DART-PIM: DNA read mApping acceleRaTor Using Processing-In-Memory

***Abstract*—Genome analysis has revolutionized fields such as personalized medicine and forensics. Modern sequencing machines generate vast amounts of fragmented strings of genome data called *reads*. The alignment of these reads into a complete DNA sequence of an organism (the *read mapping* process) requires extensive data transfer between processing units and memory, leading to execution bottlenecks. Prior studies have primarily focused on accelerating specific stages of the read-mapping task. Conversely, this paper introduces a holistic framework called DART-PIM that accelerates the entire read-mapping process. DART-PIM integrates emerging memory technologies that facilitate in-memory computation for an end-to-end acceleration of the entire read-mapping process, from indexing using a unique data organization schema to filtering and read alignment with an optimized Wagner–Fischer algorithm. A comprehensive performance evaluation with real genomic data shows that DART-PIM achieves a 5.7× and 335× improvement in throughput and a 67× and 5.3× energy efficiency enhancement compared to state-of-the-art GPU and PIM implementations, respectively.**

I. Introduction

Genome analysis serves as the cornerstone of numerous scientific and medical breakthroughs, playing a vital role in the age of personalized medicine [1]–[4] and advanced forensics [5], [6]. Modern genome sequencing machines [7]–[10] produce vast volumes of short random fragmented strings of bases (A, C, G, T in DNA), commonly referred to as *reads*, which are extracted from the longer original DNA sequence [11]. The process of reconstructing the original DNA sequence typically begins by approximating the location of each read in the overall genome and then using the reads at each location to deduce the most likely reference base. This procedure involves complex algorithms with high memory requirements that constitute a bottleneck to the entire genome analysis process.

Sequence alignment is a popular approach that localizes read fragments based on their similarity to a corresponding *reference genome* of the target species. This approach is feasible thanks to the high degree (above 99%) of resemblance between genomes of different specimens within the same species. This computational process, called *read mapping* [11], [12], involves offline *indexing* of the reference genome (utilizing short representative fragments known as minimizers [13]), followed by three consecutive online steps (illustrated in Figure 1): (1) *Seeding* of potential locations (*PLs*) in the reference genome based on read and reference minimizer similarity; (2) *pre-alignment filtering* to eliminate false PLs; and (3) *read alignment* that employs complex algorithms to determine the more probable PL according to a similarity metric. This latter stage is by far the most computationally intensive task, invoking string matching algorithms [14]–[16] for each and every read and PL within the complete genome.

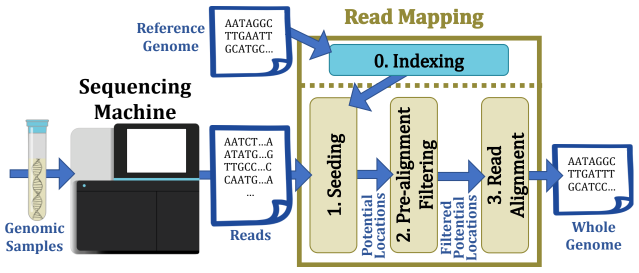
Reference Genome
AATAGGC TTGAATT GCATGC...
Read Mapping
0. Indexing
Sequencing Machine
AATCT...A ATATG...G TTGCC...C CAATG...A ...
Reads
Genomic Samples
1. Seeding
Potential Locations
2. Pre-alignment Filtering
Filtered Potential Locations
3. Read Alignment
AATAGGC TTGATTT GCATCC...
Whole Genome

Fig. 1. Genome sequence alignment process. The genomic samples are input into a **sequencing machine** that fragments them into small segments. The sequencing machine generates short strings known as **reads**, which are then processed during the **read-mapping** procedure. Read mapping involves an offline indexing stage followed by the online seeding, pre-alignment filtering, and read alignment stages.

State-of-the-art read alignment techniques predominantly employ a dynamic programming (*DP*)-based approach, such as the Needleman-Wunsch or Smith-Waterman algorithms [17]–[19]. The complexity of these algorithms follows quadratic scaling in execution time and memory usage, thereby generating a significant latency and energy bottleneck [11], [20]–[23]. This read-mapping bottleneck has been exacerbated by the recent enhancements of read generation rates on the one hand and insufficient corresponding growth in computational power on the other [11], [24].

To bridge the gap between read volumes and compute capabilities, prior work focused mainly on accelerating each read-mapping stage separately. This effort often prioritizes optimizing read alignment due to its significant computational demands, which can consume over 70% of the read-mapping execution time [22], [25]–[29]. Read mapping, however, involves not only the processing of extensive datasets, but also the frequent movement of huge volumes of data between the processor(s) and memory units, which adversely impacts both execution time and energy consumption [11], [30]. Since the data being transferred throughout the read-mapping process is roughly 100× larger than the original input reads, acceleration of any specific computational tasks across the process will still be bottlenecked by the data transfer between the processes.

Numerous studies have proposed alternative approaches to efficiently process large amounts of reads [22], [25]–[29]. For instance, the state-of-the-art read mapper GenASM [25] leverages near-memory computing [31] within a 3D-stacked memory architecture to mitigate costly chip-to-chip data transfers. Consequently, GenASM's read alignment process alone achieves a $111\times$ speedup over existing CPU-executed software read mappers, alongside a $33\times$ power consumption reduction. Nevertheless, when handling the entire read-mapping process, GenASM's performance demonstrates only a $1.9\times$ speedup for short reads.

A promising solution to overcoming the data transfer bottleneck involves conducting read-mapping computations directly within the memory. This approach utilizes emerging memristive memory technologies [32]–[34] to enable high parallelism, eliminating the need for frequent data transfer between the memory and the processor [35]. This paper presents *DART-PIM*, a DNA read mApping acceleRaTor using Processing-In-Memory, which is a comprehensive framework for accelerating the entire read-mapping process. The uniqueness of DART-PIM lies in its ability to execute all read-mapping stages for a given read and PL inside a single memristive crossbar array. The majority of computations are performed using the memory cells, thus circumventing the data transfer bottleneck. A small number of complementary operations are executed in adjacent RISC-V cores. A key element for efficient computation in DART-PIM is careful organization of the reads and the reference genome within memory instances to minimize data transfer along all steps of read mapping. Within these instances, the data allocation is optimized for maximal utilization of the inherent massive parallelism offered by in-memory processing. Consequently, DART-PIM considerably enhances the performance and energy efficiency of the entire end-to-end read-mapping process. This paper makes the following main contributions:

1) We propose a novel end-to-end accelerator architecture for the entire read-mapping process.
2) We present a data organization technique for accelerated indexing and seeding of a reference genome.
3) We develop a high-performance, memory efficient, in-memory pre-alignment filtering mechanism based on the linear Wagner-Fischer algorithm [36].
4) We improved read alignment in-memory performance by enhancing the Wagner–Fischer algorithm with an affine-gap penalty and traceback capability.
5) We build simulators of real genome datasets that ensure precise evaluation of DART-PIM's performance and compare it to state-of-the-art implementations (NVIDIA Parbricks [37] and GenASM [25]), demonstrating faster execution time ($5.7\times$ and $335\times$ improvement, respectively) and reduced energy consumption ($67\times$ and $5.3\times$, respectively).

II. Background: Read Mapping and Its Limitations

The read-mapping process comprises three sequential online stages, preceded by an offline indexing procedure. The classical indexing approach generates a database that matches any sub-string of length k (typically, 12) appearing in the reference genome (called *k-mer*) with pointers to all locations of this sub-string within the genome [38]. Although lossless, the practicality of this approach is limited due to its extensive memory demands. A more practical approach indexes the genome to select *minimizers* [13], rather than k-mers, which reduces the number of k-mers associated with each reference segment while achieving accurate compressed representation.

Once the indexing stage is completed, the database is stored and each new DNA sample can be sequenced by employing the following three steps: (1) seeding, (2) pre-alignment filtering, and (3) read alignment. These steps are performed sequentially for any new read (typically, strings comprising 100 to 2000 bases) entering the system. The purpose of the seeding stage is to identify PLs of the read within the reference genome. The read is processed in sliding windows of length $W + k - 1$, where W is the window size (typically $W = 30$), and each window is represented by its minimizer. The PLs of the read are defined as the set of addresses of all its (unique) minimizers in the reference genome (taken from the indexing database).

Since the sets of PLs are typically large, a filtering procedure is required to reduce the number of PLs per read. Pre-alignment filtering commonly employs heuristic techniques to discard PLs based on dissimilarity between the read and a reference genome segment. For instance, a popular filtering approach, called the base count filter [39], compares histograms of bases for a read and a corresponding segment (eliminating 68% of PLs on average).

The main step of read mapping involves optimal sequence alignment algorithms for read alignment. Since the goal is to determine the most accurate location of each read within the genome, it commonly involves computationally-intensive dynamic-programming algorithms such as the Smith-Waterman (*SW*) algorithm [40] or the Needleman-Wunsch (*NW*) algorithm [41]. While optimal sequence alignment is the obvious candidate for hardware acceleration due to its compute-bound nature, the primary portion of energy and execution time stems from data transfer between the memory and computing cores throughout the sequencing process.

Based on in-house simulations we conducted with a human reference genome (0.8GB in size), seeding generates an average of 1000 (32 bits) PLs per read, which amounts to 1556GB for a typical amount of 389M reads of length 150 (14.6GB in size). Note that while the input data volume to the seeding stage is 14.6GB, it generates approximately a 100 times larger output data volume, which has to be transferred to the memory and back to a pre-alignment filtering accelerator. This data transfer is the system's primary bottleneck, even with significant acceleration of each individual step. Motivated by this observation, this paper proposes a solution that executes all stages of read mapping within the same memristive crossbar array, thereby eliminating the need for data transfers between the different stages.

III. Modified Read-Mapping Algorithms Used in DART-PIM

This section describes the read-mapping algorithms that are employed in DART-PIM for efficient processing in-memory. The basic algorithms are the *Linear Wagner-Fischer* and *Affine Wagner-Fischer* algorithms. The former is used for pre-alignment filtering and the latter for read alignment. The preceding stages of read mapping, i.e., indexing and seeding, only require that the data be carefully arranged within the memory prior to pre-alignment filtering.

Existing state-of-the-art read mappers commonly employ either the Smith-Waterman (*SW*) algorithm [40] or the Needleman-Wunsch (*NW*) algorithm [41]. Since the compared strings in the DNA sequencing setting are typically similar, these algorithms, which consider sub-string matches, require relatively large bit-width representation (e.g., 8 bits) for storing the similarity score. Hence, supporting read mapping by processing it within the memory may impose a relatively large memory footprint or restrict the sequence length that can be processed efficiently.

To enhance read mapping efficiency, we propose an alternative algorithmic approach that adopts the Wagner-Fischer (WF) algorithm [36], originally used for edit distance computation. We modify the algorithm to output not only the distance score, but also the traceback (i.e., the aligned sequence itself). Since the WF algorithm counts mismatches, rather than matches, it requires fewer bits (e.g., 3) per score, thereby being a better fit for in-memory processing.

We use two variants of the WF algorithm. The first variant, used for pre-alignment filtering, is an adaptation of the classical WF algorithm. We call this the *Linear* WF algorithm. The second variant, used for read alignment, extends the linear WF by enhancing the WF algorithm by (1) accounting for the affine-gap penalty of the SW and NW algorithms, and (2) enabling traceback to reconstruct the aligned sequence. We refer to this as the *Affine* WF algorithm. The affine-gap penalty combines constant and linear gap penalties and is commonly used in biological applications [42], [43].

A. Linear Wagner-Fischer Algorithm

Given two input strings S_1 and S_2 (with respective lengths n and m), the linear WF algorithm constructs the *WF distance matrix*, D, according to the conditions of the WF algorithm, with a size of $(n+1)\times(m+1)$):

1) Initialize the first row and first column as follows:

$$D_{i,0}=\sum_{k=1}^{i} w_{del},\ \text{ for } 1\leqslant i\leqslant n,$$
$$D_{0,j}=\sum_{k=1}^{j} w_{ins},\ \text{ for } 1\leqslant j\leqslant m. \tag{1}$$

2) Fill the matrix D: If $S_1(i)=S_2(j)$, then $D_{i,j}=D_{i-1,j-1}$; otherwise,

$$D_{i,j}=\min\left\{\begin{matrix} D_{i-1,j}+w_{del}, \\ D_{i,j-1}+w_{ins}, \\ D_{i-1,j-1}+w_{sub}\end{matrix}\right\}. \tag{2}$$

where w_{del}, w_{ins}, and w_{sub} are the costs of a deletion, an insertion, and a substitution, respectively. The cost values, as well as additional WF parameters, appear in Table III.

B. Modified Affine Wagner-Fischer Algorithm

The affine WF algorithm enhances score accuracy (at the cost of complexity) by using a penalty that is represented as $w_{op}+w_{ex}(L-1)$, where w_{op} denotes the cost of opening a gap, w_{ex} represents the cost of extending an existing gap while L corresponds to the gap length. Since this score achieves higher accuracy at the cost of higher complexity (larger execution time and memory storage), we perform the linear WF algorithm for each PL for pre-alignment filtering, and employ the affine WF only for selected locations based on their linear WF score. The explicit affine-gap WF matrix is given by

$$D_{i,j}=\begin{cases} D_{i-1,j-1}, & S_1(i)=S_2(j) \\ \min\left\{\begin{matrix} M1_{i,j}, & \text{(ins)} \\ M2_{i,j}, & \text{(del)} \\ D_{i-1,j-1}+w_{sub} & \text{(sub)}\end{matrix}\right\}, & \text{otherwise,}\end{cases} \tag{3}$$

where

$$M1_{i,j}=\min\left\{\begin{matrix} M1_{i-1,j}+w_{ex}, & \text{(extend S1 gap)} \\ D_{i-1,j}+w_{op}+w_{ex} & \text{(open S1 gap)}\end{matrix}\right\} \tag{4}$$

and

$$M2_{i,j}=\min\left\{\begin{matrix} M2_{i,j-1}+w_{ex}, & \text{(extend S2 gap)} \\ D_{i,j-1}+w_{op}+w_{ex} & \text{(open S2 gap)}\end{matrix}\right\} \tag{5}$$

Due to the iterative nature of the WF algorithms, the (i,j)-th matrix value $D_{i,j}$ originates from one of three possible preceding cells (see Eq. 3), the (i,j)-th matrix values $M1_{i,j}$, $M2_{i,j}$ originate from one of two possible preceding cells (see Eq. 4 and 5). The affine WF saves the "directions" of the predecessors (4-bit representation) for each matrix cell. The optimal sequence alignment can, therefore, be inferred without having to save the entire matrix for traceback calculation.

IV. In-Memory Execution of Read-Mapping Algorithms

The execution of read mapping in DART-PIM utilizes in-memory processing. We chose to employ bulk bitwise processing-in-memory (PIM) [44], [45] for this purpose. This section outlines the basic principles of this approach using the MAGIC NOR gate [46]. Then, it describes how a WF iteration is implemented within a single row of the crossbar memory array.

A. Memristive In-Memory Processing

Memristive memories [32], [47], [48] can be used as non-volatile memories. Each memristive memory cell within the memory crossbar array consists of a single memristive device whose resistance encodes its logical state. When writing or reading data to/from a crossbar array, appropriate voltages are applied across the desired wordlines and bitlines to modify/sense the resistance of the target memristive devices [47]. Memristive memories are capable of performing computations

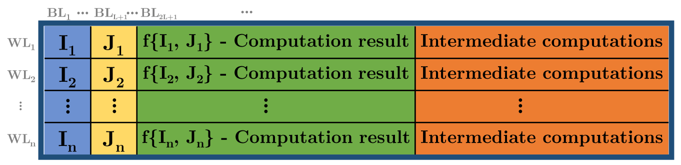
BL_1 ⋯ BL_{L+1} ⋯ BL_{2L+1} ⋯
WL_1
WL_2
WL_n
I_1
J_1
f{I_1, J_1} - Computation result
Intermediate computations
I_2
J_2
f{I_2, J_2} - Computation result
Intermediate computations
I_n
J_n
f{I_n, J_n} - Computation result
Intermediate computations

Fig. 2. Area allocation within a memory crossbar array: each row conducts a series of logical operations, starting from L-bits inputs I_x (blue) and J_x (yellow), using intermediate results (orange), to obtain the final output stored in the memory (green). All n rows perform the same logical operations (over different inputs) concurrently. WL_x and BL_x are, respectively, wordline (row) and bitline (column) number x.

TABLE I
SUMMARY OF EXECUTION CYCLES FOR MAGIC-NOR-BASED OPERATIONS WITH N-BIT OPERANDS (UNLESS SPECIFIED OTHERWISE).

Logical Operation	# Cycles
AND	$3N$
XNOR	$4N$
XOR	$5N$
Copy	$1+N$
Addition of two N-bit in-memory numbers	$9N$
Addition of an N-bit and a single-bit in-memory numbers	$5N$
Addition of one in-memory number and a constant	$5N$
Subtraction of two in-memory numbers	$9N$
Mux between two in-memory numbers	$3N+1$
Minimum of two in-memory numbers	$12N+1$

using the memory cells themselves [46], [49]–[52]. Although the reliability of memristive logic is still challenging, specific strategies have been developed [53]–[55] that substantially improve their correct execution and lifetime even with existing technologies.

One notable technique for conducting logic operations within an RRAM (memristive) array is the MAGIC NOR method [46]. Table I lists the operations supported by MAGIC NOR for execution and their latency. Executing a MAGIC NOR gate involves applying voltages to two input memory cells and one output cell that are located in the same row. The output memory cell changes its state according to the NOR function applied on the logical states of the two inputs. The structure of MAGIC NOR allows parallel execution of multiple gates, each in a different row, as long as their columns are aligned. Tools such as [56], [57] can determine the latency-optimal sequence of such operations to accomplish a desired computation. This sequence is implemented using cells within the same row, operating sequentially. When all available cells become occupied by the outputs, the unused cells are initialized and reused for the remaining computations. An illustration of this procedure is depicted in Figure 2.

Implementing read mapping using in-memory processing requires a variety of logical operations. In Table I we list these operations and characterize their corresponding required numbers of clock cycles. The hardware implementation of all these operations was carefully optimized for MAGIC NOR in-memory compute. A more detailed discussion of these operations and their implementation is included in the paper's GitHub repository [58].

B. Wagner-Fischer Algorithm in a Single Crossbar Row

Maximizing the parallelism of bulk-bitwise PIM is crucial in DART-PIM. Full potential is achieved when an entire WF algorithm is executed within a single crossbar row. In this configuration, each row processes a separate instance of the WF algorithm, with different subsets of columns dedicated to specific algorithmic steps. This setup allows simultaneous operations on multiple reads/PLs across the crossbar's columns.

To facilitate the single-row requirement, we truncate the maximal allowed values of the WF matrix elements to eth, beyond which the strings are considered too different and the value is saturated. The value selection of eth trades off alignment accuracy for complexity reduction, as it allows the calculation in each row only $2eth+1$ of (unsaturated) cells located around the WF matrix diagonal, rather than the entire matrix [59] (that will be mostly saturated). Since we assign a 3-bit representation for the linear WF (and 5-bit for the affine WF), we set eth to be 6 (31 for affine WF). The expected accuracy with these values is even better than that of the commonly used $eth = 5$ for linear SW [9].

Algorithm 1 details the execution steps needed to calculate a single cell in the WF matrix using MAGIC NOR operations within a single memory row. The majority of row cells are allocated to store the read (S_1) and the reference segment (S_2), while the remaining cells are used for the computation itself. The actual WF distances are assigned a designated subset of cells (defined as *WF distances buffer*), and the remaining cells are utilized for intermediate computations. This allocation is exemplified in Figure 3 and the mapping of WF matrix cells is illustrated in Figure 4. As this is the core of the entire alignment stage, the process was optimized to perform the exact computation with a minimal number of clock cycles.

To fit all WF computations into a single row, we aim to allocate the minimal required amount of crossbar row bits for the input data (strings S_1 and S_2), as well as for the computations. Note that the computation of each WF matrix cell requires only its upper, upper-left, and left neighbors, rather than the entire matrix. The calculation of WF matrix cells within the crossbar row is performed sequentially, where only the previous cells required for the present computation are stored in the crossbar at any given time. These WF matrix cells are computed iteratively, until the final result in the lower-left matrix cell is reached. In our proposed architecture, the aligned string is not needed for the linear WF and, therefore, traceback is not required. The computation of the entire WF matrix using a buffer size of only $2eth+1$ cells is provided in Algorithm 2.

Similarly, the computation of the affine WF involves three matrices, each consisting of $2eth+1$ cells in every matrix row. To reconstruct the aligned string (traceback), we track the edits path of the affine WF by storing the direction of the origin cell that leads to each matrix cell. According to the definition of the WF matrix in Eq. (3), two bits are needed per cell due to the four optional origin cell directions. The other matrices used for the computation, M1 from Eq. (4) and M2

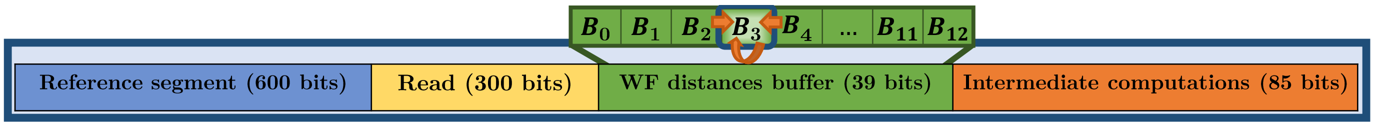
B_0 B_1 B_2 B_3 B_4 ... B_{11} B_{12}
Reference segment (600 bits)
Read (300 bits)
WF distances buffer (39 bits)
Intermediate computations (85 bits)

Fig. 3. Mapping of a linear WF matrix calculation into a single crossbar row for $eth = 6$. The reference segment (blue) and read (yellow) are the computation inputs. Only $2eth + 1$ WF distances are needed at any point (green). To compute the current WF distance, only the distances in adjacent cells (storing the top and left WF matrix distances) and the previous value of the current cell (storing the top-left WF matrix distance) are required. The intermediate results generated while computing the distances are stored in temporary row cells (orange), and due to limited number of cells, are re-used when necessary. The total number of bits in the row is 1024.

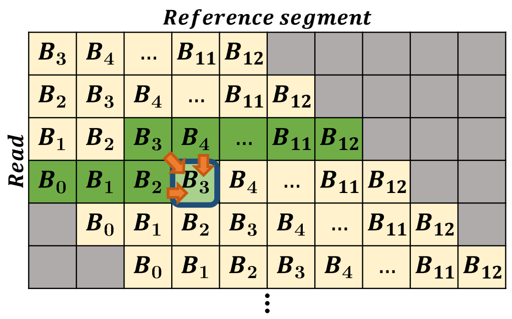

Fig. 4. Example of the mapping of a banded WF matrix (with $eth = 6$) onto the WF distances buffer in a single crossbar row (contained $2eth + 1$ cells). The computed cell is denoted by light green "B_3" (location $(4, 4)$). The remaining green cells are stored within the WF distances buffer during computation. The computation is performed using only top, top-left, and left cells (marked with orange arrows). Upon completion, the computed value replaces the target cell's value in the WF distances buffer. Note that gray cells are not computed as they represent distances larger than eth.

Algorithm 1 $D_{i,j}$ calculation for the linear WF with MAGIC (# MAGIC operations (ops) for a b bit cell, derived from Table I).

```
    Input: String characters $S_1(i), S_2(j)$ and previous WF values $D_{i-1,j}, D_{i,j-1}, D_{i-1,j-1}$
    Output: Next WF value $D_{i,j}$
1:  $X = \min\{D_{i-1,j}, D_{i,j-1}\}$, $Y = \min(X, D_{i-1,j-1})$
                                   ▷ $2 \cdot 13b$ ops (adding two $b$-bit numbers)
2:  $Z = Y + 1$                     ▷ $w_{del} = w_{ins} = w_{sub} = 1$
                                   ▷ $5b$ ops (adding $b$-bit and single bit numbers)
3:  Set $S1 = 7$ if Y=7, and 0 otherwise
                                   ▷ MUX1 select, 6 ops (two single-bit ANDs)
4:  Set $MUX1 = Y$ if $S1 = 1$, and $Z$ otherwise
                                   ▷ $3b + 1$ ops
5:  Set $S2 = 1$ if $S_1(i) = S_2(j)$, and 0 otherwise
                                   ▷ MUX2 select, 11 ops (two XNORs + single AND)
6:  return MUX2 value as $D_{i-1,j-1}$ if S2=1, and MUX1 otherwise
                                   ▷ $3b + 1$ ops
                                   ▷ Total: $37b + 19$ ops for a single cell
```

from Eq. (5), require only a single bit due to their two optional origin directions per cell. The traceback information is stored in designated rows within the crossbar memory, typically occupying 7× as many rows as used for the computation.

The advantages of the above WF optimizations are twofold: (1) A reduced memory footprint and (2) lower computational complexity, both of which are due to the decreased bit-width per WF cell and consideration of few sub/super-diagonals rather than the full matrix. These enhancements notably decrease the latency of the linear WF algorithm by 2.8× compared to SW, and enable the efficient computation of a full matrix within a single memristive crossbar row (rather than two rows in SW).

V. DART-PIM: DNA Read-Mapping Accelerator

This section presents the architecture of DART-PIM, which integrates bulk-bitwise memristive PIM with RISC-V cores. We commence by describing the structure and properties of DART-PIM's architecture. Then, we move to describe how each stage of read mapping is performed using DART-PIM.

A. DART-PIM Architecture

The DART-PIM architecture is based on a PIM module, which includes a PIM controller that controls multiple memristive chips, similarly to standard DRAM main memories [60] (where a PIM module is analogous to a DRAM rank). Each PIM module incorporates several memory chips, each equipped with multiple instances of two primary components:

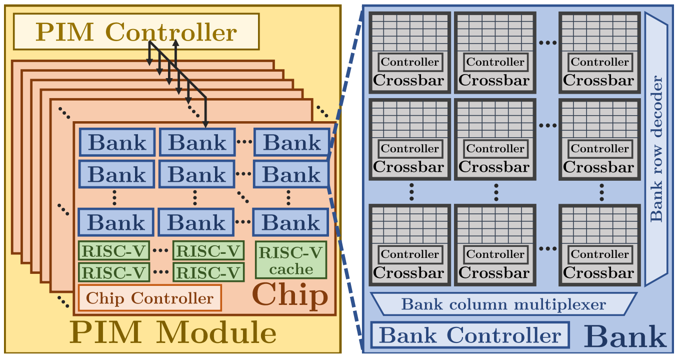

Fig. 5. DART-PIM architecture featuring hybrid RISC-V cores and computing memristive memories. The memory consists of a single PIM module, which contains RISC-V cores with private L1 cache memories integrated within the memristive memory chips. Each chip is divided into banks, each equipped with a dedicated controller. Each crossbar in the bank is responsible for a single reference minimizer.

(1) The DART-PIM memory (*DP-memory*) – A set of banks with a controller and multiple crossbars [60], [61], and (2) the DART-PIM RISC-V (*DP-RISC-V*) – A set of standard RISC-V processors performing a small portion (0.16%) of the affine WF instances, with one of the RISC-V processors (the *main RISC-V*) managing the data arrangement and high-level processing instructions for all DP-memory cores. The

Algorithm 2 The linear WF matrix calculation with $2eth+1$ buffer size.

```
    Input: Strings $S_1$ (read of length $N$) and $S_2$ (reference of length $M$)    ▷ $eth$ parameter set to 6
    Output: Linear WF distance between $S_1$ and $S_2$
 1: Initialize to zero an array WFd of size $2eth+1$ and 3-bit cells    ▷ WF distances buffer
 2: for $i = 0$ to $N-1$ do
 3:     for $j = 0$ to $2eth$ do
 4:         Set $WFd[j]$ to one of the following:
 5:         if ($j == 0$) then    ▷ left edge of WFd
                $\min\left\{\mathrm{WFd}[j] + \mathbb{1}_{S_2[i] \neq S_1[i+j]}, \mathrm{WFd}[j+1]+1\right\}$
 6:         else if ($j == 2eth$) then    ▷ right edge of WFd
                $\min\left\{\mathrm{WFd}[j] + \mathbb{1}_{S_2[i] \neq S_1[i+j]}, \mathrm{WFd}[j-1]+1\right\}$
 7:         else
                $\min\left\{\begin{array}{c}\mathrm{WFd}[j] + \mathbb{1}_{S_2[i] \neq S_1[i+j]}, \quad \mathrm{WFd}[j-1]+1, \\ \mathrm{WFd}[j+1]+1\end{array}\right\}$
 8:         end if
 9:         $\mathrm{WFd}[j] = \min\{\mathrm{WFd}[j], eth+1\}$
10:     end for
11: end for
12: return WFd[$eth$]
```

TABLE II
SUMMARY OF DART-PIM ARCHITECTURE CONFIGURATION.

Module property	Value
Total memory capacity	$256GB$
# PIM Module	1
# Chips per PIM module	16
# Banks per chip	1024
# Crossbars per bank	512
# Columns / rows / Bytes per crossbar	1024 / 256 / 32KB
# RISC-V cores per chip	8
Cache capacity per chip	$256kB$
RISC-V to memory banks BUS width	512 bits

hierarchic structure of DART-PIM is illustrated in Figure 5 with specific architectural parameters given in Table II.

In addition to memories standard read/write capabilities, the memristive crossbars in DART-PIM also support PIM operations with the aid of dedicated per-chip controllers. Each such chip controller oversees multiple banks and their respective crossbars. Upon initiating a PIM request, the pertinent information is transmitted to the target chip in a write operation format [62]. Subsequently, the designated chip controllers issue the necessary sequence of MAGIC NOR operations to all associated banks and crossbars, aligning with the specific instruction. Since all crossbars execute identical tasks, the controllers are quite simple, hence diminishing the burden on the system's area and energy consumption.

The role of each crossbar (or several crossbars, if needed) is to handle all read-mapping stages of a specific minimizer from the reference genome. Given the crossbar dimensions, we chose to work with reads of length 150. The size of each crossbar is 1024 columns by 256 rows, where the rows are partitioned into three disjoint subsets corresponding to the three online read-mapping stages: seeding, pre-alignment filtering (via the linear WF), and read alignment (via the affine WF). Specifically, the rows in each crossbar are divided into the following subsets:

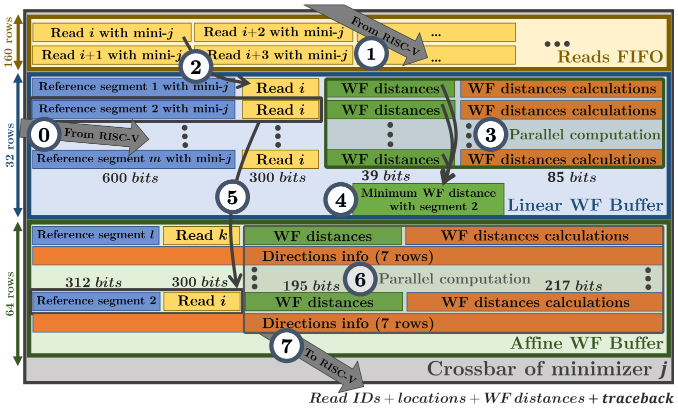

Fig. 6. A memory crossbar with 256 rows and 1024 columns, used for the execution of the WF algorithm between reads and reference segments with minimizer j (*mini-j*), and eth=6. The crossbar is divided into three components: (1) Reads FIFO (light yellow), (2) Linear WF Buffer (light blue), and (3) Affine WF Buffer (light green). The numbered circles represent the steps of the entire read-mapping process: (1) indexing – step ⓪, (1) seeding – step ①, (2) pre-alignment filtering – steps ③ and ④, and (4) read alignment – step ⑥. Steps ② and ⑤ involve intra-crossbar data movements, while step ⑦ entails transmitting the results to the RISC-V. Detailed explanations about this process are given in Sections V-B to V-E.

1) *Reads FIFO* contains 160 rows and stores three reads per row (total of 480 reads associated with the same minimizer).
2) *Linear WF Buffer* contains 32 rows executing multiple instances of the linear WF concurrently for the target minimizer (one instance per row). Each row accommodates a different segment of the reference.
3) *Affine WF Buffer* contains 64 rows, where each eight rows facilitate a single affine WF instance (one row for score calculations and seven rows for optimal alignment recovery). Thus, eight concurrent execution of the affine WF are supported.

Given a target minimizer, Figure 6 depicts the layout of the crossbar and the execution flow. The numbering refers to the specific tasks performed in each area. Due to high variations in minimizer frequency within the reference genome, different minimizers require different compute resources. Infrequent minimizers induce crossbars with low utilization of the linear WF buffer, thus reducing the effective compute per area efficiency. We, therefore, assign the WF calculation of such minimizers to the DP-RISC-V cores offline, saving significant crossbar area.

The frequency of minimizers within the reads also varies significantly. In this case, frequent minimizers are problematic as they fill the Reads FIFO, creating a latency bottleneck. To address this, we design the Reads FIFO to be significantly larger (compared to compute buffers) and limit the maximal

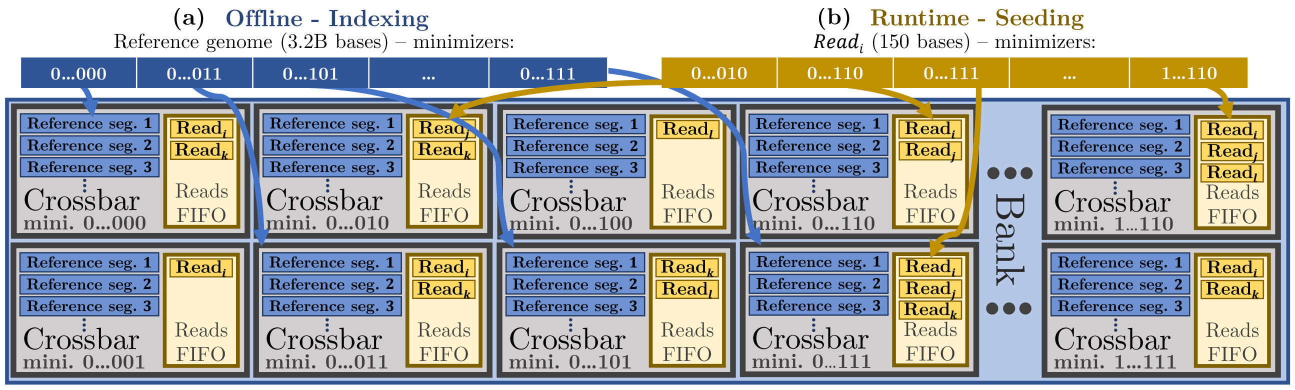

Fig. 7. The crossbars' arrangement of (a) the reference genome during the offline indexing stage, and (b) the reads during the runtime seeding stage.

allowed number of WF matrices computed for the same minimizer. The latter limitation comes at the cost of a slight accuracy degradation (shown to be negligible in Section VII-A), while improving execution time dramatically.

B. Offline Indexing

The offline indexing stage of DART-PIM imports the reference segments themselves (rather than only their addresses) into the linear WF buffers inside the DP-memory crossbars. Each linear WF buffer row is occupied by a single reference segment, representing a single PL, for the minimizer assigned to the relevant crossbar.

The length of the reference segment written to the crossbar is selected to support read alignment regardless of the location of the minimizer within the read. Therefore, the minimum required length is approximately twice the read length, as the minimizer may be at the left/right edge of the read (the precise length is $2(rl + eth) - k$, where *rl* represents the read length). An illustration of the arrangement of indexing data inside a crossbar is depicted near the ⓪ marker in Figure 6. Figure 7a illustrates how indexing is arranged across different crossbars.

Although this approach duplicates the reference segments across crossbars, it is beneficial as it completely eliminates transfers of reference data throughout the computation. Specifically for the human genome, the storage overhead grows about 17×, from 800MB for standard hash-table methods to 13.3GB for DART-PIM. Our results confirm that this strategy is beneficial in terms of latency and energy, saving around 300 billion reference segment transfers from the memory.

C. Online Seeding

The first online stage of DART-PIM is the seeding of every read to its corresponding reference segments according to the read's set of minimizers. Reads continue to enter the Reads FIFO (① in Figure 6) until one of the crossbars has filled its Reads FIFO. Figure 7b depicts the arrangement of reads across different crossbars according to their minimizers.

The seeding process commences when the main RISC-V core sends a list of reads and their minimizers to the PIM controllers. Each such controller is aware of all minimizers relevant to any of its descendent controllers in DART-PIM's hierarchy. This way, only relevant reads are propagated throughout DART-PIM to the crossbar controllers, which add them to their Reads FIFO. When one of the Reads FIFO fills up, it signals the PIM controller to stop sending reads; and the same command is sent to the main RISC-V core to stop the read stream and proceed to the pre-alignment filtering stage.

D. Online Pre-Alignment Filtering

Pre-alignment filtering in DART-PIM is based on the linear WF algorithm. Each minimizer of every read is processed by the linear WF to calculate its similarity scores with all the reference segments of its PLs. By discarding (filtering) reference segments with low scores, the number of PLs per read is reduced.

The filtering process commences with commands sent by the main RISC-V core to the PIM controller, initiating WF cell calculations in all DP-memory crossbars. As the data (read and reference segment) are already stored in-memory, the main RISC-V core can send the same instruction to all chip controllers. Then, each crossbar is assigned a sequence of MAGIC NOR operations and performs WF matrix cell calculations in all its linear WF buffer rows concurrently, instructed by the chip controller. We call this procedure a *linear WF iteration*.

After each linear WF iteration, the processed read is removed from the Reads FIFO. The RISC-V core continues to send reads and repeats the process until all reads are processed and all buffers are empty. This stage is computationally intensive, commonly posing a bottleneck for the entire read-mapping process, hence its throughput is crucial. Within each crossbar, the following steps are performed sequentially to implement a linear WF iteration (which parallels multiple WF instances):

1) First, a read is copied from the Reads FIFO to the crossbar's linear WF buffer (② in Figure 6). At a given linear WF iteration, a crossbar maps a specific minimizer (within a specific read) to multiple reference segments (representing multiple PLs). Hence, the same read is written to all linear WF buffer rows in parallel. Each row performs its computations on a different section of the reference segment, depending on the minimizer's location

within the read. This location is given as an address offset to the crossbar controller alongside the read itself.

2) The next stage is the *linear WF matrix calculation* (③ in Figure 6). The linear WF matrix is computed within the linear WF buffer according to instructions generated by the chip controller for all crossbars in parallel. When this stage is completed, each row contains a linear WF score.
3) Consequently, the minimal value is extracted from the scores stored in the different rows of the linear WF buffer (④ in Figure 6). Although computed serially within each crossbar, the minimum extraction is performed in parallel across many crossbars.
4) Finally, the selected reference segment, corresponding to the minimal WF score, is copied alongside the read to an empty row in the affine WF buffer (⑤ in Figure 6). Since the read and the reference segment are already aligned at this stage, it is possible to copy only the required sub-segment, rather than the entire reference segment that was twice as long (see Section V-B).

E. Online Read Alignment

As in the linear WF iteration, an *affine WF iteration* starts within a crossbar only when the Affine WF buffer is full. Once invoked, the affine WF iteration (⑥ in Figure 6) is calculated in parallel over all crossbar rows as well as across different crossbars. Recall that an affine WF iteration includes both the WF matrix calculation and traceback recovery.

Finally, the results of the affine WF iteration are sent from each crossbar (specifically, each Affine WF buffer row) to the main RISC-V core (⑦ in Figure 6). The results include the read index, the PL within the reference genome, the affine score, and the traceback. The RISC-V core maintains in dedicated crossbars a list of reads and their "best so far" PL candidate, obtained by selecting the PL with minimal WF score received from the DP-memory for every read.

VI. DART-PIM Evaluation Methodology

DART-PIM evaluation consists of assessing its accuracy, execution time, energy, and area. We evaluate the results using several simulators and compare them to two platforms: (1) NVIDIA Parabricks [37], executed on an NVIDIA DGX A100 (with eight A100 Tensor Core GPUs), and (2) state-of-the-art GenASM [25] which accelerates computations within the logic layer of a 3D-stacked memory [63]–[66].

To comprehensively evaluate the DART-PIM architecture, we developed four simulators: (1) a full-system simulator to test the overall behavior and performance; (2) a single-crossbar simulator designed to validate functionality and generate precise metrics for a single WF iteration; (3) a DP-RISC-V simulator for latency evaluation and that includes a main RISC-V module and the remaining RISC-V cores; and (4) synthesized controllers for energy consumption and area evaluation. The implementations of all simulators, listed below, are accessible in the paper's GitHub repository [58]:

1) ***Full-System Simulator****:* This is an in-house C++ simulator that emulates the entire system operation. It incorporates the full-size PIM memory and executes all (offline and online) stages of read mapping using the aforementioned datasets. During the offline phase, the simulator partitions the reference genome into crossbars, as described in Section V-B. It then conducts seeding, filtering, and read alignment, as discussed in Sections V-C, V-D, and V-E. The simulator also provides the exact number of linear and affine WF instances and iterations performed by DART-PIM throughout the process. Additionally, it quantifies the total utilized memory capacity and the resulting read-mapping accuracy.

2) ***Single-Crossbar Simulator****:* This is an in-house MATLAB [67] simulator that models a single crossbar operation cycle by cycle. This simulator verifies the functionality of algorithms implemented using MAGIC NOR operations within DART-PIM. It provides an exact calculation of the execution time by computing the number of cycles required for executing linear and affine WF, including pre- and post-processing operations. Additionally, it quantifies the energy consumption by counting the number of write/MAGIC NOR switches and number of read bits.

3) ***RISC-V Simulator****:* The simulator developed for the DART-PIM RISC-V cores (*DP-RISC-V*) is based on GEM5 [68]. A designated module emulates the main RISC-V core, while the other RISC-V cores, used for computing tasks, are modeled by other modules, each with a cache memory. This simulator evaluates the execution time of each RISC-V core within the DP-RISC-V, emulating the WF computation distribution between the DP-RISC-V and DP-memory, as well as the communication between them.

4) ***PIM/Chip/Bank/Crossbar Controllers****:* We synthesized the controllers included in DART-PIM, as depicted in Figure 5, to estimate their area and power consumption. The controllers were designed and implemented in SystemVerilog and synthesized using Synopsys Design Compiler [69], targeting TSMC 28nm CMOS technology.

Our evaluation datasets comprise the full Homo Sapiens (human) reference genome (GRCh38_latest_genomic.fna [70]) along with real short-read datasets generated by Illumina HiSeq X [9] (HG002.hiseqx.pcr-free.30x.R[1-2].fastq [70]). These read datasets comprise 389 million reads, each 150 base pairs long.

VII. Experimental Results and Evaluation

The evaluation of DART-PIM is based on extensive simulations performed using the simulators outlined in Section VI. Table III lists the parameters used in the simulations. General parameters (i.e., read-mapping and Wagner-Fischer parameters) were set according to custom values according to existing literature and remain constant across all simulations. The remaining parameters, which are DART-PIM specific, were selected using an exhaustive search aimed at optimizing system latency and energy consumption. The linear and affine buffer sizes were selected to be 32 and 64 rows, respectively,

TABLE III
SUMMARY OF DART-PIM USED PARAMETERS.

Parameter	Context	Value
Read length (rl)	Read mapping	150
Minimizer length (k)	Read mapping	12
Minimizer window length (W)	Read mapping	30
Linear (affine) error threshold (eth)	Read mapping	6 (31)
w_{sub}=w_{ins}=w_{del}=w_{op}=w_{ex}	Wagner-Fischer	1
Reads FIFO # rows	DART-PIM	160
Linear Buffer # rows	DART-PIM	32
Affine Buffer # rows	DART-PIM	64
Low threshold (lowTh)	DART-PIM	3
Maximum # allowed read (max reads)	DART-PIM	50K

TABLE IV
SUMMARY OF CYCLE AND SWITCH COUNTS FOR EACH MEMORY OPERATION FROM THE SINGLE-CROSSBAR SIMULATOR DURING A SINGLE WF CALCULATION.

		MAGIC NOR	Writes	Reads	Total
Linear WF	# Cycles	254,585	4035	0	258,620
	# Switches	254,384	255,499	0	509,883
Affine WF	# Cycles	1,288,281	20,418	0	1,308,699
	# Switches	1,271,921	1,277,495	0	2,549,416

resulting in a Reads FIFO with $256 - 32 - 64 = 160$ rows (recall that the crossbar size is 256×1024).

Another important parameter is the low threshold *lowTh* $=$ 3. The low threshold balances between DP-memory capacity and RISC-V workload by determining whether affine WF computations are performed by DP-memory (minimizer frequency above *lowTh*) or DP-RISC-V (minimizer frequency below *lowTh*). To avoid highly non-uniform assignment of reads per crossbar, we also bound the number of read handled by any single crossbar to, at most, *maxReads*=50K. This value was found to yield a reasonable trade-off between execution time and accuracy impact. The description of the parameter search is included in the paper's GitHub repository [58].

This section details the simulated results of DART-PIM's accuracy, execution time, energy consumption, and area, and compares them to NVIDIA Parabricks [37] and GenASM [25].

A. DART-PIM Accuracy

We compare DART-PIM to *BWA-MEM* [71]. Specifically, we compare the locations identified by DART-PIM and by BWA-MEM. The accuracy is defined as the fraction of DART-PIM mappings that match the BWA-MEM results. DART-PIM accuracy is 99.4%, while Parabricks' is 99.9% [37] and GenAsm's 96.6% [25].

B. Single Crossbar Evaluation

The fundamental building block of our performance evaluation is a cycle-accurate simulation of executing WF instances

TABLE V
SUMMARY OF CONSERVATIVELY SCALED MAGIC NOR AND WRITE SWITCHING ENERGY AND CYCLE TIME FROM [72].

Operations	Energy/Time
MAGIC/write conservatively scaled cycle	2ns
MAGIC conservatively scaled energy	90fJ/bit
Write conservatively scaled energy	90fJ/bit

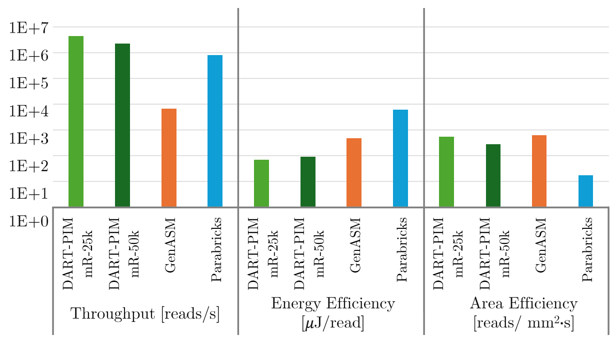

Fig. 8. Comparison of throughput (reads per second), energy efficiency (read per joule), and area efficiency (throughput per squared millimeter) of DART-PIM with different values of maxReads (mR = 25K, 50K) and state-of-the-art approaches.

on a single crossbar, precisely as employed by DART-PIM. The Single-Crossbar simulator counts the overall number of MAGIC NOR switches, number of read bits and write switches, and the number of cycles per operation, encompassing both linear and affine algorithms.

The number of cycles and switches utilized by a single WF instance is outlined in Table IV. Note that thanks to inter-crossbar and intra-crossbar (different rows) parallelism, we are able to perform multiple WF instances at the same number of cycles. The energy consumption of nucleus operations (write / MAGIC NOR switches), required for calculating the total energy per WF instance, appears in Table V. Given these values, the number of cycles required for the linear (affine) WF is $258,620$ $(1,308,699)$, while the overall energy consumed by the linear (affine) WF amounts to $509,883 \times 90fJ = 45.9nJ$ $(2,549,416 \times 90fJ = 229nJ)$.

The simulated number of cycles and switches fits our theoretical estimates: the total MAGIC NOR cycles for a single linear WF computation is the product of 1950 WF cells ($2eth + 1 = 13$ cells per row times 150 rows) and 130 cycles per cell, which amounts to $253,500$ cycles (vs. $254,585$ simulated cycles). The $\tilde{1}000$ cycles gap is due to the first matrix row/column initializations, in addition to step ④ in Figure 6.

C. DART-PIM Execution Time and Throughput

Figure 8 (left subplot) compares the throughput of DART-PIM, NVIDIA Parabricks [37], and GenASM [25], measured by mapped reads per second. For our selected dataset (389M reads), DART-PIM execution time spans between 86.5s (for maxReads = 25k) to 174s (for maxReads = 50k). On the same data, Parabricks execution time is 8.3 minutes (498s) on DGX A100 while GenASM requires 16.2 hours ($58,350s$ based on the reported 200K reads/30s). DART-PIM's times (using the slower maxRead = 50k) compared to Parabricks and GenASM are 5.7× and 335×, respectively, better.

Figure 9a shows the breakdown of DART-PIM's execution time. The execution time is determined by the maximum of three latency factors: (1) Reading the results from the DP-memory (light green), (2) overall latency of DP-RISC-

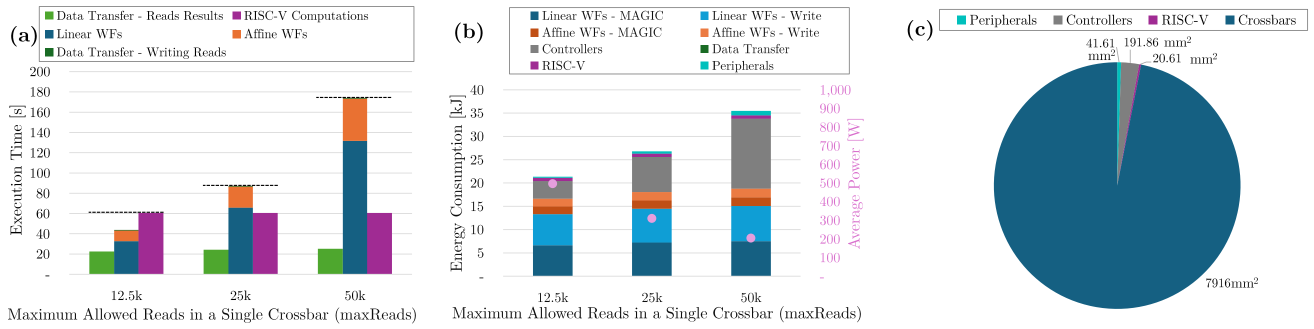

Fig. 9. Performance of DART-PIM for 389 million reads: (a) execution time across various maxReads settings, alongside time of the data transfer and DP-RISC-V computations, (b) energy consumption and average power across different maxReads configurations, and (c) area of crossbars, periphery, controllers, and RISC-V.

V computations (purple), and (3) the time required to write the reads to DP-memory and compute within the DP-memory (aggregate of remaining colors). To ensure that DART-PIM is primarily constrained by the computational capabilities of PIM rather than being limited by data transfers or RISC-V processing, the latter should be the most time-consuming.

The DP-memory execution time is determined as

$$T_{\text{DPmemory}} = (K_L \cdot N_L + K_A \cdot N_A) \cdot T_{clk}, \qquad (6)$$

where T_{clk} is the clock cycle time (see Table V), K_L and K_A are the number of linear and affine iterations (determined by the full-system simulator), N_L is the number of cycles per linear WF iteration, and N_A is the number of cycles per affine WF iteration (see Table IV).

The maximal number of reads per crossbar trades off accuracy and execution time: An increase in maxReads enhances accuracy, at the expense of a linearly higher total execution time. As the number of linear/affine WF instances grows sub-linearly with maxReads, the throughput (reads per second) decreases.

We chose to have 128 RISC-V cores (8 per chip) to guarantee that DP-RISC-V computations do not pose a bottleneck. We evaluated its total execution time to be $60.6s$ assuming all cores are working in parallel, on (only) affine WF instances. This underlines the extreme efficiency of DP-memory, as it computes 99.84% of the affine WF instances in only three times the latency.

D. DART-PIM Energy Efficiency

Figure 8 (middle subplot) compares the energy efficiency [73] of DART-PIM, NVIDIA Parabricks, and GenASM, measured as the consumed energy per mapped read. For our selected dataset ($389M$ reads), the energy consumed by Parabricks and GenASM is 2.4MJ and 188.3KJ, based on average power consumption of 4850W and 3.23W, respectively. The energy consumed by DART-PIM (for maxReads = 50k) is only 35.1KJ (average power of 203W), thereby achieving a 67× and 5.3× respective improvement over the competing approaches.

Figure 9b shows a breakdown of DART-PIM's energy. The energy of DART-PIM is consumed by three components: DP-memory (controller, peripheral circuits, and crossbars), DP-RISC-V (cores and caches), and the mutual data transfer. Given an energy consumption of $\mathcal{E}_{MAGIC}$ and $\mathcal{E}_{WRITE}$ for MAGIC and WRITE operations, respectively, the crossbars energy is determined by

TABLE VI
TIME, ENERGY, AND AREA OF DATA TRANSFER, RISC-V CORES, PERIPHERALS, AND CONTROLLERS (OF A SINGLE UNIT). [CS] INDICATES DATA GENERATED BY THE CONTROLLER SIMULATOR AND [RVS] DENOTES DATA FROM THE RISC-V SIMULATOR.

Operations	Time	Energy/Power	Area	# Units
Write (DP-RISC-V to DP-memory)	32GB/s [72]	11.7pJ/bit [62]	-	-
Read (DP-memory to DP-RISC-V)	32GB/s [72]	5.64pJ/bit [62]	-	-
RISC-V (single affine SW instance)	$215\mu s$ [RVs]	40mW [74]	$0.11mm^2$ [74]	128
RISC-V cache	-	8mW [75]	$0.05mm^2$ [75]	128
Crossbar controller [Cs]	-	$9.43\mu W$	$21\mu m^2$	8M
Bank controller [Cs]	-	0.42mW	$939\mu m^2$	16k
Chip controller [Cs]	-	9.4mW	$20{,}091\mu m^2$	16
PIM controller [Cs]	-	0.5mW	$938\mu m^2$	1
Decode and Drive Unit [72]	-	$129.1\mu W$	$277\mu m^2$	16k
R/W Circuit [72]	-	10pW	$0.06\mu m^2$	8M
Selector Passgates [72]	-	20pW	$0.001\mu m^2$	$1024\times 8M$
Driver Passgate [72]	-	20pW	$0.001\mu m^2$	$256\times 8M$

$$\begin{aligned}\mathcal{E}_{\text{crossbars}} &= (\mathcal{E}_{\text{MAGIC}} \cdot S_L^{\text{MAGIC}} + \mathcal{E}_{\text{WRITE}} \cdot S_L^{\text{WRITE}}) \cdot J_L \\ &+ (\mathcal{E}_{\text{MAGIC}} \cdot S_A^{\text{MAGIC}} + \mathcal{E}_{\text{WRITE}} \cdot S_A^{\text{WRITE}}) \cdot J_A,\end{aligned} \qquad (7)$$

where S_L^{MAGIC} (S_A^{MAGIC}) is the number of linear (affine) MAGIC switches, S_L^{WRITE} (S_A^{WRITE}) is the number of linear (affine) write switches, and J_L (J_A) is the number of linear (affine) instances. The values of these parameters are given in Tables V and IV.

The power analysis of the RISC-V cores was conducted using the AndesCore AX25 [74] and is detailed in Table VI. All RISC-V cores with their caches consume 6.1W of power, as shown in Figure 9b (purple). Based on the controller configuration from Table II and corresponding controller simulation results listed in Table VI, the aggregated average power consumption is 86W, yielding the energy consumption appearing in Figure 9b (grey).

The remaining energy consumption corresponds to memory peripherals and data transfers. The peripheral circuits were evaluated by circuit synthesis analysis conducted by RACER [72] and amounted to 5.7W (see details in Table VI). The data transfer energy comprises the writeout of reads to DP-memory (1.1J) and readout of computation results from the DP-memory (75.4J), as evaluated by CONCEPT [62]. Figure 9b presents the estimates for memory peripherals (turquoise) and data transfers (green).

As the value of maxReads increases, both the controllers' and peripheral devices' energy consumption grows linearly, along with a linear increase in execution time, while the power consumption remains relatively stable. The energy consumption for DP-memory computations, however, rises only from 16.6kJ at maxReads = 12.5k to 18.8kJ at maxReads = 50k. This increase corresponds to the marginal growth in WF instances computed by DP-memory.

E. DART-PIM Area Efficiency

Figure 8 (right subplot) compares the area efficiency [73] of DART-PIM, NVIDIA Parabricks, and GenASM, evaluated as the read-mapping throughput per chip area (reads per second per millimeter square). DART-PIM's area efficiency depends on maxReads, spanning between $273\frac{\text{reads}}{\text{mm}^2\cdot s}$ for maxReads=25K and $545\frac{\text{reads}}{\text{mm}^2\cdot s}$ for maxReads=50K. This positions DART-PIM close to GenASM's $624\frac{\text{reads}}{\text{mm}^2\cdot s}$ [25] and significantly better than NVIDIA Parabricks' $17\frac{\text{reads}}{\text{mm}^2\cdot s}$.

We used the reported 10.69mm^2 as GenASM's area [25]. The area of Parabricks was calculated considering eight A100 GPUs, each combined with six HBMs stacks of nine 8Gb (eight units and a 8Gb buffer die), each with an area of 91.99mm^2 [76], [77]. The total area of A100 GPUs and their HBMs is, therefore, $8 \cdot (826\text{mm}^2 + 6 \cdot 9 \cdot 91.99\text{mm}^2) = 46,348\text{mm}^2$.

The total area for DART-PIM amounts to 8170mm^2, based on aggregating its four components: (1) crossbar arrays, (2) controllers, (3) memory peripherals, and (4) RISC-V units. Figure 9c shows a breakdown of DART-PIM's area, illustrating that crossbars occupy 96.9% of the area. Since all crossbars performs similar tasks, we were able to substantially simplify the CMOS-based crossbars controllers and save significant chip area.

The area values for each of DART-PIM's components are listed in Table VI for CMOS 28nm technology. Specifically, the area of 128 DP-RISC-V cores amounts to 14.2mm^2 (based on AndesCore AX25 [74]), and the area of their caches is 6.4mm^2; the area of the controllers sums up to 191.9mm^2 (determined by synthesis results); and the peripherals occupy 53.6mm^2 (based on 15nm synthesis from RACER [72] using the Synopsis Design Compiler, scaled to the target 28nm technology [78]). The remaining area of DART-PIM comprises $8M$ crossbars, as extracted from the full-system simulator. Based on [79], each crossbar cell has a feature size (F) of 30nm, resulting in a memory cell area of $4F^2 = 3600\text{nm}^2$. With a crossbar size of 256×1024, the area per crossbar is $944\mu\text{m}^2$ (and 7916mm^2 for all crossbars). Note that the total memory capacity of DART-PIM is $8M \times 1024 \times 256$ resulting in $256GB$. This confirms the practicality of DART-PIM as it aligns with common memory DIMMs, such as the Intel Optane persistent memory [80].

VIII. Related Work

To the best of our knowledge, DART-PIM represents the first attempt to improve the entire read mapping by eliminating data movement between the different read mapping stages. By consolidating all stages of read mapping within the same physical memory, PIM effectively eliminates the need for external data transfer. Recent research efforts have focused primarily on two distinct approaches.

The first approach targets the acceleration of the pre-alignment filtering stage in the read-mapping process to alleviate the computational burden during the subsequent read alignment stage. Various filters have been proposed for this purpose, including FastHASH [81] for standard CPUs, GRIM Filter [82] implemented within 3D-stacked memory, embedded dynamic memory based RASSA [83] and GenCache [22], an in-cache accelerator enhancing the filtering efficiency of GenAx [84]. Additionally, FPGA-based solutions such as MAGNET [85], GateKeeper [86] and Shouji [20], alongside PIM-based filters [29], leverage parallelism to expedite the filtering process.

The second approach utilizes specialized software or hardware accelerators tailored to the demanding read alignment process. Noteworthy software solutions include Minimap2 [87] and BWA-MEM [71]. On the hardware side, Parasail [88] leverages SIMD-capable CPUs, while GPU-RMAP [89] and GSWABE [90] utilize GPUs. Additionally, FPGAs are employed by FPGASW [91] and ASAP [92], while ASIC-based accelerators such as Darwin [23] and GenAx [84] have also been proposed. Recently, PIM-based solutions such as [93], RADAR [94], FindeR [95], AligneR [96], RAPID [26], and BioSEAL [28] have emerged as promising alternatives for enhancing read alignment efficiency.

While GenASM [25] markedly enhances both filtering and read alignment, its reliance on data transfers between different stages restricts overall performance improvement. In contrast, DART-PIM mitigates the data transfer bottleneck and enables very high parallelism by implementing all stages of read mapping in-memory.

IX. Conclusion

We introduce DART-PIM, a novel framework for accelerating the entire DNA read-mapping process using in-memristive memory processing. DART-PIM integrates all stages of read mapping within the same physical crossbars, eliminating the performance bottleneck and energy costs associated with data transfers. Our evaluation demonstrates that DART-PIM, utilizing RRAM and memristive logic, achieves significant speedup and improved energy efficiency compared to state-of-the-art existing accelerator and GPU implementations.

The DART-PIM concept and architecture are not limited to RRAM or memristive logic and could be adapted to leverage other PIM techniques, including those based on DRAM, SRAM, and different emerging memory technologies.

References

[1] G. S. Ginsburg and H. F. Willard, "Genomic and personalized medicine: foundations and applications," *Translational research*, vol. 154, no. 6, pp. 277–287, 2009.

[2] L. Chin, J. N. Andersen, and P. A. Futreal, "Cancer genomics: from discovery science to personalized medicine," *Nature medicine*, vol. 17, no. 3, pp. 297–303, 2011.

[3] E. A. Ashley, "Towards precision medicine," *Nature Reviews Genetics*, vol. 17, no. 9, pp. 507–522, 2016.

[4] M. Hassan, F. M. Awan, A. Naz, E. J. deAndrés Galiana, O. Alvarez, A. Cernea, L. Fernández-Brillet, J. L. Fernández-Martínez, and A. Kloczkowski, "Innovations in genomics and big data analytics for personalized medicine and health care: A review," *International journal of molecular sciences*, vol. 23, no. 9, p. 4645, 2022.

[5] Y. Yang, B. Xie, and J. Yan, "Application of next-generation sequencing technology in forensic science," *Genomics, proteomics & bioinformatics*, vol. 12, no. 5, pp. 190–197, 2014.

[6] C. Børsting and N. Morling, "Next generation sequencing and its applications in forensic genetics," *Forensic Science International: Genetics*, vol. 18, pp. 78–89, 2015.

[7] Illumina, Inc., "NextSeq 1000 and 2000." [Online]. Available: https://www.illumina.com/systems/sequencing-platforms/nextseq-1000-2000/specifications.html

[8] ——, "NovaSeq 6000Dx." [Online]. Available: https://www.illumina.com/systems/sequencing-platforms/novaseq-6000dx.html

[9] ——, "HiSeq X." [Online]. Available: https://www.illumina.com/systems/sequencing-platforms/hiseq-x.html

[10] Oxford Nanopore Technologies, "PromethION." [Online]. Available: https://nanoporetech.com/

[11] M. Alser, Z. Bingöl, D. S. Cali, J. Kim, S. Ghose, C. Alkan, and O. Mutlu, "Accelerating genome analysis: A primer on an ongoing journey," *IEEE Micro*, vol. 40, no. 5, pp. 65–75, 2020.

[12] S. Canzar and S. L. Salzberg, "Short read mapping: an algorithmic tour," *Proceedings of the IEEE*, vol. 105, no. 3, pp. 436–458, 2015.

[13] M. Roberts, W. Hayes, B. Hunt, S. Mount, and J. Yorke, "Reducing storage requirements for biological sequence comparison," *Bioinformatics (Oxford, England)*, vol. 20, pp. 3363–9, 01 2005.

[14] P. A. Hall and G. R. Dowling, "Approximate string matching," *ACM computing surveys (CSUR)*, vol. 12, no. 4, pp. 381–402, 1980.

[15] E. Ukkonen, "Algorithms for approximate string matching," *Information and control*, vol. 64, no. 1-3, pp. 100–118, 1985.

[16] G. Navarro, "A guided tour to approximate string matching," *ACM computing surveys (CSUR)*, vol. 33, no. 1, pp. 31–88, 2001.

[17] V. I. Levenshtein *et al.*, "Binary codes capable of correcting deletions, insertions, and reversals," in *Soviet physics doklady*, vol. 10, no. 8. Soviet Union, 1966, pp. 707–710.

[18] S. B. Needleman and C. D. Wunsch, "A general method applicable to the search for similarities in the amino acid sequence of two proteins," *Journal of molecular biology*, vol. 48, no. 3, pp. 443–453, 1970.

[19] T. F. Smith, M. S. Waterman *et al.*, "Identification of common molecular subsequences," *Journal of molecular biology*, vol. 147, no. 1, pp. 195–197, 1981.

[20] M. Alser, H. Hassan, A. Kumar, O. Mutlu, and C. Alkan, "Shouji: a fast and efficient pre-alignment filter for sequence alignment," *Bioinformatics*, vol. 35, no. 21, pp. 4255–4263, 2019.

[21] W. Huangfu, X. Li, S. Li, X. Hu, P. Gu, and Y. Xie, "Medal: Scalable dimm based near data processing accelerator for dna seeding algorithm," in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, 2019, pp. 587–599.

[22] A. Nag, C. Ramachandra, R. Balasubramonian, R. Stutsman, E. Giacomin, H. Kambalasubramanyam, and P.-E. Gaillardon, "Gencache: Leveraging in-cache operators for efficient sequence alignment," in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, 2019, pp. 334–346.

[23] Y. Turakhia, G. Bejerano, and W. J. Dally, "Darwin: A genomics co-processor provides up to 15,000 x acceleration on long read assembly," *ACM SIGPLAN Notices*, vol. 53, no. 2, pp. 199–213, 2018.

[24] E. C. Hayden, "Technology: The $1,000 genome," *Nature*, vol. 507, no. 7492, pp. 294–296, 2014.

[25] D. S. Cali, G. S. Kalsi, Z. Bingöl, C. Firtina, L. Subramanian, J. S. Kim, R. Ausavarungnirun, M. Alser, J. Gomez-Luna, A. Boroumand *et al.*, "GenASM: A high-performance, low-power approximate string matching acceleration framework for genome sequence analysis," in *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2020, pp. 951–966.

[26] S. Gupta, M. Imani, B. Khaleghi, V. Kumar, and T. Rosing, "Rapid: A reram processing in-memory architecture for dna sequence alignment," in *2019 IEEE/ACM International Symposium on Low Power Electronics and Design (ISLPED)*, 2019, pp. 1–6.

[27] S. Diab, A. Nassereldine, M. Alser, J. Gómez Luna, O. Mutlu, and I. El Hajj, "A framework for high-throughput sequence alignment using real processing-in-memory systems," *Bioinformatics*, vol. 39, no. 5, p. btad155, 2023.

[28] R. Kaplan, L. Yavits, and R. Ginosasr, "Bioseal: In-memory biological sequence alignment accelerator for large-scale genomic data," in *Proceedings of the 13th ACM International Systems and Storage Conference*, 2020, pp. 36–48.

[29] M. Khalifa, R. Ben-Hur, R. Ronen, O. Leitersdorf, L. Yavits, and S. Kvatinsky, "Filtpim: In-memory filter for dna sequencing," in *2021 28th IEEE International Conference on Electronics, Circuits, and Systems (ICECS)*. IEEE, 2021, pp. 1–4.

[30] O. Mutlu, S. Ghose, J. Gómez-Luna, and R. Ausavarungnirun, "Processing data where it makes sense: Enabling in-memory computation," *Microprocessors and Microsystems*, vol. 67, pp. 28–41, 2019.

[31] J. Reuben, R. Ben-Hur, N. Wald, N. Talati, A. H. Ali, P.-E. Gaillardon, and S. Kvatinsky, "Memristive logic: A framework for evaluation and comparison," in *2017 27th International Symposium on Power and Timing Modeling, Optimization and Simulation (PATMOS)*, 2017, pp. 1–8.

[32] H. Akinaga and H. Shima, "Resistive random access memory (reram) based on metal oxides," *Proceedings of the IEEE*, vol. 98, no. 12, pp. 2237–2251, 2010.

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

[34] G. Hu, J. Lee, J. Nowak, J. Z. Sun, J. Harms, A. Annunziata, S. Brown, W. Chen, Y. Kim, G. Lauer *et al.*, "Stt-mram with double magnetic tunnel junctions," in *2015 IEEE International Electron Devices Meeting (IEDM)*. IEEE, 2015, pp. 26–3.

[35] O. Mutlu, S. Ghose, J. Gómez-Luna, and R. Ausavarungnirun, "Processing data where it makes sense: Enabling in-memory computation," *Microprocessors and Microsystems*, vol. 67, pp. 28–41, 2019. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0141933118302291

[36] R. A. Wagner and M. J. Fischer, "The string-to-string correction problem," *Journal of the ACM (JACM)*, vol. 21, no. 1, pp. 168–173, 1974.

[37] NVIDIA, "Parabricks." [Online]. Available: https://www.nvidia.com/en-u/clara/parabricks/

[38] H. Li, "Minimap and miniasm: fast mapping and de novo assembly for noisy long sequences," *Bioinformatics*, vol. 32, no. 14, pp. 2103–2110, 2016.

[39] M. Alser *et al.*, "Accelerating genome analysis: A primer on an ongoing journey," *IEEE Micro*, 2020.

[40] T. F. Smith, M. S. Waterman *et al.*, "Identification of common molecular subsequences," *Journal of molecular biology*, vol. 147, no. 1, pp. 195–197, 1981.

[41] S. B. Needleman and C. D. Wunsch, "A general method applicable to the search for similarities in the amino acid sequence of two proteins," *Journal of molecular biology*, vol. 48, no. 3, pp. 443–453, 1970.

[42] O. Gotoh, "An improved algorithm for matching biological sequences," *Journal of molecular biology*, vol. 162, no. 3, pp. 705–708, 1982.

[43] M. O. Dayhoff, "A model of evolutionary change in proteins," *Atlas of protein sequence and structure*, vol. 5, pp. 89–99, 1972.

[44] B. Perach, R. Ronen, and S. Kvatinsky, "On consistency for bulk-bitwise processing-in-memory," in *2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, 2023, pp. 705–717.

[45] B. Perach, R. Ronen, B. Kimelfeld, and S. Kvatinsky, "Understanding bulk-bitwise processing in-memory through database analytics," *IEEE Transactions on Emerging Topics in Computing*, vol. 12, no. 1, pp. 7–22, 2024.

[46] S. Kvatinsky, D. Belousov, S. Liman, G. Satat, N. Wald, E. G. Friedman, A. Kolodny, and U. C. Weiser, "MAGIC - Memristor-Aided Logic," *IEEE Transactions on Circuits and Systems II: Express Briefs*, vol. 61, no. 11, pp. 895–899, nov 2014.

[47] C. Xu, D. Niu, N. Muralimanohar, R. Balasubramonian, T. Zhang, S. Yu, and Y. Xie, "Overcoming the challenges of crossbar resistive memory

architectures," in *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*, Feb 2015, pp. 476–488.

[48] Z. Wei, T. Takagi, Y. Kanzawa, Y. Katoh, T. Ninomiya, K. Kawai, S. Muraoka, S. Mitani, K. Katayama, S. Fujii *et al.*, "Demonstration of high-density reram ensuring 10-year retention at 85 c based on a newly developed reliability model," in *2011 international electron devices meeting*. IEEE, 2011, pp. 31–4.

[49] J. Borghetti, G. S. Snider, P. J. Kuekes, J. J. Yang, D. R. Stewart, and R. S. Williams, "'Memristive' Switches Enable 'Stateful' Logic Operations via Material Implication," *Nature*, vol. 464, no. 7290, pp. 873–876, Apr. 2010.

[50] S. Gupta, M. Imani, and T. Rosing, "FELIX: Fast and Energy-Efficient Logic in Memory," in *2018 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, 2018, pp. 1–7.

[51] B. Hoffer, V. Rana, S. Menzel, R. Waser, and S. Kvatinsky, "Experimental Demonstration of Memristor-Aided Logic (MAGIC) Using Valence Change Memory (VCM)," *IEEE transactions on electron devices*, vol. 67, no. 8, pp. 3115 – 3122, 2020. [Online]. Available: https://juser.fz-juelich.de/record/885492

[52] N. Peled, R. Ben-Hur, R. Ronen, and S. Kvatinsky, "X-magic: Enhancing pim using input overwriting capabilities," 10 2020, pp. 64–69.

[53] O. Leitersdorf, B. Perach, R. Ronen, and S. Kvatinsky, "Efficient error-correcting-code mechanism for high-throughput memristive processing-in-memory," in *2021 58th ACM/IEEE Design Automation Conference (DAC)*. IEEE, 2021, pp. 199–204.

[54] O. Leitersdorf, R. Ronen, and S. Kvatinsky, "Making memristive processing-in-memory reliable," in *2021 28th IEEE International Conference on Electronics, Circuits, and Systems (ICECS)*. IEEE, 2021, pp. 1–6.

[55] M. M. Sabry Aly, T. F. Wu, A. Bartolo, Y. H. Malviya, W. Hwang, G. Hills, I. Markov, M. Wootters, M. M. Shulaker, H.-S. Philip Wong, and S. Mitra, "The n3xt approach to energy-efficient abundant-data computing," *Proceedings of the IEEE*, vol. 107, no. 1, pp. 19–48, 2019.

[56] R. Ben-Hur, R. Ronen, A. Haj-Ali, D. Bhattacharjee, A. Eliahu, N. Peled, and S. Kvatinsky, "SIMPLER MAGIC: Synthesis and Mapping of In-Memory Logic Executed in a Single Row to Improve Throughput," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 39, no. 10, pp. 2434–2447, 2020.

[57] R. Ben-Hur, N. Wald, N. Talati, and S. Kvatinsky, "Simple magic: Synthesis and in-memory Mapping of logic execution for memristor-aided logic," in *2017 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, Nov 2017, pp. 225–232.

[58] Anonymous, "DART-PIM Github Repository." [Online]. Available: https://anonymous.4open.science/r/DART_PIM_Repository-BD54/

[59] E. Ukkonen, "Algorithms for approximate string matching," *Information and Control*, vol. 64, no. 1, pp. 100–118, 1985, international Conference on Foundations of Computation Theory. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0019995885800462

[60] B. Jacob, D. Wang, and S. Ng, *Memory systems: cache, DRAM, disk*. Morgan Kaufmann, 2010.

[61] N. Talati, S. Gupta, P. Mane, and S. Kvatinsky, "Logic Design Within Memristive Memories Using Memristor-Aided loGIC (MAGIC)," *IEEE Transactions on Nanotechnology*, vol. 15, no. 4, pp. 635–650, July 2016.

[62] N. Talati, H. Ha, B. Perach, R. Ronen, and S. Kvatinsky, "CONCEPT: A Column Oriented Memory Controller for Efficient Memory and PIM Operations in RRAM," *IEEE Micro*, vol. 39, pp. 33–43, 2 2019.

[63] Hybrid Memory Cube Consortium, "Hybrid Memory Cube Specification 2.1," 2015.

[64] JEDEC Solid State Technology Assn., "JESD235C: High Bandwidth Memory (HBM) DRAM," 2020.

[65] D. Lee, S. Ghose, G. Pekhimenko, S. Khan, and O. Mutlu, "Simultaneous multi-layer access: Improving 3d-stacked memory bandwidth at low cost," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 12, no. 4, pp. 1–29, 2016.

[66] J. Ahn, S. Yoo, O. Mutlu, and K. Choi, "Pim-enabled instructions: A low-overhead, locality-aware processing-in-memory architecture," *ACM SIGARCH Computer Architecture News*, vol. 43, no. 3S, pp. 336–348, 2015.

[67] MathWorks, "Matlab R2021a." [Online]. Available: https://www.mathworks.com/products/matlab.html

[68] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti *et al.*, "The gem5 simulator," *ACM SIGARCH computer architecture news*, vol. 39, no. 2, pp. 1–7, 2011.

[69] Synopsys, "Design Compiler." [Online]. Available: https://www.synopsys.com/implementation-and-signoff/rtl-synthesis-test/dc-ultra.html

[70] N. N. C. for Biotechnology Information, "National Library of Medicine," 2023. [Online]. Available: https://www.ncbi.nlm.nih.gov/datasets/genome/GCF_000001405.40/

[71] H. Li, "Aligning sequence reads, clone sequences and assembly contigs with BWA-MEM," *arXiv preprint arXiv:1303.3997*, 2013.

[72] M. S. Truong, E. Chen, D. Su, L. Shen, A. Glass, L. R. Carley, J. A. Bain, and S. Ghose, "RACER: Bit-pipelined processing using resistive memory," in *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, 2021, pp. 100–116.

[73] A. Pedram, S. Richardson, S. Galal, S. Kvatinsky, and M. A. Horowitz, "Dark Memory and Accelerator-Rich System Optimization in the Dark Silicon Era," *IEEE Design Test*, vol. 34, no. 2, pp. 39–50, Apr. 2017.

[74] Andrey Technology, "AndesCore AX25." [Online]. Available: https://www.andestech.com/en/products-solutions/andescore-processors/riscv-ax25/

[75] S.-Y. Wu, J. Liaw, C. Lin, M. Chiang, C. Yang, J. Cheng, M. Tsai, M. Liu, P. Wu, and C. Chang, "A highly manufacturable 28nm cmos low power platform technology with fully functional 64mb sram using dual/tripe gate oxide process," in *2009 Symposium on VLSI Technology*. IEEE, 2009, pp. 210–211.

[76] A. Shilov, "JEDEC Publishes HBM2 Specification as Samsung Begins Mass Production of Chips." [Online]. Available: https://www.anandtech.com/show/9969/jedec-publishes-hbm2-specification

[77] NVIDIA, "NVIDIA A100 Tensor Core GPU Architecture." [Online]. Available: https://resources.nvidia.com/en-us-genomics-ep/ampere-architecture-white-paper?xs=169656#page=1

[78] A. Stillmaker and B. Baas, "Scaling equations for the accurate prediction of cmos device performance from 180 nm to 7 nm," *Integration*, vol. 58, pp. 74–81, 2017.

[79] M.-C. Hsieh, Y.-C. Liao, Y.-W. Chin, C.-H. Lien, T.-S. Chang, Y.-D. Chih, S. Natarajan, M.-J. Tsai, Y.-C. King, and C. J. Lin, "Ultra high density 3d via rram in pure 28nm cmos process," in *2013 IEEE international electron devices meeting*. IEEE, 2013, pp. 10–3.

[80] Intel, "Intel Optane Memory." [Online]. Available: https://www.intel.com/content/www/us/en/products/details/memory-storage/optane-memory.html

[81] H. Xin, D. Lee, F. Hormozdiari, S. Yedkar, O. Mutlu, and C. Alkan, "Accelerating read mapping with fasthash," in *BMC genomics*, vol. 14. Springer, 2013, pp. 1–13.

[82] J. S. Kim, D. Senol Cali, H. Xin, D. Lee, S. Ghose, M. Alser, H. Hassan, O. Ergin, C. Alkan, and O. Mutlu, "Grim-filter: Fast seed location filtering in dna read mapping using processing-in-memory technologies," *BMC genomics*, vol. 19, pp. 23–40, 2018.

[83] R. Kaplan, L. Yavits, and R. Ginosar, "Rassa: resistive prealignment accelerator for approximate dna long read mapping," *IEEE Micro*, vol. 39, no. 4, pp. 44–54, 2018.

[84] D. Fujiki, A. Subramaniyan, T. Zhang, Y. Zeng, R. Das, D. Blaauw, and S. Narayanasamy, "Genax: A genome sequencing accelerator," in *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2018, pp. 69–82.

[85] M. Alser, O. Mutlu, and C. Alkan, "Magnet: understanding and improving the accuracy of genome pre-alignment filtering," *arXiv preprint arXiv:1707.01631*, 2017.

[86] M. Alser, H. Hassan, H. Xin, O. Ergin, O. Mutlu, and C. Alkan, "Gatekeeper: a new hardware architecture for accelerating pre-alignment in dna short read mapping," *Bioinformatics*, vol. 33, no. 21, pp. 3355–3363, 2017.

[87] H. Li, "Minimap2: pairwise alignment for nucleotide sequences," *Bioinformatics*, vol. 34, no. 18, pp. 3094–3100, 2018.

[88] J. Daily, "Parasail: Simd c library for global, semi-global, and local pairwise sequence alignments," *BMC bioinformatics*, vol. 17, pp. 1–11, 2016.

[89] A. M. Aji, L. Zhang, and W.-c. Feng, "Gpu-rmap: accelerating short-read mapping on graphics processors," in *2010 13th IEEE International Conference on Computational Science and Engineering*. IEEE, 2010, pp. 168–175.

[90] Y. Liu and B. Schmidt, "Gswabe: faster gpu-accelerated sequence alignment with optimal alignment retrieval for short dna sequences," *Concurrency and Computation: Practice and Experience*, vol. 27, no. 4, pp. 958–972, 2015.

[91] X. Fei, Z. Dan, L. Lina, M. Xin, and Z. Chunlei, “Fpgasw: accelerating large-scale smith–waterman sequence alignment application with backtracking on fpga linear systolic array,” *Interdisciplinary Sciences: Computational Life Sciences*, vol. 10, pp. 176–188, 2018.

[92] S. S. Banerjee, M. El-Hadedy, J. B. Lim, Z. T. Kalbarczyk, D. Chen, S. S. Lumetta, and R. K. Iyer, “Asap: Accelerated short-read alignment on programmable hardware,” *IEEE Transactions on Computers*, vol. 68, no. 3, pp. 331–346, 2018.

[93] R. Kaplan, L. Yavits, R. Ginosar, and U. Weiser, “A resistive cam processing-in-storage architecture for dna sequence alignment,” *IEEE Micro*, vol. 37, no. 4, pp. 20–28, 2017.

[94] W. Huangfu, S. Li, X. Hu, and Y. Xie, “Radar: A 3d-reram based dna alignment accelerator architecture,” in *Proceedings of the 55th Annual Design Automation Conference*, 2018, pp. 1–6.

[95] F. Zokaee, M. Zhang, and L. Jiang, “Finder: Accelerating fm-index-based exact pattern matching in genomic sequences through reram technology,” in *2019 28th International Conference on Parallel Architectures and Compilation Techniques (PACT)*. IEEE, 2019, pp. 284–295.

[96] F. Zokaee, H. R. Zarandi, and L. Jiang, “Aligner: A process-in-memory architecture for short read alignment in rerams,” *IEEE Computer Architecture Letters*, vol. 17, no. 2, pp. 237–240, 2018.