Acceleration of Smith-Waterman Using Recursive Variable Expansion

Zubair Nawaz, Zaid Al-Ars, Koen Bertels
Computer Engineering Lab
Delft University of Technology
The Netherlands
{z.nawaz, z.al-ars, k.l.m.bertels}@tudelft.nl

Mudassir Shabbir
Mentor Graphics, Pakistan
mudassir_shabbir@mentor.com

Abstract

The Smith-Waterman (SW) algorithm is a local sequence alignment algorithm that attempts to align two biological sequences of varying length such that the alignment score is maximum. In this paper, we propose a new approach to reduce the time needed to perform the SW algorithm. This is done by applying the concept of recursive variable expansion, which exposes more parallelism in the algorithm than any other published method. The paper estimates the speed and hardware overhead for the newly proposed approach relative to other known acceleration methods. Using the new approach, it is possible to achieve a minimum speedup of 400 times better than the serial case for a typical sequence length of 500, which is 1.6 times higher than any other published method. The paper also shows that further speedup can be achieved using extra hardware to expose even more parallelism in the algorithm.

1 Introduction

Local sequence alignment is an important problem in computational biology, as it is helps in discovering functional, structural and evolutionary information in biological sequences of DNA, RNA and proteins. It is used to optimally align two apparently dissimilar sequences which include some pattern which is highly conserved. The local alignment algorithm will find this pattern and ignore the patterns that show little similarity. Smith-Waterman (SW) algorithm [16] is a such well known local alignment algorithm. This algorithm is based on dynamic programming, which has time and space complexity *O(mn)*, where m and n are lengths of the sequences being aligned. Although this complexity seems to be acceptable, the exponential growth in bio-sequence databases of known sequences makes this complexity intolerable [8, 2]. Therefore as the database size grows larger, faster algorithms become important to quickly compare and align the sequences.

One way to avoid such expensive solutions is to use heuristic techniques like FASTA [12] and BLAST [1]. Both compute the local alignment and are fast but less sensitive than SW, as the time complexity is reduced at the cost of accuracy. Therefore, an optimal alignment may not always be found through these techniques. Another way to reduce the time complexity is to accelerate SW algorithm through parallel processing. Researchers have been able to parallelize the SW algorithm on parallel machines [15, 18]. However, on the one hand, the amount of acceleration achieved by this method is theoretically bound. On the other hand, keeping in mind the growing size of the database, prevalent methods require further acceleration to match this growth.

In this paper, we show the way to apply *Recursive Variable Expansion* (RVE) to the SW algorithm, which reveals new previously unexplored type of data parallelism in the algorithm. Using RVE, the amount of speedup achievable for a typical sequence length of 500 is at least 400 times better than the serial case (one element is computed in hardware at a time), which is higher than the maximum speedup gained by traditional hardware acceleration methods by a factor of at least 1.6 using low hardware overhead.

The rest of the paper is organized as follows. In section 2, we discuss the background and related work for the parallelization of biological sequence alignment. Our approach and implementation is discussed in Section 3. Section 4 estimates the needed time and hardware for our approach along with a comparison of time and hardware needed for other prevalent parallel techniques, finally, we conclude the paper in Section 5.

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

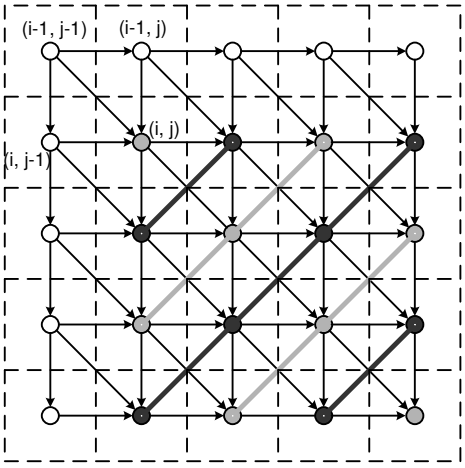


Figure 2. Data dependence graph for Equation 1 (different shades of gray in circles show the elements which can be executed in parallel).

2 Background and Related Work

2.1 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.

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

where $F(0,0) = F(0,j) = F(i,0) = 0$, for $1 \leq i \leq m$ and $1 \leq j \leq n$. The $x(i,j)$ is the score for match/mismatch, depending upon whether $S[i] = T[j]$ or $S[i] \neq T[j]$. The g is some constant penalty for inserting a gap in any sequence. For local alignment, the lowest score for match/mismatch is greater than the recommended gap penalty, otherwise the alignment will have more gaps and will eventually change from local to global type of alignment, even though a local alignment algorithm is used [6, 10, 17]. We will use this observation later in our proposed approach.

To compute optimal alignment score $F(i,j)$ as given by Equation 1 for $1 \leq i \leq m$ and $1 \leq j \leq n$, dynamic programming is applied. In dynamic programming, a bottom-up approach is used, in which initially the boundary conditions are computed and then F is computed from smaller sub-sequences to larger ones till it reaches the entire length of the sequences. 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. The matrix is filled using Equation 1 from the top-left corner and 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 then 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. In the trace back, if the corresponding row and column element match then the alignment is definitely 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 computation of the optimal alignment score *F(i, j)* as given by Equation 1 takes constant time, and since there are $m \times n$ elements to be computed, the time complexity for SW algorithm is *O(mn)*. The trace back takes $O(m+n)$ steps, as the longest path in $m \times n$ matrix is from top left to bottom right, which is $O(m+n)$, and the time to determine the source of computation for an element is constant. We need to keep the table of size $m \times n$ to compute the *F(i, j)* and for trace back, therefore the space complexity for the algorithms is also *O(mn)*.

To parallelize the SW algorithm we need to look at

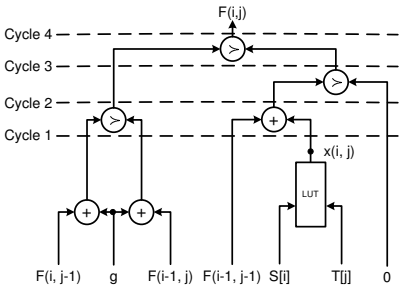


Figure 3. Circuit for the computation of an element $F(i,j)$ in Equation 1, where $\succ$ is the Max operator, and LUT stands for the Look-up table that generates the match/mismatch scores.

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 data dependencies. Therefore we need to change the way the elements are traversed like starting from the top, elements with one shade of gray in anti-diagonal can be executed in parallel followed by the next anti-diagonal with different shade of gray due to dependency constraint. 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) is as follows.

$$l_d = \min(m, n) \tag{2}$$

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 is equal to the number of anti-diagonals required to reach the bottom-right element is as follows.

$$m + n - 1 \tag{3}$$

So far this is the best technique for parallelization and has been used by many researchers [14, 11, 18]. Yamaguchi [18] implemented the SW on FPGA and achieved a speedup of 327 times faster than a desktop computer with Pentium III, 1 GHz for a sequence length of 2048. Oliver [14] achieved a speedup of 170 as compared to software implementation on Pentium IV, 1.6 GHz, for a sequence length of 756. Similarly recently Jiang [11] has improved the speedup to at least 330 times faster than software implementation on 2.8 GHz, Xeon processor for sequence length of 4000.

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 RVE is applied 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
```

Figure 3 shows the implementation to compute one element. This unit contains three adders, one *look up table* (LUT) and three comparators. The time to compute one element is 4 cycles. We have assumed that the time for each cycle is equal to the latency of one adder, comparator or LUT operations. The same assumption holds to compute the latency in the rest of the paper.

2.2 Recursive Variable Expansion

Recursive Variable Expansion (RVE) [13] is a kind of loop transformation which removes all the data dependencies from the program, thereby making it prone to more 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 of waiting 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 replace the computation of that statement with the variable to make it independent of that statement. This step is recursively repeated until the statement G_i is not dependent on any other statement rather only inputs or known values, which essentially means that G_i can be computed without waiting for the computation of any other statement. The technique is very beneficial when most of the operations are associative. 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 a large number of operations in parallel and efficiently

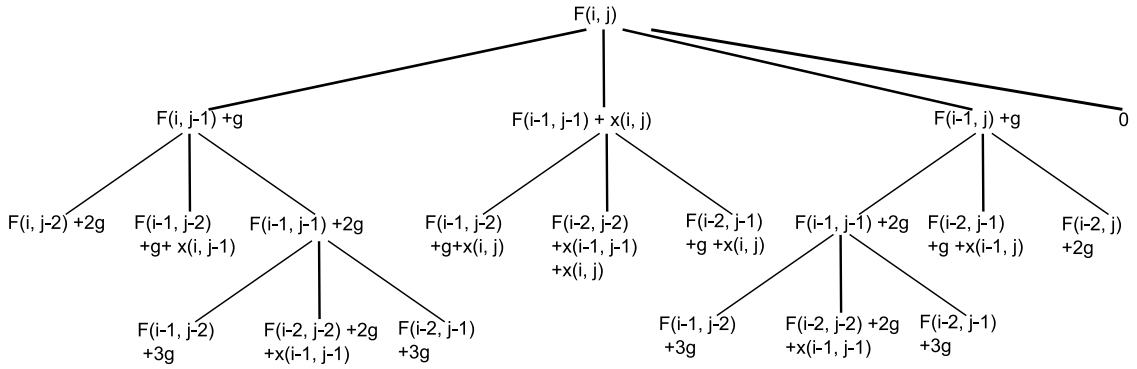


Figure 5. Three level recursion tree for the SW algorithm.

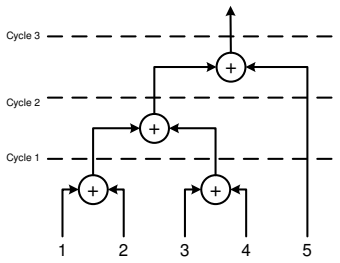


Figure 4. Circuit for Example 2.

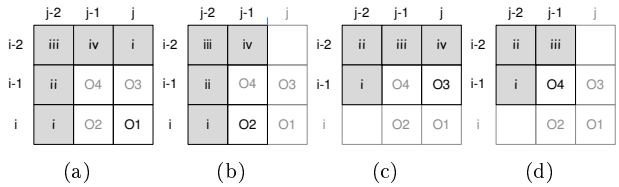


Figure 6. Matrices to show the elements from which $F(i-i^{'}, j-j^{'})$ are computed. Shaded square represent already known values.

using binary tree structure as shown in Figure 4.

2.3 Traditional acceleration of SW

As mentioned in Section 2.1, the best known hardware acceleration of the SW algorithm takes $m+n-1$ steps to complete (Equation 3). Since each step takes 4 cycles (Figure 3), the best known time to compute the SW equation is $4(m+n-1)$ cycles as described by Equation 1.

A lot of work has been done to accelerate the biological sequence alignment using different hardwares. In addition to specific architectures designed for sequence alignment, many solutions for special purpose hardware, SIMD and FPGAs have been devised [9].

Several implementations for SIMDs have been proposed as MGAP, Kestrel and Fuzion [5, 3, 15]. A recent implementation was done on Intel Xeon 2.0 GHz using a technique called Striped Smith-Waterman, which claims to achieve a speedup of six times over other SIMDs implementations [7]. SIMDs contains general purpose processors therefore it is programmable and is used for a wider range of applications like image processing and scientific computing. The drawback is that they are expensive.

Reconfigurable systems are good candidates for accelerating biological sequence alignment algorithms. Reconfigurable systems are composed of GPP coupled with Field Programmable Gate Arrays (FPGA). FPGAs are programmable using some hardware description languages like VHDL or Verilog and virtually any algorithm can be mapped on it. FPGAs can also be reconfigured during system operation, called Run-Time Reconfiguration, which makes them suitable if the algorithm or gap penalty is changed at runtime. Some of the solutions based on FPGAs are given in [18, 14]. Recently Jiang [11] modified the SW formula by introducing a new variable and thereby reducing the critical path to compute a single cell.

In this paper, we described how to improve the time needed to compute sequence alignment using any of the above methods by a constant factor by exposing more parallelism.

3 SW acceleration using RVE

3.1 Applying RVE to the SW algorithm

We applied RVE partially on Equation 1 to expose three levels of data parallelism. The recursion tree after the application of RVE is shown in Figure 5. *F(i, j)* can be written in equation as shown by the leaf nodes in Figure 5.

$$F(i,j)=\max\begin{cases} i & F(i,j-2)+2g \\ ii & F(i-1,j-2)+g+x(i,j-1) \\ iii & F(i-1,j-2)+3g \\ iv & F(i-2,j-2)+2g+x(i-1,j-1) \\ v & F(i-2,j-1)+3g \\ vi & F(i-1,j-2)+g+x(i,j) \\ vii & F(i-2,j-2)+x(i-1,j-1)+x(i,j) \\ viii & F(i-2,j-1)+g+x(i,j) \\ ix & F(i-1,j-2)+3g \\ x & F(i-2,j-2)+2g+x(i-1,j-1) \\ xi & F(i-2,j-1)+3g \\ xii & F(i-2,j-1)+g+x(i-1,j) \\ xiii & F(i-2,j)+2g \\ xiv & 0 \end{cases} \quad (4)$$

Equation 1, which transformed to Equation 4 is now written as the maximum of fourteen sub-equations. All the terms are independent to each other, therefore sub-equations can be computed in parallel. Since finding maximum is associative, then the efficient way to find maximum is by making a complete binary tree from the result of fourteen sub-equation, which requires four cycles as $\lceil \log_2 14 \rceil = 4$. Can we find $F(i,j)$ better than this? Yes, if we look closely at Equation 4, unique $F(i-i^{'}, j-j^{'})$, for $0 \leq i^{'} \leq 2$ and $0 \leq j^{'} \leq 2$, terms are only *five*. If a unique $F(i-i^{'}, j-j^{'})$ is present in more than one sub-equations and as mentioned before in Section 2.1 that the lowest score in substitution matrices is greater than the recommended gap penalty g, we can eliminate some sub-equations with out the loss of generality based on the smallest value of $x(i-i^{'}, j-j^{'})$, which we call x_l. For example, $F(i-1, j-2)$ is present in equation *ii, iii, vi* and *ix*. So these sub-equations can be written as follows.

$$\begin{aligned} ii &\quad F(i-1,j-2)+g+x_l \\ iii &\quad F(i-1,j-2)+3g \\ vi &\quad F(i-1,j-2)+g+x_l \\ ix &\quad F(i-1,j-2)+3g \end{aligned} \quad (5)$$

So sub-equations *iii* and *ix* can be simply discarded, they can never be maximum as $g+x_l > 3g$ for $x_l > g$. There is a tie between *ii* and *vi*, as we are not certain about the values does $x(i,j-1)$ and $x(i,j)$. Using this reduction method for all the sub-equations, Equation 4 can be reduced to the following equation of eight sub-equations.

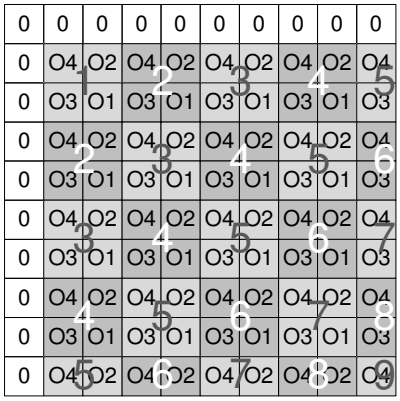


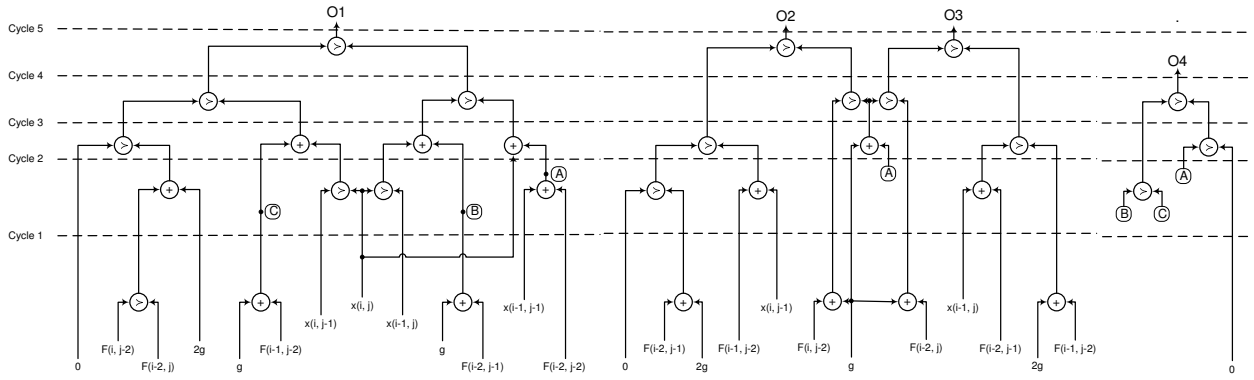
Figure 7. Sequence of fill of the $F(i,j)$ scoring matrix of Equation 1, starting from the top left light shaded square numbered 1 (represent the time instance to compute) and moving diagonally down as shown by trailing numbers. All the squares with the same number can be executed in parallel.

$$F(i,j)=\max\begin{cases} i & F(i,j-2)+2g \\ ii & F(i-1,j-2)+g+x(i,j-1) \\ iii & F(i-1,j-2)+g+x(i,j) \\ iv & F(i-2,j-2)+x(i-1,j-1)+x(i,j) \\ v & F(i-2,j-1)+g+x(i-1,j) \\ vi & F(i-2,j-1)+g+x(i,j) \\ vii & F(i-2,j)+2g \\ viii & 0 \end{cases} \quad (6)$$

To find the maximum of eight sub-equations, we need $\lceil \log_2 8 \rceil = 3$ cycles, which is better than 4 cycles as needed for Equation 4.

Even if the gap penalty is equal to the smallest value in the substitution matrix, the above equations will prevail as equations with only gap penalties will be eliminated, as with $x(i-i^{'}, j-j^{'})$, there is a chance that a better score can come up. However if the gap penalty g is greater than the smallest value x_l of substitution matrix , then there may or may not be any elimination and in worst case we may have to keep all the sub-equations, which means that 4 cycles will be required to find the maximum.

In implementation, we would like to reduce the hardware as much as possible for the same acceleration. Since the first two terms in the sub-equations *ii* and *iii* of Equation 6 are the same, we can write both of them



(a) Circuit for O1 (b) Circuit for O2 and O3 (c) Cir. for O4

Figure 8. Circuit for computing the 2×2 block as shown in Figure 6, where A = *x(i-1, j-1)+F(i-2, j-2),* B = *F(i-2, j-1)+g* and C = *F(i-1, j-2)+g.*

in one sub-equation as

$$\max \begin{cases} ii & F(i-1,j-2)+g+x(i,j-1) \\ iii & F(i-1,j-2)+g+x(i,j) \end{cases}$$
$$= F(i-1,j-2)+g+\{x(i,j-1) \succ x(i,j)\}$$

where $\succ$ is the *binary max* operation, which returns the max of two value. Similarly without losing generality, we can reduce the whole of Equation 6 into the following equation.

$$F(i,j)=\max \begin{cases} i & \{F(i,j-2) \succ F(i-2,j)\}+2g \\ ii & F(i-1,j-2)+g+\{x(i,j-1) \succ x(i,j)\} \\ iii & F(i-2,j-2)+x(i-1,j-1)+x(i,j) \\ iv & F(i-2,j-1)+g+\{x(i-1,j) \succ x(i,j)\} \\ v & 0 \end{cases} \quad (7)$$

Equation 7 when mapped on to matrix form gives us a 3×3 matrix, where the terms to be computed (*O1* to *O4*) are represented by a 2×2 block as shown in Figure 6. We define the size of the unknown block as the *blocking factor (b)*, here *b=2*. Figure 6(a) shows how $F(i, j)$ (i.e. *O1*) is calculated from Equation 7. Similarly we can compute $F(i, j-1)$ (i.e. *O2* in Figure 6(b)), $F(i-1, j)$ (i.e. *O3* in Figure 6(c)) and $F(i-1, j-1)$ (i.e. *O4* in Figure 6(d)) using the similar method for *O1*. The formulas for $F(i, j-1)$, $F(i-1, j)$ and $F(i-1, j-1)$ after applying Recursive Variable Expansion partially and elimination is given by Equation 8, 9 and 10, respectively.

$$F(i,j-1)=\max \begin{cases} i & F(i,j-2)+g \\ ii & F(i-1,j-2)+x(i,j-1) \\ iii & F(i-2,j-2)+g+x(i-1,j-1) \\ iv & F(i-2,j-1)+2g \\ v & 0 \end{cases} \quad (8)$$

$$F(i-1,j)=\max \begin{cases} i & F(i-1,j-2)+2g \\ ii & F(i-2,j-2)+g+x(i-1,j-1) \\ iii & F(i-2,j-1)+x(i-1,j) \\ iv & F(i-2,j)+g \\ v & 0 \end{cases} \quad (9)$$

$$F(i-1,j-1)=\max \begin{cases} i & F(i-1,j-2)+g \\ ii & F(i-2,j-2)+x(i-1,j-1) \\ iii & F(i-2,j-1)+g \\ iv & 0 \end{cases} \quad (10)$$

Once the boundary conditions are applied, the rest of the matrix can be filled as shown by Figure 7. The figure also shows how the matrix will be filled if the length of the sequences is not a multiple factor of the blocking factor.

4 Time and Hardware estimation

In this section, we will show that our approach in which we have expanded the SW (Equation 1) using Recursive Variable Expansion is some constant times faster than any known parallel implementation to date,

Table 1. Time and Hardware estimation

	Time (cycles)			Hardware								
				+			≻			LUT		
	variable	value[1]	speedup[2]	variable	value[1]	overhead ratio[2]	variable	value[1]	overhead ratio[2]	variable	value[1]	overhead ratio[2]
Serial case	$4mn$	1000000	1	3	3	1	3	3	1	1	1	1
Best HW accel.	$4(m+n-1)$	3996	250	$3\times l_d$	1500	500	$3\times l_d$	1500	500	$1\times l_d$	500	500
RVE with b=2	$5(\lceil \frac{m}{2} \rceil + \lceil \frac{n}{2} \rceil - 1)$	2495	401	$14\times n_2$	3500	1167	$17\times n_2$	4250	1417	$4\times n_2$	1000	1000
RVE with b=3	$7(\lceil \frac{m}{3} \rceil + \lceil \frac{n}{3} \rceil - 1)$	2331	429	$54\times n_3$	27000	9000	$54\times n_3$	27000	9000	$9\times n_3$	1500	1500

[1]values calculated for m=500 & n=500, [2]with respect to the serial case

l_d=min(m,n)=500, n_2=min($\lfloor \frac{m}{2} \rfloor, \lfloor \frac{n}{2} \rfloor$)+$\frac{\min(m,n) \bmod 2}{2}$=250[1], n_3=min($\lfloor \frac{m}{3} \rfloor, \lfloor \frac{n}{3} \rfloor$)+$\frac{\min(m,n) \bmod 3}{3}$=167[1]

depending upon the size of the blocking factor chosen. Next in this section, we will discuss the different known SW implementations. Then we will show the time and hardware estimation along with its comparison with the best known parallel approach.

4.1 Serial Case

First we will look at the *serial case*, in which every element is calculated serially in hardware starting from top left corner and moving left to right and top to down. As shown in Figure 3, it takes 4 cycles to compute an element by using 3 adders and 4 comparators. There are mn elements in total in the scoring matrix, therefore to align two sequences of length m and n, it requires $4mn$ cycles. As all the elements are computed serially, the hardware is required for just one element, which is 3 adders and 4 comparators.

4.2 Best known hardware acceleration

Now we will look at the best known hardware acceleration technique, in which the elements $F(i,j)$ are computed as parallel as possible restrained by the data dependency. As given by Equation 3, the number of subsequent steps required is $m+n-1$, which takes a total computation time of $4(m+n-1)$ cycles. At any time, the number of elements in the longest anti-diagonal is the maximum number of elements to be computed in parallel, given by Equation 2 is $l_d = \min(m,n)$. If h_e quantifies the amount of hardware used to compute a single $F(i,j)$ element, then the maximum amount of hardware needed is $h_e \times l_d$.

4.3 Recursive variable expansion

In order to estimate the hardware and time for our implementation, we have drawn the circuits in Figure 8 for Equations 7, 8, 9 and 10. The circuits for the given equations are optimized to use minimum hardware. These circuits can be easily implemented on FPGA. According to the circuits in Figure 8, a block of 2×2 requires 5 cycles to compute. The maximum number of sequential blocks that should be calculated in subsequent anti-diagonals for sequences of lengths m and n is given by $\lceil \frac{m}{b} \rceil + \lceil \frac{n}{b} \rceil - 1$. The last block can be partially filled, if $b \nmid m$ or $b \nmid n$ ($b \nmid m$ means that b does not evenly divide m). Therefore for 2×2 block, upper bound for the time to compute an alignment between two sequences of length m and n is $5(\lceil \frac{m}{2} \rceil + \lceil \frac{n}{2} \rceil - 1)$ cycles. Even if the gap penalty is greater than the lowest score in substitution matrix, the time for sequence alignment in that case will be $6(\lceil \frac{m}{2} \rceil + \lceil \frac{n}{2} \rceil - 1)$ cycles, which is approximately equal to $3(m+n-1)$, is still better than $4(m+n-1)$ of the best hardware acceleration case. The number of blocks required to be computed in parallel is $n_b = \min(\lfloor \frac{m}{b} \rfloor, \lfloor \frac{n}{b} \rfloor) + p$, which is the length of longest diagonal in blocks, where $p = 0$, if $b \mid \min(m,n)$, otherwise $0 < p < 1$, which means the block is partially filled and $p = \frac{\min(m,n) \bmod b}{b}$. If h_b is some number of hardware used to compute a block and it is also assumed that the hardware used for a partial block is equal to the ratio of the partial block size to the actual block size, then the total hardware used is $n_b \times h_b$.

Following is the estimate of hardware h_2 used by one block of 2×2 as verified by the Figure 8.

No. of '+' used = 14
No. of '≻' used = 17
No. of LUT used = 4

Similarly, we have expanded the SW Equation 1 further to blocking factor $b = 3$. We got $3 \times 3 = 9$ equations for unknowns and then we drew the optimized circuits for all those equations to get an estimate about the time and hardware they would take. If all the nine equations are computed in parallel, then it takes only seven cycles to compute a block of 3×3 elements. The hardware estimates h_3 are as under.

No. of '+' used = 54
No. of '$\succ$' used = 54
No. of LUT used = 9

4.4 Summary of result

The time and hardware estimation for all the techniques is summarized in Table 1. The estimate for time and hardware is given in generic terms of m and n as well as with some specific values, *m=n=500,* which is typical length in [4] to simplify the comparison. The best known parallel technique is linear in m and n as compared to quadratic in m and n in the serial case. In case of *m=n=500*, it is 250 times faster than the *serial case*. This acceleration comes at the expense of 500 times the hardware required by the *serial case*. When the SW is accelerated with RVE with blocking factor *b=2*, the speedup is 401 times the *serial case* and the hardware used is around 1250 times the *serial case*. Similarly, for RVE with blocking factor *b=3*, the speedup is increased to 429 times the *serial case* and the hardware used is around 9000 times the *serial case*. It is clear from this trend that hardware utilization is more than linear as compared to speedup beyond the best known hardware acceleration. The reason is that in RVE, we have given priority to parallelization as compared to the hardware utilization and do many repeated computation to achieve the speedup, which increases the hardware utilization. The speedup can be increased further by increasing the blocking factor provided we can dedicate more hardware for that.

5 Conclusion

In this paper, we have presented a new technique to parallelize the SW algorithm with linear gap penalties, which has the capability to expose more parallelism than the prevalent parallel techniques. We have shown that this techniques increases the speedup by a factor of *1.6* and *1.71* for blocking factor *b=2* and *b=3* respectively, as compared to the best known parallel technique. This does not represent the maximum achievable speedup using this method, rather we can improve this further by increasing the blocking factor, given we have enough hardware.

References

[1] S. F. Altschul et al. Basic local alignment search tool. *J. Mol. Biol.*, pages 403–410, 1990.

[2] D. A. Benson et al. Genbank. *Nucl. Acids Res.*, 28(1):15–18, 2000.

[3] A. D. Blas et al. The kestrel parallel processor. *IEEE Transactions on Parallel and Distributed Systems*, 16(1):80–92, 2005.

[4] B. Boeckmann et al. The swiss-prot protein knowledgebase and its supplement trembl in 2003. *Nucleic Acids Research*, 31:365–370, 2003.

[5] M. Borah et al. A simd solution to the sequence comparison problem on the mgap. In *ASAP*, 1994.

[6] M. O. Dayhoff. Survey of new data and computer methods of analysis. In *Atlas of Protein Sequence and Structure*, volume 5, page 29, 1978.

[7] M. Farrar. Striped smith-waterman speeds database searches six times over other simd implementations. *Bioinformatics*, 23(2):156–161, 2007.

[8] M. Y. Galperin. The molecular biology database collection: 2007 update. *Nucleic Acids Research*, 35:D3–D4(1), January 2007.

[9] L. Hasan, Z. Al-Ars, and S. Vassiliadis. Hardware acceleration of sequence alignment algorithms - an overview. In *DTIS'07*, pages 96–101, September 2007.

[10] S. Henikoff and J. Henikoff. Amino acid substitution matrices from protein blocks. In *Proceedings of the National Academy of Sciences*, volume 89, pages 10915–10919, 1992.

[11] X. Jiang et al. A reconfigurable accelerator for smith-waterman algorithm. *IEEE Transactions on Circuits and Systems II*, 54(12):1077–1081, Dec. 2007.

[12] D. J. Lipman and W. R. Pearson. Rapid and sensitive sequence comparison with fastp and fasta. *Methods Enzymol.*, 183:63–98, 1990.

[13] Z. Nawaz et al. Recursive variable expansion: A loop transformation for reconfigurable systems. In *ICFPT'07*, 2007.

[14] T. Oliver, B. Schmidt, and D. Maskell. Reconfigurable architectures for bio-sequence database scanning on fpgas. *IEEE Transactions onCircuits and Systems II*, 52(12):851–855, Dec. 2005.

[15] B. Schmidt, H. Schröder, and M. Schimmler. Massively parallel solutions for molecular sequence analysis. In *IPDPS '02*, page 201, 2002.

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

[17] D. States, W. Gish, and S. Altschul. Improved sensitivity of nucleic acid database search using application-specific scoring matrices. In *Methods: A companion to Methods in Enzymology*, volume 3, pages 66 – 77, 1991.

[18] Y. Yamaguchi, Y. Miyajima, T. Maruyama, and A. Konagaya. High speed homology search using runtime reconfiguration. In *FPL '02*, pages 281–291, 2002.