Hardware Implementation of the Smith-Waterman Algorithm Using Recursive Variable Expansion

Laiq Hasan   Zaid Al-Ars   Zubair Nawaz   Koen Bertels
Delft University of Technology
Computer Engineering Laboratory
Mekelweg 4, 2628 CD Delft, The Netherlands
L.Hasan@ewi.tudelft.nl

Abstract: *In this paper we adapted a novel approach for accelerating the Smith-Waterman (S-W) algorithm using Recursive Variable Expansion (RVE), which exposes extra parallelism in the algorithm, as compared to any other technique. The results demonstrate that applying the recursive variable expansion technique speeds up the performance by a factor of 1.36 to 1.41, as compared to traditional acceleration approaches at the cost of using 1.25 to 1.28 times more hardware resources.*

Keywords: *Sequence Alignment, Smith-Waterman Algorithm, Systolic Array, Recursive Variable Expansion, FPGA*

1 Introduction

Sequence alignment is an important activity in the field of bioinformatics that enables us to compare DNA strands with each other and promises to help us understand possible genetically transmitted diseases. *Smith-Waterman (S-W)* is the most accurate sequence alignment algorithm available, but its computational complexity makes it very slow in real applications [1]. Faster algorithms like FASTA [2] and BLAST [3] are available, but they achieve high speed at the cost of reduced accuracy. Thus it is highly desirable to accelerate the S-W algorithm in hardware.

Various approaches have been adapted to accelerate the S-W algorithm by implementing either the whole algorithm or some parts of it in hardware and compare the performance with the software-only implementation [4], [5], [6], [7], [8], [9]. An overview of such approaches is given in [10].

This paper adapts a novel approach for accelerating the S-W algorithm using *Recursive Variable Expansion (RVE)*, and compares the results with the implementation using a traditional acceleration approach. The speedups thus achieved are reported in the paper.

The remainder of the paper is organized as follows: Section 2 gives a brief description of the S-W algorithm, discusses its inherent data dependencies and briefly explains the RVE approach. Section 3 discusses the implementation using the traditional acceleration approach and the results thus obtained. Section 4 demonstrates the results obtained by applying the RVE technique. Section 5 discusses the results obtained and their significance in comparison with the related work. Section 6 provides a brief conclusion.

2 Background

Based on *dynamic programming (DP)* [11], the S-W algorithm [1] is a method used for local sequence alignment (i.e., identifying common regions in sequences that share local similarity characteristics). In the following subsections we give a brief description of the algorithm, its inherent data dependencies and a brief discussion about the RVE approach.

2.1 S-W Description

When calculating the local alignment, a matrix $H_{i,j}$ is used to keep track of the degree of similarity between the two sequences to be aligned (A_i and B_j). Each element of the matrix $H_{i,j}$ is calculated according to the following equation:

$$H_{i,j} = \max \begin{cases} 0 \\ H_{i-1,j-1} + S_{i,j} \\ H_{i-1,j} - d \\ H_{i,j-1} - d \end{cases} \tag{1}$$

where $S_{i,j}$ is the similarity score of comparing sequence A_i to sequence B_j and d is the gap penalty. The whole algorithm is divided into the following three steps:

1. Initialization step
2. Matrix fill step
3. Trace back step

The matrix is first initialized with $H_{0,j} = 0$ and $H_{i,0} = 0$, for all i and j. This is referred to as the *initialization step*. After the initialization, a *matrix fill step* is carried out using Equation 1, which fills out all entries in the matrix. The final step is the *trace back step*, where the scores in the matrix are traced back to inspect for optimal local alignment. The trace back starts at the cell with the highest score in the matrix and continues up to the cell, where the score falls down to a predefined minimum threshold. In order to start the trace back, the algorithm requires to find the cell with the maximum value, which is done by traversing the entire matrix.

The time complexity of the initialization step is $O(M + N)$, where M is the number of rows and N is the number of columns in the matrix. During the matrix fill step, the entire $H_{i,j}$ matrix needs to be filled according to Equation 1, making its time complexity equal to the number of cells in the matrix or $O(MN)$. The time complexity of the traceback is also $O(MN)$, as the entire matrix needs to be traversed during this step. Thus the total time complexity of the S-W algorithm is $O(M+N)+O(MN)+O(MN) = O(MN)$. The total space complexity of the S-W algorithm is also $O(MN)$, as it fills a single matrix of size MN.

In order to reduce the $O(MN)$ complexity of the matrix fill stage, multiple entries of the $H_{i,j}$ matrix can be calculated in parallel. This is however complicated by data dependencies, whereby each $H_{i,j}$ entry depends on the values of three neighboring entries $H_{i,j-1}$, $H_{i-1,j}$ and $H_{i-1,j-1}$, with each of those entries in turn depending on the values of three neighboring entries, which effectively means that this dependency extends to every other entry in the region $H_{x,y} : x \leq i,\ y \leq j$. This implies that it is possible to simultaneously compute all the elements in each anti diagonal, since they fall outside each others data dependency regions. Figure 1 shows a sample $H_{i,j}$ matrix for two sequences, with the bounding boxes indicating the elements that can be computed in parallel. The right bottom cell is highlighted to show that its data dependency region is the entire remaining matrix. The dark diagonal arrow indicates the direction in which the computation progresses. At least 9 cycles are required for this computation, as there are 9 bounding boxes representing 9 anti diagonals and a maximum of 5 cells may be computed in parallel.

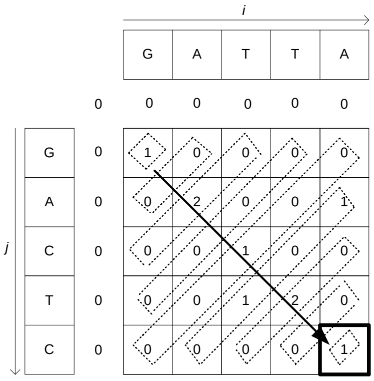

Figure 1: A sample $H_{i,j}$ matrix, where the dotted rectangles show the elements that are computed in parallel.

The degree of parallelism is constrained to the number of elements in the anti diagonal and the maximum number of processing elements required will be equal to the number of elements in the longest anti-diagonal (l_d), where

$$l_d = \min(M, N) \tag{2}$$

Here, we have assumed that the processing elements are equal in number to the length of the shorter sequence. Theoretically, the lower bound to the number of steps required in this parallel implementation, equal to the number of anti-diagonals required to reach the bottom-right element, is $m + n - 1$ [12].

So far this is the best technique for parallelization and has been used by many researchers [13], [14], [5]. Figure 2 shows the implementation to compute an element of the $H_{i,j}$ matrix. This unit contains three adders, a sequence comparator circuit (*SeqComp*) and three max operators. The sequence comparator circuit compares the corresponding characters of two input sequences and outputs a match/mismatch score, depending on wether the two characters are equal or not. Each max operator compares its inputs and outputs the maximum of the two. The time to compute an element is 4 cycles. We have assumed that the time for each cycle is equal to the latency of one add or compare operation.

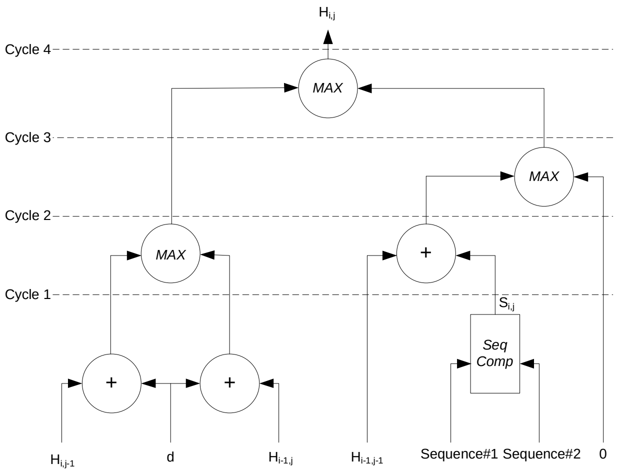

Figure 2: Circuit to compute an element in the $H_{i,j}$ matrix, where $+$ is an adder, *MAX* is a max operator and *SeqComp* is the sequence comparator that generates match/mismatch scores

2.2 Recursive Variable Expansion

Recursive Variable Expansion (RVE) [15] is a kind of loop transformation which removes all data dependencies from a program, so that the program is parallelized to its maximum. The basic idea is that if any statement G_i is dependent on statement H_j for some iteration i and j, then instead we wait for H_j to complete and then execute G_i, we will replace all the occurrences of the variable in G_i that create dependency with H_j with the computation of that variable in H_j. In this way there is no need to wait for the statement H_j to complete and statement G_i can be executed independently of H_j. This step is recursively repeated until the statement G_i is not dependent on any other statement, other than inputs or known values, which essentially means that G_i can be computed without any delays. This transformation is explained clearly in Example 1, which adds the loop counter. Therefore after applying the RVE, we get an expression with five terms to be added, as shown in Example 2. In this way, the whole expanded statement in Example 2 can be computed in any order by computing the large number of operations in parallel and efficiently using binary tree structure as shown in Figure 3. The major drawback of this technique is that the speed up is achieved at the cost of redundancy, which consumes a lot of resources.

Example 1: A simple example which adds the loop counter

```
A[1]  =  1
for  i  =  2 to 5
A[i]  =  A[i-1] + i   ——-       ($G_i$)
end for
```

The RVE approach is discussed in detail in [16], where the authors conclude that the RVE approach is 1.6 times

Example 2: After applying RVE on Example 1

$$\begin{aligned} A[5] &= A[4] + 5 \\ &= A[3] + 4 + 5 \\ &= A[2] + 3 + 4 + 5 \\ &= A[1] + 2 + 3 + 4 + 5 \\ &= 1 + 2 + 3 + 4 + 5 \end{aligned}$$

faster than the traditional acceleration approach, however the conclusion is based on theoretical discussion and is not validated by any implementation results.

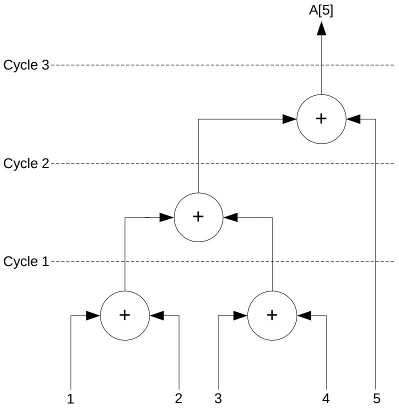

Figure 3: Circuit for the example 2

3 Implementation using Traditional Acceleration Approach

Figure 4 shows a block diagram of a basic cell for computing elements of the $H_{i,j}$ matrix according to a traditional acceleration approach normally referred to as a systolic array approach. In Figure 4, Comp1 is a comparator that compares the two input sequences and outputs the corresponding value of $S_{i,j}$, depending on the values of the match and mismatch scores, such that $S_{i,j}$ = match score, if the corresponding characters in Sequence1 and Sequence2 are equal, otherwise $S_{i,j}$ = mismatch score. Add1 is an adder that adds the diagonal element $H_{i-1,j-1}$ and the value of $S_{i,j}$. Comp2 is a comparator that compares the output of the Add1 with a constant value 0 and outputs the greater of the two numbers. Add2 is an adder that adds the left element $H_{i-1,j}$ and $-d$, where d is the gap penalty. Add3 is an adder that adds the upper element $H_{i,j-1}$ and $-d$. Comp3 compares the outputs of Add2 and Add3 and outputs the greater of the two numbers. Comp4 compares the outputs of Comp2 and Comp3 and outputs the greater of the two numbers. The output of Comp4 is the corresponding $H_{i,j}$ value, which is stored in register $R_{i,j}$. The block diagram shown in Figure 4 is implemented in VHDL and the post place and route simulations show that the time consumed by such a cell is 9.8 ns, where the frequency of the clock used is 50 MHz and the clock period is 20 ns. While implemented on Xilinx XC2VP30 FPGA, one cell consumes 19 out of 13696 slices, where a slice is the basic hardware building element. The cell design shown in Figure 4 can be used to implement a systolic array of any size depending on the availability of hardware resources. Figure 5, shows a 10×10 systolic array, which is implemented using the cell design shown in Figure 4. Figure 6, shows how various neighboring cells are connected in this array. The matrix is initialized with the value zero. The gap penalty is assumed to have a value zero and a simple scoring scheme is assumed, such that $S_{i,j}$ = 2, if there is a match, otherwise $S_{i,j}$ = 0. The remaining values of the $H_{i,j}$ matrix are computed using the systolic array structure, shown in Figure 5. Table 1 shows the filled matrix obtained using this systolic array implementation. The bold digits in Table 1 show the trace back path. Since the elements within each anti diagonal are independent of each other, they are computed in parallel in the array. Therefore the time consumed by an anti diagonal is the same as the time consumed by one cell, which is 9.8 ns. Furthermore since there are 19 anti diagonals in a 10×10 systolic array, the speedup factor (calculating the elements in anti diagonals in parallel) = 100/19 = 5.26. The latency is equivalent to 19 clock cycles = 380 ns. The resources utilized for implementation of a 10×10 systolic array without considering input output overhead are equivalent to 1880 slices. The number of slices utilized by the array, with input output hardware overhead is 2096, thus

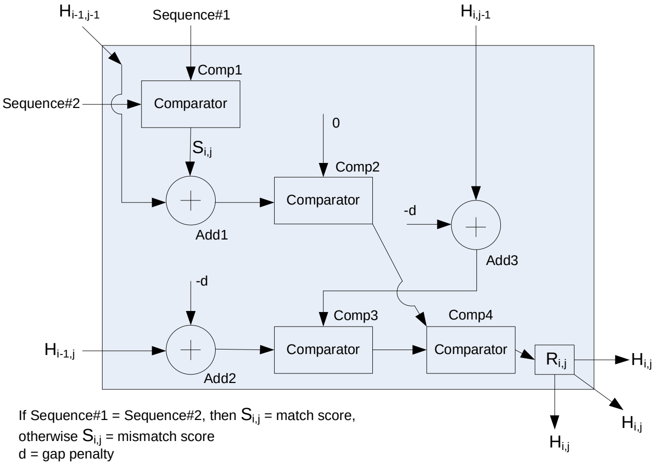

Figure 4: Block diagram description of a basic cell for computing $H_{i,j}$ values of Equation 1

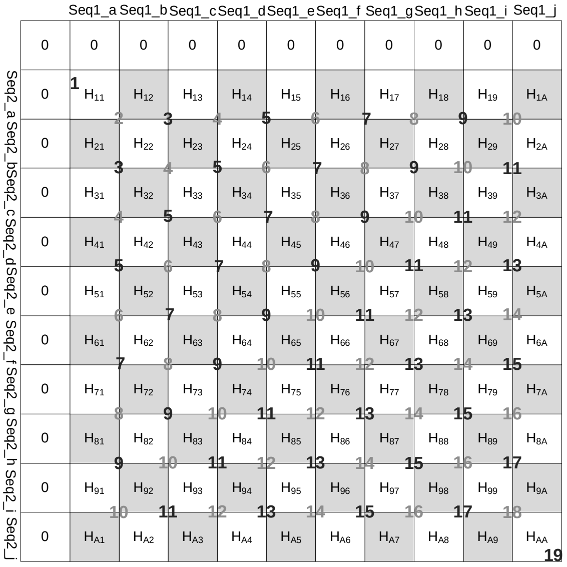
Seq1_a Seq1_b Seq1_c Seq1_d Seq1_e Seq1_f Seq1_g Seq1_h Seq1_i Seq1_j
Seq2_a Seq2_b Seq2_c Seq2_d Seq2_e Seq2_f Seq2_g Seq2_h Seq2_i Seq2_j
0
H_{11} … H_{AA}
1
19

Figure 5: Block diagram description of a 10×10 systolic array

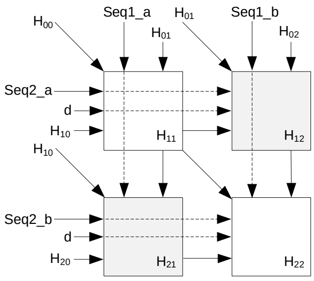

Figure 6: Block diagram description of connectivity between various neighboring cells

Table 1: Filled matrix obtained using the systolic array implementation, as shown in Figure 5.

		A	G	T	A	A	G	T	A	T	A
	0	*0*	*0*	*0*	*0*	*0*	*0*	*0*	*0*	*0*	*0*
G	*0*	**0**	2	2	2	2	2	2	2	2	2
G	*0*	0	**2**	2	2	2	4	4	4	4	4
T	*0*	0	2	**4**	4	4	4	6	6	6	6
C	*0*	0	2	**4**	**4**	4	4	6	6	6	6
A	*0*	2	2	4	6	**6**	6	6	8	8	8
G	*0*	2	4	4	6	6	**8**	8	8	8	8
T	*0*	2	4	6	6	6	8	**10**	10	10	10
A	*0*	2	4	6	8	8	8	10	**12**	12	12
T	*0*	2	4	6	8	8	8	10	12	**14**	14
A	*0*	2	4	6	8	10	10	10	12	14	**16**

a maximum of 653 PEs can be fitted on a Xilinx Virtex-II Pro (XC2VP30) FPGA. We extended the array to 28x20, which consumed 10751 out of 13696 slices, thereby showing that in practice, 713 PEs can be fitted on a virtex-II Pro FPGA. There are 47 anti diagonals in 28×20 array, so the speedup factor = 560/47 = 11.91. The latency is equivalent to 47 clock cycles = 47×20 = 940 ns.

We considered a software equivalent of the basic systolic cell written in C language. We run it on a 100 MHz IBM power PC and measured its runtime, which was 2790 ns. This runtime when compared with the runtime of the basic cell, as shown in Figure 4, gives the relative speedup.

Speedup = 2790 / 9.8 = 284.7.

Speedup (10×10 array) = 284.7 ×5.26 = 1497.52.

Speedup (28×20 array) = 284.7 ×11.91 = 3390.78.

4 Implementation by Applying Recursive Variable Expansion

Figure 7 shows the way to fill a 2x2 $H_{i,j}$ matrix using RVE approach, as per Equations 3, 4, 5 and 6, where S is the match/mismatch score and g is the gap penalty [16]. In each case the cell to be filled is highlighted along with the cells which are required for its computation.

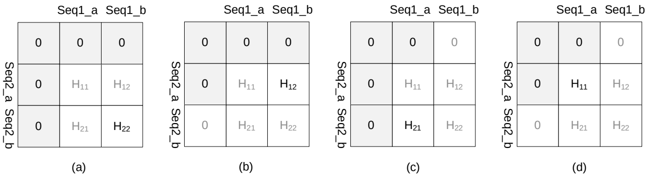

Figure 7: Filling a 2x2 $H_{i,j}$ matrix using Recursive Variable Expansion

$$H_{i-1,j-1} = \max \begin{cases} H_{i-1,j-2} + g \\ H_{i-2,j-2} + S_{i-1,j-1} \\ H_{i-2,j-1} + g \\ 0 \end{cases} \tag{3}$$

$$H_{i-1,j} = \max \begin{cases} H_{i-1,j-2} + 2g \\ H_{i-2,j-2} + g + S_{i-1,j-1} \\ H_{i-2,j-1} + S_{i-1,j} \\ H_{i-2,j} + g \\ 0 \end{cases} \tag{4}$$

$$H_{i,j-1} = \max \begin{cases} H_{i,j-2} + g \\ H_{i-1,j-2} + S_{i,j-1} \\ H_{i-2,j-2} + g + S_{i-1,j-1} \\ H_{i-2,j-1} + 2g \\ 0 \end{cases} \tag{5}$$

$$H_{ij} = \max \begin{cases} (H_{i,j-2}\ \mathit{MAX}\ H_{i-2,j}) + 2g \\ H_{i-1,j-2} + g + (S_{i,j-1}\ \mathit{MAX}\ S_{i,j}) \\ H_{i-2,j-2} + S_{i-1,j-1} + S_{i,j} \\ H_{i-2,j-1} + g + (S_{i-1,j}\ \mathit{MAX}\ S_{i,j}) \\ 0 \end{cases} \tag{6}$$

We define the size of RVE block as the *blocking factor (b)*. So for a 2×2 array, implemented using RVE, the blocking factor b = 2. When implemented in VHDL, this block with b = 2 consumes 13 ns, where the clock period is 30 ns and the frequency is 33.33 MHz. Using this block as a macro design, we implemented a 5×5 array, such that it is comparable to the 10×10 systolic array without using RVE. Figure 8 shows the block diagram representation of this implementation with detailed pin outs of the RVE block with b = 2. Four pins are reserved for the corresponding characters of the input sequences S and T. Five pins are for H inputs, one for gap penalty and two for clock and reset. The four output pins are O_1, O_2, O_3 and O_4. If we relate the output pins with Figure 7, then O_1 is for H_{22}, O_2 is for H_{12}, O_3 is for H_{21} and O_4 is for H_{11}. Figure 9 shows, how a 10×10 array is constructed by using RVE blocks with b = 2. For the blocks in first row and first column of Figure 9, all the inputs come from outside, as shown by external input pins of Figure 8. The four outputs of each block go to

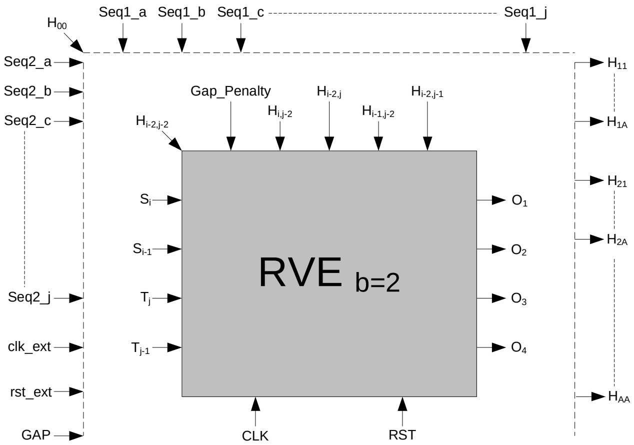

Figure 8: Block diagram representation of a 10x10 array using RVE with b = 2

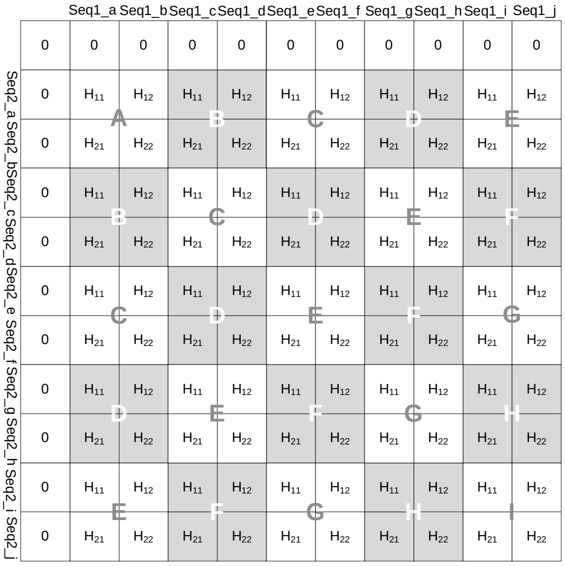
Seq1_a Seq1_b Seq1_c Seq1_d Seq1_e Seq1_f Seq1_g Seq1_h Seq1_i Seq1_j
Seq2_a Seq2_b Seq2_c Seq2_d Seq2_e Seq2_f Seq2_g Seq2_h Seq2_i Seq2_j
0 0 0 0 0 0 0 0 0 0 0
0 H11 H12 H11 H12 H11 H12 H11 H12 H11 H12
A B C D E
0 H21 H22 H21 H22 H21 H22 H21 H22 H21 H22
0 H11 H12 H11 H12 H11 H12 H11 H12 H11 H12
B C D E F
0 H21 H22 H21 H22 H21 H22 H21 H22 H21 H22
0 H11 H12 H11 H12 H11 H12 H11 H12 H11 H12
C D E F G
0 H21 H22 H21 H22 H21 H22 H21 H22 H21 H22
0 H11 H12 H11 H12 H11 H12 H11 H12 H11 H12
D E F G H
0 H21 H22 H21 H22 H21 H22 H21 H22 H21 H22
0 H11 H12 H11 H12 H11 H12 H11 H12 H11 H12
E F G H I
0 H21 H22 H21 H22 H21 H22 H21 H22 H21 H22

Figure 9: 10x10 array using RVE with b = 2

the inputs of corresponding neighboring blocks, where the remaining inputs for those blocks come from outside. The entire design consumes 2409 out of 13696 slices without considering input output hardware overhead. The resources utilized with input output hardware overhead are equivalent to 2630, thus a maximum of 130 PEs (RVE blocks with b=2) can be fitted, while implementing on a Xilinx Virtex-II Pro (XC2VP30) FPGA. Since four $H_{i,j}$ elements are calculated per PE, the maximum number of $H_{i,j}$ elements calculated is $130 \times 4 = 520$. We extended the design to 14x10, which is equivalent to 28×20 systolic array and consumed 13694 out of 13696 slices, thereby showing that in practice, 140 PEs can be fitted on a Xilinx Virtex-II Pro FPGA. There are 9 anti diagonals in a 5×5 array using RVE with b = 2, represented by letters A, B, C, D, E, F, G, H and I in Figure 9. Each anti diagonal is computed in one clock cycle, so the latency is equivalent to 9 clock cycles = 9×30 = 270 ns. In case of 14×10 array, there are 23 anti diagonals, so the latency is equivalent to 23 clock cycles = 23×30 = 690 ns.

In case of 10×10 array, the performance gain in terms of latency, achieved by the RVE implementation, as compared to a traditional systolic array = 380/270 = 1.41. This performance gain is achieved at the cost of utilizing 2630/2096 to 2409/1880 = 1.25 to 1.28 times more resources. In case of 28×20 array, the performance gain = 940/690 = 1.36, at the cost of utilizing 13694/10751 = 1.27 times more resources.

5 Discussion and Results

Systolic array is the best known implementation of the S-W algorithm thus far, as it exploits the maximum parallelism available in the algorithm. This inherent parallelism is limited by the data dependencies in the algorithm. The RVE approach adopted in this paper eliminates this limitation by expanding all the variables to their maximum capacity. The result is an improved performance at the cost of using additional resources. The degree of expansion for the variables depends on the availability of resources on the platform being utilized. So its a trade off between the speedup achieved and the resources utilized.

Table 2 reports the results achieved from our implementations. The first part of the table presents a comparison between software implementation and systolic array implementation, which demonstrates that the basic cell design is 284.7 times faster, the 10×10 systolic array implementation is 1497.52 times faster and the 28×20 systolic array implementation is 3390.78 times faster than their corresponding equivalent software implementations. The first part of Table 2 also shows that the basic cell consumes 19 slices, whereas the 10×10 systolic array consumes 1880 out of 13696 slices without considering the input output hardware overhead and 2096 slices out of 13696 slices with overhead. It also shows that the 28×20 systolic array consumes 10751 slices. The FPGA used for implementations is Xilinx XC2VP30, which has a maximum of 13696 slices. The second part of Table 2 presents a comparison between systolic array and RVE implementations, which demonstrates that the 10×10 array using RVE with b = 2 is 1.41 times faster than the equivalent 10×10 systolic array implementation and 28x20 array using RVE with b = 2 is 1.36 times faster than its equivalent systolic array implementation. The second part of the table also shows that the 10×10 RVE implementation consumes 2409 out of 13696 slices without considering input output hardware overhead and 2630 out of 13696 slices with overhead. Similarly 28×20 RVE implementation consumes 13694 slices. This means that the speedup of 1.41 is achieved at the cost of utilizing 1.25 to 1.28 times more resources and the speedup of 1.36 is achieved at the cost of utilizing 1.27 times more resources. Thus by applying the RVE technique, we improved the performance by a factor of 3390.78 ×1.36 = 4611.5, as compared to its equivalent software implementation. The speedup achieved by applying RVE increases with the increasing blocking factor (b), but resource utilization also increases as a consequence. Thus the limiting factor is the availability of resources on the device utilized for implementation (Xilinx XC2VP30 in our case).

6 Conclusion

In this paper we presented an implementation of the S-W algorithm using traditional acceleration approach and compared its performance with a software equivalent. The comparison shows that the implementation using traditional acceleration approach is 3390.78 times faster than its equivalent software implementation. To explore more parallelism and to eliminate the limitations due to inherent data dependencies, we applied the RVE technique. The implementation using this technique improves the performance by a factor of 1.36 to 1.41, as compared to the implementation using traditional acceleration approach, at the cost of using 1.25 to 1.28 times more resources. Thus the performance achieved by implementation using RVE technique is 4611.5 times higher, as compared to its equivalent software implementation.

References

[1] T. F. Smith and M. S. Waterman, "Identification of common molecular subsequences", *Journal of Molecular Biology*, vol. 147, pp: 195–197, 1981.

Table 2: Comparison between software, systolic array and RVE implementations

Comparison between software and systolic array implementations

Implementation	**Time consumed**	**Clock frequency**	**Speedup w.r.t. software implementation**	**Number of slices**	**Number of slices with overhead**
software	2790 ns	100 MHz	1	—	—
basic cell	9.8 ns	50 MHz	284.7	19 out of 13696	19 out of 13696
10×10 systolic array	380 ns	50 MHz	284.7×5.26 = 1497.52	1880 out of 13696	2096 out of 13696
28×20 systolic array	940 ns	50 MHz	284.7×11.91 = 3390.78	—	10751 out of 13696

Comparison between systolic array and RVE implementations

Implementation	**Time consumed**	**Clock frequency**	**Speedup w.r.t. systolic array implementation**	**Number of slices**	**Cost**	**Number of slices with overhead**	**Cost with overhead**
10×10 systolic array	380 ns	50 MHz	1	1880 out of 13696	1	2096 out of 13696	1
10×10 array using RVE with b = 2	270 ns	33.3 MHz	1.41	2409 out of 13696	1.28	2630 out of 13696	1.25
28×20 systolic array	940 ns	50 MHz	1	—	—	10751 out of 13696	1
28×20 array using RVE with b = 2	690 ns	33.3 MHz	1.36	—	—	13694 out of 13696	1.27

[2] W. R. Pearson and D. J. Lipman, "Rapid and Sensitive Protein Simlarity Searches", *Science*, vol. 227, pp: 1435–1441, 1985.

[3] S. F. Altschul, Gish, W. Miller, W. Myers and D. J. Lipman, "A Basic Local Alignment Search Tool", *Journal of Molecular Biology*, vol. 215, pp: 403–410, 1990.

[4] J. Chiang, M. Studniberg, J. Shaw, S. Seto and K. Truong, "Hardware Accelerator for Genomic Sequence Alignment", *Proceedings of the 28th IEEE EMBS Annual International Conference*, Aug 30–Sept 3, 2006, New York City, USA.

[5] Y. Yamaguchi, Y. Miyajima, T. Maruyama, and A. Konagaya, "High Speed Homology Search Using Run-Time Reconfiguration", *FPL 2002*.

[6] M. Borah, R. S. Bajwa, S. Hannenhalli and M. J. Irwin, "A SIMD Solution to the Sequence Comparison Problem on the MGAP", *Proceedings of the International Conference on Application Specific Array Processors*, 1994.

[7] A. Di Blas et. al., "The UCSC Kestrel Parallel Processor", *IEEE Transactions on Parallel and Distributed Systems*, vol. 16(1), pp: 80–92, 2005.

[8] A. Schroder et. al., "Bio-Sequence Database Scanning on a GPU" *HICOMB*, 2006.

[9] Laiq Hasan and Zaid Al-Ars, "Performance Improvement of the Smith-Waterman Algorithm", *Annual Workshop on Circuits, Systems and Signal Processing (ProRISC 2007)*, November 29–30, 2007, Veldhoven, The Netherlands.

[10] L. Hasan, Z. Al-Ars and S. Vassiliadis, "Hardware Acceleration of Sequence Alignment Algorithms - An Overview", *Proceedings of International Conference on Design & Technology of Integrated Systems in Nanoscale Era (DTIS'07)*, pp: 96–101, September 2–5, 2007, Rabat, Morocco.

[11] R. Giegerich, "A systematic approach to dynamic programming in bioinformatics", *Bioinformatics*, vol. 16, pp: 665–677, 2000.

[12] H. Y. Liao, M. L. Yin and Y. Cheng, "A Parallel Implementation of the Smith-Waterman Algorithm for Massive Sequences Searching", *Proceedings of the 26th Annual International Conference of the IEEE EMBS", September 1–5, 2004, San Francisco, CA, USA.,.*

[13] Steve Margerm, Cray Inc, "Reconfigurable Computing in Real-World Applications", *FPGA and Structured ASIC Journal (www.fpgajournal.com)*, February 7, 2006.

[14] C. W. Yu, K. H. Kwong, K. H. Lee and P. H. W. Leong, "A Smith-Waterman Systolic Cell", *FPL 2003.,.*

[15] Z. Nawaz, O. S. Dragomir, T. Marconi, E. M. Panainte, K. Bertels and S. Vassiliadis, "Recursive Variable Expansion: A Loop Transformation for Reconfigurable Systems", *proceedings of International Conference on Field-Programmable Technology 2007*, Kokurakita, Kitakyushu, JAPAN, December 2007.

[16] Z. Nawaz, M. Shabbir, Z. Al-Ars, K.L.M. Bertels, "Acceleration of Smith-Waterman Using Recursive Variable Expansion", *proceedings of 11th Euromicro Conference on Digital System Design 2008*, Parma, Italy, September 2008.