A New Processor Array Structure for Protein Sequence Alignment using Smith-Waterman Algorithm

ATEF IBRAHIM
Prince Sattam Bin Abdulaziz University
Department of Computer Engineering
Alkharj
SAUDI ARABIA
attif_ali2002@yahoo.com

HAMED ELSIMARY
Prince Sattam Bin Abdulaziz University
Department of Computer Engineering
Alkharj
SAUDI ARABIA
hamed@eri.sci.eg

ABDULLAH ALJUMAH
Prince Sattam Bin Abdulziz University
Department of Computer Engineering
Alkharj
SAUDI ARABIA
aljumah88@hotmail.com

Abstract: This paper proposes a new processor array structure for the Smith-Waterman with affine gap penalty algorithm to align protein sequences. This architecture is extracted by applying a nonlinear mapping methodology to the Smith-Waterman with affine gap penalty algorithm after expressing it as Regular Iterative Algorithm (RIA). This methodology uses a data scheduling and node projection techniques to explore the processor array structures of the algorithm. The proposed structure is one of the explored structures and has the advantage that it can be modified to enable hardware reuse rather than replicating processing elements of the processor array on a cluster of FPGAs. The proposed hardware structure and the previously reported conventional one are described at the Register Transfer Level (RTL) using VHDL language and implemented using the FPGA technology. The implementation results show that the proposed design has significant higher normalized speed-up (up to 124%) over the conventional design for query sequence lengths less than 512 residues. According to the UniProtKB/Swiss-Prot protein knowledgebase (release 2014_07) statistics, the largest number of sequences (about 80%) have sequence length less than 512 residues that makes the proposed design outperforms the conventional design in terms of speed and area in this sequence lengths range.

Key–Words: High performance computing, Parallel processing, Processor arrays, Bio-Computing, Protein sequence alignment, Reconfigurable Computing

1 Introduction

Protein alignment by Dynamic Programming based (DP-based) algorithms using general purpose processors (microprocessors) results in quadratic time complexities. Because of the exponential growth of biological databases, an enormous growth in research which focuses on accelerating DP-based algorithms has happened. These algorithms are accelerated in parallel architectures such as linear single instruction multiple data (SIMD) arrays and processor arrays. Both architectures are good candidates for fine-grained parallel architectures for the acceleration of sequence alignment with DP-based algorithms. Coarse-grained parallelism is another approach, where computations of DP-based algorithms are distributed over multiprocessor clusters. In spite of the fact that coarse-grained parallelism significantly increases computation speed, such implementations consume significant amounts of energy as well as involving increased size, operational costs, and maintenance. On the other hand, fine-grained parallelism using processor arrays has been implemented on both FPGA and ASIC platforms. The latter implements processor arrays in a single-purpose chip and has provided relatively good area/speed ratios; however, the single purpose hardware lacks the re-programmability which is important for sequence alignment. Over the last decades, reconfigurable FPGAs have becoming an important alternative to the expensive and large energy consumption of high performance supercomputers and multiprocessor clusters.

The amazing speed-up of FPGAs in accelerating biocomputing algorithms has led to such reconfigurable computing platforms being used as acceleration platforms for scientific computing.

Advances in IC technology over the last decade have led to production of FPGAs with large sizes that can accommodate complex designs available nowadays. This has led researchers to utilize FPGAs for the implementation of the Smith Waterman alignment algorithm with the affine gap penalty [1]. Human Genome Project (HGP) in 2003, provided databases with massive numbers of biological sequences [2]. Due to the massive number of biological sequences, they require a large time and resources to be processed, which represents a real challenge to the available technology. The implementation of the Smith-Waterman with affine gap penalty on FPGA as in [3] was among the early works reported in literature. Yamaguchi et al. [4] in 2002 implemented the Smith-Waterman with affine gap function on the RC 1000-PP Celoxica board with Virtex-II FPGA. During that time, Virtex-II was the latest FPGAs and the Xilinx XCV2000E device fitted a maximum of 144 processing elements. Oliver et. al. in [3] presented in their work the run time reconfiguration of PEs in order to reuse the resources. Jacobi et. al. [5] realized a reconfigurable system to implement the algorithm on Virtex-II FPGA board. Similar work presented by VanCourt and Herbordt in [6] and Hoang et. al. in [7]. Mohamed Abouellail et. al. in [8] presented PE processor arrays on a cluster of multiple FPGAs, as reported to solve the problem of long queries. Another approach to solve for long queries has also been reported in [9], [10], [11], [12], [13]. They presented the reuse of PEs in a technique known as folding. Examples of such an approach were reported by Xianyang et al. [13]. Zhang et. al. in [14] presented a new implementation of the Smith-Waternam algorithm on reconfigurable FPGAs in a supercomputer fashion by redesigning the PEs in a way to reduce the storage accompanied with each PE to enable rapid access to the substitution matrix as it was stored in the PE. Yamaguchi et. al in [15] presented similar technique to store the substitution matrix in the PE for multiple-pass computation. Isa et. al. in [16] presented a processor array with reconfigurable PE, they presented a scheduling strategy to optimize the overall run time overlapped between computation and configuration of the architecture.

In this paper, the authors propose a new folded processor array architecture with reconfigurable PE for the Smith-Waterman with affine gap penalty alignment algorithm that are more efficient in speed and area than the folded processor array architecture presented in [16] specially for short query sequences. This will be achieved by applying a nonlinear mapping methodology to the Smith-Waterman with affine gap penalty alignment algorithm after expressing it as Regular Iterative Algorithm (RIA). This methodology uses a data scheduling and node projection techniques to explore the processor arrays of the algorithm. Also, we will present the HW realization of the processing element (PE) of the processor array structure and apply the scheduling strategy of [16] to the PE structure to reuse the processor array for multiple pass processing without sacrificing more time required for configuration.

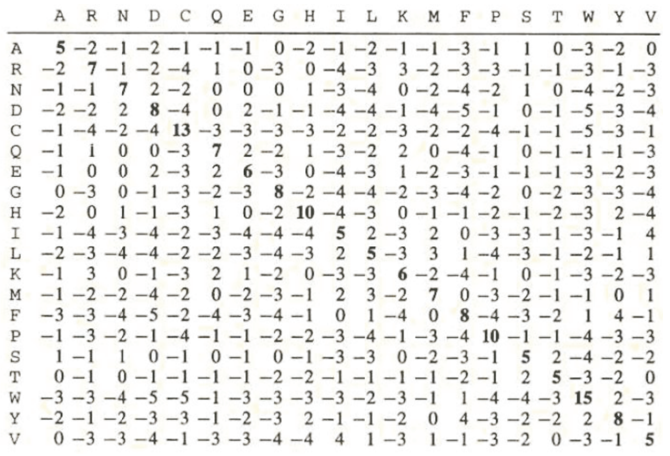
This paper is organized as follows. Section 2 presents the Smith-Waterman with affine gap penalty algorithm. Section 3 presents the proposed methodology employed to explore the processor array architectures and describes the modification of the explored architectures to be reused for multiple pass processing using folding technique. Section 4 compares the resulting processor array architectures in terms of area and speed. Finally Section 5 concludes the paper.

2 Smith-Waterman with affine gap penalty algorithm

Equation (1) shows the Smith-Waterman algorithm with a linear gap penalty [17]. Given a query sequence, $X = x_1, x_2, x_3 \ldots x_i \ldots x_M$ (of length M) and a subject sequence, $Y = y_1, y_2, y_3 \cdots y_j \cdots y_N$ (of length N), this DP-based alignment algorithm searches for the best alignment between subsequences of X and Y using alignment matrix $F(i, j)$. This matrix calculates the maximum score among the four alternatives and it is built recursively using Equation (1).

$$F(i,j) = \max \begin{cases} 0 \\ F(i-1, j-1) + s(x_i, y_j) \\ F(i-1, j) - d \\ F(i, j-1) - d \end{cases} \tag{1}$$

the $s(x_i, y_j)$ is the relating substitution matrix score for residue i in query sequence X and residue j in subject sequence Y, respectively. This score is a probabilistic score which depicts biological relationship of residues x_i and y_j as indicated in the substitution matrix in Fig. 1. This figure presents an illustration of the BLOSUM50 substitution matrix [18]. There are different probabilistic models like BLOSUM62 and PAM that illustrates the biological relationships between amino acids. The elements highlighted in bold on the main diagonal represent indistinguishable residue pairs. The 20 $\times$ 20 matrix comprises of 20 residues (or amino-acids).



	A	R	N	D	C	Q	E	G	H	I	L	K	M	F	P	S	T	W	Y	V
A	**5**	−2	−1	−2	−1	−1	−1	0	−2	−1	−2	−1	−1	−3	−1	1	0	−3	−2	0
R	−2	**7**	−1	−2	−4	1	0	−3	0	−4	−3	3	−2	−3	−3	−1	−1	−3	−1	−3
N	−1	−1	**7**	2	−2	0	0	0	1	−3	−4	0	−2	−4	−2	1	0	−4	−2	−3
D	−2	−2	2	**8**	−4	0	2	−1	−1	−4	−4	−1	−4	−5	−1	0	−1	−5	−3	−4
C	−1	−4	−2	−4	**13**	−3	−3	−3	−3	−2	−2	−3	−2	−2	−4	−1	−1	−5	−3	−1
Q	−1	1	0	0	−3	**7**	2	−2	1	−3	−2	2	0	−4	−1	0	−1	−1	−1	−3
E	−1	0	0	2	−3	2	**6**	−3	0	−4	−3	1	−2	−3	−1	−1	−1	−3	−2	−3
G	0	−3	0	−1	−3	−2	−3	**8**	−2	−4	−4	−2	−3	−4	−2	0	−2	−3	−3	−4
H	−2	0	1	−1	−3	1	0	−2	**10**	−4	−3	0	−1	−1	−2	−1	−2	−3	2	−4
I	−1	−4	−3	−4	−2	−3	−4	−4	−4	**5**	2	−3	2	0	−3	−3	−1	−3	−1	4
L	−2	−3	−4	−4	−2	−2	−3	−4	−3	2	**5**	−3	3	1	−4	−3	−1	−2	−1	1
K	−1	3	0	−1	−3	2	1	−2	0	−3	−3	**6**	−2	−4	−1	0	−1	−3	−2	−3
M	−1	−2	−2	−4	−2	0	−2	−3	−1	2	3	−2	**7**	0	−3	−2	−1	−1	0	1
F	−3	−3	−4	−5	−2	−4	−3	−4	−1	0	1	−4	0	**8**	−4	−3	−2	1	4	−1
P	−1	−3	−2	−1	−4	−1	−1	−2	−2	−3	−4	−1	−3	−4	**10**	−1	−1	−4	−3	−3
S	1	−1	1	0	−1	0	−1	0	−1	−3	−3	0	−2	−3	−1	**5**	2	−4	−2	−2
T	0	−1	0	−1	−1	−1	−1	−2	−2	−1	−1	−1	−1	−2	−1	2	**5**	−3	−2	0
W	−3	−3	−4	−5	−5	−1	−3	−3	−3	−3	−2	−3	−1	1	−4	−4	−3	**15**	2	−3
Y	−2	−1	−2	−3	−3	−1	−2	−3	2	−1	−1	−2	0	4	−3	−2	−2	2	**8**	−1
V	0	−3	−3	−4	−1	−3	−3	−4	−4	4	1	−3	1	−1	−3	−2	0	−3	−1	**5**

Figure 1: BLOMSUM50 substitution matrix [18].

The constant d in Equation (1) represents the linear gap penalty that penalizes gaps of length g linearly, i.e. $penalty(g) = -g * d$. GOTOH [1] in 1982, presented a more productive gap penalty which is referred to as affine gap penalty as indicated in Equation (2). In this type of gap penalty, a fixed gap cost is given when opening a new gap (gap opening or d), while a linear and often smaller gap penalty is given for following gap extensions (e) i.e. $penalty(g) = -d - (g-1) * e$. The $F(i, j)$ is the score up to (i, j) where residue x_i is aligned to residue y_j. The $I_x(i, j)$ is the best score, where residue x_i is aligned to a gap and finally the $I_y(i, j)$ is the best score, where residue y_j is aligned to a gap. In this paper, we will apply a proposed methodology to Equation (2) to explore all possible processor array architectures and describe the modifications of the explored architectures to be reused for multiple pass processing using folding technique as shown in the following sections.

$$F(i,j) = \max \begin{cases} F(i-1, j-1) + s(x_i, y_j) \\ I_x(i-1, j-1) + s(x_i, y_j) \\ I_y(i-1, j-1) + s(x_i, y_j) \end{cases}$$

$$I_x = \max \begin{cases} F(i-1, j) - d \\ I_x(i-1, j) - e \end{cases} \tag{2}$$

$$I_y = \max \begin{cases} F(i, j-1) - d \\ I_y(i, j-1) - e \end{cases}$$

3 A Systematic Methodology for Processor Array Design

Systematic methodologies to design processor arrays allow for design space exploration for optimizing performance according to certain specifications while satisfying design constrains. Several methodologies were proposed earlier [19], [20], [21], [22]. However, most of these methodologies were not able to deal with algorithms that have dimensions more than two. The authors of [22], [23] proposed a systematic methodology that deals with algorithms of arbitrary dimensions. They proposed a formal algebraic procedure for processor array design starting from a Regular Iterative Algorithm (RIA) for a three-dimensional digital filter which gives rise to a dependency graph in six-dimensional space. In this work, we used this formal technique to develop processor arrays for Smith-Waterman with affine gap penalty algorithm.

3.1 Obtaining the Algorithm Dependency Graph (DG)

The Smith-Waterman with affine gap penalty algorithm explained in Equation (2) can be easily defined on a two dimensional (2D) domain since there are two indices (i, j). The DG is shown in Fig. 2. The computation domain is the convex hull in the 2D space, where the algorithm operations are defined as indicated by circles in the 2D plane [22], [23], [24]. Also, from this figure we notice that the input variables x_i, $F(i, j-1)$ and $I_y(i, j-1)$ are represented by horizontal lines, the input variables y_j, $F(i-1, j)$ and $I_x(i-1, j)$ are represented by vertical lines, and the output variables $F(i, j)$, $I_x(i, j)$ and $I_y(i, j)$ are represented by the slanted lines. the zero inputs at the left and upper borders of DG represents the initial values of the alignment matrix, $F(i, 0)$, $I_x(i, 0)$, $I_y(i, 0)$ and $F(0, j)$, $I_x(0, j)$, $I_y(0, j)$.

3.2 Data Scheduling

Pipelining or broadcasting the variables of an algorithm is determined by the choice of a timing function that assigns a time value to each node in the DG. A simple but useful timing function is an affine scheduling function of the form [22].

$$t(p) = Gp - g \tag{3}$$

where the function $t(p)$ associates a time value t to a point p in the DG. Value of G is chosen to ensure that only positive time index values are obtained. The row vector $G = [g_1\ g_2]$ is the scheduling vector and g is an integer. The affine scheduling function must satisfy

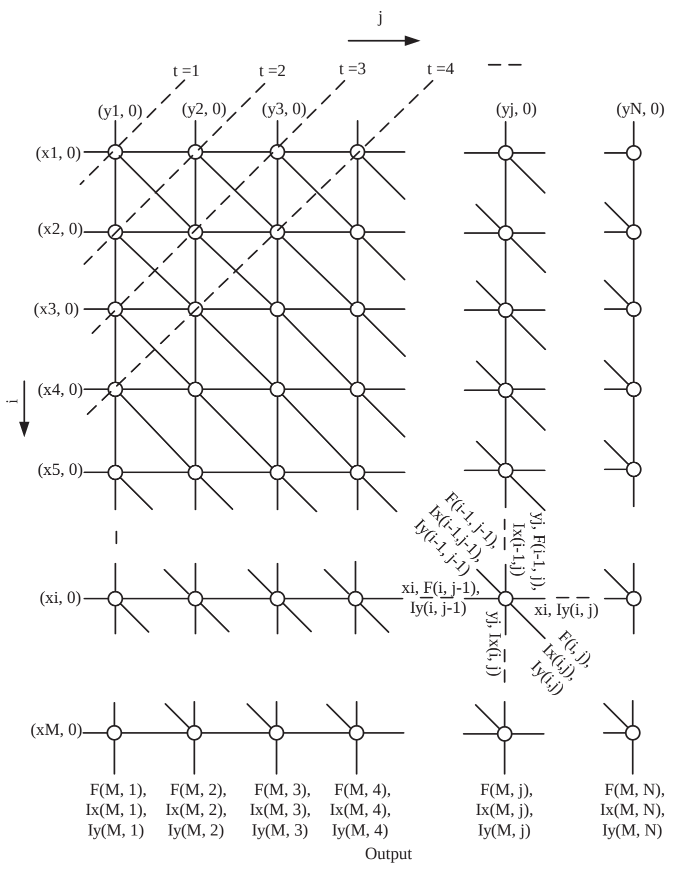

j
t =1
t =2
t =3
t =4
(y1, 0)
(y2, 0)
(y3, 0)
(yj, 0)
(yN, 0)
(x1, 0)
(x2, 0)
(x3, 0)
(x4, 0)
(x5, 0)
(xi, 0)
(xM, 0)
i
F(i-1, j-1), Ix(i-1, j-1), Iy(i-1, j-1)
yj, F(i-1, j), Ix(i-1,j)
xi, F(i, j-1), Iy(i, j-1)
xi, Iy(i, j)
yj, Ix(i, j)
F(i, j), Ix(i,j), Iy(i,j)
F(M, 1), Ix(M, 1), Iy(M, 1)
F(M, 2), Ix(M, 2), Iy(M, 2)
F(M, 3), Ix(M, 3), Iy(M, 3)
F(M, 4), Ix(M, 4), Iy(M, 4)
F(M, j), Ix(M, j), Iy(M, j)
F(M, N), Ix(M, N), Iy(M, N)
Output

Figure 2: Smith-Waterman with affine gap penalty algorithm dependency graph.

several conditions. From Fig.2, we observe that in each column the output variable $I_x(i, j)$ of each node depends on the output variables $I_x(i-1, j)$ and $F(i-1, j)$ form the previous node in the same column, thus we can write

$$t(p(i-1, j)) < t(p(i, j)) \tag{4}$$

Applying our scheduling function in Equation (3) to this inequality, we get

$$\begin{bmatrix} g_1 & g_2 \end{bmatrix} \begin{bmatrix} i-1 \\ j \end{bmatrix} < \begin{bmatrix} g_1 & g_2 \end{bmatrix} \begin{bmatrix} i \\ j \end{bmatrix} \tag{5}$$

$$ig_1 - g_1 + jg_2 < ig_1 + jg_2 \tag{6}$$

Which could be simplified to

$$g_1 > 0 \tag{7}$$

Similarly From Fig.2, In each row we observe that the output variable $I_y(i, j)$ depends on the previous output variables $I_y(i, j-1)$ and $F(i, j-1)$ of the same row, thus we can write

$$\begin{bmatrix} g_1 & g_2 \end{bmatrix} \begin{bmatrix} i \\ j-1 \end{bmatrix} < \begin{bmatrix} g_1 & g_2 \end{bmatrix} \begin{bmatrix} i \\ j \end{bmatrix} \tag{8}$$

$$ig_1 + jg_2 - g_2 < ig_1 + jg_2 \tag{9}$$

Which could be simplified to

$$g_2 > 0 \tag{10}$$

From Equations (7) and (10) there are many solutions for G, the most reasonable and simplest one is

$$G = [1\ 1] \tag{11}$$

If we want to pipeline or allocate a variable whose nullvector is θ , we must have

$$G\theta^t \neq 0 \tag{12}$$

where θ is the nullvector of the variable dependence matrix [22]. On the other hand, if we want to broadcast a variable whose nullvector is θ , we must have [22]

$$G\theta^t = 0 \tag{13}$$

To study the timing of the variables x_i, y_j, $I_x(i, j), I_y(i, j)$ and $F(i, j)$, we first find their nullvectors

$$\theta_{x_i} = [0\ 1] \tag{14}$$

$$\theta_{y_j} = [1\ 0] \tag{15}$$

$$\theta_{I_x(i,j)} = [-1\ -1] \tag{16}$$

$$\theta_{I_y(i,j)} = [-1\ -1] \tag{17}$$

$$\theta_{F(i,j)} = [-1\ -1] \tag{18}$$

The product of G and these nullvectors gives

$$G\theta^t_{x_i} = 1 \tag{19}$$

$$G\theta^t_{y_j} = 1 \tag{20}$$

$$G\theta^t_{I_x(i,j)} = -2 \tag{21}$$

$$G\theta^t_{I_y(i,j)} = -2 \tag{22}$$

$$G\theta^t_{F(i,j)} = -2 \tag{23}$$

Therefore, The input and output variables will be pipelined or allocated.

3.3 DG Node Projection

The projection operation is a many-to-one function that maps several nodes of the DG onto a single node, which constitutes the resulting processor array. Thus, several operations in the DG are mapped to a single PE. The projection operation allows hardware economy by multiplexing several operations in the DG on a single PE. The author of [22] explained how to perform the projection operation using a projection matrix P. To obtain the projection matrix, we need to define a desired projection direction d. The vector d belongs to the null-space of P. Since we are dealing with a two-dimensional DG, matrix P is a row vector and d is a column vector [22]. A valid projection direction d must satisfy the inequality [22]

$$Gd \neq 0 \tag{24}$$

In the following, we will discuss design space explorations based on the timing function G obtained in Equation (11). There are many projection directions that satisfy Equation (24) for the scheduling function in Equation (11). For simplicity we choose three of them as follows:

$$d_1 = [1\ 0]^t \tag{25}$$

$$d_2 = [0\ 1]^t \tag{26}$$

$$d_3 = [1\ 1]^t \tag{27}$$

The corresponding projection vectors are given by

$$P_1 = [0\ 1] \tag{28}$$

$$P_2 = [1\ 0] \tag{29}$$

$$P_3 = [-1\ 1] \tag{30}$$

Our processor design space now allows for three processor array configurations for each projection vector for the timing function G. The processor array associated with the projection vector P_2 is the same as the conventional one perviously reported in the literature [16]. The processor array associated with the projection vector P_3 is not regular and have high complexity that makes it not suitable for VLSI implementation. Thus, we will ignore this design in this research paper. In the following subsection, we study the processor arrays associated with the projection vectors P_1 and P_2.

3.3.1 Design1: using $P_1 = [0\ 1]$

A point in the DG $p = [i\ j]^t$ will be mapped by the projection vector $P_1 = [0\ 1]$ onto the point

$$p' = P_1 p = j \tag{31}$$

The resulting processor array corresponding to the projection vector P_1 consists of N PEs. Only at most M PEs are active at each time step. Fig. 3 shows the processor activity for the case $N = 6$ and $M = 3$, where the black nodes represent active PEs and white nodes represent idle PEs. Since only maximum M PEs are active at a given time step, the PEs are not well utilized. To improve PE utilization, we need to reduce the number of processors. We note from Fig. 3 that PE$_j$ and PE$_{j+M}$ are active at non-overlapping time steps. Thus, each pair of PEs (PE$_j$ and PE$_{j+M}$) can be mapped to a single PE without causing time conflicts. This can be achieved by mapping PEs with indices j (in Fig. 3) to PEs with indices j' using the following nonlinear projection operation:

$$j' = j \bmod M \tag{32}$$

The resulting processor array for different values of N and M is shown in Fig. 4. To the best of our Knowledge, this processor array is new and was not reported before in the literature. The processor array consists of M PEs. Input bits of subject sequence $y(kM + j)$ should be allocated to each processing element in the processor array during the computation cycle (M clock cycles), where k has values in range from 0 to $\lceil N/M \rceil - 1$. On the other hand, Input bits of the query sequence x_i and intermediate out bits of $I_x(i, j)$, $I_y(i, j)$ and $F(i, j)$ are pipelined between adjacent PEs. A tristate buffer at the output of each PE ensures that it is the only output fed to the output bus.

3.4 Design1 folded processor array

To solve the problem of long queries, this design architecture has been modified to be reused for multiple pass processing and implemented on a single FPGA instead of replicating the processing elements on a cluster of multiple FPGAs. This design approach is called folding approach [20], [25] and it has the advantage of decreasing the significant amounts of energy consumed as well as reducing design size, operational costs and maintenance. The design modification starts by partitioning the algorithm in hand into small alignment steps and map the partitioned algorithm on a fixed size linear processor array. This problem is well studied in the VLSI design arena [20], [25]. The following illustrates the modification process.

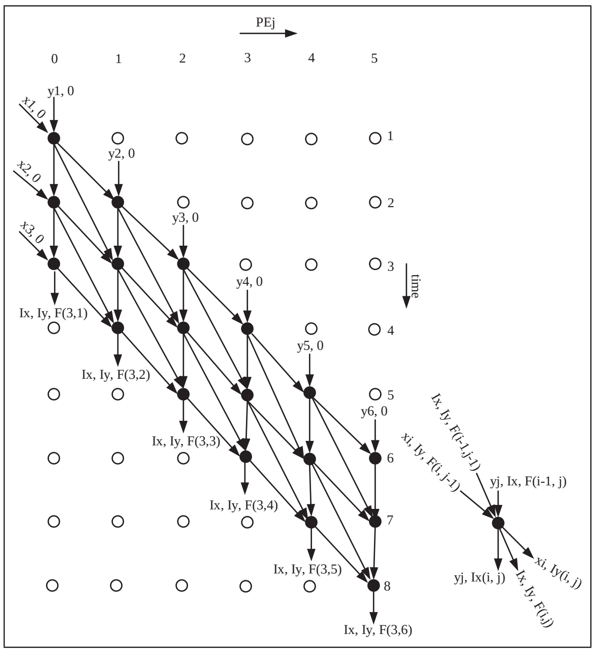


Figure 3: processor activity at different time steps for $d_1 = [1\ 0]^t$, $N = 6$, and $M = 3$.

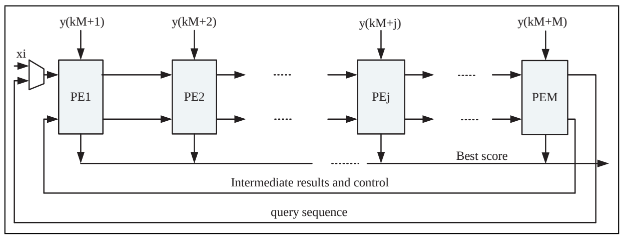


Figure 4: processor array for $d_1 = [1\ 0]^t$ for different values of N and M.

let us presume the common situation of a query sequence length of M and a linear processor array of size v, where $M > v$ and $q = \lceil M/v \rceil$. First, the linear processor array of length M is conceptually expanded to an array of length $q \times v$ with the last $qv - M$ PEs filled with zero-values. Thereby, these additional PEs do not effect the total alignment result. After this conceptual step, the produced linear processor array of length $q \times v$ is folded into the actual array of length v. As a result of this folding process, the alignment process is accomplished in kq passes over the linear array. For this, we need a first-in-first-out (FIFO) to store $M - v$ bits of the query sequence, and intermediate results from each pass before they are fed back to the array input for the next pass (see Fig. 5). The depth of the FIFO is dictated by the length of the query sequence.

The processing over multiple passes requires a

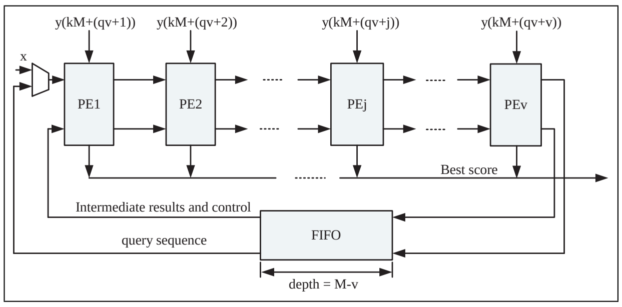


Figure 5: Design1 folded fixed size processor array

different set of substitution matrix columns for each pass computation. The substitution matrix column (In [16], the authors referred to substitution matrix column as the configuration element (CE) and we will use this term here) is dictated by the subject residue held by the PE for an alignment matrix computation. The design of the proposed sequence alignment core architecture is based on the scheduling strategy (Known as overlapped computation and configuration (OCC)) proposed by [16]. We used fixed CEs (two CEs) for each processing element and adopted the double buffering technique proposed by [16] to manage the fixed CEs for alignment matrix computation and CE configuration in the folded processor array. In this case, the designed PE architecture optimizes logic resources in the PE by having fixed numbers of CEs. In the double buffering approach proposed by [16], one of the CEs is configured with different probability scores at different folds while the other CE holds a column of substitution matrix scores for the corresponding fold computation. A main controller [16] is utilized to schedule both configuration and computation modes to run at the same time. Therefore, The overlapping between configuration and computation modes virtually removes the time taken for CE configuration throughout every fold computation. Also, Another logic unit called "parallel loader" is proposed by [16] for efficient scheduling between tasks in computing the alignment matrix and configuring the CE for following pass computation. This loader is intended to configure CEs in parallel with limited CE configuration time regardless of the number of PEs or the length of the subject sequence. Thus, the time taken to configure the CE in all PEs is less than the time which has passed away in computing the alignment matrix. This enables the smooth scheduling of the concurrent operations (alignment matrix computation and CE configuration) during each fold computation. The subject residue-to-CE mapping is a part of the CE configuration task which is required to load only the subject-related substitution matrix columns into their corresponding CEs during CE configuration phase. The logic component that is responsible for mapping the CE with its corresponding subject sequence residue is called "subject loader". The operation of this loader is similar to the operation of the "query loader" proposed by [16]. For more details regarding the "main controller", the "parallel loader", and the "query loader" - that is similar to the "subject loader" - are given in [16].

Figure 6 shows the hardware implementation of each PE of the folded processor array. All computation parameters of gap data width (*gw*), compute data width (*cw*) and the substitution matrix score data width (*w*) in each PE are parameterizable. Four bits of (*gw*) are sufficient to represent the gap open (*d*) and gap extension (*e*) penalty scores for the affine gap function, while five bits of (*w*) are sufficient to represent the substitution matrix scores that each substitution matrix column includes 20 amino acids. The PE consists of three arithmetic units; the best score $F(i, j)$ where residues x_i and y_j are aligned to each other, the best score $I_x(i, j)$ where residue x_i is aligned to a gap, and the best score $I_y(i, j)$ where residue y_j is aligned to a gap. The PE "Best Score unit" calculates the 'maximum so far' of the alignment scores, taking into account the PE_j and PE_{j-1} best scores. In this architecture, the input $y(kM + j)$ represents a residue of the subject sequence Y. It is loaded in a register during the configuration phase and it is applied for the whole computational cycle (M clock cycles). The input x_i represents the residue of the query sequence X that is pipelined to the next PE through register. The resulted best score (maximum score of $I_x(M, kM + j)$ or $I_y(M, kM + j)$ or $F(M, kM + j)$) will be available on the output bus after M clock cycles through a tristate buffer. If the resulted best score satisfies a given threshold value, the corresponding subject sequence address besides the resulted best score itself are stored in a best score FIFO, otherwise they are ignored. There is a data path control signal called "Last-clk-cycle", that indicates the last cycle in each iteration. It controls the tristate buffer, at the output of each PE, to feed the resulted score to the output bus at the proper time. This control signal, in each PE, is delayed one clock cycle before propagating to the next PE. This is done by propagating this signal through registers between the PEs.

During the configuration mode, the subject sequence residue selects its corresponding substitution matrix column (CE) through multiplexors "mux1" or "mux2". The CE briefly holds a column of substitution matrix scores for alignment matrix computation. Since both CEs are used alternately for computation, a multiplexer "mux3" is used to select substitution matrix scores either from CE0 or CE1, whereby the selection is dictated by the CE-SEL port. The CE selection procedure is based on the computational passes. During all even numbered fold computations, CE0 is ready to supply its substitution matrix scores for PE computation. So also, during all odd-numbered computation, CE1 is ready to supply its substitution matrix scores for computation. During computation mode, query sequence residues selects the substitution matrix score, $S(x_i, y(kM + j))$, for the PE to process the elementary functions of the DP alignment algorithm.

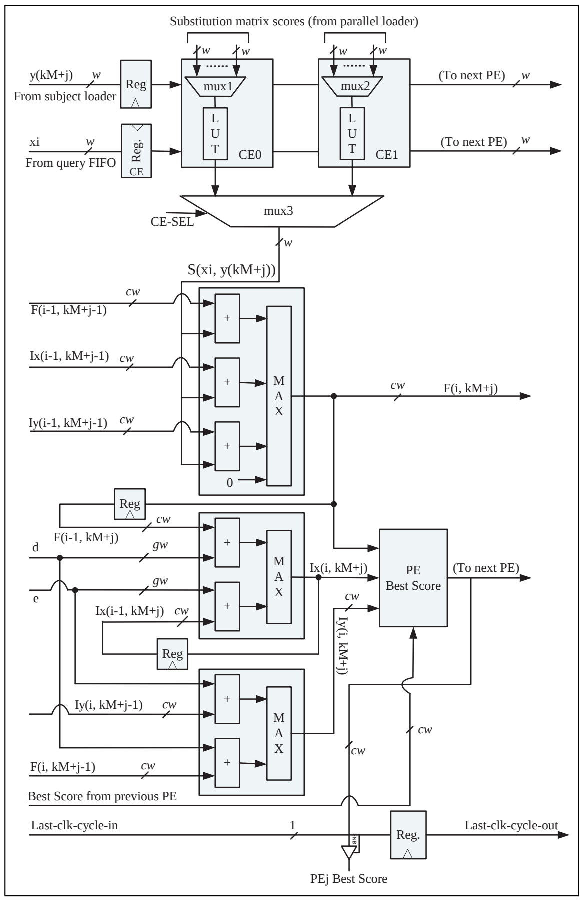


Figure 6: Design1 PE logic diagram.

3.5 Design2: using $P_2 = [1\ 0]$

A point in the DG $p = [i\ j]^t$ will be mapped by the projection matrix $P_2 = [1\ 0]$ onto the point

$$p' = P_2 p = i \tag{33}$$

The resulting processor array corresponding to the projection matrix P_2 consists of M PEs. Fig. 7 shows the processor activity for the case $N = 6$ and $M = 3$, where the black nodes represent active PEs and white nodes represent idle PEs. At most time steps, the maximum number of PEs, M, are active and this results in a good utilization of PEs. The resulting processor array, for different values of N and M is shown in Fig. 8. This processor array is similar to the one reported in [26], [27], [28], [29], [9], [30], [31], [32], [33], [34], [35], [36], [37], [38]. The processor array now consists of M PEs. Input bits of query sequence x_i should be allocated to each PE (as reported in the previous publications). On the other hand, input bits of the subject sequence y_j and the intermediate output bits $I_x(i, j)$, $I_y(i, j)$, and $F(i, j)$ $c(i, j)$ of each PE are pipelined to the next PE with higher index. The output score is obtained serially from the last PE (PEM) of the array after latency of M clock cycle.

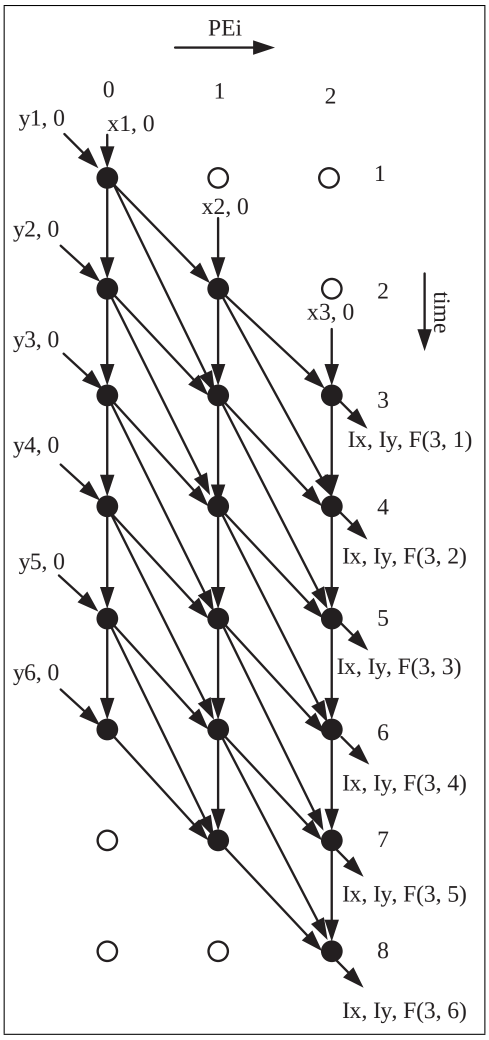


Figure 7: Processor activity at different time steps for $d_2 = [0\ 1]^t$, $N = 6$, and $M = 3$.

3.6 Design2 folded processor array

To solve the problem of long queries, we used the folding technique discussed in Sec. 3.4 and [16]. The resulted folded processor array for Design2 is shown in Fig. 9. Similar to the folded processor array of Design1, the alignment process using this folded processor array is accomplished in q passes over the linear array. For this, we need a FIFO to store the subject sequence and intermediate results from each pass before they are fed back to the array input for the next pass (see Fig. 9). In this case, the depth of the FIFO is dictated by the maximum length of the subject sequence in the protein database (the subject sequence maximum length in UniprotKB/Swiss-Port protein knowledgebase ,Release 2014_7 of July 2014, is 35,213 amino acids) as opposed to the case of Design1 that the depth of FIFO is dictated by the aligned query sequence.

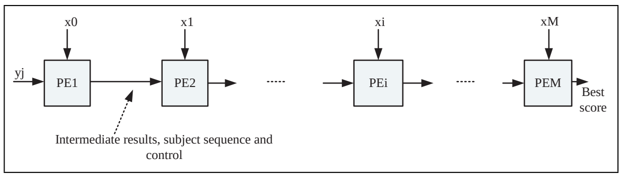


Figure 8: Processor array for $d_2 = [0\ 1]^t$ for different values of N and M.

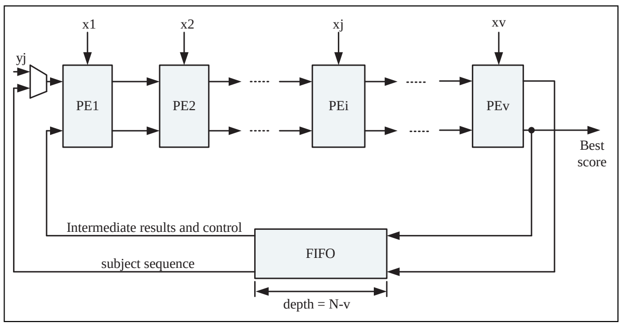


Figure 9: Design2 Folded fixed size processor array.

Figure 10 shows the hardware implementation of each PE of the folded processor array. The PE consists of three arithmetic units; the best score $F(i, j)$ where residues x_i and y_j are aligned to each other, the best score $I_x(i, j)$ where residue x_i is aligned to a gap, and the best score $I_y(i, j)$ where residue y_j is aligned to a gap. The PE "Best Score unit" calculates the 'maximum so far' of the alignment scores, taking into account the PE_j and PE_{j-1} best scores. In this architecture, the input x_i represents a residue of the query sequence X. It is loaded in a register during the configuration phase and it is applied for the whole computational cycle (M clock cycles). The input y_j represents the residue of the subject sequence Y that is pipelined to the next PE through register. The resulted best score (maximum score of $I_x(M, j)$ or $I_y(M, j)$ or $F(M, j)$) will be available on the output bus after M clock cycles from the last PE in the processor array. If the resulted best score satisfies a given threshold value, the corresponding subject sequence address besides the resulted best score itself are stored in a best score FIFO, otherwise they are ignored.

During the configuration mode, the query sequence residue selects its corresponding substitution matrix column (CE) through multiplexors ”mux1” or ”mux2”. The CE briefly holds a column of substitution matrix scores for alignment matrix computation. Since both CEs are used alternately for computation, a multiplexer ”mux3” is used to select substitution matrix scores either from CE0 or CE1, whereby the selection is dictated by the CE-SEL port. The CE selection procedure is based on the computational passes. During all even numbered fold computations, CE0 is ready to supply its substitution matrix scores for PE computation. So also, during all odd-numbered computation, CE1 is ready to supply its substitution matrix scores for computation. During computation mode, subject sequence residues selects the substitution matrix score, $S(x_i, y_j)$, for the PE to process the elementary functions of the DP alignment algorithm.

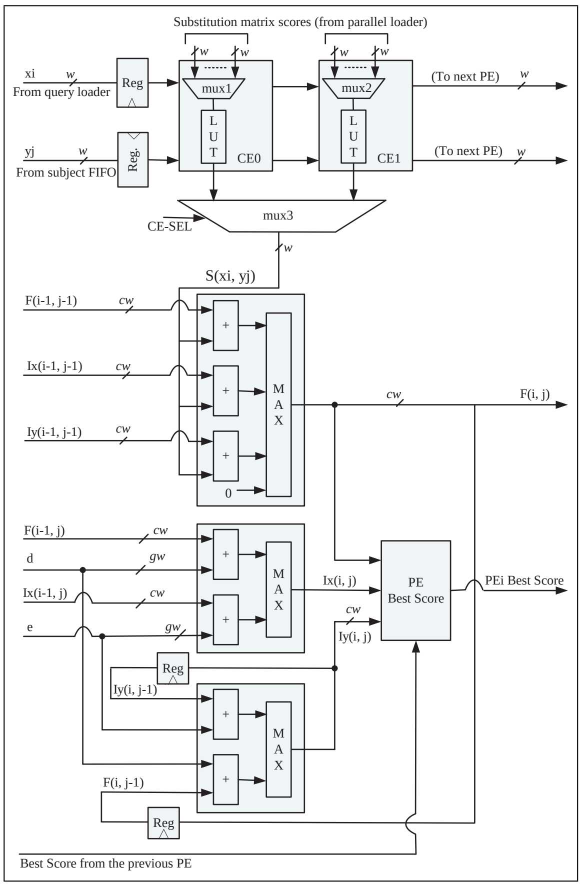


Figure 10: Design2 PE logic diagram.

4 Implementation Results

This section discusses the resource utilization and performance evaluation of the proposed novel design (Design1) and the previously reported design (Design2). The two designs are described in VHDL language and implemented on Alpha Data ADM-XRC-5LX card - with Xilinx Vertix-5 FPGA (XC5VLX110) on it - using Xilinx ISE8.1 tools. Both architectures are synthesized with query sequences lengths ranging from 64 residues to 2048 residues from the protein knowledgebase (UniPortKB). Each of the query sequences is aligned against subject sequences of lengths ranging from 2 to 2048 in processor array of size $v = 64$ with different folds.

In Table 1, the columns entitled ”ET1” and ”ET2” represent the execution time, in μs, needed to complete an alignment operation for Design1 and Design2, respectively. The columns entitled ”#LC1” and ”#LC2” represent the number of logic cells (LCs) occupied by each processor array for Design1 and Design2, respectively. The ”Speed up”, ”Area Ratio” and ”Area Normalized Speed up” design metrics are calculated using the synthesis results in order to measure the degree of optimization achieved in each design. The ”speed up” is calculated by dividing the execution time of Design2 ”ET2” by the execution time of Design1 ”ET1”, while the ”Area ratio” is calculated by dividing ”#LC1” by ”#LC2”. Moreover, the ”Area Normalized Speed up” is calculated by dividing ”Speed up” by ”Area Ratio”.

The Execution time for each design is calculated as the product of the total number clock cycles it takes and the clock period. The critical path delays obtained from synthesis results determines the clock period. The total number of clock cycles needed for Design1 are given by Equation 34, while the total number of clock cycles needed for Design2 are given by Equation 35.

$$T_{Design1} = qkM \tag{34}$$

$$T_{Design2} = qN \tag{35}$$

We notice from Table 1 that the area of the proposed design (#LC1) has different values depending on the length of the query sequence. This is attributed to the depth of FIFO in the processor array of Design1 is dictated by the length of the current aligned query sequence, while the depth of the FIFO in the processor array of the conventional design (Design2) is dictated by the maximum subject length ($N = 2048$) in the sample of subject sequences that the query sequence is

Table 1: Execution time and normalized speed-up of the proposed design (Design1) and the conventional design (Design2) using processor array of size $v = 64$ with different folds.

query sequence M	q	Design1		Design2		Speed-up	Area Ratio	Area Normalized Speed up
		ET1(μs)	#LC1	ET2(μs)	#LC2			
64	1	12.06	41,657	23.04	49,156	1.9	0.85	2.24
128	2	29.65	42,546	46.08	49,156	1.6	0.86	1.86
256	4	68.49	43,456	90.11	49,156	1.3	0.87	1.49
512	8	176.33	44,758	178.59	49,156	1.01	0.91	1.11
1024	16	491.52	46,623	357.173	49,156	0.73	0.95	0.77
2048	32	1,802.24	49,712	714.35	49,156	0.4	1.01	0.39

aligned against. This demonstrates the fixed area values of the conventional design (Design2). Also, we notice from this table that the proposed design (Design1) has a normalized speed up of 11% for query length $M = 512$ and increases linearly as the query length decreases (up to 124% for $M = 64$). On the other hand, the proposed design (Design1) has lower normalized speed up for query values greater than 512 and decreases linearly as the query length increases.

Figure 11 shows the distribution of biological sequences by length (number of residues) in the UniProtKBSwiss-Prot database [39]. As of July 2014, the UniprotKB/Swiss-Port database (Release 2014_7) comprises 546,000 sequences or 194,259,968 residues. The average sequence length is 355 amino acids, with the shortest of 2 amino acids and the longest 35,213 residues [39]. We notice from this figure that the largest number of protein sequences have lengths located roughly between 50 and 600 residues. Therefore, most of the query sequence lengths lies in the range less than 512, which makes the proposed design (Design1) more suitable for protein sequence alignment than the conventional design (Design2) as it outperforms the conventional design in speed and area in this sequence lengths range.

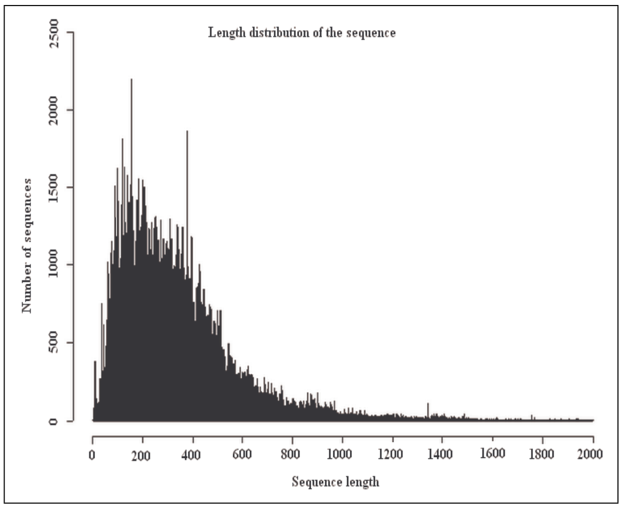


Figure 11: UniProtKB/Swiss-Prot knowledge base sequences by length-distribution [39].

5 Summary and conclusion

This paper presented novel reconfigurable processor array architecture for the Smith-Waterman with affine gap penalty algorithm to align protein sequences with optimal results. This architecture is extracted by applying a nonlinear mapping methodology to the

Smith-Waterman algorithm after expressing it as Regular Iterative Algorithm (RIA). Also, This architecture has been modified using folding technique to enable hardware reuse to avoid replicating processing elements of the processor array in multiple FPGAs. The implementation results showed that the proposed design is more efficient in speed and area - for query length values less than 512 residues - than the conventional processor array architecture previously reported in the literature.

Acknowledgements: The authors would like to acknowledge the support of the deanship of scientific research at Prince Sattam Bin Abdulaziz university under the research project #2014/01/2120.

References:

[1] G. Osamu, An improved algorithm for matching biological sequences, *Journal of Molecular Biology* 162, 1982, pp. 705–708.

[2] D. Benson, I. Karsch-Mizrachi, D. Lipman, J. Ostell, B. Rapp, and D. Wheeler, “Genbank, *Nucleic Acids Research* 28, 2000, pp. 15–18.

[3] T. Oliver, B. Schmidt, and D. Maskell, Reconfigurable architectures for biosequence database scanning on fpgas, *IEEE Transactions on Circuits and Systems II* 52, 2005, pp. 851–855.

[4] Y. Yamaguchi, T. Maruyama, and A. Konagaya, High speed homology search with fpgas, in *The Pacific Symposium on Biocomputing*, 2002, pp. 271– 282.

[5] R. Jacobi, M. Ayala-Rincon, C. Llanos, and R. Hartenstein, Reconfigurable systems for sequence alignment and for general dynamic programming, *Genetics and Molecular Research* 4, 2005, pp. 543–552.

[6] T. V. Court and M. Herbordt, Families of fpga-based accelerators for approximate string matching, *Microprocessors and Microsystems*, 31, 2007, pp. 135–145.

[7] D. Hoang, Fpga implementation of systolic sequence alignment, in *International Workshop on Field Programmable Logic and Applications*, (Vienna, Austria), 1992.

[8] A. Mohamed, E. Esam, and T. Mohamed, Dna and protein sequence alignment with high performance reconfigurable systems, in *Second NASA/ESA Conference on Adaptive Hardware and Systems: IEEE Computer Society*, 2007.

[9] K. Benkrid, L. Ying, and A. Benkrid, A highly parameterized and efficient fpga-based skeleton for pairwise biological sequence alignment, *IEEE Transactions on VLSI Systems* 17, 2009, pp. 561–570.

[10] S. Lloyd and Q. Snell, “Hardware accelerated sequence alignment with traceback,” *International Journal of Reconfigurable Computing*, 2009, pp. 10–18.

[11] X. Meng and V. Chaudhary, Boosting data throughput for sequence database similarity searches on fpgas using an adaptive buffering scheme, *Journal of Parallel Computing* 35, 2009, pp. 1–11.

[12] M. Isa, K. Benkrid, T. Clayton, C. Ling, and A. Erdogan, An fpga-based parameterized and scalable optimal solutions for pairwise biological sequence analysis, in *NASA/ESA Conference on Adaptive Hardware and Systems (AHS’2011)*, 2011.

[13] J. Xian-yang, L. Xinchun, X. Lin, Z. Peiheng, and S. Ninghui, A reconfigurable accelerator for smith waterman algorithm, *IEEE Transactions on Circuits and Systems II* 54, 2007, pp. 1077–1081.

[14] P. Zhang, G. Tan, and G. R. Gao, Implementation of the smith-waterman algorithm on a reconfigurable supercomputing platform, *Altera Corporation*, 2007.

[15] Y. Yamaguchi, H. Tsoi, and W. Luk, Fpga-based smith-waterman algorithm: Analysis and novel design, *Lecture Notes in Computer Science; Reconfigurable Computing: Architectures, Tools and Applications: Springer Berlin Heidelberg* 6578, 2011, pp. 181–192.

[16] M. Isa, K. Benkrid, and T. Clayton, Efficient architecture and scheduling technique for pairwise sequence alignment, *ACM, SIGARCH Computer Architecture News* 40, 2012, pp. 26–31.

[17] T. Smith and M. Waterman, Identification of common molecular subsequences, *J. Mol. Biol* 147, 1981, pp. 195–197.

[18] R. Durbin, S. Eddy, A. Krogh, and G. Mitchison, *Biological Sequence Analysis: Probabilistic Models for Proteins and Nucleic Acids*. Cambridge University Press: Cambridge University, 1998.

[19] S. Rao and T. Kailath, Regular iterative algorithms and their implementation on processor arrays, *Proc. IEEE*, 76, 1988, pp. 259–269.

[20] S. Kung, *VLSI Array Processors*. Englewood Cliffs, N.J.: Prentice- Hall, 1988.

[21] E. Abdel-Raheem, *Design and VLSI Implementation of Multirate Filter Banks*. PhD thesis, University of Victoria, Victoria, BC, 1995.

[22] F. El-Guibaly and A. Tawfik, Mapping 3d iir digital filter onto systolic arrays, *Multidimensional Systems and Signal Processing* 7, 1996, pp. 7–26.

[23] A. Ibrahim, F. Gebali, H. El-Simary, and A. Nassar, Processor array architectures for scalable radix 4 montgomery modular multiplication algorithm, *IEEE Trans. on Parallel and Distributed Systems* 22, 2011 pp. 1142–1149.

[24] A. Refiq and F. Gebali, Processor array architectures for deep packet classification, *IEEE Trans. on Parallel and Distributed Systems*, 17, 2006, pp. 241–252.

[25] D. Moldovan and J. Fortes, Partitioning and mapping of algorithms into fixed size systolic arrays, *IEEE Trans. on Computers* 35, 1986, pp. 1–12.

[26] P. Guerdoux-Jamet and D. Lavenier, Samba, hardware accelerator for biological sequence comparison, *Bioinformatics* 13, 1997, pp. 609–615.

[27] E. Chow, J. Peterson, M. Waterman, T. Hunkapiller, and A. Zimmermann, "systolic array processor for biological information signal processing," in *Proc. of the 5th International Conference on Supercomputing (ICS 91)*, (USA), 1991, pp. 216–223.

[28] C. White and etl., Bioscan, vlsi-based system for bio-sequence analysis, in *Proc. Of IEEE International Conference on Computer Design (ICCD 91)*, 1991, pp. 504–509.

[29] T. Han and S. Parameswaran, Swasad, an asic design for high speed dna sequence matching, in *Proc. of the 2002 Asia and South Pacific Design Automation Conference (ASP- DAC 02)*, (USA), 2002, pp. 541–546.

[30] M. Gokhale and etl., "Splash: a reconfigurable linear logic array," in *Int. Conf. on Parallel Processing*, 1990, pp. 526–532.

[31] G. Caffarena, C. Pedreira, C. Carreras, S. Bojanic, and O. Nieto-Taladriz, "Fpga acceleration for dna sequence alignment," *J. Circuits Syst. Compu* 16, 2007, pp. 245–266.

[32] T. Oliver, B. Schmidt, and D. Maskell, Hyper customized processors for bio-sequence database scanning on fpgas, in *Proc. Of 13th International Symposium Field-Programmable Gate Arrays (FPGA05)*, 2005, pp. 229–237.

[33] S. Guccione and E. Keller, Gene matching using jbits, in *Proc. of 12th Int. Conference on Field- Programmable Logic and Applications (FPL 02)*, 2002, pp. 1168–1171.

[34] C. Bio, White paper on clc bioinformatics cube 1.03, in *Technical Report, CLC Bio*, (Denmark), 2007.

[35] N. Sebastio, T. Dias, N. Roma, and P. Flores, Integrated accelerator architecture for dna sequences alignment with enhanced traceback phase, in *International Conference on High Performance Computing and Simulation (HPCS)*, 2010, pp. 16–23.

[36] S. Lloyd and Q. Snell, Sequence alignment with traceback on reconfigurable hardware, in *International Conference on Reconfigurable Computing and FPGAs (ReConFig 08)*, 2008, pp. 259–264.

[37] N. Sebastio, T. Dias, N. Roma, and P. Flores, Hardware accelerator architecture for simultaneous short-read dna sequences alignment with enhanced traceback phase, *Microprocessors and Microsystems* 36, 2012, pp. 96–109.

[38] J. Arram, k. Tsoi, W. Luk, and P. Jiang, Hardware acceleration of genetic sequence alignment, *Reconfigurable Computing: Architectures, Tools and Applications, Lecture Notes in Computer Science* 7806, 2013, pp. 13–24.

[39] UniProtKB/Swiss-Prot, Uniprotkb/swiss-prot protein knowledgebase release 2014_07. http://web.expasy.org/docs/relnotes/relstat.html, 2014.