16.1.1–16.1.4. https://doi.org/10.1109/IEDM.2016.7838428

[32] Shuangchen Li, Cong Xu, Qiaosha Zou, Jishen Zhao, Yu Lu, and Yuan Xie. 2016. Pinatubo: A processing-in-memory architecture for bulk bitwise operations in emerging non-volatile memories. In *Proceedings of the 53rd Annual Design Automation Conference.* 1–6.

[33] Bicheng Liu, Shouzhen Gu, Mingsong Chen, Wang Kang, Jingtong Hu, Qingfeng Zhuge, and Edwin H-M Sha. 2017. 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, 383–390.

[34] Jialong Liu, Mingyuan Ma, Zhenhua Zhu, Yu Wang, and Huazhong Yang. 2019. HDC-IM: Hyperdimensional Computing In-Memory Architecture based on RRAM. In *2019 26th IEEE International Conference on Electronics, Circuits and Systems (ICECS).* 450–453. https://doi.org/10.1109/ICECS46596.2019.8964906

[35] Mengjie Mao, Wujie Wen, Yaojun Zhang, Yiran Chen, and Hai Li. 2017. An Energy-Efficient GPGPU Register File Architecture Using Racetrack Memory. *IEEE Trans. Comput.* 66, 9 (2017), 1478–1490.

[36] Fabio Montagna, Abbas Rahimi, Simone Benatti, Davide Rossi, and Luca Benini. 2018. PULP-HD: Accelerating Brain-Inspired High-Dimensional Computing on a Parallel Ultra-Low Power Platform. In *2018 55th ACM/ESDA/IEEE Design Automation Conference (DAC).* 1–6. https://doi.org/10.1109/DAC.2018.8465801

[37] Justin Morris, Mohsen Imani, Samuel Bosch, Anthony Thomas, Helen Shu, and Tajana Rosing. 2019. CompHD: Efficient Hyperdimensional Computing Using Model Compression. In *2019 IEEE/ACM International Symposium on Low Power Electronics and Design (ISLPED).* 1–6. https://doi.org/10.1109/ISLPED.2019.8824908

[38] Sébastien Ollivier, Donald Kline Jr., Roxy Kawsher, Rami Melhem, Sanjukta Banja, and Alex K. Jones. 2019. 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.

[39] Sébastien Ollivier, Stephen Longofono, Prayash Dutta, Jingtong Hu, Sanjukta Bhanja, and Alex K. Jones. 2021. PIRM: Processing In Racetrack Memories. *arXiV* (August 2021). arXiv:2108.00000

[40] Yu Pan, Peng Ouyang, Yinglin Zhao, Wang Kang, Shouyi Yin, Youguang Zhang, Weisheng Zhao, and Shaojun Wei. 2018. A Multilevel Cell STT-MRAM-Based Computing In-Memory Accelerator for Binary Convolutional Neural Network. *IEEE Transactions on Magnetics* 54, 11 (2018), 1–5. https://doi.org/10.1109/TMAG.2018.2848625

[41] E. Park, S. Yoo, S. Lee, and H. Li. 2014. Accelerating graph computation with racetrack memory and pointer-assisted graph representation. In *2014 Design, Automation Test in Europe Conference Exhibition (DATE)*. 1–4. https://doi.org/10.7873/DATE.2014.172

[42] S. Parkin, M. Hayashi, and L. Thomas. 2008. Magnetic Domain-Wall Racetrack Memory. 320 (05 2008), 190–194.

[43] Stuart Parkin and See-Hun Yang. 2015. Memory on the Racetrack. 10 (03 2015), 195–198.

[44] Farhana Parveen, Zhezhi He, Shaahin Angizi, and Deliang Fan. 2018. HieIM: Highly Flexible in-Memory Computing Using STT MRAM. In *Proceedings of the 23rd Asia and South Pacific Design Automation Conference* (Jeju, Republic of Korea) *(ASPDAC '18)*. IEEE Press, 361–366.

[45] Jorge Puebla, Junyeon Kim, Kouta Kondou, and Yoshichika Otani. 2020. Spintronic devices for energy-efficient data storage and energy harvesting. *Communications Materials* 1, 1 (2020), 1–9.

[46] Uwe Quasthoff, Matthias Richter, and Chris Biemann. 2006. Corpus Portal for Search in Monolingual Corpora. *Proceedings of LREC-06* (01 2006).

[47] Abbas Rahimi et al. [n.d.]. HDC Language Recognition. https://github.com/abbas-rahimi/HDC-Language-Recognition. Accessed: 2021-07-05.

[48] Abbas Rahimi, Sohum Datta, Denis Kleyko, Edward Paxon Frady, Bruno Olshausen, Pentti Kanerva, and Jan M. Rabaey. 2017. High-Dimensional Computing as a Nanoscalable Paradigm. *IEEE Transactions on Circuits and Systems I: Regular Papers* 64, 9 (2017), 2508–2521. https://doi.org/10.1109/TCSI.2017.2705051

[49] Abbas Rahimi, Pentti Kanerva, and Jan M. Rabaey. 2016. A Robust and Energy-Efficient Classifier Using Brain-Inspired Hyperdimensional Computing. In *Proceedings of the 2016 International Symposium on Low Power Electronics and Design* (San Francisco Airport, CA, USA) *(ISLPED '16)*. Association for Computing Machinery, New York, NY, USA, 64–69. https://doi.org/10.1145/2934583.2934624

[50] Fabrizio Riente, Giovanna Turvani, Marco Vacca, and Mariagrazia Graziano. 2021. Parallel Computation in the Racetrack Memory. *IEEE Transactions on Emerging Topics in Computing* (2021), 1–1. https://doi.org/10.1109/TETC.2021.3078061

[51] Kawsher Roxy, Sébastien Ollivier, Arifa Hoque, Stephen Longofono, Alex K Jones, and Sanjukta Bhanja. 2020. A Novel Transverse Read Technique for Domain-Wall "Racetrack" Memories. *IEEE Transactions on Nanotechnology* 19 (2020), 648–652.

[52] Sahand Salamat, Mohsen Imani, Behnam Khaleghi, and Tajana Rosing. 2019. F5-HD: Fast Flexible FPGA-Based Framework for Refreshing Hyperdimensional Computing *(FPGA '19)*. Association for Computing Machinery, New York, NY, USA, 53–62. https://doi.org/10.1145/3289602.3293913

[53] Sahand Salamat, Mohsen Imani, and Tajana Rosing. 2020. Accelerating hyperdimensional computing on fpgas by exploiting computational reuse. *IEEE Trans. Comput.* 69, 8 (2020), 1159–1171.

[54] Kenny Schlegel, Peer Neubert, and Peter Protzel. 2020. A comparison of vector symbolic architectures. *arXiv preprint arXiv:2001.11797* (2020).

[55] Ali Shafiee, Anirban Nag, Naveen Muralimanohar, Rajeev Balasubramonian, John Paul Strachan, Miao Hu, R. Stanley Williams, and Vivek Srikumar. 2016. ISAAC: A Convolutional Neural Network Accelerator with In-Situ Analog Arithmetic in Crossbars. In *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*. 14–26. https://doi.org/10.1109/ISCA.2016.12

[56] Emma Strubell, Ananya Ganesh, and Andrew McCallum. 2019. Energy and Policy Considerations for Deep Learning in NLP. arXiv:1906.02243 [cs.CL]

[57] Neil C. Thompson, Kristjan Greenewald, Keeheon Lee, and Gabriel F. Manso. 2020. The Computational Limits of Deep Learning. arXiv:2007.05558 [cs.LG]

[58] Tommi Vatanen, Jaakko J Väyrynen, and Sami Virpioja. 2010. Language Identification of Short Text Segments with N-gram Models.. In *LREC*. Citeseer.

[59] Rangharajan Venkatesan, Vivek Kozhikkottu, Charles Augustine, Arijit Raychowdhury, Kaushik Roy, and Anand Raghunathan. 2012. TapeCache: A High Density, Energy Efficient Cache Based on Domain Wall Memory *(ISLPED '12)*. ACM, New York, NY, USA, 185–190. https://doi.org/10.1145/2333660.2333707

[60] Rangharajan Venkatesan, Shankar Ganesh Ramasubramanian, Swagath Venkataramani, Kaushik Roy, and Anand Raghunathan. 2014. STAG: Spintronic-Tape Architecture for GPGPU Cache Hierarchies. In *Proceeding of the 41st Annual International Symposium on Computer Architecuture* (Minneapolis, Minnesota, USA) *(ISCA '14)*. IEEE Press, 253–264.

[61] Rangharajan Venkatesan, Mrigank Sharad, Kaushik Roy, and Anand Raghunathan. 2013. DWM-TAPESTRI-an energy efficient all-spin cache using domain wall shift based writes. In *Proc. of DATE*. 1825–1830.

[62] Shuo Wang, Yun Liang, Chao Zhang, Xiaolong Xie, Guangyu Sun, Yongpan Liu, Yu Wang, and Xiuhong Li. 2016. Performance-centric register file design for GPUs using racetrack memory. In *2016 21st Asia and South Pacific Design Automation Conference (ASP-DAC)*. 25–30. https://doi.org/10.1109/ASPDAC.2016.7427984

[63] Tony F. Wu, Haitong Li, Ping-Chen Huang, Abbas Rahimi, Gage Hills, Bryce Hodson, William Hwang, Jan M. Rabaey, H.-S. Philip Wong, Max M. Shulaker, and Subhasish Mitra. 2018. Hyperdimensional Computing Exploiting Carbon Nanotube FETs, Resistive RAM, and Their Monolithic 3D Integration. *IEEE Journal of Solid-State Circuits* 53, 11 (2018), 3183–3196. https://doi.org/10.1109/JSSC.2018.2870560

[64] H. Xu, Y. Alkabani, R. Melhem, and A. K. Jones. 2016. FusedCache: A Naturally Inclusive, Racetrack Memory, Dual-Level Private Cache. *IEEE Transactions on Multi-Scale Computing Systems* 2, 2 (April 2016), 69–82. https://doi.org/10.1109/TMSCS.2016.2536020

[65] Haifeng Xu, Yong Li, Rami Melhem, and Alex K. Jones. 2015. Multilane Racetrack caches: Improving efficiency through compression and independent shifting. In *The 20th Asia and South Pacific Design Automation Conference*. 417–422. https://doi.org/10.1109/ASPDAC.2015.7059042

[66] Hao Yu, Yuhao Wang, Shuai Chen, Wei Fei, Chuliang Weng, Junfeng Zhao, and Zhulin Wei. 2014. 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)*. 191–196. https://doi.org/10.1109/ASPDAC.2014.6742888

[67] Chao Zhang, Guangyu Sun, Xian Zhang, Weiqi Zhang, Weisheng Zhao, Tao Wang, Yun Liang, Yongpan Liu, Yu Wang, and Jiwu Shu. 2015. Hi-fi playback: Tolerating position errors in shift operations of racetrack memory. In *ACM SIGARCH Computer Architecture News*, Vol. 43-3. ACM, 694–706.

[68] Y. Zhang, W. Zhao, J. Klein, D. Ravelsona, and C. Chappert. 2012. Ultra-High Density Content Addressable Memory Based on Current Induced Domain Wall Motion in Magnetic Track. *IEEE Transactions on Magnetics* 48, 11 (Nov 2012), 3219–3222. https://doi.org/10.1109/TMAG.2012.2198876

[69] Yue Zhang, WS Zhao, Dafiné Ravelosona, J-O Klein, Joo-Von Kim, and Claude Chappert. 2012. Perpendicular-magnetic-anisotropy CoFeB racetrack memory. *Journal of Applied Physics* 111, 9 (2012), 093925.

[70] W. Zhao, N. Ben Romdhane, Y. Zhang, J. Klein, and D. Ravelosona. 2013. Racetrack memory based reconfigurable computing. In *2013 IEEE Faible Tension Faible Consommation*. 1–4. https://doi.org/10.1109/FTFC.2013.6577771

## ACRONYMS


**AP** *access port* . . . 6

**AM** *associative memory* . . . 14

**CIM** *compute in memory* . . . 7

**DW** *domain wall* . . . 6

**DBC** *domain wall block cluster* . . . 7

**DWM** *domain wall memory* . . . 3

**HDC** *hyperdimensional computing* . . . 2

**HDCR** *HyperDimensional Computing in Racetrack* . . . 3

**HV** *hypervector* . . . 4

**IM** *item memory* . . . 5

**LR** *language recognition* . . . 5

**MTJ** *magnetic tunnel junction* . . . 8

**PG** *processing group* . . . 14

**RTM** *racetrack memory* . . . 3

**TR** *transverse read* . . . 3

**TRD** *transverse read distance* . . . 8

**TW** *transverse write* . . . 11