PIM-Aligner: A Processing-in-MRAM Platform for Biological Sequence Alignment

Shaahin Angizi*, Jiao Sun†, Wei Zhang† and Deliang Fan‡
*Department of Electrical and Computer Engineering, University of Central Florida, Orlando, FL 32816
†Department of Computer Science, University of Central Florida, Orlando, FL 32816
‡School of Electrical, Computer and Energy Engineering, Arizona State University, Tempe, AZ 85287
Email: angizi@knights.ucf.edu, wzhang.cs@ucf.edu, dfan@asu.edu

***Abstract*—In this paper, we propose a high-throughput and energy-efficient Processing-in-Memory accelerator (*PIM-Aligner*) to execute DNA short read alignment based on an optimized and hardware-friendly alignment algorithm. We first reconstruct the existing sequence alignment algorithm based on BWT and FM-index such that it can be fully implemented in PIM platforms. It supports exact alignment and also handles mismatches to reduce excessive backtracking. We then develop *PIM-Aligner* platform that transforms SOT-MRAM array to a potential computational memory to accelerate the reconstructed alignment-in-memory algorithm incurring a low cost on top of original SOT-MRAM chips (less than 10% of chip area). Accordingly, we present a local data partitioning, mapping, and pipeline technique to maximize the parallelism in multiple computational sub-array while doing the alignment task. The simulation results show that *PIM-Aligner* outperforms recent platforms based on dynamic programming with ∼3.1× higher throughput per Watt. Besides, *PIM-Aligner* improves the short read alignment throughput per Watt per mm^2 by ∼9× and 1.9× compared to FM-index-based ASIC and processing-in-ReRAM designs, respectively.**

I. Introduction

Advances in high-throughput sequencing technologies have enabled accurate and fast generation of large-scale genomic data for each individual, and is capable of measuring molecular activities in cells. Genomic analyses, including mRNA quantification, genetic variants detection, and differential gene expression, promise to help improve phenotype predictions and provide more accurate disease diagnostics [1]. The sequencing data generated from one patient sample consists of tens of millions of short DNA sequences (reads) that range from 50 to thousands *nt* in length. Most genomic pipelines rely on the alignment of sequencing reads with respect to the reference genome [2], which remains to be a time-consuming and technically difficult step. Specifically, the human reference genome is comprised of two twistings, paired strands and each strand carries approximately 3.2 billion nucleotide bases (A, T, C, G), and the bases on two strands follow the complementary base pairing rule: A-T and C-G [3]. Therefore, the DNA sequence alignment task is becoming to determine the read's likely point of origin on the 3.2 billion base pair (*bp*) reference genome. Although several sequence alignment algorithms have been developed in recent years, the continuously increasing volume of DNA sequencing data still calls for rapid and accurate aligners. Even the most efficient algorithm such as BWA [2] or Bowtie [4] using Burrows-Wheeler Transformation (BWT) require hours or days to align such large amount of data using powerful CPU/GPU-based systems.

Today's sequencing acceleration platforms including CPU, GPU [5], ASIC [6]–[8], and FPGA [9] are mostly based on the Von-Neumann architecture with separate computing and memory components connecting via buses and inevitably consume a large amount of energy in data movement between them. Besides, Processing-in-Memory (PIM) architectures, as a potentially viable way to solve the memory wall challenge, have been well explored for different applications that lead to remarkable savings in off-chip data communication energy and latency. The PIM platform has become even more intriguing when integrated with emerging Non-Volatile Memory (NVM) technologies, such as Resistive RAM (ReRAM) [3], [10]. The most recent ReRAM-based PIM solutions for short read alignment [10], [11] rely on Ternary Content-Addressable Memory (TCAM) arrays that unavoidably impose significant area and energy overheads to the system [3] due to associative processing dealing with Smith-Waterman (SW)-based algorithms that require many write operations and takes 75% of the ReRAM cells to store the intermediate data [12]. Alternatively, RADAR [10] and AligneR [3] present ReRAM-based PIM architectures that can directly map more efficient algorithms such as BLASTN and FM-index-based searches, respectively. In addition, Magnetic RAM (MRAM) is another promising high performance NVM paradigm, due to its ultra-low switching energy and compatibility with CMOS technology [13].

In this work, we propose a solid software-hardware alignment-in-memory solution to perform DNA sequence alignment efficiently. The main contributions of this work are listed below: (1) We reconstruct the existing sequence alignment algorithm based on BWT and FM-index such that it can be fully implemented in PIM platforms. It supports exact alignment and handles mismatches. (2) We design a reconfigurable PIM platform based on SOT-MRAM, *PIM-Aligner*. We develop a set of new microarchitectural and circuit-level schemes that make *PIM-Aligner* a massive data-parallel computational unit for short read alignment; (3) We propose a local data partitioning methodology, mapping, and pipeline technique to maximize the parallelism in multiple computational sub-arrays while doing the alignment task. (4) We extensively assess and compare *PIM-Aligner*'s performance and energy-efficiency with recent short read alignment accelerators based on GPU, ASIC, FPGA, processing-in-ReRAM, etc.

II. BWT-based Read Mapping Background

The BWT of a string is a reversible permutation of the characters in the string. Short read alignment algorithms (e.g., BWA [2] and Bowtie [4]) take all the advantages of BWT and index the large reference genome-*S* to do the read alignment efficiently. Exact alignment finds all occurrences of the m-bp short-read *R* in the n-bp reference genome-*S*. Fig. 1 gives an intuitive example of such alignment of a sample read-$R = CTA$ to a sample reference $S = TGCTA$\$ extracted from a gene, where \$ denotes the end of a sequence. BW

978-3-9819263-4-7/DATE20/©2020 EDAA

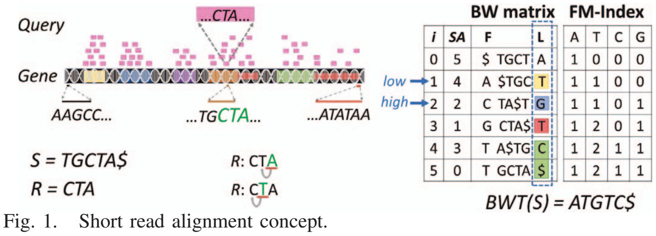


Fig. 1. Short read alignment concept.

matrix is constructed by circulating string *S* and then lexicographically sorting them. Thus, the Suffix Array (S_A) of a reference genome-*S* is a lexicographically-sorted array of the suffixes of *S*, where each suffix is represented by its position in *S*. In this way, BWT of the reference-*S* is given by the last column in the BW matrix, $BWT(S) = ATGTC\$$. The FM-Index is then built on top of BWT providing the occurrence information of each symbol in BWT. The S_A interval (*low*, *high*) covers a range of indices where the suffixes have the same prefix. Then a backward search of the matched positions in the reference genome-*S* is executed for each short read-*R* starting from the rightmost nucleotide (A in Fig. 1). During the backward search, the matched lower bound (*low*) and upper bound (*high*) in a S_A of the *S* for each nucleotide in *R* are determined based on FM-Index and count function [2]. Thus, the result of read searching is represented as an S_A interval. At the end of search, if $low < high$, *R* has found a match in *S*. Conversely, if *low* ≥ *high*, it has failed to find a match. Such alignment algorithm complexity is linearly proportional to the number of nucleotides in a read ($\mathcal{O}(m)$) in contrast to dynamic programming algorithms such as Smith-Waterman (SW) with $\mathcal{O}(nm)$ complexity [14]. Backtracking can simply extend the BWT technique to allow mismatches to support approximate alignment. In this approach, the DNA short read is permuted using edit operations (substitutions, insertions or deletions).

III. PIM-Aligner DNA Sequencing Algorithm

The DNA alignment algorithm consists of two stages: exact alignment and inexact alignment. For most sequencing data, up to ∼70% of short reads should be exactly aligned to the reference genome after stage one [9]. The remaining reads are then processed through the stage two. Most genome variations are relatively small. If we only allow exact match between short reads and the reference genome, the reads contain the genome variations from the sample cannot map to the reference. In addition, the genome variations (e.g., single nucleotide mutations) cannot be identified based on the exact alignment algorithm. Thus, such potential molecular signatures cannot be applied for disease phenotype prediction. In the following, we elaborate these two stage, respectively.

Exact Alignment-in-Memory Algorithm: Our alignment algorithm is developed based on BWT and FM-Index sequencing algorithm [2], but reconstructed to use particular in-memory functions that are parallelable in hardware. As the first step of such process, shown in Fig. 2, some important tables are needed to be pre-computed based on reference genome-*S*. However, it is just a one-step computation and only BWT, Marker Table (M_T), and S_A will be stored in the memory, which will consume ∼12GB of memory space. To enable fast memory access and parallel in-memory computing, these data has to be reconstructed and saved into different memory arrays, banks and chips. Such data reconstruction and mapping methodology will be discussed in Section V.

Algorithm 1 PIM-Aligner's Exact Match.

```
Require: : Pre-Compute and Data Mapping to PIM-Aligner: Partition pre-computed BWT,
    Marker Table ($M_T$) and Suffix Array ($S_A$) into PIM-Aligner chip.
    input: DNA Short Read-$R$
    output: positions of short read-$R$ in reference genome-$S$
    Step-1. Initialization:
 1: $low \leftarrow 0$, $high \leftarrow |S| - 1$
    Step-2. Backward Search:
 2: for $i := |R| - 1$ to 0 do
 3:     $low \leftarrow$ LFM$(M_T[\lfloor low/d \rfloor], R[i], low)$
 4:     $high \leftarrow$ LFM$(M_T[\lfloor high/d \rfloor], R[i], high)$
 5:     if $low \geq high$ then
 6:         break & return 0                         ▷ there is no exact alignment
    Step-3. Get matched positions from stored suffix array based on search result:
 7: for $j := low$ to $high - 1$ do
 8:     $positions \leftarrow$ MEM$(S_A[j])$        ▷ Read positions from Suffix Array memory
    Define procedure LFM:
 9: Procedure: LFM$(M_T, nt, id)$                  ▷ compute matched interval
10: $count\_match \leftarrow 0$
11: for $j := 0$ to $j < (id \bmod d)$ do          ▷ count number of $nt$ within the BWT region
12:     if XNOR_Match$(nt, BWT[id - (id \bmod d) + j]) == 1$ then
13:         $count\_match = count\_match + 1$
14: $marker \leftarrow$ MEM$(M_T[\lfloor id/d \rfloor], nt])$   ▷ Read Marker Table value
15: return IM-ADD$(marker, count\_match)$
16: end Procedure
```

In Fig. 2, $Count(nt)$ represents the number of nucleotides in the first column of BW matrix that are lexicographically smaller than the nucleotide-nt. It only contains 4 elements for DNA sequence computation. The Occurrence (Occ.) table, also called FM-index, is built upon the BWT, where each element-$Occ[i, nt]$ indicates the number of occurrences of nucleotide-nt in the BWT from position 0 to $i-1$. Due to its large size, it is sampled every d positions (bucket width) to construct another Sampled Occ-table. Thus, the table size is reduced by a factor of d. Then, M_T is constructed by element-wise addition of Sampled Occ-table with $Count(nt)$, which leads to the same size as Sampled Occ-Table. M_T contains the matched position of the nucleotides in BWT in the First Column and helps to efficiently retrieve the values of *low* and *high* in each iteration. As shown in Algorithm-1, the read searching function can be reconstructed through the proposed hardware-friendly $LFM(M_T, nt, id)$ procedure (line-9) performed on BWT, which computes the updated interval bound (either *low* or *high*) value from M_T with bucket width d and input index-id. Such procedure is iteratively used in every step of 'for' loop and *PIM-Aligner* will be especially developed to handle such computation-intensive load through performing comparison and addition of the current 'marker' value with the occurrence counting result of the needed nucleotides between checkpoint position and remaining positions in BWT.

In-exact Alignment-in-Memory Algorithm: Here, we propose to extend our alignment algorithm to handle inexact match during short read alignment as shown in Algorithm-2. Such inexact alignment-in-memory algorithm has a tolerance for number of mismatches between short read-*R* and reference genome-*S*. As an extension of exact sequence alignment, inexact matching searches for intervals-I that match *R* with no more than z differences. We can handle mismatch by recursively calculating the intervals that match $R[0, i]$ with no more than z differences on the condition that $R[i+1]$ matches{*low*,

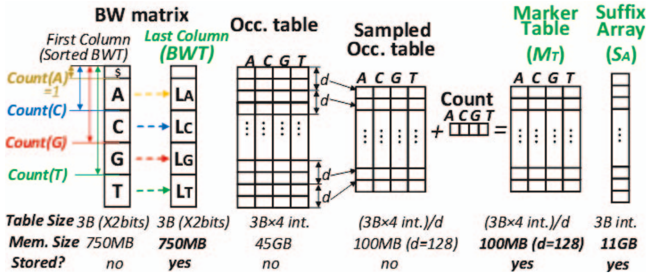

BW matrix
First Column (Sorted BWT)
Last Column (BWT)
Count(A) =1
Count(C)
Count(G)
Count(T)
$
A
C
G
T
LA
LC
LG
LT
Occ. table
A C G T
d
d
d
d
Sampled Occ. table
A C G T
Count
A C G T
+
=
Marker Table (MT)
A C G T
Suffix Array (SA)
Table Size 3B (X2bits) 3B (X2bits) 3B×4 int. (3B×4 int.)/d (3B×4 int.)/d 3B int.
Mem. Size 750MB 750MB 45GB 100MB (d=128) 100MB (d=128) 11GB
Stored? no yes no no yes yes

Fig. 2. Pre-computation needed in *PIM-Aligner*'s algorithm.

Algorithm 2 PIM-Aligner's Inexact Match.

```
Require: : Pre-Compute and Data Mapping to PIM-Aligner: Partition pre-computed BWT,
    Marker Table ($M_T$) and Suffix Array ($S_A$) into PIM-Aligner chip.
    input: DNA Short Read-$R$, $z$ mismatches allowed in the alignment.
    output: positions of short read-$R$ in reference genome-$S$ with up to $z$ mismatches.
    Step-1. Initialization:
 1: $low \leftarrow 0$, $high \leftarrow |S| - 1$
 2: return I= InexactRecursive($R, |R|, low, high, z$):
 3: for $i := |I| - 1$ to 0 do
 4:     $positions \leftarrow$ MEM($S_A[I[i]]$)
    Define procedure InexactRecursive :
 5: Procedure: InexactRecursive($R, i, low, high, z$)        ▷ $z$ is the number of mismatches
    allowed
 6: if $z < 0$ then
 7:     break & return 0
 8: if $i < 0$ then
 9:     return [low,high]
10: $I \leftarrow 0$
11: $I \leftarrow I \cup$ InexactRecursive($R, i-1, low, high, z-1$)                ▷ Insertion
12: for each $b \in \{A, C, G, T\}$ do
13:     $low \leftarrow$ LFM($M_T[\lfloor low/d \rfloor], R[i], low$)
14:     $high \leftarrow$ LFM($M_T[\lfloor high/d \rfloor], R[i], high$)
15:     if $low < high$ then
16:         $I \leftarrow I \cup$ InexactRecursive($R, i, low, high, z-1$)            ▷ Deletion
17:         if $b = R[i]$ then
18:             $I \cup$ InexactRecursive($R, i-1, low, high, z$)              ▷ Exact Match
19:         else
20:             $I \cup$ InexactRecursive($R, i-1, low, high, z-1$)          ▷ Inexact Match
21: return $I$
22: end Procedure
```

high$\}$. As long as there is still tolerance for differences up to current position i, we should consider all possible alignments when updating the intervals I. The intervals I for position i should take union for all intervals including intervals for match (line 16) and mismatch (line 18). At the end, we report all the target positions (line 4) in the reference genome that the short read can map to with no more than z mismatches. We can see Algorithm-2 still iteratively uses previously-proposed *LFM* function and can be readily accelerated by a PIM platform.

IV. PIM-Aligner Platform

A. Macro-architecture

We design and develop *PIM-Aligner* as an energy-efficient and high-performance accelerator on top of main memory architecture. To run the proposed DNA exact and inexact algorithms supporting backtracking with a hardware PIM platform, we use the block level accelerator architecture shown in Fig. 3. It includes BWT-based mapping based on iteratively-used *LFM* procedure and S_A and M_T query as the crucial operations. A Digital Processing Unit (DPU) is associated with the *PIM-Aligner* to control the entire process through different steps. The DPU takes the reference genome-S and number of mismatches-z as the inputs and accordingly adjusts the controller unit to govern timing and data flow of the alignment task. The index-*low* and index-*high* boundaries are initialized to the length of the reference DNA, 0 and N, respectively. As mentioned earlier, the possible locations of the suffixes candidate are indicated with the range of index-*low* and index-*high*. The backtracking is then performed in each iteration based on the alignment algorithms discussed in Section III. To implement the *LFM* procedure totally within memory, we exploits three functions, i.e. *MEM* (memory read), *XNOR_Match* (Comparison-XNOR2), and *IM_ADD* (Addition-add), as highlighted in Algorithm-1 and -2 and also Fig. 3. *MEM* function is to access data in the saved M_T or S_A based on the provided index. *XNOR_Match* is to conduct parallel in-memory XNOR2 logic to determine if current input-nt matches with BWT elements stored in the whole word-line. *IM_ADD* is to conduct 32-bit integer (index range) addition operation within memory to enable fast 'marker+count_match' computation without need to send to other computing units. To reduce the computation load by the down sampling scheme for reconstructing Occ. table *PIM-Aligner* exploits multiple *XNOR_Match* modules in parallel. Given a 512×256 memory sub-array, 128-bps could be compared to maximize the query speed for 2-bit DNA bases. This will be elaborated in Section V. To handle one and two mismatch alignment based on input-z, we exploit an additional control logic (in DPU) to perform bi-directional backtracking. For each allowed mismatch, DPU's registers store the state (i.e. symbol, *low* and *high*). *PIM-Aligner* then uses the states with the updated values of *high* and *low* pointers (after running *LFM* procedure) to control the backtracking based on Algorithm-2. The architecture is designed to locate all possible alignment hits for the given number of mismatches and then readily perform the insertion and deletion tasks without sending data to off-chip processing unit. The alignment results are then stored locally in memory.

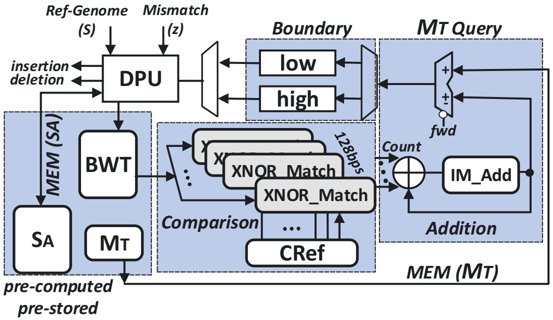


Fig. 3. The block level architecture of *PIM-Aligner* to support the alignment algorithms with backtracking.

B. Micro-architecture

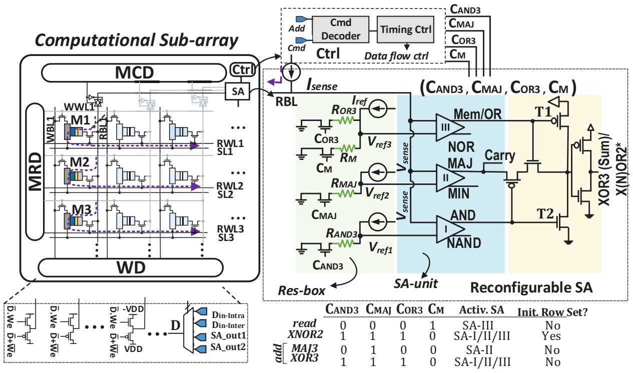
PIM-Aligner is designed based on typical main memory hierarchy. Each memory chip is divided into multiple memory banks that contains 2D sub-arrays of memory cells. The computational memory sub-arrays of *PIM-Aligner* based on SOT-MRAM is shown in Fig. 4a. The controller (Ctrl) unit can configure the sub-arrays to perform data-parallel intra-sub-array computations according to the physical address of operands within memory.

This architecture mainly consists of Write Driver (WD), Memory Row Decoder (MRD) , Memory Column Decoder (MCD), reconfigurable Sense Amplifier (SA), and can be adjusted by Ctrl unit to work in dual-mode that perform both memory write/read and bit-line computing. For each SOT-MRAM cell, the resistance of MTJ with parallel magnetization in both magnetic layers (data-'0') is lower than that of MTJ with anti-parallel magnetization (data-'1'). Each cell located in computational sub-arrays is associated with the Write Word Line (WWL), Read Word Line (RWL), Write Bit Line (WBL), Read Bit Line (RBL), and Source Line (SL). To *write* '1' (/'0') in a cell, e.g. in the cell of 1st row and 1st column (M1), the WD connected to WBL1 is set to positive (/negative) write voltage. This allows sufficient charge current flows from VDD to GND (/-VDD to GND), leading to MTJ resistance in High-R_{AP} (/Low-R_P). The main idea to perform memory and bit-line computing operations in *PIM-Aligner* is to select different thresholds (references) within the *Res-box* (see Fig. 4b) when sensing the selected memory cell(s). The proposed SA, as depicted in Fig. 4b, consists of three sub-SAs and totally four reference-resistance branches that can be selected by enable control bits (C_{AND3}, C_{MAJ}, C_{OR3}, C_M) by the sub-array's Ctrl to realize the memory read and computation schemes, as tabulated in the table in Fig. 4b. Such reconfigurable SA could implement memory read and one-threshold based logic functions only by activating one enable at a time e.g. by setting C_{AND3} to '1', 3-input AND/NAND logic can be readily implemented between operands located in the same bit-line. The computational sub-array of *PIM-Aligner* is optimized to perform *LFM* procedure and its three bulk bit-wise in-memory operations (*MEM*, *XNOR_Match*, *IM_ADD*) between the operands located in the same bit-line.

MEM: For memory *read*, a read current flows from the selected cell to ground, generating a sense voltage ($\mathrm{V_{sense}}$) at the input of SA-III, which is compared with memory mode reference voltage activated by C_M ($\mathrm{V_{sense,P}}<\mathrm{V_{ref,M}}<\mathrm{V_{sense,AP}}$). If the path resistance is higher (/lower) than R_M (memory reference resistance), i.e. R_{AP} ($/R_P$), then the SA produces High (/Low) voltage indicating logic '1' (/'0').

XNOR_Match: *PIM-Aligner*'s SA exploits a unique circuit design that allows single-cycle implementation of XOR3 in-memory logic. To realize XOR3 in-memory logic, every three bits stored in the identical column can be selected employing the MRD [15] and sensed simultaneously, as depicted in Fig. 4a. Then, the equivalent resistance of such parallel connected cells and their cascaded access transistors are compared with three programmable references by SA ($R_{AND3}, R_{MAJ}, R_{OR3}$). Through selecting these reference resistances simultaneously, the sub-SAs can perform basic 3-input in-memory Boolean functions (i.e. AND3,MAJ, OR3). The idea of voltage comparison between $\mathrm{V_{sense}}$ and $\mathrm{V_{ref}}$ to realize these functions is shown on Fig. 5a. After *SA-unit*, we used six control transistors to realize XOR3 function. Assuming one row in memory sub-array initialized to one, XNOR2 can be readily implemented in a single memory cycle out of XOR3 function. Therefore, every memory sub-array can potentially perform *XNOR_Match* function in Algorithm 1 and 2.

IM_ADD: *PIM-Aligner*'s sub-array can perform addition/subtraction (add/sub) operation quite efficiently. The carry-out of the full-adder can be directly produced by MAJ function (Carry in Fig. 4b) just by setting C_{MAJ} to '1' in a single memory cycle. Now, assume M1, M2, and M3 operands (Fig. 4a), the *PIM-Aligner* can generate Carry-MAJ and Sum-XNOR3 in-memory logics in a single memory cycle. The Ctrl's configuration for such add operation is tabulated in Fig. 4b. To validate the variation tolerance of the sensing circuit, we have performed Monte-Carlo simulation with 10000 trials. A $\sigma = 2\%$ variation is added to the Resistance-Area product ($\mathrm{RA_P}$), and a $\sigma = 5\%$ process variation is added on the Tunneling MagnetoResistive (TMR) of SOT-MRAM cells. The simulation result of $\mathrm{V_{sense}}$ distributions in Fig. 5b shows the sense margin for in-memory operations. We observe that sense margin gradually reduces when increasing the number of fan-ins. To avoid logic failure and guarantee the SA output's reliability, we have limited the number of sensed cells to three. In order to provide a larger sense margin for MAJ3 operation, we increased SOT-MRAM cell's t_{ox} from 1.5nm to 2nm leading to ∼45mV increase in the sense margin which considerably enhances the reliability.



	CAND3	CMAJ	COR3	CM	Activ. SA	Init. Row Set?
read	0	0	0	1	SA-III	No
XNOR2	1	1	1	0	SA-I/II/III	Yes
add MAJ3	0	1	0	0	SA-II	No
add XOR3	1	1	1	0	SA-I/II/III	No

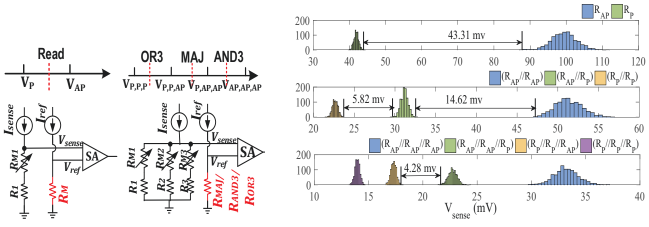
(a) (b)

Fig. 4. (a) The *PIM-Aligner*'s computational sub-array based on SOT-MRAM and (b) the proposed reconfigurable SA to realize in-memory functions.

V. Correlated and Localized Computation

Partitioning: The pre-computed tables (BWT, M_T, and S_A) require a large memory space, therefore, to fully leverage *PIM-Aligner*'s parallelism, and maximize alignment throughput, we come up with a partitioning, mapping and pipeline design. Given a BWT index range, the accessed memory region of M_T and BWT could be readily predicted and computation could be localized if we store such correlated region into the same memory sub-array. The correlated data partitioning and mapping methodology, as shown in Fig. 6, locally stores correlated regions of BWT and M_T vectors in the same memory sub-array to enable fully local computation (i.e. *XNOR_Match* and *IM_ADD* completely within the same sub-array without inter-bank/chip communication). To do it, each *PIM-Aligner*'s sub-array (512×256) is spilt into four zones to save four different data types, i.e. BWT, CRef, M_T, and reserved space for *IM_ADD* (Fig. 6a). First, 256 rows are filled with the corresponding BWT, where each row stores up to 128 bps (encoded by 2 bits). Besides, 4 nucleotide computational reference vectors (*CRef*) are initialized, in which each vector gives one type of nucleotide with vector size of number of bits in one word-line. CRef is designed to enable fully parallel match operation- *XNOR_Match*. Next to it, the value of markers (M_T) is check-pointed every d (=128) positions (one row), and vertically stored to keep the size in check within *PIM-Aligner* platform. Hence, 256 columns are allocated for storing M_T, each storing 4-byte value for bps (128-bit). After partitioning, starting from the rightmost symbol in *R*,*LFM* procedure runs and returns *low* and *high* for next symbol.

Mapping and Computation Considering current input nucleotide is T and input index as id (in Fig. 6b), *PIM-Aligner* converts the BWT index into the corresponding memory WL and BL addresses storing data $BWT[id - (id \bmod d)]$ to $BWT[id]$. Then, such bits and corresponding CRef-T can implement the parallel comparison operations (*XNOR_Match*). If the XNOR output is '1' (a match is found), DPU's embedded counter counts up to eventually compute $count\_match$ for next operation. Fig. 6b intuitively shows the *XNOR_Match* procedure to locate Ts in a sub-array. When counting is done, the sub-array returns the $count\_match$ and marker address ($marker\_add$), shown in Fig. 6c. The correlated



(a) (b)

Fig. 5. (a) Reference comparison to realize in-memory operations, (b) Monte-Carlo simulation of $\mathrm{V_{sense}}$ distribution in different operations.

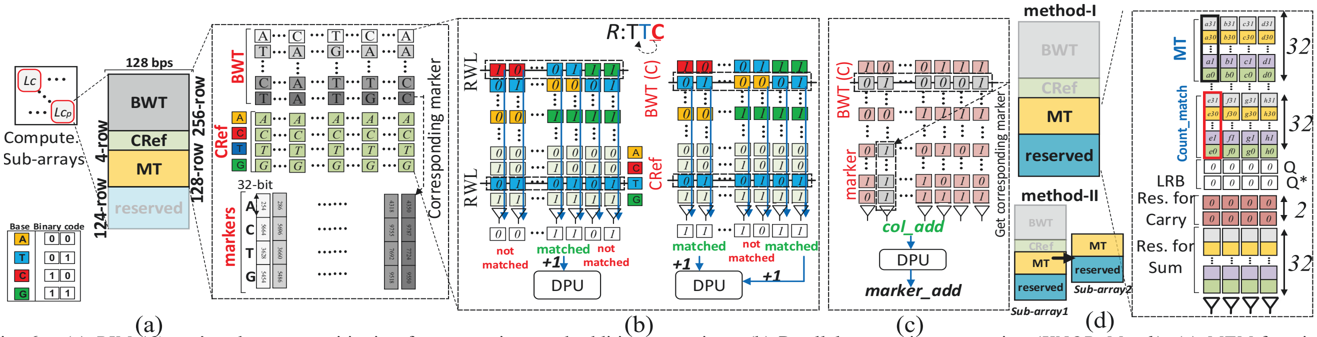


Fig. 6. (a) *PIM-Aligner*'s sub-array partitioning for comparison and addition operations, (b) Parallel comparison operation (*XNOR_Match*), (c) *MEM* function to retrieve marker_add, (d) *IM_ADD* function with two methods.

data partitioning methodology guarantees the read of $marker$ value (*MEM*) is always a local memory access within the same memory array (Fig. 6c). Now, the $marker$ and just computed and transposed $count\_match$ are buffered in M_T and reserved memory spaces, respectively, as shown in Fig. 6d. To further implement *IM_ADD* function, we propose two distinct hardware-friendly methods; method-I performs the bit-line addition within the same computational sub-array based on *PIM-Aligner*'s in-memory addition operation though it degrades the system performance as other sub-array resources (*MEM* and *XNOR_Match*) are not used. To alleviate this issue, method-II essentially duplicates the number of sub-arrays, where only in-memory addition computation is transferred to a second sub-array.

Pipeline Design: To improve the base-line *PIM-Aligner*'s performance, the processing of multiple reads is considered such that in each pipeline stage a different short read-*R* could be processed. We take the partitioning method-II for pipeline design. With a careful observation of DNA alignment computation phases, we realized that the different computing resources of a single sub-array could be set free by copying the sub-array data into a new sub-array. Therefore, we define P_d as parallelism degree (i.e. # of the leveraged sub-arrays) to control the trade-off of resources and performance metrics. For instance, comparison resources of a particular sub-array can be set free after duplicating (P_d=2) that sub-array (method-II). This pipeline technique is intuitively shown in Fig. 7 for a sample 3 reads; when the $R1$ is being processed for *IM_ADD* in the second sub-array, $R2$ can exploits the parallel *XNOR_Match* resources in the first sub-array to increase the parallelism. This can be generalized to more number of sub-arrays where more than two sub-arrays contribute to the computation at the cost of a higher energy consumption.

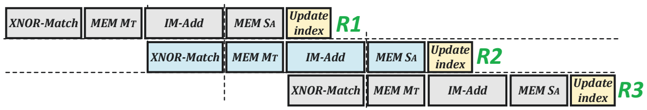


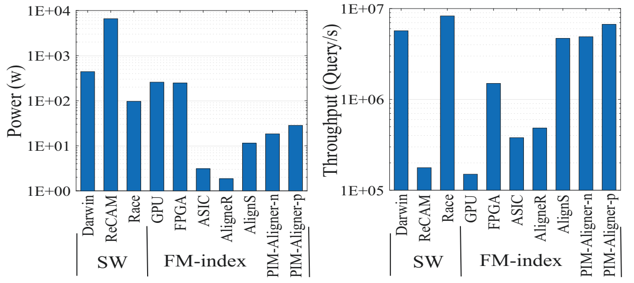
Fig. 7. The proposed pipeline technique with P_d=2.

VI. Evaluations

Evaluation Framework: To evaluate the performance of *PIM-Aligner*, a comprehensive device-to-architecture evaluation framework with two in-house simulators were developed. At the device level, we jointly used the Non-Equilibrium Green's Function (NEGF) and Landau-Lifshitz-Gilbert (LLG) with spin Hall effect equations to model SOT-MRAM bit-cell [16]. For the circuit level, a Verilog-A model of 2T1R SOT-MRAM device was developed to co-simulate with the interface CMOS circuits in Cadence Spectre and SPICE. We used 45nm North Carolina State University (NCSU) Product Development Kit (PDK) library in SPICE to analyze the proposed design and achieve the performance. Besides, we built an architectural-level simulator on NVSim [17]. It can change the configuration files (.cfg) corresponding to different array organization and report performance for PIM operations based device/circuit level data. Then we fed the performance data to a behavioral simulator based on Matlab to calculate the latency and energy that *PIM-Aligner* spends on alignment task based on the algorithms. We perform an extensive comparison with the counterpart computing platforms including SW-based Darwin [7], ReCAM [18] and RaceLogic [6], as well as FM-Index-based platforms including Soap3-dp [5] on GPU, FPGA [9], ASIC [8], AlignS [13], AligneR [3]. In the interest of limited space, we refer the readership to the papers for the detailed configuration of each accelerator. Note that, to perform short read alignment on GPU, we used Soap3 [5] considering only reads with ≤2 mismatches. We re-implemented, ReRAM-, SOT-MRAM, and CAMs with NVSim [17]. For evaluation, we generated 10 million 100-bps short read queries via ART simulator [19] and align them to the human genome Hg19 with different computing platforms. Note that the population variation and genome error rate were set to 0.1% and 0.2%.

Power & Throughput: The power consumption of the DNA alignment task for different accelerators is calculated and shown in Fig. 8a. We implemented the baseline (*PIM-Aligner*-n) and the pipelined *PIM-Aligner* (P_d=2, *PIM-Aligner*-p). Our first observation is that SW-based platforms (except for RaceLogic [6]) require a larger power-budget as we expected, compared with FM-index-based designs. Besides, among FM-index-based platforms, the PIMs generally show less power consumption. ReRAM-based AligneR [3], ASIC [8] and SOT-MRAM-based AlignS [13] respectively consume the least power. *PIM-Aligner*-n stands as the fourth power-efficient design. It is noteworthy that *PIM-Aligner* uses three SAs per bit-line to perform the computation in a single cycle, while the AlignS [13] has two SAs and a two-cycle addition scheme. That is why our design consumes more power compared to the SOT-MRAM counterpart. The throughput results for different platforms are reported in Fig. 8b. We observe that *PIM-Aligner*-p shows the highest throughput compared with other platforms except RaceLogic due to its massively-parallel and local computational scheme. Based on this plot, our pipeline technique with P_d=2 has improved the performance by ∼40% compared to the baseline design.

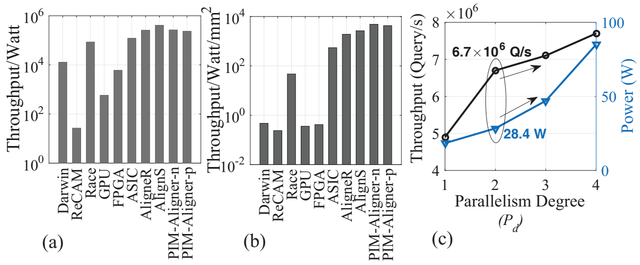
Trade-off: The performance/power trade-off can be better explained by correlated parameters, as plotted in Fig. 9a-b. We observe that SOT-MRAM-AlignS achieves the highest



(a) (b)

Fig. 8. (a) Power consumption and (b) Throughput of different accelerators compared to *PIM-Aligner* (Y-axis:Log scale)

throughput per Watt compared to other platforms. Where *PIM-Aligner*-n stands as the second most efficient design. Our design improves short read alignment's performance by 3.1× over the RaceLogic [6], the best SW-based accelerator, and ∼ 2×, 43.8×, 458× over ASIC [8], FPGA [9], and GPU [5] platforms, respectively. Fig. 9b takes estimated area of the chips into account. Considering the area factor, we observe that *PIM-Aligner* improves read alignment performance significantly over all the other solutions, e.g. by ∼9× and 1.9× compared to FM-index-based ASIC and processing-in-ReRAM designs, respectively. Fig. 9c shows the trade-off between power and throughput w.r.t. parallelism degree. We can see that by increasing the P_r, both power consumption and throughput will increase. Therefore, P_r can be tailored according to the system constraints to provide the best solution.



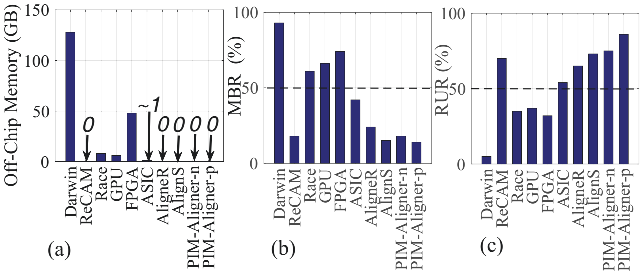
(a) (b) (c)

Fig. 9. (a) Throughput/Watt, (b) Throughput/Watt/Area, and (c) Power-throughput trade-off w.r.t. P_d.

Off-Chip Memory Access: Figure 10a shows the required off-chip memory access for different accelerators. We observe that FM-index-based GPU [5] and FPGA [9] platforms heavily rely on off-chip memory consuming humongous energy to fetch data from stored tables and queries. Note that, ASIC design performs the alignment with only 1GB off-chip memory after compression. Figure 10b reports the Memory Bottleneck Ratio (MBR). Based on this, *PIM-Aligner* spends less than ∼18% time for memory access and data transfer. It is worth pointing out that other PIM platforms also spend less than 25% time waiting for the loading data. AligneR solution shows higher memory bottleneck ratio compared with *PIM-Aligner* owning to its unbalanced computation and data movement. The less MBR can be translated as the higher Resource Utilization Ratio (RUR) for the computing platforms, shown in Fig. 10c. We can see that *PIM-Aligner*-p shows the highest resource utilization with up to ∼86%.

VII. Conclusion

In this work, we presented *PIM-Aligner* to execute DNA alignment based on a hardware-friendly alignment algorithm. We then developed *PIM-Aligner* platform based on SOT-MRAM array to accelerate our reconstructed algorithm. The results show that *PIM-Aligner* outperforms recent platforms based on dynamic programming with ∼3.1× higher throughput per Watt and improves throughput per Watt per mm^2 by ∼9× and 1.9× compared to FM-index-based ASIC and processing-in-ReRAM designs, respectively.



(a) (b) (c)

Fig. 10. (a) Off-chip memory, (b) Memory Bottleneck Ratio, (c) Resource Utilization Ratio for different accelerators.

Acknowledgements

This work is supported in part by the National Science Foundation under Grant No.1740126, No.1908495, No.1931871, No. 1755761 and Semiconductor Research Corporation nCORE.

References

[1] H. Li and N. Homer, "A survey of sequence alignment algorithms for next-generation sequencing," *Briefings in bioinformatics*, vol. 11, no. 5, pp. 473–483, 2010.

[2] H. Li and R. Durbin, "Fast and accurate short read alignment with burrows–wheeler transform," *bioinformatics*, vol. 25, 2009.

[3] F. Zokaee *et al.*, "Aligner: A process-in-memory architecture for short read alignment in rerams," *IEEE Computer Architecture Letters*, 2018.

[4] B. Langmead *et al.*, "Ultrafast and memory-efficient alignment of short dna sequences to the human genome," *Genome biology*, vol. 10, 2009.

[5] R. Luo *et al.*, "Soap3-dp: fast, accurate and sensitive gpu-based short read aligner," *PloS one*, vol. 8, p. e65632, 2013.

[6] A. Madhavan *et al.*, "Race logic: A hardware acceleration for dynamic programming algorithms," in *ACM SIGARCH Computer Architecture News*, vol. 42, 2014.

[7] Y. Turakhia *et al.*, "Darwin: A genomics co-processor provides up to 15,000 x acceleration on long read assembly," in *23rd ASPLOS*. ACM, 2018, pp. 199–213.

[8] Y.-C. Wu *et al.*, "A 135-mw fully integrated data processor for next-generation sequencing," *IEEE TBioCAS*, vol. 11, pp. 1216–1225, 2017.

[9] J. Arram *et al.*, "Leveraging fpgas for accelerating short read alignment," *IEEE/ACM TCBB*, vol. 14, pp. 668–677, 2017.

[10] W. Huangfu, S. Li, X. Hu, and Y. Xie, "Radar: a 3d-reram based dna alignment accelerator architecture," in *55th DAC*. ACM, 2018, p. 59.

[11] S. K. Khatamifard *et al.*, "A non-volatile near-memory read mapping accelerator," *arXiv preprint arXiv:1709.02381*, 2017.

[12] L. Yavits *et al.*, "Resistive associative processor," *IEEE Computer Architecture Letters*, vol. 14, pp. 148–151, 2015.

[13] S. Angizi *et al.*, "Aligns: A processing-in-memory accelerator for dna short read alignment leveraging sot-mram," in *DAC*, 2019, p. 144.

[14] S. Canzar *et al.*, "Short read mapping: An algorithmic tour," *Proceedings of the IEEE*, pp. 436–458, 2017.

[15] S. Li, C. Xu *et al.*, "Pinatubo: A processing-in-memory architecture for bulk bitwise operations in emerging non-volatile memories," in *DAC*. IEEE, 2016.

[16] X. Fong *et al.*, "Spin-transfer torque devices for logic and memory: Prospects and perspectives," *IEEE TCAD*, vol. 35, 2016.

[17] X. Dong *et al.*, "Nvsim: A circuit-level performance, energy, and area model for emerging nonvolatile memory," *IEEE TCAD*, vol. 31, 2012.

[18] R. Kaplan *et al.*, "A resistive cam processing-in-storage architecture for dna sequence alignment," *IEEE Micro*, vol. 37, pp. 20–28, 2017.

[19] W. Huang *et al.*, "Art: a next-generation sequencing read simulator," *Bioinformatics*, vol. 28, pp. 593–594, 2011.