BIOINFORMATICS ORIGINAL PAPER

Vol. 23 no. 2 2007, pages 156–161
doi:10.1093/bioinformatics/btl582

*Sequence analysis*

# Striped Smith–Waterman speeds database searches six times over other SIMD implementations

Michael Farrar

Received on June 22, 2006; revised on November 13, 2006; accepted on November 14, 2006
Advance Access publication November 16, 2006
Associate Editor: Nikolaus Rajewsky

**ABSTRACT**

**Motivation:** The only algorithm guaranteed to find the optimal local alignment is the Smith–Waterman. It is also one of the slowest due to the number of computations required for the search. To speed up the algorithm, Single-Instruction Multiple-Data (SIMD) instructions have been used to parallelize the algorithm at the instruction level.

**Results:** A faster implementation of the Smith–Waterman algorithm is presented. This algorithm achieved 2–8 times performance improvement over other SIMD based Smith–Waterman implementations. On a 2.0 GHz Xeon Core 2 Duo processor, speeds of >3.0 billion cell updates/s were achieved.

**Availability:** http://farrar.michael.googlepages.com/Smith-waterman

**Contact:** farrar.michael@gmail.com

## 1 INTRODUCTION

The Smith–Waterman (Smith and Waterman, 1981) algorithm is one of the slowest and most sensitive sequence search algorithms. As the size of the GenBank/EMBL/DDBJ double every 15 months (Benson *et al*., 2000), faster implementations of the Smith–Waterman algorithm are needed to keep pace. One recent optimization has been adopting the algorithm to Single-Instruction Multiple-Data (SIMD) microprocessors. A SIMD instruction is able to perform the same operation on multiple pieces of data in parallel.

One of the first Smith–Waterman SIMD implementations was Alpern *et al*. (1995). This approach divided the 64-bit Z-buffer registers of the Intel Paragon i860 processor into four parts. Each part of the register contained a value from four different database sequences. A 6-fold speedup was achieved over the original implementation.

Wozniak (1997) presented an implementation of the Smith–Waterman algorithm running on the Sun Ultra SPARC using its SIMD instructions, the Visual Instruction Set. The SIMD registers contained values parallel to the minor diagonal. An advantage to this implementation is that there are no conditional branches in the inner loop. Therefore, the execution time is dependent on the length of the query string and the database, not the scoring matrix or gap penalties. A major drawback of this implementation is the query profile must be computed for each database sequence. A speedup of over two times was reported over the traditional implementation.

Rognes and Seeberg (2000) presented an implementation of the Smith–Waterman algorithm running on the Intel Pentium processor using the MMX SIMD instructions. The SIMD registers contained values parallel to the query sequence. A major optimization was computing the query profile once for the entire database search. A disadvantage introduced by processing the values vertically is that conditional branches are placed in the inner loop to compute $F$. With conditional code the execution time is dependent on the length of the query string and the database, the scoring matrix and gap penalties. A speedup of over six times was reported over an optimized non-SIMD implementation.

This paper presents a new Smith–Waterman implementation where the SIMD registers are parallel to the query sequence, but are accessed in a striped pattern. Like the Rognes implementation, the query profile is calculated once for the database search, but the conditional $F$ calculations are moved outside the inner loop. Calculations speeds of >3.0 GCUPS are achieved. This is a speedup of 2–8 times over the Wozniak and Rognes SIMD implementations.

## 2 METHODS

### 2.1 Smith–Waterman algorithm

The algorithm used to compute the optimal local alignment is the Smith–Waterman (Smith and Waterman, 1981) with the Gotoh (1982) improvements for handling multiple sized gap penalties. The two sequences to be compared, the query sequence and the database sequence, are defined as $Q = q_1, q_2 \dots q_m$ and $D = d_1, d_2 \dots d_n$. The length of the query sequence and database sequence are $m = |Q|$ and $n = |D|$, respectively. A scoring matrix $W(q_i, d_j)$ is defined for all residue pairs. Usually the weight $W(q_i, d_j)$ $\leq 0$ when $q_i <> d_j$ and $W(q_i, d_j) > 0$ when $q_i = d_j$. The penalty for starting a gap and continuing a gap are defined as $G_{\text{init}}$ and $G_{\text{ext}}$, respectively. The alignment scores ending with a gap along $D$ and $Q$ are $E$ Equation (1) and $F$ Equation (2), respectively.

$$E_{i,j} = \max \left\{ \begin{array}{l} E_{i,j-1} - G_{\text{ext}} \\ H_{i,j-1} - G_{\text{init}} \end{array} \right\} \quad (1)$$

$$F_{i,j} = \max \left\{ \begin{array}{l} F_{i-1,j} - G_{\text{ext}} \\ H_{i-1,j} - G_{\text{init}} \end{array} \right\} \quad (2)$$

The alignment score for $H_{ij}$ where $1 \leq i \leq m$ and $1 \leq j \leq n$ is defined by Equation (3).

$$H_{i,j} = \max \left\{ \begin{array}{l} 0 \\ E_{i,j} \\ F_{i,j} \\ H_{i-1,j-1} - W(q_i, d_j) \end{array} \right\} \quad (3)$$

The values for $H_{ij}$, $E_{ij}$ and $F_{ij}$ are equal to 0 when $i < 1$ or $j < 1$.

### 2.2 Implementation

*2.2.1 Striped query profile* When calculating $H_{i,j}$ the value from the scoring matrix $W(q_i, d_j)$ is added to $H_{i-1,j-1}$. To avoid the lookup of $W(q_i, d_j)$ for each cell, Rognes and Seeberg (2000) calculated a query profile parallel to the query for each possible residue. The query profile is calculated just once for each database search. Then the calculation of $H_{i,j}$ requires just an addition of the pre-calculated score to the previous $H_{i,j}$. The striped Smith–Waterman implementation takes a similar approach by pre-calculating the query profile, but with a different layout than Rognes (Fig. 1).

The layout used by the query profile is a striped access parallel to the query sequence. The query is divided into equal length segments, $S$. The number of segments, $p$, is equal to the number of elements being processed in the SIMD register. When processing byte integers (8 bit values) the $p = 16$ and when processing word integers (16 bit values) $p = 8$. The length of each segment, $t$, is $(|Q| + p - 1)/p$. If the query is not long enough to fill all the segments, $t > |Q|$, the segments are padded with null entries that have a weight of zero. The query segments are defined as $S_n = q_{t(n-1)+1}, q_{t(n-1)+2}, \ldots, q_{t(n-1)+t}$ where $1 \le n \le p$.

Each element of the SIMD registers maps to one segment. The first element in the vector maps to $S_1$, the second element in the vector maps to $S_2$, till the last element in the vector maps to $S_p$. The vectors move uniformly across the segments, so $\langle H_{i,j} \rangle$ processes the $i$-th element of all the segments. Equation (4) shows the segment layout when $p = 8$ and the elements processed by the SIMD register when $i = 2$.

$$\begin{array}{rc|c|ccc}
\cline{3-3}
S_1 = & q_1 & q_2 & q_3 & \cdots & q_t \\
S_2 = & q_{t+1} & q_{t+2} & q_{t+3} & \cdots & q_{2t} \\
S_3 = & q_{2t+1} & q_{2t+2} & q_{2t+3} & \cdots & q_{3t} \\
S_4 = & q_{3t+1} & q_{3t+2} & q_{3t+3} & \cdots & q_{4t} \\
S_5 = & q_{4t+1} & q_{4t+2} & q_{4t+3} & \cdots & q_{5t} \\
S_6 = & q_{5t+1} & q_{5t+2} & q_{5t+3} & \cdots & q_{6t} \\
S_7 = & q_{6t+1} & q_{6t+2} & q_{6t+3} & \cdots & q_{7t} \\
S_8 = & q_{7t+1} & q_{7t+2} & q_{7t+3} & \cdots & q_{8t} \\
\cline{3-3}
\end{array} \tag{4}$$

The vectors making up the scoring profile $W$, like the $H$ vectors, also moves uniformly across the segments. The layout of the scoring profile, when $p = 8$, is:

$$\begin{array}{rllll}
\langle W_{1j} \rangle = & \{W(q_1, d_j), & W(q_{t+1}, d_j), & \cdots & W(q_{7t+1}, d_j)\}, \\
\langle W_{2j} \rangle = & \{W(q_2, d_j), & W(q_{t+2}, d_j), & \cdots & W(q_{7t+2}, d_j)\}, \\
\cdots & & & \cdots & \\
\langle W_{tj} \rangle = & \{W(q_t, d_j), & W(q_{2t}, d_j), & \cdots & W(q_{8t}, d_j)\}
\end{array}$$

The query profile is stored in memory on 16-byte boundaries. By aligning the profile on a 16-byte boundary, the values are read with a single aligned load instruction, which is faster than reading unaligned data. Figure 2 has the pseudo code for generating the query profile.

Both the Wozniak (1997) and Rognes and Seeberg (2000) implementations have data dependencies between the previous $H$ vector and the current $H$ vector, Figure 3. This is also true when calculating $F$. Before $H$ or $F$ are calculated, the last element in the previous vector is moved to the first element in the current vector. By using the striped query access, these data dependencies are moved out of the inner loop and done just once in the outer loop when processing the next database residue.

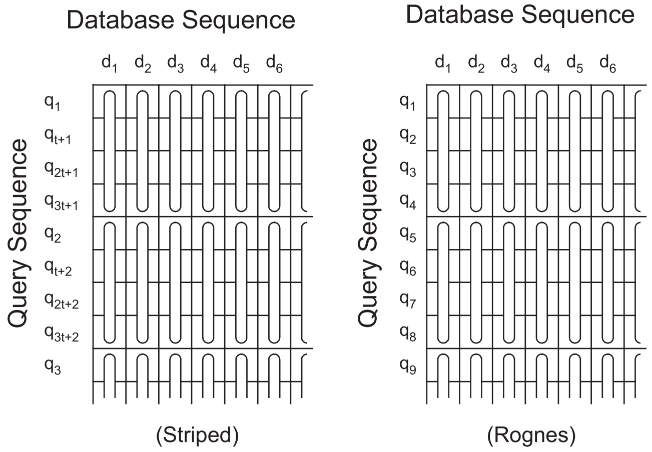

**Fig. 1.** The memory layout for the query profile used by the striped and Rognes implementations. The vectors in both implementations run parallel to the query sequence, but the striped implementation access the query in a stripped pattern unlike the sequential access for Rognes.

```
// Calculate the length of the segments so that the
//  query fits evenly in the different segments.
segLen := (length (Q) + 15) / 16;

// Build the query profile for all possible residues
foreach a in AminoAcids
   h := 0;
   for i := 0 … segLen
      j := i ;
      for k := 1 … 16
         if (j > length (Q))
            // We are beyond the length of the query
            // string so pad the weights with neutral
            // scores.
            queryProfile[a][h] := 0;
         else
            // Set the score to the weight in the scoring
            // matrix.
            queryProfile[a][h] := W(a,q[j]);
         endif
         h := h + 1;
         j := j + segLen;
      endfor
   endfor
endfor
```

**Fig. 2.** Pseudo code for generating the query profile for SIMD registers processing 16 elements.

*2.2.2 Smith–Waterman SIMD implementation* The striped Smith–Waterman implementation was written for Intel processors supporting SSE2 instructions. The pseudo code for the implementation is shown in Figure 5. The code was written in C using Intel's SSE2 intrinsics for portability.

To maximize the number of cells calculated per instruction, the SIMD SSE2 registers are divided into their smallest unit possible. The 128-bit wide registers are divided into 16 8-bit elements for processing. One instruction can therefore operate on 16 cells in parallel. Dividing the register into 8-bit elements limits the cell's range from 0 to 255. In most cases, the scores fit in the 8-bit range unless the sequences are long and similar. If a query's score exceeds the cells maximum, that query is recalculated using a higher precision.

For those queries that do require a higher precision, the register is divided into 8, 16-bit elements. This gives each cell a possible range from 0 to 65 535. The obvious drawback to using 16-bit elements is now each instruction is processing half as many cells per instruction compared with using 8-bit elements.

Due to limitations in the SSE2 instruction set, unsigned byte integers are used in the 8-bit calculations. The SSE2 instruction set supports only maximum on unsigned byte integers. Since the maximum instruction is needed to compute $E$, $F$ and $H$, unsigned bytes are used in the low precision calculations. To use unsigned bytes, the query profile is biased to the smallest value in the scoring matrix. After $W$ is added to $H$, the bias is subtracted from $H$. When the score is >255 − bias, the search is rerun with higher precision calculations. This approach was used in the Rognes and Seeberg (2000) implementation.

For the higher precision calculations signed short integers are used to speed up the inner loop. When using signed integers, the initial $E$, $F$ and

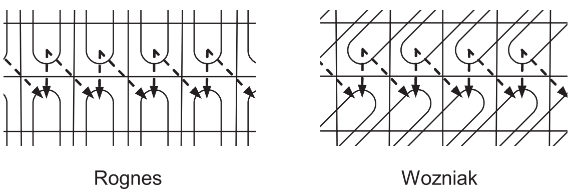

**Fig. 3.** Data dependencies between SIMD registers holding the $H$ and $F$ values with the Rognes and Wozniak implementations. The last element in the previous vector is inserted in the current vector when calculating the next $H$ and $F$ vectors.

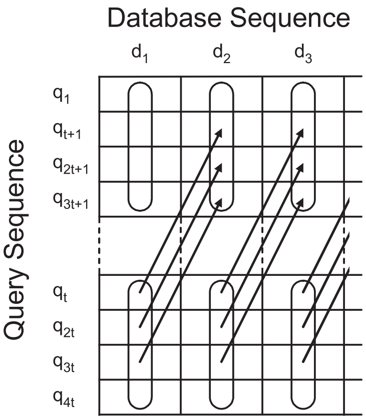

**Fig. 4.** Data dependencies between the last $H$ vector and the first $H$ vector of the next column. The values in the last $H$ vector are shifted to the left so the values are aligned with the next segment over.

$H$ values are biased by the minimum signed short integer value, $-32\ 768$ instead of 0. By biasing the initial value with its minimum possible value, the complete range of the element can be used. When the search is done the bias is added back to $H$ returning a score between 0 and 65 535. Using signed arithmetic, the weight is not biased, therefore the instruction subtracting bias from $H$ is not needed in the inner loop keeping calculation times down.

The $H_{i,j}$ calculation is dependent on the previous value on the major diagonal, $H_{i-1,j-1}$. To simplify the code for handling this dependency, two buffers are allocated for storing the $H$ values. On the first pass, one buffer is used to read the previous $H$ values and the other buffer is used to store the new $H$ values. On the next pass, the buffers are swapped, so the input $H$ buffer is now the output $H$ buffer and vice-versa.

The computation of $\langle H_{i,j}\rangle$ where $1 \leq i \leq T$, is the addition of the weight $\langle W_{i,j}\rangle$ to $\langle H_{i-1,j-1}\rangle$ to access the $H$ values on the major diagonal. If $i = 1$ then $\langle H_{T,j}\rangle$ is shifted to the left by one element and added to $\langle W_{1,j}\rangle$. Figure 4 shows the data dependencies between the last $H$ vector and the first $H$ vector of the next column. The inner loop therefore no longer requires the extra operations to insert the $H$ value into the next SIMD register. The only shifting of $H$ is done once in the outer loop to get $\langle H_{T,j}\rangle$ in the correct order.

The computation of $\langle E_{i,j}\rangle$ where $1 \leq i \leq T$, is the subtraction of the gap extension penalty, $G_{\text{ext}}$, from $\langle E_{i,j-1}\rangle$ to access the $E$ values to the left of the current cells. If $i = 1$ then zeros are used for the value of $E$.

The computation of $\langle F_{i,j}\rangle$ where $1 \leq i \leq T$, is the subtraction of the gap extension penalty, $G_{\text{ext}}$, from $\langle F_{i-1,j}\rangle$ to access the $F$-values above the current cells. If $i = 1$ then the initial calculation of $\langle F_{1,j}\rangle$ is dependent on $\langle F_{T,j}\rangle$ shifted to the left by one. Since the values of $\langle F_{1,j}\rangle$ are unknown until the inner loop has completed, zeros are substituted and any errors introduced are corrected in a second pass.

*2.2.3 Lazy F evaluation* For most cells in the matrix, $F$ remains at zero and does not contribute to the value of $H$. Only when $H$ is greater than $G_{\text{init}} + G_{\text{ext}}$ will $F$ start to influence the value of $H$. In many instances the second pass at correcting errors introduced by $F$ is not necessary. Figure 5 has the pseudo code for the lazy $F$ loop. After the inner loop has completed $\langle F_{T,j}\rangle$ is checked against the values of $\langle H_{1,j}\rangle$ to see if the second pass is necessary. The values in $\langle F_{T,j}\rangle$ are shifted to the left by one and if any elements are greater than $\langle H_{1,j}\rangle - G_{\text{init}}$, then $H$ is recalculated because $F$ can change the value of $H$.

The second pass loop is executed until all elements in $F$ are less than $H - G_{\text{init}}$. If this loop processes all the segments without an early exit, an additional pass might be needed to recalculate $F$. Since each element in the vector represents a different segment of the query sequence, after processing the last vector $\langle F_{T,j}\rangle$, the values in $\langle F_{T,j}\rangle$ are shifted to the left by one to move their values to the next segment. Figure 6 shows the data dependencies between the last $F$ vector and the first. If any elements in $\langle F_{T,j}\rangle$ are still greater than $\langle H_{1,j}\rangle - G_{\text{init}}$, the loop is executed again. This loop is repeated until all elements in $F$ are below the threshold.

One advantage of this approach is all branches are moved out of the inner loop to the outer loop. Modern processors use branch prediction to limit the impact of branching on the run time. As execution pipelines get deeper to support higher clock rates, the penalty for a misprediction increases and therefore conditional branches should be eliminated if possible (Intel, Optimization Ref. Man.). The execution pipeline on the Pentium 4 is documented as being twice as long as for the Pentium III, which should indicate that the branch misprediction penalty is at least 20 cycles (Hinton *et al.*, 2001).

## 3 RESULTS

To get meaningful comparisons of the different execution times, a test framework was developed to use both the Wozniak (1997) and Rognes and Seeberg (2000) Smith–Waterman implementations along with the striped algorithm presented in this paper. All three Smith–Waterman implementations were written in C using Intel SSE2 intrinsic functions. The programs were compiled using Microsoft Visual C++ 2005 with optimization set for maximum speed. By using SSE2 intrinsic functions instead of assembler, the compiler was responsible for optimizations, such as register usage, instruction selection and instruction scheduling.

The programs were tested on a 2.0 GHz Xeon Core 2 Duo processor with 2 GB of RAM running Windows XP SP2. The program was run as a single-threaded application, so the number of CPU cores had no affect on the execution time. All queries were run against Swiss-Prot release 49.1 comprising 75 841 138 amino acids in 208 005 sequence entries. To test the different Smith–Waterman implementations, 11 query sequences were used ranging is size from 143 to 567 amino acids. These sequences were used to test other algorithms including BLAST 2 (Altschul *et al.*, 1997) and SWMMX (Rognes and Seeberg, 2000) Smith–Waterman implementation.

To test the different Smith–Waterman implementations, three different tests were run using different scoring matrices and different gap penalties. The two scoring matrices used in the testing were BLOSUM62 and BLOSUM50 (Henikoff *et al.*, 1992). The two scoring matrices were used to test the runtime performance of the Rognes and striped implementations. One factor affecting the runtime performance of the Rognes and striped implementations is the scoring matrix. If the $H$ values get above $G_{\text{init}}$, then the $F$ values need to be calculated. The other factor affecting the number of $F$ calculations is the gap penalties. One

```
// Calculate the number of iterations needed to
// process the query string.  This calculation assumes
// there are 16 elements in the SIMD register.
segLen := (length (Q) + 15) / 16;

// Outer loop to process the database sequence
for i := 0 … dbLen
   // Initialize F value to zeros.  Any errors to vH values
   // will be corrected in the Lazy-F loop.
   vF := <0, …, 0>;

   // Adjust the last H value to be used in the next
   // segment over
   vH := vHStore[segLen - 1] << 1;

   // Swap the two H buffers
   swap (vHLoad, vHStore);

   // Inner loop to process the query sequence
   for j := 0 … segLen
      // Add the scoring profile to vH
      vH := vH + vProfile[i][j];

      // Save any vH values greater than the max
      vMax := max (vMax, vH);

      // Adjust vH with any greater vE or vH values
      vH := max (vH, vE[j]);
      vH := max (vH, vF);

      // Save the vH values off
      vHStore[j] := vH;

      // Calculate the new vE and vF based on the
      // gap penalties for this search
      vH := vH – vGapOpen;
      vE[j] := vE[j] – vGapExtend;
      vE[j] := max (vE[j], vH);
      vF := vF – vGapExtend;
      vF := max (vF, vH);

      // Load the next vH value to process
      vH := vHLoad[j];
   endfor

   // - - -   Lazy-F Loop   - - -
   // Shift the vF left so its values can be used to
   // correct the next segment over.
   vF := vF << 1;

   // Correct the vH values until there are no elements
   // in vF that could influence the vH values.
    j := 0;
   while (AnyElement (vF > vHStore[j] – vGapOpen)
      vHStore[j] := max (vHStore[j], vF);
      vF := vF – vGapExtend;
      if (++j >= segLen)
         // If we processed the entire segment, we need
         // to carry the vF values to the next segment.
         vF := vF << 1;
         j := 0;
      endif
   endwhile
endfor
```

**Fig. 5.** The pseudo code for the striped Smith–Waterman implementation, when processing 16 elements per SIMD register. The code is made up of the inner loop, which does the basic dynamic programming calculations followed by the lazy $F$ loop to correct any errors to $H$.

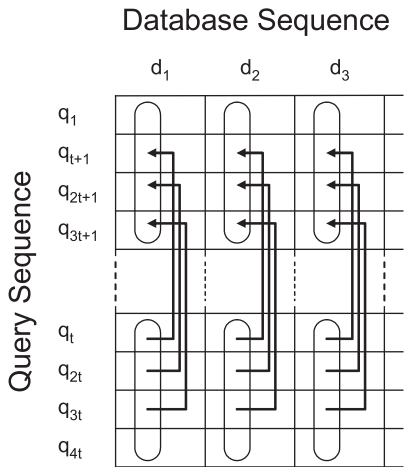

**Fig. 6.** The Data dependencies between the last $F$ vector and the first. The values in the last $F$ vector are shifted to the left so the values are aligned with the next segment over.

test uses different gap penalties to show the runtime characteristics of the different implementations. The higher the gap open and gap extension penalties, the fewer iterations are needed to calculate $F$. The Wozniak implementation is not affected by different scoring matrices or gap penalties, since there are no conditional calculations of $F$. An additional comparison was run comparing the execution times of the striped Smith–Waterman and different search programs using heuristic algorithms. All tests were run three times with the minimum scan time used as the final result. The MCUP rating (million cell updates per second) was calculated by $|Q| * |D|/$ ScanTime/$10^6$.

For the first test, the scoring matrix BLOSUM62 with a gap-open penalty of 10 and a gap-extension penalty of 1 were used. The same scoring matrix and gap penalties were used to evaluate BLAST2 and SWMMX. The search times for each of the 11 query sequences are shown in Figure 7. The Wozniak implementation completed the search in 821 s with an average of 352 MCUPS and a peak of 367 MCUPS. The Rognes implementation turns in a better search time of 354 s with an average of 816 MCUPS and a peak of 865 MCUPS. Finally, the striped implementation completed the search in 113 s with an average of 2553 MCUPS and a peak of 2998 MCUPS.

The next test used the same gap penalty, 10 − k, but utilized the BLOSUM50 scoring matrix. The results are shown in Figure 8. With the higher $H$ scores, more time was spent calculating the value of $F$. The Wozniak implementation stayed the same taking a total of 821 seconds still averaging 351 MCUPS with a peak of 367 MCUPS. The Rognes implementation turned in a slightly better time of 771 s with the average MCUPS dropping to 374 with a peak of 419 MCUPS. The striped implementation was also affected by the higher $H$ values taking 159 s to run the search averaging 1817 MCUPS with a peak of 2256 MCUPS.

The third test used the same 11 query sequences with the BLOSUM50 and BLOSUM62 scoring matrices, but with four different gap penalties, 10 − k, 10 − 2k, 14 − 2k and 40 − 2k. The total search times for all of the 11 query sequences are shown in Figure 9. With a large gap penalty of 40 − 2k, most of the $F$ calculations were skipped for the Rognes and striped implementations, basically testing just the efficiency of the inner loop. The

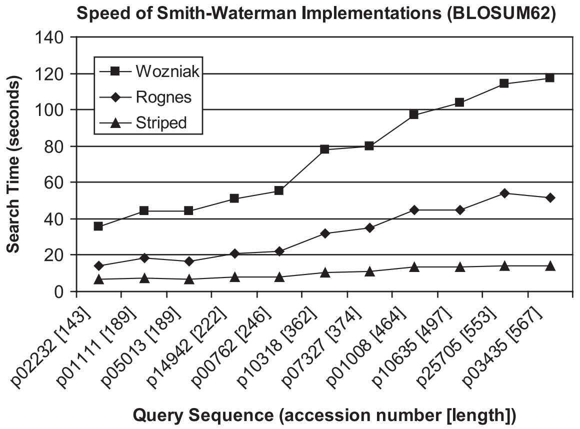

**Fig. 7.** The calculation times for the different Smith–Waterman implementations using the BLOSUM62 scoring matrix with a gap penalty of 10 − k.

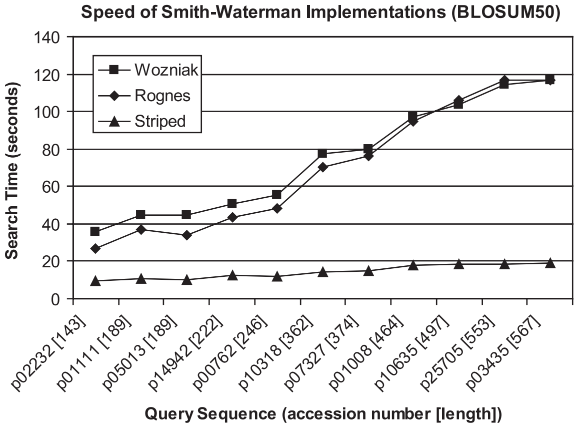

**Fig. 8.** The calculation times for the different Smith–Waterman implementations using the BLOSUM50 scoring matrix with a gap penalty of 10 − k.

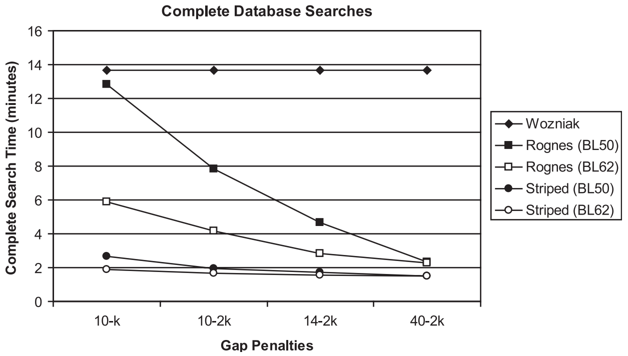

**Fig. 9.** Total calculation times for the different Smith–Waterman implementations using BLOSUM50 and BLOSUM62 with gap penalties of 10 − k, 10 − 2k, 14 − 2k and 40 − 2k.

Wozniak implementation total search time was 13.68 min. The search times for the Rognes implementation using the gap penalty of 40 − 2k, took 2.31 min for both scoring matrices, a 60–80% improvement over the 10 − k times. The striped implementation using the gap penalty of 40 − 2k took 1.51 min, only a 20–40% improvement over the 10 − k times.

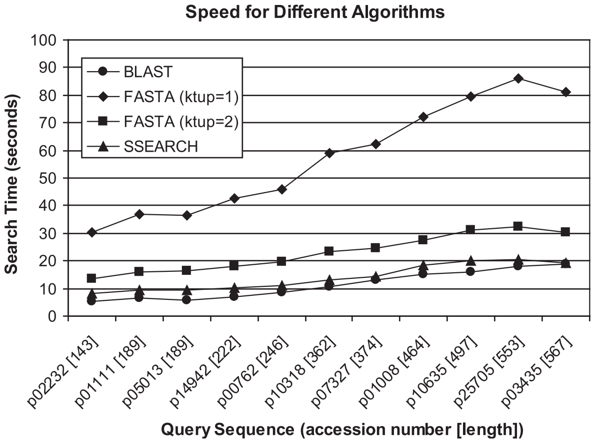

**Fig. 10.** Search times for different programs using heuristic algorithms and SSEARCH using the striped Smith–Waterman implementation. The searches were run using the BLOSUM50 scoring matrix with a gap penalty of 10 − 2k.

The final comparison is against heuristic programs FASTA 34t26b4 (Pearson and Lipman, 1988) and NCBI BLAST 2.2.14 (Altschul *et al*., 1997). The executables used in testing were downloaded from their respective web sites, the University of Virginia and the NCBI. For a more meaningful comparison, SSEARCH was modified to use the striped Smith–Waterman implementation. All the searches were run using the BLOSUM50 scoring matrix and gap penalties of 10 − 2k. The options for all programs were to display the top 500 scores with no alignments. The search times for the 11 query sequences are shown in Figure 10. FASTA was run with both ktup = 1 and 2. On the whole, the striped Smith–Waterman was faster than FASTA, more than 50% faster when ktup = 2 and four time faster when ktup = 1. SSEARCH averaged about 30% slower runtimes when compared to BLAST.

## 4 DISCUSSION

Due to the number of iterations in the Smith–Waterman (Smith and Waterman, 1981) calculations, reducing the number of instructions in the inner loop had a significant effect on the execution time. By using pre-calculated weights, removing the SIMD register data dependencies and moving all branches out of the inner loop, the striped Smith–Waterman has a very efficient inner loop. This paper presents an efficient SIMD implementation of the dynamic programming algorithm that might be adapted to other biological problems.

The current implementation uses block substitution matrices as the scoring matrix. The implementation could easily be adapted to use other types of scoring functions, such as position-specific scoring matrices (PSSM) (Gribskov *et al*., 1987) and possibly profile hidden Markov model (profile HMM) (Eddy, 1999). The profiles need to be re-arranged in the same stripped-pattern as the query profiles in order to work with this implementation.

Another possible use for this algorithm, in addition to database searches, would be for aligning two sequences. Other software packages use a SIMD implementation to find high scoring matches and then use a scalar Smith–Waterman to align the two sequences.

This implementation could easily be modified to find the high score and the location of the scoring sequence. The location could then be used in the Hirschberg algorithm (Chao *et al*., 1994) to align the two sequences. This would allow faster alignments of larger sequences in linear space.

Dynamic programming is used for global alignment (Needleman and Wunsch, 1970) as well as local alignment. Other uses include assembling DNA sequence data from the fragments from automated sequencing machines (Anson and Myers, 1997), and to determine the intron/exon structure of eukaryotic genes (Gelfand and Royberg, 1993). It is also used to predict the secondary structure of functional RNA genes or regulatory elements (Zuker, 1989). All of these problems might benefit from an efficient SIMD implementation of the dynamic programming algorithm.

## ACKNOWLEDGEMENTS

I thank William Pearson for providing the source code for SSEARCH, which was used in the initial prototyping of the striped Smith–Waterman implementation.

*Conflict of Interest*: none declared.

## REFERENCES

Alpern,B., Carter,L. and Gatlin,K.S. (1995) Microparallelism and high performance protein matching. In: *Proceedings of the 1995 ACM/IEEE Supercomputing Conference,* 3–8 December 1995, San Diego, California. Available at: http://www-cse.ucsd.edu/users/carter/Micro/sc95.html (Accessed Dec 4, 2006).

Altschul,S.F. *et al*. (1997) Gapped BLAST and PSI-BLAST: a new generation of protein database search programs. *Nucleic Acids Res*., **25**, 3389–3402.

Anson,E.L. and Myers,G.W. (1997) Realigner: a program for refining DNA sequence multi-alignments. In: *Procedings of the 1st ACM Conference on Computational Molecular Biology,* ACM Press, New York, pp. 9–16.

Benson,D.A. *et al*. (2000) Genbank. *Nucleic Acids Res*., **28**, 15–18.

Chao,K.M., Hardison,R.C. and Miller,W. (1994) Recent developments in linear-space alignment methods: a survey. *J. Comput. Biol*., **4**, 271–291.

Eddy,S. (1999) Profile hidden Markov models. *Bioinformatics*, **14**, 755–763.

Gelfand,M.S. and Roytberg,M.A. (1993) A dynamic programming approach for predicting the exon–intron structure. *Biosystems*, **30**, 173–182.

Gotoh,O. (1982) An improved algorithm for matching biological sequences. *J. Mol. Biol*., **162**, 705–708.

Gribskov,M., McLachlan,A.D. and Eisenberg,D. (1987) Profile analysis: detection of distantly related proteins. *Proc. Natl Acad. Sci. USA*, **84**, 4355–4358.

Henikoff,S. and Henikoff,J.G. (1992) Amino acid substitution matrices from protein blocks. *Proc. Natl Acad. Sci. USA*, **89**, 10915–10919.

Hinton,G., Sager,D., Upton,M., Boggs,D., Carmean,D., Kyker,A. and Roussel,P. (2001) The Microarchitecture of the Pentium 4 Processor. *Intel Technology Journal Q1, 2001*.

Intel (2004), *IA-32 Intel Architecture Software Developer's Manual Volume 1: Basic Architecture*, Available at: http://www.intel.com/products/processor/manuals/index.htm (Accessed Dec 4, 2006).

Intel (2005), *IA-32 Intel Architecture Software Developer's Manual Volume 2B: Instruction Set Reference, N-Z*, Available at: http://www.intel.com/products/processor/manuals/index.htm (Accessed Dec 4, 2006).

Intel (2005), *IA-32 Intel Architecture Optimization Reference Manual*, Available at: http://www.intel.com/products/processor/manuals/index.htm (Accessed Dec 4, 2006).

Needleman,S.B. and Wunsch,C.D. (1970) A general method applicable to the search for similarities in the amino acid sequence of two proteins. *J. Mol. Biol*., **48**, 443–453.

Pearson,W.R. and Lipman,D.J. (1988) Improved tools for biological sequence comparison. *Proc. Natl Acad. Sci. USA*, **85**, 2444–2448.

Rognes,T. and Seeberg,E. (2000) Six-fold speed-up of Smith–Waterman sequence database searches using parallel processing on common microprocessors. *Bioinformatics*, **16**, 699–706.

Smith,T.F. and Waterman,M.S. (1981) Identification of common molecular subsequencees. *J. Mol. Biol*., **147**, 195–197.

Wozniak,A. (1997) Using video-oriented instructions to speed up sequence comparison. *Comput. Appl. Biosci*., **13**, 145–150.

Zuker,M. (1989) On finding all suboptimal foldings of an RNA molecule. *Science*, **244**, 48–52.