Interdiscip Sci Comput Life Sci
DOI 10.1007/s12539-017-0225-8

ORIGINAL RESEARCH ARTICLE

# FPGASW: Accelerating Large-Scale Smith–Waterman Sequence Alignment Application with Backtracking on FPGA Linear Systolic Array

**Xia Fei[1] · Zou Dan[2] · Lu Lina[3] · Man Xin[1] · Zhang Chunlei[1]**

Received: 6 June 2016 / Revised: 21 February 2017 / Accepted: 9 March 2017


**Abstract** The Smith–Waterman (SW) algorithm based on dynamic programming is a well-known classical method for high precision sequence matching and has become the gold standard to evaluate sequence alignment software. In this paper, we propose fine-grained parallelized SW algorithms using affine gap penalty and implement a parallel computing structures to accelerating the SW with backtracking on FPGA platform. We analysis the dynamic parallel computing features of anti-diagonal elements and storage expansion problem resulting from backtracking stage, and propose a series of optimization strategies to eliminate data dependency, reduce storage requirements, and overlap memory access latency. Our implementation is capable of supporting multi-type, large-scale biological sequence alignment applications. We obtain a speedup between 3.6 and 25.2 over the typical SW algorithm running on a general-purpose computer configured with an Intel Core i5 3.2 GHz CPU. Moreover, our work is superior to other FPGA implementations in both array size and clock frequency, and the experiment results show that it can get a performance closed to that of the latest GPU implementation, but the power consumption is only about 26% of that of the GPU platforms.

**Keywords** Smith–Waterman · FPGA · Sequence alignment · Fine-grained parallelization · Hardware accelerator

## 1 Introduction

Biological sequence alignment has become a fundamental task of bioinformatics and modern life science to identify regions of similarity that may be a consequence of functional or structural relationships. The Smith–Waterman (SW) algorithm [1] based on dynamic programming (DP) for pairwise sequence alignment has become a standard method for high precision pairwise sequence alignment, since it was proposed by Smith and Waterman [1]. It has become the core algorithm of multiple sequence alignment, database searching and sequence analysis, and it has been integrated into some well-known sequence alignment tools, such as BLAST [2], ClustalW [3], BWA [4], Bowtie [5], and so on, for exact sequence alignment in different calculation steps.

Although the SW algorithm is efficient in the classical sense, the computing cost is intolerable in execution time and storage requirements with the explosive growth of gene sequence database in recent years. High-performance computer systems equipped with multi-core processors have extensively been used for sequence analysis. However, the parallel efficiency of these systems is constrained by the diversification of program features: bit-wised parallelism, complicated, and multi-dimensional data dependency. Although general-purpose architecture computers have powerful peak performance, efficiently executing the task

Xia Fei and Zou Dan authors have contributed equally to this work.

✉ Xia Fei
xcyphoenix@nudt.edu.cn

[1] Electronic Engineering College, Naval University of Engineering, Wuhan, People's Republic of China

[2] Academy of Ocean Science and Engineering, National University of Defense Technology, Changsha, People's Republic of China

[3] College of Mechatronic Engineering and Automation, National University of Defense Technology, Changsha, People's Republic of China

Published online: 21 April 2017

of bio-sequence analysis on parallel computing platforms would be a challenge.

The compute power and memory bandwidth of algorithm accelerators are superior to general-purpose processors. It has been proved that fully optimized SW algorithm on GPU/FPGA platforms obtains significant speedup compared with its counterpart on CPU platforms. Recently, the use of algorithm accelerators such as FPGA and GPU by exploiting flexible coarse and fine-grained custom design has been proved to be an effective method for the parallel implementation of bioinformatics algorithms. The newly developed SW parallel algorithms are mainly based on system with algorithm accelerators, including FPGAs [6–27] and GPUs [28–30].

The original FPGA implementations [6–11] of SW algorithm are mainly concerned with DNA sequence alignment using linear gap penalty scoring strategy. Splash [6] is an early representative of FPGA hardware accelerator. Then, the Splash2 [7] was 2500 times faster than the SPARC 10 workstation at that time for executing pairwise DNA sequence alignment. In 1993, Fagin et al. [8] reported that an Actel FPGA device operating at 10 MHz clock frequency achieves 15× speedup over IBM RS6000 workstation for executing the same DNA sequence alignment task. In recent years, affine gap penalty [12–15] is widely used for accelerating DNA sequence alignment on FPGA.

Researches in [10, 16, 17] implemented protein pairwise alignment on FPGA. Compared with DNA alignment, the scoring model of protein is more complex, and the storage and access of replacement matrix consumes more logic and storage resources. In 2005, Oliver et al. [13] fitted a SW algorithm core on XC2V6000 chip and both linear and affine penalty models are also supported. The latest implementation also includes [11, 18, 19]. After that, TomVan Court [12] and Benkrid [14] implemented the parameterized SW algorithm accelerator by choosing different sequence types and scoring rules. In addition, Scott Lloyd [20] implemented the scoring and backtracking stage for small scale sequence alignment on FPGA chip, but only support the specific sequence type and scoring model.

In computer industry field, some manufacturers have also implemented the SW algorithm on FPGA, including Paracels GeneMatcher [21], Compugens Bioccelerator [22], SciEngines COPACOBANA 5000 [23] and DeCypher [24] from TimeLogic. Those accelerators can achieve the performance of 50 GCUPS(Giga Cell Updates Per Second) on average. In 2010, Convey announced their HC1 system based on Virtex5 Xilinx chips can achieve a peak performance of 172.8 GCUPS [25]. Then, Erik Vermij [26] achieved 460 GCUPS on the Convey HC1 system composed of four XC5VLX330 FPGA chips for SW sequence alignment without backtracking. However, at the same time, the general-purpose CPUs can only reach the peak performance of 35 GCUPS [27], average value of each thread only reaches 3.75–4.38 GCUPS.

In recent years, there are several new studies [31–36] that focus on short sequence alignment on new generation Virtex-5 or Virtex-6 FPGA-based system. Those literatures only consider the hardware implementation for scoring stage in the SW algorithm (without backtracking) and only support short DNA read mapping less than 512 base pairs, except for the last one [36], which implemented the longer read alignment with the length up to 4096 base pairs.

In summary, compared with CPU, FPGA has obvious advantages in terms of flexibility, scalability, and performance/power consumption ratio. However, most of the hardware implementations of accelerating the SW algorithm only consider the score computing without backtracking, but did not consider the overall acceleration effect for large-scale actual sequence alignment application at all. Moreover, compared to Protein sequence alignment, hardware implementation of DNA sequence alignment is much easier (benefit from smaller data width, smaller storage requirements, more simplified logic design of PE units, since it is only four nucleotides in DNA sequence). The accessing of substitution matrix and scoring of amino acid in protein is much more complicated. Thus, the hardware design complexity for accelerating a similar-sized Protein sequence alignment is much larger than that of DNA sequence.

In GPU research field, Manavski [28] implemented the SW algorithm on NVIDIA GeForce 8800 GTX, and achieved ten times speedup compared with Xeon 3.2 GHz Intel dual core processor. Researches in [29, 37] accelerated multiple sequence alignment application which reported 10–36 times speedup compared with P4 3.0 GHz CPU on GeForce GTX 280 GPU platform. Recently, the SW accelerator researches based on CUDA GPU platform include CUDASW++ [38], CUDASW++ 2.0 [39], CUDASW++ 3.0 [40], CUDA-SWfr [41], GPU-HKA [42], CUDASW++ [43], and so on. They achieved an order of magnitude or higher acceleration performance using a multi-level task parallelization strategy. Especially, the latest implementation, CUDASW++ 3.0 [40], achieves a peak performance of 119 GCUPS on GTX680 card for the SW sequence alignment.

From aspect of computing, the core of sequence alignment is the computation and storage of scoring and backtracking matrices. Especially, the storage expansion caused by backtracking has an important influence on accelerator architecture. However, the current related works based on FPGA are mainly concerned about the hot spots of SW code. Specifically, only pay attention to accelerate the computing of scoring matrix without considering the overall application performance of sequence alignment with backtracking.

In this paper, we focus on sequence alignment with the constrained FPGA resources. The main difference of our heterogeneous architecture with others is that we use the hardware to accelerate the whole Smith–Waterman sequence alignment with traceback and our implementation is capable of processing ultra long sequences due to the task partitioning strategy. We propose fine-grained parallelized SW algorithms using affine gap penalty and implement a parallel computing structures to accelerating the SW with backtracking on FPGA platform. We analysis the dynamic computing features of SW algorithms and propose a series of strategies to eliminate data dependency and reduce storage requirements. We obtain a speedup between 3.6 and 25.2 over the typical SW algorithm running on a general-purpose computer configured with an Intel Core i5 3.2 GHz CPU. Moreover, our work is superior to other FPGA implementations in both array size and clock frequency, and the experiment results show that it can get a performance closed to that of the latest GPU implementation, but the power consumption is only about 26% of that of the GPU platforms. Thus, our implementation achieves the double target of improving overall computing performance while reducing system power consumption.

## 2 Overview of the Smith–Waterman Algorithm

The SW algorithm was proposed to perform local sequence alignment (similarity score/alignment result or both) of two input strings $S_1S_2...S_i...S_M$ and $L_1L_2...L_j...L_N$ with length $M$ and $N$, respectively. The computational complexity is $O(M \times N)$ and the complete calculation process includes two steps: *Computing* and *Backtracking*.

### 2.1 Computing

The SW algorithm iteratively calculates the two-dimensional dynamic programming (DP) matrix $W(i,j)$ for $1 \le i \le M, 1 \le j \le N$, and the recurrence relationship is as follows (affine penalty model):

$$\begin{cases} W(i,0) = \max\left\{0, E(i,j), F(i,j), W(i-1,j-1) + P(S_i, L_j)\right\} \\ E(i,j) = \max\left\{W(i,j-1) - \alpha, E(i,j-1) - \beta\right\} \\ F(i,j) = \max\left\{W(i-1,j) - \alpha, F(i-1,j) - \beta\right\} \end{cases} \tag{1}$$

where $P$ is the character substitution cost table, $\alpha$ and $\beta$ represent the gap opening penalty and the gap-extension penalty, respectively. $W(i,j)$ is the maximum alignment score of two subsequences $S_1S_2...S_i$ and $L_1L_2...L_j$. The initialization condition is $W(i,0) = W(0,j) = E(i,0) = F(0,j) = 0 (1 \le i \le M, 1 \le j \le N)$.

The final score of SW alignment is the maximum value of DP matrix $W$. As shown in Fig. 1a, $W(i,j)$ depends on its upper, left, and upper-left neighbors. It shows that all the element on the same anti-diagonal has no data correlation and can be calculated at the same time. Thus, all elements along the anti-diagonal $k$ can be calculated simultaneously from the anti-diagonals $k$-2, $k$-1, as shown in Fig. 1a.

### 2.2 Backtracking

After computing, backtracking stage starts, as illustrated in Fig. 1b. The starting point is set to the cell in matrix $W$ with the maximum score. The next trace-back position is determined by the directional flag, which indicates that current result was calculated by which direction, upper, left, or upper-left. Then, it moves to the next location until reaches the point with the threshold value (usually set to 0). At last, the backtracking path constitutes the best alignment.

It should be emphasized that backtracking is not required at some time. The advantage of omission backtracking is that it can avoid storage explosion, because only the highest score is concerned. The time cost of backtracking is linear, but the space cost is $O(M \times N)$, because the matrix $W$ and the corresponding path information (matrix $T$) must be recorded for backtracking operation. However, in many cases, users not only care about the similarity score, but also need to get the exact matching results. Thus, the backtracking step is also worthy of more attention.

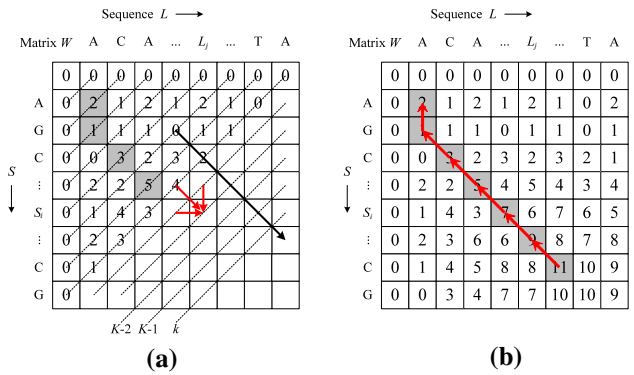


**Fig. 1** **a** Example of the computing stage of the SW algorithm. The *red lines with arrow* represent the data dependency. The wave-front computation is performed along the diagonals one by one from *top-left to the bottom-right*. **b** Example of the backtracking stage of the SW algorithm. The starting point is in the cell with the highest score. In addition, the termination point is preset to the element located in the matrix below the threshold value

## 3 Characteristics of the SW Algorithm

The input of the SW algorithm with backtracking includes two sequences and a character substitution cost table, and the output is similarity score and two extended sequences. The essence of this problem is to construct a set of sequences which makes the similarity measure function to achieve the maximum value. From the point of view of calculation, the core is the computation and storage of scoring and path tracing matrices.

### 3.1 Observation 1: Inter-Task Parallelization

No data dependency is presented in the process of the alignment between different database sequences and a single query. Thus, pairwise sequence alignment can be processed in parallel. The data need to be shared are the query sequence and the substitution cost matrix. The main concern of this parallel implementation focuses on the design of load balance and data partition strategy.

### 3.2 Observation 2: Intra-Task Parallelization

It can be implied from recurrence formula (1) that element $W(i,j)$ depends on its upper neighbor $F(i-1,j)$ $W(i-1,j)$, upper-left neighbor $W(i-1,j-1)$, and left neighbor $E(i,j-1)$ $W(i,j-1)$. There is data synchronization among adjacent anti-diagonals, but there is no data correlation among the cells in the same anti-diagonal. Thus, all elements located in the anti-diagonal $k$ can be calculated simultaneously from anti-diagonals $k$-2, $k$-1, which can be performed in a wave-front style along the diagonal from top-left to bottom-right (Fig. 1a).

The execution time and space complexity of the SW algorithm for two input sequences with size $M$ and $N$ are $O(M \times N)$, and the computing complexity of database sequence alignment is $O(K \times N^2)$ for $K$ sequences with length $N$. In recent years, the size of bio-sequence databases is rapidly expanding. The GenBank now includes over 190 million sequences and more than 200 billion base pairs [44]. Thus, although the SW algorithm is efficient for pairwise sequence alignment, the computational time and space overhead are intolerable on the whole genome database scale (more than $10^9$ bases), especially in contemplation of the case of backtracking.

### 3.3 Observation 3: About Backtracking

Backtracking is an important component of bio-sequence alignment. Before the execution of backtracking, the scoring matrix $W$ and the path information matrix $T$ must be recorded. Data transferring of matrices $W$ and $T$ between internal and external memory will cause storage bandwidth to be the main design bottleneck in the case of resource constrained.

Since the element $W(i,j)$ is valid, it can be known that it was calculated by which of the three adjacent elements and we can use 2-bit binary number (00, 01, 10) to identify the directions. Thus, matrix $T$ can be arranged in the same parallel strategy (in a wave-front mode) as matrix $W$. The computing and storage complexity of backtracking is $O(M+N)$ and $O(M \times N)$, respectively. In addition, it can only be executed serially.

## 4 Parallel Methods

### 4.1 Fine-Grained Parallel SW Algorithm with Traceback

To implement the SW algorithm on FPGA platforms, the original serial flow should be converted into a fine-grained parallel pattern that is suitable for linear systolic array architecture. All PE units perform this hardware algorithm at the same time. The design concept is to compute the anti-diagonal in parallel using linear array architecture consisting of multiple processing elements (PE).

Figure 2 presents the fine-grained parallel SW algorithm in the SPMD (Single Program Multiple-Data) style. The computing process of SW algorithm consists of two stages: *Initialization stage* and *Calculation stage*. The PE array controller pushes the query sequence $S$ into the linear array (S1 and S2) and saves one character before the calculation stage begins, and then, the sequences($L$) in database stream flows through the array, one character per cycle. Each PE obtains a current score and a character in sequence $L$ from the previous PE (S3), calculated alignment score (S4, S5), records path information for backtracking (S6), and then sends current character and score to the next one in one clock cycle (S7). This process repeats until the last character flows pass current PE unit.

### 4.2 Linear Systolic Computing Array

Figure 3 illustrates how each anti-diagonal in matrix $W$ is computed in parallel with a linear PE array. The computing stage begins at clock T0. The query sequence (ACA CAC) has been sent in PE array after initialization phase, and each one holds a character in its register. The first 'A' in subject sequence is delivered into the first PE (PE1) at clock T1. Then, PE1 compares the input character 'A' with its local character (also 'A') and looks up the substitute matrix to calculate the alignment score. The second character 'G' is delivered into PE1 and the previous character 'A' is delivered to PE2 at T2 clock. At that time, PE1 computes character 'A' and 'G', and PE2 compares character 'C' and

**Input:**

*S_in*: current char in *S* sequence;
*L_in*: current char in *L* sequence;
*Score_in*: calculation result by PE[*n*-1];

**Output:**

*S_out*: current char in *S* sequence send to PE[*n*+1];
*L_out*: current char in *L* sequence send to PE[*n*+1];
*Score_out*: calculation result by PE[*n*];

**Variables Define**

*PID*: current PE's identification number; *Shift_cnt*: record how many chars have been delivered;
*H*(*i*-1,*j*-1)/*F*(*i*,*j*)/*E*(*i*,*j*): alignment score calculated from three different locations (diagonal/above/left);
*S_reg*: register char in *S* sequence; *Score_max*: register alignment score calculated by PE[*n*];
Sbt(*S*[*i*],*S*[*j*]) / $\alpha$ / $\beta$: they are parameters in substitution matrix; *F*1/*F*2/*E*1/*E*2: temporary variables;
*Trace_1*/*Trace_2*/*Trace_3*: represent the backtracking locations from diagonal, above and left respectively;
*Trace_back_flag*: backtracking result calculated by current PE;

BEGIN: Parallel Smith-Waterman Algorithm

For all processing element *PID* ∈ {1, 2, ⋯, n} do **in parallel** {

**Initialization:**

S1: *Score_max* ← 0; *Shift_cnt* ← 0; // put the query sequence *S* in PE array

S2: if (*Shift_cnt* = *Seq_length*)

*S_reg* ← *S_in*; // each PE holds one character in its register

else { *S_out* ← *S_in*; *Shift_cnt* ← *Shift_cnt* +1;}

**Calculation:**

S3: load *L_in*, *Score_in* from previous PE's Trans Regs;

S4: *H*(*i*-1, *j*-1) ← *Score_in* + Sbt(*S*[*i*], *S*[*j*]);
*F*1 ← *H*(*i*-1, *j*) - $\alpha$;
*F*2 ← *F*(*i*-1, *j*) - $\beta$;
*E*1 ← *H*(*i*, *j*-1) - $\alpha$;
*E*2 ← *E*(*i*, *j*-1) - $\beta$;

S5: *E*(*i*, *j*) ← Max{*E*1, *E*2};
*F*(*i*, *j*) ← Max{*F*1, *F*2};
*Score_max* ← Max{*H*(*i*-1, *j*-1), *E*(*i*, *j*), *F*(*i*, *j*), 0};

S6: Case (*Score_max* )

*H*(*i*-1, *j*-1): *Trace_back_flag* ← *Trace_1*; // record backtracking location

*E*(*i*, *j*): *Trace_back_flag* ← *Trace_2*;

*F*(*i*, *j*): *Trace_back_flag* ← *Trace_3*;

S7: *L*_out ← *L*_in; Score_out ← Score_max; // send current char and score to the next PE

}

END

**Fig. 2** Fine-grained parallel SW algorithm with traceback for each PE

'A'. From the point of view of matrix computing, we can see that the first anti-diagonal [the shaded cell (1,1)] is calculated at T1 within PE1, the second anti-diagonal [composed of cell (1,2) and (2,1)] is calculated at T2 within PE1 and PE2, and then the third one [composed of cell (1,3), (2,2), and (3,1)] is calculated at T3 within PE1, PE2, and PE3. The computing of matrix $T$ is complete synchronization with $W$, which is arranged in the same wave-front mode.

The sequence in database (AGCACA, for example) flows through the PE array with clock cycle step by step. Each PE computes a pair of characters and all the PE units perform the computation of an anti-diagonal in parallel. The matrix $W$ can be computed in $M + N$ cycles, where $M$ and $N$ represent the size of two input sequences, respectively. The overall computing overhead of the SW algorithm includes the time of array initialization ($M$ cycles), the execution time for database stream flow through the array ($M + N$ cycles), and the time for data loading and section advance if the length of query sequence ($M$) exceeds the PE number. The overhead of the latter is $O(L)$, thus the computing overhead of the fine-grained parallel SW algorithm running on FPGA array structure can be simplified approximately as $(2M + N) \times T$, where $T$ is the clock period of linear systolic array.

## 5 Parallel Implementation of SW Algorithm

### 5.1 System Architecture with Traceback

FPGA is a large-scale integrated circuit chip that can be programmed in the field after manufacture. Our pairwise sequence alignment system consists of a reconfigurable FPGA board and a host computer, as illustrated in Fig. 4. The FPGA board is acted as the algorithm accelerator, which is built based on one FPGA chip (Xilinx Virtex7 XC7VX485T), three DDR3 1600 8 GB DRAM modules, and an SFP+ optical interface. The DDR3 DRAM Memory Controllers are implemented in FPGA chip, which directly

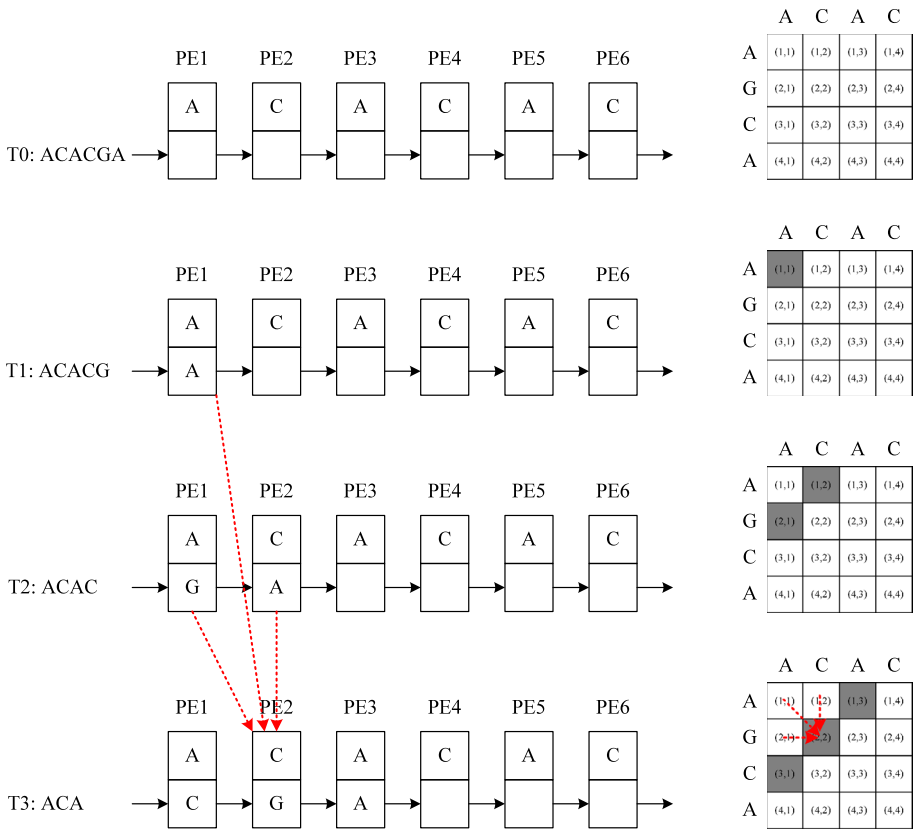


**Fig. 3** Illustration of the SW algorithm for alignment score calculation

control the I/O of the DRAM chips. The sequence database, alignment results, and backtracking path (matrix $T$) are stored in the DDR3 DRAMs.

The SW algorithm accelerator is mainly consist of a *Data interface* (SFP+ optical interface), *PE Array Controller*, Linear *PE array* consisting of multiple PEs, *IO Channels*(including IO Channel Controller and three independent memory access controllers), and *Data Back-writing & Backtracking Controller* integrated a data buffer. The FPGA GTX Transceivers are used as the *Data interface*, which load sequence database from host and store it to external memory. The *PE Array Controller* is in charge of data initialization, configuration command generation, dynamic task partitioning, and return similarity score to the host. The linear *PE Array* holds the query string (a character per PE) and computing those cells at the same anti-diagonal in a wave-front mode when the database sequence travels through the PE array. Two characters (i.e., $S_i$ and $L_j$, in query and subject sequence) and the current alignment score from the previous PE are used as the input signal of each PE. The output of the PE consists the up-to-date score and character $L_j$ which is going to be delivered to the next PE. The structure of the PE is illustrated in Fig. 4. IO Channels Controller consists of a finite state machine and multiple parallel data paths. The purpose of using multiple independent memory controllers and DDR3 DRAM modules is to increase memory access bandwidth, so that the computation and back-writing operations can be overlap.

The Data Back-writing & Backtracking Controller is in charge of gathering path information, computing synchronizing, and save data (matrix $T$ and the most right column of the current computing area in matrix $W$, which is going to be reused for computing the next section) back to external DRAM. The storage overhead of the backtracking path is hidden in the computation overhead. When the calculation is complete, it starts the backtracking process and writing the alignment results back to DRAM. Multiple PE modules fill the score matrix in parallel. Each PE fills one column in matrices $T\&W$ following the corresponding recurrence (1). Every PE is configured with a block memory for storing the BLOSUM62 substitution matrix, and a data buffer for saving the data (including path information, Max/Min score and corresponding position for backtracking). The transitive registers are set between adjacent PEs to deliver sequence characters and middle results for matrix computing.

The Intermediate result memory groups are designed to save the result calculated by PE modules, which composed of multiple independent memory modules with dual port. One of them is connected to the last PE and the first one, and dedicated to store the data in the most right column generated by the last PE. Then, it can be accessed by

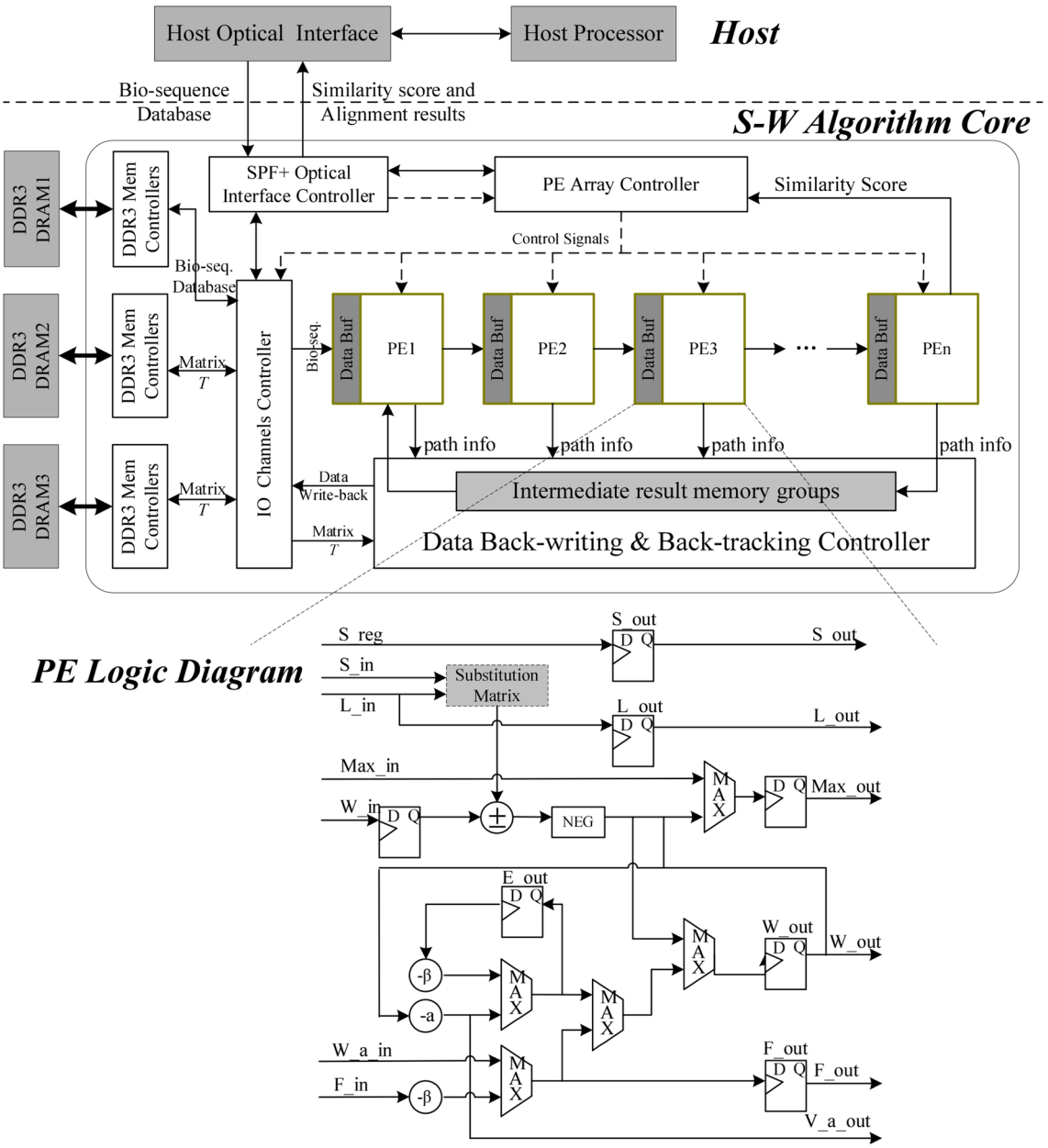


**Fig. 4** Design of the SW parallel algorithm and the structure of PE core with affine penalty model

the first PE. Thus, it is actually formed a cycle PE chain structure.

### 5.2 Rotary Task Partitioning

The standard SW algorithm fills the matrix element one by one from top-left to the bottom-right, which is shown in Fig. 1. This implementation cannot be directly applied on the FPGA platform. First, the whole matrices *W&T* cannot be saved in the limited on-chip memory. Second, it is impossible to implement enough PEs to perform the calculation of the whole anti-diagonal in parallel contemporarily, given the limited on-chip logic resource.

To minimize the storage requirements, we divide the matrix *W* into small sections following the longitudinal direction. Each section is composed of *P* columns, where *P* is the PE number. We computes the scores of those sections individually from left to right. As illustrated in Fig. 5, the matrix *W* is divided into three sections and the free space in the last section is filled with empty characters. Each section is composed of *P* columns. The red lines with arrow represent the filling direction of matrix elements and the numbers 1, 2, 3⋯ indicate subsequent

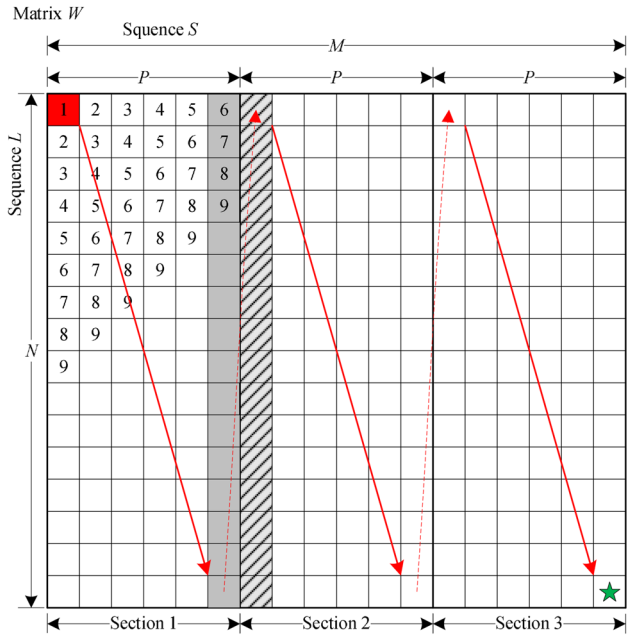


**Fig. 5** Work mapping and the computing sequence of systolic array consisting of $P$ computing units

computing stages. As implied by observation 2, current section is divided into $P$ columns and each column is mapped to one PE. The filling process begins from the cell at the top-left corner, which is indicated as Sect. 1 (marked in red color). The wave-front calculation is carried out along anti-diagonals from the top-left to the bottom-right until it reaches the lower right corner of Sect. 1. After that, Sect. 2 is mapped to the PE array. The initial calculation position goes back to the upper-left corner and it is calculated in the same manner. This loop executes recurrently until the last element (the position marked by a green star) in the last section is computed. Similarly, the matrices $T$ are computed in the same way.

According the data dependencies described by recurrence (1), the first column (with slash gray fills) in the next section depends on the most right column (with gray fills) in the previous section. Thus, the results generated by the last PE in the array should be stored and it will be reused by the first PE for computing the next section. We take the protein sequence with the length of 64 K amino acid characters as an example, the maximum score in the BLOSUM62 substitution matrix was 11; therefore, the data width of the intermediate result memory groups is at least 20 bit ($2^{20} > 64\,\mathrm{K} \times 11$). Thus, the storage requirements is $20 \times 64\,\mathrm{K}$=1.25 Mb. At present, the storage capacity of high-end FPGA chip is generally more than 20 Mb, so the intermediate result storage caused by rotary task partitioning will not become performance bottleneck.

## 5.3 Backtracking

Since the current element can only be derived from three adjacent positions, we can use 2-bit binary number to identify this direction (e.g., 00, 01, and 10 represents left, up, and up-left path flag, respectively). Suppose, two input string $S$ and $L$ with length $M$ and $N$, respectively. The storage requirement for the backtracking path is $2M \times N$. It will exceed the upper limit of storage capacity of the latest generation of FPGA chip when input sequence size is larger than 1K. Therefore, the path information matrix $T$ should be moved to external DRAM. Compared with the SW algorithm without backtracking, the biggest challenge problem of FPGA implementation is the storage and loading of the path information matrix $T$, the memory access bandwidth of external memory is the major design bottleneck.

### *5.3.1 Backtracking Path Storage*

The memory bandwidth requirement of the backtracking path is proportional to the size of the PE array. The storage requirement of each element in matrix $T$ is 2-bit, so the data size generated by PE array per clock cycle is $2P$-bit. Where, $P$ is PE array size. Suppose, the clock frequency of PE array is $F$, and the effective memory bandwidth is $B_d$, the memory access bandwidth requirement ($B_t$) for moving matrix $T$ to external equals $2PF$ bit/s. In order to ensure the overlap between data back-writing and computing overhead, $B_t$ should be less than or equal to $B_d$, that is $B_t \leq B_d$. Therefore, $P \leq B_d/2F$.

Suppose the effective bandwidth of single DDR3-1600 module is 9.6 GB/s (conservative estimate), and the frequency of PE array is 100 MHz. The PE array size P must be less than 384. In addition, it can be doubled, reaching 768 PEs, if a dual channel memory scheme is adopted. Even so, the array size is also smaller than the actual sequence length involved in alignment. Therefore, we adopt the rotated task allocation strategy, as shown in Fig. 5.

Suppose the length of the input string $S$ and $L$ is $M$ and $N$, respectively, and $P < M < N$. Then, we compute the sub matrix $W$' with the size of $N \times P$, while generate the sub matrix $T$' with the same size each time. If the size of the PE array $P = 4r$, to make the storage order consistent with the data generation, we adopt the byte merging method to improve the efficiency of memory write operation. As shown in Fig. 6, we merge the four adjacent elements in each row of the sub matrix $T$' into one byte, the address of each byte is in ascending then store the merged data into external memory.

The element coordinates $(x, y)$ in the backtracking path matrix $T$, the data address in external memory $A_{x,y}$, and the position offset in bytes meets the following relationship:

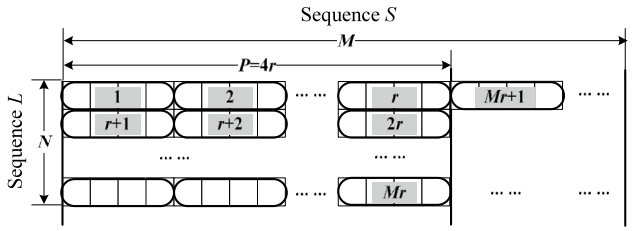


**Fig. 6** Address mapping relation of the elements in path information matrix $T$

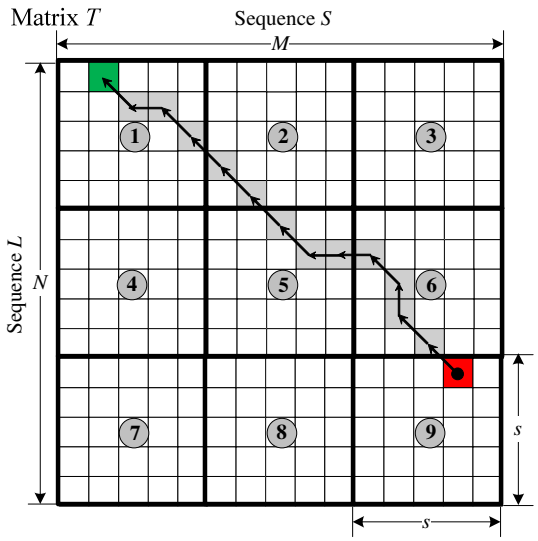


**Fig. 7** Data partitioning and prefetching of the backtracking path matrix $T$

$$\begin{cases} x = \frac{4A_{x,y}}{P} \bmod N + 1 \\ y = \left\lfloor \frac{4A_{x,y}}{NP} \right\rfloor \times P + A_{x,y} \bmod \frac{4}{P} + O_{x,y} \end{cases}. \tag{2}$$

*5.3.2 Backtracking Path Loading*

Because of the obvious spatial locality in the process of backtracking, we divide the path information matrix $T$ into a number of sub matrices with the length of a side $s$, and then load the data into FPGA. A prefetching scheme is adopted to load the data block to be used into FPGA on-chip memory in backtracking stage so as to hide the memory access latency. As shown in Fig. 7, matrix $T$ is divided into nine sub matrices with a scale of $s \times s$, and the numbers 1, 2, 3$\cdots$ are used to represent sub matrices.

For example, suppose that the backtracking starting point is located in the sub matrix #9 (filled in red color). First, access to this cell. Then, determine the location of the next backtracking point according to the path flags (00, 01, and 10 represent left, up, up-left paths, respectively, and 11 represents the threshold value, usually set to 0) stored in the current cell. At the same time, the three adjacent sub matrices #5, #6, and #8 are prefetched into internal memory. Once the backtracking point moves into the sub matrix # 6, the space occupied by data block #8 is released immediately, because this area will definitely not be accessed. At the same time, the Backtracking Controller starts prefetching the data blocks #2, #3, and #5. In fact, only #2 and #3 need to be prefetched, since block #5 is adjacent to block # 9, which has been loaded already. This loop executes recurrently until the termination condition (path flag=11) is satisfied. As shown in Fig. 7, the position filled in green color represents termination point of backtracking. At last, all the positions that backtracking point arriving construct the backtracking path (gray squares with arrow marks).

In the data block with the size of $s \times s$, the shortest backtracking path is $s$, and the longest path is $2s - 1$. Assuming the working frequency of DDR3 controller is 200 MHz with 128-bit data width, and the frequency of backtracking controller is $F$ ($F$=200 MHz in our implementation), the clock cycles for each data block prefetching $T_T$ is $T_T = 2s^2/128$. If the initialization of memory controller requires 10 cycles, the prefetching overhead $T_Y$ equals $T_Y = 3 \cdot (T_T + 10)/200$ (up to three data blocks need to be prefetched each time).

In each data block, the shortest backtracking path is the main diagonal of matrix. Assuming the backtracking operation for each location requires 2 clock cycles, the minimum overhead of backtracking ($T_B$) is $2s/F$. To ensure the overlap between data prefeching and backtracking overhead, the maximum overhead for data block prefetching ($T_Y$) should less than the minimum overhead of data block backtracking ($T_B$), that is $T_Y < T_B$. Therefore, $s < 40$. To facilitate address conversion, $s$ is set to 32 in our implementation.

## 6 Evaluation

### 6.1 Experimental Design

The evaluation platform of our fine-grained parallel SW accelerator mainly consists of one large-scale Xilinx Virtex7 XC7VX485T FPGA chip and three 8GB DDR3-1600 DRAM units. An SFP+ optical interface is implemented to provide 9.6 GB/s data transmission bandwidth with host computer for each DRAM unit, which is built on 7 Series FPGA GTX Transceivers. Our RTL design is implemented in Verilog and compiled by Xilinx Vivado 14.1. Our accelerator is capable to be reconfigured dynamically. Users are able to quickly switch the FPGA's logic function among several pre-designed parallel SW algorithm versions according to current sequence type, database size, and the

largest sequence length. The reconfiguration time is less than 60 ms, which shows that the reconfiguration efficiency is improved by 2–3 orders of magnitude than traditional JTAG-based methods.

The original SW software implementation is originated from ClustalW 2.0 [45]. The query sequences are randomly generated, whose lengths range from 128 to 8192. The reference sequence database is the Swiss-Prot database (release 2012_11). We use BLOSUM62 as the substitution matrix. The gap-open penalty and gap-extension penalty are set as 30 and 6, respectively. The widely used performance metric of SW algorithm is GCUPS (Giga cell updates per second), where cell represents the amount of work for calculating one unit of the score matrix. The time overhead for our FPGA accelerator consists of the time of calculation and backtracing, the loading time for query sequence and reference database, and the transmitting time for score and backtracing result. All the evaluations were carried out on a host computer configured with one Intel Core i5 3.2 GHz CPU, 8GB memory and Ubuntu 13.01 operation system. All the GPU-based evaluations were performed on the NVidia Geforce GTX680 GPU.

### 6.2 FPGA Resource Utilization

Using Xilinx XC7VX485T FPGA as target, we perform the synthesis of our SW algorithm accelerating engine to evaluate the resource utilization. As illustrated in Table 1, the SW algorithm accelerator array consists of 512 PEs consumes 57,870 slices and 896 block RAMs (28 Mbits memory). The memory resource utilization rate is over 87%, most of which are used by the substitution matrix and result buffer for PE array.

On the basis of scalability-oriented system design, our SW accelerator is able to reach a post-place and route clock frequency of over 200 MHz. It is conservatively estimated that 1024 PE units can be implemented on one Xilinx XC7VX1140T FPGA. Considering the task partitioning strategy descripted in section 5.2, the size of input sequence has no limits. This means that our FPGA-based SW accelerator can support larger-scaled sequence alignment applications.

Compared to software backtracking scheme [36], it seems that our implementation consumes more hardware resources, and trace-back operation is not required for every sequence alignment (typically only those with a significantly high similarity score). However, in practical, there will be a lot of comparison results with (non)optimal local alignment scores need to be considered. We can separate the score computing and path backtracking using global dynamic reconfiguration strategy. Thus, backtracking will not be a burden to hardware design, and it can improve the utilization of hardware resources.

As XC7VX1140T chip is not available for our group, we validate our FPGA design with a comprehensive simulation and verification solution using the Xilinx EDA tool Vivado v14.1. The Xilinx EDA tools guarantee the consistency of the simulated performance and the actual performance. The FPGA synthesis shows that 1024 PE units could be implemented on one Xilinx XC7VX1140T FPGA. With the integrated circuit continued to improve, a larger PE array can be implemented on one FPGA chip and the performance will be improved by hardware accelerator in the future.

### 6.3 Performance Comparison with FPGA Platforms

Main technical specifications of several typical SW algorithm accelerators based on linear systolic array architecture are shown in Table 2. We get the following conclusions: (1) compared to hardware SW alignment with backtracking [12, 14, 20], our implementation is able to implement larger PE array and obtains higher frequency and performance. (2) Compared to other hardware SW implementations without backtracking, our comprehensive computing performance is also superior to others obviously. It should be explained that [31] and [33] integrated several small-scaled Score Processing Units for multiple DNA Short-Read Mapping tasks in parallel on a single FPGA chip. First, no data dependency exists among multiple coarse-grained tasks for short-read mapping. Second, the hardware design complexity for DNA alignment is significantly smaller than that of the protein. As a result, a larger linear PE array can be fitted in a single device. Thus, the theoretical peak performance (equals array size $\times$ clock frequency $\times$ the number of score processing units) achieved in paper [31] and [33] is slightly higher than our implementations.

**Table 1** FPGA utilization summary (hardware SW with traceback)

| FPGA | Array size | Slices/(%) | Memory/(%) | Frequency (MHz) |
|---|---|---|---|---|
| XC7VX485T | 512 PE | 57870/(76%) | 896 BRAM/(87%) | 200 |
| XC7VX1140T[a] | 1024 PE | 103676/(58%) | 1664 BRAM/(88%) | 200 |

[a]Synthesis result, untested on real accelerator platform

**Table 2** Compare the performance (GCUPS) with related FPGA implementations

| Related works | Backtracking (yes or no) | PE number in array | Frequency (MHz) | Performance (GCUPS) |
|---|---|---|---|---|
| Oliver [13] | N | 252 | 55.0 | 13.9 |
| Altera [18] | N | 384 | 66.7 | 25.6 |
| Olaf [19] | N | 128 | 125 | 16.0 |
| Van [12] | Y | 303 | 77.5 | 23.5 |
| Khaled [14] | Y | 168 | 62.5 | 10.5 |
| Scott [20] | Y | 256 | 100 | 25.6 |
| DNAlign [31] | N | 100 | 111 | $11.1 \times 12$ |
| SMap [32] | N | – | 250 | Not mentioned |
| SRM [33] | N | <100 | 125 | $16 \times 8$ |
| SRM [34] | N | 100 | 175 | 17.5 |
| LRM [35] | N | 200 | 200 | 40 |
| SLA [36] | N | 128 | 60 | 7.62 |
| Ours[a] | Y | 512 | 150 | 76.8 |
| Ours[b] | Y | 512 | 200 | 105.9 |

[a]Only supports short sequence alignment with the max length less than 512 chars. The path information matrix $T$ is stored in FPGA on-chip memory (so-called backtracking on-chip)

[b]Without limitation to the length of two input sequences theoretically. The path information matrix $T$ for backtracking is stored in external DRAM (so-called backtracking off-chip)

### 6.4 Comparison with CPU and GPU Platforms

With the development of the current large-scale computing platform, the power dissipation wall is a problem that many researchers pay more and more attention to. We compare the implementations of SW sequence alignment in different computing platforms using performance and the ratio of overall performance and power consumption (GCUPS/Watt).

Due to the existing real-time measurement software of power consumption such as CPU EVEREST Ultimate cannot reflect the actual power consumption of computing system, a galvanometer (type HIOKI 3290) is used in our experiment to detect and measure electric current. The power consumption is calculated in the following steps: (1) we record the idle electric current of host platform without accelerator. (2) We record the electric current of the complete system working at computing state for three configurations, including the CPU stand-alone system, CPU–GPU heterogeneous system and CPU-FPGA heterogeneous system. (3) We record the electric current introduced by the SW alignment task by calculating the difference between the current of the idle state and that of the computing state. (4) We calculate the power consumption according to the formula of voltage × current × power coefficient. (5) We obtain the performance per Watt. Main performance specifications of several typical implementations to accelerate the SW algorithm and our evaluation are shown in Table 3. The column titled 'Speedup' is filled with speedup ratio of the specified implementation compared with ours.

**Table 3** Performance & Power consumption comparison of typical CPU, GPU and our FPGA accelerator

| Related works | Platforms | GCUPS | Speedup | Watt | GCUPS/W |
|---|---|---|---|---|---|
| SSEARCH [46] | CPU | 4.2 | 25.2 | 22 | 0.19 |
| SWIPE [47] | CPU | 29.1 | 3.6 | 25 | 1.16 |
| SW-CUDA [28] | GPU | 4.5 | 23.5 | 170 | 0.03 |
| CUDASW++ [38] | GPU | 9.7 | 10.9 | 173 | 0.06 |
| CUDASW++ 2.0 [39] | GPU | 18.1 | 5.9 | 171 | 0.11 |
| CUDASW++ 3.0 [40] | GPU | 106.2 | 1 | 182 | 0.58 |
| CUDA-SWfr [41] | GPU | 4.9 | 21.6 | 174 | 0.03 |
| GPU-HKA [42] | GPU | 20.3 | 5.2 | 179 | 0.11 |
| CUDASW++ [43] | GPU | 15.8 | 6.7 | 172 | 0.09 |
| SW-ITE [30] | GPU | 7.3 | 14.5 | 178 | 0.04 |
| Ours | FPGA | 105.9 | 1 | 44 | 2.41 |

As listed in Table 3, our FPGA-based system obtains a 3.6–25.2 speedup ratio compared with CPU and 1–23.5 compared with GPU. The power consumption generated by SW workload of CPUs ranges from 22 to 25 W and that of GPUs ranges from 170 to 182 W. In contrast, our FPGA platform produces 44W power consumption, saving more than 74% compared with GPU. Although the performance of the latest GPU implementation CUDASW++ 3.0 [40] is slightly higher than that of us, considering the factor of power consumption (more than 4 times as much as us), as for the measurement unit, GCUPS/W ($10^6$ CUPS/W), our FPGA platform achieves a factor of nearly 4.2× over CUDASW++ 3.0 for accelerating sequence alignment application.

It is worth mentioning that the CPU execution time is increasing very fast with the growth of query size during our evaluation. The reason is the cache miss rate of CPU increases with sequence size [48, 49]. In more detail, the look-up table operation of substitution matrix produces a lot of discrete fine-grained memory access, while the size of DP matrix row exceeds the cache capacity. Higher cache miss ratio means higher memory access overhead. The time on our accelerator increases very slowly with the growth of query size during our evaluation. This is due mainly to execution overhead of our accelerator equals to the time of array initialization + database stream flow through the array + array pause time. In addition, with the carefully designed data reuse mechanism and overlap of memory

access and computing, the memory access overhead is minimized.

## 7 Conclusions

Sequence alignment is the basis of biological sequence analysis. The SW algorithm is a classical algorithm and gold standard to evaluate other algorithms in the field of pairwise sequence alignment, which has been implemented on a range of platforms, including CPUs, GPUs, and FPGAs. The CPU-based SW implementations take advantage of the SIMD unit to accelerate the alignment process. Most GPU-based SW implementations [28, 29, 37] adopt intra-task parallelization and focus on memory access optimization. All FPGA-based SW implementations [6–20] use the systolic array architecture as the basis for their designs. However, those implementations either simplify the scoring rules, or only take into account the scoring matrix calculations, or only support small scale sequence alignment with specific sequence type and scoring model.

In this paper, we propose fine-grained parallelized SW algorithms using affine gap penalty and implement a parallel computing structures to accelerating the SW with backtracking on FPGA platform. We analysis the dynamic parallel computing features of anti-diagonal elements and storage expansion problem resulting from backtracking stage, and propose a series of optimization strategies to eliminate data dependency and reduce storage requirements. Our implementation is capable of supporting multi-type, large-scale biological sequence alignment applications with backtracking and it can basically meet the computing demand of pairwise sequence alignment.

The evaluation shows that our implementation obtains a 3.6–25.2 speedup over the typical SW algorithm on a multi-core CPU platform with Intel Core i5 3.2 GHz Quad CPU. Our work is superior to other FPGA implementations in both array size and clock frequency, and the GCUPS performance is comparable to that of the latest GPU implementation [40]. Nevertheless, the FPGA accelerator only produces 44 W power consumption, which is only about 26% of that of the state-of-the-art GPU platforms. Thus, our implementation achieves the double target of improving overall computing performance while reducing system power consumption. Moreover, the computing performance of our FPGA-based SW algorithm can be almost linearly increased when using multiple FPGA chips. With the development of FPGA technology, our application-specific parallel accelerator design is promising to resolve the massive bio-sequence analysis problems.

**Acknowledgements** This research is partially supported by the National Natural Science Foundation of China (61572515, 61502516, 61501484) and China Postdoctoral Science Foundation (2016M593023).

## References

1. Smith TF, Waterman MS (1981) Identification of common molecular subsequences. J Mol Biol 147(1):195–197
2. Altschul SF, Gish W, Miller M, Myers EW, Lipman DJ (1990) Basic local alignment search tool. J Mol Biol 215(3):403–410
3. Thompson J, Higgins D, Gibson T (1994) Clustalw: improving the sensitivity of progressive multiple sequence alignment through sequence weighting position specific gap penalties and weight matrix choice. Nucleic Acids Res 22(22):4673–4680
4. Li H, Durbin R (2009) Fast and accurate short read alignment with Burrows-Wheeler transform. Bioinformatics 25(14):1754–1760
5. Langmead B, Trapnell C, Pop M, Salzberg SL (2009) Ultrafast and memory-efficient alignment of short DNA sequences to the human genome. Genome Biol 10(3):R25
6. Hoang DT (1992) A systolic array for the sequence alignment problem, Technical Report, Brown University Providence, RI, USA
7. Hoang DT (1993) Searching genetic databases on splash 2. Proc IEEE Workshop FPGAs Custom Comput Mach 1993:185–191
8. Fagin B, Watt JG, Robert G (1993) A special-purpose processor for gene sequence analysis. Comput Appl Biosci 9:221–226
9. Yamaguchi Y, Maruyama T et al (2002) High speed homology search with FPGAs. Proc Pac Symp Biocomput 2002:271–282
10. Dydel S, Bala P (2004) Large scale protein sequence alignment using FPGA reprogrammable logic devices. Proc IEEE Int Conf Field Program Logic Appl 2004:23–32
11. Sahoo B, Padhy S (2009) A reconfigurable accelerator for parallel longest common protein subsequence algorithm. Proc IEEE Int Adv Comput Conf 2009:260–265
12. VanCourt T, Herbordt MC (2007) Families of FPGA-based algorithms for approximate string matching. J Microprocess Microsyst 31:135–145
13. Oliver T, Schmidt B, Maskell D (2005) Hyper customized processors for bio-sequence database scanning on FPGAs. In: Proceedings of ACM/SIGDA 13th international symposium on field programmable gate arrays, pp 229–237
14. Benkrid K, Liu Y, Benkrid A (2009) A highly parameterized and efficient FPGA-based skeleton for pairwise biological sequence alignment. IEEE Trans Very Large Scale Integr 17(4):561–570
15. Gok M, Yilmaz C (2006) Efficient cell designs for systolic Smith-Waterman implementation. In: Proceedings of 16th international conference on field programmable logic and applications, pp 1–4
16. Moritz GL, Jory C, Lopes HS, Lima CRE (2006) Implementation of a parallel algorithm for protein pairwise alignment using reconfigurable computing. In: Proceedings of IEEE international conference on reconfigurable computing and FPGAs (ReConfig'06), pp 1–7
17. Marongiu A, Palazzari P, Rosato V (2003) Designing hardware for protein sequence analysis. Bioinformatics 19(14):1739–1740
18. Zhang P, Tan G, Gao GR (2007) Implementation of the Smith-Waterman algorithm on a reconfigurable supercomputing platform. In: Proceedings of 1st international workshop on high-performance reconfigurable computing technology and applications, pp 39–48

19. Storaasli O, Yu W, Strenski D, Maltby J (2007) Performance evaluation of FPGA-based biological applications. In: Proceedings of cray users group, Seattle
20. Lloyd S, Snell QO (2008) Sequence alignment with traceback on reconfigurable hardware. In: Proceedings of the 2008 international conference on reconfigurable computing and FPGAs (ReConFig'08), pp 259–264
21. Paracel Corporation (2004) Applied high-performance computing, revision 6.1. http://www.paracel.com. Accessed 20 Mar 2016
22. Compugen Ltd. (1994) The bioccelerator for GCG users. Mol Biotechnol 2(2):205. doi:10.1007/BF02824817
23. Pfeiffer G, Baumgart S, Schroder J, Schimmler M (2009) A massively parallel architecture for bioinformatics. In: Proceedings of the 9th international conference on computer science, pp 994–1003
24. TimeLogic Ltd. (2010) TimeLogic Biocomputing Engine Introduction. http://www.timelogic.com/decypher_intro.html
25. Convey (2010) Convey website. http://www.convey.com
26. Vermij E (2011) Genetic sequence alignment on a supercomputing platform, MS Thesis, TU Delft, Netherlands
27. Aldinucci M, Meneghin M, Torquati M (2010) Efficient Smith-Waterman on multi-core with fastflow. J Parallel Distrib Netw Based Proc PDP 1(3):195–199
28. Manavski SA, Valle G (2008) CUDA compatible GPU cards as efficient hardware accelerators for Smith-Waterman sequence alignment. BMC Bioinf 9(2):10–19
29. Liu Y, Schmidt B, Maskell DL (2009) Parallel reconstruction of neighbor-joining trees for large multiple sequence alignments using CUDA. In: Proceedings of IEEE international symposium on parallel and distributed processing, pp 1–8
30. Ligowski L, Rudnicki W (2009) An efficient implementation of Smith-Waterman algorithm on GPU using CUDA, for massively parallel scanning of sequence databases. In: Proceedings of the 23rd IEEE international parallel and distributed processing symposium (IPDPS '09), pp 1–8
31. Alachiotis N, Berger SA, Stamatakis A (2011) Accelerating phylogeny-aware short DNA read alignment with FPGAs. In: IEEE international symposium on field-programmable custom computing machines, pp 226–233
32. Olson CB, Kim M, Clauson C, Kogon B (2012) Hardware acceleration of short read mapping. IEEE Int Symp Field Program Custom Comput Mach 282(1):161–168
33. Preuber TB, Knodel O, Spallek RG (2012) Short-read mapping by a systolic custom FPGA computation. IEEE Int Symp Field Program Custom Comput Mach 282(1):169–176
34. Tang W, Wang W, Duan B et al (2012) Accelerating millions of short reads mapping on a heterogeneous architecture with FPGA accelerator. IEEE Int Symp Field Program Custom Comput Mach 282(1):184–187
35. Chen P, Wang C, Li X, Zhou X (2014) Accelerating the next generation long read mapping with the FPGA-based system. IEEE/ACM Trans Comput Biol Bioinf 11(5):840–852
36. Sebastiao N, Roma N, Flores P (2012) Integrated hardware architecture for efficient computation of the n-Best bio-sequence local alignments in embedded platforms. IEEE Trans Very Large Scale Integr VLSI Syst 20(7):1262–1275
37. Liu Y, Schmidt B, Maskell DL (2009) MSA-CUDA: multiple sequence alignment on graphics processing units with CUDA. In: Proceedings of the 20th IEEE international conference on application specific systems, architectures and processors, pp 121–128
38. Liu Y, Maskell D, Schmidt B (2009) CUDASW++: optimizing Smith-Waterman sequence database searches for CUDA-enabled graphics processing units. BMC Res Notes 2(1):73–82
39. Liu Y, Schmidt B, Maskell DL (2010) CUDASW++2.0: enhanced Smith-Waterman protein database search on CUDA enabled GPUs based on SIMT and virtualized SIMD abstractions. BMC Res Notes 3(1):93. doi:10.1186/1756-0500-3-93
40. Liu Y, Wirawan A, Schmidt B (2013) CUDASW++ 3.0: accelerating Smith-Waterman protein database search by coupling CPU and GPU SIMD instructions. BMC Bioinf 14(1):117
41. Liu Y, Hong Y, Lin CY, Hung CL (2015) Accelerating Smith-Waterman alignment for protein database search using frequency distance filtration scheme based on CPU-GPU collaborative system. Int J Genom Hindawi Publ Corp 2015:12. doi:10.1155/2015/761063
42. Hasan L, Kentie M, Al-Ars Z (2011) GPU-accelerated protein sequence alignment. In: Proceedings of the 33rd annual international conference of the IEEE engineering in medicine and biology society (EMBS '11), pp 2442–2446
43. Hains D, Cashero Z, Ottenberg M, Bohm W, Rajopadhye S (2011) Improving CUDASW++, a parallelization of smithwaterman for CUDA enabled devices. In: Proceedings of the 25th IEEE international parallel and distributed processing symposium, workshops and Phd forum (IPDPSW'11), pp 490–501
44. NCBI (2016) GenBank growth statistics. http://ww.ncbi.nlm.nih.gov/genbank/statistics/. Accessed 21 Aug 2016
45. ClustalW 2.0 (2016) Center of bioinformatics. http://www.cbi.pku.edu.cn/chinese/documents/bioinfor/overview/web4/1.html. Accessed 21 Aug 2016
46. Farrar M Striped Smith–Waterman speeds database searches six times over other SIMD implementations
47. Rognes T (2011) Faster Smith-Waterman database searches with inter-sequence SIMD parallelization. BMC Bioinf 12:221
48. Jaleel A, Mattina M, Jacob B (2006) Last level cache (LLC) performance of data mining workloads on a CMP-a case study of parallel bioinformatics workloads. In: Proceedings of the twelfth international symposium on high-performance computer architecture (23):88–98
49. Wang D, Tang ZM (2004) The implementation and analysis of Smith-Waterman algorithm on systolic array. Chin J Comput 27(1):12–20