Improving SIMD Implementations of the Smith-Waterman Algorithm

Abstract. The Smith-Waterman (SW) algorithm is a key component of several bioinformatic applications, more precisely, those related to sequence alignment. Despite its popularity, the quadratic running time of the SW algorithm severally limits its application. Therefore several techniques have been employed to accelerate the execution of this algorithm. This article focus on speeding-up the SW algorithm with SIMD instructions. We propose, two techniques to improve the performance of SIMD implementations: parallel vertical dependency propagation and offset representation. Experimental results obtained from the SIMD prototype show that these techniques provide a speedup that is always greater than 2, when compared to a plain SIMD implementation. Hence we prove the suitability and efficiency of the new techniques.

1 Introduction

As the amount of biological data gathered from sequenced genomes increases, the performance of alignment algorithms becomes an even more critical issue. The Smith-Waterman (SW) algorithm, that computes the optimal local alignment between a query and a database of genome sequences, has significant computational requirements. This handicap partially justifies the popularity of several probabilistic sub-optimal algorithms, such as BLAST [1]. On the other hand, in the last few years, there were several proposals to speed-up the SW algorithm, namely using on the Single Instruction Multiple Data (SIMD) techniques, supported by the instruction set extensions currently available on many processors (e.g.: MMX/SSE2 from Intel, 3DNow! from AMD, etc.). These operations allows us to simultaneously process several data elements in parallel, i.e., supporting fast computation of vector-wise operations [2–6].
A determinant design decision that SIMD implementations of the SW algorithm is the orientation the vectors that are processed in parallel. Namely because the underlying instruction set must have enough operations to support the computation of the vertical, horizontal and diagonal dependencies of the algorithm. Moreover this decision usually has a significant impact on the speed of the resulting algorithm. As a result, sequential execution patterns are often required to comply with such data dependencies (e.g. commitment of vertical dependencies [4]). In this paper we use vertically oriented vectors, which means that the vertical dependencies cannot be computed in parallel and therefore become the bottleneck of the algorithm. In this paper we present a new loop optimization technique that significantly reduces the number of configurations where this slow-down can occur.

Another important issue of SW SIMD implementations is the range of the score values that can be processed with SIMD vectors. For example, the maximum number of simultaneous computations that Intel's SSE2 extension supports is 16 element data vectors, each of which with 8-bit resolution. This restricts the evaluated score values to a maximum amplitude of 256, making it difficult to compute entries that require a wider range of values. Previous solutions for this problem were naive, in the sense that the algorithm needed to restart the computation with wider dynamic ranges [6], thus compromising the computation speed to improve the precision of the evaluated scores. We propose a new solution for this problem that uses 8-bit vector elements without sacrificing the dynamic range of the computed score values.
We show experimentally that applying these two techniques, simultaneously, improves the straightforward SW algorithm by a factor of around 4.5 times and the plain SIMD implementation by a factor of around 2. Moreover, we expect, that these factors hold for alternative SIMD implementations of the SW algorithm, for example by Rognes and Seeberg [4].

1.1 Smith-Waterman (SW) algorithm

The Smith-Waterman (SW) algorithm is a Dynamic Programming (DP) algorithm that computes best local alignment of two strings $P = p_1 p_2 \dots p_n$ and $T = t_1 t_2 \dots t_m$, with sizes n and m, respectively. Its time complexity is $\mathcal{O}(nm)$. The algorithm starts by building a DP matrix $\mathbf{H}$, of size $(n+1) \times (m+1)$, by setting $H_{i0} = H_{0j} = 0$ for $0 \leq i \leq n$ and $0 \leq j \leq m$. The remaining values of the matrix are recursively obtained with the following equation, for $1 \leq i \leq n$ and $1 \leq j \leq m$:

$$H_{ij} = \max \begin{cases} H_{(i-1)(j-1)} + Sbt(p_i, t_j) \\ H_{(i-1)j} - gapcost \\ H_{i(j-1)} - gapcost \\ 0 \end{cases} \quad (1)$$

where $Sbt(x, y)$ is the symbol substitution cost function. Typically, for genome alignments, $Sbt(x, y)$ has a positive value when $x = y$ and a negative value when $x \neq y$. The *gapcost* value represents the cost for opening or extending a gap in any of the two sequences (linear gap penalty).
After filling the entire $\mathbf{H}$ matrix, the cell with the highest value represents the local alignment score between the two sequences, its location in matrix $\mathbf{H}$ represents the indexes of the last symbol of the subsequences that align. The alignment can then be obtained by executing a traceback procedure from that location upwards.

1.2 Parallel alignment algorithms

For quadratic time algorithms, such as SW, the computation time increases rapidly, as the size of the inputs grows. Therefore many techniques were proposed to parallelize these algorithms. Usually, these techniques can be classified into three levels of parallelism:

- *Coarse-grained parallelism* – often implemented by dividing the set of query sequences in several parts, which are subsequently distributed between the several nodes within a computer cluster. However, the communication time between the several nodes, for transferring the input and resolving dependencies, can be a huge bottleneck and may actually make the algorithm slower.
- *Middle-grained parallelism* – consists in slicing the sequences pair under processing into chunks, which are distributed between the several processors in a shared memory multi-processor system [7]. The downside of this technique is that the dependencies between many intermediate values within the DP algorithm make it difficult to scale, as more processors are added.
- *Fine-grained parallelism* – consists in applying SIMD techniques implemented within the processor arithmetic unit, by accommodating character sets into vectors and performing several operations in parallel. Though this set of techniques can be used on both CPUs and GPUs, the latter often impose a non-negligible bottleneck to transfer data between the CPU and the GPU.

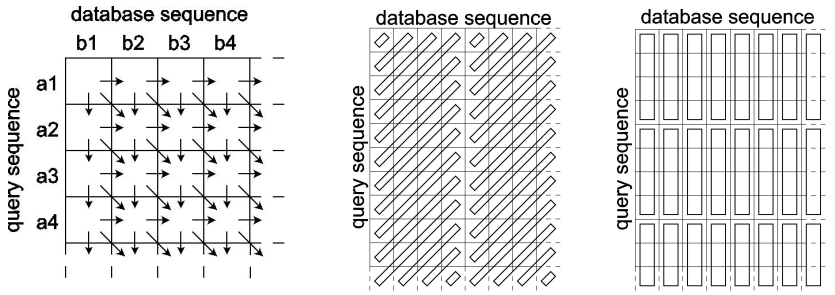
Among these three parallelization approaches, the *fine-grained* parallelization techniques are particularly popular. The application of SIMD instructions may be used to speed-up algorithms that can be abstracted in terms of vector operations, by accommodating several data elements in the processor registers and by simultaneously processing them at once. Intel's SSE2 extension, for example, makes use of 128-bit registers and supports operations either on 2-element vectors (64-bit each), 4-element vectors (32-bit each), 8-element vectors (16-bit each) and 16-element vectors (8-bit each). Nevertheless, since these technologies were developed mostly for multimedia applications, not all operations of the instruction set are available for some types of vectors (e.g.: vector-wise *maximum* of signed 8-bit vectors and vector-wise *maximum* of unsigned 16-bit vectors).

2 Related work

The determining design decision of SIMD SW algorithms is the orientation of the vectors of cells, see an illustration of the computational dependencies in Fig. 1(a). Several orientation were studied in previous literature: along the matrix's minor diagonal, see Fig. 1(b), first considered by Wozniak [3] and Hughey [2]; vertically, see Fig. 1(c), proposed by Rognes and Seeberg [4] and Farrar [6].

Using minor diagonal vectors is particularly efficient during the processing phase, since the SW algorithm has no dependencies along a minor diagonal. However it requires a pre-processing stage that can compromise the overall execution time. Namely because it requires $O(n\sigma^k)$ time and space, where m is the database size, n is the query size, k is the vector size and σ the size of the underlying alphabet. Naturally this expression may be less than $O(nm)$, the cases larger than this bound can be reduced. However this pre-processing step severely limits the vector size, since for $k = 16$ we have that $O(n\sigma^k)$ is $o(nm)$ only for more than 4

Gigabytes of DNA. Hence choosing this vector orientation is fast because it reduces the time to compute dependencies but it is also slow because it limits the size of the vectors, which in turn affect the time performance.



(a) Matrix dependencies. (b) Diagonal representation. (c) Vertical representation.

Fig. 1. Dependencies and possible vector representations of the DP matrix of the SW algorithm.

On the other hand vertically oriented approaches, must deal with the vertical dependencies contained inside a single vector, making the scan process slower, but they support large vector sizes and have small memory requirements. In fact using vertically oriented vectors we can maintain only two DP at a time, making the overall space requirements $O(n\sigma)$, mainly for storing a *query specific score matrix* table that we explain in Section 3.1.

Both approaches use 8-bit unsigned integer vectors to represent the scores, which provides a resolution in the range $[0, 255]$. In case the 8-bit precision is not enough it is necessary to restart the algorithm with 16-bit signed integer vectors that supports a resolution range of $[0, 32767]$. Notice from Equation (1) that the SW cell values have a minimum value of 0. Notice that this technique is very inefficient, not only due to the repeated computation, but also because of the 2 slowdown that occurs when changing from $k = 16$ element vectors to $k = 8$ element vectors. The reason for using 16-bit signed integer vectors instead of unsigned is that Intel's MMX/SSE2 technology, used on both approaches, does not support the element-wise *maximum* operation for 16-bits unsigned integer vectors, whereas it is supported for signed vectors. Asymmetrically for 8-bit vectors the element-wise *maximum* operation is supported for unsigned elements but for not signed elements.

3 A new SIMD implementation

This section describes two new techniques to improve the performance of SSE2 implementations of the SW algorithm: *i*) parallelization of the vertical dependencies within a SIMD vector; *ii*) definition of a dynamic

scaled map of offset values to provide a larger dynamic range, even when 8-bit unsigned vectors are applied. Hence we obtain taking advantage of the faster 8-bit implementation were previously 16-bit versions where necessary to guarantee the validity of the computation.

3.1 Base SSE2 Implementation

To obtain a fast algorithm we choose vertically oriented vectors, like Rognes [4] and Farrar [6], see the arguments presented in Section 2. An initialization step computes a *query specific score matrix* from the substitution cost function $Sbt(x, y)$, i.e., $Qss(i, y) = Sbt(P[i], y)$ for letters x, y and index i. We divide the query string into substrings of size k, the query specific score matrix is defined as a set of k-element vectors that contain the $Sbt(x, y)$ scores corresponding to each substring. This matrix is used to compute the diagonal dependencies of the SW algorithm, by adding the query specific score matrix's vector that corresponds to the current position of the query sequence and to the current database sequence character, to the diagonal values of the previous database sequence character.

An example of a precomputed query specific score matrix can be seen in Figure 2(b), which shows the precomputed matrix for the query "CGTTAATC" obtained with the $Sbt(x, y)$ function defined in Figure 2(a), where a +2 score was considered for matches and a -1 score for mismatches.

Figure 2(c) illustrates an example of a diagonal dependencies computation using the query specific score matrix presented in Figure 2(b), where a database sequence starting in "AT" is being aligned to the query sequence "CGTTAATC", assuming 4-elements SIMD vectors. The diagonal dependencies under evaluation are represented in bold.

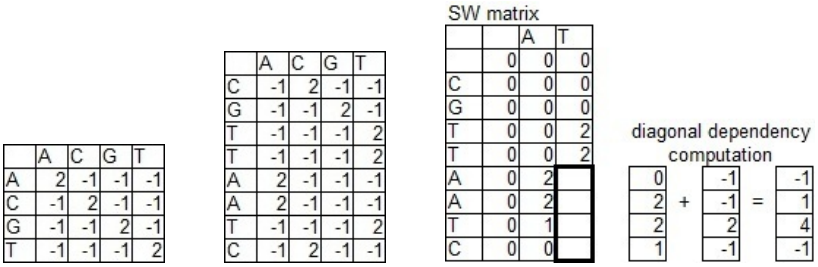
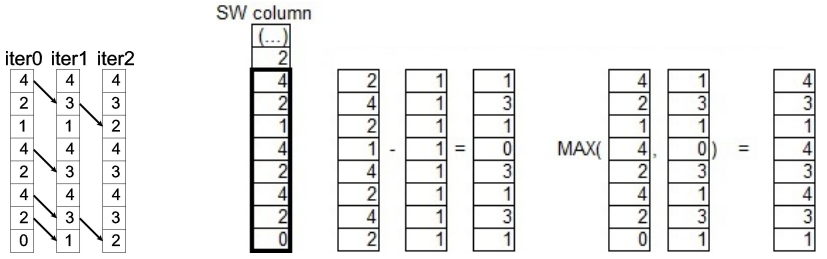


Fig. 2. Alignment example for the query sequence CGTTAATC: (a) Substitution cost function $Sbt(x, y)$; (b) Query specific score matrix; (c) Algorithm evaluation step.



(a) Vertical propagation phase.

(b) Evaluation of iteration 1 of the vertical propagation phase.

Fig. 3. Parallelization of vertical dependencies.

3.2 Parallelization of vertical dependencies

Representing the score matrix as SIMD vectors parallel to the query sequence, the vertical dependencies cannot be parallelized. As such, they must propagate inside each vector and have to be sequentially calculated to ensure the algorithm's correctness. However, it should be observed that several independent propagations may simultaneously occur inside a vector. We propose that those propagations are also parallelized, to reduce the overhead of computing them. Namely because this process is in the critical path of the algorithm and moreover it can effectively nullify the whole speed-up of the SIMD approach. Notice that even with our technique this phenomena can this occur, although for many less configurations.

After calculating the diagonal and horizontal dependencies of a given vector, the algorithm iteratively computes all the still existing vertical dependencies. A new iteration is issued whenever any propagation is required to be processed and the algorithm is stopped as soon as no further changes occur in the vector.

Some iterations of this technique are illustrated in Figure 3(a). For better comprehension of the process, the computation of iteration 1 is illustrated in detail in Figure 3(b), where the area in bold represents the vector whose vertical dependencies are being evaluated, using a vector filled with an insertion score of -1.

3.3 Pseudocode

Figure 4 shows the pseudo-code that implements the vertical propagation technique we have just described. The algorithm receives the following arguments:

- `QM` – Precomputed query-specific score matrix;
- `DBS` – Database sequence;
- `ds` – Deletion score;
- `is` – Insertion score;
- `b` – Translation base;

```
1   FUNCTION SWSSE2(QM,DBS,ds,is,b,m,n)
2
3   CONST VECTOR INSERTION = {is,is,is,is,is,is,is,is,is,is,is,is,is,is,is,is}
4   CONST VECTOR DELETION = {ds,ds,ds,ds,ds,ds,ds,ds,ds,ds,ds,ds,ds,ds,ds,ds}
5   CONST VECTOR BASE = {b,b,b,b,b,b,b,b,b,b,b,b,b,b,b,b}
6
7   VECTORS DP[n/16][m], DiagD, InsD, DelD
8   INTEGERS i,j
9
10  INITIALIZE_FIRST_ROW_AND_COLUMN()
11
12  FOR j=1 to m do {
13    FOR i=1 to (n+15)/16 do {
14      /* Diagonal dependency computation */
15      DiagD = LSHIFT(DP[i][j-1],1)
16      DiagD = OR(DiagD,RSHIFT(DP[i-1][j-1],15))
17      DiagD = (DiagD + QM[DBS[j]][i]) - BASE
18
19      /* Horizontal dependency computation */
20      DelD = DP[i][j-1] - DELETION
21      DP[i][j] = MAX(DiagD,DelD)
22
23      /* Vertical parallelization cycle */
24      WHILE DP[i][j] is altered DO {
25        InsD = LSHIFT(dp[i][j],1)
26        InsD = OR(InsD,RSHIFT(DP[i-1][j],15))
27        InsD = InsD - INSERTION
28        DP[i][j] = MAX(DP[i][j],InsD)
29      }
30    }
31  }
32
33  RETURN DP
```

Fig. 4. Pseudocode of the proposed algorithm, assuming a SIMD execution with 16 unsigned vector elements, each one with 8-bits.

`m` – Database sequence length;
`n` – Query sequence length.

The algorithm starts by initializing three constant vectors that will be required during the computation of the dependencies:

`INSERTION` - Filled with the insertion score elements;
`DELETION` - Filled with the deletion score elements;
`BASE` - Filled with a pre-defined *translation base*.

The need for the translation base arises from the previously referred restrictions of the SSE2 instruction set extension, which imposes the usage of unsigned integer vectors. As such, there can be no negative values in `QM`. To circumvent such restriction, a translation base value (`b`), corresponding to the absolute value of the `QM`'s lowest negative score, is added to every element of `QM`. As an example, if `QM` has values in the range [-1,2], 1 is added to each element so that `QM` will have values in the range [0,3]. As a consequence, this `BASE` vector will be subsequently needed to subtract from the result of the diagonal dependencies computation.
The internally declared vectors have the following meaning: `DP` represents the resulting output matrix and `DiagD`, `InsD` and `DelD` stores the intermediate values required to resolve the *diagonal*, *insertion* and *deletion* dependencies.
Before the actual implementation starts, the first row and column of the DP matrix must be initialized with 0s, just as in the regular implementation.

The main body of the alignment algorithm is then implemented by two nested loops, where the SIMD vectors corresponding to the DP matrix are evaluated along the database sequence direction. Such computation is performed by first resolving the diagonal and horizontal dependencies. This cycle is concluded by implementing the proposed optimization technique to accelerates the computation of the in-vector vertical dependencies (`WHILE` loop).
In the pseudocode represented in Figure 4, the following set of SIMD operations was appropriately mapped with Intel's SSE2 instructions:

- `LSHIFT(VEC,a)` - Shifts all elements in `VEC` a positions to the left, inserting 0s to fill the `a` right elements;
- `RSHIFT(VEC,a)` - Shifts all elements in `VEC` a positions to the right, inserting 0s to fill the `a` left elements;
- `OR(VEC1,VEC2)` - Bitwise OR of the vectors;
- `MAX(VEC1,VEC2)` - 8-bit element-wise maximum;
- `VEC1 + VEC2` - 8-bit element-wise addition;
- `VEC1 - VEC2` - 8-bit element-wise subtraction.

3.4 Dynamic scaled map of offset values

Intel's SSE2 extension to the instruction set makes use of 128-bit processor registers, where SIMD vectors with $k = 16$ elements, each with 8-bit resolution, can be declared. This implies that 16 operations are executed in parallel, instead of the 8 operations formerly permitted by the MMX extension.
However, the usage of 8-bit vector elements imposes a limit on the magnitude of each score to the range of [0,255]. Rognes & Seeberg [4] partially solve this problem by starting the execution of the algorithm with a precision of only 8-bits and by restarting the computation, with 16-bit vector elements, as soon as an overflow is detected. Though effective, this solution can hardly provide efficient results, since it imposes an important loss of time due to restarted computation and moreover it incurs in 2 slowdown because of the change from $k = 16$ element vectors to $k = 8$ element vectors.
In contrast we present a more flexible and efficient approach for this problem. We define offset bases that implicitly provide an unlimited extension of the dynamic range of the score values under evaluation.
Hence, to support score values greater than 255, the DP matrix is divided into several blocks, each one assigned to an appropriate offset value. In the conducted implementation, each block was defined as a square group of 16×8 adjacent score values. For each of such blocks, the *effective offset* is defined by multiplying a pre-evaluated *offset scale* by a fixed *offset base*, as defined in the following equation.

$$\textit{effective offset} = \textit{offset scale} \times \textit{offset base}$$

Our implementation has a fixed offset base of 64. The offset scale values are assigned to the blocks dynamically in order to confine the variation

interval of the corresponding scores within a safe range. The real values of the evaluated scores are obtained by summing up the effective offset, corresponding to the considered block, with the partial values represented in the DP matrix:

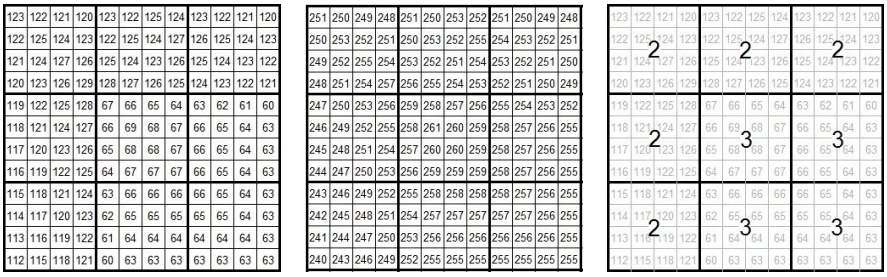
$$real\ score = partial\ score + (offset\ scale \times offset\ base)$$

Figure 5 illustrates this process, it shows blocks of 4×4 adjacent score values and using an offset base of 64. For example for the top-left score element of the central block, the real value is calculated by summing the partial score value (67) with the corresponding effective offset ($192 = 3 \times 64$), leading to the real score of 259. This value could not be represented if a straightforward 8-bit precision had been adopted.

In the conducted implementation, the offset scale is evaluated before the first iteration (column) of such block, in order to confine the majority of the scores to the range of $[64, 191]$. This provides a tolerance margin that will assure that the subsequent scores, that will be evaluated in such block, will be confined within the allowed $[0, 255]$ range. Upon the completion of the processing of each block, the offset scale that will be applied and the following block is re-evaluated:

- If the score values dropped to the range of 0-64 and the offset scale is greater than 1, the offset scale value is reduced by one;
- If the score values reached the range of 192-255, the offset scale value is increased by one.

After this scale re-definition, the score values of the following block are adjusted accordingly.



(a) Partial scores values. (b) Real score values. (c) Offset map.

Fig. 5. Application example of the proposed dynamic scaled offset technique.

Hence, the offset technique allows the real score values to assume an unlimited dynamic range, represented by the combination of the implicit offset map and the partial 8-bit score vectors, accommodated in the DP matrix. Such feature is offered at a cost of a minor computational overhead, since the considered offset values only have to be re-evaluated at the edges of the considered blocks.

4 Experimental Results

To evaluate the performance improvements provided by the set of optimization techniques that have been proposed in the previous section, the following set of implementations of the Smith-Waterman algorithm were evaluated:

- `SISD` - Non-optimized *vanilla* implementation (sequential Single Instruction Single Data (SISD) approach);
- `SIMD_0` - Basic SSE2 implementation: 8 SIMD vector elements (16-bits each);
- `SIMD_1` - `SIMD_0` implementation with *parallel resolution of vertical dependencies*: 8 SIMD vector elements (16-bits each);
- `SIMD_2` - `SIMD_1` implementation with *dynamic scaled offsets*: 16 SIMD vector elements (8-bits each).

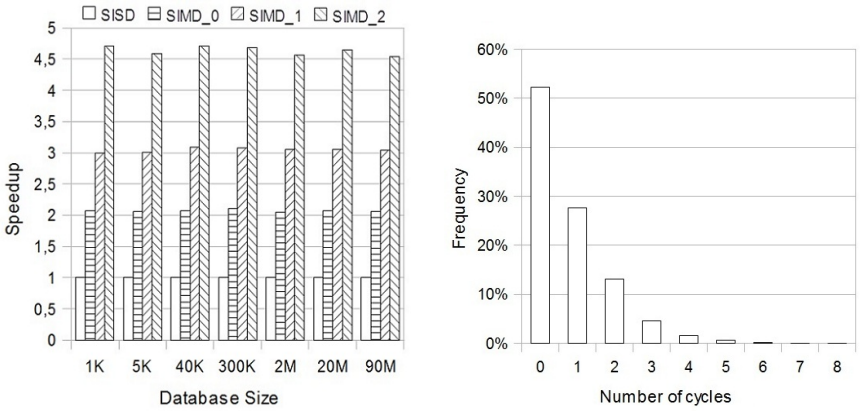
Since the two proposed techniques provide improvements of entirely independent aspects of the algorithm, they may coexist within the same setup. Implementation `SIMD_2` presents an example of such scenario.
To properly evaluate the proposed techniques, real DNA data sequences with quite different lengths, obtained from the *genebank* and *uniprot* repositories, were extensively adopted. The selected sample set for the *database* and *query* sequences is presented in Table 1. The source of such sequences are identified by their accession code and pair range (between square brackets), whenever applicable, otherwise we use the first pairs that from the sequence. The tests were executed using an Intel Core

Table 1. Considered DNA datasets.

DB	Source	Query	Source
1K	CY060745.1	20	20 pairs of GV638921.1
5K	AB543626.1	70	70 pairs of NC_014003.1
40K	NC_000002.11[119981384-120023228]	148	AB555751.1
300K	NC_000002.11[122095352-122407052]	210	210 pairs of AB555718.1
2M	NC_006047.1[1-2037969]	280	280 pairs of AB543626.1
20M	NW_876305.1[1-23897727]	350	350 pairs of CY060745.1
90M	NT_032977.9[1-90908613]		

2 Quad Q9550 processor, running at 2.83 GHz, characterized by a 32 kBytes 8-way set associative L1 data cache and an unified 6 MB 24-way set associative L2 cache.
The chart presented in Figure 6(a) depicts the performance gains that can be obtained with the proposed techniques. Although such enhancements are focused on improving specific aspects of any generic SIMD implementation, the presented evaluation considered a standard sequential implementation (`SISD`) of the SW algorithm as the *baseline* setup. In

the same trend, the SIMD_0 setup corresponds to a plain and straightforward SIMD implementation of the original SISD setup. When applied to the considered set of *database* and *query* sequences, such direct parallel implementation gave rise to an average speedup of 2.07. It is important to notice that that implementation does not guarantee the correctness of the resulting computation, since the cell values may overflow at 255. Hence the performance of SIMD_0 is shown mainly to demonstrate the importance of the vertical dependency optimization and the importance of the offset technique. Notice that even though we were unable to test the implementation by Rognes and Seeberg [4], which provides a speedup of about 6, we expect that our techniques would likewise improve that implementation in a similar way.



(a) Performance speedups. (b) Vertical propagation pattern.

Fig. 6. Obtained experimental results.

The histogram presented in Figure 6(b) depicts the number of iterations that are required by the proposed optimization to *propagate the vertical dependencies* of each SIMD vector of the DP matrix. Contrasting to a standard implementation (which would require a fixed number (8) of iterations), the proposed optimization requires a significantly smaller number of iterations: 0.79 (on average). This justifies the average speedup value (of about 47%) that was obtained by the SIMD_1 setup when compared with the SIMD_0 implementation.

The rightmost bars of the chart presented in Figure 6(a) represent the speedup values that were obtained with the implementation of the SW algorithm by considering 16 SIMD vector elements (8-bits each) and using the proposed improvement based on a *dynamic scaled offset map* (SIMD_2 setup). From the results it can be observed that such enhancement gives rise to a further performance gain (of about 53%), when compared to the SIMD_1 setup.

From a cumulative optimization point of view, it was observed that the simultaneous application of the two proposed enhancement techniques

provided an average speedup of about 2.24, when compared to a standard SIMD implementation (`SIMD_0` setup). When compared with the *vanilla* SISD implementation, the proposed improvements provided a performance gain of about 4.64 times.

5 Conclusions

We presented two improvements for SIMD implementations of the SW algorithm. The first technique accelerates the evaluation of the vertical dependencies, as a means to take advantage of independent vertical dependencies propagation in the computation of SIMD vectors parallel to the query sequence. The other technique provides the usage of SIMD vectors with smaller sized elements, without any precision loss, hence providing support for large, and fast, vectors that were previously sacrificed in favor of precision. This offset base representation can also be applied in the implementations of other algorithms, as a viable and more efficient alternative to the restart and representation change scheme previously proposed, thus achieving greater speedups on input batteries in which a great number of restarts are expected. Using both techniques simultaneously we obtained a SIMD implementation that is two times faster than the straightforward SIMD approach and 4.5 faster than a plain SISD implementation.

References

1. Shpaer, E.G., Robinson, M., Yee, D., Candlin, J.D.: Sensitivity and selectivity in protein similarity searches: A comparison of Smith-Waterman in hardware to BLAST and FASTA. Genomics (1996) 179–191
2. Hughey, R.: Parallel hardware for sequence comparison and alignment. Computer Applications in the Biosciences **12**(6) (1996) 473–479
3. Wozniak, A.: Using video-oriented instructions to speed up sequence comparison. Computer Applications in the Biosciences **13**(2) (1997) 145–150
4. Rognes, T., Seeberg, E.: Six-fold speed-up of Smith-Waterman sequence database searches using parallel processing on common microprocessors. Bioinformatics **16**(8) (2000) 699–706
5. Meng, X., Chaudhary, V.: Exploiting multi-level parallelism for homology search using general purpose processors. In: Proc. International Conference on Parallel and Distributed Systems (ICPADS'2005), IEEE Computer Society (2005) 331–335
6. Farrar, M.: Striped Smith-Waterman speeds database searches six times over other SIMD implementations. Bioinformatics **23**(2) (2007)
7. Almeida, T., Roma, N.: A parallel programming framework for multi-core DNA sequence alignment. In: Proc. Int. Conference on Complex, Intelligent and Software Intensive Systems (CISIS'2010). (2010) 907–912