Received 1 February 2020; revised 7 March 2020; accepted 7 April 2020.
Date of publication 13 April 2020; date of current version 6 July 2020.

Digital Object Identifier 10.1109/JXCDC.2020.2987527

A DNA Read Alignment Accelerator Based on Computational RAM

ZAMSHED I. CHOWDHURY[1] (Graduate Student Member, IEEE), MASOUD ZABIHI[1] (Graduate Student Member, IEEE), S. KAREN KHATAMIFARD[1] (Associate Member, IEEE), ZHENGYANG ZHAO[1] (Member, IEEE), SALONIK RESCH[1] (Student Member, IEEE), MEISAM RAZAVIYAYN[2] (Member, IEEE), JIAN-PING WANG[1] (Fellow, IEEE), SACHIN S. SAPATNEKAR[1] (Fellow, IEEE), and ULYA R. KARPUZCU[1] (Member, IEEE)

[1]Department of Electrical and Computer Engineering, University of Minnesota, Minneapolis, MN 55455 USA
[2]University of Southern California, Los Angeles, CA 90007 USA

CORRESPONDING AUTHOR: Z. I. CHOWDHURY (chowh005@umn.edu)

This work was supported in part by NSF under Grant SPX-1725420.

ABSTRACT Recent years have witnessed an increasing interest in the processing-in-memory (PIM) paradigm in computing due to its promise to improve the performance through the reduction of energy-hungry and long-latency memory accesses. Joined with the explosion of data to be processed, produced in genomics—particularly genome sequencing—PIM has become a potential promising candidate for accelerating genomics applications since they do not scale up well in conventional von Neumann systems. In this article, we present an in-memory accelerator architecture for DNA read alignment. This architecture outperforms corresponding software implementation by >49X and >18 000X, in terms of throughput and energy efficiency, respectively, even under conservative assumptions.

INDEX TERMS Accelerator, BWA, computational RAM (CRAM), DNA, genome sequence, processing in memory (PIM), spin Hall effect magnetic tunnel junction (SHE-MTJ).

I. INTRODUCTION

THE evolution in different domains of science and technology, from social networking to astronomical observation, has brought us into the age of large-scale data where an abundance of data is available for analysis to aid in the investigation of a wide range of problems. This presents itself with a new kind of challenge for conventional computing since the applications that use large-scale data do not scale up well with such a model of computing. Such applications suffer from increased energy consumption and latency due to high overhead from data movement between physically separate compute logic and memory. Processing in memory (PIM), also known as compute-in-memory or CiM, is an exciting solution to this issue that fuses the capability to perform logic operations with the standard memory functionalities—effectively reducing the separation between the compute logic and memory. Moreover, a high degree of parallelism, in the form of a column or row parallelism in a 2-D array of memory cells, can boost the throughput of in-array or *in situ* logic operations. PIM presents an opportunity to achieve better performance, in terms of latency and energy consumption, in comparison with conventional computing substrates, such as CPU or GPU.

The technology of genome sequencing has improved rapidly over the last decade, generating an abundance of data for analysis in different domains of research and precision medicine, e.g., finding causes for diseases, such as cancer and Alzheimer's. However, applications that utilize these large-scale data suffer from low scalability associated with the classical von Neumann systems due to excessive data movement overhead. The presence of a high number of memory accesses and a high degree of parallelism available in these applications make them good candidates for PIM-based acceleration.

In this article, we propose an end-to-end PIM accelerator for genomics, BWA-CRAM, which is based on the spintronic computational RAM (CRAM) [1] as the PIM substrate. CRAM uses spin Hall effect magnetic tunnel junction (SHE-MTJ)-based memory cells. As computation directly takes place in the memory array, CRAM can perform many bitwise operations in parallel, at very low energy. BWA-CRAM represents a PIM accelerator for the Burrows–Wheeler transform (BWT)-based DNA short read alignment (BWA). The BWT-based DNA sequence alignment has become a standard critical tool for sifting through the abundance of sequence data, e.g., DNA, available from the next-generation genome sequencing platforms. We provide the architectural details of BWA-CRAM and characterize the performance in terms of throughput and energy efficiency of alignment. We also compare the performance against a state-of-the-art software implementation of

BWA and a PIM-based DNA sequence alignment accelerator that uses similar memory technology as BWA-CRAM. In a nutshell, our contributions in this article are as follows.

1) We present an accelerator architecture for an emerging and critical genomic application, which, by itself, represents a key first computational step for many different types of genomic analysis.
2) In doing so, we significantly reduce the total required memory footprint utilizing PIM features.
3) We showcase that even without altering an algorithm designed for a conventional system to better exploit the underlying PIM substrate, significant performance benefits can be achieved.

The rest of this article is organized as follows. Section II discusses the basics of the BWA sequence alignment algorithm and CRAM, with the architectural details of mapping of BWA in CRAM explained in Section III. Sections IV and V report the evaluation setup and the outcome of the performance characterization of the accelerator. Finally, Section VI qualitatively compares other similar approaches available in the literature, and Section VII concludes this article.

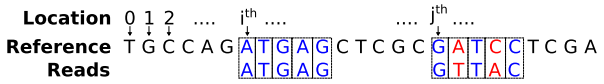

FIGURE 1. DNA sequence alignment.

II. BASICS

A. DNA SEQUENCE ALIGNMENT

A DNA sequence is a collection of four nucleotide base pairs (bp)–(A)denine, (C)ytosine, (G)uanine, and (T)hymine. The sequences of bases are recognized by a sequencing machine that encodes the sequences using a string of characters from the alphabet {A, C, G, T}. DNA sequence alignment is the problem of finding out the location of the highest degree of similarity between a very long sequence—a reference—and an order of magnitude shorter sequence–a read. A typical length of a short read is 100 bp, while a reference sequence can be in the order of billions of bp. Fig. 1 shows two reads aligned with an example reference sequence at two locations. The first read aligned at the *i*th location of the reference is an exact alignment since all bps in the reference and the read are the same. This is not the case for the second read since there are mismatches between the read and the reference (when aligned at the *j*th location of the reference), known as inexact alignment. Finding inexact alignment is similar, in the context of the algorithm, to finding exact alignment, however more expensive in terms of required computation.

B. BWT-BASED ALIGNMENT (BWA)[1]

BWT is a data compression technique [2], later adopted for fast alignment of short DNA reads [3]. The first step in BWA is to generate a BWT of the reference, which we will refer to as BWT throughout the rest of this article. Fig. 2 shows the generation of the BWT of an example string. The string, i.e., a reference sequence, in this case, is assumed to be terminated with the character "$" where no character in the reference is lexicographically smaller than "$." As shown in Fig. 2, all possible cyclic rotations of the reference sequence are created and then lexicographically sorted based on the first character—residing in the column labeled by F(irst) in the figure. The L(ast) column is the BWT of the reference sequence, and stored as the reference—residing in

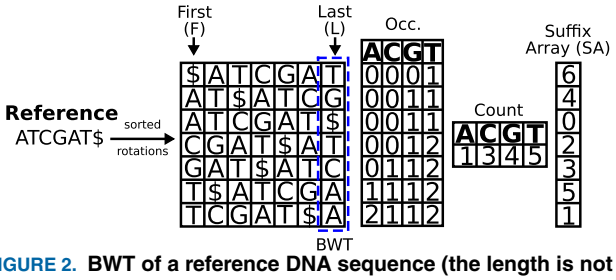

FIGURE 2. BWT of a reference DNA sequence (the length is not up-to-scale to ease illustration).

the column labeled by L(ast) in the figure. Only BWT is stored in memory.

To perform alignment of short reads, i.e., read sequences, some additional data structures are required. These include the Occurrence Table (Occ) that records, at each index in BWT, the number of occurrences of each character in the alphabet of the reference until that index. Fig. 2 illustrates how Occ keeps track of all occurrences of each character in the alphabet. The second data structure is relatively much smaller than Occ: Count that stores the number of lexicographically smaller characters in the F(irst) column, for each character in the alphabet. Finally, the suffix array (SA) stores the occurrence (index) of suffixes of all characters in the order they appear in the F(irst) column. Typically, an SA is constructed by generating all suffixes of a sequence, i.e., DNA reference, before sorting it lexicographically and storing the sorted suffix indices. As a result, an index in F refers to the same index in SA. The suffixes of the reference in Fig. 2 are [0] "ATCGAT," [1] "TCGAT," [2] "CGAT," [3] "GAT," [4] "AT," [5] "T," and [6]"$." Sorting the suffixes in lexicographical order of the first characters also sorts the corresponding indexes: {6, 4, 0, 2, 3, 5, 1} that is the SA of the reference sequence.

Fig. 3 shows an example of how BWA works. BWA searches whether a DNA read is present in the original reference, by considering one character of the read at a time in reverse order. In the example, we start with the last character, A, of the read CGA, and try to find it in BWT that contains two A's. The next step is finding the indices of these two A's in F. To this end, we consult the corresponding entries of Occ and Count tables. From the definition of these tables and BWT, it follows that the index in F simply is the sum of the entry (for each A in BWT, as readout) from Occ and the corresponding entry (for A) from Count. This renders $1 + 1 = 2$ for the first A and $2 + 1 = 3$ for the second A, respectively. As we start indices of F from 0, we then subtract a 1 from these values, arriving at 1 for the first A and 2 for the second A. Then, we repeat this procedure for the next character in sequence, i.e., G, but this time we limit our search to the BWT entries that reside at F indices 1 and 2. Recall that as F and BWT form the first and last columns of the same square matrix, F indices demarcate row indices in BWT, where we perform the character matching at each step. BWT entries at row indices 1 and 2 are G and \$. Hence, this time there is only one match in BWT, G—with Occ entry 1 and Count entry 4, which renders an F index of $1+4-1 = 4$ for the next search. Finally, search for the character, C, is confined to BWT row index 4,

[1]Here, we sketch the basic mechanics of this commonly used algorithm and refer the interested reader to numerous BWA references from the literature for more detailed algorithmic discussion.

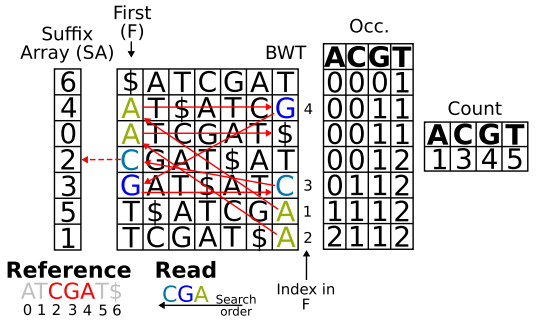

FIGURE 3. **BWA alignment of DNA read.**

where a C resides. This C indicates a match and resides at index (row) $1+3-1=3$ of F, following the same procedure as earlier. From the definition of F, BWT, and SA, it follows that the row of SA demarcated by this index of F, i.e., 3, holds the starting location (index) of the read within the original reference, i.e., 2, which, as depicted in Fig. 3, was indeed the case.

To summarize, the alignment of a character is dependent on L(ast), i.e., BWT, to F(irst) mapping. A character being aligned is first searched in BWT, which identifies a range for that character in BWT. This range (bound by low and high indices) corresponds to ranks, i.e., order, of that character that is the same in both BWT and F(irst). Once the ranks are known, the corresponding indices in F(irst) are computed. Each such range in F(irst) refers to a range of indices in BWT for the alignment of the next character (in reverse order) in the read.

C. MEMORY REQUIREMENT

Among the data structures required to be stored, in addition to the BWT, the Occ table is the largest with the number of entries equal to the length of BWT and each entry storing four multibit numbers. To reduce the required memory size, instead of the entire data structure, we can keep a reduced table where every ith entry is stored—reducing the required size by a factor of i. This does not compromise the correctness. Since not all Occ entries are stored, the rank computation during the search process can be performed with the help of additional character matching operations, trading storage for computation. Fig. 4 shows an example for $i = 5$, where the rank of a character A is needed, but the corresponding Occ entry is not stored (marked in red). In this case, we can sum the number of A's, between that index and the index corresponding to the nearest stored Occ entry (i.e., 2) together with the respective stored Occ entry (i.e., 253) to derive the rank of that particular A. To reduce computational complexity, the values of Count table entries are added to sampled Occ table entries, which just needs to be performed one time for the whole reference.

Algorithm 1 Read Alignment

1: $idx\_l = 0, idx\_h = len(BWT) - 1$
2: **for** all characters Ch in the read **do**
3: $idx\_l = interval(Occ[idx\_l/i], Ch, idx\_l)$
4: $idx\_h = interval(Occ[idx\_h/i], Ch, idx\_h)$
5: **if** $idx\_l > idx\_h$ **then**
6: No alignment found
7: **end if**
8: **end for**

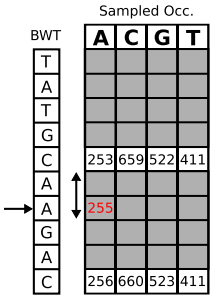

FIGURE 4. **Rank calculation with sampled Occ.**

D. BWA ALGORITHM: Putting It All Together

BWA is comprised of two stages: 1) alignment of a read and 2) accessing SA when the alignment is done. In the following, we provide the pseudocode that formally summarizes the procedure and suits well to the column-parallel CRAM implementation (as detailed in Section III). Algorithm 1 captures the steps involved in the first stage. For all characters in a read, in reverse order, the characters are aligned, and a set of high (idx_h) and low (idx_l) indices is computed (by interval function) for each character. This procedure continues until all characters are aligned. No alignment is found if low index becomes greater than the high index.

Algorithm 2 covers the interval function that has three inputs: nearest sampled Occ table entry, character to be aligned, and the BWT index (low/high), over which the character is to be aligned. The rank of input character is computed, as explained in Section II-C, by counting the number of times the input character is found between the input BWT index and the BWT index corresponding to the input Occ entry. The COMPARISON operation (line 3) performs character comparison and outputs 1 upon a match. *interval()* returns the computed index (through addition of number of character matches with the corresponding Occ table entry), which is used as the (low/high) index for alignment of the next character in the respective read. Recall that, Occ is augmented with Count entries—no further addition is needed. Here, i is the sampling factor for the reduced Occ table. COMPARISON is essentially a bitwise operation (as each character is encoded in multiple bits), which makes the ADD (line 7) a bitwise operation, as well.

In the second stage of the algorithm, after all characters from a read are aligned through interval function ($idx_l = idx_h$), the location of alignment for the respective read is found by accessing idx_l entry in SA.

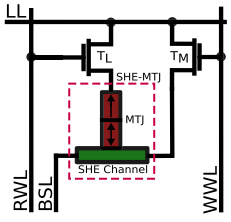

FIGURE 5. **CRAM cell architecture.**

E. CRAM BASICS

CRAM, being a PIM architecture, supports logic operations on top of standard memory operations (read and write). Fig. 5 illustrates a CRAM cell. At the heart of a CRAM cell lies an MTJ, placed on top of heavy metal (SHE) channel, hence SHE-MTJ. Each MTJ has two layers of ferromagnets; orientation—either Parallel (P) or Antiparallel (AP)—of the

magnetization of the layer adjacent to the fixed resistance SHE channel, i.e., the free layer, can be controlled by conducting a current through the SHE channel—greater than a threshold value (switching current). The other layer has fixed magnetization, and the relative orientation of the magnetization in the fixed and free layers brings a SHE-MTJ to high (P, AP) and low (P, P) resistance states, which is used to encode logics 1 and 0, respectively. Each SHE-MTJ in a CRAM cell is connected to an additional wire called the logic line or LL, through two transistors (T_L and T_M)—controlled by read and write word lines (RWL and WWL), respectively.

Algorithm 2 *interval* Function

1: $num\_match = 0$
2: **for** all characters *char* in $BWT[idx, idx/i]$ **do**
3: **if** $COMPARISON(char, Ch) == 1$ **then**
4: $num\_match+ = 1$
5: **end if**
6: **end for**
7: return $ADD(num\_match + Occ[idx/i])$

TABLE 1. Two-input NAND truth table.

Input Cell 1	Input Cell 2	Output Cell	$I_{Out} = I_1 + I_2$	
0 (R_{low})	0 (R_{low})	1	I_{00}	$> I_{crit}$
0 (R_{low})	1 (R_{high})	1	I_{01}	$> I_{crit}$
1 (R_{high})	0 (R_{low})	1	$I_{10} = I_{01}$	$> I_{crit}$
1 (R_{high})	1 (R_{high})	0	I_{11}	$< I_{crit}$

1) MEMORY MODE

In the memory mode, the LL is connected to a voltage source. During the read operation, T_L is ON that connects the MTJ and part of the SHE channel to the LL. The voltage applied between LL and bit select line (BSL) conducts a current through MTJ, and the resistance level of SHE-MTJ, i.e., stored data, is sensed. For write, T_M is ON, and a current, result of the voltage between LL and BSL, is conducted through the SHE channel to change the orientation of the magnetization of the free layer. The direction of the current determines P to AP or AP to P switching. When both transistors are OFF, the data in SHE-MTJ is retained.

2) LOGIC MODE

Through LL, CRAM cells are connected as inputs and outputs of logic gates. While there is no restriction on which cells can be input or output, it is required that input and output cells connect to opposite types of BSL available: Even (EBSL) or Odd (OBSL). MTJ(SHE channel) in each CRAM cell participating as input(output) to a logic gate is connected to LL when RWL(WWL) is logic 1. Fig. 6 shows an example logic gate formation where input cells 1 and 2 are connected to the output cell through LL. A voltage applied, between EBSL and OBSL, conducts a current through the MTJs in the input cell and SHE channels in the output cell. The magnitude of the current depends on the resistance levels of the input cells, i.e., stored data, which might change the resistance state of the output cell if greater than the switching current—effectively achieving a logic operation. Fig. 6(b) shows the equivalent circuit of this configuration. The value and polarity of V_{gate}, along with the data in the output cell prior to the logic operation, i.e., PRESET, determine the type of logic operation.

Table 1 illustrates the truth table of a NAND gate with corresponding current values through input cells: I_1 and I_2.

TABLE 2. Logic gates to implement XOR in CRAM.

In_0	In_0	S_1= NOR(In_0,In_1)	S_2= COPY(S_1)	S_3= COPY(S_1)	Out= TH(In_0,In_1,S_2,S_3)
0	0	1	1	1	0
0	1	0	0	0	1
1	0	0	0	0	1
1	1	0	0	0	0

The output cell is PRESET to logic 0, i.e., low resistance. The current through output cell, I_{Out}, changes the resistance level of output cell in all input cases as I_{Out} is greater than the critical (switching) current (I_{crit}), except for the case when both input cells store logic 1, i.e., high resistance.

3) COMPLEX LOGIC OPERATIONS

CRAM is Boolean complete and can implement complex logic operations, such as Exclusive-OR (XOR) as well. XOR is particularly important in character matching operations. This is achieved by using NOR, COPY, and TH (reshold) gates. NOR and COPY gates are implemented in a similar approach explained above. The TH gate is a four-input gate that outputs a logic 1 when >2 inputs are logic 0 (shown in Table 2).

Arithmetic operations, such as 1-bit addition, can be implemented using multiple gates, one after another. For example, a full adder with two 1-bit inputs, A and B, and a 1-bit carry input, C_{in}, are fed through the following steps to generate sum output S_o and carry output C_{out}.

Step 1: $C_{\text{out}} = \text{MAJ}\ (A, B, C_{\text{in}})$.
Step 2: $S_1 = \text{INV}\ (C_{\text{out}})$.
Step 3: $S_2 = \text{INV}\ (C_{\text{out}})$.
Step 4: $S_o = \text{MAJ}\ (A, B, C_{\text{in}}, S_1, S_2)$.

Here, INV and MAJ correspond to inverter and majority gates. For multibit additions, C_{out} from 1-bit addition at the previous bit position is used as C_{in}.

4) PARALLELISM

BSL spans across all rows, while RWL and WWL span across all columns of CRAM, making it possible to perform same logic operations simultaneously on all (or a subset of) columns in CRAM.

III. HIGH-LEVEL ARCHITECTURE

From a high-level of abstraction, BWA-CRAM is comprised of a number of functional blocks. Fig. 7(a) illustrates the overview of the architecture. The core task of aligning reads through BWT is performed by processing elements (PEs), whereas SA vectors and sampled SA are used to find the location of alignment once the alignment is complete. A global controller schedules and orchestrates all operations in BWA-CRAM. All data necessary for alignment, e.g., BWT, Occ, and *SSA*, are generated one time for a given reference database on a conventional von Neumann system and stored in the BWA-CRAM. The cost of such is amortized over the alignment of millions of reads against that reference database.

A. PROCESSING ELEMENT

Each PE stores part of the BWT and corresponding entries in the sampled Occ, in column-major order. PEs are designed from a collection of CRAM tiles, i.e., collection of CRAM cells arranged in 2-D with local controller and peripheral to perform memory and logic operations. The majority of the tiles store BWT, while others store the sampled Occ entries. Adjacent tiles are columnwise connected through transistors to perform computation across tiles. Each PE has its own controller that controls the tiles. Fig. 7(b) shows the organization of tiles in a PE. The layout of the BWT in a PE

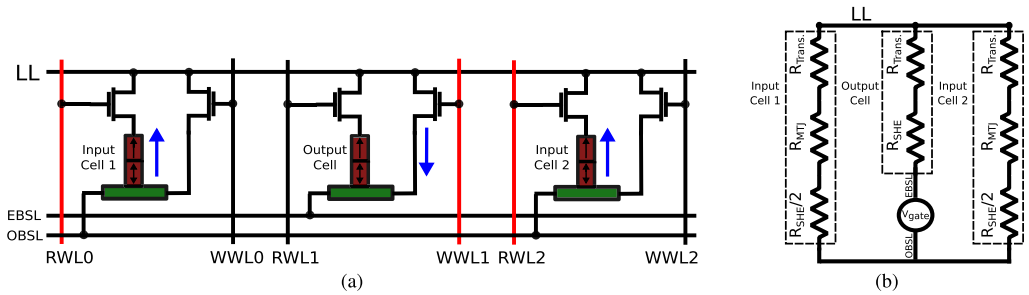

FIGURE 6. Bitwise logic in CRAM (transposed view). (a) Cells connected to perform logic operation. (b) Equivalent circuit.

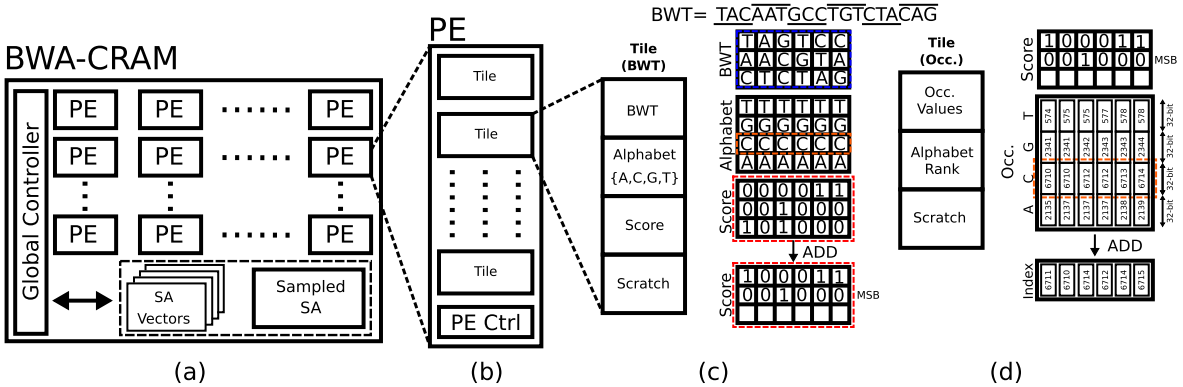

FIGURE 7. High-level architecture of BWA-CRAM. (a) Functional blocks. (b) Architecture of PE. (c) Data layout and BWT search in PE. (d) Index computation in PE.

is captured in Fig. 7(c). Characters in BWT are represented using 2 bits each. Each tile stores part of the BWT stored by the entire PE with a few rows dedicated to storing of the alphabet {A, C, G, T}. There are a number of rows allocated, in each tile, for use in computation: Score (to store the number of character matches) and scratch (for intermediate steps of computation). The purpose of dividing the stored BWT, in a PE, over a number of tiles is to utilize the tile-level parallelism as these tiles can perform computation in parallel. Fig. 7(c) illustrates how a BWT is segmented and stored in such a tile where (over)underline indicates segments stored in different columns.

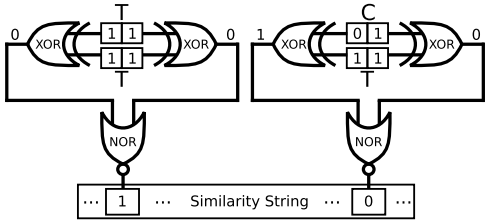

FIGURE 8. Bitwise character comparison.

PE executes the interval function in Algorithm 2, which is dependent on character comparison and addition, both bitwise operations. During the first stage of alignment, character comparison is performed between a character from the stored alphabet (which corresponds to the queried character from the read) and all BWT characters (by bitwise XOR) in column(s). The result of the comparison is stored in Score designated rows in respective column(s). Fig. 8 illustrates the bitwise matching of the characters. The similarity string, in a column, captures the bits in Score designated rows that indicate how many character matches are there in that particular column for a specific character being aligned. Next, bitwise additions over the similarity string bits are performed in respective column(s) to compute the binary representation of the number of matches, i.e., population count. In this example illustration, a character *C* is being aligned; therefore, *C* from the alphabet is compared against all BWT characters in column(s). The output of the comparison is stored and bitwise added to produce the number of matches in respective columns. The output of this addition is the rank for the character being aligned.

The tiles storing the sampled Occ in a PE corresponds to the BWT stored in that particular PE. Specifically, all four numbers in a column (in Fig. 7(d), for four characters in the alphabet, each 32-bit) correspond to an entry of the sampled Occ associated with the index of the first BWT character stored in that particular column. Fig. 7(d) shows such a tile storing the entries of a sampled Occ where the full Occ is sampled at every third BWT character, in this example. Now, to compute the index for the character being aligned—*C*, the number stored in Score, i.e., rank, and the Occ number corresponding to character *C* are bitwise added, which outputs the index for that character.

When a character is scheduled for alignment to a PE, the input (high or low) index value to interval function is converted to a column index for all tiles in that PE. That is, only one column performs the logic operation in each PE during alignment. However, the columns in all tiles are capable of performing logic operations in parallel as well; hence, the optimization opportunity is there to schedule multiple index computations to a single PE simultaneously.

B. ACCESSING SA

Storing the entire SA requires memory that is proportional to the size of the reference. For a large reference, it becomes a problem in the context of practical system design. Memory requirement of storing the SA can be reduced by sampling the SA—sampled SA or SSA, at regular intervals along the original reference. Although SA sampling is not a new idea, the contribution lies in accessing the SSA since not all entries in the SA are stored. Let us take a look at the example

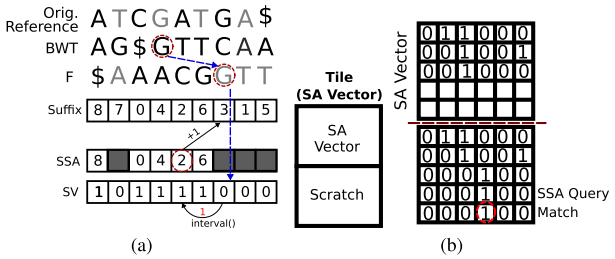

FIGURE 9. **(a) Accessing suffix for an index in SSA. (b) Search for an *F* index in SSA.**

illustrated in Fig. 9(a). It shows an example BWT and the corresponding F column along with the original reference. The suffixes are sampled at every even location $\{0, 2, \ldots\}$ of the original reference. Now, the lengths of the SSA and F are different, i.e., a suffix corresponding to an index on F might not be stored in SSA. To be able to access such suffixes, a suffix vector (SV) is stored, equal in length to the BWT (or F), where each bit indicates whether the suffix value at a particular location along SA is sampled in SSA. In case of an SSA access for a suffix that is not stored (corresponding to the second G on BWT, which is the same as the second G in F), as shown in Fig. 9(a), the suffix can be computed by executing the interval function in iterations. Each iteration returns the F index of the character that precedes it in the original reference. Since the SA is sampled at every kth location of the original reference, eventually, the interval function will return an F index whose corresponding suffix value is stored in SSA, with at most $k - 1$ iterations. In this example, after executing interval function 1 time, SV indicates that the corresponding suffix is stored in SSA. The suffix value is computed by bitwise addition between the number of times interval function is executed (=1) and the stored suffix value (=2).

The SV is stored in a collection of tiles. Each such tile stores a part of the entire SV, with two rows allocated for performing the check, i.e., bitwise AND operation, whether the suffix corresponding to an F index is stored in the SSA. The first allocated row is for storing index, and the second one stores the result of the bitwise comparison between the first row and another row in SV. Fig. 9(b) demonstrates that a query F index is written as logic 1 in the SSA Query row. After AND operation, the result is stored in the second allocated row, which is read out through standard read mechanism of CRAM. Similar to the tiles in PE, tiles storing SV are also capable of performing column-parallel logic operations, which is utilized to perform multiple index checks at the same time. This optimization, as well, directly follows from the basic definitions and mathematical characteristics of the algorithm.

C. GLOBAL CONTROLLER

The global controller is responsible for scheduling the characters to PE for alignment and performing the index check in SV and access SA—once the alignment is complete.

1) RUNTIME SCHEDULING

There are two invocations of interval computation for each character during alignment, which are scheduled to at most two PEs at the same time—leaving a high number of PEs idle even though each PE can run in parallel. This hints at the opportunity to schedule multiple characters to PEs, i.e., multiple interval computations at the same time. For this purpose, the global controller hosts a runtime scheduler that schedules multiple characters to PEs simultaneously. Since the alignment of each read is sequential, i.e., one character after another, this translates into scheduling multiple characters from multiple reads. This involves storing multiple reads concurrently and dynamically evaluating which characters can be scheduled together. A straightforward implementation would be to store a large number of reads so that the probability of finding and scheduling a minimum number of characters at the same time is high enough. This comes with a memory overhead since some additional information, for each read, is required to be stored, e.g., next character to schedule, and high and low indices from the last interval computation.

TABLE 3. **Technology Parameters.**

Parameters	Value
MTJ Type	Interfacial PMTJ
MTJ Diameter (nm)	10
TMR (%)	100
RA Product ($\Omega\mu m^2$)	20
Critical Current I_{crit} (μA)	3.0 (SHE Channel)
Switching Latency (ns)	1
R_P, R_{AP} ($K\Omega$)	253.97, 507.94
R_{SHE} ($K\Omega$)	64
$R_{Trans.}$ ($K\Omega$)	1
V_{INV} (V), V_{COPY} (V)	1.07-1.83
V_{NOR} (V)	0.64-0.77
V_{AND} (V)	0.77–1.02
V_{MAJ3} (V)	0.55-0.62
V_{MAJ5} (V)	0.42-0.45
V_{THR} (V)	0.44-0.47

2) INDEX COMPARISON IN SV

Although intermediate interval computations during index comparison in SV are scheduled to PEs, this phase of BWA does not overlap with the alignment stage, for simplicity of design. A more optimized controller can interleave interval computations from both stages. For each SSA access, a count value is required to be updated each time an interval computation is scheduled for a PE. This count operation is also executed using a CRAM tile that supports multiple count operations at the same time through the addition of a 32-bit value with logic 1.

D. SYSTEM INTERFACE

The interface between BWA-CRAM and host is modeled after SpinPM [4]. The programming interface provides abstraction between host and BWA-CRAM. Being an accelerator, BWA-CRAM does not have access to the virtual memory space. The interface is similar to the loosely coupled CPU–GPU interface of modern platforms.

IV. EVALUATION SETUP

A. SIMULATION

We evaluate the design by an in-house CRAM simulator that incorporates all low-level details of the system, e.g., energy and latency values of the logic operations in BWA-CRAM, spatiotemporal scheduling. This tool simulates the design at logic operation granularity—in order to capture the throughput performance and energy consumption of BWA-CRAM with the technology parameters listed in Table 3. I_{crit} refers to the threshold current, through the SHE channel, for the MTJ to the switch resistance state. The peripheral overheads are

modeled with NVSIM [5] to extract the row decoder, mux, precharge, and sense amplifier-induced energy and latency overheads considering parasitics. All peripheral overheads and the access transistors in each memory cell are modeled at 22-nm (HP) PTM [6].

B. DATA SET

Real human genome [7] is used as the reference, and a set of 10 million reads [8] is used as the DNA reads for BWA-CRAM. The reference genome is 3×10^9 bp in length, while each read is 100 bp long.

C. PE MODELING

The tiles in a PE are assumed to be 128×128 in dimensions. An array of transistors between adjacent tiles connects the corresponding columns during computation, when required. The overhead for these transistors is considered in the simulation. Each PE stores 512×128 characters of BWT reference in 16 tiles. There are two additional tiles in each PE for storing the sampled Occ that is sampled at every 512 BWT characters.

D. SA STORAGE

The entire SA is sampled at every 32^{nd} location from the beginning of the reference (SSA) and stored in the memory. The resultant memory size is ∼358 MB. To store the bit vector that represents the presence of a particular suffix in the SSA, a collection of 128×128 tiles are used. Each tile stores 126 vectors, each 128-bit long. The remaining rows in each tile are used for bitwise AND operation to check whether a particular suffix is stored.

E. RUNTIME SCHEDULER

The scheduler assumes a default dispatch rate of 1000 characters, i.e., 1000 read sequences, simultaneously. This is a conservative assumption in that it represents ∼4.37% utilization of the available PEs.

F. DESIGN SIZING

The entire design, with the selected data set, requires 45 777 PE. Considering all memory overheads, i.e., all tiles in all PEs and storage for SSA, the total memory footprint reaches ∼2.3 GB.

G. BASELINES FOR COMPARISON

For the purpose of performance comparison, a BWA-based DNA sequence alignment software, soap3-dp [9], is considered. The soap3-dp represents the state-of-the-art highly optimized GPU implementation. A Tesla K40 GPU is used for running the soap3-dp without, for a fair comparison, allowing any mismatch during alignment with seeding. The seeding algorithm helps pruning the alignment space and favors the GPU baseline. To demonstrate the performance improvement achieved by BWA-CRAM, we also consider AlignS [10]—a PIM BWA DNA sequence alignment accelerator that uses spin-orbit torque (SOT)-MRAM as the memory cell technology.

V. EVALUATION

The characterization is in terms of two metrics: 1) throughput, in K(ilo)Reads/s that represents the rate at which the DNA reads are aligned and 2) energy efficiency, in KReads/s/W, which represents how many DNA reads are aligned by BWA-CRAM per unit of energy. The baselines for comparison are also evaluated in terms of these metrics.

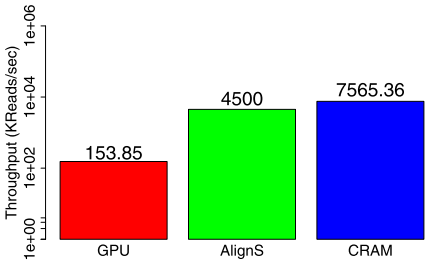

FIGURE 10. Throughput comparison of BWA-CRAM (log scale).

A. PERFORMANCE

The throughput performance of BWA-CRAM is shown in Fig. 10. Although GPU, running soap3-dp, has a fairly high alignment throughput, it suffers from large data movement overhead between GPU cores and the global DRAM, which is evident in the throughput value that is much lower than the PIM architectures—AlignS and BWA-CRAM. Being a true PIM that eliminates intermediate data movements during in-memory computation, BWA-CRAM outperforms both GPU and AlignS baselines by 49.17X and 1.68X, respectively. The relatively modest improvement in comparison to AlignS is due to the fact that BWA-CRAM performs more in-memory computations than AlignS. For instance, AlignS uses CMOS peripheral circuitry for some operations, e.g., counting the number of character match, which is performed in-memory by BWA-CRAM. Also, there is a computational overhead to search through the SSA for an index, which contributes toward this relatively smaller throughput improvement.

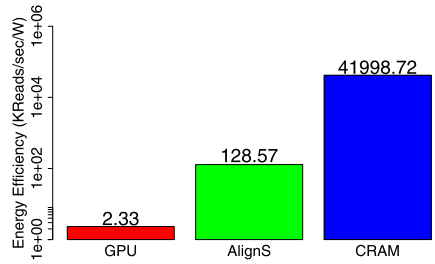

FIGURE 11. Energy efficiency of BWA-CRAM (log scale).

The energy efficiency of the baselines and BWA-CRAM is illustrated in Fig. 11. It is no surprise that BWA-CRAM outweighs the GPU baseline by 18025.2X due to the elimination of a significant number of energy-hungry and long-latency memory accesses. Interestingly, compared with AlignS, BWA-CRAM is ∼327X more energy efficient although both AlignS and BWA-CRAM use similar spintronic memory cell technology. This improvement in energy efficiency is the result of not using the peripheral circuitry to perform in-memory computing, unlike AlignS.

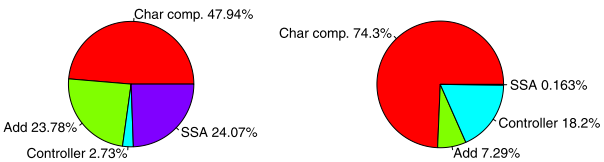

FIGURE 12. Breakdown of (a) latency and (b) energy.

Fig. 12 shows the latency and energy breakdowns of the design. On the latency front, character comparison operations consume the most latency, while addition operations consume around half of that. The controller consumes the least amount

of latency. SSA access latency, including all intermediate computations, takes ∼25% of the total latency of the design. Due to the serialized access to SSA memory, it can become a bottleneck with a very large number of reads scheduled to BWA-CRAM simultaneously. Reducing this bottleneck is possible by careful spatiotemporal scheduling of interval computations during SA access so that it overlaps with the alignment of the next batch of reads, which is left as a future work. On the energy side as well, a similar pattern holds. The majority of the energy is consumed by the character comparison operations. The next highest energy-consuming component is the controller—taking up ∼20% of the energy due to CMOS-based implementation. Both latency and energy components corresponding to reading out of index from PE, after interval computation is complete, are less than 2%.

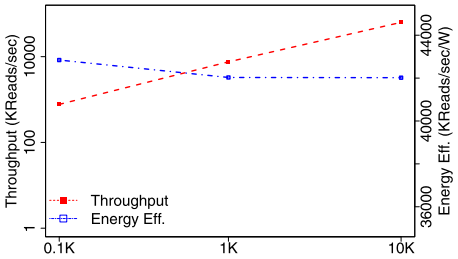

FIGURE 13. Impact of simultaneous scheduling of characters.

B. IMPACT OF RUNTIME SCHEDULER

The throughput performance reported here corresponds to a specific number of reads, conservatively assumed, scheduled to BWA-CRAM by the runtime scheduler. A more optimized scheduler would increase the throughput performance manyfold. Fig. 13 captures the impact on performance as more characters (reads) are scheduled simultaneously to BWA-CRAM (X-axis captures the increasing number of characters scheduled to BWA-CRAM). As the intuition suggests, the throughput performance increases almost linearly as more characters are scheduled simultaneously. As for energy efficiency, it tends to drop a little due to the sequential SA access by BWA-CRAM posing as a serial bottleneck.

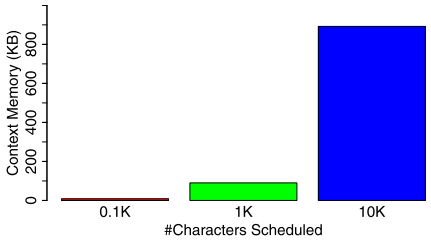

FIGURE 14. Required context storage in BWA-CRAM.

However, more reads handled by the scheduler translate into higher memory requirements for context storage. Context is the additional information related to a stored read, e.g., the last character from the read to be scheduled to BWA-CRAM, the current low, and high index values for a character. Fig. 14 shows how the context memory requirement scales as more characters are scheduled to BWA-CRAM. Here, we conservatively assume that during runtime, only 10% of the stored reads will have one character, each of which can be scheduled to BWA-CRAM simultaneously. Even with 10K characters scheduled simultaneously, i.e., 100K read contexts stored, the storage requirement is <1 MB.

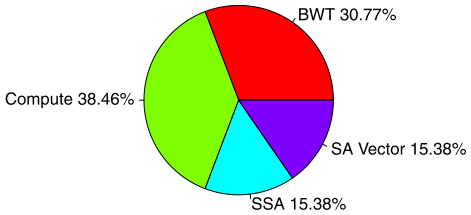

FIGURE 15. Memory footprint breakdown of BWA-CRAM.

C. DESIGN SIZE

The memory footprint used by BWA-CRAM is roughly 25% of that by AlignS [10]. Fig. 15 illustrates the distribution of the memory requirements for BWA-CRAM. Unsurprisingly, the majority of the memory is used to store the BWT and the corresponding overhead for in-memory computation, i.e., PE. The rest of the memory stores the SSA and the SA vector. In comparison to SA, this represents a reduction of ∼93%, at the expense of additional computation in BWA-CRAM. The size of the SSA can be reduced further, with more computation during the SA access stage of the alignment.

D. INEXACT ALIGNMENT

Although the current design of BWA-CRAM supports only exact alignment, it can be extended to include inexact alignment as well—at the expense of more computations. Specifically, it involves executing interval function recursively to generate SA intervals that match a given read with no more than allowed number of mismatches or gaps. The changes required in the design are, mainly, on the scheduling side.

As an example, we implemented such a BWA-CRAM design that allows up to two mismatches during alignment. This design exhibits >7X improvement in alignment throughput over SOAP3-dp while maintaining 1820.5X better energy efficiency. Recall that BWA-CRAM does not feature seeding, as opposed to the GPU baseline. If we augmented BWA-CRAM with a seeding algorithm to prune the alignment space, throughput improvement would be even higher.

VI. RELATED WORKS

BWT/A is the current state-of-the-art for DNA sequence alignment, with many software [11], [12], [13], [14] and hardware implementations such as MEDAL [15], a near-DRAM accelerator only for the seeding portion; FPGA design [16], [17] which consume orders of magnitude higher power than BWA-CRAM, including cloud-scale (FPGA) platforms of very high throughout [18]. Other exotic solutions [19], [20] target Smith–Waterman algorithm. Another PIM acceleration for BWA, significantly slower than BWA-CRAM, uses ReRAM [22]. Finally, [4] covers a basic CRAM design for large-scale string matching.

VII. CONCLUSION

PIM has emerged as an efficient computing paradigm for analyzing large scale data such as DNA sequence. In this article, we map BWT-based DNA sequence alignment (BWA) to SHE-MTJ-based CRAM substrate to accelerate DNA read alignment with low energy consumption. We show that CRAM-based architecture, BWA-CRAM, outperforms the GPU and PIM baselines, in terms of alignment throughput and energy efficiency, even under conservative assumptions. Furthermore, we show that the reduction of BWA data structures is also possible using in-memory computing features of CRAM.

REFERENCES

[1] J.-P. Wang *et al.*, "General structure for computational random access memory (CRAM)," U.S. Patent 9 224 447 B2, Dec. 29, 2015.

[2] Z. Arnavut and S. S. Magliveras, "Block sorting and compression," in *Proc. DCC Data Compress. Conf.*, 1997, pp. 181–190.

[3] H. Li and R. Durbin, "Fast and accurate short read alignment with burrows-wheeler transform," *Bioinformatics*, vol. 25, no. 14, pp. 1754–1760, Jul. 2009.

[4] Z. I. Chowdhury *et al.*, "Spintronic in-memory pattern matching," *IEEE J. Explor. Solid-State Computat. Devices Circuits*, vol. 5, no. 2, pp. 206–214, Dec. 2019.

[5] X. Dong, C. Xu, Y. Xie, and N. P. Jouppi, "NVSim: A circuit-level performance, energy, and area model for emerging nonvolatile memory," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 31, no. 7, pp. 994–1007, Jul. 2012.

[6] *Predictive Technology Model*. Accessed: Feb. 1, 2020. [Online]. Available: http://ptm.asu.edu/

[7] *1000 Genomes Project*. Accessed: Feb. 1, 2020. [Online]. Available: ftp://ftp-trace.ncbi.nih.gov/1000genomes/ftp/technical/reference/

[8] *NA12878*. Accessed: Feb. 1, 2020. [Online]. Available: ftp://ftp-trace.ncbi.nlm.nih.gov/giab/ftp/data/NA12878/

[9] R. Luo *et al.*, "SOAP3-dp: Fast, accurate and sensitive GPU-based short read aligner," *PLoS ONE*, vol. 8, no. 5, 2013, Art. no. e65632.

[10] S. Angizi, J. Sun, W. Zhang, and D. Fan, "Aligns: A processing-in-memory accelerator for dna short read alignment leveraging sot-mram," in *Proc. 56th ACM/IEEE Design Autom. Conf. (DAC)*, Jun. 2019, pp. 1–6.

[11] B. Langmead, C. Trapnell, M. Pop, and S. L. Salzberg, "Ultrafast and memory-efficient alignment of short DNA sequences to the human genome," *Genome Biol.*, vol. 10, no. 3, p. R25, 2009.

[12] B. Langmead and S. L. Salzberg, "Fast gapped-read alignment with bowtie 2," *Nature Methods*, vol. 9, no. 4, pp. 357–359, Apr. 2012.

[13] H. Li and R. Durbin, "Fast and accurate short read alignment with burrows-wheeler transform," *Bioinformatics*, vol. 25, no. 14, pp. 1754–1760, Jul. 2009.

[14] H. Li and R. Durbin, "Fast and accurate long-read alignment with Burrows–Wheeler transform," *Bioinformatics*, vol. 26, no. 5, pp. 589–595, Mar. 2010.

[15] 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 *Proc. 52nd Annu. IEEE/ACM Int. Symp. Microarchit.*, Oct. 2019, pp. 587–599.

[16] W. Vanderbauwhede and K. Benkrid, *High-Performance Computing Using FPGAs*. New York, NY, USA: Springer, 2013.

[17] G. Tan, C. Zhang, W. Tang, P. Zhang, and N. Sun, "Accelerating irregular computation in massive short reads mapping on FPGA co-processor," *IEEE Trans. Parallel Distrib. Syst.*, vol. 27, no. 5, pp. 1253–1264, May 2016.

[18] L. Wu *et al.*, "FPGA accelerated INDEL realignment in the cloud," in *Proc. IEEE Int. Symp. High Perform. Comput. Archit. (HPCA)*, Feb. 2019, pp. 277–290.

[19] A. Madhavan, T. Sherwood, and D. Strukov, "Race logic: A hardware acceleration for dynamic programming algorithms," in *Proc. ACM/IEEE 41st Int. Symp. Comput. Archit. (ISCA)*, Jun. 2014.

[20] 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.

[21] W. Huangfu, S. Li, X. Hu, and Y. Xie, "RADAR: A 3D-ReRAM based DNA alignment accelerator architecture," in *Proc. 55th ACM/ESDA/IEEE Design Autom. Conf. (DAC)*, Jun. 2018, pp. 1–6.

[22] F. Zokaee, H. R. Zarandi, and L. Jiang, "AligneR: A process-in-memory architecture for short read alignment in ReRAMs," *IEEE Comput. Archit. Lett.*, vol. 17, no. 2, pp. 237–240, Jul. 2018.