Accelerating the Smith-Waterman Algorithm Using Bitwise Parallel Bulk Computation Technique on GPU

Takahiro Nishimura*, Jacir L. Bordim†, Yasuaki Ito*, and Koji Nakano*
*Department of Information Engineering, Hiroshima University
1-4-1 Kagamiyama, Higashi-Hiroshima, 739-8527, Japan
†Department of Computer Science, University of Brasília
70910-900 Brasília - DF - Brazil

***Abstract*—The bulk execution of a sequential algorithm is to execute it for many different inputs in turn or at the same time. It is known that the bulk execution of an oblivious sequential algorithm can be implemented to run efficiently on a GPU. The bulk execution supports fine grained bitwise parallelism, allowing it to achieve high acceleration over a straightforward sequential computation. The main contribution of this work is to present a Bitwise Parallel Bulk Computation (BPBC) to accelerate the Smith-Waterman Algorithm (SWA). More precisely, the dynamic programming for the SWA repeatedly performs the same computation $O(mn)$ times. Thus, our idea is to convert this computation into a circuit simulation using the BPBC technique to compute multiple instances simultaneously. The proposed BPBC technique for the SWA has been implemented on the GPU and CPU. Experimental results show that the proposed BPBC for SWA accelerates the computation by over 447 times as compared to a single CPU implementation.**

***Index Terms*—Smith-Waterman, GPU, parallel algorithms, bulk computation, bitwise operations;**

I. Introduction

The GPU (Graphics Processing Unit) is a specialized circuit designed to accelerate computation for building and manipulating images [1], [2], [3]. Latest GPUs are designed for general purpose computing and can perform computation in applications traditionally handled by the CPU. Hence, GPUs have recently attracted the attention of many application developers [1], [4], [5], [6]. NVIDIA provides a parallel computing architecture called *CUDA* (Compute Unified Device Architecture) [7], the computing engine for NVIDIA GPUs. CUDA gives developers access to the virtual instruction set and memory of the parallel computational elements in NVIDIA GPUs. In many cases, GPUs are more efficient than multicore processors [8], since they have hundreds of processor cores and very high memory bandwidth.

CUDA uses two types of memories in the NVIDIA GPUs: *the shared memory* and *the global memory* [7]. The shared memory is an extremely fast on-chip memory with lower capacity, say, 16-48 Kbytes. The global memory is implemented as an off-chip DRAM, it has a large capacity (currently 1.5-12Gbytes) but its access latency is long. The efficient usage of the shared memory and the global memory is the key element for CUDA developers to accelerate applications using GPUs. In particular, we need to consider *bank conflict* of the shared memory access and *coalescing* of the global memory access [5], [8], [9]. The address space of the shared memory is mapped into several physical memory banks. If two or more threads access the same memory banks at the same time, the access requests are processed one by one. Hence, to maximize the memory access performance, threads of CUDA should access distinct memory banks to avoid the bank conflicts of the memory accesses. To maximize the bandwidth between the GPU and the DRAM chips, consecutive addresses of the global memory must be accessed at the same time. Thus, CUDA threads should perform coalesced access when they access the global memory.

A sequential algorithm is *oblivious* if an address accessed at each time unit is independent of the input [10]. For example, the prefix-sums of an array b of size n can be computed by executing $b[i] \leftarrow b[i] + b[i-1]$ for all i $(1 \leq i \leq n-1)$ in turn. This prefix-sum algorithm is oblivious because the address accessed at each time unit is independent of the values stored in b. *The bulk execution* of a sequential algorithm is to execute it for many different inputs in turn or at the same time. For example, suppose that we have p arrays $b_0, b_1, \ldots, b_{p-1}$ of size n each. We can compute the prefix-sums of each b_j $(0 \leq j \leq p-1)$ by executing the prefix-sum algorithm on a single CPU in turn or on a parallel computer in parallel. The bulk execution has many applications. For example, the conventional FFT algorithm [11] for n points running in $O(n \log n)$ time is oblivious. In practical signal processing, an input stream is equally partitioned into many blocks, and the FFT algorithm is executed for each block in turn or in parallel. This is exactly the bulk execution of the FFT algorithm. In our previous paper [10], we have introduced the bulk execution of sequential algorithms and show that they can be implemented in the GPU very efficiently if they are oblivious. In [12], we have also developed a conversion tool C2CU, which automatically generates an efficient CUDA program for the bulk computation from an oblivious sequential C language program. Quite surprisingly, the bulk computation can achieve speed-up factor of more than 100, even if it does not use the shared memory of the GPU.

In our previous papers [13], [14], we have introduced a novel technique, *the Bitwise Parallel Bulk Computation (BPBC) technique*. The bulk execution for oblivious algorithms presented in our previous papers [10], [12] is *wordwise* in the sense that each of data is stored in a word such as a 32-bit integer. On the other hand, the BPBC technique shown in [13], [14] supports ultimate fine grained bitwise parallelism and thus can achieve very high acceleration over the straightforward sequential computation. The BPBC technique simulates a combinational logic circuit for a lot of instances at the same time. More formally, let f be a function computed by a combinational logic circuit or a Boolean formula and $X_1, X_2, \ldots, X_M$ be the M inputs. By the BPBC technique $f(X_1)$, $f(X_2)$, ..., $f(X_M)$ can be computed very efficiently. The idea of the BPBC technique is

- to store data bits of each input instance in a particular bit of words of data, say 32-bit integers, and
- to simulate the combinational logic circuit for 32 input vectors at the same time by bitwise logic operations supported by computing devices such as CPUs and GPUs.

In [13], we showed an efficient simulation of the Conway's Game of Life [15], a well-known cellular automaton, using the BPBC technique. The next state of each cell in a cellular automaton is computed by simulating a combinational logic circuit. Thus, a state of each cell is stored in a bit of a 32-bit integer, and the combinational logic circuit to compute the next state is simulated by bitwise logic operations. We also showed that the CKY parsing of [14] can be done very efficiently by the BPBC technique. It is know that the CKY parsing can be done by repeatedly evaluating the same combinational circuit many times [16], [17]. The BPBC technique for the CKY parsing of [14] evaluate this circuit by bitwise logic operations.

Dynamic programming is a key technique for solving complex problems which takes exponential time by straightforward algorithms. The idea of the dynamic programing is to partition a problem into subproblems, solve them and store their solutions in appropriate data structures. Their solutions are combined to obtain the best solution of the problem. Since subproblems can be solved in parallel, several dynamic programming based parallel algorithms for the matrix chain product problem [18], the optimal polygon triangulation [5], the approximate string matching problem [19], [20] have been implemented in the GPU. Likewise, several GPU implementation for accelerating the Smith-Waterman algorithm have been proposed in the literature [21], [22], [23]. However, these implementations are wordwise. The main contribution of this paper is to present a GPU implementation for the Smith-Waterman using the BPBC technique.

The Smith-Waterman Algorithm (SWA) [24] employs dynamic to identify homologous regions between sequences by searching for optimal local alignments. Suppose that a sequence string X and a sequence string Y with length m and n ($m \leq n$), respectively, are given. The SWA computes the similarity between any pairs of sequences X and Y in $O(mn)$ time. The SWA is well-known for its accuracy and for being fairly demanding in time. Hence, a large number of development and optimizations to speed-up the SWA computation have been proposed in the literature [21], [22]. In [21], the authors propose a mechanism to accelerate the SWA for database search on a cluster of GPUs. Among their contributions, the authors proposed a memory allocation scheme and a data reuse scheme. The first scheme aims at reducing data transfers between the CPU and GPU as well as to reduce the amount of on-chip memory necessary for computing the SWA. The second scheme aims at improving data parallelism, which is achieved by "packing" the string elements. For further details, we refer the reader to [21]. In [22], the authors propose the incremental speculative traceback strategy to provide the alignment of very long DNA sequences in multi-GPU platforms using the exact SWA. This paper distinguishes from the aforementioned work as it explores bitwise arithmetic operations to speed-up the SWA computation. Also, the use of BPBC allows us to compute multiple input instances simultaneously. We expect that the proposed scheme can be used to enhance other Smith-Waterman strategies to further accelerate its computation.

The main contribution of this paper is to apply the BPBC technique to the Smith-Waterman for DNA strands each of which can be represented as a sequence of four bases A (adenine), G (guanine), C (cytosine), and T (thymine). In other words, we focus on the Smith-Waterman of strings of 2-bit characters and use the BPBC technique to accelerate this task Basically, the dynamic programming for the SWA repeatedly performs the same computation $O(mn)$ times [25]. Our idea is to convert this computation into circuit simulation. By the BPBC technique, circuit simulation for multiple instances can be done at the same time. The BPBC technique for the SWA has been implemented on the GPU and CPU. Experimental results show that the proposed BPBC for SWA accelerates the computation by over 293 times as compared to a single CPU implementation.

The remainder of this paper is organized as follows. Section II presents the BPBC technique applied to a straightforward string matching algorithm. Section III presents a brief overview of the Smith-Waterman Algorithm and Section IV shows the details of the proposed BPBC technique applied to the SW algorithm. Section V shows the details of the BPBC GPU implementation while Section VI presents the experimental results. Finally, Section VII concludes this work.

II. The BPBC technique

This section introduces the Bitwise Parallel Bulk Computation (BPBC) technique using a straightforward string matching as a simple example. Let $X = x_0x_1 \cdots x_{m-1}$ and $Y = y_0y_1 \cdots y_{n-1}$ be two strings of length m and n each such that $m \ll n$. The string matching is a task to find j such that $x_0x_1 \cdots x_{m-1} = y_jy_{j+1} \cdots y_{j+n-1}$. A straightforward algorithm shown below can find such j in $O(mn)$ time.

[**Straightforward string matching**]
for $j \leftarrow 0$ to $n - m$ do

$d[j] \leftarrow 0$;
for $i \leftarrow 0$ to $m-1$ do
  if($x_i \neq y_{i+j}$) $d[j] \leftarrow 1$

It should be clear that $d[j] = 0$ if $x_0x_1 \cdots x_{m-1} = y_jy_{j+1} \cdots y_{j+n-1}$. For example, for input strings $X = ATTCG$ and $Y = AAATTCGGGA$, the resulting values of d is 110111. Since there are a number of more efficient algorithms for computing the string matching [11], [26], the reader may think that it makes no sense to accelerate this inefficient, straightforward, algorithm. However, we use it as a simple example to understand how the BPBC technique can be applied. Furthermore, the approximate string matching that we will show later is an extension of the straightforward string matching.

Suppose that the input strings are DNA strands with four bases A (adenine), G (guanine), C (cytosine), and T (thymine). In other words, strings are sequences of characters in $\{A, G, C, T\}$, each of which can be encoded by 2 bits. For example, we can define a 2-bit encoding such that $A = 00$, $G = 10$, $C = 11$, and $T = 01$. Usually, string matching algorithms are implemented as *wordwise format* where each character is stored in an 8-bit or 32-bit word. If 8-bit words are used, only 2 bits of a word can be non-zero and the remaining 6 bits are always zero. Such wordwise format wastes memory space and memory bandwidth. We may use *packed format* in which four 2-bit characters are stored in an 8-bit word. However, reading and writing 2-bit characters needs messy bitwise operations for 8-bit words. Although the *packed format* can reduce the memory space, and it does not reduce memory bandwidth. The BPBC technique uses the *bit-transpose format* where a 32-bit word stores 32-bits of 32 instances [27]. Using the BPBC technique, the computation for 32 instances can be done in parallel.

Suppose that we have 32 pairs of DNA strands X_k and Y_k ($0 \leq k \leq 31$) with length m and n each such that $m \ll n$. The BPBC technique uses the bit-transpose format, arrays X^H, X^L, Y^H, and Y^L of 32-bit words of length m, m, n and n each defined as follows. Let $x_{k,i}$ ($0 \leq k \leq 31, 0 \leq i \leq m-1$) denote the i-th character of X_k, and let $x^H_{k,i}x^L_{k,i}$ be two bits of each $x_{k,i}$. Also, let $X^H_i = x^H_{31,i}x^H_{30,i} \cdots x^H_{0,i}$ and $X^L_i = x^L_{31,i}x^L_{30,i} \cdots x^L_{0,i}$ ($0 \leq i \leq m-1$) be 32-bit words. Similarly, let $y_{k,j} = y^H_{k,j}y^L_{k,j}$ ($0 \leq k \leq 31, 0 \leq j \leq n-1$) be the j-th character of Y_k and let $Y^H_j = y^H_{31,j}y^H_{30,j} \cdots y^H_{0,j}$ and $Y^L_i = y^L_{31,i}y^L_{30,i} \cdots y^L_{0,j}$ ($0 \leq i \leq n-1$) be 32-bit words. We have four input sequences $X^H = X^H_0X^H_1 \cdots X^H_{m-1}$, $X^L = X^L_0X^L_1 \cdots X^L_{m-1}$, $Y^H = Y^H_0Y^H_1 \cdots Y^H_{n-1}$, and $Y^L = Y^L_0Y^L_1 \cdots Y^L_{n-1}$ in bit-transpose format.

We can execute the straightforward string matching for 32 pairs of X_k and Y_k ($0 \leq i \leq 31$). Let $d[0], d[1], \ldots, d[m-n]$ be 32-bit words, if inputs are given as bit-transpose format. The straightforward string matching for 32 pairs can be performed by the BPBC technique as follows:

[**BPBC straightforward string matching**]
for $j \leftarrow 0$ to $n-m$ do
$d[j] \leftarrow 0$;
for $i \leftarrow 0$ to $m-1$ do
  $d[j] \leftarrow d[j] \vee (x^H_i \oplus y^H_{i+j}) \vee (x^L_i \oplus y^L_{i+j})$

The k-bit of the resulting 32-bit of $(x^H_i \oplus y^H_{i+j}) \vee (x^L_i \oplus y^L_{i+j})$ is 0 if $x_{k,i} = y_{k,i+j}$. Thus, when the BPBC straightforward string matching terminates, the k-th bit of $d[j] = 0$ is 0 if $x_{k,i}x_{k,i+1} \cdots x_{k,i+m-1} = y_jy_{j+1} \cdots y_{j+n-1}$. The running time of the BPBC straightforward string matching is also $O(mn)$. However, it can compute the string matching for 32 pairs of strings at the same time.

For the reader's benefits, we will show how the BPBC straight forward string matching works using 4-bit words. If this is the case, the string matching is executed for 4 inputs. Suppose that we have 4 pairs of strings as follows:

$X_0 = ATCGA$ $\quad Y_0 = AATCGACA$
$X_1 = TCGAC$ $\quad Y_1 = AATCGACA$
$X_2 = AAAAA$ $\quad Y_2 = AAAAAAAA$
$X_3 = TTTTT$ $\quad Y_3 = AATTTTTT$

These strings are converted to 4-bit words in the bit-transpose format as follows:

$X^H_0 = 0000$	$X^L_0 = 1010$	$Y^H_0 = 0000$	$Y^L_0 = 0000$
$X^H_1 = 0010$	$X^L_1 = 1011$	$Y^H_1 = 0000$	$Y^L_1 = 0000$
$X^H_2 = 0011$	$X^L_2 = 1001$	$Y^H_2 = 0000$	$Y^L_2 = 1011$
$X^H_3 = 0001$	$X^L_3 = 1000$	$Y^H_3 = 0011$	$Y^L_3 = 1011$
$X^H_4 = 0010$	$X^L_4 = 1010$	$Y^H_4 = 0011$	$Y^L_4 = 1000$
		$Y^H_5 = 0000$	$Y^L_5 = 1000$
		$Y^H_6 = 0011$	$Y^L_6 = 1011$
		$Y^H_7 = 0000$	$Y^L_7 = 1000$

By executing the BPBC straightforward string matching, we have the following result:

$d[0] = 0100$
$d[1] = 0101$
$d[2] = 1110$
$d[3] = 1100$

From the value in d's columns, one can observe the positions of matching. For instance, the resulting values of d for X_0 and Y_0 is 0100.

To use the BPBC technique, we need to convert inputs into the bit-transpose format. This can be done by transpose of a bit matrix. We show the first characters of 32 DNA strands of X, $x_{0,0}x_{1,0} \ldots x_{31,0}$ are converted into the bit-transpose format. The other characters can be converted in the same way. We assume that these characters are stored in 32 32-bit words $A[0], A[1], \ldots, A[31]$. Let $a_{i,j}$ ($0 \leq i, j \leq 31$) denote the j-th of $A[i]$. In other words, $A[i] = a_{i,31}a_{i,30} \cdots a_{i,0}$ for all i. After the bit transpose, each $A[i]$ stores $a_{31,i}a_{30,i} \cdots a_{0,i}$.

A straightforward bit-by-bit exchanging is very costly. However, we can perform transpose very efficiently using bit matrix transpose technique shown in Subsection 7.3 of [27]. For simplicity, we show how an 8×8 bit matrix stored in eight 8-bit words $A[0], A[1], \ldots, A[7]$. The idea is to repeat swap operations for square blocks as illustrated in Figure 1. It should be clear that bit transpose for an 8×8 matrix can be done in three steps.

Let us evaluate the total number operations necessary to perform bit transpose. We use the following function "swap"

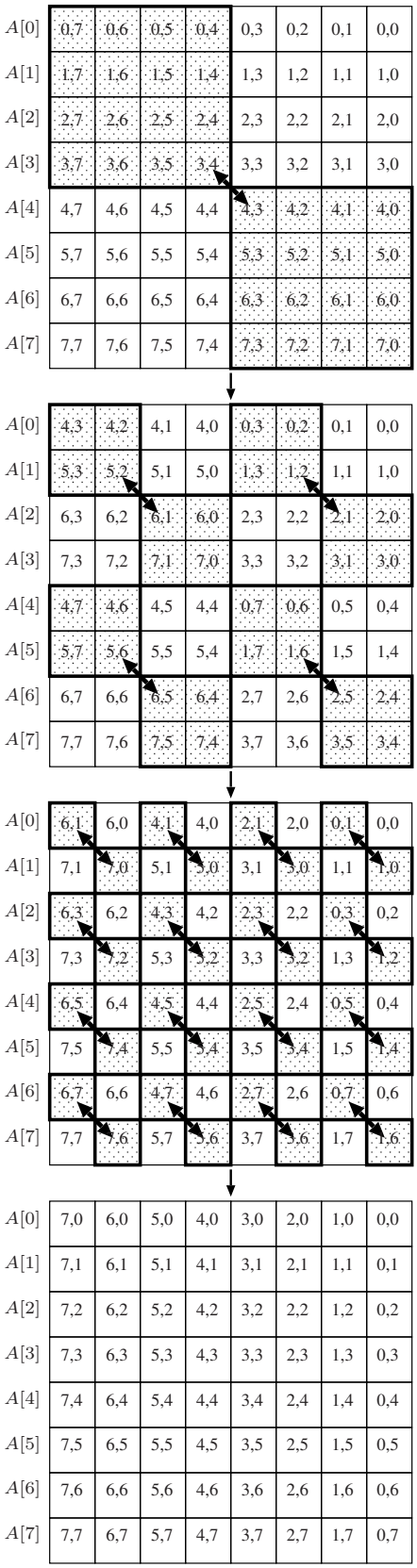

Fig. 1. Bit transpose for an 8×8 matrix

that exchanges bits of two words A and B such that bits in b specifies the bits to be exchanged for each of k bits.

swap(A, B, k, b)
$C \leftarrow ((A >> k) \wedge b) \oplus (B \wedge b)$;
$A \leftarrow A \oplus (C << k)$; $B \leftarrow B \oplus C$;

Using swap function, bit matrix transpose can be done by the following function:

[Bit transpose of 8×8 bit matrix]
bit-transpose(A)
swap$(A[0], A[4], 4, 00001111)$; swap$(A[1], A[5], 4, 00001111)$;
swap$(A[2], A[6], 4, 00001111)$; swap$(A[3], A[7], 4, 00001111)$;
swap$(A[0], A[2], 2, 00110011)$; swap$(A[1], A[3], 2, 00110011)$;
swap$(A[4], A[6], 2, 00110011)$; swap$(A[5], A[7], 2, 00110011)$;
swap$(A[0], A[1], 1, 01010101)$; swap$(A[2], A[3], 1, 01010101)$;
swap$(A[4], A[5], 1, 01010101)$; swap$(A[6], A[7], 1, 01010101)$;

Let us evaluate the total number of operations necessary to perform bit transpose. Each swap operation performs 7 operations including bit shift, bitwise AND, and bitwise XOR. Since swap operation is performed $4 \times 3 = 12$ times, bit transpose of an 8×8 bit matrix performs 84 operations. It should be clear that swap operation is performed $16 \times 5 = 80$ times for bit transpose of a 32×32 matrix. Thus, we have,

Lemma 1: A 32×32 bit matrix can be transposed by 560 operations.

If the value stored in every $A[i]$ is small, we can reduce the number of operations. For example, every $A[i]$ stores a 2-bit number if bit transpose is used for string matching of DNA strands. Suppose that eight 8-bit words $A[0], A[1], \ldots, A[7]$ store 2-bit numbers. In other words, $a_{i,j} = 0$ for all j ($2 \leq j \leq 7$). After bit transpose, 2-bit number are stored in $A[0]$ and $A[1]$. Thus, we can reduce the total number of operations. We use the following function "copy" to implement transpose for this case.

copy(A, B, k, b)
$A \leftarrow (A \wedge b) \vee ((B \wedge b) << k)$;

Clearly, this function performs 4 operations. Using copy function, bit transpose of eight 4-bit saturation numbers stored in eight 8-bit words can be done as follows:

bit-transpose(A)
copy$(A[0], A[4], 4, 00001111)$; copy$(A[1], A[5], 4, 00001111)$;
copy$(A[2], A[6], 4, 00001111)$; copy$(A[3], A[7], 4, 00001111)$;
copy$(A[0], A[2], 2, 00110011)$; copy$(A[1], A[3], 2, 00110011)$;
swap$(A[0], A[1], 1, 01010101)$;

Since copy function performed 6 times and swap function performed once, the total number of operations is $6 \times 4 + 1 \times 7 = 31$.

We can apply this technique to a 32×32 bit matrix stored in 32 32-bit words. Table I summarizes the number of swap and copy functions called in each of the five steps and the total number of operations for inputs with s-bit numbers. For preprocessing of the BPBC straightforward string matching,

TABLE I
THE NUMBER OF OPERATIONS PERFORMED FOR BIT TRANSPOSE OF A 32×32 BIT MATRIX

Step	1		2		3		4		5		total		total
swap/copy	swap	copy	swap	copy	swap	copy	swap	copy	swap	copy	swap	copy	operations
$s = 32$	16	-	16	-	16	-	16	-	16	-	80	-	560
$s = 16$	-	16	8	-	8	-	8	-	8	-	16	40	272
$s = 8$	-	16	-	8	4	-	4	-	4	-	12	24	180
$s = 7$	-	16	-	8	4	-	4	-	3	1	11	25	177
$s = 6$	-	16	-	8	4	-	2	2	2	2	8	28	168
$s = 5$	-	16	-	8	4	-	2	2	2	1	8	27	164
$s = 4$	-	16	-	8	-	4	2	-	2	-	4	28	140
$s = 3$	-	16	-	8	-	4	-	2	1	1	1	31	131
$s = 2$	-	16	-	8	-	4	-	2	1	-	1	30	127

we use bit transpose with 2-bit numbers, which performs only 127 operations.

It should be clear that "bit-untranspose" can be done by executing operations performed by bit transpose backwards.

III. SMITH-WATERMAN ALGORITHM

The main purpose of this section is to present a brief overview of the Smith-Waterman Algorithm (SWA). Please see [24] for further details.

Sequence alignment is an important problem in bioinformatics as it allows the analyses of evolutionary relationships and the extraction of structural information from sequences of DNA, RNA, or proteins. The Smith-Waterman algorithm is a "local-alignment" method based on dynamic programming. Unlike the global alignment problem, where entire strings are to be matched, local alignment problem identifies highly similar substrings. For this reason, local alignment is the preferred choice for biological applications.

Given a sequence $X = x_1x_2 \cdots x_m$ of length m and a sequence $Y = y_1y_2 \cdots y_n$ of length n, the SWA computes the similarity between any pairs of elements in these sequences. For this purpose, the SWA uses a *scoring matrix* d of size $m \times n$. For each element $d[i][j]$, $(0 \leq i < m, 0 \leq j < n)$, the scoring matrix is computed based on the following recurrence:

$$d[i][j] = \max \begin{cases} 0, \\ d[i-1][j] - gap, \\ d[i][j-1] - gap, \\ d[i-1][j-1] + w(x_i, y_j), \end{cases}$$

where $d[i][j]$ represents the optimal cost for substring X and Y, the gap is the cost for creating a gap between the sequences being evaluated and $w(x_i, y_j)$ represent the mismatching/matching cost, defined as:

$$w(x_i, y_j) = \begin{cases} c_1 & \text{if } x_i = y_j, \\ -c_2 & \text{if } x_i \neq y_j. \end{cases}$$

The scoring matrix is composed only of non-negative values and has the effect of stopping considering regions of high dissimilarity between the sequences. Using the above recurrence, all values of matrix d can be computed by a sequential algorithm as follows.

[Sequential algorithm for the SWA]
for $j \leftarrow -1$ to $n-1$ do $d[-1][j] \leftarrow 0$
for $i \leftarrow 0$ to $m-1$ do $d[i][-1] \leftarrow 0$
for $j \leftarrow 0$ to $n-1$ do
  for $i \leftarrow 0$ to $m-1$ do
    $d[i][j] \leftarrow \max(0, d[i][j-1] - gap,$
      $d[i-1][j] - gap,$
      $d[i-1][j-1] + w(x_i, y_j))$
Function $w(x_i, y_j)$
  if $(x_i \neq y_i)$ return c_2 else return c_1

Table II shows the values of d for strings $X = TACTG$ and $Y = GAACTGA$. In this example, $c_1 = 2$ and $c_1 = -1$ and $gap = -1$. The boldfaced values show the local alignment with highest score, which can be obtained by finding the highest score in matrix d and traceback along its diagonal. From the table, we can see that the SWA of X and Y finds the highest values of d for subsequences $x_1x_2x_3x_4$ and $y_2y_3y_4y_5$. When there is a gap in the sequences, the values in d line up vertically or horizontally. The SWA often uses a *traceback* matrix to record the direction of the alignment from one cell to another along the path. In what follows, we restrict our attention to the scoring matrix, as the traceback matrix can computed along with the scoring matrix. Alternatively, when multiple instances are being evaluated, those with higher score could be selected for computing traceback matrices.

TABLE II
THE VALUES OF MATRIX d FOR THE SWA

		G	A	A	C	T	G	A
	0	0	0	0	0	0	0	0
T	0	0	0	0	0	2	1	0
A	0	0	2	**2**	1	1	1	3
C	0	0	1	1	**4**	3	2	2
T	0	0	0	0	3	**6**	5	4
G	0	2	1	0	2	5	**8**	7

In this work, the proposed BPBC technique is used identify the input strings in which the maximum value of the scoring matrix is larger than a given threshold τ. In other words, BPBC technique is used to find the longest match between the two input sequences X and Y. Once such strings are identified, a detailed matching can be computed by a conventional SWA on the CPU, where the score and traceback matrices can be used to identify similar regions between them. For instance, in the example of Table II, the maximum value of the scoring matrix is 8, which is shown in the bottom-most line of the matrix.

Clearly, if $\tau < 8$, the input strings in the above example would be selected for further evaluation.

Next, we will parallelize the sequential algorithm. The idea is to compute every row from left to right in parallel. The value of $d[i][j]$ is computed only after $d[i-1][j]$ is obtained. Thus, we compute values of $d[0][t], d[1][t-1], \ldots, d[m-1][t-m+1]$ at the same time. This parallel computation is performed for $t = 0$ to $n+m-2$. The details of the parallel SWA are spelled out as follows:

[Parallel algorithm for the SWA]
for $j \leftarrow -1$ to $n-1$ do in parallel $d[-1][j] \leftarrow 0$
for $i \leftarrow 0$ to $m-1$ do in parallel $d[i][-1] \leftarrow 0$
for $t \leftarrow 0$ to $n+m-2$ do
 for $i \leftarrow 0$ to $m-1$ do in parallel
  $j \leftarrow t-i+1$;
  if$(1 \leq j \leq n)$
  $d[i][j] \leftarrow \max(0, d[i][j-1] - gap,$
  $d[i-1][j] - gap,$
  $d[i-1][j-1] + w(x_i, y_j))$
Function $w(x_i, y_j)$
 if $(x_i \neq y_i)$ return c_2 else return c_1

The reader should refer to Table III showing the values of t when each element of d is computed. The values are computed from top-left to right-bottom.

TABLE III
THE VALUES OF t WHEN EACH ELEMENT OF d IS COMPUTED

		G	A	A	C	T	G	A
	-	-	-	-	-	-	-	-
A	-	1	2	3	4	5	6	7
C	-	2	3	4	5	6	7	8
C	-	3	4	5	6	7	8	9
T	-	4	5	6	7	8	9	10
G	-	5	6	7	8	9	10	11

IV. BPBC TECHNIQUE APPLIED FOR SWA

The main purpose of this section is to use the BPBC technique to accelerate the Smith-Waterman algorithm. In what follows, let l be the number of bits necessary to encode the characters of the input strings. For example, a DNA strand with four basis A (adenine), G (guanine), C (cytosine), and T (thymine) can be encoded with $\epsilon = 2$ bits. Let s denote the number of bits necessary to hold the maximum value of the scoring matrix. Clearly, for $m \ll n$, the maximum value in matrix d occurs when there is a sequence matching. In this case, s needs at most $\lceil \log_2(c_1 \cdot m) \rceil$ bits.

A. Bitwise arithmetic for SWA

We begin by presenting the comparison algorithm, which computes p such that $p = 0$ if $A < B$ and $p = 1$ if $A > B$. Note that p can take any value if neither $A < B$ nor $A > B$. The following function greaterthan(A, B) returns such p.

[Compare Function]
greaterthan(A, B)
 $p \leftarrow \overline{a_0} \wedge b_0$;
 for $i \leftarrow 1$ to $s-1$ do
  $p \leftarrow (b_i \wedge p) \vee (\overline{a_i} \wedge (b_i \oplus p))$;
 return $\overline{p}$;

Basically, p stores the borrow of each bit when subtraction $A - B$ is computed from the least significant bit. In other words, $p = 1$ if $b_i + p > a_i$, that is,

- $b_i = p = 1$ or
- $a_i = 0$ and either $b_i = 1$ or $p = 1$

Thus, the borrow from the next bit is computed by evaluating $(b_i \wedge p) \vee (\overline{a_i} \wedge (b_i \oplus p))$.

Using greaterthan(A, B), we can compute $Q = q_{s-1}q_{s-2}\cdots q_0$ such that Q is the maximum of A and B. The following function $\max_B(A, B)$ computes and returns such Q.

[Maximum Function]
$\max_B(A, B)$
$p \leftarrow$ greaterthan(A, B);
for $i \leftarrow 0$ to $s-1$ do
 $q_i \leftarrow (a_i \wedge p) \vee (b_i \wedge \overline{p})$;
return Q;

Since greaterthan(A, B) performs $3 + 5(s-1) = 5s-2$ operations and $\max_B(A, B)$ performs $4s$ operations, we have,

Lemma 2: $\max_B(A, B)$ performs $9s-2$ operations.

Next, we present the addition algorithm $\mathrm{add_B}(A, B)$, which computes the sum of two s-bit binary values A and B and stores its value on the return variable Q.

[Addition Function]
$\mathrm{add_B}(A, B)$
 $p \leftarrow q_0 \leftarrow a_0 \oplus b_0$;
 for $i \leftarrow 1$ to $s-1$ do
  $q_i \leftarrow (a_i \oplus b_i \oplus p)$;
  $p \leftarrow (a_i \wedge (b_i \oplus p)) \vee (b_i \wedge p)$;
 return Q;

Since $\mathrm{add_B}(A, B)$ algorithm performs $1+6(s-1) = 6s-5$ operations, we have,

Lemma 3: $\mathrm{add_B}(A, B)$ performs $6s-5$ operations.

Next, we present the saturation subtraction algorithm $\mathrm{SSub_B}(A, B)$, which computes the return value $Q = \max(A-B, 0)$. The $\mathrm{SSub_B}$ is defined as follows:

[Saturation Subtraction Fuction]
$\mathrm{SSub_B}(A, B)$
 $q_0 \leftarrow a_0 \oplus b_0$;
 $p \leftarrow \overline{a_0} \wedge b_0$;
 for $i \leftarrow 1$ to $s-1$ do
  $q_i \leftarrow (a_i \oplus b_i \oplus p)$;
  $p \leftarrow (\overline{a_i} \wedge (b_i \oplus p)) \vee (b_i \wedge p)$;
 for $i \leftarrow 0$ to $s-1$ do
  $q_i \leftarrow q_i \wedge \overline{p}$;
 return Q;

Since $\mathrm{SSub_B}(A, B)$ algorithm performs $3+7(s-1)+2s = 9s-4$ operations, we have:

Lemma 4: $\mathrm{SSub_B}(A, B)$ performs $9s-4$ operations.

Finally, we present the $\mathrm{matching_B}(C, x, y)$ algorithm, which computes $Q = C + w(x, y)$. Recall form the parallel SWA that the evaluation of function w checks whether x equals to y or not. The $\mathrm{matching_B}(C, x, y)$ algorithm computes both $C + c_1$ and $C + c_2$. The algorithm uses a flag e to check if $x \neq y$. If so, the algorithm returns $Q = C + c_2$, otherwise it returns $Q = C + c_1$. The details of the matching algorithm are presented below:

[Matching Function]

$\mathrm{matching_B}(C, x, y)$
  $R \leftarrow \mathrm{add(C, c_1)}$;
  $T \leftarrow \mathrm{SSub(C, c_2)}$;
  $e \leftarrow 0$;
  for $i \leftarrow 0$ to $\epsilon - 1$ do
    $e \leftarrow e \vee (x_i \oplus y_i)$;
  for $i \leftarrow 0$ to $s - 1$ do
    $q_i \leftarrow (r_i \wedge \overline{e}) \vee (t_i \wedge e)$;
  return Q;

The $\mathrm{matching_B}(C, x, y)$ algorithm calls $\mathrm{add_B}(C, c_1)$ and $\mathrm{SSub_B(C, c_2)}$ algorithms, which requires $6s - 5 + 9s - 4 = 15s - 9$ operations. In addition, it uses $4s + 2\epsilon < 6s$ operations to execute the for-loops. Thus, we have:

Lemma 5: $\mathrm{matching_B}(C, x, y)$ can be computed using at most $21s - 9$ operations.

For the computation of the Smith-Waterman algorithm, we will evaluate the function $\mathrm{SW}(A, B, C, x, y) = \max(0, A - gap, B - gap, C + w(x, y)$ as follows:

[SW Function]

$\mathrm{SW}(A, B, C, x, y)$
  $T \leftarrow \mathrm{max}_B(A, B)$;
  $U \leftarrow \mathrm{SSub_B(T, gap)}$;
  $T \leftarrow \mathrm{matching_B}(C, x, y)$;
  $T \leftarrow \mathrm{max}_B(T, U)$;
  return T;

Note that $\mathrm{max_B}, \mathrm{SSub_B}$ and $\mathrm{matching_B}$ return non-negative values. Hence, it suffices to compute the maximum between temporary values of T and U to obtain the final value, which is stored in variable T. The computation of $SW(A, B, C, x, y)$ calls the max_B function twice, $\mathrm{SSub_B}$ and $\mathrm{matching_B}$ functions once. Thus, we have:

Theorem 6: $\mathrm{SW}(A, B, C, x, y)$ performs $48s - 18$ operations.

B. The BPBC technique for the SWA

We use the BPBC technique to perform the Smith-Waterman on 32 input sequences in parallel. As before, we consider the input to be 32 pairs of DNA strands X_k and Y_k $(0 \leq k \leq 31)$ with length m and n each such that $m \ll n$. Let $x_{k,i}$ $(0 \leq k \leq 31, 0 \leq i \leq m - 1)$ denote the i-th character of X_k and $x^H_{k,i}x^L_{k,i}$ be two bits of $x_{k,i}$. Let $X^H_i = x^H_{31,i}x^H_{30,i}\cdots x^H_{0,i}$ and $X^L_i = x^L_{31,i}x^L_{30,i}\cdots x^L_{0,i}$ $(0 \leq i \leq m - 1)$ be 32-bit words. Similarly, let $y_{k,j} = y^H_{k,j}y^L_{k,j}$ $(0 \leq k \leq 31, 0 \leq j \leq n - 1)$ be the j-th character of Y_k and let $Y^H_j = y^H_{31,j}y^H_{30,j}\cdots y^H_{0,j}$ and $Y^L_i = y^L_{31,i}y^L_{30,i}\cdots y^L_{0,j}$ $(0 \leq i \leq n - 1)$ be 32-bit words. Let D_k $(0 \leq k \leq 31)$ denote the table d of size $(m + 1) \times (n + 1)$ for the computation of $\mathrm{SWA}(X_k, Y_k)$ and $D_k[i][j]$ be the (i, j) element of D_k. To apply the BPBC technique, let $D^l[i][j] = D^l_{31}[i][j]D^l_{30}[i][j]\cdots D^l_0[i][j]$ $(0 \leq i \leq m - 1, 0 \leq j \leq n - 1, 0 \leq l \leq s - 1)$ be a 32-bit word corresponding to the l-th bits of (i, j) element of 32 tables.

Let 1^{32} denote consecutive 32 1's in binary, that is, $2^{32} - 1$. Also, let gap, denote the gap cost in the input sequences, c_1 be the matching cost and c_2 be the mismatching cost. The sequential BPBC technique for the Smith-Waterman algorithm uses the aforementioned SW function. The details of the sequential SWA are spelled out as follows:

[BPBC sequential for SWA]

for $j \leftarrow 0$ to $n - 1$ do
  for $h \leftarrow 0$ to $s - 1$ do
    $D^h[-1][j] \leftarrow 0$;
for $i \leftarrow 0$ to $m - 1$ do
  for $h \leftarrow 0$ to $s - 1$ do
    $D^h[i][-1] \leftarrow 0$;
for $i \leftarrow 0$ to m do
  for $j \leftarrow 0$ to n do
    $D[i][j] \leftarrow \mathrm{SW}(D[i-1][j],$
      $D[i][j-1], D[i-1][j-1], x, y)$;

The parallel BPBC technique for the Smith-Waterman algorithm can be defined similarly as follows:

[BPBC parallel for SWA]

for $j \leftarrow 0$ to $n - 1$ do in parallel
  for $h \leftarrow 0$ to $s - 1$ do
    $D^h[-1][j] \leftarrow 0$;
for $i \leftarrow 0$ to $m - 1$ do in parallel
  for $h \leftarrow 0$ to $s - 1$ do
    $D^h[i][-1] \leftarrow 0$;
for $t \leftarrow 0$ to $n + m - 2$ do
  for $i \leftarrow 0$ to $m - 1$ do in parallel
    $j \leftarrow t - i + 1$;
    if$(0 \leq j \leq n - 1)$
      $D[i][j] \leftarrow \mathrm{SW}(D[i-1][j],$
        $D[i][j-1], D[i-1][j-1], x, y)$;

Note that the sequential and parallel SWAs using the BPBC technique require the input arrays to be bit-transpose. Clearly, we can use bit transpose for 2-bit numbers shown in Section II to convert input strings into X_H, X_L, Y_H and Y_L.

V. GPU IMPLEMENTATION OF THE BPBC PARALLEL FOR SWA

This section presents a GPU implementation of the BPBC for the Smith-Waterman algorithm. We assume that the pairs of input strings X_k and Y_k, $(0 \leq k \leq 31)$ with length m and n each, are stored in the main memory of the host PC. The resulting values of the $\mathrm{SWA}(X_k, Y_k)$, that is, maximum score is computed and then transferred back to the host PC. We require that the input strings and the resulting values to be wordwise format, that is, the value of each element is stored

in a word of memory. Note that most applications consider the input strings to be in wordwise format.

Our GPU implementation using the BPBC technique consists of the following steps:

Step 1: All pairs of input strings X_k and Y_k in wordwise format are copied to the global memory of the GPU.

Step 2: All input strings are bit transposed to convert them in bit transposed format. The resulting input strings in bit transposed format are stored in the global memory.

Step 3: SWA(X_k, Y_k) are computed by the BPBC parallel SWA, and the resulting values are written in the global memory in bit transpose format.

Step 4: The resulting values of SWA(X_k, Y_k) are bit-untranspose to convert them in wordwise format.

Step 5: The maximum score of the SWA(X_k, Y_k), in wordwise format, are transferred to the host PC.

Steps 1 and 5 can be realized using the `cudaMemcpy` function. One CUDA kernel is invoked for each Step 2, 3, and 4. In Step 2, each thread performs bit transpose for 32 characters. We use CUDA blocks of 1024 threads each to maximize occupancy. Similarly, Step 4 performs bit-untranspose by a CUDA kernel call with CUDA blocks of 1024 threads each. Step 3 calls a CUDA kernel of CUDA blocks with m threads each to compute SWA(X_k, Y_k) for 32 pairs of X_k and Y_k, where m is the length of X_k. Each CUDA kernel executes the BPBC parallel for SWA. Each thread i ($0 \leq i \leq m-1$) computes all elements in the i-th row of 32 tables from left to right. More specifically, each thread i performs the following operations to compute $d[i][t-i+1]$ using three values $d[i-1][t-i], d[i-1][t-i+1]$, and $d[i][t-i]$ in each t-th iteration:

1) If $t = 0$ then read $x_{k,j}$ from the global memory.
2) Read $y_{k,t-i+1}$ from the global memory.
3) Compute $d[i][t-i+1]$ from $d[i-1][t-i], d[i-1][t-i+1]$, and $d[i][t-i]$ and store it in a local register $R_{k,i} = \max_B(d[i][t-i+1], R_{k,i})$.
4) Send $d[i][t-i+1]$ to thread $i+1$ and receive $d[i-1][t-i+2]$ from thread $i-1$.
5) If $j = n-1$, then send the value $R_{k,i}$ to thread $i+1$. If $n-2$, receive $R_{k,i-1}$ from thread $i-1$ and computes $R_{k,i} = \max_B(R_{k,i}, R_{k,i-1})$. When $t = n+m-2$, the bottom-most thread writes the value $R_{k,m-1}$ to the global memory.

Figure 2 illustrates the computation of value $d[i][t-i+1]$ by thread i. When the bottom-most thread reaches the bottom-right conner, it holds the $\max_B\{R_{k,0}, \ldots, R_{k,m-1}\}$ and writes it to the global memory. The communication among the active threads can be performed in a way similar to that proposed in [20]. More precisely, shuffle operations can be employed to transfers values among threads in the same warp, thus reducing the number of read and write operations to the shared memory. We use two registers to store each $x_{k,i}$ and $y_{k,t-i+1}$ of the 32 pairs of inputs. Since $x_{k,i}$ is fixed for each thread i, it is sufficient to read it once. Also, we use s 32-bit registers each to store $d[i][t-i+1], d[i-1][t-i], d[i-1][t-i+1]$, and $d[i][t-i]$ of 32 tables. Thus, each thread uses $4s+4$ 32-bit registers to store these values. For sending $d[i][t-i+1]$ and receiving $d[i-1][t-i+2]$, data transfer is necessary. We use the shared memory of a CUDA block (or streaming multiprocessor) for this purpose. For example, thread i writes $d[i][t-i+1]$ in the shared memory and thread $i+1$ read it for sending $d[i][t-i+1]$ to thread $i+1$.

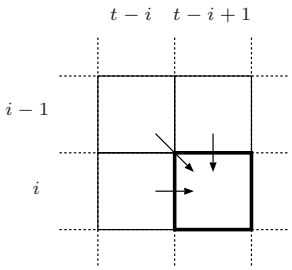

Fig. 2. The computation of $d[i][t-i+1]$ by thread i

VI. Experimental results

The main purpose of this section is to show the performance of algorithms for SWA.

We have evaluated the running time for the SWA for 32K ($= 32768$) pairs with pattern strings of a fixed length of $m = 128$ and data strings of length $n = 1024, 2048, \ldots, 65536$. To see the potentiality of the BPBC technique, we have implemented both the conventional wordwise implementation and the proposed bitwise implementation. We have used an Intel Core i7-6700 (3.6GHz) and NVIDIA GeForce GTX TITAN X. GeForce GTX TITAN X has 28 streaming multiprocessors with 128 cores each. Sequential algorithms are executed on a single thread running on the Intel Core i7-6700. We may accelerate these sequential algorithms using multiple threads and/or SIMD instructions. However, these acceleration techniques for Core i7 CPU are out of scope of this work as our goal is not to compare the capability of NVIDIA GPU and Intel Core i7 CPU. The speedup factors of GPU implementations over CPU implementations shown in our experiments are merely for reference purpose.

Table IV summarizes the running time for the Smith-Waterman Algorithm. In the table, the columns "W2B" and "B2W" represent the time taken to perform the bit-transpose and bit-untranspose of the input and resulting values, respectively. Recall that in this work we are interested in computing the maximum score of the input strings. The "SWA" column refers to the time to compute the Smith-Waterman Algorithm. In GPU implementations, "H2G" (Host PC to GPU transfer time for input strings) and "G2H" (GPU to Host PC transfer time for resulting values) are evaluated. Recall that the size of data to be bit-transposed and bit-untransposed is $2(n+m)$ bits and $\lceil log_2(c_1 \cdot m) \rceil$ bits, respectively. Therefore, when n is larger, the time of W2B is longer. On the other hand, the time of B2W is slightly larger or almost the same in CPU

TABLE IV
THE RUNNING TIME IN MILLISECONDS FOR THE SWA FOR 32K PAIRS

		CPU				GPU					
	n	W2B	SWA	B2W	Total	H2G	W2B	SWA	B2W	G2H	Total
Bitwise 32-bits	1024	153.89	10990.03	0.15	11144.07	5.51	0.14	6.91	0.01	0.08	12.66
	2048	306.70	21918.45	0.16	22225.32	10.60	0.22	12.61	0.01	0.08	23.52
	4096	715.70	45065.72	0.15	45781.57	19.01	0.32	24.17	0.01	0.07	43.59
	8192	1451.89	90114.62	0.21	91566.72	38.00	0.56	48.29	0.01	0.07	86.94
	16384	3063.70	180065.17	0.18	183129.05	79.54	1.02	96.56	0.01	0.08	177.21
	32768	5907.22	357122.10	0.26	363030.58	153.31	1.85	196.03	0.01	0.08	351.27
	65536	8924.32	720876.85	0.27	729800.04	299.47	3.35	392.52	0.01	0.08	695.42
Bitwise 64-bits	1024	232.54	5434.08	0.09	5666.71	5.71	2.76	10.72	0.01	0.08	19.28
	2048	471.38	10871.87	0.11	11343.36	10.81	5.13	20.47	0.01	0.08	36.51
	4096	944.04	21894.50	0.13	22838.67	19.61	9.84	38.43	0.01	0.08	67.97
	8192	2051.98	43544.63	0.14	45596.74	37.89	19.22	75.44	0.01	0.07	132.64
	16384	3890.75	86937.86	0.17	90828.78	76.21	37.76	150.08	0.01	0.08	264.14
	32768	6593.45	174271.58	0.23	180865.26	151.97	75.33	301.07	0.01	0.08	528.46
	65536	8973.66	348896.24	0.24	357870.14	297.54	150.59	605.80	0.01	0.09	1054.04
Wordwise 32-bits	1024	-	6803.99	-	6803.99	5.78	-	30.66	-	0.08	36.51
	2048	-	13590.92	-	13590.92	10.46	-	52.66	-	0.07	63.20
	4096	-	27169.32	-	27169.32	20.22	-	111.62	-	0.07	131.91
	8192	-	54358.14	-	54358.14	39.83	-	203.41	-	0.08	243.32
	16384	-	108680.38	-	108680.38	78.52	-	446.47	-	0.08	525.07
	32768	-	217621.17	-	217621.17	156.89	-	835.81	-	0.08	992.78
	65536	-	435637.82	-	435637.82	315.53	-	1861.36	-	0.07	2176.96

TABLE V
THE THROUGHPUT IN GCUPS AND SPEED-UP FACTORS FOR THE SWA USING BPBC FOR 32K PAIRS

n	CPU	GPU	Speed-up
1024	0.76	1877.40	447.6
2048	0.76	2022.85	482.3
4096	0.75	2197.58	523.9
8192	0.75	2199.75	524.5
16384	0.76	2149.79	512.5
32768	0.76	2159.60	514.9
65536	0.77	2158.43	514.6

and GPU implementations. Also, we can see that, even if data transfer time between host PC and GPU is included, the parallel computation using the GPU is faster than the sequential computation using the CPU. In fact, with $n = 1024$ the GPU bitwise implementation accelerates the SWA by 293 and 880 times using, respectively, 64-bits and 32-bits. As the size of n increases, the bitwise GPU implementation provides further acceleration of the SWA. With $n = 65526$, the GPU bitwise implementation accelerates the SWA by 339 and 1049 times using, respectively, 64-bits and 32-bits. This represents an improvement of nearly 15% and 19%, respectively for the bitwise implementation using 64-bits and 32 bits. As for the the conventional CPU implementation of the SWA using wordsize of 32-bits, the GPU implementation provides acceleration surpassing 186 times. Compared to the wordwise CPU implementation, the bitwise implementation with 64-bits reduces the running time by $\approx 20\%$ on average.

Table V shows the throughput in GCUPS (billion Cell Updates Per Second), which is one of the performance metrics often used in bioinformatics. The table shows the speed-up factors for the SWA using BPBC for 32K pairs when 64-bits and 32-bits are used in the CPU implementation and the GPU implementation, respectively, which is the best wordsize selected in each implementation. According to the table, the performance of our GPU implementation reaches at most 2022 GCUPS. Munekawa et al. [21] shows that a peak performance of their GPU implementation of SWA using GeForce GTX 280 is 8.32 GCUPS. Although the utilized GPU and scoring computation are different, the performance of our GPU implementation is incomparably higher than that of the existing GPU implementation. Furthermore, the GPU implementation runs 447 to 524 times faster than the sequential CPU implementation. Thus, the proposed BPBC technique of for the SWA is suitable to the GPU acceleration.

VII. CONCLUSION

This work presented a Bitwise Parallel Bulk Computation (BPBC) to accelerate the Smith-Waterman Algorithm (SWA). More precisely, this work explores bitwise arithmetic operations to speed-up the SWA computation. Our idea is to convert the dynamic programming computation for the SWA into circuit simulation and use the BPBC technique to compute multiple instances simultaneously. The proposed BPBC technique for the SWA has been implemented on both GPU and CPU. Experimental results show that the proposed BPBC for SWA accelerates the computation by over 293 times as compared to a single CPU implementation. We believe that the proposed BPBC can be coupled with other SWA strategies

to further accelerate its computation. As a future work, we plan to explore such alternatives.

REFERENCES

[1] W. W. Hwu, *GPU Computing Gems Emerald Edition*. Morgan Kaufmann, 2011.

[2] D. Man, K. Uda, Y. Ito, and K. Nakano, "A GPU implementation of computing Euclidean distance map with efficient memory access," in *Proc. of International Conference on Networking and Computing*, Dec. 2011, pp. 68–76.

[3] A. Uchida, Y. Ito, and K. Nakano, "Fast and accurate template matching using pixel rearrangement on the GPU," in *Proc. of International Conference on Networking and Computing*. IEEE CS Press, Dec. 2011, pp. 153–159.

[4] K. Ogawa, Y. Ito, and K. Nakano, "Efficient Canny edge detection using a GPU," in *Proc. of International Conference on Networking and Computing*. IEEE CS Press, Nov. 2010, pp. 279–280.

[5] K. Nishida, Y. Ito, and K. Nakano, "Accelerating the dynamic programming for the optimal polygon triangulation on the GPU," in *Proc. of International Conference on Algorithms and Architectures for Parallel Processing (ICA3PP, LNCS 7439)*, Sept. 2012, pp. 1–15.

[6] A. Uchida, Y. Ito, and K. Nakano, "An efficient GPU implementation of ant colony optimization for the traveling salesman problem," in *Proc. of International Conference on Networking and Computing*. IEEE CS Press, Dec. 2012, pp. 94–102.

[7] NVIDIA Corporation, "NVIDIA CUDA C programming guide version 7.0," Mar 2015.

[8] D. Man, K. Uda, H. Ueyama, Y. Ito, and K. Nakano, "Implementations of a parallel algorithm for computing Euclidean distance map in multicore processors and GPUs," *International Journal of Networking and Computing*, vol. 1, no. 2, pp. 260–276, July 2011.

[9] NVIDIA Corporation, "NVIDIA CUDA C best practice guide version 3.1," 2010.

[10] K. Tani, D. Takafuji, K. Nakano, and Y. Ito, "Bulk execution of oblivious algorithms on the unified memory machine, with GPU implementation," in *Proc. of International Parallel and Distributed Processing Symposium Workshops*, May 2014, pp. 586–595.

[11] T. H. Cormen, C. E. Leiserson, and R. L. Rivest, *Introduction to Algorithms*. MIT Press, 1990.

[12] D. Takafuji, K. Nakano, and Y. Ito, "A CUDA C program generator for bulk execution of a sequential algorithm," in *Proc. of International Conference on Algorithms and Architectures for Parallel Processing*, Aug. 2014, pp. 178–191.

[13] T. Fujita, K. Nakano, and Y. Ito, "Fast simulation of Conway's Game of Life using bitwise parallel bulk computation on a GPU," *International Journal of Foundations of Computer Science*, to appear.

[14] ——, "Bitwise parallel bulk computation on the GPU, with application to the CKY parsing for context-free grammars," in *Proc. of International Parallel and Distributed Processing Symposium Workshops*, May 2016, pp. 589–598.

[15] M. Gardner, "Mathematical games: The fantastic combinations of John Conway's new solitaire game "life"," *Scientific American*, vol. 223, pp. 120–123, Oct. 1970.

[16] J. L. Bordim, Y. Ito, and K. Nakano, "Accelarating the CKY parsing using FPGAs," *IEICE Transactions on Information and Systems*, vol. E86-D, no. 5, pp. 811–818, 2003.

[17] J. L. Bordim, O. H. Ibarra, Y. Ito, and K. Nakano, "Instance-specific solutions to accelerate the CKY parsing for large contex-free grammars," *International Journal on Foundations of Computer Science*, vol. 15, no. 2, pp. 403–416, Apr. 2014.

[18] K. Nishida, Y. Ito, and K. Nakano, "Accelerating the dynamic programming for the matrix chain product on the GPU," in *Proc. of International Conference on Networking and Computing*, Dec. 2011, pp. 320–326.

[19] D. Man, K. Nakano, and Y. Ito, "An optimal implementation of the approximate string matching on the hierarchical memory machine, with performance evaluation on the GPU," *IEICE TRANSACTIONS on Information and Systems*, vol. E97-D, no. 12, pp. 3063–3071, Dec. 2014.

[20] L. Saad, J. Bordim, K. Nakano, and Y. Ito, "A fast approximate string matching algorithm on gpu," in *Proc. International Symposium on Computing and Networking*, Dec. 2015, pp. 188—192.

[21] Y. Munekawa, F. Ino, and K. Hagihara, "Accelerating Smith-Waterman algorithm for biological database search on CUDA-compatible GPUs," *IEICE Transactions on Information and Systems*, vol. E93.D, no. 6, pp. 1479–1488, 2010.

[22] E. F. d. O. Sandes, G. Miranda, X. Martorell, E. Ayguade, G. Teodoro, and A. C. M. Melo, "Cudalign 4.0: Incremental speculative traceback for exact chromosome-wide alignment in gpu clusters," *IEEE Transactions on Parallel and Distributed Systems*, vol. 27, no. 10, pp. 2838–2850, Oct 2016.

[23] D. Lipman and W. Pearson, "Rapid and sensitive protein similarity searches," *Science*, vol. 227, no. 4693, pp. 1435–1441, 1985. [Online]. Available: http://science.sciencemag.org/content/227/4693/1435

[24] T. F. Smith and M. S. Waterman, "Identification of common molecular subsequences." *Journal of molecular biology*, vol. 147, no. 1, pp. 195–197, Mar. 1981. [Online]. Available: http://view.ncbi.nlm.nih.gov/pubmed/7265238

[25] P. H. Sellers, "The theory and computation of evolutionary distances: Pattern recognition," *Journal of Algorithms*, vol. 1, no. 4, pp. 359–373, December 1980.

[26] R. S. Boyer and J. S. Moore, "A fast string searching algorithm," *Communication of the ACM*, vol. 20, pp. 762–772, Oct. 1977.

[27] H. S. Warren, *Hacker's Delight (2nd Edition)*. Addison-Wesley, Sept. 2012.