Fast Smith-Waterman hardware implementation

Zubair Nawaz, Koen Bertels
Computer Engineering Lab
Delft University of Technology,
The Netherlands
{*z.nawaz, k.l.m.bertels*}*@tudelft.nl*

H. Ekin Sümbül
Electronic Engineering Program
Sabancı University
Turkey
ekinsumbul@su.sabanciuniv.edu

***Abstract*—The Smith-Waterman (SW) algorithm is one of the widely used algorithms for sequence alignment in computational biology. With the growing size of the sequence database, there is always a need for even faster implementation of SW. In this paper, we have implemented two Recursive Variable Expansion (RVE) based techniques, which are proved to give better speedup than any best dataflow approach at the cost of extra area. Compared to dataflow approach, our HW implementation is 2.29 times faster at the expense of 2.82 times more area.**

***Keywords*-computational biology; sequence alignment; hardware acceleration; RVE**

I. Introduction

THE Smith-Waterman (SW) algorithm is one of the widely used algorithms for sequence alignment in computational biology. With the growing size of the sequence database, there is always a need for even faster implementation of SW.

The main contributions of this paper[1] is the actual implementation of RVENP and RVEP for different versions of SW and blocking factor for RVE as described in our earlier work [1], [2]. Furthermore, we have implemented conventional linear systolic array dataflow implementation of a single element for SW for comparison. The results shows that RVEP give 129% more speedup at the cost of 182% more area than the dataflow approach. FPGAs are a good choice to accelerate the sequence alignment as they provide a lot of parallelism. A linear systolic implementation of the dataflow approach is widely used for it. Some of the solutions based on FPGAs using the dataflow are given in [3], [4], [5]. Recently Jiang [6] modified the SW formula by introducing a new variable and thereby reducing the critical path to compute a single cell.

Implementations similar to our are presented in [7], [8]. Both of them are mere implementation of RVENP technique [1] and model only SW with linear gap penalty. In [7], the hardware implementation was done based on the rectangular systolic array implementation, whereas in [8], the linear systolic array implementation is done. [8] is more close to our current implementation. We have not only implemented RVENP but also RVEP using the linear systolic array for SW with linear and affine gap penalities. [8] also takes lot more time (19900 ns) than our equivalent RVENP implementation (690 ns) for the table of size 50×50.

[1]This work is sponsored by the hArtes (IST-035143), the MORPHEUS (IST-027342) and RCOSY (DES-6392) projects.

		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 SW algorithm, when $a = -2$ and $x(i,j) = +2$ when *S[i]=T[j]* otherwise -1. Elements in the trace back are shown in bold.

The rest of the paper is organized as follows. In section II, we briefly describe the Smith Waterman Algorithm and how it is implemented on a uniprocessor. Section III describes the way the RVE technique is applied to SW and also shows the mapping to actual circuits. Section IV discusses the experimental setup and obtained results showing the time and hardware results for different versions SW with different blocking factor of RVE. Finally, we conclude the paper in Section V.

II. 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.

$$F[i,j] = \max \begin{cases} D[i,j] \\ F[i-1,j-1] + x[i,j] \\ E[i,j] \\ 0 \end{cases} \quad (1)$$

and

$$D[i,j] = \max \begin{cases} F[i,j-1] + a \\ D[i,j-1] + b \end{cases}$$

$$E[i,j] = \max \begin{cases} F[i-1,j] + a \\ E[i-1,j] + b \end{cases}$$

where $F[0,0] = D[0,0] = E[0,0] = F[0,j] = D[0,j] = E[0,j] = F[i,0] = D[i,0] = E[i,0] = 0$, for $1 \leq i \leq m$ and $1 \leq j \leq n$. The $x[i,j]$ is the similarity score obtained from a scoring/similarity matrix for the corresponding characters $S[i]$ and $T[j]$. Equation 1 is called *SW for affine gap penalties* [9].

Sometimes, a simplified version of SW is also used, in which $a = b$. This is called *SW for linear gap penalties* and Equation 1 can be simplified as following:

$$F[i,j] = \max \begin{cases} F[i,j-1] + a \\ F[i-1,j-1] + x[i,j] \\ F[i-1,j] + a \\ 0 \end{cases} \tag{2}$$

To align the two sequences *S[1..m]* and *T[1..n]*, the overall optimal score $F[n,m]$ needs to be computed. An example of the SW algorithm is shown in Figure 1, where a matrix is made and the two sequences are put along the row and column. First, the top row and the left column are filled with boundary condition. Then the rest of the matrix is filled using Equation 1 or 2, depending upon the choice made by the user, from the top-left corner. The elements are filled from left to right and from top to bottom. Once the whole matrix is filled, we find the maximum score in the whole matrix and start a trace back from that element to one of the three elements from which alignment score is calculated. This process is repeated till the score drops below a certain threshold or to zero.

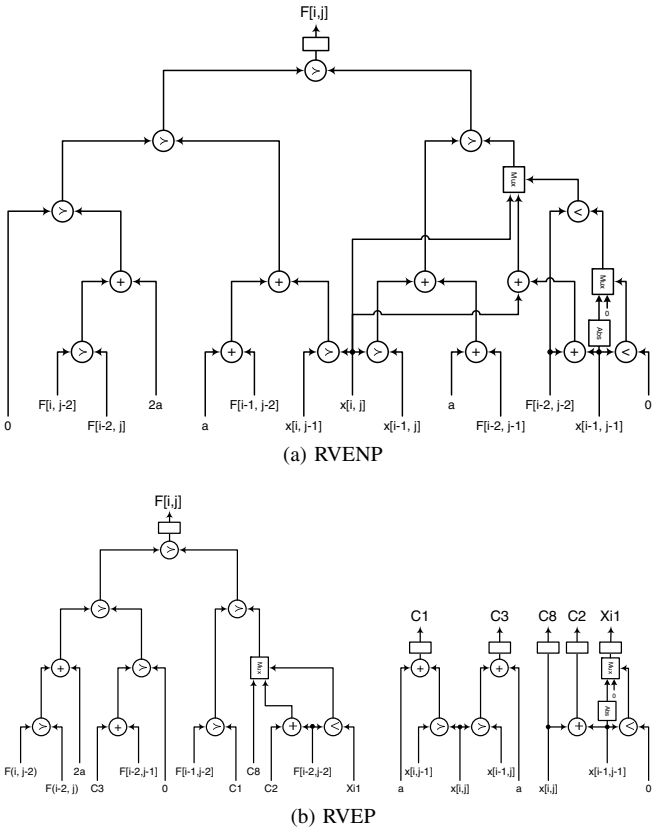
The profiling of the SW shows that filling the matrix takes 98.6% of the overall time to find the optimal alignment [10]. Therefore filling the matrix is obvious choice to be accelerated on FPGA.

III. Application of RVE to SW

Instead of computing an element, we can compute a block of $k \times k$ elements in parallel, by partially applying the RVE. When it is applied to Equation 1 and Equation 2, we get the following equations for $F[i,j]$ in 2×2 block [2].

$$F[i,j] = \max \begin{cases} i & D[i,j-2] + 2b \\ ii & F[i,j-2] + c_2 \\ iii & F[i-2,j] + c_2 \\ iv & E[i-2,j] + 2b \\ v & (D[i-1,j-2] \succ E[i-2,j-1]) + c_1 \\ vi & F[i-1,j-2] + c_3 \\ vii & F[i-2,j-1] + c_4 \\ viii & F[i-2,j-2] + c_5 \\ ix & 0 \end{cases} \tag{3}$$

where $c_1 = b + x[i,j]$, $c_2 = a + b$, $c_3 = a + (x[i,j] \succ x[i,j-1])$, $c_4 = a + (x[i,j] \succ x[i-1,j])$ and $c_5 = x[i,j] + x[i-1,j-1]$ in Equation 3. Here $\succ$ is defined as the max operator.



(a) RVENP

(b) RVEP

Figure 2. $F[i,j]$ computation in a block for SW with linear gap penalties

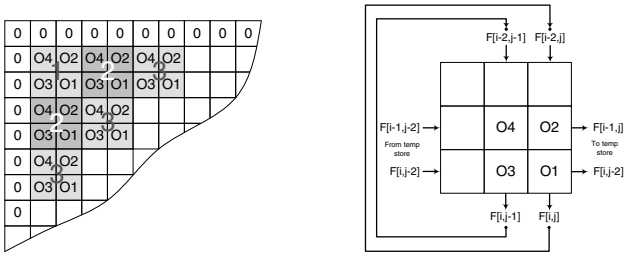
$$F[i,j] = \max \begin{cases} i & (F[i,j-2] \succ F[i-2,j]) + 2a \\ ii & F[i-1,j-2] + C_1 \\ iii & F[i-2,j-2] + C_2 \\ iv & F[i-2,j-1] + C_3 \\ v & 0 \end{cases} \tag{4}$$

where $C_1 = a + (x[i,j-1] \succ x[i,j])$, $C_2 = x[i,j] + x[i-1,j-1]$ and $C_3 = a + (x[i,j] \succ x[i-1,j])$ in Equation 4.

Likewise Equation 3 and 4 to compute $F[i,j]$, the equations to compute $F[i-1,j]$, $F[i,j-1]$ and $F[i-1,j-1]$ (O1, O2, O3 and O4 respectively in Figure 3a) in parallel can be produced for affine and linear gap penalties. These unknown also define the blocking factor B, here it is $B = 2$.

A. Mapping Equations to Circuits

There are two ways Equation 3 and 4 can be mapped to circuits [2]. First, called as RVENP (RVE with no pre-computation), is a simple implementation of the RVE equation for each iteration (i,j). In second implementation we divide the RVE equation for an iteration (i,j) into two parts. One that can be computed prior to the current



(a) Sequence of fill (b) Systolic array

Figure 3. Filling the whole matrix

iteration (i, j) (pre-computed) as it is known earlier and second, which becomes known at the current iteration (i, j), this implementation is called as RVEP (RVE with pre-computation). In Equation 3, $c_1, c_2, \ldots, c_5$ for any iteration can be computed in advance as all its contents are known in the start of computation. Similarly in Equation 4, C_1, C_2 and C_3 can be pre-computed. RVENP implementation for the SW with linear gap penalty is shown in Figure 2a and RVEP implementation is shown in Figure 2b.

The pre-computation increases the parallelism by dividing the same equation into two parts and computing both of them in parallel. This however slightly increases the area with the introduction of extra registers to save the pre-computed values. This way, pre-computation may also reduce the critical path depending upon the problem. First implementation, RVENP for SW with linear gap penalties has the critical path from $F[i, j-2]$ to $F[i, j]$ as shown in Figure 2a, which has 5 levels. However, in RVEP implementation, the critical path is reduced to 4 levels from $F[i, j-2]$ to $F[i, j]$ as shown in Figure 2b.

Once a block is computed using circuit in Figure 2a or 2b and other circuits to compute $F[i-1, j]$, $F[i, j-1]$ and $F[i-1, j-1]$, the whole matrix can be computed by making the circuits for all the blocks with the same number as shown in Figure 3a, which is like a dataflow at block level. This structure of filling motivates us to use a systolic array as shown in Figure 3b. This shows that when a block is computed, the output taken out in the vertical direction are fed back as input for the same block circuit to compute the next iteration in the fill. The horizontal data is stored in some temporary storage like BRAM, which is used in some later iteration.

The query sequence and the corresponding row of the scoring matrix is loaded as the preprocessing step. This loading is done once for all the sequence alignment with all the known sequences in the database. However when a new sequence is to be aligned with all the sequences in the database, we can load the new unknown sequence and its corresponding row of scoring matrix by utilizing the partial reconfiguration of the reconfigurable fabric.

We have implemented the protein sequence alignment using Blossum 62 as scoring matrix, in which the highest value is 11 and the lowest is -4. Therefore 5 bits can used to store an element in the LUT. The data width for each block computation is 16 bit, which is sufficient to align a sequence of more than 5k length. We have chosen the protein sequence alignment as it has higher values for $x[i, j]$ and therefore the same design can be used for the DNA sequence alignment by changing the corresponding LUTs.

The number of block PEs p available for computing the sequence alignment depends upon the size of the FPGA used. This also limits the size l of the query sequence used, as the length of the query sequence that can be accommodated on the FPGA is equal to $q = B \times p$. In this paper, we are producing the results when $l \leq q$. However, if the size of the query sequence l is larger than that can be fit on available number of PEs (i.e. $l > q$) on the FPGA, then two solutions can be used. First simpler and easier solution is to split the length l into k parts, such that $((k-1) \times q) < l \leq (k \times q)$, and then execute k passes sequentially by storing in the intermediate results in some temporary storage. The second method which is more suitable for this kind of implementation, but expensive to implement is to use high end machines like CrayXD1, which can have 150 FPGA mounted on a single machine [10]. All these FPGA can be used to map the circuits and run in parallel when required.

There is no issue to accommodate the length d of the database sequence in the design. It can be very large as it only depends upon the size of the temporary storage available as in our case is BRAM to store the intermediate results, which is large enough to do it.

IV. Performance Evaluation

We have compared two variants of the RVE implementation with single element systolic array implementation for three different cases by changing the blocking factor B and two variants of SW. Results show that the RVE implementation is faster than the single element systolic array implementation. Furthermore, results shows that RVEP implementation is faster than RVENP. We have described the PE design in VHDL and implemented on Xilinx Virtex II pro platform XC2VP30 FPGA, which contains 13696 slices. The code was simulated and synthesized on ModelSIM SE 6.5 and Xilinx ISE 10.1 respectively.

First, we have implemented the linear systolic dataflow implementation of Equation 1 and 2, which is equivalent to other comparable implementation on FPGA [3], [5], [4], [6]. We call this implementation as *Element* in Table I.

In the first case, we have chosen SW with linear gap penalties and have applied both RVE techniques with $B = 2$ and compared with Element. To compute a block of 40×40 elements, RVEP is 65% faster than the Element implementation. Similarly RVENP is 46% faster than the Element implementation. This speedup is at the cost of around 100%

Table I
RESULTS TO SHOW TIME AND HARDWARE

Type	Implementation	Frequency (MHz.)	Time to compute $n \times m$ elements (ns)	Speedup	Slices
SW (B=2)	RVEP	79.33	491.60	1.65	9150
Linear	RVENP	70.31	554.74	1.46	8683
$n = 40, m = 40$	Element	97.59	809.51	1	4468
SW (B=3)	RVEP	73.215	286.818	2.29	12335
Linear	RVENP	56.148	374.01	1.76	11790
$n = 36, m = 36$	Element	98.797	657.93	1	4380
SW (B=2)	RVEP	60.35	414.25	1.68	12497
Affine	RVENP	59.32	421.48	1.65	12872
$n = 26, m = 26$	Element	72.959	699.01	1	4380

area on top of Element implementation. In the second case, we compute a block of 36×36 elements and kept the SW same, however blocking factor for RVE is increased to 3. By increasing the blocking factor, the speedup is increased to 129% in case of RVEP and 76% in case of RVENP as compared to Element implementation for the same block computations. This speed up is again at the cost of around 182% more area than the Element implementation. This increase of speedup by increasing the blocking factor shows that the speedup can be improved by increasing the blocking factor which however also increases the area consumed. The higher speedup of RVEP as compared to RVENP is due to the reduction of the critical path, which is done by splitting the computations into two parts. This requires bit more intermediate registers to store the pre-computation results. Therefore it consumes little more area than RVENP.

Finally, we implemented SW with the affine gap penalty and chose $B = 2$ for RVE implementations. RVEP shows a speedup of 1.68 more than Element implementation, which is also more than RVENP's 1.65 speedup. The interesting thing was that its area consumption was less than RVENP, despite the fact it used extra registers to store the intermediate pre-computation results. Both of these RVE implementations were optimized for speed. The critical path of both RVE techniques took the same number of levels, that's why the difference in speedup is not big. However, in the case of RVEP, we were able to increase the reuse of sub-expression in Equation 4 as compared to RVENP, while keeping the speed as the priority. Therefore in case of RVEP, we were able to process the same block in less area than RVENP. This also reduced the net delay for RVEP as compared to RVENP and created a small difference in time. These results are summarized in Table I.

V. CONCLUSION

In this paper, we have shown that when high performance is the main objective and area utilization is not the major restriction then RVE based acceleration can be a good choice. We have implemented two RVE based techniques, RVENP and RVEP, for two versions of SW formulation and with varying blocking factor which produce more speedup for SW than the widely used systolic array dataflow approach. We implemented RVENP and RVEP on Xilinx Virtex II pro platform showed 129% more speedup at the cost of 182% more area than dataflow approach. We also showed that RVEP gives better results for performance as compared to RVENP. As a future work, we will implement it on real benchmarks and implement it on a board with more than one FPGAs.

REFERENCES

[1] Z. Nawaz, M. Shabbir, Z. Al-Ars, and K. Bertels, "Acceleration of smith-waterman using recursive variable expansion," in *DSD-2008*, pp. 915–922, September 2008.

[2] Z. Nawaz, T. P. Stefanov, and K. Bertels, "Efficient hardware generation for dynamic programming problems," in *ICFPT'09*, December 2009.

[3] Y. Yamaguchi, Y. Miyajima, T. Maruyama, and A. Konagaya, "High speed homology search using run-time reconfiguration," in *FPL '02*, pp. 281–291, 2002.

[4] T. Oliver, B. Schmidt, and D. Maskell, "Reconfigurable architectures for bio-sequence database scanning on fpgas," *IEEE Transactions on Circuits and Systems II*, vol. 52, pp. 851–855, Dec. 2005.

[5] C. W. Yu, K. H. Kwong, K. H. Lee, and P. H. W. Leong, "A smith-waterman systolic cell," in *FPL'03*, 2003.

[6] 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, no. 12, pp. 1077–1081, Dec. 2007.

[7] L. Hasan, Z. Al-Ars, Z. Nawaz, and K. Bertels, "Hardware implementation of the smith-waterman algorithm using recursive variable expansion," in *IDT'08*, December 2008.

[8] L. Hasan and Z. Al-Ars, "An efficient and high performance linear recursive variable expansion implementation of the smith-waterman algorithm," in *IEEE EMBS'09*, 2009.

[9] O. Gotoh, "An improved algorithm for matching biological sequences.," *J Mol Biol*, vol. 162, pp. 705–708, December 1982.

[10] O. Storaasli and D. Strenski, "Experiences on 64 and 150 fpga systems," in *RSSI'08*, 2008.