Hardware Acceleration of the STRIKE String Kernel Algorithm for Estimating Protein to Protein Interactions

Fadi N. Sibai, Ali El-Moursy, Abu Asaduzzaman, and Sohaib Majzoub

Abstract—Protein-protein interaction (PPI) is an important field in bioinformatics which helps in understanding diseases and devising therapy. PPI aims at estimating the similarity of protein sequences and their common regions. STRIKE was introduced as a PPI algorithm which was able to achieve reasonable improvement over existing PPI prediction methods. Although it consumes a lower execution time than most of other state-of the-art PPI prediction methods, its compute-intensive nature and the large volume of protein sequences in protein databases necessitate further time acceleration. In this paper, we develop hardware accelerator designs for the STRIKE algorithm. Results indicate that the weighted STRIKE accelerator execution times are about 10x longer than the unweighted STRIKE accelerator execution times. To further accelerate the performance of the weighted STRIKE, a parallel module accelerator organization duplicating the weighted STRIKE modules is introduced, achieving near linear speedups for long sequences of 100 or more characters. As demonstrated by Verilog simulations and FPGA runs, the weighted STRIKE module accelerator exhibits three orders of magnitude speed improvement over multi-core and cluster computers. Much higher speedups are possible with the parallel module accelerator.

Index Terms—Bioinformatics, protein-protein interactions, hardware acceleration architectures, performance analysis

1 Introduction

Proteins are large molecules formed by long chains of amino acids and play an important role in several cellular functions. Proteins help regulate the human body's cells, organs and tissues and are important ingredients for body building and repairing. Proteins cooperate with other proteins by forming a large network of protein-protein interactions (PPIs). Estimating and predicting PPI has been the goal of scientists to understand and diagnose diseases. Many PPI methods have been developed with potential to identify functional relationships between proteins. Such techniques are however costly and very time consuming, necessitating the development of computational tools capable of automating PPI identifications. Each of the developed computational techniques to predict PPI has its own benefits and drawbacks, especially in regard to the method's sensitivity and specificity.

- *Fadi N. Sibai is with the College of Computer Engineering and Science, Prince Mohammad Bin Fahd University, Al-Khobar 31952, Saudi Arabia. E-mail: fsibai@pmu.edu.sa.*
- *Ali El-Moursy is with Computer Engineering Department, University of Sharjah, Sharjah, UAE. E-mail: aelmoursy@sharjah.ac.ae.*
- *Abu Asaduzzaman is with Electrical Engineering and Computer Science Department, Wichita State University, Wichita, KS 67260 USA. E-mail: abu.asaduzzaman@wichita.edu.*
- *Sohaib Majzoub is with Electrical Engineering Department, University of Sharjah, Sharjah, UAE. E-mail: smajzoub@sharjah.ac.ae.*

Manuscript received 7 January 2020; revised 18 August 2020; accepted 13 March 2021. Date of publication 17 March 2021; date of current version 8 August 2022.
(Corresponding author: Fadi N. Sibai.)
Digital Object Identifier no. 10.1109/TCBB.2021.3066591

Among the techniques for predicting PPI are the Association Method (AM) [1], Maximum Likelihood Estimation (MLE) [2], Maximum Specificity Set Cover (MSSC) [3] and Domain-based Random Forest (DRF) [4]. In domain-based methods, molecular interactions are typically mediated by a wide variety of interacting domains. Other developed techniques, such as PIPE (Protein-Protein Interaction Prediction Engine) [5], assume that some of the interactions between proteins are mediated by a finite number of short polypeptide sequences. Such assumption facilitates the identification of short polypeptide sequences used repeatedly in different proteins within the cell, given that these sequences are normally shorter than the classical domains. However, identifying domains or short polypeptide sequences is lengthy and computationally intensive. Domain-based methods and methods depending on short polypeptide sequences are also not universal because the accuracy and reliability of these methods is dependent on the domain information of the protein partners.

STRIKE [6] is an innovative algorithm which was developed to estimate and predict protein-protein interactions. Based on the String Kernel (SK) method with good performances on text categorization [7] and protein sequence classification [8], STRIKE compares a pair of protein sequences by matching common and fixed-length subsequences occurring within this sequence pair. The string kernel is built on the kernel method introduced by [9] and [10]. The kernel computes similarity scores between protein sequences, one score for each pair, without extracting the features. A subsequence is defined as any ordered sequence of amino acids occurring in the protein sequence, where the amino acids

are not necessarily contiguous, for instance, G _ _ T, GT, and G _ T. The subsequences are weighted by an exponentially decaying factor of their full length in the sequence, hence emphasizing those occurrences that are more contiguous, i.e., with shorter gaps between the subsequence's characters. Subsequence similarities between two protein sequences may not necessarily indicate interaction, but reveal a high probability for interaction occurrences and helps in inferring homology.

Multi-core and many-core [11] computers are now present in a multitude of hardware vendor products packing tens to hundreds of processing cores per chip and providing hundreds of GFLOPS of performance per Watt. Programming languages such as CUDA [12] and OpenCL [13] support the development of code on many-core GPUs. In [6], we developed a multithreaded version of STRIKE on multicore systems. In [14], we also developed and studied the performance of parallel STRIKE versions on multicore computers and Message Passing Interface Standard (MPI)-based clusters. On long protein sequence sets, the execution time of a parallel implementation of this bioinformatics algorithm was reduced from about a week on a serial uniprocessor x86 laptop to about 2 hours on 128 parallel cluster nodes. This parallel implementation was shown to scale very well with increasing data size and number of nodes. While the parallel implementation on multi-cores and computer clusters greatly improved STRIKE's execution time, because of the huge number of sequence matchings required in large protein databases, we seek further performance improvements and present, in this paper, two different, unweighted and weighted, hardware versions of the STRIKE algorithm.

This paper is organized as follows. In Section 2, we survey the PPI literature. In Section 3, we briefly explain the serial version of the STRIKE algorithm. Section 4 reviews the parallel STRIKE algorithm on multi-core computers. In Section 5, we review the multithreaded and message-passing implementations of STRIKE and summarize their performance. In Section 6, we present the unweighted and weighted STRIKE hardware accelerator designs and parallel module organization for the Weighted version. Section 7 presents the performances of the three hardware accelerator designs, unweighted, weighted and parallel module organization, and compares them to multithreaded and message-passing versions on multicore computers and computer clusters, respectively. Results of the Verilog simulation and FPGA implementation of the weighted STRIKE are also presented. The paper concludes in Section 8.

2 Related Work

According to [25], protein-protein interactions (PPIs) can modify the properties of enzymes, play a role in substrate channeling, inactivate a protein, or change the specificity of a protein. For these reasons, PPI assists in the identification of drug targets. Many databases support PPI work such as BIND, BioGRID, and MIPS [25]. Prior methods for identifying PPIs are both time-consuming and expensive, including domain knowledge methods, such as AM [1], MLE [2], MSSC [3] and DRF [4], and methods which focus on short polypeptide sequences such as PIPE [5]). This is exacerbated by the large number of very long protein sequences to match in protein databases. Other novel PPI methods, such as STRIKE [6], use only the information of protein sequences and have been parallelized [6], [14], but have potential to be further accelerated by dedicated and customized hardware accelerators.

Comparing the sensitivity and specificity of these PPI methods using the same source of data is an effective way to compare their performance. Sensitivity is the percentage of protein pairs that are recovered using a certain keyword or a group of keywords when they are applied back to the training dataset. Specificity, or precision, is the percentage of protein pairs recovered when keywords are applied to the predicted (i.e., the test) datasets. On Pfam [15], a protein domain family database, domain-based methods yielded 39 percent specificity and 79.7 percent sensitivity [8] outperforming PIPE, which achieved 61 percent sensitivity, 89 percent specificity and 75 percent overall accuracy [5]. STRIKE achieved the highest performance results among all the mentioned algorithms above, scoring 89 percent accuracy, 83.1 percent specificity, and 98 percent recall sensitivity [16]. Another novel algorithm [17] which examines information of protein sequences is based on Extreme Learning Machine and uses a novel representation of local protein sequence descriptors. When tested on the PPI data of Pfam, the extreme learning machine method achieved 89.09 percent prediction accuracy, 89.25 percent sensitivity, and 88.96 percent specificity, hardly an improvement over STRIKE.

Another PPI method [18] represents each protein sequence by a vector of pairwise similarities against large subsequences of amino acids created by a shifting window which passes over concatenated protein training sequences. One major drawback of this method is that each protein is represented by computing the Smith-Waterman (SW) score against a large subsequence created by concatenating protein training sequences. In fact, comparing short sequences to very long ones results in some potentially valuable alignments to be missed. However, the SK handles this weakness by capturing any match or mismatch which exists in the protein sequence of interest.

Beyond algorithm diversity, hardware acceleration has been experimented on PPI algorithms. Field Programmable Gate Arrays (FPGA) acceleration of the HMMER bioinformatics application was discussed in [19] and shown to perform competitively with respect to a multithreaded version of HMMER on quad-core Xeon computer. Local alignment of protein sequences is performed by the SW algorithm to obtain similarity results of common regions within the sequence pairs. Various FPGA-based accelerations of the SW algorithm were presented in [20]–[23]. Other bioinformatics applications on GPU include BLAST [24]. A GPU-based implementation of the SW algorithm [25] on nVIDIA Pascal GPU expedited its execution time by over 300 percent over an older implementation on nVIDIA Kepler. Hence, bioinformatics algorithms for pairwise similarity can greatly benefit from acceleration by customized hardware, multicore and manycore processors, GPUs, and FPGAs. While GPUs and FPGAs can greatly result in impressive speedups, customized hardware such as ASIC design usually results in the highest speedups.

3 Serial Implementation of the STRIKE Algorithm

In this section, we briefly describe the serial basic version of STRIKE [6] using s1 = lql and s2 = lqal, two strings of characters representing two short protein sequences. Each sequence is decomposed into a number of substrings or subsequences. The shortest substring to match is two characters long. In other words, these sequences are implicitly transformed into feature vectors, where each feature vector is indexed by two characters which need not be consecutive. Next, we decompose each of the two sequences into 2-character substrings. Each sequence is decomposed into all possible ordered (from left to right) combinations of characters included in the sequence, as follows.

	lq	ll	ql	la	qa	al
s1 = lql	λ^2	λ^3	λ^2	0	0	0
s2 = lqal	λ^2	λ^4	λ^3	λ^3	λ^2	λ^2

Above, the 2-character substrings (lq, ll, and ql) represent the decomposition of the s1 sequence, while all six 2-character substrings represent the decomposition of the second sequence s2. When a 2-character substring appears in a sequence such that these 2 characters are consecutive in the sequence, such as lq, the substring's degree of matching (DOM) in that sequence equals λ^2, where λ is a decay factor less than 1. When these 2 characters are separated by another single character (gap of 1), such as ll in the first sequence, the substring's DOM is λ^3. Generally, when the two characters in the appearing substring are further spaced by *gap* characters, the DOM is represented by λ^{2+gap}. The DOMs for the substrings of the first sequence and all substrings for the second sequence are computed in that fashion as shown above. The 2-character substrings to match in the two sequences must appear in both sequences in the exact same order, but the gap spacings between the characters of the 2-character substrings need not match in the two sequences. For instance, l _ q in one sequence matches lq, l _ q, l _ _ q, and l _ _ _ q in the second sequence, but not ql. To reflect the degree of matching between the s1 and s2 sequences, the un-normalized string kernel (SK) for the above 2 sequences, k(lql, lqal), is computed as the dot product of all DOMs, i.e., $\lambda^2\,\lambda^2 + \lambda^3\,\lambda^4 + \lambda^2\,\lambda^3 + 0\,\lambda^3 + 0\,\lambda^2 + 0\,\lambda^2 = \lambda^4 + \lambda^7 + \lambda^5$. For a decay factor λ of 0.5, k(lql, lqal) = 0.102. In general, the higher the un-normalized kernel is, the higher are the indication of matching and interaction between the two sequences.

As shown in Fig. 1, the serial STRIKE implementation consists of a main procedure which reads the input protein sequences, one from the training sequence set and the other from the testing sequence set. Then for each sequence in the training and testing sequences sets, a pairing operation is done for each amino acid with a subsequent amino acid in the same sequence. This generates 2-character substrings. Afterwards, a matching step is conducted, as in the previous example, between each protein sequence in the training set and all sequences found in the testing set. This matching step generates the pairs of matching amino acids and their inter-distances, and computes the score matrix. The score matrix contains at row i and column j a number representing the SK score for matching sequence i of the training set with sequence j from the testing set.

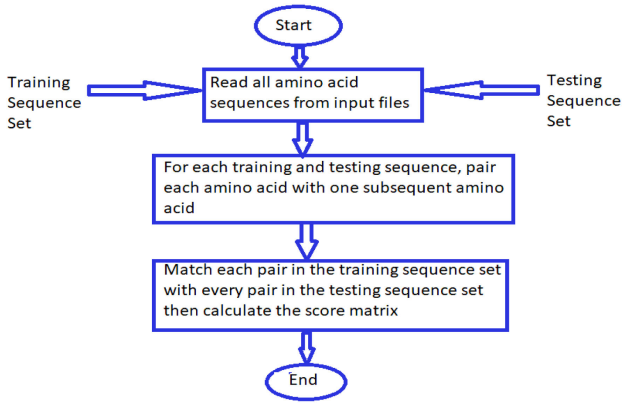

Fig. 1. Flowchart of the serial STRIKE application.

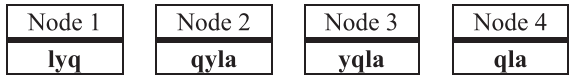

Fig. 2. Allocation of sequences to processing nodes.

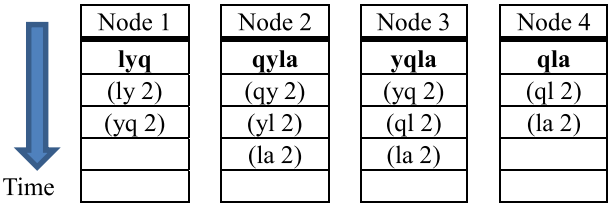

Fig. 3. Decomposition of each protein sequences into substrings of Length = 2 and Gap = 0.

4 Parallel STRIKE Algorithm

Given the large number of 2-character strings in long protein sequences, STRIKE is compute-intensive. To speed up the execution of this algorithm, a parallel STRIKE algorithm [6] was developed consisting of the following 3 steps: (i) Decomposition; (ii) Sorting; (iii) Inner Product.

As shown in Fig. 2, in the decomposition step, the amino acid sequences are first allocated to processing nodes, one sequence per node. A node could be a core in a multi-core processor, or a computing node in a parallel computer with its own CPU. We will refer to either core or node by node. Assuming that the SKs of the 4 amino acid sequences lyq, qyla, yqla and qla are computed on four cores to assess mutual interactions between these 4 sequences.

In parallel in the four nodes, the decomposition of each protein sequence is conducted. Each sequence is decomposed into 2-amino acid substrings starting with adjacent amino acids, i.e., those with a gap of 0, as shown in Fig. 3. The "2" in (ly 2) refers to the power of the weighted decay factor (i.e., λ^2), indicating no gap between the 2 characters.

Given that the contiguity of the amino acids in the resulting substring is not necessary, the decomposition proceeds for all substrings of gap 1, as shown by Fig. 4.

The "3" in (lq 3) refers to the power of the weighted decay factor (i.e., λ^3), meaning that the l and q are separated

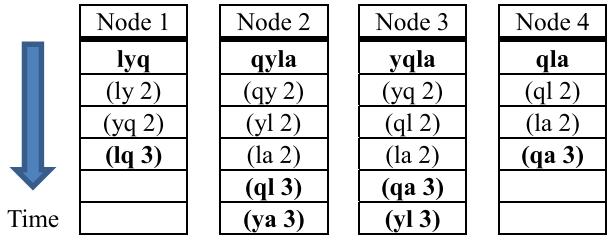

Fig. 4. Decomposition into substrings of Gap = 1.

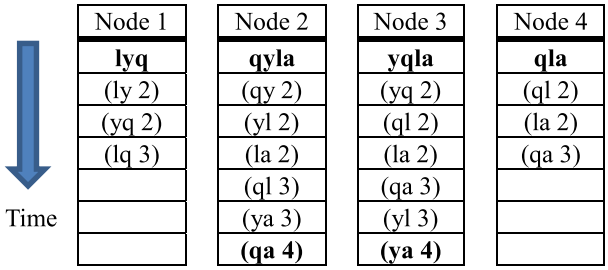

Fig. 5. Decomposition into substrings of Gap = 2.

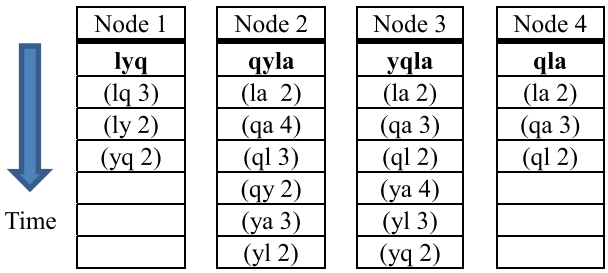

Fig. 6. Content of the nodes after sorting.

by another amino acid y in the sequence lyq. Next, the decomposition into substrings with a gap of 2 characters occurs, as shown in Fig. 5.

Nodes 1 and 4 complete the decomposition step ahead of processing nodes 2 and 3, owing to their shorter length and immediately proceed to the sorting step, to be later followed by nodes 2 and 3. In the sorting step, the 2-amino acid substrings generated in the decomposition step are sorted in parallel, and alphabetically based on their 2-letter string content. After step 2 completes, the 4 nodes will have for content the sorted strings shown in Fig. 6.

In the inner product step, the inner products are carried out on half the nodes (2 in our example) with the largest substring set cardinality. This choice is made to minimize the total inter-node communication time. In our example, nodes 2 and 3 have the highest number of generated substrings. Each of these nodes maintains its 2-amino acid substrings and receives 3 amino acid strings generated by the node which is allocated the other sequence to match with its sequence. As a result, data communications occur as shown in Fig. 7.

During the inter-node communication step, node 1 sends its generated 2-amino acid substrings to node 2, and node 4 sends its generated 2-amino acid substrings to node 3. After the communication done, the contents of each node will be as shown in Fig. 8.

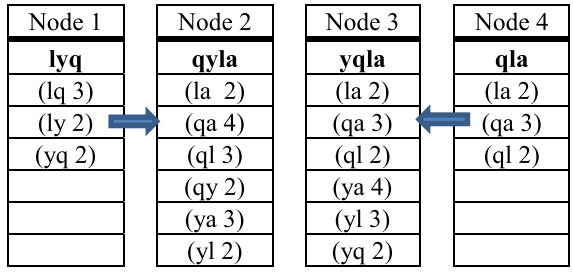

Fig. 7. Inter-node communications.

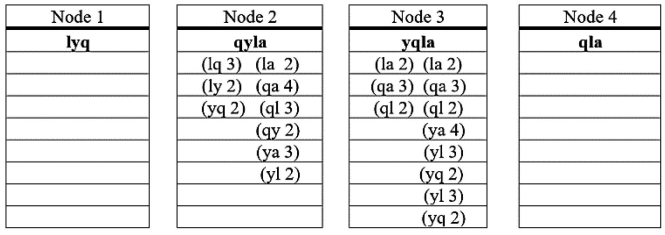

Fig. 8. Contents of each node after inter-node communications.

Node	Result
2	0
3	4, 6, 4: $\lambda^4 + \lambda^6 + \lambda^4 = 2\,\lambda^4 + \lambda^6$

Fig. 9. Results of inner products on nodes 2 and 3.

Afterwards, nodes 2 and 3 calculate the inner products between their strings generated in the decomposition step and the received strings generated by the neighboring node, as shown in Fig. 9. The inner product (α n). (β m) succeeds when the 2 strings match i.e., $\alpha = \beta$, producing the number n+m (representing λ^{n+m}). Otherwise when α is different from β, it is a mismatch (resulting in 0). In our example, there are no string matches in Node 2. Node 3 produces the following inner product matches

$$(\mathrm{la2}) \,.\, (\mathrm{la2}) \;\rightarrow 4$$
$$(\mathrm{qa3}).\, (\mathrm{qa3}) \rightarrow 6$$
$$(\mathrm{ql2}) \,.\, (\mathrm{ql2}) \;\rightarrow 4$$

At the end of the string matching, the results are presented by each node involved in the inner product step as follows:

The results on nodes 2 and 3 are then aggregated in node 3 to provide the final score. The parallel STRIKE algorithm is capable of matching as many sequences in parallel as desired based on the availability of processing nodes, achieving excellent performance scalability with increasing hardware resources.

5 Multithreaded Implementation on Multicores and Message-Passing Implementation on Clusters

Experimentation with the parallel STRIKE led to a number of performance enhancements. To improve the matching accuracy, we modify the SK to be the weighted inner product of the DOMs (α n). (β m), i.e., from λ^{n+m} to λ^{n+m} x matrix(c1) x matrix(c2), where c1 and c2 are the first and second characters appearing in the matching substrings $\alpha = \beta = ''\mathrm{c1\,c2}''$, and matrix(c1) is a weight given to characters, such that characters A, B, C ... can be assigned

different weightage. These weights help in directing the matching towards specific desired character combinations, resulting in the SK score emphasizing these combinations. This will be referred to as the weighted version of STRIKE, while the procedure described in Section 4 without weights is referred to as the unweighted STRIKE.

The parallel multithreaded implementation of the weighted STRIKE algorithm consists of a main procedure which reads the input protein sequences, one from the training set and the other from the testing set, and amino acid weight matrix, launches parallel jobs which are assigned an equal number of sequences to match and which generate the pairs of amino acids and their inter-distances and computes the portion of the SK score matrix corresponding to the sequences assigned to these jobs. The weight matrix contains all amino acid weights corresponding to all characters in the protein sequence.

For a pair of short protein sequences [6], the code was optimized as follows. Initially the multithreaded application took 4.5 minutes on a standard PC to process a test set of 168 protein sequences with an 84–sequence training set. This was reduced to 50 seconds with compiler options related to fast code generation and single-instruction multiple-date vectorization (SSE2 SIMDization) and further down to 2 seconds by limiting the pairs of characters in the 2-character substring to those with gaps not exceeding 8 protein characters in step 2, and only matching protein pairs in both sequences if their resulting distance $\mathrm{dist}_{i,j}$ does not exceed 8. As λ is raised to the power $\mathrm{dist}_{i,j}$, and λ is less than1, when $\mathrm{dist}_{i,j}$ exceeds 8, $\lambda^{\mathrm{dist}\,i,j}$ becomes negligible. Hence, ignoring amino acid pairs at such large inter-distances saved a lot of computation time while not compromising the accuracy of the protein sequence matching.

Furthermore, the performance of the parallel STRIKE was evaluated [14] on dual-core and quad-core computers and on a 16-desktop computer heterogeneous cluster and a 128-node SUN computer cluster. On long protein sequence sets, it was shown that the execution time of the parallel implementation of STRIKE was reduced from about a week on a serial uniprocessor machine to about 16.5 hours on 16 computing nodes, down to about 2 hours on 128 parallel nodes. The PC cluster with 128 nodes consumed a long communication time compared to a lower number of nodes, but it scaled the computation time near linear.

Despite very good scalability of both shared memory and message passing implementations, it is desirable to further accelerate the execution time to handle more sequence matchings and longer sequences, given the huge sequence databases and daily sequence matching volume. In the next Section, we describe hardware accelerator designs for the STRIKE algorithm unweighted and weighted versions (not based on parallel algorithm of Section 4).

6 Hardware Accelerator Design

The hardware accelerator design of the STRIKE algorithm places the two amino acid sequences to be matched in two one-directional shift left registers, as shown in Fig. 10. The second register holding the second string has additional circular shift capability, i.e., after shifting, the most significant character becomes the least significant one.

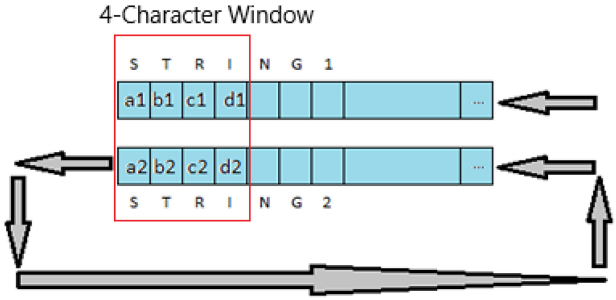

Fig. 10. The 2 strings for matching in shift registers.

Four character pairs in a 4-character window on the left side of the two shift registers are considered at a time in each iteration. This assumes that a matching of such two 2-character strings where the characters have a maximum gap of 2 in each string will produce a minimum partial score of λ^3. $\lambda^3 = \lambda^6$, and that partial scores which are multiple of λ^7 or smaller (λ^8, λ^9, ...) are negligible (recall that $\lambda < 1$). Circular-shifted characters in STRING 2 no longer participate in the 4-character window matching.

The leftmost character in String 1, a1, is matched, one by one, with all the characters of String 2, starting with a2, and the partial scores are aggregated. This requires String 1 to freeze in its position while the circular shift register holding String 2 makes a full counterclockwise rotation (N shifts total for an N-character string), where after each shift, a1 b1 c1 d1 is matched with the 4-character string in the window below it. After shifting all the characters of String 2, a2 reaches back its position as shown in Fig. 10 and the matching of a1 with all the characters of String 2 is over. Next, the top shift register holding String 1 shifts one time to the left, resulting in b1 occupying a1's position. The same matching steps, as above, repeat between b1 c1 d1 e1 and a2 b2 c2 d2, then b1 c1 d1 e1 and b2 c2 d2 e2, ... until all of String 2 has made a full rotation.

This 4-character window of Strings 1 and 2 facilitates the matching of a1 b1 c1 d1 of String 1 with a2 b2 c2 d2 of String 2, and limits the hardware cost. Wider windows increase the accuracy but require more logic and increase the hardware size and cost. The various combinations of 2-character substring matches resulting from this window size, and these two 4-character strings and their unweighted partial scores (UPS, unweighted version) and weighted partial scores (WPS, weighted version) are shown in the Table 1 of the Supplement.

6.1 Hardware Design of the Unweighted STRIKE

The unweighted STRIKE hardware accelerator design is illustrated in Fig. 11. It assumes 8-bit ASCII characters and uses eight 8-bit comparators to simultaneously compare a1 with a2, b1 with b2, b1 with c2, b2 with c1, b1 with d2, c1 with c2, and b2 with d1. If b1 and b2 are different, a1b1 and a2 b2 do not match. Similarly, if d1 is different from b2, this means that a1d1 and a2b2 do not match.

If a1 is different from a2, the 4-character window matching is immediately terminated and the bottom shift register holding String 2 is shifted once to the left. This is repeated until the leftmost characters in the two shift registers match, i.e., a1 = a2.

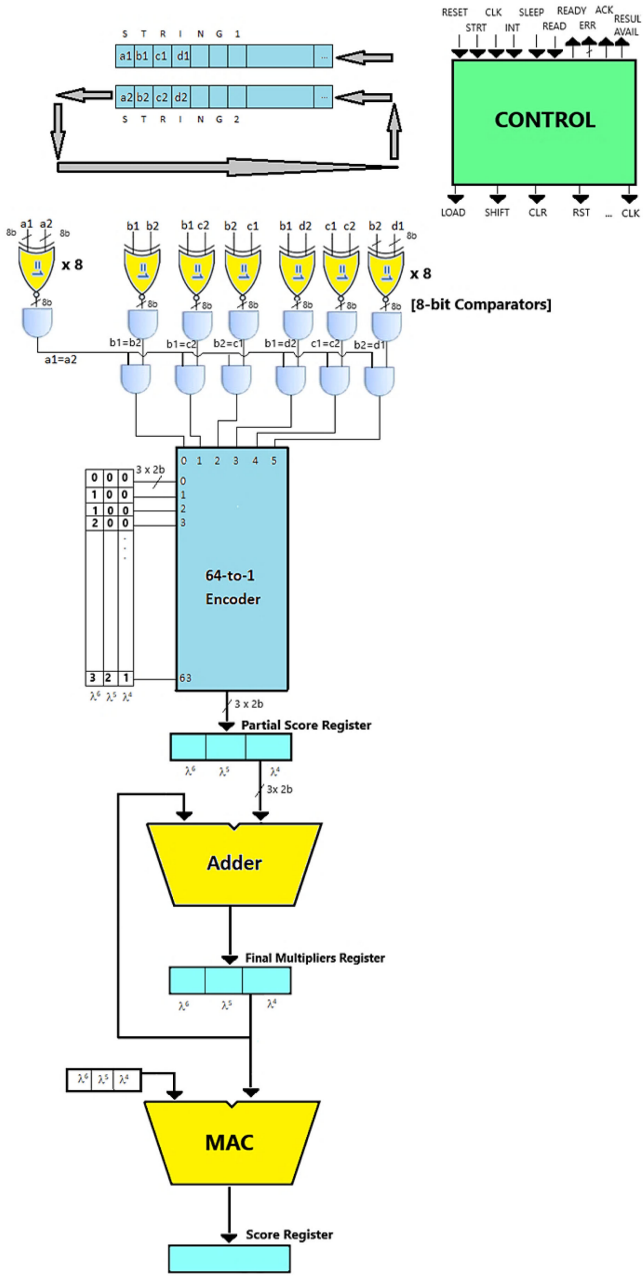

Fig. 11. Datapath for the unweighted STRIKE accelerator.

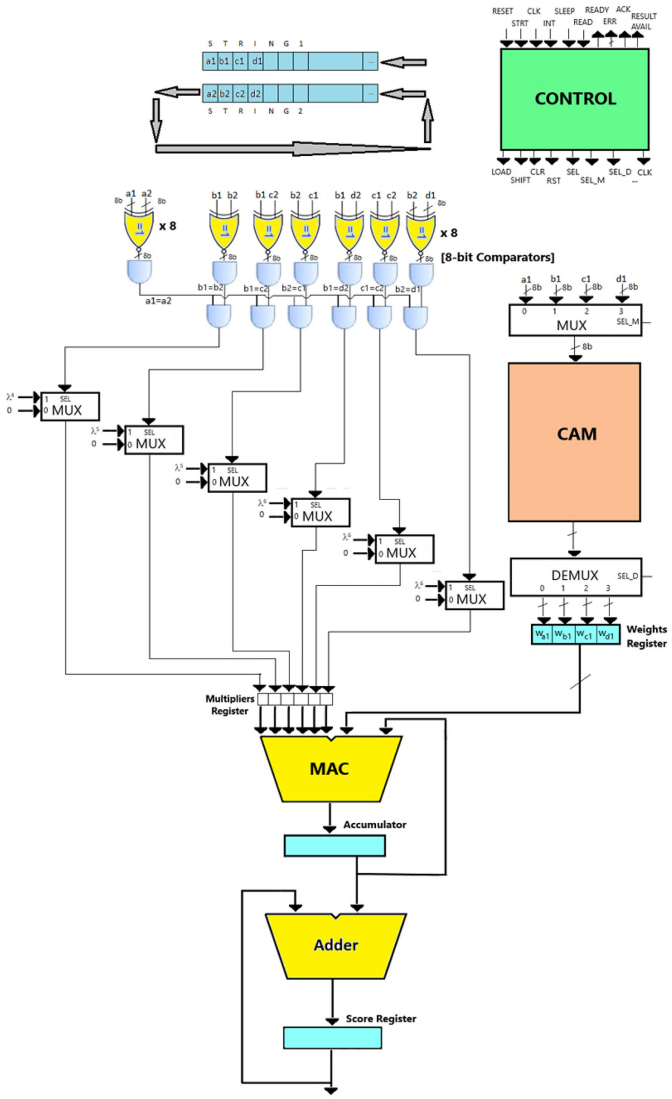

Fig. 12. Datapath for the weighted STRIKE accelerator.

If a1 and a2 match, the results of the comparators matching, b1 with b2, b1 with c2, b2 with c1, b1 with d2, c1 with c2, and b2 with d1, feed the select inputs of a 64x1 encoder. The data inputs of the encoder are derived from the 7th column of Table 1 of the Supplement. These encoder inputs are hardwired as three 2-bit numbers which represent the multipliers of λ^6, λ^5, and λ^4, respectively. For instance, for input 63 (bottom most input of the encoder, and bottom most row of Table 1 of the Supplement), $3\,\lambda^6 + 2\,\lambda^5 + \lambda^4$ produces 32 1. A Partial Score Register holds these encoder output values. An adder then accumulates the saved multipliers of the λ^6, λ^5, and λ^4, held in the Final Score Register, with the encoder output read values held in the Partial Score Register. This keeps going until all the characters of String 2 have been matched with the characters of String 1: String 2 rotating circularly, and String 1 shifting by one character to the left. Note that this integer adder adds 3 integers with 3 other integers given that the decay factor λ has not been introduced yet in the computation. When the saved multipliers have been finalized at the end of the 2 string matching, the final multipliers are fed to a floating point multiply-add circuit (MAC) to multiply the λ multipliers with the λs, then add the terms to generate the final score. These long floating-point multiplications happen only once at the end of the 2-character string matchings.

The MAC includes 3 partial product registers (not pictured in Fig. 11) to hold the partial products during the MAC operations. The contents of the partial products are then multiplied by the remaining multiplicands in the right column equations of Table 1 of the Supplement to create the full product terms which are then added together.

6.2 Hardware Design of the Weighted STRIKE

The Weighted STRIKE controls the matching of specific protein sequences. The weighted STRIKE hardware accelerator design is illustrated in Fig. 12. The weighted version proceeds exactly as the start of the unweighted version. It also assumes 8-bit ASCII characters and uses 88-bit comparators

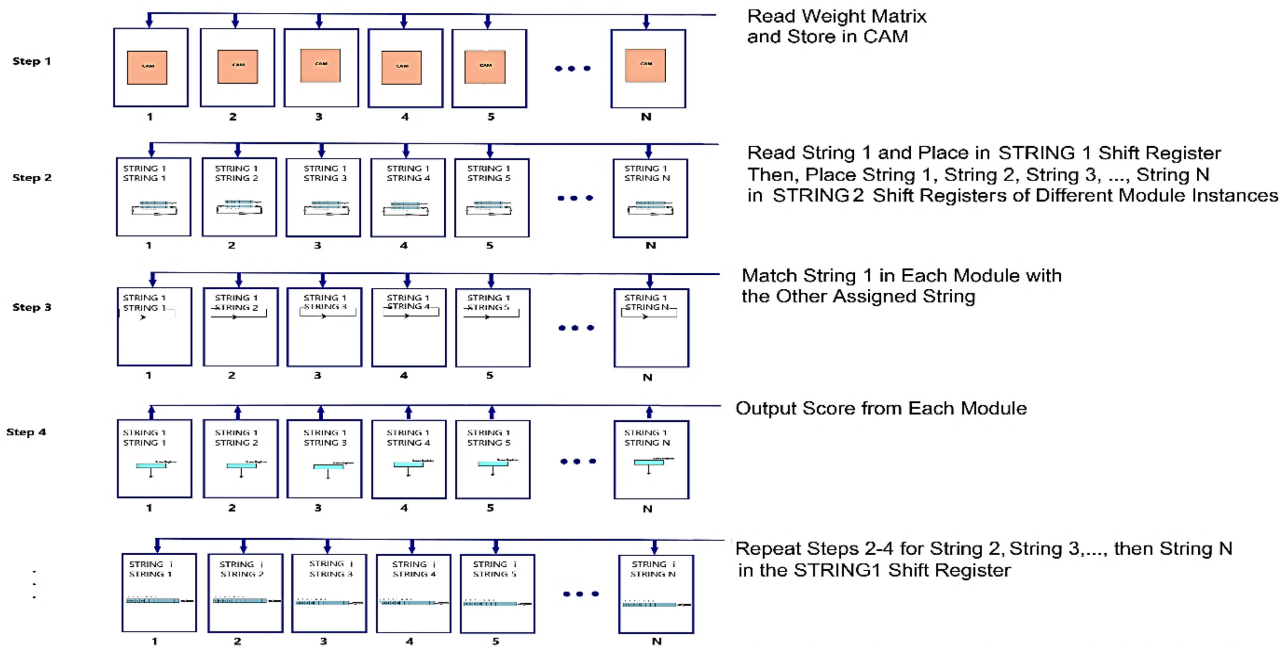

Fig. 13. Parallel module organization for further enhanced performance.

to simultaneously compare a1 with a2, b1 with b2, b1 with c2, b2 with c1, b1 with d2, c1 with c2, and b2 with d1. If a1 is different from a2, the 4-character window matching is terminated and the top shift register holding String 2 is shifted to the left. This is repeated until the leftmost characters in the two shift registers match, i.e., a1 = a2.

If a1 and a2 match, the results of the comparators matching b1 with b2, b1 with c2, b2 with c1, b1 with d2, c1 with c2, and b2 with d1, feed the Select inputs of six multiplexers (MUX). The multiplexers choose between a power of λ as in row number 2 of Table 1 of the Supplement) when the characters match, and 0 otherwise. The MUX outputs are then stored in the Multipliers register. Simultaneously as the comparator and MUXes are operating, a content addressable memory (CAM) is searched with the ASCII characters for a1, b1, c1, and d1, to retrieve the weights for these characters, which are then stored in a Weights register. As the weights and the powers of λ are floating point and not integers, the hardware above the CAM is integer, while the CAM and λ MUXes and all circuits below support floating point data.

A Multiply Add/Accumulate (MAC) circuit then multiplies the powers of λ (λ^6, λ^5, λ^4) in the Multipliers register by the weights in the Weights Register to calculate the first term in the equation given in the rightmost column of Table 1 of the Supplement, stores the partial sum in the accumulator, computes the next term in the equation in Table 1 of the Supplement, and adds it to the accumulated sum, and repeats to fully perform all the MAC operations for all the terms of the equation in the rightmost column of Table 1 of the Supplement. This MAC saves the partial products in a temporary product register (not pictured in Fig. 12) to hold the partial products of the partial multiplications.

An adder then adds the sum in the accumulator with the content of the Score register to accumulate the partial sums from previous match operations. This keeps going until the String 2 register makes a full rotation and all characters of String 2 have been matched with the 4-characted window of String 1 starting with a1. Then String 1 register shifts to the left and b1 becomes the new a1. The previous operations are repeated until all characters of String 1 register have been matched with all characters of the String 2 register, and all partial sums accumulated to generate the final score.

6.3 Parallel Modules Organization

To raise the weighted STRIKE performance to the next level, parallel identical instances of the weighted STRIKE hardware module of Fig. 12 can be organized and operated together as illustrated in Fig. 13. We illustrate this organization with N copies of the Weighted STRIKE hardware modules, assuming that both training and testing sets each contain N sequences. A shared bus connects the N parallel modules to a shared external memory (not pictured). An identical sequence from the training set will be placed in the STRING 1 registers of all hardware modules, while STRING 2 registers of all modules will each hold a different sequence from the testing set. This organization training set with all sequences of the testing set. Once this is done, another sequence from the training set replaces the existing sequence in the STRING 1 registers of all hardware modules, and this repeats until each sequence of the training set has had its turn and has been simultaneously matched with all sequences of the testing set.

In step 1 of Fig. 13, the Weights matrix containing the weights for each character is read by the modules over the shared bus and stored in their CAMs. In step 2, String 1 of the training set is read by the modules and stored in their STRING1 registers. The N sequences of the testing set are then read from memory (not pictured) by the hardware modules, one sequence per module, such that sequences 1, . . ., N are placed in the circular STRING 2 shift registers of Modules 1, . . ., N, respectively. In step 3, each module. In step 3, each module matches the contents of its STRING 1 register, holding the first sequence of the Training set, with the sequence

TABLE 1
Unweighted STRIKE Hardware Times

Step	Component	Time (clocks)
1	Shift Register (load/shift)	1
2	Load the 2 strings (4 clocks memory write, and 1 clock to shift into Shift Register) from external memory over the shared bus	(N1+N2) × (14+1) = 15 (N1+N2)
3	Comparator	1
4	Encoder	1
5	Partial Score Register (load)	1
6	Adder	1
7	Final Multipliers Register (load)	1
8	FP MAC (3 multiplies, 2 adds, write into product registers)	3 × (4+1)+2 × (2+1) = 21
9	Score Register (load)	1
10	Save Score in Score Register into External Memory	14

stored in its STRING 2 register as described in Section 6.2. In step 4, each module outputs its final score to be saved in memory to form the final score matrix.

The next steps repeat Steps 2–4 except that in Step 2, the second sequence of the training set is read and placed in the STRING 1 register of all modules. When step 4 completes for the second time, step 2 is repeated with the third sequence of the training set being read by all modules and placed in the String 1 registers. This keeps going until all N sequences of the training sets have been matched in all hardware modules by all sequences of the testing set, one testing set sequence per hardware module, and the complete final score matrix components written in memory.

7 Performance Evaluation

7.1 Unweighted STRIKE

The performance of the unweighted STRIKE accelerator of Fig. 11 is evaluated in this Section for 2 sequences String 1 of size N1 and String 2 of Size N2. The following hardware component times are assumed based on 1 GHz commercial chips.

After the sequences are loaded, there are $(N1 - 1) \times (N2 - 1)$ computation iterations, as at the end of the iterations, at least 2 characters must remain in each shift register to make a successful 2-character string match. Each computation iteration consumes 6 clocks (Steps 3–7, and 1 in Table 1). Therefore,

$$\text{Computation iterations time} = 6(N1 - 1)\ (N2 - 1) \tag{1}$$

After adding the iteration times to the times of steps 8, 9, 10 and 1, the total time, in clocks, for the Unweighted STRIKE accelerator is

$$\begin{aligned}\text{Total time}_{US} &= 15(N1 + N2) + 6(N1 - 1)\ (N2 - 1)+21 + 1 + 14\\ &= 15(N1 + N2)+6(N1 - 1)\ (N2 - 1)+36\end{aligned} \tag{2}$$

7.2 Weighted STRIKE

The performance of the weighted STRIKE accelerator of Fig. 12 is evaluated in this Section for 2 sequences String 1 of size N1 and String 2 of Size N2. The following component times are assumed.

TABLE 2
Weighted STRIKE Hardware Times

Step	Component	Time (clocks)
1	Shift Register (load/shift)	1
2	Load the 2 strings (14 clocks memory write, and 1 clock to shift into Shift Register) from external memory over the shared bus	(N1+N2) × (14+1) = 15 (N1+N2)
3	Comparator	1
4	Multiplexer	1
5	Multiplier Register (load)	1
6	CAM Mux	1
7	CAM (write/search)	4
8	CAM DeMux	1
9	Weight Register	1
10	FP MAC (average of 3 partialproducts: 6 multiplies and 2 adds, and load into temporary product register)	6 × (4+1) + 2 × (2+1) = 36
11	Accumulator (load)	1
12	FP Adder	2
13	Score Register (load)	1
14	Save Score in Score Register into External Memory	14

The 22 CAM entries for the 22 protein characters must be initially loaded in the CAM (memory read and steps 6–7), consuming 22 × (14 clocks for external memory read + 1 + 4) = 418 clocks. After the sequences are loaded, there are $(N1 - 1) \times (N2 - 1)$ computation iterations as at the end of iteration at least 2 characters much remain in each shift register to make a successful 2-character string match. Each computation iteration, the following steps take place concurrently:

a. Comparing the 2 strings in the window, MUX to choose power of λ or 0, save power of λ or 0 in the Multipliers Register (steps 3–5). This step consumes 1+ 1 +1 = 3 clocks; and
b. Search and read the weights of a1, b1, c1, and d1 from the CAM, and save the Weights in the Weights Register (steps 6-9). This step consumes 1 +4 + 1 + 1 = 7 clocks for each character, or 28 clocks for 4 characters.

Both steps a and b take max (3, 28) = 28 clocks. Although the STRING1 weights do not change during the circular shifts of the STRING2 register, we assume worst case time of 28 clocks for the concurrent operations of steps a and b. The MAC operation time depends on which row of Table 1 of the Supplement is involved. On average, there are 3 terms in the right side equations of Table 1 of the Supplement. Therefore, the average MAC time is 6 (4+1) + 2 (2+1) = 36 clocks per iteration. Accounting for Table 2 steps 11–13 and 1, the average time per iteration, in clocks, is given by

$$\begin{aligned}\text{Computation iteration time} &= (28 + 36 + 1 + 2 + 1 + 1)\\ &\quad \times (N1 - 1)\ (N2 - 1)\\ &= 69(N1 - 1)\ (N2 - 1)\end{aligned} \tag{3}$$

After adding the iteration times to the times of step 14, the CAM loading time, and the string load time (step 2), the total time for the Weighted STRIKE accelerator, in clocks, is

TABLE 3
Unweighted and Weighted STRIKE Times (clocks)

		UNWEIGHTED	UNWEIGHTED	WEIGHTED	WEIGHTED
		w/o mem.	with mem.	w/o mem.	with mem.
N1	N2	Total time (clocks)	Total Time (clocks)	Total time (clocks)	Total Time (clocks)
10	10	522	822	5589	6321
100	10	5382	7032	61479	63561
500	10	26982	34632	309879	317961
1000	10	53982	69132	620379	635961
10	100	5382	7032	61479	63561
100	100	58842	61842	676269	679701
500	100	296442	305442	3408669	3418101
1000	100	593442	609942	6824169	6841101
10	500	26982	34632	309879	317961
100	500	296442	305442	3408669	3418101
500	500	1494042	1509042	17181069	17196501
1000	500	2991042	3013542	34396569	34419501
0	1000	53982	69132	620379	635961
100	1000	593442	609942	6824169	6841101
500	1000	2991042	3013542	34396569	34419501
1000	1000	5988042	6018042	68862069	68892501

$$\begin{aligned}\text{Total time}_{\text{WS}} &= 15(\text{N1}+\text{N2})+69(\text{N1}-1)\ (\text{N2}-1)+14+418\\ &= 15(\text{N1}+\text{N2})+69(\text{N1}-1)\ (\text{N2}-1)+432\end{aligned} \tag{4}$$

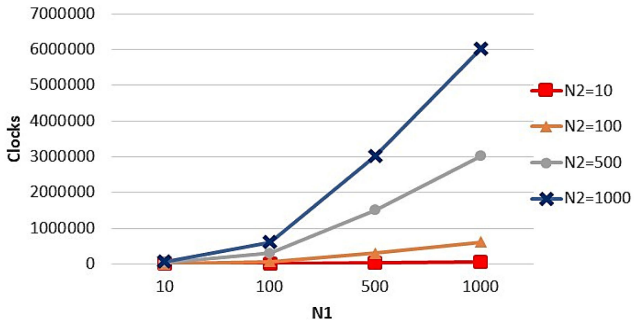

Fig. 14. Total time of the unweighted STRIKE.

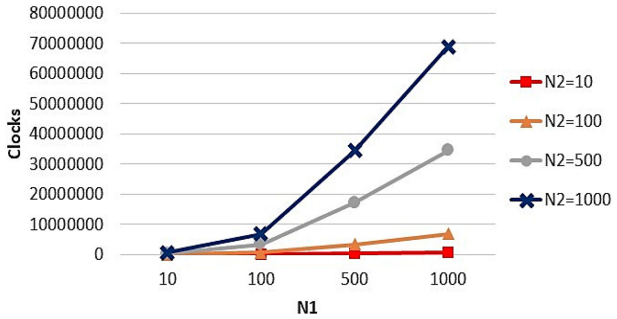

Fig. 15. Total time of the weighted STRIKE.

7.3 Parallel Modules

The performance of the Parallel Module accelerator design based on the Weighted STRIKE of Fig. 13 is evaluated in this Section. As in Eq. (4), 418 clocks are required to initially load the CAM weights. One sequence is loaded simultaneously onto the STRING 1 register of all N parallel modules. Different sequences are then loaded onto STRING2 of each parallel module sequentially over the shared bus from external memory. Therefore, the sequence load time, in clocks, assuming no bus contention, is given by

$$\text{Sequence load time} = 15\ \text{N1} + 15\ \text{N}\ \text{N2} \tag{5}$$

The computation iterations time 69 (N1 − 1) (N2 − 1) is consumed by N parallel modules simultaneously. Also, unlike in Eq. (4), N external memory writes are sequentially made at the end to save the Score Registers from all N modules in external memory. Therefore, the total time for the parallel modules, in clocks, again assuming no bus contention, is given by

$$\begin{aligned}\text{Total time}_{\text{PM}} &= 15(\text{N1}+\text{N}\ \text{N2})+69(\text{N1}-1)\ (\text{N2}-1)\\ &\quad +14\text{N}+418\end{aligned} \tag{6}$$

This time is compared to performing the Weighted STRIKE operation serially (Eq. (4)) N times.

7.4 Performance Evaluation

The performance of the three designs is addressed in this Section. Table 3 displays the total times with and without loading input data and writing output data in external memory for the Unweighted (Eq. (2)) and Weighted (Eq. (4)) STRIKE.

The total times of the Unweighted and Weighted STRIKE accelerator are plotted in Figs. 14 and 15, respectively, for various values of N1 and N2. The weighted STRIKE times are about 10 times longer than the unweighted STRIKE times.

For long sequences with N2 = 1000, the plots of the total times of the Parallel Module accelerator with N = 8–64 almost overlap, matching the plot of Fig. 15 with N2 = 1000, although the amount of work, i.e., throughput, increases from 8 parallel string matches (N = 8) to 64 parallel string matches (N = 64). So the execution times are identical but the amount of work completed is higher depending on the number of parallel modules in operation, resulting in higher throuputs with more parallel modules in operation.

The speedup of the parallel module accelerator over the single Weighted STRIKE accelerator is given below.

$$\begin{aligned}\text{Speedup} &= \text{total time}_{\text{WS}}\ \text{of N string matchings}/\text{total time}_{\text{PM}}\\ &= \text{N x total time}_{\text{WS}}/\text{total time}_{\text{PM}}\end{aligned} \tag{7}$$

Eq. (7) is evaluated in Table 4 for N of 8, 16, 32, and 64 modules, where the total time includes the time to load input data from memory and store the result in memory.

The speedup is plotted in Fig. 16 for N2 of 1000. Note that for N of 16, 32, and 64, the speedup starts below N for N1 of 10, and then eventually reaches N for N2 of 1000 and above. When the speedup is calculated for the times excluding input data loading and output data writing in external memory, and exclusively accounting for the computation time, the speedup approaches N, reaching to high sequence lengths linear scaling. For N of 8 or lower, the speedup is relatively flat irrespective of N1.

To evaluate the efficiency of the parallel module accelerator with N modules, we convert the execution times of the

TABLE 4
Speedup of N Parallel Modules Over the Weighted STRIKE

		N = 8	N = 16	N = 32	N = 64
N1	N2	Speedup	Speedup	Speedup	Speedup
10	10, 100, 500, or 1000	6.7–6.8	11.5–11.8	17.7–18.4	24.3–25.7
100	10, 100, 500, or 1000	7.8–8.0	15.4–15.5	29.6–30.0	55.1–56.2
500, or 1000	10, 100, 500, or 1000	7.9–8.0	15.9–16.0	31.5–31.8	62.0–63.1

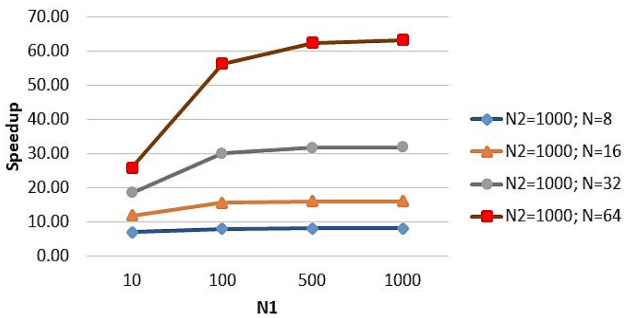

Fig. 16. Speedup of the parallel module accelerator over the single weighted STRIKE accelerator.

Weighted STRIKE of Table 3 to nanoseconds, assuming 1 GHz clock frequency, and then normalize to one character-pair matching, yielding the following Parallel Module accelerator times, in clocks, for one character-pair matching

$$\text{Time}_{\text{PM,one character pair}} = \text{Total time}_{\text{PM}}/(\text{N} \times \text{N1} \times \text{N2}) \quad (8)$$

where Total time$_{PM}$ is given by Eq. (6). The parallel module times for one character-pair matching are displayed in Table 5 for various sequence lengths (N1, N2) and for various numbers of character-pairs (N1 × N2). For equal N1 × N2 products, the N1 and N2 combinations were sorted from best to worst.

In general, we observe that for N of 8 or 16 modules, one character-pair matching times slightly degrade for higher N1 × N2 products. However, for N of 32 or 64, the reverse is true. This is a consequence of the serial external memory access during loading input data and writing the result, which becomes more significant for large N1 × N2.

Table 6 compares the average length (N1 = N2 = 100) sequence pair match time of the weighted STRIKE accelerator, running at 1 GHz, to the time of the message-passing STRIKE software version on 16-node and 128-node computer clusters [14] which a total of 14112 performed sequence pair matches (168 sequences in testing set x 84 sequences in the training set). From Table 3, the matching of an average sequence pair with N1 = N2 = 100 consumes 679701 clocks or 0.68 mseconds on the Weighted STRIKE accelerator. The hardware accelerator speeds up the average sequence pair matching by 6176 times and 750 times over 16-node and 128-node clusters, respectively, by eliminating the long cluster inter-node communication time and cluster overhead times such as operating system times. Further speedups occur for longer sequence lengths.

Furthermore, when using the parallel module accelerator, Nx additional speedups are possible to make the performance gap even wider. For instance, with N = 32 modules, 32 pairs of average length sequences will be matched in about 726635 clocks or 0.727 mseconds, i.e., 29.93 times faster than with a single module. This translates to speedups of 184K and 22K for the 32 parallel module accelerator over 16-node and 128-node clusters, respectively. Thus, the proposed designs are very promising for accelerating protein to protein interactions.

TABLE 5
Total Parallel Module Times (ns) for One Character-Pair Matching

N1	N2	N1 × N2	N = 8	N = 16	N = 32	N = 64
10	10	100	7.4	4.0	2.3	1.4
10	100	1000	7.4	3.7	1.9	0.9
100	10	1000				
500	10	5000				
10	500	5000	7.8	4.2	2.3	1.4
500	100	50000				
500	500	250000				
100	100	10000	8.0–8.1	4.0–4.1	2.0–2.1	1.0–1.1
10	1000	10000				
1000	10	10000				
100	500	50000				
1000	100	100000				
100	1000	100000				
500	1000	500000				
1000	500	500000				
1000	1000	1000000				

TABLE 6
Average Length Sequence Pair Match Times (msec.) and Speedups (N1 = N2 = 100)

Time (ms)	Time (ms)	Time (ms)	Speedup	Speedup
16-node cluster	128-node cluster	Hardware Accelerator	16-node cluster	128-node cluster
4200	510	0.68	6176	750

7.5 Verilog Simulation and FPGA Implementation

Note that Eqs. 2 and 4 assume that a1 equals a2 in every iteration and are pessimistic. In the real world, iterations can be skipped and conclude fast as soon as the comparator step reveals that a1 and 2 are different.

We simulated the weighted STRIKE acceleration hardware of Fig. 12 with Table 2 clock information in Verilog Hadware description Language of ModelSim -Altera 6.5e Starter Edition, and ran it on 20 different pairs of 100-character sequences. Loading the CAM with new weights was also assumed and accounted for in every simulation run. For two equal strings all with one character repeated 100 times, the Verilog number of clocks was 665945 clocks, compared to 679701 clocks from Eq. (4) with N1 = N2 = 100 which assumes that a1 = a2, and that there are exactly 3 terms in the Table 1 equation of the Supplement. Averaging the Verilog simulation times of 20 runs, each with a pair of random sequences of 100 characters each with randomly generated characters, yields 220235.85 ns per run (at

1 GHz), or 220236 clock cycles per run, below the Eq. (4) numbers, as many iterations were skipped when a1≠a2. According to Table 6, this is 19,000x and 2,351x faster than 16-node and 128-node clusters respectively.

We also implemented the Weighted STRIKE accelerator on Intel-Altera DE1-SoC FPGA board running at 50 MHZ. The average measured time for matching 17 pairs of sequences, 100 characters each, was 9806 clocks or 0.196 msec. excluding the sequence pair loading time from memory (done during FPGA programming), very competitive with Table 6 times. With higher FPGA frequencies, this time will be much improved. The utilized hardware board resources were very small: 2 percent logic, and 3 percent DSP blocks (multipliers, adders).

8 Conclusion

Protein to protein interaction (PPI) is an important field in bioinformatics which identifies physical interactions between pairs of proteins. This in turn sheds light into how biochemical processes and signaling paths between cells take place helping in understanding diseases, modeling protein structures, and designing therapy. Many PPI algorithms exist but are expensive in time. A novel algorithm termed STRIKE [6] was also introduced to predict PPI by comparing pairs of protein sequences by matching common subsequences of fixed length.

The performance of the parallel STRIKE was evaluated [6] on multicore computers [6] and PC clusters [14] for a test set of 168 protein sequences and an 84–sequence training set with good algorithm scalability. The 128-node PC cluster performed this matching in about 2 hours compared to about a week on a single core x86 laptop.

For further performance acceleration, this paper proposed three hardware-dedicated designs for STRIKE: one for the unweighted version, one for the weighted version, and a parallel module organization for the Weighted version. The paper also presented performance models for the three hardware accelerators, and evaluated the execution times and speedups. Results indicate that the weighted STRIKE accelerator times are about 10 times longer than the unweighted STRIKE accelerator times. Although the weighted STRIKE accelerator consumes much longer times than the unweighted version, the weighted version enables the controlling of the matching to match specific protein strings, while the unweighted version treats all protein strings equally. To further improve the performance of the weighted STRIKE, an embarassignly-parallel module STRIKE accelerator organization duplicating the weighted STRIKE accelerator module was proposed achieving near linear speedups for long sequences, of 100 or more characters. As demonstrated by Verilog simulations and FPGA runs, the weighted STRIKE hardware accelerator exhibits 3 orders of magnitude speed improvement over multi-core and cluster computers. Furthermore, the parallel module accelerator can achieve much higher speedups.

Acknowledgments

Maha Alameddin, Laboratory Engineer at the University of Sharjah, UAE, developed the Verilog code.

References

[1] E. Sprinzak and H. Margalit, "Correlated sequence-signatures as markers of protein-protein interaction," *J. Mol. Biol.*, vol. 311, pp. 681–692, 2001.
[2] M. Deng, S. Mehta, F. Sun, and T. Cheng, "Inferring domain-domain interactions from protein-protein interactions," *Genome Res*, vol. 12, 2002, pp. 1540–1548.
[3] T. Huang *et al.*, "POINT: A database for the prediction of protein-protein interactions based on the orthologous interactome," *Bioinformatics*, vol. 20, 2004, pp. 3273–3276.
[4] C. Xue-Wen and L. Mei, "Prediction of protein–protein interactions using random decision forest framework," *Bioinformatics*, vol. 21, 2005, pp. 4394–4400.
[5] P. Sylvain *et al.*, "PIPE: A protein-protein interaction prediction engine based on the re-occurring short polypeptide sequences between known interacting protein pairs," *BMC Bioinf.*, vol. 7, 2006, Art. no. 365.
[6] F. N. Sibai and N. Zaki, "Parallel protein sequence matching on multicore computers," in *Proc. 2nd IEEE Int. Conf. Soft Comput. Pattern Recognit.*, Dec. 2010, pp. 285–290.
[7] H. Lodhi *et al.*, "Text classification using string kernels," *J. Mach. Learn. Res.*, vol. 2, 2002, pp. 419–444.
[8] N. Zaki, D. S., and I. R., "Application of string kernels in protein sequence classification," *Appl. Bioinf.*, vol. 4, 2005, pp. 45–52.
[9] D. Haussler, "*Convolution Kernels On Discrete Structures*," Univ. California Santa Cruz, Santa Cruz, CA, Tec. Rep. UCSC-CRL- 99-10, 1999.
[10] C. Watkins, "Dynamic alignment kernels," in *Advances Large Margin Classifiers*, Cambridge, MA, USA: MIT Press, 2000, pp. 39–50.
[11] J. Sanders and E. Kandrot, *CUDA By Example: An Introduction to General-Purpose GPU Programming*, London, U.K.: Pearson, 2010.
[12] nVIDIA, CUDA C Programming Guide v. 9.1, 2018. [Online]. Available: https://docs.nvidia.com/cuda/archive/9.1/pdf/CUDA_C_Programming_Guide.pdf
[13] A. Munshi, "The OpenCL specification Version 1.2," 2012. [Online]. Available: https://www.khronos.org/registry/OpenCL/specs/opencl-1.2.pdf
[14] A. El-Moursy, W. Afifi, F. N. Sibai, and S. Nassar, "Parallel PPI prediction performance study on HPC platforms," *J. Circuits Syst. Comput.*, vol. 24, no. 5, 2015, Art. no. 28.
[15] E. Bateman *et al.*, "The pfam protein families database," *Nucleic Acids Res.*, vol. 28, pp. 263–266, 2000.
[16] N. Zaki, W. El-Hajj, H. Kamel, and F. N. Sibai, "A protein–protein interaction classification approach," in *Software Tools and Algorithms for Biological Systems*, Berlin, Germany: Springer, 2011.
[17] Z. You, Z. Ming, H. Huang, and X. Peng, "A novel method to predict protein-protein interactions based on the information of protein sequence," in *Proc. IEEE Conf. Control Syst. Comput. Eng .*, 2012, pp. 210–215.
[18] N. Zaki *et al.*, "Protein protein interaction based on pairwise similarity," *BMC Bioinf.*, vol. 10, pp. 10–150, 2009.
[19] N. Abbas, "Acceleration of a bioinformatics application using high-level synthesis," Doctoral Dissertation, L'Universite Europeene de Bretagne, France, 2012.
[20] I. Li, S. W., , and K. Truong, "160-fold acceleration of the Smith-Waterman algorithm using a field programmable gate array (FPGA)," *BMC Bioinf.*, vol. 8, 2007, Art. no. 185.
[21] B. Schmidt, "Algorithms and tools for bioinformatics on GPUs," *nVIDIA GTC Conf.*, 2012. [Online]. Available: https://on-demand.gputechconf.com/gtc/2012/presentations/S0008-Algorithms-and-Tools-for-Bioinformatics-on-GPUs.pdf
[22] L. Ligowski, W. R. Rudnicki, Y. Liu, and B. Schmidt, "Accurate scanning of sequence databases with the Smith-Waterman algorithm," *GPU Comput. Gems*, vol. 1, pp. 155–171, 2011.
[23] Y. Liu, D. L. Maskell, and B. Schmidt, "CUDASW++: Optimizing Smith-Waterman sequence database searches for CUDA-enabled graphics processing units," *BMC Res. Notes*, vol. 2, 2009, Art. no. 73.
[24] P. Vouzis and N. Sahinidis, "GPU-BLAST: Using graphics processors to accelerate protein sequence alignment," *Bioinformatics*, vol. 27, no. 2, pp. 182–188, 2011.
[25] J. Pérez-Serrano *et al.*, "DNA sequences alignment in multi-GPUs: Acceleration and energy payoff," *BMC Bioinf.*, vol. 19, no. 421, pp. 161–176, 2018.
[26] V. Srinivasa Rao, K. Srinivas, G. N. Sujini, and G. N. Sunand Kumar, "Protein-protein Interaction detection: Methods and analysis," *Int. J. Proteomic.*, vol. 2014, 2014, Art. no. 147648.

Fadi N. Sibai received the BS degree in electrical engineering from the University of Texas at Austin, and the MS and PhD degrees in electrical engineering from Texas A&M University. He joined Prince Mohammad Bin Fahd University, Saudi Arabia, in 2019 as dean of the College of Computer Engineering and Science. Between 2011 and 2019, he worked for Saudi Aramco. Between 2006 and 2011, he directed the Computer Systems Design program and IBM Cell Competence Center at the UAE University where he received IBM's highest research Award. Between 1996 and 2006, he managed programs and engineering teams at Intel Corporation, CA. Between 1990 and 1996, he was an assistant professor of Electrical Engineering with the University of Akron. He authored or coauthored more than 230 publications and technical reports and served on the Organizing or Program Committees of more than 20 Conferences. He holds PMP, CISSP, CCNA, and CQRM certifications. He is a member of PMI, $(ISC)^2$, & Eta Kappa Nu.

Ali A. El-Moursy received the PhD from the University of Rochester, USA, in 2005. He worked for Software Solution Group, Intel Corp., CA, till early 2007. In 2007, he joined Electronics Research Institute, Egypt. His research interest include high-performance computer architecture, and parallel and cloud computing. He also participated with IBM Cairo Technology Development Center, Egypt, as a visitor research scientist for the period from Feb. 2007 till Jan. 2010. In Sep. 2010, he joined the ECE Department at University of Sharjah, UAE, as an assistant professor, and was promoted to the associate professor rank in 2017.

Abu Asaduzzaman (Member, IEEE) received the PhD and MS degrees in computer engineering from Florida Atlantic University, in 2009 and 1997, respectively. He is an associate professor with the Department of Electrical Engineering and Computer Science, Wichita State University. He received research grants from NSF KS EPSCoR, NVIDIA, NetApp, Wiktronics, and M2SYS. He has authored more than 70 papers. His research interests include computer architecture, high performance computing, and embedded systems. He received both the ISERD Excellent Paper, and the IEEE ICAEE best paper awards in 2015. He is also a member of the ASEE.

Sohaib Majzoub (Member, IEEE) received the PhD degree from the University of British Columbia, Canada. He is an associate professor with the Electrical Engineering Department, University of Sharjah, UAE. He was previously an Assistant Professor at King Saud University, and the American University in Dubai.