CONCURRENCY AND COMPUTATION: PRACTICE AND EXPERIENCE
Concurrency Computat.: Pract. Exper. 2015; **27**:958–972
Published online 9 September 2014 in Wiley Online Library (wileyonlinelibrary.com). DOI: 10.1002/cpe.3371

SPECIAL ISSUE PAPER

GSWABE: faster GPU-accelerated sequence alignment with optimal alignment retrieval for short DNA sequences

Yongchao Liu*,† and Bertil Schmidt

Institut für Informatik, Johannes Gutenberg Universität Mainz, 55099 Mainz, Germany

SUMMARY

In this paper, we present GSWABE, a graphics processing unit (GPU)-accelerated pairwise sequence alignment algorithm for a collection of short DNA sequences. This algorithm supports all-to-all pairwise global, semi-global and local alignment, and retrieves optimal alignments on Compute Unified Device Architecture (CUDA)-enabled GPUs. All of the three alignment types are based on dynamic programming and share almost the same computational pattern. Thus, we have investigated a general tile-based approach to facilitating fast alignment by deeply exploring the powerful compute capability of CUDA-enabled GPUs. The performance of GSWABE has been evaluated on a Kepler-based Tesla K40 GPU using a variety of short DNA sequence datasets. The results show that our algorithm can yield a performance of up to 59.1 billions cell updates per second (GCUPS), 58.5 GCUPS and 50.3 GCUPS for global, semi-global and local alignment, respectively. Furthermore, on the same system GSWABE runs up to 156.0 times faster than the Streaming SIMD Extensions (SSE)-based SSW library and up to 102.4 times faster than the CUDA-based MSA-CUDA (the first stage) in terms of local alignment. Compared with the CUDA-based gpu-pairAlign, GSWABE demonstrates stable and consistent speedups with a maximum speedup of 11.2, 10.7, and 10.6 for global, semi-global, and local alignment, respectively. Copyright © 2014 John Wiley & Sons, Ltd.

Received 5 February 2014; Revised 4 August 2014; Accepted 20 August 2014

KEY WORDS: Needleman-Wunsch; Smith-Waterman; Sequence alignment; CUDA; GPU

1. INTRODUCTION

Pairwise sequence alignment is undoubtedly one of the fundamental techniques in many biological applications such as biological sequence database search [1, 2], multiple sequence alignment [3, 4] and next generation sequencing (NGS) read alignment [5–8]. There are three basic and frequently used types of pairwise alignment: global alignment, semi-global alignment, and local alignment. Needleman and Wunsch [9] pioneered an exact algorithm for finding the optimal global alignment between sequence pairs, known as the Needleman–Wunsch (NW) algorithm. This algorithm has a dynamic programming kernel and has already been extended to address different alignment problems, among which semi-global alignment and local alignment are well-known. Semi-global alignment follows the same formula as the NW algorithm, but does not penalize gaps at the beginning or at the end of the alignment. Local alignment was proposed by Smith and Waterman [10] to find similar regions between two sequences rather than align them globally. This algorithm, known as the Smith–Waterman (SW) algorithm [10, 11], has a different formula from the NW algorithm but follows the dynamic programming scheme originating from the NW algorithm. All of the three alignment types can be computed in linear space but have quadratic time complexities with respect to sequence length. This quadratic runtime makes pairwise alignment computationally demanding,

*Correspondence to: Yongchao Liu, Institut für Informatik, Staudingerweg 9, Mainz 55128, Germany.
†E-mail: liuy@uni-mainz.de

especially for large-scale datasets. This has therefore driven a substantial amount of research to parallelize pairwise alignment on high-performance computing architectures ranging from loosely-coupled to tightly-coupled ones, including clouds [12], clusters [13, 14], and accelerators [15, 16]. Among these architectures, accelerators, including single instruction multiple data (SIMD) vector processing units (VPUs) affiliated to CPUs, field programmable gate arrays (FPGAs), and general-purpose GPUs, have recently been the predominant techniques.

The SIMD VPUs affiliated to CPUs are the most widely used techniques. Two general approaches have been investigated to meet the computational features of SIMD vectors: one is the inter-task (or inter-sequence) parallelization model and the other is the intra-task (or intra-sequence) model. The inter-task model performs multiple alignments in individual SIMD vectors with one vector lane computing one alignment (e.g. [17]). The intra-task model computes in parallel the alignment of a single sequence pair in vectors based on two computational patterns: vectorized computation parallel to minor diagonals in the alignment matrix [18] and vectorized computation parallel to the query sequence in a sequential [19] or striped [20] layout. The two kinds of models provide a general framework for other accelerators with SIMD VPUs, including Cell Broadband Engine and general-purpose GPUs. Few implementations [21, 22] have been proposed on Cell Broadband Engine, and all of them are based on the intra-task model with the striped layout. On general-purpose GPUs, open graphics library was initially used for programming [23]. As the advent of the CUDA programming model, a number of implementations (e.g. [24–32]) have been developed using CUDA, among which CUDASW++ software package [25] is popular. As for FPGAs, linear systolic arrays (e.g [33, 34]) and custom instructions (e.g. [35]) have been proposed to efficiently accelerate pairwise alignments. In addition, the Xeon Phi coprocessor has recently been emerging as a new type of accelerator, and we have already seen some pioneering work (e.g. SWAPHI [36]) to accelerate pairwise alignments on the coprocessors.

However, almost all GPU-based implementations merely calculate optimal alignment scores. MSA-CUDA [37] is the first algorithm that is able to retrieve optimal alignments on CUDA-enabled GPUs. This algorithm employs the Myers–Miller algorithm [38], whose major advantage is the probability of aligning very long sequences as the alignment retrieval works in linear space. gpupairAlign [29] proposed to directly record alignment moves while performing alignments. This approach stores alignment moves in four bit-wise backtracking matrices and has a linear time complexity for alignment retrieval. Although the backtracking matrices take much GPU device memory, this approach has been shown to work well for short protein sequences. Recently, CUDAlign [31] and SW# [39] have been developed to support alignment retrieval. However, they merely work well for pairwise alignment of very long DNA sequences. In addition, on other high-performance computing platforms, some programs such as mpiBLAST [40], scalaBLAST [41], and pGraph [42] have been developed to perform all-to-all sequence comparison.

In this paper, we present GSWABE, an extension of our GSWAB algorithm [43] to comprehensively offer three types of pairwise alignment, that is, global alignment, semi-global alignment, and local alignment, on CUDA-enabled GPUs for a collection of short DNA sequences. Both GSWAB and GSWABE perform all-to-all pairwise alignments, as well as trace back optimal alignments, on CUDA-enabled GPUs. GSWAB targets local alignment and presents a tiled SW implementation using CUDA to facilitate fast retrieval of optimal local alignments. In contrast, in this new GSWABE algorithm, we have provided support for three types of alignment. Moreover, we have further investigated a general tiled approach for all of the three alignment types based on the CUDA programming model, because all of them are based on dynamic programming and share almost the same alignment matrix computational pattern. It should be noted that alignment matrix tiling has already been a popular technique for CUDA-based sequence alignment implementations in order to significantly improve memory access efficiency. However, almost all of existing CUDA-based implementations merely employ this technique to compute optimal alignment scores. How to efficiently combine a tiled design with alignment retrieval has not been well investigated for each alignment type. On the other hand, a general tiled dynamic programming framework, with alignment retrieval supported, has also not yet been deeply explored on CUDA-enabled GPUs. The performance of GSWABE has been evaluated on a Kepler-based Tesla K40 GPU using a variety of short DNA sequence

DOI: 10.1002/cpe

datasets. This performance has been further compared with that of the SSE-based SSW library [44] and two CUDA-based programs: MSA-CUDA [37] and gpu-pairAlign [29], running on the same system platform.

2. BACKGROUND

2.1. Global alignment

Given a sequence R, we define $R[i, j]$ to denote the substring that starts at position i and ends at position j, and $R[i]$ to denote the i-th symbol. For a sequence pair R_1 and R_2, the recurrence of global alignment, that is, the NW algorithm, with affine gap penalties is defined as

$$\begin{aligned} H_{i,j} &= max \begin{cases} H_{i-1,j-1} + sbt(R_1[i], R_2[j]) \\ E_{i,j} \\ F_{i,j} \end{cases} \\ E_{i,j} &= max \begin{Bmatrix} E_{i-1,j} \\ H_{i-1,j} - \alpha \end{Bmatrix} - \beta \\ F_{i,j} &= max \begin{Bmatrix} F_{i,j-1} \\ H_{i,j-1} - \alpha \end{Bmatrix} - \beta \end{aligned} \tag{1}$$

where $H_{i,j}$, $E_{i,j}$, and $F_{i,j}$ ($1 \leqslant i \leqslant |R_1|$ and $1 \leqslant j \leqslant |R_2|$) represent the global alignment score of two prefixes $R_1[1, i]$ and $R_2[1, j]$ with $R_1[i]$ aligned to $R_2[j]$, $R_1[i]$ aligned to a gap and $R_2[j]$ aligned to a gap, respectively. α is the gap open penalty, β is the gap extension penalty, and *sbt* is a scoring function, which is usually represented as a scoring matrix and defines the matching/mismatching scores between symbols. For protein sequences, we have a set of well-established scoring matrices to use, such as the BLOSUM [45] and PAM [46] families. For DNA sequences, we usually assign fixed scores for matches and mismatches. From Equation (1), we can see that each cell depends on its upper, left, and upper-left (or diagonal) neighbor cells. For the convenience of discussion, we assume that R_1 and R_2 are indexed vertically and horizontally in the alignment matrix, respectively.

The recurrence is initialized as follows. The first row and the first column of matrix H are initialized according to Equations (2) and (3), respectively, defined as

$$H_{i,0} = -i \times \beta - \alpha \tag{2}$$

$$H_{0,j} = -j \times \beta - \alpha \tag{3}$$

It needs to be stressed that $H_{0,0}$ is always set to zero. For matrices E and F, their first rows and columns are filled with negative infinity. After finishing the alignment matrix computation, the cell ($|R_1|$, $|R_2|$) in matrix H stores the optimal global alignment score. The alignment backtracking starts from the cell ($|R_1|$, $|R_2|$) and continues until reaching the cell (0, 0). To obtain optimal global alignment, a straightforward approach is to store all alignment moves in a backtracking matrix. In this way, the alignment retrieval is able to work in linear time complexity. However, the quadratic space requirement of this approach makes it not well suited to long sequences. An alternative approach is to use the linear-space Myers–Miller algorithm [38], whose alignment backtracking merely requires a linear space and thus provides the possibility of aligning very long sequences on modest compute resources. However, this algorithm has a drawback of quadratic time complexity for the alignment retrieval, thus possibly not as efficient as the backtracking-matrix-based approach for shorter sequences.

DOI: 10.1002/cpe

2.2. Semi-global alignment

Semi-global alignment computes the alignment matrices following the same recursion with global alignment, but additionally allows to behave liberal on the end gaps for one sequence or both. This additional feature of semi-global alignment leads to some differences from global alignment in terms of alignment matrix initialization, optimal alignment score obtaining and alignment retrieval. For R_1 and R_2, semi-global alignment is often used in the following three cases:

(i) do not penalize gaps at both the beginning and the end of R_1;
(ii) do not penalize gaps at both the beginning and the end of R_2;
(iii) do not penalize gaps at both the beginnings and the ends of both sequences.

For case (i), the first column of matrix H will still be initialized using Equation (3), but the first row will be initialized using Equation (4), instead of Equation (2).

$$H_{i,0} = 0 \tag{4}$$

In this case, we obtain the optimal alignment score from the cell with the greatest value in the last row of H, and trace back the optimal alignment starting from this cell until any cell in the first row is reached. For case (ii), we initialize the first row of matrix H using Equation (2), but initialize the first column using Equation (5), instead of Equation (3).

$$H_{0,j} = 0 \tag{5}$$

In this case, the optimal alignment score corresponds to the cell with the greatest value in the last column of H, and the optimal alignment backtracking starts from this cell until reaching any cell in the first column. For case (iii), the first row and the first column of matrix H are initialized using Equations (4) and (5), respectively. The optimal alignment score can be obtained from the cell with the greatest value in both the last row and the last column of H. Accordingly, the optimal alignment backtracking starts from the cell and finishes when any cell in either the first row or the first column is met, which can carried out by means of the backtracking-matrix-based approach. In this paper, we have merely investigated case (iii).

2.3. Local alignment

Given R_1 and R_2, the recurrence of local alignment, that is, the SW algorithm, with affine gap penalties is defined as

$$\begin{aligned} H_{i,j} &= max \begin{cases} H_{i-1,j-1} + sbt(R_1[i], R_2[j]) \\ E_{i,j} \\ F_{i,j} \\ 0 \end{cases} \\ E_{i,j} &= max \begin{Bmatrix} E_{i-1,j} \\ H_{i-1,j} - \alpha \end{Bmatrix} - \beta \\ F_{i,j} &= max \begin{Bmatrix} F_{i,j-1} \\ H_{i,j-1} - \alpha \end{Bmatrix} - \beta \end{aligned} \tag{6}$$

Compared with Equation (1), the only one difference in the recursion is that each cell value in matrix H must be non-negative, which leads to different alignment matrix initialization, optimal alignment score obtaining, and alignment backtracking. Firstly, the SW algorithm initializes the first row and the first column of matrix H to zero using Equations (4) and (5), respectively. Meanwhile, for matrices E and F, their first rows and columns are usually also filled with zeros. Secondly, the optimal alignment score corresponds to the cell (i^*, j^*) with the greatest value within the entire matrix H. Finally, the optimal alignment backtracking starts from (i^*, j^*) and continues until reaching a cell with zero value. The optimal local alignment can be obtained either by means of the

DOI: 10.1002/cpe

backtracking-matrix-based approach or the Myers–Miller algorithm. However, prior to using the Myers–Miller algorithm, we have to conduct two additional runs of score-only SW algorithm to gain the global alignment range corresponding to the optimal local alignment.

2.4. GPU architecture

A CUDA-enabled GPU is built around a fully configurable array of scalar processors (SPs) and further organizes the SPs into a set of multi-threaded streaming multiprocessors (SMs). The GPU architecture has evolved through three generations: Tesla, Fermi, and Kepler. From generation to generation, some substantial changes in the architecture have been made, such as the SM architecture, the configurability of shared memory size, and local/global memory caching.

For an architecture, it may own varied number of SMs per GPU from product to product, but has a fixed number of SPs per SM. Tesla configures each SM to contain eight SPs and Fermi 32 SPs. Kepler adopts a new SM architecture with 192 SPs per SM. In Tesla, all SPs per SM share a fixed-size 16 KB shared memory. Fermi introduces a size-configurable shared memory (16 or 48 KB) for the first time, whereas Kepler further provides more flexible configurations (16, 32, or 48 KB). As for local memory, the memory size per thread is up to 16 KB in Tesla. However, both Fermi and Kepler allow up to 512 KB local memory per thread. Tesla does not cache both local and global memory. From Fermi, a L1/L2 caching hierarchy has been introduced to cache local/global memory. The L1 cache is size-configurable and provides data caching for individual SMs, whereas the L2 cache is fixed-size and provides unified data caching for the whole device. In Fermi, the global memory caching in L1 can be disabled at compile time, but the local memory caching in L1 cannot. Different from Fermi, Kepler does not allow L1 to cache global memory any more, but reserves it only for local memory accesses such as register spills and stack data. Kepler further opens the 48 KB read-only data cache, which is only accessible by texture units in Fermi, to cache read-only global memory data.

3. PARALLELIZATION USING CUDA

Given a collection of short DNA sequences, GSWABE comprehensively offers all-to-all pairwise global, semi-global, and local alignment and returns the optimal alignments in the `CIGAR` format [47], a compact representation of short-read (i.e. short DNA sequence) alignments, in order to reduce the number of accesses to GPU device memory. In this algorithm, we have investigated a tile-based dynamic programming framework for efficient alignment matrix computation as well as optimal alignment backtracking on CUDA-enabled GPUs. This tiled computing framework enables our algorithm to deeply exploit the powerful compute capability of CUDA-enabled GPUs.

3.1. Tile-based dynamic programming framework

For R_1 and R_2, our framework allocates a single backtracking matrix of size $|R_1| \times |R_2|$ with 2 bits representing the value of each cell, because each cell depends on its left, upper, and upper-left neighbors. For short DNA sequence (e.g. of a few hundred nucleotides), we can afford the memory overhead incurred by the backtracking matrix. We have allocated the backtracking matrix in local memory in order to benefit from the L1/L2 cache hierarchy on the Fermi and Kepler architectures. In the backtracking matrix, each cell must hold one of the four alignment move types: move from the left cell (ML), move from the upper cell (MU), move from the diagonal (i.e. upper-left) cell (MD), and a stop code (STOP). The stop code has merely been used in the backtracking matrix of local alignment and will be assigned only if the corresponding cell value in matrix H is zero. In this way, the alignment backtracking can be accomplished by only using this memory-reduced backtracking matrix.

3.1.1. Dynamic programming kernel. In our implementation, both the alignment and backtracking matrices are partitioned into small tiles of size 4×4. The whole dynamic programming computation is conducted by computing all tiles one-by-one. Figure 1 illustrates the tile-based processing of the alignment matrix. This tile-based computation can significantly improve data access performance

DOI: 10.1002/cpe

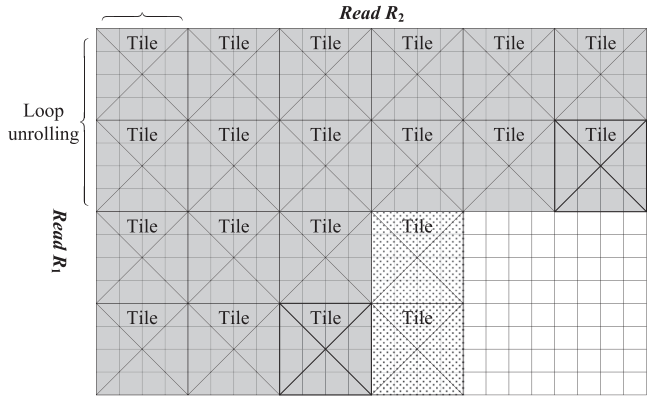

Figure 1. Tile-based processing of the dynamic programming based alignment.

because of the following two reasons. First, all alignment moves in a single tile can be represented by a 32-bit integer because each cell takes 2 bits. This means that only a single write to the backtracking matrix is needed for one tile computation, significantly reducing the number of writes to external device memory. Secondly, the data access performance to the backtracking matrix can get improved while tracing back the alignment. This is because all alignment moves in a single tile, represented as a 32-bit integer, can be realized by only a single data fetch from the backtracking matrix. While tracing back the alignment, if the next cell lies in the same tile with the current cell, we can reuse the current tile value with no need of reloading. Albeit the existence of caches, the alignment backtracking still can benefit from our data reuse, as the current tile value is possibly swapped out of the caches. Algorithm 1 gives the kernel of our tile-based dynamic programming framework, where we have indexed the strings of nucleotides in R_1 and R_2 from zero. In the kernel, every sequence length must be aligned to 8. For a sequence whose length is not multiple of 8, we will pad the sequence with dummy symbols, which have a zero substitution score for any symbol.

Both global and local alignments work well with the kernel shown in Algorithm 1. Compared with local alignment, global alignment conducts less computation. On one hand, local alignment has to ensure each cell value in matrix H to be non-negative as observed from Equations (1) and (6), thus resulting in one more arithmetic operation per cell. On the other hand, local alignment has to additionally trace the optimal local alignment score as well as the corresponding coordinate. Although semi-global alignment shares the same recursion with global alignment, the former is slightly more complex than the latter in terms of CUDA-based programming. This is because semi-global alignment has to check if the current cell lies in the first row or the first column of matrix H, in order to determine the optimal alignment score as well as the alignment backtracking starting coordinate. This check operation will result in conditional branches for the computation per cell, causing execution path divergence between threads within a warp. In this regard, in order to significantly reduce such divergences, we have separated the computation of the inner tiles from the boundary tiles (i.e. the tiles covering the last row or the column of matrix H). In this way, the execution path divergence can be avoided in the computation of the inner tiles, and will merely occur in the boundary tiles. For the computation of the inner tiles, we have used the tile-based dynamic programming kernel as shown in Algorithm 1. For each type of alignment, we have configured each CUDA kernel to use 48 KB L1 cache at the runtime.

As a query profile is not well suited to all-to-all pairwise alignments, we did not use this data structure in our algorithm. Instead, we calculate the matching/mismatching scores between symbols by directly comparing their values. On the other hand, the intermediate buffers for the dynamic programming are allocated in local memory, rather than in global memory. This is because on the Kepler architecture, writable global memory accesses can only be cached by the unified L2 cache, whereas local memory can obtain additional caching from the L1 cache per SM. All reads are stored in texture memory. To facilitate the tile-based data access and to reduce the number of texture fetches, for each read, we have packed four successive symbols using the integer data type with each symbol occupying 8 bits. In this manner, we can realize four symbols by a single texture fetch.

DOI: 10.1002/cpe

Algorithm 1 Kernel of the tile-based dynamic programming framework

```
for (i = 0; i < |R1|; i+ = 8) do        ▷ /*Assume that both |R1| and |R2| are multiple of 8*/
    initialize related variables;
    load four consecutive symbols starting from index i in R1;
    load four consecutive symbols staring from index i + 4 in R1;
    for (j = 0; j < |R2|; j+ = 4) do
        load the packed 4 symbols with indices [j, j + 3] to a register rB;
        reset registers rA1 and rA2 storing alignment moves for tiles T1 and T2;
                                ▷ /*(i/4, j/4) is the coordinate of T1 and (i/4 + 1, j/4) of T2*/
        for (k = 0; k < 4; k + +) do
            get the (j + k)-th symbol of R2 from rB;
            load the alignment scores of cell (i − 1, j + k) from the intermediate buffer;
            compute the k-th column of T1, and save the alignment moves in rA1;
            save the cell (i*, j*) with the maximum score for local alignment [conditional];
            compute the k-th column of T2, and save the alignment moves in rA2;
            save the cell (i*, j*,) with the maximum score for local alignment [conditional]
            save the alignment scores of cell (i + 7, j + k) to the intermediate buffer
        end for
        save rA1 to the alignment backtracking matrix;
        save rA2 to the alignment backtracking matrix;
    end for
end for
```

3.1.2. Optimal alignment backtracking. In our implementation, we have indexed R_1 and R_2 by row i ($0 \leqslant i < |R_1|$) and column j ($0 \leqslant j < |R_2|$), respectively, as mentioned earlier. In this case, $i = -1$ and $j = -1$ mean that the corresponding cell lies in the first row and the first column of matrix H, respectively, for example,. $(-1, -1)$ corresponds to $H_{0,0}$. For all of the three alignment types, their alignment backtracking differ in terms of both starting coordinate and stop criterion. However, they perform the same operations when moving to neighbor cells in the backtracking matrix, while tracing back the optimal alignment. In the alignment backtracking procedure, a move to any of the three neighbor cells can be reflected in the final alignment as follows.

(i) a move to the upper cell inserts a gap symbol into R_2 in the alignment;
(ii) a move to the left cell inserts a gap symbol into R_1 in the alignment;
(iii) a diagonal move means the alignment of the corresponding symbols in both sequences.

Assume that the alignment backtracking starts from the coordinate (i^*, j^*) for any of the three alignment types. As mentioned earlier, (i^*, j^*) corresponds to the cell $(|R_1| - 1, |R_2| - 1)$ for global alignment, the cell holding the greatest value in the last row and the last column of matrix H for semi-global alignment, and the cell with the maximum alignment score in the entire H for local alignment. For global alignment, the alignment backtracking ends at the cell (0, 0). For semi-global alignment, the alignment backtracking does not stop until either of the two coordinates is less than zero (meaning that it has reached a cell either in the first row or the first column). For local alignment, the alignment backtracking continues until either of the two stop criteria is met: one is that either of the two coordinates is less than zero; and the other is that the current cell holds the stop code. During the alignment retrieval, if the value held by the current cell (i, j) is ML, it means a deletion at position j of R_2. Subsequently, we will decrease j by 1 and move to the next cell. If the current cell value is MU, it means a deletion at position i of R_1, and then we will decrease i by 1. If the current cell value is MD, it means an alignment of symbols $R_1[i]$ and $R_2[j]$. In this case, we will decrease both i and j by 1.

Based on these discussions, we can see that the alignment retrieval for both semi-global and local alignments can be implemented by the same kernel. Besides an optimal alignment, the kernel also returns the alignment starting coordinate in order to decipher the alignment. Because an opti-

DOI: 10.1002/cpe

mal global alignment always starts from (0, 0), we do not need to additionally return the starting coordinate. Therefore, we have used a separate kernel for global alignment. Algorithm 2 shows the alignment backtracking kernel for global alignment and Algorithm 3 the kernel for both semi-global and local alignments.

Algorithm 2 Alignment backtracking kernel for global alignment

```
numOps = 0;
lastMove = STOP;
op = CIGAR_NULL;
ncigars = 0;
while (1) do
                                                    ▷ /*get the alignment move*/
    move = (i* < 0||j* < 0) ? STOP : GET_ALIGN_MOVE(i*, j*);
                                                    ▷ /*save the current operation?*/
    isDiagonal = (move == MD);
    if (lastMove == move) then ++numOps;
    else
        if (numOps) then cigars[ncigars++] = (numOps << 2)| op;
        end if
                                                    ▷ /*re-initialize the current operation*/
        numOps = 1;
        lastMove = move;
        op = isDiagonal ? CIGAR_M : (move == ML ? CIGAR_I: CIGAR_D);
    end if
    if (move == STOP) then
        break;
    end if
                                                    ▷ /*adjust the coordinate*/
    i* -= (isDiagonal || move == MU);
    j* -= (isDiagonal || move == ML);
end while
                                    ▷ /*corresponding to the first row or the first column of matrix H*/
if (i* < 0&&j* ≥ 0) then
    cigars[ncigars++] = ((j* + 1) << 2)| CIGAR_I;
else if (j* < 0&&i* ≥ 0) then
    cigars[ncigars++] = ((i* + 1) << 2)| CIGAR_D;
end if
```

3.2. Alignment launching

As the tasks of all-to-all pairwise alignments can be conceptualized as a task matrix, our algorithm distributes all alignment tasks onto the GPU using the cell-block-based task assignment approach in [37]. This task distribution divides the upper-triangle (or lower-triangle) of the whole task matrix into many equally-sized cell blocks (i.e. sub-matrices), and assigns a single thread block to process a single cell block. Within a thread block, a single thread is assigned to align a single sequence pair for simplicity. To alleviate global memory pressure on the storage of optimal alignments, we have conducted all alignments in a multi-pass way, thus allowing for processing a large number of sequences (e.g. millions of sequences). In each pass, we calculate the number N_{TB} of thread blocks at runtime as

$$N_{TB} = \frac{C \times N_{SM} \times N_{MRT}}{N_{TPB}} \tag{7}$$

DOI: 10.1002/cpe

Algorithm 3 Alignment backtracking kernel for semi-global and local alignments

```
numOps = 0;
lastMove = STOP;
op = CIGAR_NULL;
ncigars = 0;
istart = i*;
jstart = j*;
while (1) do
                                                    ▷ /*get the alignment move*/
    move = (i* < 0||j* < 0) ? STOP : GET_ALIGN_MOVE(i*, j*);
                                                    ▷ /*save the current operation?*/
    isDiagonal = (move == MD);
    if (lastMove == move) then ++numOps;
    else
        if (numOps) then cigars[ncigars++] = (numOps << 2)| op;
        end if
                                                    ▷ /*re-initialize the current operation*/
        numOps = 1;
        lastMove = move;
        op = isDiagonal ? CIGAR_M : (move == ML ? CIGAR_I: CIGAR_D);
    end if
    if (move == STOP) then
        break;
    end if
                                                    ▷ /*save the previous coordinate*/
    istart = i*;
    jstart = j*;
                                                    ▷ /*adjust the coordinate*/
    i* -= (isDiagonal || move == MU);
    j* -= (isDiagonal || move == ML);
end while
                                                    ▷ /*return the starting coordinate*/
return (istart, jstart);
```

where C is a scaling factor (default=4), N_{SM} is the number of SMs on the GPU, N_{MRT} is the maximum number of resident threads per SM, and N_{TPB} is the number of thread per thread block configured by users (default=64). In this case, the number of sequence pairs processed by a single pass is equal to $N_{pair} = N_{TB} \times N_{TPB}$ (the last pass possibly has fewer pairs).

Each time one pass is finished, the optimal alignments of the current pass are transferred back to the host. We have allocated a two-dimensional (logically) alignment buffer on the device to store the optimal alignments. This alignment buffer has a size of $N_{cigar} \times N_{pair}$ and is represented by the data structure `thrust::device_vector<>`. N_{cigar} denotes the maximum number of CIGAR entries allowed in an optimal alignment per sequence pair. Assuming that the maximum sequence length is L_{max} in an input dataset, N_{cigar} is impossible to exceed $2L_{max} + 1$, which can be derived from the alignment matrix. To confine the memory overhead of this alignment buffer, we calculate N_{cigar} as $min\{2L_{max}+1, 1024\}$ in our algorithm and abandon to report the optimal alignments of $> N_{cigar}$ CIGAR entries. Additionally, we have used the `thrust::copy()` function to transfer data.

4. PERFORMANCE EVALUATION

We have assessed our algorithm from three aspects: (i) performance evaluation using fixed-length sequences; and (ii) performance evaluation using variable-length sequences; and (iii) performance

DOI: 10.1002/cpe

Table I. Information of Illumina-like datasets used.

Dataset	Read length	No. of reads
D10K100	100	10 000 (10 K)
D10K250	250	10 000 (10 K)
D10K500	500	10 000 (10 K)
D50K100	100	50 000 (50 K)
D50K250	250	50 000 (50 K)
D50K500	500	50 000 (50 K)
D100K100	100	100 000 (100 K)
D100K250	250	100 000 (100 K)
D100K500	500	100 000 (100 K)

comparison to the SSE-based SSW library and two CUDA-based programs: MSA-CUDA and gpu-pairAlign. All of the following tests are conducted on a workstation with two hex-core Intel Xeon X5650 2.67 GHz CPUs and 96 GB RAM, running the Linux operating system (Ubuntu 12.04 LTS). This workstation has been further equipped with a Kepler-based Tesla K40 GPU. This GPU comprises 15 SMs (a total of 2 880 CUDA cores), works at a GPU clock rate of 876 MHz and a memory clock rate of 3 004 MHz, and has 12 GB device memory associated with a unified L2 cache of size 1.5 MB. All of the CUDA-based programs evaluated are compiled using CUDA toolkit 5.5 by specifying the GPU architecture as `-arch sm_35`.

To measure the speed, we usually use the runtime (measured in wall clock time) and GCUPS metrics. However, it needs be stressed that because of the variable number of cell accesses (usually depending on sequence similarities) in the alignment retrieval procedure, the GCUPS metric might not be able to reflect the speed as precisely as the score-only implementations. Nevertheless, this metric does provide a more convenient approach to facilitating users to estimate the runtime of an algorithm on a certain dataset. In this regard, our evaluations have used the two aforementioned metrics.

4.1. Evaluation on fixed-length sequences

We have first evaluated the performance of GSWABE on the Tesla K40 GPU (mentioned earlier), using nine Illumina-like 100-basepair (bp), 250-bp and 500-bp datasets with a uniform base error rate 2% (see Table I). These datasets are simulated from the *Escherichia coli K12 MG1655* genome using the *wgsim* utility in SAMtools [47]. This genome contains 4 639 675 nucleotides and has an accession number NC_000913 in GenBank. For each read length (i.e. 100, 250, and 500), we have generated three datasets with the different number of reads, that is, 10, 50, and 100 K, respectively.

Table II shows the performance of GSWABE on all of the Illumina-like datasets. For the datasets of the same number of sequences, the performance goes higher with the increasing of read length. For a fixed read length, however, the performance does not always becomes better as the number of sequences increases. The highest performance comes out when using the D100K500 dataset (100 K reads of length 500 bps each), where GSWABE yields a performance of 59.1 GCUPS for global alignment, 58.5 GCUPS for semi-global alignment and 50.3 GCUPS for local alignment. For each dataset, global alignment is the fastest and local alignment the slowest, although the performance differences are tiny. This phenomenon can be explained by the two following reasons. One is that local alignment performs more computation per cell than both global alignment and semi-global alignment, as mentioned earlier. The other is that semi-global alignment requires checking for each cell if it lies in the last row or the last column, and therefore has more computational overhead than global alignment, even though the code for semi-global alignment has been tuned in order to significantly reduce the number of such check operations.

Furthermore, to assess how much data communication between host and device affects the overall performance, we have measured the runtime proportion taken by the data transfer between the host and the GPU (Table II), which is overwhelmingly dominated by the transfer of alignment results from device to host in our algorithm. From the table, for a fixed number of sequences, the runtime proportion decreases as the increasing of read length. For a fixed read length, however, it does not

DOI: 10.1002/cpe

Table II. Runtime (in seconds), GCUPS and data transfer runtime proportion (in %) for GSWABE on Tesla K40 using Illumina-like sequences.

Read length	Type	No. of reads: 10 K Time	10 K GCUPS	10 K Trans.	50 K Time	50 K GCUPS	50 K Trans.	100 K Time	100 K GCUPS	100 K Trans.
100	Global	18	27.8	34.53	573	21.8	34.42	2 580	19.4	43.52
	Semi-global	19	26.3	32.99	592	21.1	33.37	2 631	19.0	42.56
	Local	20	25.0	31.52	619	20.2	31.06	2 704	18.5	40.35
250	Global	80	39.1	17.95	2 115	36.9	17.81	7 021	44.5	24.27
	Semi-global	81	38.6	17.71	2 155	36.3	17.63	7 133	43.8	24.12
	Local	94	33.2	14.96	2 478	31.5	15.11	7 965	39.2	20.98
500	Global	281	44.5	9.75	7 170	43.6	9.93	21 136	59.1	13.97
	Semi-global	286	43.7	9.82	7 264	43.0	9.79	21 376	58.5	13.85
	Local	340	36.8	8.26	8 602	36.3	8.24	24 859	50.3	11.87

Table III. Information of 454 real datasets used.

Dataset	No. of reads	Average length	Min length	Max length	Total no. of cell updates†
SRR000021	50 000	208±58	100	494	54 323 169 449 015
ERR307589	50 000	311±29	100	497	121 510 591 132 218
SRR1425907	50 000	455±28	400	500	258 992 697 638 208

† means the total number of cell updates in the computation of all-to-all pairwise alignments, which is calculated by summing up the length product of every sequence pair.

show very consistent trends when increasing the number of sequences. Additionally, for a specific dataset, the runtime proportion is reduced when moving from global alignment, via semi-global alignment, to local alignment (with an exception that global alignment has negligibly larger runtime proportion than semi-global alignment when using dataset D10K500). This observation is in accord with our expectation and can be explained by the fact that local alignment has more computational overhead than semi-global alignment, and semi-global alignment performs more computation than global alignment.

In addition, we have evaluated the performance of GSWABE at the scale of millions of sequences. For all-to-all pairwise alignments, as we know, the overall computational workload (represented as the total number of cell updates) will quadratically increase as either the number of sequences or sequence length becomes larger. To enable our algorithm to complete its execution in a few days, we have simulated one million reads of length 100 bps each in this evaluation, resulting in a computation workload of about 5 Peta cell updates for each alignment type. GSWABE has finished the computation in 167 080 s (about 46.41 h) for global alignment, 171 835 s (about 47.73 h) for semi-global alignment, and 179 871 s (about 49.96 h) for local alignment, leading to a performance of 29.9 GCUPS, 29.1 GCUPS and 27.8 GCUPS for global, semi-global and local alignment, respectively. As for the data transfer overhead, the runtime proportion is 33.97% for global alignment, 33.59% for semi-global alignment and 32.13% for local alignment.

4.2. Evaluation on variable-length sequences

Secondly, we have evaluated the performance of GSWABE on sequences of varied lengths. In this evaluation, three real datasets from 454 sequencing (Table III) have been used, all of which are publicly available at NCBI Sequence Read Archive and are named after their accession numbers. It needs to be stressed that besides 454 sequencing, our algorithm can also be used to align variable-length short-reads produced by other sequencing technologies such as Ion Torrent. In Table III, SRR000021 is sequenced from a *human individual NA15510*, ERR307589 from *fungi* and SRR1425907 from *Ochromonas sp. CCMP1393*. For each dataset, we did not use all of its

DOI: 10.1002/cpe

reads. Instead, we sampled 50 K reads, whose lengths are between 100 and 500 bps, to generate a subset of the original dataset. From the table, the average sequence length is 208±58 for dataset SRR000021, 311±29 for dataset ERR307589, and 455±28 for dataset SRR1425907. In addition, for each dataset, prior to alignment we have sorted all of its reads in ascending order of sequence length.

Table IV shows the performance of GSWABE on the real datasets comprising variable-length 454 reads. Our algorithm demonstrates a maximum performance of 42.6 GCUPS, 41.9 GCUPS, and 35.6 GCUPS for global, semi-global and local alignment, respectively. As read length increases, the performance becomes better and the runtime proportion taken by data transfer goes down for each case. This is also consistent with our observation from the simulated datasets.

4.3. Comparison to other counterparts

Thirdly, we have compared GSWABE with SSW [44], MSA-CUDA [37], and gpu-pairAlign [29]. SSW is a SSE-based pairwise local alignment library for genomic sequences, which extends the Farrar's striped SW algorithm [20] to enable the retrieval of optimal local alignments. This library has been further deployed in MOSAIK [8] to accelerate NGS read alignment to the reference genome. In order to enable all-to-all pairwise alignments, we have written a simple wrapper program based on this library. In MSA-CUDA, the first stage computes a pairwise distance matrix by means of all-to-all pairwise local alignments. Hence, in this evaluation, we have only taken the first stage of MSA-CUDA into account. For fair comparisons, we have used the local alignment implementations of both GSWABE and gpu-pairAlign. Additionally, we have used a single thread to run SSW on the aforementioned workstation, and executed all of the GPU-based algorithms on the Tesla K40 GPU.

Because SSW is very slow and neither MSA-CUDA nor gpu-pairAlign runs on the large datasets of $\geqslant$ 50 K reads, we have merely used the three small Illumina-like datasets comprising 10 K reads (Table I). Figure 2 shows the speedups of GSWABE over the three other programs in terms of local alignment. SSW achieves a roughly constant performance of 0.2 GCUPS for each dataset. GSWABE has demonstrated significantly superior performance to both SSW and MSA-CUDA, yielding a speedup of up to 156.0 over the single-threaded SSW and a speedup of up to 102.4 over MSA-CUDA. As observed from the figure, the speedups over SSW grow as read length increases, whereas

Table IV. Runtime (in seconds), GCUPS and data transfer runtime proportion (in %) for GSWABE on Tesla K40 using 454 real reads.

	Global alignment			Semi-global alignment			Local alignment		
Dataset	Time	GCUPS	Trans	Time	GCUPS	Trans	Time	GCUPS	Trans
SRR000021	1 983	27.4	37.46	2 032	26.7	37.09	2 215	24.5	33.84
ERR307589	3 301	36.8	22.02	3 398	35.8	21.40	3 876	31.3	18.78
SRR1425907	6 084	42.6	11.74	6 176	41.9	11.60	7 279	35.6	9.80

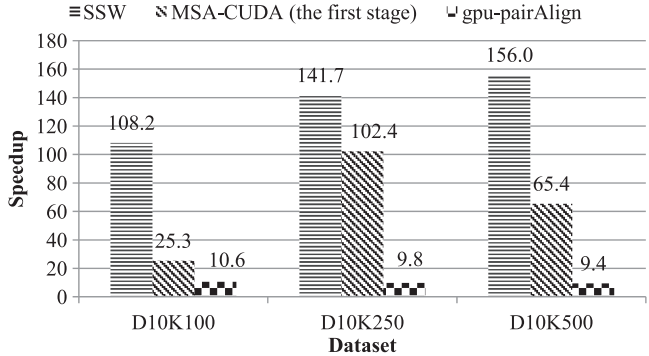

Figure 2. Speedups over SSW, MSA-CUDA and gpu-pairAlign on Tesla K40 in terms of local alignment.

DOI: 10.1002/cpe

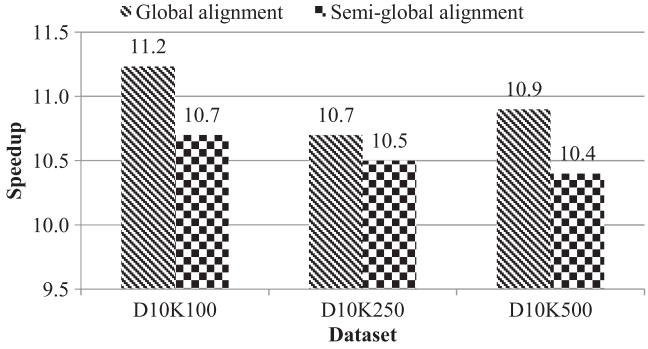

Figure 3. Speedups over gpu-pairAlign on Tesla K40 in terms of global and semi-global alignments.

the speedups over MSA-CUDA show relatively large fluctuations and do not always increase as read length becomes larger. Unlike SSW and MSA-CUDA, the speedups over gpu-pairAlign are relatively stable and consistent, where GSWABE achieves an average speedup of 9.9 $\pm$ 0.6 with the maximum speedup of 10.6. On the other hand, the speedup differences for MSA-CUDA and gpu-pairAlign might be because of the different time complexities of the alignment backtracking procedure, where the time complexity is quadratic for MSA-CUDA, but linear for both GSWABE and gpu-pairAlign.

Finally, we have compared our algorithm with gpu-pairAlign in terms of both global and semi-global alignments, using the three small datasets of 10 K reads (Figure 3). Because neither SSW nor MSA-CUDA implements global and semi-global alignment, we have excluded both of them from this evaluation. For either of the two alignment types, GSWABE produces stable speedups over gpu-pairAlign for each dataset, similar to the case of local alignment. Specifically, GSWABE achieves an average speedup of 10.9 $\pm$ 0.3 (with the maximum speedup of 11.2) for global alignment and an average speedup of 10.5 $\pm$ 0.2 (with the maximum speedup of 10.7) for semi-global alignment. Revisiting Figure 2, we could say that for each alignment type, GSWABE is able to achieve stable and consistent speedups over gpu-pairAlign.

5. CONCLUSIONS

We have presented GSWABE, a GPU-accelerated sequence alignment algorithm for a collection of short DNA sequences. This algorithm comprehensively provides support for all-to-all pairwise global, semi-global, and local alignment, and retrieves optimal alignments on CUDA-enabled GPUs. In GSWABE, we have investigated a tile-based dynamic programming framework to facilitate fast alignment backtracking on GPUs based on the Fermi and Kepler architectures. In our algorithm, for each sequence pair, we have employed a memory-reduced backtracking matrix to store the alignment moves along with the alignment matrix computation. Although its memory footprint has been significantly reduced, the backtracking matrix is still the most memory-consuming data structure in our algorithm, whose device memory footprint scales quadratically with the maximum allowable sequence length (default=640 and configurable at compile time).

We have evaluated the performance of GSWABE on a Kepler-based Tesla K40 GPU using both simulated and real short DNA sequence datasets, and have further compared this performance with SSW, MSA-CUDA (the first stage), and gpu-pairAlign, all of which are executed on the same system platform. Our performance evaluation revealed that GSWABE can yield a performance of up to 59.1 GCUPS, 58.5 GCUPS, and 50.3 GCUPS for global, semi-global, and local alignment, respectively. In terms of local alignment, GSWABE achieves a speedup of up to 156.0 over the SSE-based SSW and a speedup of 102.4 over the CUDA-based MSA-CUDA (the first stage). For each alignment type, GSWABE has demonstrated stable and consistent speedups over the CUDA-based gpu-pairAlign, with a maximum speedup of 11.2, 10.7 and 10.6 for global, semi-global and local alignment, respectively. Additionally, GSWABE is able to finish the alignment of one million reads of length 100 bps each in about two days for each alignment type.

DOI: 10.1002/cpe

Albeit designed for all-to-all pairwise alignments, our algorithm can also be easily adopted to pure pairwise alignments with no change of the core code. For example, we have deployed the proposed implementation of local alignment in our open-source CUSHAW2-GPU algorithm [48] to produce final alignments of NGS reads to the reference genome, which are represented in the well established `SAM` format [47]. We also expect that other NGS read aligners (e.g. [49, 50]) and other biological applications (e.g. metagenome clustering and richness estimation [51, 52]) can benefit from our algorithm. In addition, the generality of our algorithm enables the use of our tile-based dynamic programming framework on other accelerator technologies such as Xeon Phi coprocessors. In this regard, employing Xeon Phi coprocessors to accelerate sequence alignment with optimal alignment backtracking can be considered as part of our future work.

ACKNOWLEDGEMENT

We thank our colleague Dr. Jorge González-Domínguez, the editor and the anonymous reviewers for their helpful and constructive comments that helped to improve the manuscript.

REFERENCES

1. Pearson WR, Lipman DJ. Improved tools for biological sequence comparison. *Proceedings of the National Academy of Sciences of the United States of America* 1988; **85**:2444–2448.
2. Altschul SF, Gish W, Miller W, Myers EW, Lipman DJ. Basical local alignment search tool. *Journal of Molecular Biology* 1990; **215**:403–410.
3. Thompson JD, Higgins DG, Gibson TJ. CLUSTALW: Improving the sensitivity of progressive multiple sequence alignment through sequence weighting, position-specific gap penalties and weight matrix choice. *Nucleic Acids Research* 1994; **22**:4673–4680.
4. Liu Y, Schmidt B, Maskell DL. MSAProbs: Multiple sequence alignment based on pair hidden Markov models and partition function posterior probabilities. *Bioinformatics* 2010; **26**:1958–1964.
5. Liu CM, Wong T, Wu E, Luo R, Yiu SM, Li Y, Wang B, Yu C, Chu X, Zhao K, Li R, Lam TW. SOAP3: Ultra-fast GPU-based parallel alignment tool for short reads. *Bioinformatics* 2012; **28**:878–789.
6. Alachiotis N, Berger SA, Stamatakis A. Coupling SIMD and SIMT architectures to boost performance of a phylogeny-aware alignment kernel. *BMC Bioinformatics* 2012; **13**(196).
7. Liu Y, Schmidt B, Maskell DL. CUSHAW: a CUDA compatible short read aligner to large genomes based on the burrows-wheeler transform. *Bioinformatics* 2012; **28**:1830–1837.
8. Lee WP, Stromberg MP, Ward A, Stewart C, Garrison EP, Marth GT. MOSAIK: a hash-based algorithm for accurate next-generation sequencing short-read mapping. *PLoS One* 2014; **9**(e90581).
9. Needleman SB, Wunsch CD. A general method applicable to the search for similarities in the amino acid sequence of two proteins. *Journal of Molecular Biology* 1970; **48**:443–53.
10. Smith T, Waterman M. Identification of common molecular subsequences. *Journal of Molecular Biology* 1981; **147**:195–197.
11. Gotoh O. An improved algorithm for matching biological sequences. *Journal of Molecular Biology* 1982; **162**: 707–708.
12. Qiu J, Ekanayake J, Gunarathne T, Choi JY, Bae SH, Li H, Zhang B, Wu TL, Ruan Y, Ekanayake S, Hughes A, Fox G. Hybrid cloud and cluster computing paradigms for life science applications. *BMC Bioinformatics* 2010; **11**(S3).
13. Aluru S, Futamurab N, Mehrotrac K. Parallel biological sequence comparison using prefix computations. *Journal of Parallel and Distributed Computing* 2003; **63**:264–272.
14. Rajko S, Aluru S. Space and time optimal parallel sequence alignments. *IEEE Transactions on Parallel and Distributed Systems* 2004; **15**:1070–1081.
15. Ligowski Å, Rudnicki W, Liu Y, Schmidt B. *Accurate scanning of sequence databases with the smith-waterman algorithm*, 2011.
16. Sarje A, Aluru S. All-pairs computations on many-core graphics processors. *Parallel Computing* 2013; **39**:79–93.
17. Rognes T. Faster Smith-Waterman database searches with inter-sequence SIMD parallelization. *BMC Bioinformatics* 2011; **12**(221).
18. Wozniak A. Using video-oriented instructions to speed up sequence comparison. *Computer Applications in the Biosciences* 1997; **13**:145–150.
19. Rognes T, Seeberg E. Six-fold speedup of smith-waterman sequence database searches using parallel processing on common microprocessors. *Bioinformatics* 2000; **16**:699–706.
20. Farrar M. Striped Smith-Waterman speeds database searches six times over Other SIMD implementations. *Bioinformatics* 2007; **23**:156–161.
21. Wirawan A, Kwoh CK, Hieu NT, Schmidt B. CBESW: sequence alignment on playstation 3. *BMC Bioinformatics* 2008; **9**(377).

DOI: 10.1002/cpe

22. Szalkowski A, Ledergerber C, Krahenbuhl P, Dessimoz C. SWPS3-fast multi-threaded vectorized smith-waterman for IBM Cell/B.E. and x86/SSE2. *BMC Research Notes* 2008; **1**(107).
23. Liu W, Schmidt B, Voss G, Muller-Wittig W. Streaming algorithms for biological sequence alignment on GPUs. *IEEE Transactions on Parallel and Distributed Systems* 2007; **18**:1270–1281.
24. Manavski SA, Valle G. CUDA compatible GPU cards as efficient hardware accelerators for Smith-Waterman sequence alignment. *BMC Bioinformatics* 2008; **9**(S10).
25. Liu Y, Maskell DL, Schmidt B. CUDASW++: optimizing Smith-Waterman sequence database searches for CUDA-enabled graphics processing units. *BMC Research Notes* 2009; **2**(73).
26. Ligowski L, Rudnicki W. An efficient implementation of smith waterman algorithm on GPU using CUDA, for massively parallel scanning of sequence databases. *2009 IEEE International Symposium on Parallel and Distributed Processing*, Rome, Italy, 2009; 1–8.
27. Liu Y, Schmidt B, Maskel DL. CUDASW++2.0: enhanced Smith-Waterman protein database search on CUDA-enabled GPUs based on SIMT and virtualized SIMD abstractions. *BMC Research Notes* 2010; **3**(93).
28. Khajeh-Saeed A, Poole S, Perot J. Acceleration of the Smith-Waterman Algorithm Using Single and Multiple Graphics Processors. *Journal of Computational Physics* 2010; **229**:4247–4258.
29. Blazewicz J, Frohmberg W, Kierzynka M, Pesch E, Wojciechowski P. Protein alignment algorithms with an efficient backtracking routine on multiple GPUs. *BMC Bioinformatics* 2011; **12**(181).
30. Hains D, Cashero Z, Ottenberg M, Bohm W, Rajopadhye S. Improving CUDASW++, a Parallelization of Smith-Waterman for CUDA Enabled Devices. *2011 IEEE International Symposium on Parallel and Distributed Processing Workshops and Phd Forum*, Alaska, USA, 2011; 490–501.
31. de O. Sandes EF. Retrieving Smith-Waterman alignments with optimizations for megabase biological sequences using GPU. *IEEE Transactions on Parallel and Distributed Systems* 2013; **24**(5):1009–1021.
32. Liu Y, Wirawan A, Schmidt B. CUDASW++ 3.0: accelerating Smith-Waterman protein database search by coupling CPU and GPU SIMD instructions. *BMC Bioinformatics* 2013; **14**(117).
33. Oliver T, Schmidt B, Maskell DL. Reconfigurable architectures for Bio-sequence database scanning on FPGAs. *IEEE Transactions on Circuits and Systems II* 2005; **52**:851–855.
34. Benkrid K, Liu L, Benkrid A. A highly parameterized and efficient FPGA-based skeleton for pairwise biological sequence alignment. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 2009; **17**:561–570.
35. Li TI, Shum W, Truong K. 160-fold acceleration of the Smith-Waterman algorithm using a field programmable gate array (FPGA). *BMC Bioinformatics* 2007; **8**(I85).
36. Liu Y, Schmidt B. SWAPHI: Smith-Waterman Protein Database Search on Xeon Phi Coprocessors. *25th IEEE International Conference on Application-specific Systems Architectures and Processors,* in press, Zurich, Switzerland, 2014; 184–185.
37. Liu Y, Schmidt B, Maskell DL. MSA-CUDA: Multiple Sequence Alignment on Graphics Processing Units with CUDA. *20th IEEE International Conference on Application-specific Systems, Architectures and Processors*, Boston, USA, 2009; 121–128.
38. Myers EW, Miller W. Optimal alignments in linear space. *Computer Applications in the Biosciences* 1988; **4**:11–17.
39. Korpar M, Sikic M. SW#-GPU-enabled exact alignments on genome scale. *Bioinformatics* 2013; **29**:2494–2495.
40. Darling A, Carey L, Feng W. The design, implementation, and evaluation of mpiBLAST. *4th International Conference on Linux Clusters: The HPC Revolution 2003 in Conjunction with ClusterWorld Conference & Expo*, San Jose, USA, 2003; 1–14.
41. Oehmen CS, Baxter DJ. ScalaBLAST 2.0: rapid and robust BLAST calculations on multiprocessor systems. *Bioinformatics* 2013; **29**:797–798.
42. Wu C, Kalyanaraman A, Cannon WR. pGraph: efficient parallel construction of large-scale protein sequence homology graphs. *IEEE Transactions on Parallel and Distributed Systems* 2012; **23**:1923–1933.
43. Liu Y, Schmidt B. Faster GPU-accelerated Smith-Waterman algorithm with alignment backtracking for short DNA sequences. *Lecture Notes in Computer Science* 2014; **8385**:247–257.
44. Zhao M, Lee WP, Garrison EP, Marth GT. SSW library: an SIMD Smith-Waterman C/C++ library for use in genomic applications. *PLoS One* 2013; **8**(e82138).
45. Henikoff S, Henikoff J. Amino acid substitution matrices from protein blocks. *Proceedings of the National Academy of Sciences of the United States of America* 1992; **89**:10915–10919.
46. Dayhoff M, Schwartz R, Orcutt B. A model of evolutionary change in proteins. In *Atlas of Protein Sequence and Structure*, vol. 5, Nat Biomed Res Found, 1978; 345–58.
47. Li H, Handsaker B, Wysoker A, Fennell T, Ruan J, Homer N, Marth G, Abecasis G, Durbin R. 1000 Genome project data processing subgroup. The sequence alignment/map format and SAMtools. *Bioinformatics* 2009; **25**:2078–2079.
48. Liu Y, Schmidt B. CUSHAW2-GPU: empowering faster gapped short-read alignment using GPU computing. *IEEE Design & Test of Computers* 2013; **31**:31–39.
49. Rizk G, Lavenier D. GASSST: global alignment short sequence search tool. *Bioinformatics* 2010; **26**:2534–2540.
50. Langmead B, Salzberg S. Fast gapped-read alignment with bowtie 2. *Nature Methods* 2012; **9**:357–359.
51. Schloss PD, Handelsman J. Introducing DOTUR, a computer program for defining operational taxonomic units and estimating species richness. *Applied and Environmental Microbiology* 2005; **71**:1501–1506.
52. Schloss PD, Westcott SL, Ryabin T, Hall JR, Hartmann M, Hollister EB, Lesniewski RA, Oakley BB, Parks DH, Robinson CJ, Sahl JW, Stres B, Thallinger GG, Van Horn DJ, Weber CF. Introducing Mothur: Open-source, platform-independent, community-supported software for describing and comparing microbial communities. *Applied and Environmental Microbiology* 2009; **75**:7537–7541.

DOI: 10.1002/cpe