A parallel FPGA design of the Smith-Waterman traceback

Zubair Nawaz #1, Muhammad Nadeem #2, Hans van Someren *3, Koen Bertels #4

# *Computer Engineering Lab, Delft University of Technology*
The Netherlands
1 z.nawaz@tudelft.nl
2 m.nadeem@tudelft.nl
4 k.l.m.bertels@tudelft.nl

* *ACE Associated Compiler Experts bv*
The Netherlands
3 hvs@ace.nl

***Abstract*—The Smith-Waterman (SW) algorithm is the only optimal local sequence alignment algorithm. There are many SW implementations on FPGA, which show speedups of up to 100x as compared to a general-purpose-processor (GPP). In this paper, we propose a design of the SW traceback, which is done in parallel with the matrix fill stage and which gives the optimal alignment after once scanning through the whole database. Beside that, we have proposed the hardware design for the RVEP SW FPGA implementation, which demonstrates that this solution can be realized with off-the-shelf FPGA boards.**

I. Introduction

SEQUENCE alignment is one of the most widely used operations in computational biology. The Smith-Waterman (SW) algorithm [1] is the only optimal algorithm to find the local sequence alignment.

There are two stages in the Smith-Waterman (SW) algorithm namely matrix fill and traceback. First, we fill the matrix with the optimal score found, then we find the maximum of the optimal score [2]. Finally we perform the traceback starting from the maximum value. This procedure is performed for all the sequences in the database. Since the matrix fill stage takes 98.6% of the overall time [3], all FPGA implementations use FPGAs for accelerating the matrix fill stage.

There are two methods to perform the sequence alignment on a reconfigurable system. In the first method, the optimal value matrix is filled on an FPGA and then the matrix data is sent to the GPP, where the traceback is performed. This method creates a memory bottleneck in any off-the-shelf FPGA. In the second method, a sequence is shortlisted by finding the maximum value after performing the matrix fill stage for the whole database. Later, that maximum value and the index of the corresponding sequence is transferred to the GPP. The matrix fill stage for the shortlisted sequences is repeated on the GPP and the traceback is performed to get the optimal alignment.

Several people have worked on approaches using the first method. Hoang and Lopresti [4] gave a linear systolic array implementation on a SPLASH reconfigurable logic array, in which the data of the matrix fill was stored in memory and then a traceback was performed. They only used the edit distance formula, which is a special case of Smith-Waterman algorithm. This method requires substantial memory bandwidth which is not available, as will be shown in Section III. Yamaguchi et al. [5] and Moritz et al. [6] implemented SW on a linear systolic array. They both applied compression and saved direction vectors of 2 bits for each element instead of 16 bits. The compression reduced the memory bandwidth requirement, However, still it was too high to be implemented using off-the-shelf FPGA boards and became the bottleneck as described later in Section III.

Most of the implementations follow the second method, in which FPGAs are only used to find the maximum value after filling the matrix [7], [8], [9], [10], [11].

Our implementation is more close to the first method. Our goal was to avoid the memory bandwidth problem such that off the shelf FPGAs can be used.

In this paper, we propose a parallel FPGA design of the SW traceback, which gives the alignment immediately after completing the matrix fill for all the sequences in the database. This way, we can avoid the second matrix fill stage for the shortlisted sequences at the expense of more area consumption. It can be easily implemented on off-the-shelf FPGA boards as it uses the bandwidth within limits of the current FPGA boards. The main benefits of the proposed technique are as follows:

1) The proposed solution gives the alignment after scanning the database once. We show that the bandwidth requirements is within the limits of current day FPGAs.
2) The whole solution can be easily implemented as a pure FPGA based implementation without needing a GPP.
3) Our solution is generic and can be used to design hardware for any dataflow systolic array implementation.

In this paper, we propose a hardware design for an RVEP SW [2] implementation, which has a higher bandwidth requirement than the classical dataflow implementation for the same size of matrix. Nevertheless, this design can be easily adapted to address the bandwidth issue for a dataflow systolic array implementation.

The rest of the paper is organized as follows. In the next section, we describe the background to understand the SW problem, the classical dataflow implementation and the way RVEP is applied. Section III describes the resulting memory bottleneck problem. Section IV gives the overview of the proposed solution. Our hardware design for avoiding the memory bottleneck in RVEP SW is proposed in Section V. Section VI describes the memory bandwidth requirement for our solution and compares it with the normal bandwidth requirement. Finally, the paper is concluded in Section VII.

II. Background

A. The Smith-Waterman algorithm

Let $S[1..m]$ and $T[1..n]$ be two sequences of length m and n for sequence alignment. The *optimal alignment score* $F(i,j)$ for two sub-sequences $S[1..i]$ and $T[1..j]$ is given by the following recurrence equation.

$$F(i,j) = \max \begin{cases} F(i,j-1)+g \\ F(i-1,j-1)+x(i,j) \\ F(i-1,j)+g \\ 0 \end{cases} \tag{1}$$

where $F(0,0) = F(0,j) = F(i,0) = 0$, for $1 \le i \le m$ and $1 \le j \le n$. The $x(i,j)$ is the score for match/mismatch, depending upon whether $S[i] = T[j]$ or $S[i] \neq T[j]$. g is some constant penalty for inserting a gap in any sequence.

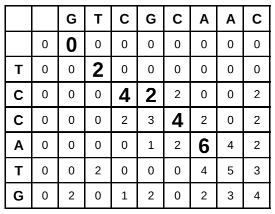
The two sequences to be aligned are placed along the row and column of the matrix. The matrix is filled row-wise starting from the top-left corner. After the matrix is filled, a traceback is performed starting from the maximum value in the matrix, like 6 from Figure 1. The traceback traverses to one of the three elements from which its alignment score is computed. This process is repeated till the score drops below a certain threshold or to zero. In the traceback phase, if the corresponding row and column element match then the alignment is computed from the top-left element otherwise it is computed from any of the three elements depending on which of them produces a maximum. When an element is computed from the top element then there is a gap in the sequence along the row and similarly when an element is computed from the left element then there is a gap in the sequence along the column. The local optimal alignment for the example in Figure 1 is as follows.

```
TCGCA
| | | | |
TC-CA
```

The next two sections briefly describe the two parallel implementations of SW on the FPGAs.

B. Classical dataflow implementation

To parallelize the matrix fill in the SW algorithm we need to look at its data dependence graph as shown in Figure 2. Blank circles are the elements after the initialization with the boundary conditions. Any iteration (i,j) cannot be executed until iterations $(i-1,j)$, $(i-1,j-1)$ and $(i,j-1)$ are executed first, due to the data dependences. However, if we traverse the elements in a wavefront manner starting from the top-left corner as shown in Figure 2, all the elements in the diagonal can be executed in parallel. This parallel execution is called *dataflow* implementation, as all the computations are done when their data is available.



		G	T	C	G	C	A	A	C
	0	**0**	0	0	0	0	0	0	0
T	0	0	**2**	0	0	0	0	0	0
C	0	0	0	**4**	**2**	2	0	0	2
C	0	0	0	2	3	**4**	2	0	2
A	0	0	0	0	1	2	**6**	4	2
T	0	0	2	0	0	0	4	5	3
G	0	2	0	1	2	0	2	3	4

Figure 1. Matrix for an example of the SW algorithm, when $g = -2$ and $x(i,j) = +2$ when *S[i]=T[j]* otherwise -1. Elements in the traceback are shown in bold.

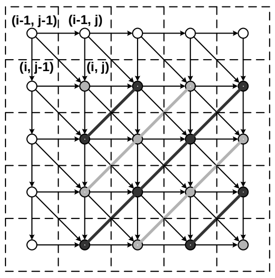


Figure 2. Data dependence graph for Equation 1 (different shades of gray in circles show the elements which can be executed in parallel).

C. Smith-Waterman RVEP implementation

RVEP [12] is like computing anti-diagonals of blocks in dataflow manner instead of anti-diagonals of elements in classical dataflow. This is shown in Figure 3, which shows the sequence of fill for the blocks with blocking factor $B = 2$. The size of the block is $b = B \times B$. Each block contains four elements O1, O2, O3 and O4. In Figure 3, first it computes all elements in the block with cycle 1, then it computes all elements in the anti-diagonal of blocks with cycle 2 and so on. This structure suggests us to use a linear systolic array to do the computation, where a same block circuit is used systolically for the computation of blocks in the same column as the anti-diagonal of blocks progresses.

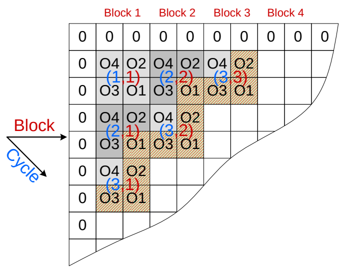


Figure 3. Sequence of fill for $B = 2$. Blocks at the top show the hardware to be used in each column. Number in red represent the block used and the number in blue represent the cycle in which it is used.

Table I
BANDWIDTH REQUIREMENT FOR DIFFERENT IMPLEMENTATIONS

Implementations	p	d_w	Freq.	Time I	Bandwidth I
		bits	MHz.	sec	Gb/sec
Zhang07 [10]	384	20	66.7	1.56×10^{-4}	49.36
Oliver05 [13]	252	16	55	1.86×10^{-4}	27
Jiang07 [8]	80	20	82	1.23×10^{-4}	13.04
Nawaz10 [2]	40	16	79.3	6.33×10^{-5}	12.6

III. MEMORY BOTTLENECK

In this section, we describe the memory bandwidth bottleneck that arises when we use the first method. Here, we assume to store only the direction vector, which is of 2 bits as in [5], [6]. A double buffering technique can be used, in which one can keep two copies of the direction matrix in which one stores the data alternatingly. When the FPGA is computing the next sequence alignment and storing that result in one buffer, the other buffer can be transferring its content to the shared memory for later traceback use by the GPP. The time to transfer the direction matrix from memory should not be more than the matrix fill time for the next pair of sequences, so that the memory bandwidth does not become the bottleneck.

In order to better quantify the memory bottleneck problem, we discuss the memory requirements for different implementations. In Table I, we present the numbers for four different SW implementations. The first three are dataflow implementations and the last is the RVEP implementation. p represents the maximum number of PEs that can be accommodated on the FPGA available in the respective implementation. We take $m = 10000$, which is a higher end value for a sequence, as only 13 out of 468851 protein sequences are longer than 10000 symbols in the UniProt database and 99.5% of the sequences are less than 1000 symbols [14]. The storage size of an element in the optimal value matrix is defined as d_w. The frequency used for computing each element in case of a dataflow implementation and for computing a block in case of the RVEP implementation is given under Frequency. Time I is the time needed to fill the whole matrix for one sequence alignment by the respective implementations. Bandwidth I in Table I is computed for the transfer of the direction matrix and is computed by the following formula:

$$\text{Bandwidth I} = \frac{2pm}{\text{Time I}} \tag{2}$$

When combining all these data, we obtain a bandwidth requirement of up to 49.36 Gb/sec which is more than what is available on even the largest FPGA's. The Xilinx-6 FPGA Connectivity Development Kit enables advanced connectivity designs with PCI Express 1.1/2.0, Ethernet, SATA and other proprietary high-speed serial protocols with line rates up to 6.5 Gb/sec [15].

IV. COMPRESSION AND BACKTRACKING

In order to solve the bottleneck as quantified above, we propose to perform back tracking in addition to the compression. Even though this twofold solution is presented here in the context of the RVEP for Smith Waterman, it can be easily modified to be useful for any classical dataflow implementation. In this section, we are using $B = 2$, $p = 40$, $m = 10000$, $n = 1000$ and $d_w = 16$, where B is the blocking factor, p represents the maximum number of PEs that can be accommodated on the FPGA, n and m are the lengths of the sequences and d_w is the data width of the optimal values in bits.

		G	T	C	G	C	A	A	C
	0	**0**	0	0	0	0	0	0	0
T	0	0	**2**	0	0	0	0	0	0
C	0	0	0	**2**	**1**	2	0	0	2
C	0	0	0	3	2	**2**	1	0	2
A	0	0	0	0	3	3	**2**	1	1
T	0	0	2	0	0	0	3	2	1
G	0	2	0	2	2	0	3	3	2

Figure 4. Direction matrix for example in Figure 1

During the matrix fill, the elements in a block can be computed only from the adjacent elements in the preceding block [2]. As depicted in Figure 3, all the elements in the anti-diagonal of blocks in cycle 4 can be computed using the adjacent elements from the preceding anti-diagonal of blocks, as shown by the pattern filled elements. Hence to compute the optimal score for the current anti-diagonal of blocks, we only need to store the optimal score data for the adjacent elements from the preceding anti-diagonal in the BRAM using a FIFO buffer. We thus avoid needing to store the entire matrix of optimal values. The size of memory required to store this is $p \times 2B \times d_w = 20 \times 2 \times 2 \times 16 = 1280$ bits.

Instead of the optimal value matrix, we only store the direction matrix which contains direction vectors to construct the sequence alignment. There are only 3 directions from which an element can be computed. So similar to [5], [6], only 2 bits are needed to indicate the direction it is computed from. As described earlier in Section II-A, the traceback is stopped beyond a threshold value. We give the direction value 0 in the direction matrix for the corresponding threshold value in an optimal value matrix. Similarly we fill a 1,2 or 3 value in the current element of the direction matrix, if the current element is computed from the left, top left or top element respectively. The maximum value 6 in the matrix from Figure 1 is computed from the top-left element, therefore the corresponding element in the direction matrix contains 2 as shown in Figure 4. The required storage space is determined by two factors: the first is the row to keep the optimal values given by $p \times 2B \times d_w$ and second to keep the direction matrix which is $2nm$. So the total space required in BRAM is $p \times 2B \times d_w + 2nm = 1280 + 2 \times 10^7$ bits, which is less than $d_w(nm) = 16 \times 10^7$ bits which are required without compression. Now there are FPGAs available with 5×10^7 bits BRAM [16].

Since the bandwidth requirement is still high after compression, we propose to move the traceback stage to the FPGA, which further reduces the required bandwidth. It means the complete solution is now on the FPGA only, which is easier

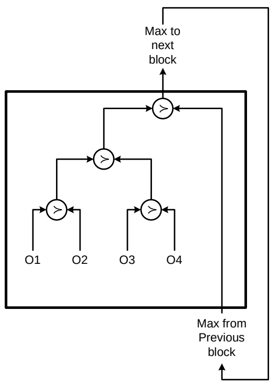


Figure 5. Block Max.

to maintain in one place. So, the traceback is performed on the FPGA and alignment results are sent to the main memory, which are far less than the whole matrix. We perform another task, which starts the traceback from the direction value in the direction matrix corresponding to the maximum value in the optimal value matrix in BRAM and transfer only direction vectors which come across the traceback path. The length of the traceback path is $O(max(m, n))$, which is the worst case and usually the length of the traceback is far less than this.

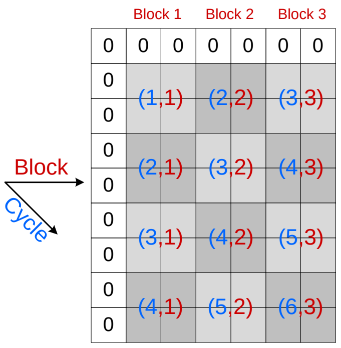
V. Design Overview

In this section, we present the design overview for the proposed technique to reduce the memory bottleneck problem in case of RVEP implementation of SW with $B = 2$, when traceback stage is done in parallel with the matrix fill stage. This design will be on top of the circuit for optimal value computation. The proposed design is composed of computing the maximum in the optimal value matrix, generating the corresponding direction matrix, storing direction values in BRAM and finally doing the traceback on the direction vectors starting from the maximum optimal value. This implementation can be easily modified for RVEP with higher blocking factors. The details are as following:

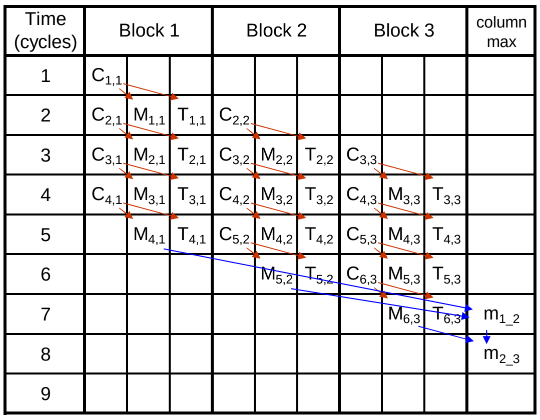
A. Computing max in the optimal value matrix

We need to find the maximum value in the optimal value matrix. We do it by first finding the maximum of the block and then finding the maximum among all the blocks in a block column and then finding the maximum value among all the block columns to find the maximum of the matrix.

As mentioned earlier, we compute the optimal values in a systolic array. After the optimal values are computed for a block, we compute the maximum of the optimal values from the previous and current blocks in the same block column by using the circuit in Figure 5. In the meanwhile, the optimal values for the next block in the same block column are computed systolically by the computation block. This continues till the optimal values for the column block and later in the next cycle the maximum of the whole column



(a) Computation block. A number in red represent the block used and the number in blue represents the cycle in which it is used.



Time (cycles)	Block 1			Block 2			Block 3			column max
1	$C_{1,1}$									
2	$C_{2,1}$	$M_{1,1}$	$T_{1,1}$	$C_{2,2}$						
3	$C_{3,1}$	$M_{2,1}$	$T_{2,1}$	$C_{3,2}$	$M_{2,2}$	$T_{2,2}$	$C_{3,3}$			
4	$C_{4,1}$	$M_{3,1}$	$T_{3,1}$	$C_{4,2}$	$M_{3,2}$	$T_{3,2}$	$C_{4,3}$	$M_{3,3}$	$T_{3,3}$	
5		$M_{4,1}$	$T_{4,1}$	$C_{5,2}$	$M_{4,2}$	$T_{4,2}$	$C_{5,3}$	$M_{4,3}$	$T_{4,3}$	
6					$M_{5,2}$	$T_{5,2}$	$C_{6,3}$	$M_{5,3}$	$T_{5,3}$	
7								$M_{6,3}$	$T_{6,3}$	m_{1_2}
8										m_{2_3}
9										

(b) Computation sequence for the computation block in Figure 6a, C stands for Computing the optimal value, M for finding the Maximum of the optimal value in a block and T for generating the direction vectors for elements in the block. Red arrows show the computation of maximum blocks and direction vectors for the corresponding computation block. Blue arrows show the computation of maximum of block columns.

Figure 6. Computation block and the sequence to compute it

is computed. The resource and time used for computing the optimal value and then the maximum for the matrix shown in Figure 6a is given in Figure 6b. It shows that first the optimal value for block $(1, 1)$ i.e. $C_{1,1}$ is computed. In the next cycle, $C_{2,1}$ is computed systolically and the maximum of block $(1, 1)$, i.e. $M_{1,1}$ is computed. The maximum of block column 1 is computed in cycle 5 and similarly the maximum of block column 2 is computed in cycle 6. The maximum of block column 1 and 2 is computed in cycle 7 by using a single comparator as given by m_{1_2}. At the same cycle 7, the maximum of the column 3 is computed and then the same comparator that is used to compute m_{1_2} in cycle 7 is used to compute the maximum of block 3 and all block columns < 3, we call it m_{2_3} in cycle 8. This way the maximum of all elements of the optimal value matrix is computed.

B. Generating the direction matrix

The direction vectors for a direction matrix block are computed after the computation of the optimal value block. In Figure 6b, the direction vectors $T_{i,j}$ for any block (i, j) are

```
if (F[i,j]=0)
   output=0
else if (F[i,j]=F[i,j-1]+a)
   output=1
else if (F[i,j]=F[i-1,j]+a)
   output=3
else
   output=2
```

Figure 7. Code to generate the direction vector for an element (i, j)

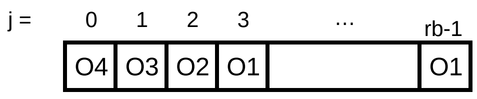


Figure 8. Elements stored in BRAM

computed in the next cycle to $C_{i,j}$. Similar to computing optimal values and computing a maximum for a block, direction vectors of a block are also computed systolically.

The pseudo-code to generate a direction value for an element is shown in Figure 7.

C. Storing direction vectors in BRAM

After the direction vectors for a block have been generated, they are stored in BRAM as intermediate result. In this section we are considering $n = 8$, $B = 2$, $b = 4$ and $r = \frac{n}{B} = 4$ as an example. We choose BRAM width $= 2rb = 32$ bits to store $rb = 16$ elements. The way elements are stored is shown in Figure 8. This whole BRAM width is filled in one cycle using one write port. The data is always filled starting from 0.

D. Traceback

The traceback is started from the point in the direction matrix corresponding to the maximum value of the optimal value matrix. The data is stored in a different coordinate system (rotated at 45 degree to horizontal) as compared to what was earlier suggested by the formula in Equation 1 and secondly each 2-dimensional block is linearized as shown in Figure 8. We need to translate the traceback formula accordingly.

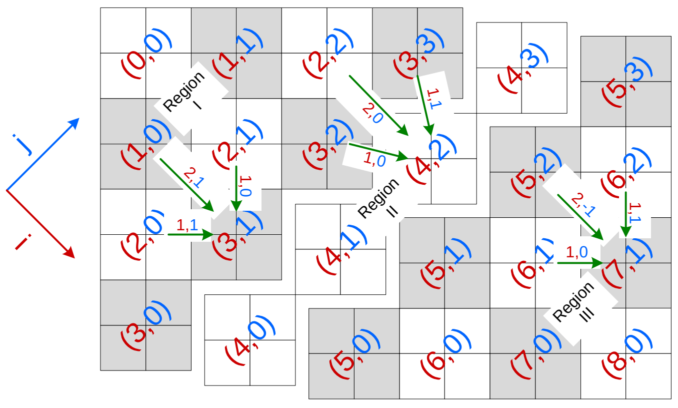


Figure 9. Classification of BRAM according to the direction vectors among the neighboring blocks, Region I: $i < r$; Region II: $i = r$ and Region III: $i > r$, here $r = 4$

```
GetTrace(i,j,trace){
  k = j mod b
  if ( ( (i<r) and ((j>2) and
  !( (j>=i*4) and (j<i*4+3) ) ) )
  or ( (i>=r) and ((j>2) and !( (j>=r*4-4)
  and (j<r*4-1) ) ) ) and (m>0) {
   I = GetI(k,m(i,j)) // Get I from Figure 11
   J = GetJ(k,m(i,j)) // Get J from Figure 11
   trace = trace + m(i,j) // + is concatenation
   GetTrace(I,J,trace)
  } else {
  trace = trace + m(i,j)
 }
}
```

Figure 10. Pseudo-code for traceback

	m(i,j)		
	1	2	3
k=0	I=i-1	I=i-2	I=i-1
	J=j-2+d	J=j-1+e	J=j+1+d
k=1	I=i-1	I=i-1	I=i
	J=j-2+d	J=j-3+d	J=j-1
k=2	I=i	I=i-1	I=i-1
	J=j-2+d	J=j+1+d	J=j+1+d
k=3	I=i	I=i	I=i
	J=j-2+d	J=j-3	J=j-1

where d=0,e=0 if i<r; d=b, e=b if i=r and d=b, e=2b if i>r

Figure 11. BRAM Address translation

There are three regions classified by the direction vectors between the blocks as shown in Figure 9. The traceback is possible to three neighboring blocks and its direction vectors are different in different regions. The direction vectors among the neighboring blocks remain the same in each region and hence the traceback formula too. The traceback code, which also takes care of each of these regions is given in Figure 10. The first call to the traceback code is GetTrace(i,j,ε), where ε is the empty string and (i, j) is the position of the maximum value in the optimal value matrix. The trace variable in Figure 10 will finally give the traceback path from which alignment can be easily computed. The traceback code has already a linear time complexity and takes less time than the time taken for matrix fill by dataflow or RVEP. The address translation is summarized in Figure 11. The $m(i, j)$ refers to the direction value at the (i, j) position in the BRAM.

VI. Experimental Validation

In order to assess the impact of our compression scheme and traceback, we explore two extreme cases of memory transfer. The memory bandwidth required for these two cases is given in Table II, which is a extension of Table I. The first case, considered as the worst case is as follows. We keep two direction matrices for double buffering to do the sequence alignment continuously. When one matrix is being filled, the other can be used to transfer the traceback result of the sequence alignment done recently. We generate the direction matrix for every alignment one after the other; however, we do

Table II
COMPARISON OF BANDWIDTH REQUIREMENT FOR DIFFERENT IMPLEMENTATIONS

Implementations	$p.$	d_w	Freq.	Time I	Time II	Bandwidth I	Bandwidth II	Bandwidth III
		bits	MHz.	sec	sec	Gb/sec	Mb/sec	Kb/sec
Zhang07 [10]	384	20	66.7	1.56×10^{-4}	2.48	49.36	128.8	8
Oliver05 [13]	252	16	55	1.86×10^{-4}	3.02	27	107.2	6.64
Jiang07 [8]	80	20	82	1.23×10^{-4}	2.02	16.3	162.4	9.92
Nawaz10 [2]	40	16	79.3	6.33×10^{-5}	1.13	12.6	316	17.6

not transfer the traceback direction vector for every alignment. We find the maximum optimal score for each sequence in the database and send its traceback direction vector only when the current maximum optimal score is greater than the global maximum.

In the worst case, we need to transfer the traceback vector for every comparison when every next sequence in the database has a higher maximum optimal value than the current sequence. In that case, we need to send $O(2 \times max(m, n))$ bits for some unit time (which is equal to the time to fill the matrix, Time I in Table II) instead of $O(2nm)$ bits, which is linear as compared to quadratic in normal case. The chances of this worst case are close to impossible. We have also computed the bandwidth requirement for our proposed technique keeping in mind the worst case referred as Bandwidth II in Table II. Here, the maximum bandwidth is 316 Mb/sec which is easily achievable in normal FPGA boards.

The best case is described as follows. We find the maximum optimal value in the optimal value matrix for all the sequences in the database and send that maximum optimal value and its corresponding traceback direction vector to main memory. In this scenario, the traceback path and maximum is sent once after the whole scanning of the database, which is 468851 protein sequences in case of UniProt [14]. The time it takes to scan the whole database is given under the header Time II. The memory bandwidth requirement for this implementation using Time II is shown as Bandwidth III in II. The maximum bandwidth under heading Bandwidth III is 17.6 Kb/sec.

The implementation strategy can be changed easily, if there is a requirement for computing some *k*-best alignments from the database. The bandwidth for any such strategy will fall in between Bandwidth II and Bandwidth III.

Our solution is better than the traditionally used solution [13], [7], [9] in the sense that it gives the best alignment between the unknown sequence and the known sequence in the database after once scanning through the database and there is no need to repeat the sequence alignment for some smaller subset. Secondly, the whole solution is based on FPGA, so there is no need to maintain the solution at two different places, which is easier.

VII. CONCLUSION

In this paper, we have proposed a parallel FPGA design of the SW traceback, which constructs the optimal alignment between the unknown sequence and its genetically closest known sequence from the database. We have seen how a naive approach to parallel design can lead to bandwidth problems. Our solution addresses this issue, and we have proposed a hardware design for a SW RVEP implementation that can be easily extended to any other dataflow systolic array implementation. The proposed solution can be easily implemented on current off-the-shelf FPGA boards. In the future, we will implement our memory bandwidth reduction solution, and integrate it with the matrix fill solution for both dataflow and RVEP implementations to find the hardware overhead of our proposed scheme.

REFERENCES

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

[2] Z. Nawaz, H. Sumbul, and K. Bertels, “Fast smith-waterman hardware implementation,” in *RAW 2010*, April 2010.

[3] O. Storaasli and D. Strenski, “Experiences on 64 and 150 fpga systems,” in *Proceedings of the Fourth Annual Reconfigurable Systems Summer Institute (RSSI’08)*, 2008.

[4] D. T. Hoang and D. P. Lopresti, “Fpga implementation of systolic sequence alignment,” in *International Workshop on Field Programmable Logic and Applications*, 1992.

[5] Y. Yamaguchi, Y. Miyajima, T. Maruyama, and A. Konagaya, “High speed homology search using run-time reconfiguration,” in *FPL ’02*, (London, UK), pp. 281–291, Springer-Verlag, 2002.

[6] G. L. Moritz, C. Jory, H. S. Lopes, and C. R. E. Lima, “Implementation of a parallel algorithm for protein pairwise alignment using reconfigurable computing,” pp. 1 –7, sep. 2006.

[7] P. Faes *et al.*, “Scalable hardware accelerator for comparing dna and protein sequences,” in *InfoScale ’06*, (New York, NY, USA), p. 33, ACM, 2006.

[8] X. Jiang, X. Liu, L. Xu, P. Zhang, and N. Sun, “A reconfigurable accelerator for smith-waterman algorithm,” *IEEE Transactions on Circuits and Systems II*, vol. 54, pp. 1077–1081, 2007.

[9] O. O. Storaasli and D. Strenski, “Exploring accelerating science applications with fpgas,” in *RSSI’07*, 2007.

[10] P. Zhang, G. Tan, and G. R. Gao, “Implementation of the smith-waterman algorithm on a reconfigurable supercomputing platform,” in *HPRCTA ’07*, 2007.

[11] K. Benkrid, Y. Liu, and A. Benkrid, “A highly parameterized and efficient fpga-based skeleton for pairwise biological sequence alignment,” *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, vol. 17, pp. 561 –570, apr. 2009.

[12] Z. Nawaz, T. P. Stefanov, and K. Bertels, “Efficient hardware generation for dynamic programming problems,” in *ICFPT’09*, December 2009.

[13] T. Oliver, B. Schmidt, and D. Maskell, “Reconfigurable architectures for bio-sequence database scanning on fpgas,” *IEEE Transactions on Circuits and Systems II: Express Briefs*, vol. 52, no. 12, pp. 851–855, Dec. 2005.

[14] E. J. Houtgast, “Scalability of bioinformatics applications for multicore architectures,” Master’s thesis, T U Delft, 2009.

[15] “New xilinx virtex-6 and spartan-6 fpga connectivity development kits. online: www.xilinx.com/products/devkits/EK-V6-ML605-G.htm.”

[16] “Stratix v fpga family overview. online: http://www.altera.com/products/devices/stratix-fpgas/stratix-v/overview/stxv-overview.html.”