A Novel Approach for Accelerating the Smith-Waterman Algorithm using Recursive Variable Expansion

Laiq Hasan Zaid Al-Ars Zubair Nawaz
Delft University of Technology
Computer Engineering Laboratory
Mekelweg 4, 2628 CD Delft, The Netherlands
Tel: +31 15 27 86172 Fax: +31 15 27 84898
L.Hasan@ewi.tudelft.nl

Abstract: *In this paper we implement in hardware, a novel approach for accelerating the S-W algorithm using Recursive Variable Expansion (RVE) technique, which enhances inherent parallelism and exposes extra parallelism as compared to any other technique. The results demonstrate that applying recursive variable expansion technique speeds up the performance by a factor of 3.83, as compared to traditional acceleration approaches at the cost of using up to 1.36 times more resources.*

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

I. Introduction

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 adopted to accelerate the S-W algorithm in hardware [4], [5], [6], [7], [8], [9]. An overview of such approaches is given in [10].

In this paper we implement in hardware, a novel approach for accelerating the S-W algorithm using the Recursive Variable Expansion (RVE) technique, and compared the results with the implementation using traditional acceleration approaches. The speedups thus achieved are reported later in the paper.

The remainder of the paper is organized as follows: Section II gives a brief description of the S-W algorithm, discusses related work using traditional acceleration approach and briefly explains recursive variable expansion. Section III discusses the implementation using traditional acceleration approaches. Section IV discusses the implementation using the RVE technique. Section V discusses the results obtained and Section VI gives a brief conclusion.

II. Background and Related Work

The S-W algorithm [1] is a method used for local sequence alignment and is based on *dynamic programming (DP)* [11]. In the following subsections we give a brief description of the algorithm, its inherent data dependencies and work related to its hardware acceleration.

A. 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:

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

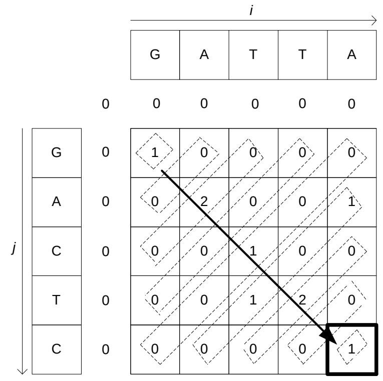
where $S_{i,j}$ is the similarity score of comparing sequence A_i to sequence B_j and d is the penalty for a mismatch. The algorithm consists of the following three steps:

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 $O(M+N)$. 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 are 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 dashed rectangles 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 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 dashed rectangles representing 9 anti diagonals and a maximum of 5 cells may be computed in parallel.



$$\begin{array}{c|c|ccccc}
 & & G & A & T & T & A \\
 & 0 & 0 & 0 & 0 & 0 & 0 \\
G & 0 & 1 & 0 & 0 & 0 & 0 \\
A & 0 & 0 & 2 & 0 & 0 & 1 \\
C & 0 & 0 & 0 & 1 & 0 & 0 \\
T & 0 & 0 & 0 & 1 & 2 & 0 \\
C & 0 & 0 & 0 & 0 & 0 & \boxed{1}
\end{array}$$

Fig. 1. A sample $H_{i,j}$ matrix, showing the parallelization possibilities

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].

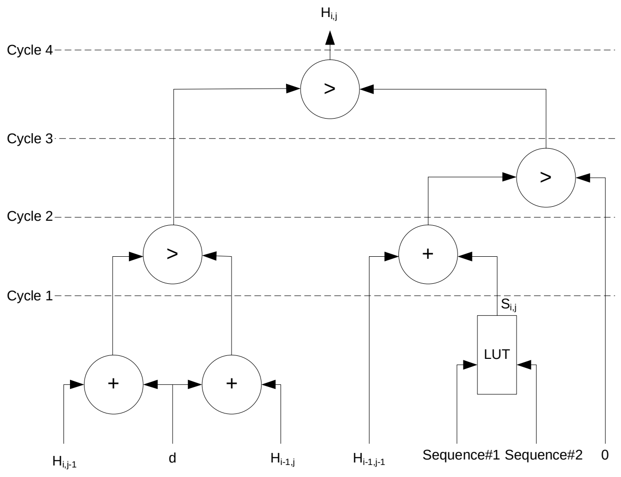


Fig. 2. Circuit to compute an element in the $H_{i,j}$ matrix, where $+$ is an adder, $>$ is a max operator and LUT stands for Look Up Table that generates match/mismatch scores

Figure 2 shows the implementation to compute an element of the $H_{i,j}$ matrix. This unit contains three adders, one Look Up Table (LUT) and three comparators. The time to compute an element is 4 cycles, where each cycle is presumably equal to the latency of one add, compare or LUT operation.

B. *Traditional Acceleration of the S-W Algorithm*

As mentioned in Section II-A, the best known hardware acceleration of the S-W algorithm takes $m+n-1$ steps to complete. Since each step takes 4 cycles, the best known time to compute the S-W algorithm is $4(m+n-1)$ cycles.

A lot of work has been done to accelerate the S-W algorithm using various hardware platforms. In addition to specific architectures designed for sequence alignment, many solutions for special purpose hardware, SIMD and FPGAs have been devised [10].

In [4], the authors studied the improvement of computational processing time of the S-W algorithm using *custom instructions (CIs)* on an FPGA board. This was done by first writing the S-W algorithm in pure software and then replacing the portion which was the most computationally intensive with an FPGA custom instruction. Particulary, they designed CIs on an Altera Nios II integrated development environment. The Nios II soft microprocessor was instantiated on an FPGA to allow rapid prototyping of new designs. Finally, they compared the processing runtime between the pure software and the hardware acceleration versions to calculate the percentage of runtime improvement. The results showed that the hardware accelerated algorithm improved the processing runtime by an average of 287%. Thus using FPGA CIs is a promising direction for further research in improving genomic sequence searching.

In [5], an approach to realize high speed sequence alignment using run-time reconfiguration is proposed. With this approach, it is demonstrated that high performance can be achieved using off-the-shelf FPGA boards. The performance is almost comparable with dedicated hardware systems. The time for comparing a query sequence of 2048 elements with a database sequence of 64 million elements by the S-W algorithm is about 34 sec, which is about 330 times faster than a desktop computer with a Pentium-III, 1.0 GHz processor.

In [6], an implementation of the S-W algorithm is described on a general purpose fine-grained SIMD architecture, the *Micro Grained Array Processor (MGAP)*. The authors of [6] show that their implementation is about 5 times faster than the rapid implementation of a genetic sequence comparator using FPGAs, called SPLASH [15]. Showing thereby, that massively parallel processor arrays, like the MGAP, possess the capability to solve computationally intensive problems in Computational Molecular Biology efficiently and inexpensively.

The Kestrel parallel processor is a single-board coprocessor with a 512-element linear array of 8-bit, SIMD processing elements [7]. As a case study, the authors of [7] implemented the S-W algorithm on the Kestrel parallel processor for different query sizes. The results show that their implementation is 17 times faster than an implementation on Ultra SPARC-II, 500 MHz processor, for a query size of 100.

In [8], it has been demonstrated that the streaming architecture of *Graphics Processing Units (GPUs)* can be efficiently used for biological sequence database scanning. To derive an efficient mapping onto this type of architecture, the authors have reformulated the S-W algorithm in terms of computer graphics primitives and claim that the evaluation of their implementation on a high-end graphics card shows a speedup of almost sixteen compared to a Pentium IV, 3.0 GHz processor. They also claim that this is the first reported implementation of the S-W algorithm on graphics hardware.

In [9], a software-only implementation of the S-W algorithm is profiled on Pentium-IV, 3.2 GHz processor, using the GNU profiler. The profiling results identify that a specific small part of the algorithm consumes a disproportionately large amount of computational time, amounting to 72.33 % of the total runtime. This part is then designed in VHDL. The processing run time of the software-only implementation on Pentium-IV, 3.2 GHz processor and hardware implementation on a Virtex II Pro FPGA are compared to evaluate the % runtime improvement. The results show that the hardware implementation is 35.82 times faster than its equivalent software-only implementation.

C. Recursive Variable Expansion

Recursive Variable Expansion (RVE) [16] is a kind of loop transformation which removes all the data dependencies from a program, so that the program gets prone to maximum parallelism. 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. 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. Similarly, if H_j is dependent on some other statement, we will imbed the computation of that statement into H_j to make it independent of that statement. 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 waiting for the computation of any other statement. This transformation can be explained clearly by 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
```

Example 2: After applying RVE on Example 1

```
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
```

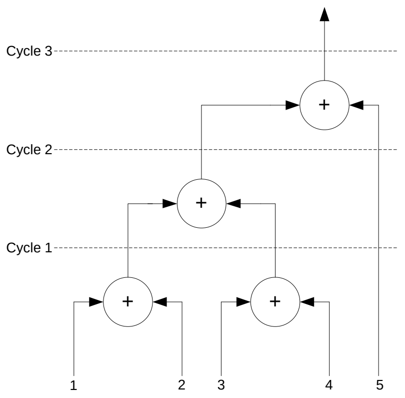


Fig. 3. Circuit for Example 2

III. 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 of 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 results 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. The synthesis report shows that while implemented on Xilinx XC2VP30 FPGA, one cell consumes 19 out of 13696 slices.

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. As a case study, we implemented a 2×2 systolic array, as shown in Figure 5.

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

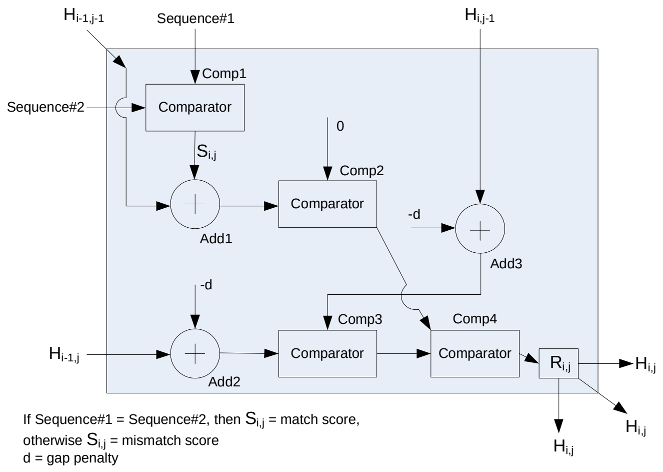


Fig. 4. Block diagram description of a basic cell for computing $H_{i,j}$ values of Equation 1

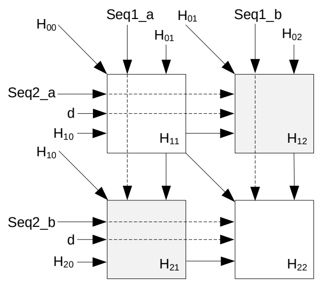


Fig. 5. Block diagram description of a 2×2 systolic array

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 I shows the filled matrix obtained using this systolic array implementation. The bold digits in Table I 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 3 anti diagonals in a 2×2 systolic array, the speedup factor (calculating the elements in anti diagonals in parallel) = 4/3 = 1.33. The latency for the entire computation, as obtained from the post place and route simulation is equivalent to 49.8 ns. The synthesis report shows that the resources utilized for implementation

TABLE I
Filled matrix obtained using the systolic array implementation, as shown in Figure 5.

		A	G
	0	*0*	*0*
G	*0*	**0**	2
G	*0*	0	**2**

of a 2×2 systolic array are equivalent to 70 slices.

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 measure its runtime, which was 2790 ns. This runtime when compared with the runtime of the basic systolic cell in hardware gives the relative speedup.

Speedup = 2790 / 9.8 = 284.7.

Speedup for 2×2 systolic array = 284.7 ×1.33 = 378.65.

IV. Implementation by Applying Recursive Variable Expansion

Figure 6 shows the way to fill a 2x2 $H_{i,j}$ matrix using RVE, as per Equations 3, 4, 5 and 6. In each case the cell to be filled is highlighted along with the cells which are required for its computation.

$$H_{11} = \max \begin{cases} H_{00} \\ H_{01} \\ H_{10} \end{cases} \tag{3}$$

$$H_{12} = \max \begin{cases} H_{00} \\ H_{01} \\ H_{02} \\ H_{10} \end{cases} \tag{4}$$

$$H_{21} = \max \begin{cases} H_{00} \\ H_{01} \\ H_{10} \\ H_{20} \end{cases} \tag{5}$$

$$H_{22} = \max \begin{cases} H_{00} \\ H_{01} \\ H_{02} \\ H_{10} \\ H_{20} \end{cases} \tag{6}$$

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, an array of any larger size may be implemented, depending on the availability of hardware resources. Figure 7 shows the block diagram representation of this RVE design with b = 2. The synthesis report shows that the design consumes 95 out of 13696 slices.

Performance gain in terms of latency, as compared to a 2×2 traditional systolic array = 49.8/13 = 3.83. This performance gain is achieved at the cost of utilizing 95/70 = 1.36 times more resources.

V. Discussion and Results

Systolic array is the best known implementation of the S-W algorithm known so far, as it exploits the maximum parallelism available in the algorithm. This inherent parallelism is limited by the data dependencies in the algorithm.

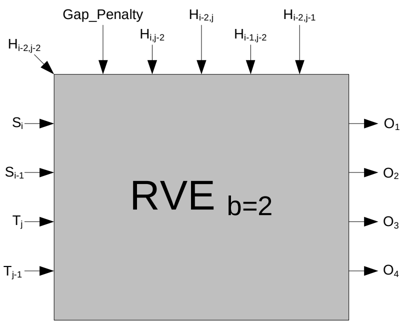


Fig. 7. Block diagram representation of an RVE design with b = 2

The RVE technique applied in this paper eliminates this limitation by expanding all the variable 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 device being utilized. So its a trade off between the speedup achieved and the resources utilized.

The results achieved from our implementations, as reported in Table II, demonstrate that the systolic array implementation is 378.65 times faster than its equivalent software implementation. Moreover applying recursive variable expansion technique speeds up the performance by a factor of 3.83, as compared to the systolic array implementation at the cost of utilizing 1.36 times more resources. The speedup achieved by applying RVE technique 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 being utilized for implementation (Xilinx XC2VP30 in our case).

VI. Conclusion

In this paper, we implemented the S-W algorithm by applying the RVE technique. This helps to explore more parallelism and to eliminate the limitation inherited by the data dependencies. The result is an improved performance at the cost of higher resource utilization as compared to traditional acceleration approaches. The results demonstrate that the implementation using this technique improves the performance by a factor of 3.83, as compared to the implementation using traditional acceleration approaches, at the cost of using 1.36 times more resources. The performance may further be improved by utilizing even more resources. So the performance gain is actually a trade off between the speedup desired and the resources available.

References

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

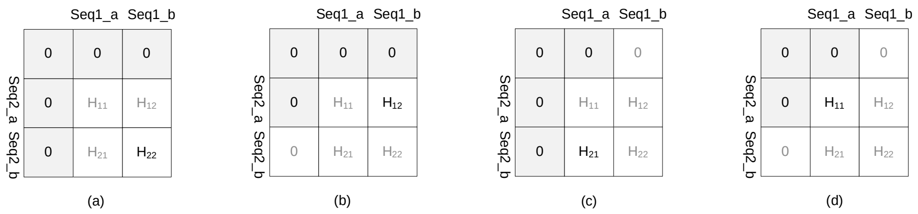


Fig. 6. Filling a 2x2 $H_{i,j}$ matrix, using Recursive Variable Expansion technique

TABLE II
COMPARISON BETWEEN SOFTWARE, SYSTOLIC ARRAY AND RVE IMPLEMENTATIONS

Comparison between software and systolic array				
Implementation	**Time consumed**	**Speedup**	**Number of slices**	**Cost**
software	2790 ns	—	—	—
basic systolic cell	9.8 ns	284.7	19 out of 13696	—
2×2 systolic array	49.8 ns	284.7×1.33 = 378.65	70 out of 13696	—

Comparison between systolic array and RVE				
Implementation	**Time consumed**	**Speedup**	**Number of slices**	**Cost**
2×2 systolic array	49.8 ns	—	70 out of 13696	—
RVE design with b = 2	13 ns	3.83	95 out of 13696	1.36

[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] Daniel P. Lopresti, "Rapid Implementation of a Genetic Sequence Comparator Using Field Programmable Logic Arrays", *Conference on Advanced Research in VLSI*, pp: 138–152, 1991.

[16] 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.