Stable SIMD Smith-Waterman for PAM Alignments

Luís Russo*[1] and Nuno Roma[1]

[1]TU Lisbon / IST / INESC-ID

Email: Luís Russo*- luis.russo@ist.utl.pt; Nuno Roma - nuno.roma@inesc-id.pt;

*Corresponding author

Abstract

In this paper we study an offset technique for Single-Instruction Multiple-Data (SIMD) implementations of the Smith-Waterman (SW) algorithm. This, dynamic programming, algorithm computes alignments between sequences, it is particularly useful for looking up a Deoxyribonucleic Acid (DNA) fragment in database. SW is guaranteed to find optimal local alignments. On the other hand its complexity is quadratic, which makes it slow, particularly for large databases. This problem has motivated, on the one hand, heuristic algorithms that miss out on potentially useful alignments, optimized exact approaches, such as using index structures or parallel computation, namely with SIMD operations. We present an optimization of state of the art SIMD implementations, by using an offset technique which guarantees that a large number of values are computed in parallel, even when the dynamic programming matrix stores large values. We implement out technique over the stripped layout of Farrar [1] and show experimental speed-ups, by a factor of 2, in hard cases. Point Accepted Mutation (PAM) matrices are one such relevant cases, where our speed-up holds.

Motivation:

Results:

Conclusions:

Availability:

Contact:

Background

Along the last decades, sequence alignment algorithms have been extensively applied in many bioinformatic applications, including sequence database searching, multiple sequence alignment, genome assembly and short read mapping. For such purpose, the Smith-Waterman (SW) algorithm [2], latter redefined and simplified by Gotoh [3], has been widely adopted whenever the best and optimum alignment is needed.

However, as the amount of gathered biological data increases, the performance of this algorithm has become a critical issue. As a consequence, some considerably faster heuristic approaches, such as FASTA [4] and BLAST [5], have been proposed. Nevertheless, the acceleration provided by these methods is obtained at the expense of sensitivity, and due to this loss of sensitivity some related sequences can not be detected in a search.

Hence, in order to accelerate the alignment procedure as much as possible, while still assuring the best alignment, several parallelization approaches of the SW algorithm have been presented. Three proposed Single-Instruction Multiple-Data (SIMD) approaches, based on the exploitation of MMX/SSE instruction set extensions, are particular relevant and were presented by Wozniak [6], Rognes and Seeberg [7, 8] and by M. Farrar [1]. Their main differences lie in the adopted data processing pattern. To assure the simultaneous commitment of as many dependencies as possible, the parallel scheme proposed by Wozniak [6] concurrently processes a set of cells along the minor diagonal of the alignment matrix. Some years later, Rognes and Seeberg [7] presented an alternative approach to simplify and accelerate the loading of the substitution scores from memory, by pre-computing a query profile for the entire database. With such technique, a vector of cells parallel to the query sequence can be simultaneously processed by each instruction. The commitment of the data dependencies are guaranteed by defining threshold conditions relating each computed score value and the insertion/extension gap penalties, allowing to disregard most comparisons related to the vertical dependencies of the algorithm. Nevertheless, such approach implied the introduction of conditional branches in the inner loop of the algorithm, making the execution time dependent on the scoring matrix and the gap penalties. Later, M. Farrar [1] improved this processing scheme by adopting a stripped access pattern. With such approach it was possible to move the conditional procedures related to the commitment of the vertical dependencies to a *lazy loop*, executed outside the inner loop. Hence, the conditional statements only have to be taken into account once, when processing the next database symbol. In the whole, such cumulative set of contributions and improvements led to significant speedup values of the alignment procedure, which justified its integration in many high performance alignment frameworks, such as in the most recent versions of SSEARCH [9]. Meanwhile, Rognes [8] presented another parallelization of the SW algorithm that exploits

other capabilities made available by modern processors. Besides using a SIMD model to simultaneously compare sixteen different database sequences with one single query sequence, it also combines a multi-threaded processing scheme to concurrently exploit the several cores available in the latest generations of SMP architectures.

Despite their processing efficiency, one important issue of these SIMD implementations of the SW algorithm is concerned with the range of the score values that can be processed with SIMD vectors. As an example, the maximum number of simultaneous operands that Intel's SSE2 extension supports on its 128-bit registers is 16, each of which with 8-bit resolution. This restricts the evaluated score values to a maximum amplitude of 255, making it difficult to compute entries that require a wider range of values (*e.g.*: when the sequences are long or very similar) without entering into an overflow condition (see See Fig. 1). The solutions that have been adopted to solve this problem are somewhat optimistic or rather inefficient. Mostly, simply restart the whole alignment algorithm, either by using a wider dynamic range [1, 7] or by adopting a different (and less efficient) implementation of the alignment procedure, thus significantly compromising the computation speed.

To circumvent this problem, a new and more efficient approach is now proposed that allows a significant improvement of the precision of the evaluated score values. The presented method is entirely compliant with

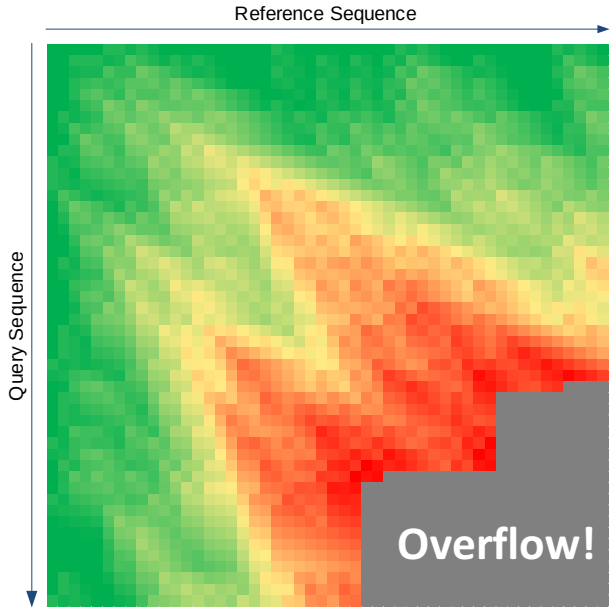


Figure 1: Early termination of the score matrix computation due to arithmetic overflow.

Farrar's implementation and combines the computation of the 8-bit vector data elements with an adaptive offset-grid that releases the dynamic range constraint of the computed score values.

The presented experimental results demonstrate that not only does this technique allow an efficient and precise computation of the alignment scores, but its integration in Farrar's implementation is possible with minimum overhead, leading to equivalent performance values.

Methods

Smith-Waterman Algorithm

The adopted formulation of the SW algorithm is similar to the one presented by M. Farrar [1], as shown in eq. 1.

$$H(i,j)=\begin{cases} \max\left\{\begin{array}{c} H(i-1,j-1)+W(q[i],d[j]) \\ E(i,j) \\ F(i,j) \\ b \end{array}\right\} & \left|\begin{array}{l} i>0 \\ \wedge \\ j>0 \end{array}\right. \\ b & \left|\begin{array}{l} i=0 \\ \vee \\ j=0 \end{array}\right. \end{cases} \tag{1}$$

$$E(i,j+1)=\max\left\{\begin{array}{l} E(i,j) \\ H(i,j)-GI \end{array}\right\}-GE \tag{2}$$

$$F(i+1,j)=\max\left\{\begin{array}{l} F(i,j) \\ H(i,j)-GI \end{array}\right\}-GE \tag{3}$$

$$A=\max_{0\le i\le m, 0\le j\le n} H(i,j)-b \tag{4}$$

The query sequence q, of length m, is defined as $q = q[1], q[2] \dots q[m]$, while the database sequence d, of length n, is defined as $d = d[1], d[2] \dots d[n]$. $H(i,j)$ is the score for aligning the prefixes of q and d ending in the alignment of symbols $q[i]$ and $d[j]$. $E(i,j)$ and $F(i,j)$ are the scores for aligning the same prefixes of q and d but ending with a gap in the query and database sequences, respectively. $W(q[i],d[j])$ is the score for aligning the symbols $q[i]$ and $d[j]$ with each other, according to the substitution score matrix W. Usually, $W(q[i],d[j])$ is negative if $q[i] \neq d[j]$ and positive otherwise. GI represents the penalty for starting a gap, while GE represents the penalty for extending a gap that was already opened. Despite their

equivalence, it is worth noting the slight difference in the definition of these constants, when compared with Farrar's formulation: GI is equivalent to Farrar's $G_{init} - G_{ext}$. It is also assumed that both GI and GE are non-negative, *i.e.,* $GI \geq 0$ and $GE \geq 0$. Variable A denotes the overall optimal local alignment score.

Just like in [1,10], negative values in the intermediate computations are avoided by adding a bias constant b in eq. 1. This bias value must be such that $b \geq GE + GI \geq 0$ and $b \geq |W(q[i], d[j])|$ for any $q[i] \neq d[j]$.

The previous equations are written coherently with the pseudo-C code shown later in Fig. 3. To keep the formulation consistent, it was defined $E(i, j+1)$ and $F(i+1, j)$, instead of $E(i, j)$ and $F(i, j)$, respectively. Accordingly, $E(i, 0)$ and $F(0, j)$ are set to b, $\forall_{i,j}$. Likewise, $H(0, 0)$ is also initialized with b.

It is worth noting that this definition omits the bias value from E and F vectors. This contrasts with a recent SIMD implementation proposed by Farrar [10] for the Cell Broadband Engine. Such biasing was mainly motivated by the fact that the Cell Broadband Engine does not have saturated math instructions (which underflow to zero), whereas SSE instructions on Intel/AMD processors do. In fact, not only can it be shown that the formulation that is now presented does not imply any strict dependency on saturated math operations, but it can also be demonstrated, from the above recurrences and for any valid i and j values, that: *i)* $H(i, j) \geq b$; *ii)* $E(i, j+1) \geq 0$; and *iii)* $F(i+1, j) \geq 0$. The first property follows from the fact that b appears as an operand of the max function in the definition of $H(i, j)$. The demonstration of the second property makes use of the first property, the assumption that $b \geq GE + GI$ and the fact that $H(i, j)$ appears as an operand of Eq. 2. A similar argument holds for the third property. Hence, by taking these assumptions into account, it can also be demonstrated that $E(i, j+1)$ and $F(i+1, j)$ cannot have any negative intermediate values. Consequently, the above referred saturated arithmetic restriction does not apply to the proposed formulation, since there can not be any underflow. In fact, the only expression where negative values can occur is in $W(q[i], d[j])$ which, as previously stated, is negative when $q[i] \neq d[j]$. To avoid such condition, the first term in Eq. 1 was re-written as $H(i-1, j-1) + W^b(q[i], d[j]) - b$, where W^b is a positive pre-processed score matrix, defined as $W^b(i, j) = W(i, j) + b$. During the execution of the algorithm, the bias constant b is subtracted to the final value. With such formulation, it is assured that the end result will be always positive.

Farrar's Striped Technique

As it was referred before, the main innovative contribution that makes Farrar's algorithm one of the fastest SIMD implementations of the SW algorithm is its *striped* processing pattern. With such layout, the precomputed query profile is accessed by following a pattern parallel to the query sequence and the computations

are carried out in several separate stripes, that cover different parts of the query sequence. This approach significantly reduces the impact of some of the data dependencies, moving them out of the inner loop and placing them in an outer *lazy* loop, where they have to be considered only once, before starting the processing of the next database symbol (see Fig. 2).

By following this approach, the query is divided into p equal length segments, where p is given by the number of data elements that can be simultaneously accommodated in a SIMD register. As an example, when 128-bit SSE2 registers are considered to process 8-bit data elements, p equals to 16. One the other hand, when greater resolution scores corresponding to 16-bit integers are used, p must be reduced to 8. The length of each segment is given by $\lfloor t = (m+p-1)/p \rfloor$, where padding zeros are inserted whenever the query is not long enough to completely fill all the segments.

The computation of the score matrix is fulfilled by assigned each data element of the SIMD register to one distinct segment, leading to the stripped access pattern illustrated in Fig. 2. Accordingly, each matrix column, corresponding to a database symbol $d[j]$ is processed in t iterations, where each iteration simultaneously processes p query symbols, separated by $t-1$ lines in the score matrix. As an example, when

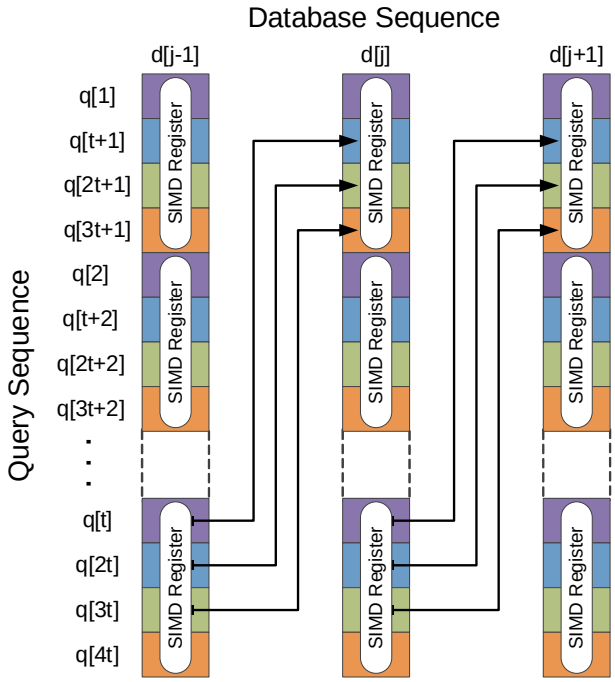


Figure 2: Data dependencies between the last segment and the first segment of the next column of the score matrix H. For simplicity, only 4 data elements are shown in each SIMD register (represented in violet, blue, green and orange), while sixteen 8-bit elements would normally be used.

$p = 16$ the second iteration of the algorithm simultaneously processes, in a SIMD register, the following query symbols:

$$q_2 = \left[\begin{array}{ccccccc} q[2] & q[t+2] & q[2t+2] & q[3t+2] & q[4t+2] & \cdots & q[15t+2] \end{array} \right] \tag{5}$$

The processing of such query symbols considers a precomputed query profile that adopts a similar pattern:

$$W_{2,j} = \left[\begin{array}{ccccccc} W[q[2],d[j]] & W[q[2t+2],d[j]] & W[q[3t+2],d[j]] & W[q[4t+2],d[j]] & \cdots & W[q[15t+2],d[j]] \end{array} \right] \tag{6}$$

The pseudo-C code of Farrar's algorithm is illustrated in Fig. 3. Variables that start with `v` refer to vectors of elements, which are stored in SIMD registers. All operations involving these variables are simultaneously applied to all elements. For example, adding a constant to a vector means adding to all data elements of that vector. In those rare cases where it is necessary to explicitly refer to a specific element inside the vector it is used the `@[]` operator. Similarly, the `@+` operator is used to represent the shift operation, where all elements in a vector are displaced one position to the left, *i.e.,* it uses a SIMD instruction that is equivalent to a cycle that sets $v[0] = 0$ and $v[k+1] = v[k]$. These operators appear in lines 8 and 27 of the algorithm in Fig. 3, respectively. It is also worth recalling that in C the `%` operator (line 33) returns the remainder of the integer division. Another operation over SIMD vectors that is worth mentioning is represented in line 28 and stated as `vF <= vF[i]`, accompanied by the expression `FOR ALL ELEMENTS`, to denote that such expression is considered true only if it holds for all the elements in the SIMD vector.

In general, variables in this pseudo-C code match with their representation in Farrar's manuscript. Nevertheless, there are few exceptions where this pseudo-code differs from the one presented by Farrar. As an example, the gap penalties GE and GI are used just like in Eqs. 1, 2 and 3. The outer j-loop processes one symbol of the database at a time, where the `vH[]` variable represents an array of SIMD vectors. This external `j-loop` is indexed by j to keep coherence with the same equations. For simplicity, the computation of A in eq. 4, *i.e.,* the overall maximum of H, is not shown in the algorithm. In the same trend, one of Farrar's variable was removed and variable `vH` is used in a different way.

From one iteration of the i-loop to the next, a copy of the $H(i,j)$ values that have just been computed has to be stored using another array of SIMD vectors `vHs[]`. At this respect, it is important to recall that although the arrays usually start at index 0, for this particular array it was assumed that there is also a -1 index position.

Finally, the `while(1)` instruction (line 25) represents an "infinite" version of the lazy F-loop, which finishes with an `if/break` statement. This avoids repeating code, which is essential for the presentation,

```
1  ...                                        /* Set initial values */
2  t = (m + 15)/ 16;                  /* 16 element SIMD register */
3  j = 0;  /*------------------------------------------ j-loop */
4  while(j < n)                     /* Process a database letter */
5    {
6      copy(vH, vHs);                      /* copy all vH to vHs */
7      vHs[-1] = vH[t-1] @+ 1;            /* Shift elements down */
8      vHs@[0] = b;                    /* Set first element to b */
9      vF[0] = {0, ... , 0};                          /* Initial F */
10     i = 0; /*------------------------------------- i-loop */
11     while(i < t)                    /* Process a query vector */
12       {
13         vH[i] = (vHs[i-1]+pWb(i, d[j])) - b;  /* H in (1) */
14         vH[i] = max(vH[i], vE[i]);             /* E in (1) */
15         vH[i] = max(vH[i], vF[i]);             /* F in (1) */
16         vH[i] = max(vH[i], b);                 /* b in (1) */
17         vE[i] = max(vE[i], vH[i] - GI) - GE;     /* Eq (2) */
18         vF[i+1] = max(vF[i], vH[i] - GI) - GE;  /* Eq (3) */
19         i++;
20       }
21
22     /*---------------------------------------- Lazy F-loop */
23     vF = vF[t];
24     i = 0;
25     while(1)
26       {
27         if(i==0) vF = max(vF[i], vF@+1); /* Cross segment */
28         if(vF == vF[i], FOR ALL ELEMENTS) break;  /* STOP */
29         vF[i] = vF;                    /* Fix wrong F values */
30         vH[i] = max(vH[i], vF[i]);             /* F in (1) */
31         vE[i] = max(vE[i], vH[i] - GI - GE);  /* H in (2) */
32         vF    = max(vF[i], vH[i] - GI) - GE;  /* H in (3) */
33         i = (i+1) % t;
34       }
35     j++;
36   }
```

Figure 3: Pseudo-C representation of the striped Smith-Waterman SIMD implementation proposed by Farrar [1].

whereas a real implementation can repeat code and perform loop unrolling. The update of i, inside the lazy F-loop, was changed with equivalent code. An extra instruction to update E in the lazy F-loop was also inserted, which was present in Farrar's prototype [1], but omitted in the original paper.

Offsets Technique

In this section we present the offset technique. The motivation for this technique follows from the observation that the variation of $H(i, j)$ values among adjacent cells is small. We seek to use this observation to perform the computation of the $H(i, j)$ over the less significant bits, therefore guaranteeing that it can essentially be computed with 8 bits per cell. This makes the overall algorithm faster by a factor of 2, since representing cell values with 16 bits is twice as slow.

From the fact that Equations (1), (2) and (3) use maximums we can easily conclude that the values of

adjacent cell decrease slowly.

$$\max\{E(i,j), F(i,j)\} \leq H(i,j) \tag{7}$$

$$H(i,j) \leq \min\{E(i,j+1), F(i+1,j)\} + GI + GE \tag{8}$$

$$H(i,j) \leq \min\{H(i,j+1), H(i+1,j)\} + GI + GE \tag{9}$$

$$H(i,j) \leq H(i+1,j+1) + \min\{2(GI+GE), -w^-\} \tag{10}$$

Where w^- is the minimum $W(q,d)$ value. Eq. (7) follows from Eq. (1), by discarding the other elements. Eq. (8) follows from Equations (2) and (3), by discarding $E(i,j)$ and $F(i,j)$. For Eq. (9), just use Eq. (7) to replace the values in Eq. (8). To obtain Eq. (10) use Eq. (9) twice, once with $H(i,j)$ and choose $H(i,j+1)$ and the second time with this value and choose $H(i+1,j+1)$, this yields $H(i,j) \leq H(i+1,j+1)+2(GI+GE)$. The other inequality follows form Eq. (1).

On the other hand alignment scores, of adjacent cells, cannot increase arbitrarily.

$$\max\{E(i,j+1), F(i+1,j)\} \leq H(i,j) - GE \tag{11}$$

$$H(i+1,j+1) \leq \min\{H(i,j), H(i,j+1), H(i+1,j)\} + w^+ \tag{12}$$

Eq. (11) uses a similar argument for $E(i,j+1)$ and $F(i+1,j)$. The argument for $E(i,j+1)$ consists in observing that both values of the max, in Eq. (2), are bounded by $H(i,j)$. Note that $E(i,j) \leq H(i,j)$ follows from Eq. (7).

In Eq. (12) w^+ is the maximum $W(q,d)$ value. This is a global property and is harder to prove by analyzing the equations. Notice that the values of $H(i,j)$ increase only with the $W(q,d)$, in Eq. (1). The remaining equations either subtract values or pass them to other cells. From $H(i,j)$ to $H(i+1,j)$ the only letter that can alter the overall score is $q[i+1]$, therefore the score cannot increase by more than $W(q[i+1],d)$, for some letter d in the database.

Now that we established bounds for the variation of values amongst adjacent cells let us focus on the offset representation of the table values. We store $H(i,j)$ as $2^{r-k}Ho(i,j) + Hl(i,j)$, where r is the number of bits in the representation of Hl and k is the number of bits that overlap with Ho. The overlapping is used precisely to avoid overflows and underflows. We plan to use $k = 8$, *i.e.,* each Hl value is represented by a byte. This guarantees the highest number of values per SIMD register. For the same reason, and moreover avoiding to convert among different value sizes, we also use 8 bits to represent the Ho values. A typical value for the overlap is $k = 3$. Therefore the total resolution for $H(i,j)$ is $2r - k = 13$ bits. Therefore we can represent alignment values between 0 and 8191. Such an alignment score is extremely high, in the

shortest sequence to obtain this value with the PAM250 matrix consists of 481 matching W's. Hence this representation is enough for most "useful" alignment values.

Likewise we represent $E(i,j+1) = 2^{r-k}Ho(i,j)+El(i,j+1)$ and $F(i+1,j) = 2^{r-k}Ho(i,j)+Fl(i+1,j)$. Notice that we avoid having different offset vectors Eo, Fo and having to convert amongst them. Computing an expression as $2^{r-k}Ho(i,j) + Hl(i,j)$, in a byte, will, most likely, overflow. Hence we define the relations $IH(i,j) = 2^{r-k}(Ho(i-1,j) - Ho(i,j))$ and $JH(i,j) = 2^{r-k}(Ho(i,j-1) - Ho(i,j))$, which we expect to be small. To avoid overflows these relations must be representable in r bits. Equations (9) and (12) show that this is a reasonable assumption, since most values of GE, GI and w^+ are much less than $2^7 = 128$. We also show experimentally that this is a reasonable assumption.

Replacing and simplifying these relations into the Gotoh recurrences we obtain the following:

$$H(i,j) = 2^{r-k}Ho(i,j)+ \max\left\{\begin{array}{l} (JH(i-1,j)+IH(i,j)) + Hl(i-1,j-1) + W(q[i],d[j]) \\ JH(i,j) + El(i,j) \\ IH(i,j) + Fl(i,j) \\ b - 2^{r-k}Ho(i,j) \end{array}\right\} \tag{13}$$

$$E(i,j+1) = 2^{r-k}Ho(i,j) + \max\left\{\begin{array}{l} JH(i,j) + El(i,j) \\ Hl(i,j) - GI \end{array}\right\} - GE \tag{14}$$

$$F(i+1,j) = 2^{r-k}Ho(i,j) + \max\left\{\begin{array}{l} IH(i,j) + Fl(i,j) \\ Hl(i,j) - GI \end{array}\right\} - GE \tag{15}$$

Notice how, besides the initial $2^{r-k}Ho(i,j)$ term, the remaining expressions are not susceptible cause overflow, or underflow, because they involve only lower order terms, Hl, El and Fl and the variation terms $IH(i,j)$ and $JH(i,j)$. The only exception is the expression $b - 2^{r-k}Ho(i,j)$ in Eq. (13). For this expression we assume that $b < 2^{r-k}$, therefore if $Ho(i,j) > 0$, the expression returns a negative value, in which case it will not be the maximum and can be omitted from the expression. Hence to avoid underflows we include b in the expression only when the respective $Ho(i,j)$ values are 0.

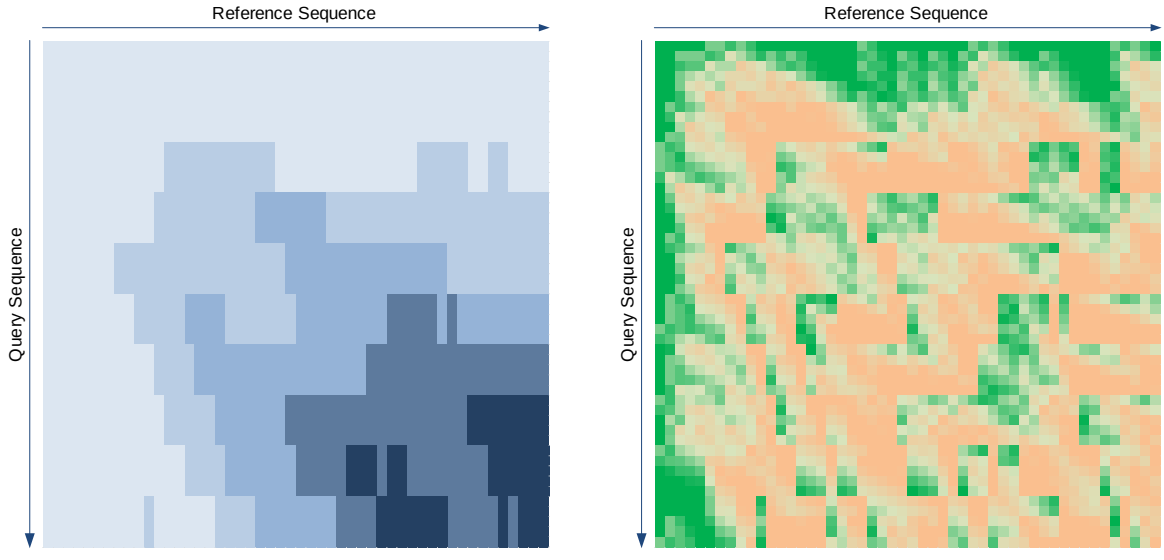
Although the above equations seem complex and, at first, it may they will imply a large overhead on the resulting algorithm, that is not, necessarily, the case. We will adopt a solution that uses the $IH(i,j)$ and $JH(i,j)$ values with parsimony. Most of the time both $IH(i,j)$ and $JH(i,j)$ are zero. Hence simplifying the equations and moving the overhead away from the critical path in the computation.

The simplest variations are the $IH(i,j)$ expressions, which affect only 3 terms. We adopt the stripped layout of Farrar, which is explained in Section , this means that $Ho(i,j)$ is constant within segments of q consecutive i values. Therefore $IH(i,j) = 0$, if i is not of the form $i'q$ for some non-negative integer i'. With this assumption adding offsets results in a small overhead to the algorithm.

For the second term we adopt a more flexible strategy. We scan the consecutive $H(i,j)$, $E(i,j)$ and $F(i,j)$ values and decide if it is necessary to reoffset. Only if we need to reoffset will the Hj value be non-zero. This simplifies the equations associated with El, and the last equation of $H(i,j)$. The change to the $H(i,j)$ and $E(i,j)$ is again performed outside the critical path of the algorithm. This process does not need to be computed every time, it can be computed periodically every other p, *i.e.,* reoffset is only computed when p is a factor of j. In a periodical verification we first check if the reoffset is necessary by determining the overall minimum s and maximum S among the $H(i,j)$, $E(i,j+1)$ and $F(i+1,j)$ values. The goal is to keep $((m+M)/2)+b$ as close to $2^{r-1}=128$ as possible, but guaranteeing representable space for the bias value.

Implementation

Now we describe how to modify the algorithm in Fig. 3 to include the offset technique. The resulting algorithm is shown in Fig. 5. We use new variables `vHl`, `vHls`, `vEl`, `vFl` which represent the lower order terms. We also use new variables `vIH` and `vHo`, that represent the $IH(i,j)$ variations and the higher order bits. Asides from renaming variables, we highlight the main differences between the algorithms, by using the symbol `+` in the comments. This occurs only in lines 6, 8, 9, 16 and 25 the ReOffset-loop, that is presented in Fig. 6. Notice that lines 6, 8 and 9 are not inside the i-loop, which is clearly the critical path in the algorithm, in most cases. Line 16 is inside the loop. Still it is simply replacing a, similar instruction, that



(a) Offsets plane $Ho(i,j)$.

(b) Incremental score variation plane $Hl(i,j)$.

Figure 4: Proposed offsets technique to avoid overflow conditions in the implementation of a SIMD score computation.

```
...                                        /* Set initial values */
t = (m + 15)/ 16;                    /* 16 element SIMD register */
j = 0;  /*---------------------------------------- j-loop */
while(j < n)                        /* Process a database letter */
  {
    copy(vHl, vHls);                     /* copy all vHl to vHls */
    vHls[-1] = (vHl[t-1]@+1) + vIH;  /*+++++++ Shift + vIH */
    vH@[0] = b;                           /* Set first element to b */
    vB = b IF (vHo == 0) AND 0 OTHERWISE; /*++++++ vB Mask */
    vFl[0] = {0, ... , 0};                        /* Initial F */
    i = 0; /*----------------------------------- i-loop */
    while(i < t)                        /* Process a query vector */
      {
        vHl[i] = (vHls[i-1]+pWb(i, d[j]))-b; /* H in (11) */
        vHl[i] = max(vHl[i], vEl[i]);        /* E in (11) */
        vHl[i] = max(vHl[i], vFl[i]);        /* F in (11) */
        vHl[i] = max(vHl[i], vB); /*++++++++++++ b in (11) */
        vEl[i] = max(vEl[i], vHl[i]-GI)-GE;    /* Eq (12) */
        vFl[i+1] = max(vFl[i], vHl[i]-GI)-GE;  /* Eq (13) */
        i++;
      }

    /*---------------------------------------- Lazy F-loop */
    vFl = vFl[t];
    i = 0;
    while(1)
      {
        if(i==0)
          vFl=max(vFl[i], (vFl@+1)+vIH); /*++ Cross + vIH */
        if(vFl == vFl[i], FOR ALL ELEMENTS) break;/* STOP */
        vFl[i] = vFl;                        /* Fix wrong F values */
        vHl[i] = max(vHl[i], vFl[i]);         /* F in (1) */
        vEl[i] = max(vEl[i], vHl[i]-GI) - GE; /* H in (2) */
        vFl    = max(vFl[i], vHl[i]-GI) - GE; /* H in (3) */
        i = (i+1) % t;
      }

    ... /*++++++++++++++++++++++++++++++++++++ ReOffset */
    j++;
  }
```

Figure 5: Offset Striped Smith-Waterman in pseudo-C code.

```
1  ... /*------------------------------------ ReOffset */
2  if((j%p) == 0)    /* change offsets according to rate */
3    {
4      vm = { 255, ... , 255 };        /* Initial minimum */
5      vM = { 0, ... , 0 };            /* Initial maximum */
6      i = 0;
7      while(i < cm)
8        {                                    /* check cells */
9          vm = min(vm, vEl[i]);             /* update min */
10         vM = max(vM, vEl[i]);             /* update max */
11         i++;
12       }
13     if(vm == 0)   REPORT UNDERFLOW;
14     if(vM == 255) REPORT OVERFLOW;
15
16     vJH = 128 - vm/2 - vM/2 - b; /* center around 128 */
17     vJH = vJH/32;                    /* Get higher bits */
18     vJH = vHo - (vHo - vJH);       /* Saturate Up/Down */
19     vHo -= vJH;                           /* update vHo */
20     vJH = vJH*32;                     /* Get lower bits */
21     vIH += (vJH@+1) - vJH;     /* update vIH with vJH */
22
23     i = 0;  /*+++++++++++++++ Update according to vJH */
24     while(i < cm)
25       {
26         vEl[i] += vJH;
27         vHl[i] += vJH;
28         i++;
29       }
30   }
```

Figure 6: Offset Striped Smith-Waterman in pseudo-C code.

already existed. It is highlighted only to indicate that it defers form the original, but its computation cost is the same. The difference is that the values of vB are not all b, instead this vector contains a b contains a b for the elements that have $Ho(i, j) = 0$ and 0 otherwise. This moves the overhead outside the i-loop. Line 24 occurs in the lazy F-loop, which is not so critical, and moreover inside an `if` condition that is most oftenly false. Hence these operations present a small overhead. Let us now focus on the Reoffset procedure.

Figure 6 shows the pseudo-C code for the Reoffset procedure. The loop in the procedure is used to compute a vector of minimums and maximums, which are then check for underflow or overflow in lines 15 and 16. If these events occur the computation aborts and it must indeed resort to 16-bit computation. The remaining lines are used to update the offset values. Once again line 18 is only conceptual, the actual computation is a bitwise "and" operation with the appropriated mask.

Results and Discussion

Conclusions

The presented experimental results demonstrated that not only does the proposed technique allow an efficient and precise computation of the alignment scores, but its integration in Farrar's implementation is possible with minimum overhead, leading to equivalent performance values. As such, it can easily contribute to the improvement of other widely used alignment frameworks that already embedded Farrar's implementation,

such as SSEARCH.

Acknowledgements

This work was supported by the Portuguese Foundation for Science and Technology (FCT) under the projects "TAGS: The power of the short - Tools and Algorithms for next Generation Sequencing applications", with reference PTDC/EIA-EIA/112283/2009, "HELIX: Heterogeneous Multi-Core Architecture for Biological Sequence Analysis", with reference PTDC/EEA-ELC/113999/2009, and the PIDDAC Program funds (INESC-ID multiannual funding).

References

1. Farrar M: **Striped Smith–Waterman speeds database searches six times over other SIMD implementations**. *Bioinformatics* 2007, **23**(2):156.
2. Smith T, Waterman M: **Identification of common molecular subsequences**. *Journal of molecular biology* 1981, **147**:195–197.
3. Gotoh O: **An improved algorithm for matching biological sequences**. *Journal of molecular biology* 1982, **162**(3):705–708.
4. Pearson W, Lipman D: **Improved tools for biological sequence comparison**. *Proceedings of the National Academy of Sciences* 1988, **85**(8):2444.
5. Altschul S, Gish W, Miller W, Myers E, Lipman D: **Basic local alignment search tool (BLAST)**. *Journal of Molecular Biology* 1990, **215**:403–410.
6. Wozniak A: **Using video-oriented instructions to speed up sequence comparison**. *Computer applications in the biosciences : CABIOS* 1997, **13**(2):145–150, [http://bioinformatics.oxfordjournals.org/content/13/2/145.abstract].
7. Rognes T, Seeberg E: **Six-fold speed-up of Smith–Waterman sequence database searches using parallel processing on common microprocessors**. *Bioinformatics* 2000, **16**(8):699.
8. Rognes T: **Faster Smith-Waterman database searches with inter-sequence SIMD parallelisation**. *Bioinformatics* 2011, **12**(221):11.
9. Pearson WR: **Searching protein sequence libraries: comparison of the sensitivity and selectivity of the Smith-Waterman and FASTA algorithms**. *Genomics* 1991, **11**(3):635–650.
10. Farrar M: **Optimizing smith-waterman for the cell broadband engine**. *Available http://farrar. michael. googlepages. com/SW-CellBE. pdf.*

Figures

Figure 1 - Sample figure title

A short description of the figure content should go here.

Figure 2 - Sample figure title

Figure legend text.

Tables

Table 1 - Sample table title

Here is an example of a *small* table in LaTeX using `\tabular{...}`. This is where the description of the table should go.

My Table		
A1	B2	C3
A2	...	..
A3	..	.

Additional Files

Additional file 1 — Sample additional file title

Additional file descriptions text (including details of how to view the file, if it is in a non-standard format or the file extension). This might refer to a multi-page table or a figure.

Additional file 2 — Sample additional file title

Additional file descriptions text.