RAPID: A ReRAM Processing in-Memory Architecture for DNA Sequence Alignment

Conference Paper · July 2019

DOI: 10.1109/ISLPED.2019.8824830

5 authors, including:

Saransh Gupta
University of California, San Diego
27 PUBLICATIONS **337** CITATIONS
SEE PROFILE

Mohsen Imani
University of California, San Diego
94 PUBLICATIONS **1,114** CITATIONS
SEE PROFILE

Behnam Khaleghi
University of California, San Diego
30 PUBLICATIONS **270** CITATIONS
SEE PROFILE

RAPID: A ReRAM Processing in-Memory Architecture for DNA Sequence Alignment

Saransh Gupta, Mohsen Imani, Behnam Khaleghi, Venkatesh Kumar, Tajana Rosing
CSE Department, UC San Diego, La Jolla, CA 92093, USA
{sgupta, moimani, bkhalegh, vrkumar, tajana}@ucsd.edu

***Abstract*—Sequence alignment is a core component of many biological applications. As the advancement in sequencing technologies produces a tremendous amount of data on an hourly basis, this alignment is becoming the critical bottleneck in bioinformatics analysis. Even though large clusters and highly-parallel processing nodes can carry out sequence alignment, in addition to the exacerbated power consumption, they cannot afford to concurrently process the massive amount of data generated by sequencing machines. In this paper, we propose a novel processing in-memory (PIM) architecture suited for DNA sequence alignment, called RAPID. We revise the state-of-the-art alignment algorithm to make it compatible with in-memory parallel computations, and process DNA data completely inside memory without requiring additional processing units. The main advantage of RAPID over the other alignment accelerators is a dramatic reduction in internal data movement while maintaining a remarkable degree of parallelism provided by PIM. The proposed architecture is also highly scalable, facilitating precise alignment of lengthy sequences. We evaluated the efficiency of the proposed architecture by aligning chromosome sequences from human and chimpanzee genomes. The results show that RAPID is at least 2× faster and 7× more power efficient than BioSEAL, the best DNA sequence alignment accelerator.**

I. Introduction

DNA comprises long paired strands of nucleotide bases, and DNA sequencing is the process of identifying the order of these bases in the given molecule. Demonstration of nucleotide bases is abstracted away by four representative letters, A, C, G, and T, respectively standing for adenine, cytosine, guanine, and thymine nucleobases. Modern techniques can be applied to human DNA to diagnose genetic diseases by identifying disease-associated structural variants [1]. DNA sequencing also plays a crucial role in phylogenetics, where sequence information can be used to infer the evolutionary history of an organism over time [2]. These sequences can also be analyzed to provide information on populations of viruses within individuals, allowing for a profound understanding of underlying viral selection pressures [3].

Sequence alignment is central to a multitude of these biological applications and is gaining increasing significance with the advent of nowadays high-throughput sequencing techniques which can produce billions of base pairs in hours, and output hundreds of gigabytes of data, requiring enormous computing effort. Different variants of alignment problems have been introduced. However, they eventually decompose the problem down to pairwise (i.e., between two sequences) alignment. The *global* sequence alignment can be formulated as finding the *optimal* edit operations, including deletion, insertion, substituting of the characters, required to transform sequence x to sequence y (and vice versa). The cost of insertion (deletion), however, may depend on the length of the consecutive insertions (deletions).

The search space of evaluating all possible alignments is exponentially proportional to the length of the sequences and becomes computationally intractable even for sequences as small as having just 20 bases. To resolve this, the Needleman-Wunsch algorithm [4] employs dynamic-programming (DP) to divide the problem into smaller ones and construct the solution by using the results obtained from solving the sub-problems, reducing the worst-case performance and space down to $O(mn)$ while delivering higher accuracy compared to the heuristic counterparts such as BLAST [5]. The Needleman-Wunsch, however, needs to create a scoring matrix $M_{m\times n}$ that has a quadratic time and space complexity dependent on the lengths of input sequences and is still compute intensive. For instance, aligning the human's largest chromosome (among 23 pairs) with the corresponding chromosome in chimpanzee (to get evolutionary insights) results in a score matrix with $\sim$ 56.9 peta-elements. Parallelized versions of Needleman-Wunsch rely on the fact that computing the elements on the same diagonal of the scoring matrix need only the elements of the previous two diagonals. The level of parallelism offered by large sequence lengths often cannot be effectively exploited by conventional processor architecture. Some effort has been made to accelerate DNA alignment using different hardware techniques to exploit the parallelism in the application ([6], [7], [8], [9]). Their benefits are limited mainly due to the limited number of cores and a large amount of data movement between the off-chip memory and the processing cores. However, these algorithms require just simple bitwise logic and addition operations rather than complex cores with general-purpose functional units.

Processing in-memory (PIM) architectures are promising solutions to mitigate the data movement issue and provide a large amount of parallelism. PIM enables in-situ computation on the data stored in the memory, hence, throttling the latency of data movement [10], [11], [12], [13]. Nevertheless, PIM-based acceleration demands a cautious understanding of the target application and the underlying architecture. PIM operations, e.g., addition and multiplication, are considerably slower than conventional CMOS-based operations. The advantage of PIM stems from the high degree of parallelism provided and minimal data movement overhead. Our proposed accelerator, RAPID, effectively exploits the properties of PIM to enable a highly scalable, accurate and energy-efficient solution for DNA alignment.

Our main contributions in this paper are as follows:

(1) We make the well-known dynamic programming-based DNA alignment algorithms, e.g., Needleman-Wunsch, compatible with and more efficient for PIM by separating the query and reference sequence matching from the computation of the corresponding score matrix.

(2) We propose a highly scalable H-tree connected architecture for RAPID. It allows low-energy within-the-memory data transfers between adjacent memory units. Also, it enables us to combine multiple RAPID chips to store huge databases and support database-wide alignment.

(3) A RAPID memory unit, consisting of three blocks, provides the capability to perform exact and highly parallel matrix-diagonal-wide forward computation while storing only two diagonals of substitution matrix rather than the whole matrix. It also stores traceback information in the form of direction of computation, instead of element-to-element relation.

(4) We evaluated the efficiency of the proposed architecture by aligning real-world chromosome sequences, i.e., from human and chimpanzee genomes. The results show that RAPID is at least 2×

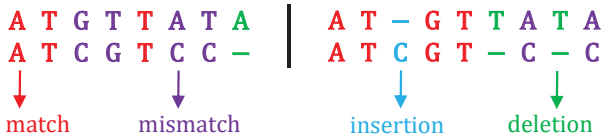
978-1-7281-2954-9/19/$31.00

faster and 7× more power efficient than BioSEAL, the best DNA sequence alignment accelerator.

II. Background and Related Work

A. Sequence Alignment

Natural evolution and mutation as well as experimental errors during sequencing poses two kinds of changes in sequences - substitutions and indels. A substitution changes a base of the sequence with another, leading to a mismatch whereas an indel either inserts or deletes a base. Substitutions are easily recognizable by Hamming distance. However, indels can be mischaracterized as multiple differences, if one merely applies Hamming distance as the similarity metric. The following figure shows comparison of two sequences $x = \mathsf{ATGTTATA}$ and $y = \mathsf{ATCGTCC}$. The left figure rigidly compares the i^{th} base of x with y, while the right figure assumes a different alignment which leads to higher number of matches, taking account the fact that not only bases might change (mismatch) from one sequence to another, insertions and deletions are quite probable. Note that the notation of dashes $(-)$ is conceptual, i.e., there is no



dash $(-)$ base in a read sequence. Dashes are used to illustrate a potential scenario that one sequence has been (or can be) evolved to the other. In short, sequence alignment aims to find out the best number and location of the dashes such that the resultant sequences yield the best similarity score.

As comparing all possible scenarios of aligning is exponentially proportional to the aggregate length of the sequences, hence computationally intractable, more efficient alignment approaches have been proposed. These methods can be categorized to heuristic methods and those based on dynamic programming, namely, Needleman-Wunsch [4] for global and Smith-Waterman [14] for local alignment. Heuristic techniques are computationally lighter but do not guarantee to find an optimal alignment solution, especially in sequences that contain a large number of indels. However, these techniques are still employed because the exact methods (based on dynamic programming) were computationally feasible. Though they might also utilize the Smith-Waterman dynamic programming approach to build-up the final gapped alignment [5], [15].

Dynamic programming based methods involve forming a substitution matrix of the sequences. This matrix computes scores of various possible alignments based on a scoring reward and mismatch penalty. These methods avoid redundant computations by using the information already obtained for alignment of the shorter subsequences. The problem of sequence alignment is analogous to the Manhattan tourist problem: starting from coordinate $(0,0)$, i.e., left upper corner, we need to maximize the overall weights of the edges down to the (m,n), wherein the weights are the rewards/costs of matches, mismatches, and indels. Fig. 1 demonstrates the alignment of our previous example. The reference sequence $x = \mathsf{ATGTTATA}$ is put as the left column and the query sequence $y = \mathsf{ATCGTCC}$ as the first row. Diagonal moves indicate traversing a base in both sequences, which results in either a match (↘) or mismatch (↘). Each → (right) and ↓ (down) edge in the alignment graph denotes an insertion and deletion, respectively.

Fig. 1(b) shows the art of dynamic programming (Needleman-Wunsch), wherein every solution point has been calculated based on the best previous solution. The number adjacent to each vertex shows the score of the alignment up to that point, assuming a score of $+1$ for matches and -1 for the substitutions and indels. As an instance, alignment of $x = \mathsf{ATG}$ with $y = \mathsf{ATC}$ corresponds to the coordinate (3,3) in Fig. 1, denoted by ●. This alignment, recursively,

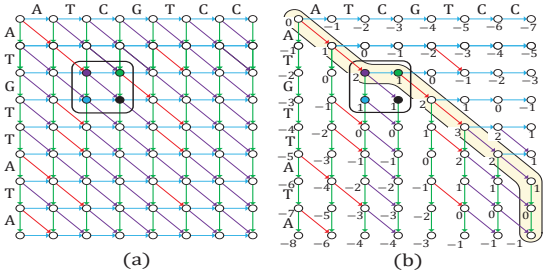


Fig. 1. (a) Alignment graph of the sequences ATGTTATA and ATCGTCC, (b) solution using dynamic programming.

can be achieved in three different ways. The first approach is to first align $x' = \mathsf{AT}$ and $y' = \mathsf{AT}$, denoted by ● in the figure, followed by concatenation of G and C into the x' and y', which causes a mismatch (↘) and score deduction by -1. Another approach could be moving from ● to ● meaning that align ATG with ATC from $x' = \mathsf{AT}-$ with $y' = \mathsf{ATC}$ by deletion (↓), i.e., making $x' = \mathsf{AT}-\mathsf{G}$ and $y' = \mathsf{ATC}-$. The third approach was forming ● by inserting C while moving from ●. As it can be perceived, though we need to solve the three immediate previous alignments, we do not need to calculate them from scratch as they would be obtained already if we fill out the alignment graph diagonally, moving from the first diagonal (which has a single element) towards the last diagonal in a baroque manner. This grants us diagonal-level parallelism as we can calculate all elements of a diagonal independently.

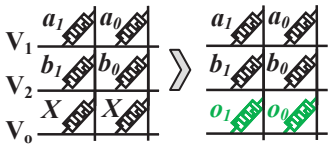
All in all, the Needleman-Wunsch global alignment can be formulated as follows. Matrix M holds the best alignment score for all pairwise alignments and $score(x_i,-)$ and $score(-,y_i)$ denote the cost of deletion and insertion. It essentially assumes the scenarios above when recursively aligning two sequences: whether either x or y ends with $-$, or a base. The difference of Smith-Waterman algorithm is it changes the negative edges to 0 to account for local alignments.

$$M_{i,j} = max \begin{cases} M_{i-1,j} + score(x_i,-) & deletion \\ M_{i,j-1} + score(-,y_i) & insertion \\ M_{i-1,j-1} + score(x_i,y_i) & match/mismatch \end{cases}$$

B. Digital Processing In-Memory

Traditionally, processing with memristors is based on reading currents through different cells. However, some recent work has demonstrated ways, both in literature [16], [17], [18], [19], [20], [21] and by fabricating chips [22], to implement logic using memristor switching. Digital processing in-memory exploits variable switching of memristors. The output device switches whenever the voltage across it exceeds a threshold [23]. This property can be exploited to implement a variety of logic functions inside memory [16], [18], [24]. Figure 2 shows the latest work in this direction where the output of operation changes with the applied voltage [16]. In-memory operations are in general slower than the corresponding CMOS-based implementations because memristor devices switch slowly. However, PIM architectures can provide significant speedup when running applications with massive parallelism. For example, it takes the same amount of time for PIM to perform operations in a single row or all rows [25].

The PIM based designs proposed in PRINS [26] and BioSEAL [26] accelerates the Smith-Waterman algorithm based on associative computing. The major issue with these works is their large amount of write operation and internal data movement to perform the sequential associative search. Another set of work accelerates short read alignment, where large sequences are broken down into smaller sequences and one of the heuristic methods is applied. The work in RADAR [27] and AligneR [28] exploited the same ReRAM to design a new architecture to accelerate BLASTN and FM-indexing for DNA alignment. The work in [29] and Darwin [30] propose new ASIC accelerators and algorithm [30] for short read



V_1	V_2	V_o	Op
0	0	$2v_{off}$	NOR
0	0	$2v_{off}/3$	NAND
v_{on}	v_{on}	0	OR

Fig. 2. Implementing operations using digital processing in-memory.

alignment. Some implementations have been done on FPGAs [8], [9], [31], of which ASAP [31] proposed a programmable hardware by using circuit delays proposed by RaceLogic [32].

III. In-Memory Implementation of RAPID

This section details RAPID, which implements the DNA sequence alignment algorithm discussed. It adopts a holistic approach where it changes the traditional implementation of the algorithm to make it compatible with memory. Then, it proposes an architecture which takes into account the structure of and the data dependencies in DNA alignment. The proposed architecture is scalable and minimizes internal data movement.

A. Implementation of Algorithm using PIM

In order to optimize the DP algorithms for in-memory computation, we made some modifications to the implementation of the substitution-matrix buildup. The score of aligning two characters depends if they do match or not. First, we avoid additional branching during the computation by pre-computing a matrix C, which separates the input sequence matching stage from the corresponding forward computation phase, allowing RAPID to achieve high parallelism. The matrix C is defined as follows:

$$C[i,j] = \begin{cases} 0 & x[i] = y[j] \\ m & x[i] \neq y[j] \end{cases} \quad (1)$$

As discussed in the previous section, DP algorithms for sequence alignment are inherently parallelizable in a diagonal-wise manner. To make it compatible with the column-parallel operations in PIM, we map each diagonal of the matrix M to a row of memory. Let us consider the set of diagonals $\{d_0, d_1, d_2, \cdots, d_{n'}\}$, where $n' = n+m$, equal to the total length of the sequences. Let $M[i,j]$ correspond to $d_k[l]$. Then, $M[i-1,j]$, $M[i,j-1]$, and $M[i-1,j-1]$ correspond to $d_{k-1}[l]$, $d_{k-1}[l-1]$, and $d_{k-2}[l-1]$ respectively.

Second, we deal solely within the domain of unsigned integers, based on the observation that writing '1' to the memory is both slower as well as more energy consuming than writing '0' [33]. Since negative numbers are sign-extended with '1,' writing small negative numbers in the form of 32-bit words has a significantly higher number of 1's. Especially, $-10 \leq m < 0$, which sets $> 90\%$ of bits to '1' as compared to $\sim 5\%$ for their positive counterparts. We enable this by changing the definition of matrix M as follows.

$$M[0,i] = \sigma \times i \quad i \in [1,L_x] M[j,0] = \sigma \times j \quad j \in [1,L_y] \quad (2)$$

with σ being the cost of indels. This modification makes two changes in the matrix build-up procedure. First, instead of subtracting the matrix elements by σ in the case of indels, now we need to add *sigma* with those elements. Consequently, we replace maximum operation with a minimum function between the three matrix elements. Hence, when $x[i] = y[j]$, the element updates as follow (note that, as shown by Equation 1, the cost of aligning the same bases is 0).

$$\min(M[i-1,j-1], M[i,j-1]+\sigma, M[i-1,j]+\sigma)$$

Using the these modifications, we can reformulate the algorithm in terms of in-memory row operations as follows.

$$M[i,j] = \min(M[i-1,j-1]+C[i,j], M[i,j-1]+\sigma, M[i-1,j]+\sigma)$$

which is equivalent to,

$$d_k[l] = \min(d_{k-2}[l-1]+C[k-2,l-1], d_{k-1}[l]+\sigma, d_{k-1}[l-1]+\sigma)$$

Since we already have d_{k-2} and d_{k-1} while computing d_k, we can express the computation of d_k as:

Algorithm-1:

1) Add σ to every element in d_{k-1} and store it in row A
2) Copy row A to row B, then shift row B right by one
3) Element-wise add C_{k-2} to d_{k-2}, and store it in row C
4) Set d_k to be the element-wise minimum of A, B and C

We perform all steps of the algorithm using only shift, addition, and minimum operations.

Once the matrix computation completes, the backtracking step must occur. As detailed in the following Section, RAPID enables backtracking efficiently in memory by dedicating small memory blocks that store the direction of traceback computation.

B. RAPID Architecture

A RAPID chip consists of multiple RAPID-units connected in H-tree structure, shown in Figure 3. The RAPID-units collectively store database sequences or reference genome and perform the alignment. For maximum efficiency, RAPID evenly distributes the stored sequence among the units. RAPID takes in a query sequence and finally outputs details of the required insertions and deletions in the form of traceback information. As shown in Figure 4b, RAPID takes in one word at a time. An iteration of RAPID evaluates one diagonal of the substitution or the alignment matrix. After every iteration, RAPID takes in a new word of the query sequence and the previous part is propagated through different units as shown in Figure 4b. This propagation results in a continuous transfer of data from a unit to the next unit. Although this transfer is limited to a few words per iteration, it may create a bottleneck if the conventional bus-like interconnect is used to connect memory blocks. We also observe that most of these transfers are local, i.e., between adjacent units. Hence, RAPID uses an H-tree interconnect to connect different units.

1) RAPID organization: The H-tree structure of RAPID directly connects the adjacent units. Figure 3a show the organization in detail. The H-tree interconnect allows low latency transfers between adjacent units. The benefits are enhanced as it allows multiple unit-pairs to exchange data in parallel. The arrows in the figure represent these data transfers, where transfers denoted by same colored arrows happen in parallel. We also enable computation on these interconnects. In RAPID, each H-tree node has a comparator. This comparator receives some alignment scores from either two units or two nodes and stores the maximum of the two along with the details of its source. These comparators are present at every level of the hierarchy and track the location of the global maximum of chip.

2) RAPID-unit: Each RAPID-unit is made up of three ReRAM blocks: a big $C-M$ block and two smaller blocks, Bh and Bv. The $C-M$ block stores the database or reference genome and is responsible for the generation of C and M matrices discussed in Section III-A. The Bh and Bv blocks store traceback information corresponding to the alignment in $C-M$.

$C-M$ block: A $C-M$ block is shown in Figure 3c. The $C-M$ block stores the database and computes matrices C and M, introduced in Section III-A. C-M is divided into two sub-blocks using switches. The upper sub-block stores the database sequences in the gray region in the figure and computes the C matrix in the green region while the lower sub-block computes the M matrix in the blue region. This physical division allows RAPID to compute both the matrices independently while eliminating data transfer between the two matrices. C matrix pre-computes the penalties for mismatches between the query and reference sequences. The C sub-block generates one diagonal of C at a time. In each iteration, RAPID stores the new input word received from the adjacent unit as c_{in1}. The

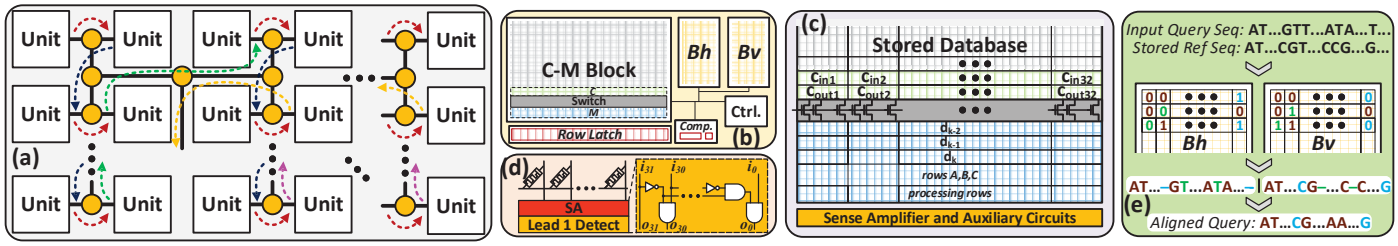


Fig. 3. RAPID architecture. (a) Memory organization in RAPID with multiple units connected in H-tree fashion. Same colored arrows represent parallel transfers. Each node in the architecture has a 32-bit comparator, represented by yellow circles. (b) A RAPID unit consisting of three memory blocks, $C-M$, Bh, and Bv. It also has a row latch and a registered-comparator (Comp.). (c) A $C-M$ block is a single memory block, physically partitioned into two parts by switches. It has three regions, gray for storing the database or reference genome, green to perform query-reference matching and build matrix C, and blue to perform the steps of Algorithm 1. (d) The sense amplifiers of $C-M$ block and the leading '1' detector used for executing minimum. (e) Bh and Bv blocks which store traceback directions and the resultant alignment.

c_{in2} to c_{in32} are formed by shifting the previous part of the sequence by one word as shown in Figure 4b. The resulted c_{in} is then compared with one of the database rows to form $C[i,j]$ for the diagonal as discussed in Equation 1. RAPID makes this comparison by XORing c_{in} with the database row. It stores the output of XOR in a row, c_{out}. All the non-zero data points in c_{out} are then set to m (Equation 1). $C[i,j]$ generation uses just two rows in the C sub-block.

The M sub-block generates one diagonal of the substitution-matrix at a time. According to Algorithm 1, this computation involves two previously computed rows of M and one row of the C matrix. Hence, for computation of a row d_k in M, d_{k-2} and d_{k-1} are required. $C[i,j]$ is made available by activating the switches. The operations involved as per Algorithm 1, namely XOR, addition, and minimum, are fully supported in memory as described in Section III-C. The rows A, B, and C in Figure 3c correspond to the rows A, B, and C in Algorithm 1. After the evaluation of d_k, we need to identify the maximum value in d_k to keep track of the maximum alignment score. Hence, RAPID reads out d_k and stores it in the row latch in the figure. A comparator next to the row latch serially processes all the words in a row of $C-M$ block and stores the value and index of the maximum alignment score. As shown in the figure, at any instance k, we just store d_k, d_{k-1}, and d_{k-2}. This is in contrast to traditional implementations, where the entire M matrix (of size $length\_query \times length\_reference$) is computed and stored. RAPID enables the storage of just two previous rows by (i) continuously tracking the global maximum alignment score and its location using H-tree node comparators and local unit comparators and (ii) storing traceback information. In total, M sub-block uses just eight rows, including two processing rows.

C-M block computational flow: Only one row of C is needed while computing a row of M. Hence we parallelize the computation of C and M matrices. The switch-based subdivision physically enables this parallelism. At any computation step, k, $C[k]$ is computed in parallel to the addition of g to d_{k-1} (1 in Algorithm-1). Then addition output is read from the row A and written back after being shifted by one (2 in Algorithm-1) to row B. Following the algorithm, $C[k]$ is added to d_{k-2} and stored in row C and finally d_k is calculated by selecting the minimum of the results of previous steps.

***Bh* and *Bv* blocks:** The matrices Bh and Bv together form the backtracking matrix. Every time a row d_k is calculated, Bh and Bv are set depending upon the output of minimum operation. Let d_{kl} represent lth word in d_k. Whenever the minimum for lth word is row A, $\{Bh[k,l], Bv[k,l]\}$ is set to $\{1,0\}$, for row B, $\{Bh[k,l], Bv[k,l]\}$ is set to $\{0,1\}$, and for row C, both $Bh[k,l]$ and $Bv[k,l]$ are reset to 0. After the completion of forward computation (substitution matrix buildup), the 1s in matrices Bh and Bv are traced from end to the beginning. The starting point for alignment is given by the index corresponding to the maximum score, which is present in the top-level registered-comparator. The values of $[Bh[i,j], Bv[i,j]]$ form the final alignment. If $[Bh_i j, Bv_i j]$ is (i) [1,0]: it represents insertion, (ii) [0,1]: it represents deletion, and (iii) [0,0]: it represents no gap.

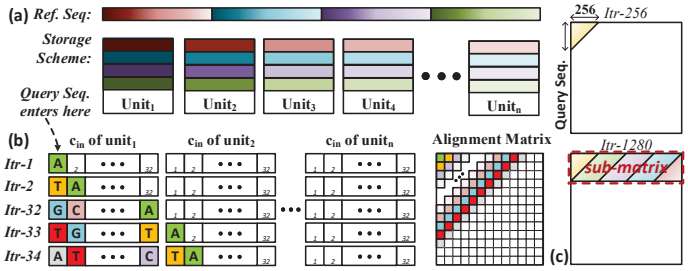


Fig. 4. (a) Storage scheme in RAPID for reference sequence, (b) propagation of input query sequence through multiple units, and (c) evaluation of sub-matrices when the units are limited.

Figure 3e presents an example of sequence alignment performed using the traceback information from Bh and Bv. The two blocks store the same number of words as the C-M block. However, each of these words is just one bit in length (0 or 1).

3) Sequence Alignment over Multiple RAPID Units: Here, we first demonstrate the working of RAPID using a small example and then generalize it.

Example Setting: Say, that the RAPID chip has eight units, each with a $C-M$ block size of 1024×1024. 1024 bits in a row result in a unit with just 32 words per row, resulting in Bh and Bv blocks of size 32×1024 each. Assume that the accelerator stores a reference sequence, seq_r, of length 1024.

Storage of Reference Sequence: The reference sequence is stored in a way to maximize the parallelism while performing DP-based alignment approaches, as shown in Figure 4a. RAPID fills a row, r_i, of all the $C-M$ blocks before storing data in the consecutive rows. Hence, in total, the first 256 data words, 8×32 (#units$\times$#words-per-row), are stored in the first rows of the units and the next 256 data words in the second rows. Since only 256 words of the reference sequence are available at a time, this chip can process just 256 elements of a diagonal in parallel.

Alignment: Now, a query sequence, seq_q, is to be aligned with seq_r. Let the lengths of the query and reference sequences be L_q and L_r, both being of length 1024. The corresponding substitution-matrix is of the size $L_q \times L_r$, 1024×1024 in this case. As our sample chip can process a maximum of 256 data words in parallel, we deal with 256 query words at a time. We divide the substitution-matrix into sub-matrices of 256 rows as shown in the Figure 4c and process one sub-matrix at a time.

The sequence seq_q is transferred to RAPID, one word at a time, and stored as the first word of c_{in}. Every iteration receives a new query word (base). c_{in} is right-shifted by one word and we append the new word received to c_{in}. The resultant c_{in} is used to generate one diagonal of substitution-matrix as explained earlier. For the first 256 iterations, RAPID computes first 256 diagonals completely as shown in Figure 4c. This processes the first 256 query words with the first 256 reference words. Now, instead of receiving new inputs, RAPID uses the same inputs but processes them with the reference words in the second row. This goes on until the current 256 words of the query have not processed with all the rows of the reference sequence.

In the end, the first submatrix of 256 rows is generated (Figure 4c). It takes 256×5 iterations ($words\_in\_row \times (\#seq_r\_rows + 1)$). Similarly, RAPID generate the following sub-matrices.

4) Suitability of RAPID for DNA alignment: RAPID instruments each storage block with computing capability. This results in low internal data movement between different memory blocks. Also, the typical sizes of DNA databases don't allow storage of entire databases in a single memory chip. Hence, any accelerator using DNA databases needs to be highly scalable. RAPID, with its mutually independent blocks, compute-enabled H-tree connections, and hierarchical architecture, is highly scalable within a chip and across multiple chips as well. The additional chips add levels in RAPID's H-tree hierarchy.

C. RAPID Circuit Details

The computation of the proposed algorithm uses three main operations: XOR, addition, and Min/Max. In the following, we explain how RAPID can implement these functionalities on different rows of a crossbar memory.

XOR: We use the PIM technique proposed in [16] to implement XOR in memory. XOR ($\oplus$) can be expressed in terms of OR ($+$), AND (.), and NAND ($(.)'$) as $A \oplus B = (A+B).(A.B)'$. We first calculate OR and then use its output cell to implement NAND. These operations are implemented by the method ealier discussed in Section II-B. We can execute this operation in parallel over all the columns of two rows.

Addition: Let A, B, and C_{in} be 1-bit inputs of addition, and S and C_{out} the generated sum and carry bits respectively. Then, S is implemented as two serial in-memory XOR operations (A$\oplus$B)$\oplus$C. C_{out}, on the other hand, can be executed by inverting the output of the Min function proposed in [16]. Addition takes a total of 6 cycles and similar to XOR, we can parallelize it over multiple columns.

Minimum: A minimum operation between two numbers is typically implemented by subtracting the numbers and checking the sign bit. The performance of subtraction scales with the size of inputs. Multiple such operations over long vectors lead to lower performance. Hence, we utilize a parallel in-memory minimum operation recently proposed in [34]. It finds the element-wise minimum in parallel between two large vectors without sequentially comparing them. First, it performs a bitwise XOR operation over two inputs. Then it uses the leading one detector circuit in Figure 3d to find the most significant mismatch bit in those words. The input with a value of '0' at this bit is the minimum of the two.

IV. Evaluation

A. Experimental setup

We created and used a cycle-accurate simulator which simulates the functionality of RAPID. For the accelerator design, we use HSPICE for circuit-level simulations to measure the energy consumption and performance of all the RAPID operations using a 45nm process node. We used the VTEAM ReRAM device model proposed in [23] with the R_{OFF}/R_{ON} of 10M/10k Ω. The device has a switching time of 1.1 ns, allowing RAPID to run at a frequency of 0.9 GHz. Energy consumption and performance are also cross-validated using NVSim [35]. The device area model is based on [36]. We used *System Verilog* and *Synopsys Design Compiler* to implement and synthesize RAPID controller.

For the comparisons, we consider RAPID with an area of $660mm^2$, similar to NVIDIA GTX-1080 Ti GPU with 4GB memory, unless otherwise stated. In this configuration, RAPID has a simulated power dissipation of 470 W as compared ~100 kW for 384-GPU cluster of CUDAlign 4.0, ~1.3 kW for PRINS, and ~1.5 kW for BioSEAL, while running similar workloads.

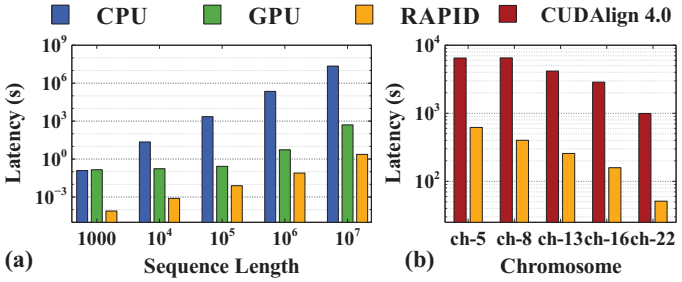


Fig. 5. Runtime comparison across different platforms

B. RAPID & Sequence Length

We first evaluate RAPID's operational efficiency by implementing a pair-wise alignment of two sequences and compare the results with in-house CPU and GPU implementations. We vary the length of sequences to observe its effect on RAPID. For CPU baseline, we use a Julia implementation of the Needleman-Wunsch algorithm, run on a 2.8 GHz Intel Core i7 processor. While for GPU baseline, we use the implementation of Needleman-Wunsch from the package CUDAlign 4.0 [7], run on NVIDIA GTX 1080 Ti GPU.

Figure 5a shows the execution time of DNA alignment on different platforms. As our evaluation shows, increasing the length of the sequence degrades the alignment efficiency. However, the change in efficiency depends on the platform. Increasing the sequence length, exponentially increases the execution time of the CPU. This increase is because the CPU does not have enough cores to parallelize the alignment computation, resulting in a large amount of data movement between memory and processing cores. Similarity, the execution time of GPU increases with the sequence length. In contrast, RAPID has much smoother increases in the energy and execution time of the alignment. RAPID enables column-parallel operations where the alignment time only depends on the number of memory rows, which linearly increases by the size of sequences. Our evaluation over varying sequence lengths shows that for a sequence length of $l = 1000$ RAPID is 1585× faster computation than CPU. The benefit increases drastically as the size of the sequence increases, with RAPID being $9.6x10^6\times$ faster than CPU for $l = 10,000,000$. The GPU implementation is observed to have significant latency overhead in the initial stages, performing very poorly on smaller test cases. However, in middle size sequences of $l = 100,000$, GPU is only 34× slower than RAPID. However, as the sequence length further increases, e.g., $l = 10,000,000$, we observe that RAPID outperforms GPU by 214×. On the other hand, RAPID works better than GPU for both small and massive datasets.

C. RAPID for Exact Chromosome-wide Alignment

Workload: To demonstrate the applicability of RAPID to long chromosome-wide alignment, we used DNA sequences from the National Center for Biotechnology Information (NCBI). We compared the 25 homologous chromosomes from humans (GRCh37) and chimpanzees (panTro4). The sizes of chromosomes vary from 26 MBP (million base pairs) to 249 MBP. In total, these chromosomes add up to 3.3 billion base pairs (BBP) for GRCh37 and 3.1 BBP for panTro4. The human genome is assumed to be pre-stored in RAPID and acts as the reference sequence. The chimpanzee genome is assumed to be transferred to RAPID and acts as the query sequence. RAPID takes in one new base every iteration and propagates it. In the time taken by the external system to send a new query base to RAPID, it processes a diagonal of the substitution matrix. In every iteration, RAPID processes a new diagonal. For example, a comparison between chromosome-1 ($ch-1$) of human genome with 249 MBP and $ch1$ of chimpanzee genome with 228 MBP results in a substitution matrix with 477 million diagonals, requiring those many forward computation operations and then traceback.

Comparison to state-of-the-art: We compare RAPID with state-of-the-art implementations of DNA sequence alignment while run-

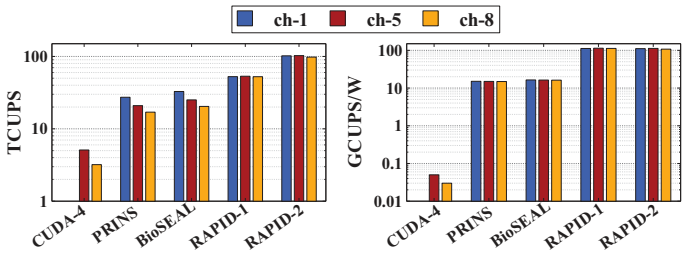


Fig. 6. Comparison of execution of different chromosome test pairs. RAPID-1 is a RAPID chip of size 660 mm^2 while RAPID-2 has an area of 1300 mm^2.

TABLE I
RELATIVE LATENCY AND POWER OF DIFFERENT RAPID CHIPS FOR $ch-1$

Chip Area (mm^2)	# Chips	Relative Latency	Relative Power	Chip Area (mm^2)	# Chips	Relative Latency	Relative Power
660	1	1.00	1.00	**168**	1	3.82	0.26
	2	0.52	1.93		4	1.03	0.97
1300	1	0.52	1.96		8	0.54	1.88
332	1	1.94	0.51	**85**	1	7.50	0.13
	2	1.01	0.99		8	1.04	0.96
	4	0.53	1.90		16	0.54	2.09

*One 660 mm^2 RAPID chip: 1081 s, 470 W

ning exact chromosome-wide alignment. We compare our results with CUDAlign 4.0 [7], the fastest GPU-based implementation and two ReCAM-based DNA alignments architecture proposed in PRINS [26] and BioSEAL [37]. Figure 5b shows the execution time of aligning different test pairs on RAPID and CUDAlign 4.0. We observe that RAPID is on an average 11.8× faster than the CUDAlign 4.0 implementation with 384 GPUs. The improvements from RAPID increase further if fewer GPUs are available. For example, RAPID is over 300× faster than CUDAlign 4.0 with 48 GPUs. We also evaluated RAPID, CUDAlign 4.0, and the ReCAM-based designs, PRINS [26] and BioSEAL [37] in terms of cell updates per sec (CUPS) as shown in Figure 6a. RAPID achieves 2.4× and 2× higher performance as compared to PRINS and BioSEAL. It is also on average, 2820× more energy efficient than CUDAlign 4.0 and 7.5× and 6.9× than PRINS and BioSEAL respectively as shown in Figure 6b. Also, when the area of the RAPID chip increases from the current 660 mm^2 to 1300 mm^2, the performance doubles without increasing the total energy consumption significantly.

Scalability of RAPID: Table I shows the latency and power of RAPID while aligning $ch-1$ pair from human and chimpanzee genomes on different RAPID chip sizes. Keeping RAPID-660 mm^2 as the base, we observe that with decreasing chip area, the latency increases but power reduces almost linearly, implying that the total power consumption remains similar throughout. We also see that, by combining multiple smaller chips, we can achieve performance similar to a bigger chip. For example, eight RAPID-85 mm^2 chips can collectively achieve a speed similar to a RAPID-660 mm^2 chip, with just 4% latency overhead. This exceptional scalability is due to the hierarchical structure of H-tree, where RAPID considers multiple chips as additional levels in H-tree organization. However, this comes with the constraint of having the number of memory blocks as powers of 2 for maximum efficiency and simple organization.

D. Area Overhead

RAPID incurs 25.2% area overhead as compared to a conventional memory crossbar of the same memory capacity. This additional area comes in the form of registered-comparators in units and at interconnect nodes (6.9%) and latches to store a whole row of a block (12.4%). We use switches to physically partition a C-M memory block, which contributes 1.1%. Using H-tree instead of the conventional interconnect scheme takes additional 4.8%.

V. CONCLUSION

We proposed a novel processing-in-memory (PIM) architecture suited for DNA sequence alignment, called RAPID. The main advantage of RAPID over the other alignment accelerators is the dramatic reduction in internal data movement while maintaining a remarkable degree of operational column-level parallelism provided by PIM. The proposed architecture is highly scalable, which facilitates precise alignment of lengthy sequences. We evaluated the efficiency of the proposed architecture by aligning chromosome sequences from human and chimpanzee genomes. The results show that RAPID is at least 2× faster and 7× more power efficient than BioSEAL, the best DNA sequence alignment accelerator.

ACKNOWLEDGEMENTS

This work was partially supported by CRISP, one of six centers in JUMP, an SRC program sponsored by DARPA, and also NSF grants #1730158 and #1527034.

REFERENCES

[1] F. E. Dewey *et al.*, "Dna sequencing: Clinical applications of new dna sequencing technologies," *Circulation*, vol. 125, no. 7, pp. 931–944, 02 2012.
[2] A. J. Drummond *et al.*, "Beast: Bayesian evolutionary analysis by sampling trees," *BMC Evolutionary Biology*, vol. 7, no. 1, p. 214, 2007.
[3] S. J. Watson *et al.*, "Viral population analysis and minority-variant detection using short read next-generation sequencing," *Philosophical Transactions of the Royal Society B: Biological Sciences*, vol. 368, no. 1614, p. 20120205, 03 2013.
[4] S. B. Needleman *et al.*, "A general method applicable to the search for similarities in the amino acid sequence of two proteins," *JMB*, vol. 48, no. 3, pp. 443–453, 1970.
[5] S. F. Altschul *et al.*, "Basic local alignment search tool," *JMB*, vol. 215, no. 3, pp. 403–410, 1990.
[6] Y. Liu *et al.*, "Swaphi-ls: Smith-waterman algorithm on xeon phi coprocessors for long dna sequences," in *2014 IEEE CLUSTER*. IEEE, 2014, pp. 257–265.
[7] E. F. de Oliveira Sandes *et al.*, "Cudalign 4.0: incremental speculative traceback for exact chromosome-wide alignment in gpu clusters," *IEEE Transactions on Parallel and Distributed Systems*, vol. 27, no. 10, pp. 2838–2850, 2016.
[8] L. Wienbrandt, "The fpga-based high-performance computer rivyera for applications in bioinformatics," in *Conference on Computability in Europe*. Springer, 2014.
[9] J. Arram *et al.*, "Leveraging fpgas for accelerating short read alignment," *IEEE/ACM TCBB*, vol. 14, no. 3, pp. 668–677, 2017.
[10] M. Gokhale *et al.*, "Processing in memory: The terasys massively parallel pim array," *Computer*, vol. 28, no. 4, pp. 23–31, 1995.
[11] J. Ahn *et al.*, "Pim-enabled instructions: a low-overhead, locality-aware processing-in-memory architecture," in *ISCA*. IEEE, 2015, pp. 336–348.
[12] S. Li *et al.*, "Pinatubo: a processing-in-memory architecture for bulk bitwise operations in emerging non-volatile memories," in *DAC*. ACM, 2016, p. 173.
[13] S. Gupta *et al.*, "Nnpim: A processing in-memory architecture for neural network acceleration," *IEEE Transactions on Computers*, pp. 1–1, 2019.
[14] T. F. Smith *et al.*, "Identification of common molecular subsequences," *JMB*, vol. 147, no. 1, pp. 195–197, 1981.
[15] D. J. Lipman *et al.*, "Rapid and sensitive protein similarity searches," *Science*, vol. 227, no. 4693, pp. 1435–1441, 1985.
[16] S. Gupta *et al.*, "Felix: Fast and energy-efficient logic in memory," in *2018 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. IEEE, 2018, pp. 1–7.
[17] N. Talati *et al.*, "Logic design within memristive memories using memristor-aided loGIC (MAGIC)," *IEEE TNano*, vol. 15, no. 4, pp. 635–650, jul 2016.
[18] S. Kvatinsky *et al.*, "MAGIC – memristor-aided logic," *TCAS II*, vol. 61, no. 11, 2014.
[19] J. Borghetti *et al.*, "Memristive switches enable stateful logic operations via material implication," *Nature*, vol. 464, no. 7290, pp. 873–876, 2010.
[20] M. Imani *et al.*, "Digitalpim: Digital-based processing in-memory for big data acceleration," in *Proceedings of the GLSVLSI*. ACM, 2019.
[21] J. Sim *et al.*, "Upim: Unipolar switching logic for high density processing-in-memory applications," in *Proceedings of the GLSVLSI*. ACM, 2019.
[22] B. C. Jang *et al.*, "Memristive logic-in-memory integrated circuits for energy-efficient flexible electronics," *Advanced Functional Materials*, vol. 28, no. 2, p. 1704725, 2018.
[23] S. Kvatinsky *et al.*, "Vteam: a general model for voltage-controlled memristors," *TCAS II*, vol. 62, no. 8, pp. 786–790, 2015.
[24] M. Imani *et al.*, "Floatpim: In-memory acceleration of deep neural network training with high precision," in *Proceedings of the ISCA*. ACM, 2019.
[25] M. Imani et al., "Ultra-efficient processing in-memory for data intensive applications," in *Proceedings of the 54th DAC 2017*. ACM, 2017, pp. 1–6.
[26] R. Kaplan *et al.*, "A resistive cam processing-in-storage architecture for dna sequence alignment," *IEEE Micro*, vol. 37, no. 4, pp. 20–28, 2017.
[27] W. Huangfu *et al.*, "Radar: a 3d-reram based dna alignment accelerator architecture," in *2018 55th ACM/ESDA/IEEE DAC*. IEEE, 2018, pp. 1–6.
[28] F. Zokaee *et al.*, "Aligner: A process-in-memory architecture for short read alignment in rerams," *IEEE Computer Architecture Letters*, vol. 17, no. 2, pp. 237–240, 2018.
[29] Y.-C. Wu *et al.*, "A 135-mw fully integrated data processor for next-generation sequencing," *IEEE transactions on biomedical circuits and systems*, vol. 11, no. 6, 2017.
[30] Y. Turakhia *et al.*, "Darwin: A genomics co-processor provides up to 15,000 x acceleration on long read assembly," in *Proceedings of 23rd ASPLOS*. ACM, 2018.
[31] S. S. Banerjee *et al.*, "Asap: Accelerated short-read alignment on programmable hardware," *IEEE Transactions on Computers*, 2019.
[32] A. Madhavan *et al.*, "Race logic: A hardware acceleration for dynamic programming algorithms," in *ACM SIGARCH Computer Architecture News*, vol. 42, no. 3, 2014.
[33] P. Knag *et al.*, "A native stochastic computing architecture enabled by memristors," *IEEE TNano*, vol. 13, no. 2, 2014.
[34] M. Zhou *et al.*, "Gram: graph processing in a reram-based computational memory," in *Proceedings of the 24th ASPDAC*. ACM, 2019, pp. 591–596.
[35] X. Dong *et al.*, "Nvsim: A circuit-level performance, energy, and area model for emerging nonvolatile memory," *IEEE TCAD*, vol. 31, no. 7, pp. 994–1007, 2012.
[36] R. Fackenthal *et al.*, "19.7 a 16gb reram with 200mb/s write and 1gb/s read in 27nm technology," in *2014 IEEE ISSCC*. IEEE, 2014, pp. 338–339.
[37] R. Kaplan *et al.*, "Bioseal: In-memory biological sequence alignment accelerator for large-scale genomic data," *arXiv preprint arXiv:1901.05959*, 2019.