Accelerating Scoring Computation of Smith-Waterman Algorithm with Mixed Word Length

Kazuki Yasui and Fumihiko Ino
Graduate School of Information Science and Technology, Osaka University
1-5 Yamadaoka, Suita, Osaka 565-0871, Japan
Email: {k-yasui, ino}@ist.osaka-u.ac.jp

***Abstract*—In this paper, we propose a graphics processing unit (GPU) accelerated Smith-Waterman (SW) method for aligning a short sequence with a long sequence. Our method deploys an 8-bit data structure to accelerate scoring computation, which can be classified into a memory-intensive operation. The proposed method is based on a mixed-word-length scheme capable of appropriately switching 8-bit mode and 32-bit mode to avoid integer overflow during computation. We integrate our data structure and scheme into a scan-based parallel approach to achieve high throughput for short sequences, which usually limit the parallelism inherent in the SW computation. Experimental results show that the proposed method is approximately three times faster than a previous scan-based approach that uniformly deploys 32-bit data structure. We also show that, for short sequences, scan-based parallel approaches are about twenty times faster than an antidiagonal-based parallel approach typically deployed for pairs of long sequences.**

1. Introduction

Pairwise sequence alignment is a process that identifies similar regions between two biological sequences such as nucleotide and protein sequences. This fundamental process is useful when analyzing functional, structural, and evolutional relationships between two sequences. There are two types of sequence alignment: global and local alignment. The former produces an end-to-end alignment of the sequences and the latter produces local regions (i.e., subsequences) with the highest similarity. Local alignment is typically used for constructing an evolutionary tree because this technique identifies regions where mutation like insertions or deletions of nucleotides occurred in the process of evolution.

The Smith-Waterman (SW) algorithm [1] is commonly used for finding the optimal local alignment. Given two sequences of length m and n ($\leq m$), the SW algorithm runs in $O(mn)$ time. The most time-consuming part of the SW algorithm is a matrix-filling phase, in which similarity scores of arbitrary regions of the sequences are computed as elements of a 2-D scoring matrix. Consequently, acceleration methods are required to apply the SW algorithm to growing biological sequences, which can have lengths of gigabase pairs (Gbp).

Owing to this growing demand, some researchers implemented the SW algorithm on accelerators such as the graphics processing unit (GPU) [2], [3], [4], [5] and Xeon Phi [6]. Among these promising accelerators, the GPU is a commodity device that has been accelerated memory- and compute-intensive applications in various fields [7], [8], [9]. A C-like programming framework called compute unified device architecture (CUDA) [10] is available for application developers to offload their time-consuming tasks from the CPU to the NVIDIA GPU.

As for the acceleration of local alignment, SW# [3] and CUDAlign [4] are useful for massively long sequences of up to 180 megabase pairs (Mbp) $\times$ 183 Mbp. These previous methods are based on an antidiagonal-based parallel approach [11], [12] in which $\min(m, n)$ matrix cells on an antidiagonal are computed in parallel. This approach is useful for sequences with similar lengths, i.e., $m \simeq n$. However, the exploited parallelism is significantly limited for short sequences, for example, 200-bp sequences enormously produced by short-read sequencing instruments. Achieving acceleration for such short sequences is a challenging issue for high-performance computing community because (1) the available data parallelism is limited due to small data and (2) recent hardware advances rely on increasing the number of cores rather than the frequency of clocks. A new strategy is required to deal with strong scaling cases, which consider fixed data size on variable numbers of processing elements.

In this paper, we present a GPU-accelerated SW method for aligning a short sequence with a long sequence. Our method deploys an 8-bit data structure to save the amount of memory access for matrix filling of the SW algorithm. Because the 8-bit data structure can suffer from integer overflow, we deploy a mixed-word-length scheme that appropriately selects either 8-bit mode or 32-bit mode according to runtime situation. We also integrate this idea into a scan-based parallel approach [13] in which $\max(m, n)$ matrix cells on a row are computed in parallel. The scan operation is further localized for optimization.

The rest of the paper is organized as follows. Section 2 gives an overview of the SW algorithm and the scan-based parallel approach [13]. Section 3 then describes our method and Section 4 shows experimental results. Finally, Section

5 concludes the paper with future work.

2. GPU-Accelerated Pairwise Alignment

The SW algorithm [1] is based on a dynamic programming approach that obtains the optimal local alignment between two sequences. Figure 1 illustrates how the SW algorithm computes the local alignment. This algorithm finds similar subsequences in two phases: (1) computation of a scoring matrix H and (2) backtracking from the matrix element with the highest score. As compared with the backtracking phase, the matrix-filling phase involves computation by more than an order of magnitude. Therefore, we focus on the matrix-filling phase, which we parallelize on the GPU. The backtracking phase can be rapidly processed on the CPU [14].

Let $A = a_1 a_2 \dots a_n$ denote a sequence of length n. Let $B = b_1 b_2 \dots b_m$ denote a sequence of length m ($\geq n$) to be compared with the sequence A. The SW algorithm then computes an $n \times m$ matrix H to obtain the similarity for any pair of subsequences. Let $E_{i,j}$ and $F_{i,j}$, where $1 \leq i \leq n$ and $1 \leq j \leq m$, be the maximum similarities involving the first i-th symbols in A and the first j-th symbols in B, respectively. The maximum similarity $H_{i,j}$ of two subsequences ending in a_i and b_j, respectively, is then recursively defined by

$$H_{i,j} = \max(0,\ E_{i,j},\ F_{i,j},\ H_{i-1,j-1} + S(a_i, b_j)), \quad (1)$$

where $S(a_i, b_j)$ represents the score of aligning a_i and b_j. The score is typically given as a substitution matrix S. Here, we define $S(a_i, b_j)$ as follows.

$$S(a_i, b_j) = \begin{cases} S_{match}, & \text{if } a_i = b_j, \\ S_{unmatch}, & \text{otherwise.} \end{cases} \quad (2)$$

Similarities $E_{i,j}$ and $F_{i,j}$ are given by

$$E_{i,j} = \max(H_{i,j-1} - G_s,\ E_{i,j-1} - G_e), \quad (3)$$
$$F_{i,j} = \max(H_{i-1,j} - G_s,\ F_{i-1,j} - G_e), \quad (4)$$

where G_s and G_e are penalties for opening a new gap and for extending an existing gap, respectively. The values for $H_{i,j}$, $E_{i,j}$, and $F_{i,j}$ are defined as zero if $i < 1$ or $j < 1$.

The penalties are called linear gap penalties if $G_s = G_e$. Otherwise, they are affine gap penalties. Our method deals with affine gap penalties. From biological point of view, staring a gap is more important than the extension of gaps, i.e., $G_s > G_e$.

2.1. Scan-based Parallelization

Khajeh-Saeed *et al.* [13] proposed a scan-based parallel approach for the SW algorithm. Their approach rewrites the recurrence relation such that matrix elements on the same row can be computed in parallel. As compared to the antidiagonal-based approach [11], [12], the exploited parallelism increases from $\min(m, n)$ to $\max(m, n)$. Similarly, the number of steps changes from $m+n-1$ steps to $n \log m$ steps. Therefore, this approach is suited for a pair of short and long sequences, i.e., $m > n$.

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

Figure 1. An overview of the SW algorithm. Given two DNA sequences TGTCGAT and CTGTAC with $S_{match} = 2$ and $S_{unmatch} = -1$, the scoring matrix H is computed according to a recurrence relation. After filling the scoring matrix, the backtracking procedure from the matrix element with the highest score produces the most similar subsequences: TGTC and TGTAC.

The scan-based approach fills the matrix in three steps by rewriting the original recurrences of Eqs. (1)–(4) in three equations. The first step partially computes the scoring matrix element $\tilde{H}_{i,j}$ as follows.

$$\tilde{H}_{i,j} = \max(H_{i-1,j-1} + S(a_i, b_j),\ F_{i,j},\ 0), \quad (5)$$

where $F_{i,j} = \max(F_{i-1,j},\ H_{i-1,j} - G_s) - G_e$, $1 \leq i \leq n$, and $1 \leq j \leq m$. Equation (5) indicates that the i-th row depends only on the $(i-1)$-th row. Therefore, this step can be parallelized by assigning the i-th row to $\max(m, n)$ processing elements.

The second step computes the row maximums:

$$\tilde{E}_{i,j} = \max_{1<k<j} (\tilde{H}_{i,j-k} - k \times G_e), \quad (6)$$

where $1 \leq i \leq n$ and $1 \leq j \leq m$. Equation (6) can be processed by performing a scan operation. Given a vector $\mathbf{x} = (x_1, x_2, \dots, x_n)$ of n elements, a scan operation with an associative binary operator $\oplus$ produces a vector $\mathbf{y} = (y_1, y_2, \dots, y_n)$, where $y_j = x_1 \oplus x_2 \oplus \dots \oplus x_j$ and $1 \leq j \leq n$. Equation (6) can be represented as a scan operation with the following associative binary operator $\oplus$:

$$x \oplus y = \max(x - G_e, y). \quad (7)$$

Similar to the first step, the second step can be processed by $\max(m, n)$ processing elements in parallel. As for GPU-based implementations, the Thrust library [15] is widely used to accelerate scan operations on the GPU.

Finally, the scoring matrix $H_{i,j}$ can be obtained by

$$H_{i,j} = \max(\tilde{H}_{i,j},\ \tilde{E}_{i,j} - G_s), \quad (8)$$

where $1 \leq i \leq n$ and $1 \leq j \leq m$.

The abovementioned three steps must be sequentially applied to every rows from top to bottom of the 2-D scoring matrix. Only two rows rather than the entire matrix must be stored in the GPU memory to find the maximum score.

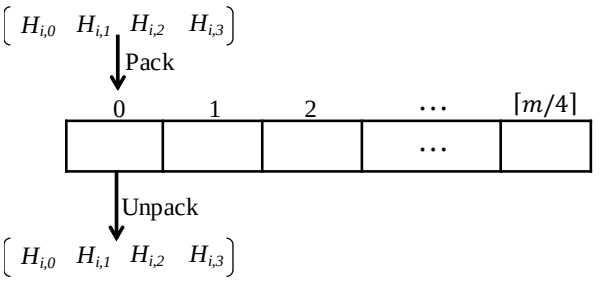


Figure 2. Proposed packed data structure. m matrix elements are stored as a 32-bit array of $\lceil m/4 \rceil$ elements. Reduced amount of memory access achieves acceleration although packing and unpacking procedures involve overheads.

```
1 int x = INT4_PACK(1, 2, 3, 4);
2 int y = INT4_PACK(5, 6, 7, 8);
3 int res;  // res = x + y = (6, 8, 10, 12)
4 asm("vadd4.u32.u32.u32.sat␣%0,␣%1,␣%2,␣%3;"
5       : "=r"(res) : "r"(x), "r"(y), "r"(0));
```

Figure 3. Example of vector addition of 8-bit data structure. Addition is implemented with inlining a PTX instruction into the CUDA code.

3. Proposed Method

The proposed alignment method focuses on a pair of short and long sequences. We integrate three optimization strategies into the scan-based parallel approach [13].

1) 8-bit data structure for saving the amount of memory access.
2) Mixed-word-length scheme for avoiding integer overflow.
3) Localized parallel scan for reducing global synchronization.

3.1. 8-bit data structure

Our data structure packs four matrix elements into a 32-bit word on global memory [10] (Fig. 2). Therefore, the data structure replaces a naive array of m elements with that of $\lceil m/4 \rceil$ elements. This packed data structure must be referred with 32-bit word access to reduce the amount of memory access by a quarter. In other words, four consecutive elements on a row are assigned to every thread, allowing threads to access four consecutive elements at a time.

The packed data should be directly processed without any overhead. To realize such direct operations, we utilize single-instruction, multiple-data (SIMD) instructions that operate four 8-bit values at a time. Figure 3 shows an example of vector addition implemented by inlining a parallel thread execution (PTX) assembly statement [10] into the CUDA code. In addition to this addition operator, the CUDA provides a set of fundamental operators, such as subtraction and maximization, needed for processing Eqs. (5)–(8) with the 8-bit data structure.

3.2. Mixed-word-length scheme

The 8-bit data structure limits the alignment scores in the range $[0, 255]$. Consequently, integer overflow occurs if final scores are greater than 255. Therefore, we have to avoid integer overflow by appropriately selecting the 8-bit mode or the 32-bit mode according to the runtime situation.

The appropriate mode can be selected for every row before proceeding to the next row. In more detail, given the i-th row, the maximum value on the $(i+1)$-th row can be estimated using the substitution matrix S. That is, our method avoids integer overflow by selecting the 32-bit mode if

$$\max_{0<j<m} H_{i,j} + S_{match} > 255, \tag{9}$$

The overhead for this runtime selection is negligible because the scan-based parallel scheme computes the maximum score for every row, as shown in Eq. (8).

On the other hand, our method switches from the 32-bit mode to the 8-bit mode if

$$\max_{0<j<m} H_{i,j} + S_{match} \leq 255 - T, \tag{10}$$

where $T = 10$ is a threshold value determined experimentally. We introduce this threshold to prevent frequent switches between 8-bit and 32-bit modes because every switch requires data conversion between 8-bit format and 32-bit format. This conversion involves additional overheads, but reducing memory access amount is more important to accelerate memory-intensive operations.

Algorithm 1 shows our mixed-word-length scheme, which computes the scoring matrix from the top to the bottom rows. As shown in line 4, the scoring matrix is computed with the appropriate data structure to save the amount of memory access. However, the data conversion at lines 12 and 15 is required to avoid integer overflow. The highest similarity score and its position are computed at lines 5–7. First, the highest scores within each thread block [10] are computed on the GPU. Secondly, these scores are transferred to the main memory to reduce them into the final score on the CPU.

3.3. Localized parallel scan

The previous method [13] scans the entire data to compute Eq. (6). However, this global scan can be localized for acceleration because the matrix element $H_{i,j}$ is affected by some previous p symbols rather than the entire of $\max(m, n) = n$ symbols. Given a vector $\mathbf{x} = (x_1, x_2, \ldots, x_n)$ of n elements, a local scan operation with an associative binary operator $\oplus$ produces a vector $\mathbf{y} = (y_1, y_2, \ldots, y_n)$, where $y_j = x_{j-p} \oplus x_{j-p+1} \oplus \ldots \oplus x_j$ and $p \leq j \leq n$; the parameter p, the maximum distance that a score at arbitrary position can affect, is given by

$$p = \lfloor n \times S_{match}/G_e \rfloor. \tag{11}$$

Equation (11) can be obtained as follows. Equation (7) indicates that the similarity score can decrease by G_e when

Algorithm 1 Mixed-word-length scheme

Input: Sequences $A = a_1 a_2 \dots a_n$, $B = b_1 b_2 \dots b_m$, and substitution matrix S
Output: Highest similarity score H and its position P

$mode \leftarrow$ "8BIT"; ▷ initialize with 8-bit mode
$H \leftarrow 0$; $P \leftarrow (0, 0)$;
for $k \leftarrow 0$ to n **do** ▷ for each row
    Compute the k-th row with mode $mode$ using Eqs. (5)–(8) on the GPU;
    Gather the highest scores within each thread block on the GPU;
    Transfer the candidate scores from the GPU to the CPU;
    Find the highest score H_k and its position P_k;
    if $H_k > H$ **then** ▷ update the highest score
        $H \leftarrow H_k$; $P \leftarrow P_k$;
    end if
    if ($mode ==$ "8BIT") and (Eq. (9) is true) **then**
        Convert 8-bit data to 32-bit data on the GPU;
        $mode \leftarrow$ "32BIT";
    else if ($mode ==$ "32BIT") and (Eq. (10) is true) **then**
        Convert 32-bit data to 8-bit data on the GPU;
        $mode \leftarrow$ "8BIT";
    end if
end for

processing a binary operator $\oplus$. On the other hand, the highest possible score on matrix H can reach $n \times S_{match}$. Therefore, this score $n \times S_{match}$ decreases to $H(k) = n \times S_{match} - k \times G_e$ when processing k operators. The parameter p is given such that $H(p) \geq 0$, which results in Eq. (11).

We implemented this idea by extending a GPU-accelerated string search implementation [9] that localized scan computation. Figure 4 shows how GPU threads process local scan in parallel. The local scan is processed in four steps. Without loss of generality, 32-bit data here is assumed to simplify the description.

1) Scan within segments. For example, the third thread block in Fig. 4 computes a vector $(x_8, x_{8\text{-}9}, x_{8\text{-}10}, x_{8\text{-}11})$ for its responsible segment $(x_8, x_9, x_{10}, x_{11})$.
2) Gather representatives from segments. To perform scan among segments, we collect the last elements from locally scanned segments. In Fig. 4, four representatives $x_{0\text{-}3}$, $x_{4\text{-}7}$, $x_{8\text{-}11}$, and $x_{12\text{-}15}$ are copied to a temporal buffer.
3) Scan for gathered values. Similar to the first step, local scan is applied to the temporal buffer. After that, the buffer holds scanned results $(x_{0\text{-}3}, x_{0\text{-}7}, x_{0\text{-}11}, x_{0\text{-}15})$, each starting from the initial element. This step can be omitted when the segment size is greater than p.
4) Scatter scanned values to segments. The scanned results obtained at the third step are scattered to all segments. The scattered results are then added to the local segments. Finally, every element holds the scanned result starting from the initial element.

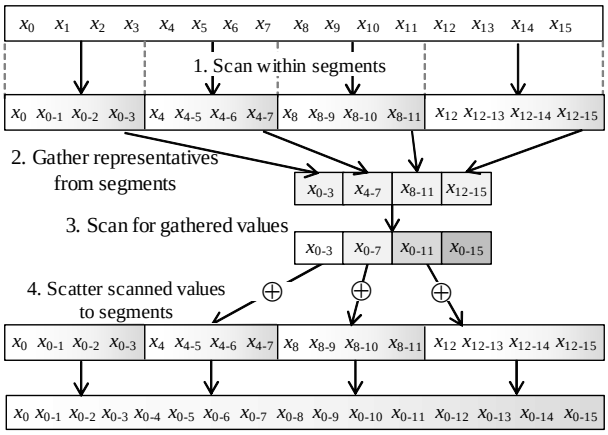
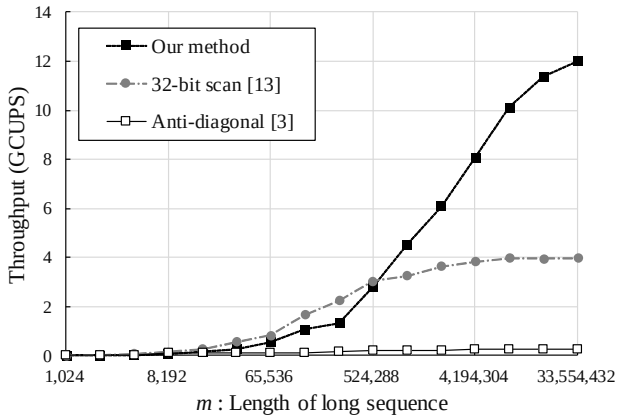


Figure 4. Localized parallel scan on the GPU. The input array is scanned in four steps. Barrier synchronization is required between steps.

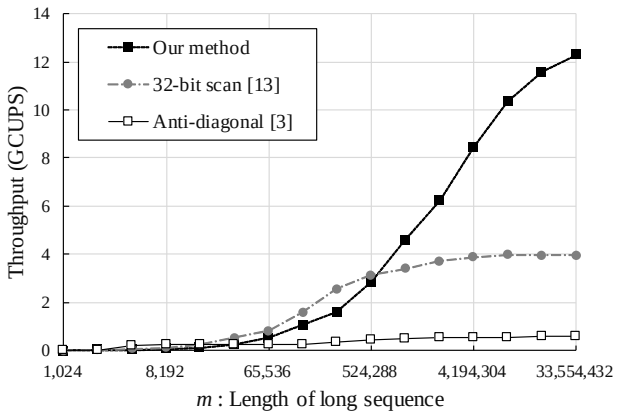
4. Experimental Results

We evaluated the proposed method in terms of execution time and compared the method with previous GPU-accelerated methods [3], [13]. We measured the alignment throughput in *cell updates per second* (CUPS), given by mn/t, where t is the execution time that contains (1) the time t_1 spent on the GPU, (2) that t_2 on the CPU, and (3) that t_3 needed for data transfer between the main memory and the GPU memory.

For experiments, we used a desk-top PC equipped with an Intel Core i7-6700 3.4 GHz CPU, 32 GB main memory, and an NVIDIA GeForce GTX 980ti GPU with 6 GB device memory. All implementations ran on Windows 10. The



(a)



(b)

Figure 5. Alignment throughput with different sequence length m. Results for (a) $n = 128$ and (b) $n = 256$. The horizontal axis is given in a logarithmic scale.

methods were implemented using Visual Studio 2013 and CUDA 7.5 [10].

As for long sequences, we used the whole genome shotgun sequence of Homo sapiens chromosome 21 (accession number: [GenBank CM000511]). For short sequences, we used a segment of Influenza A virus gene (accession number: [GenBank FJ966086]). Alignments were carried out with a affine gap penalty ($G_s = 5$ and $G_e = 1$) and a scoring matrix S such that $S_{match} = 1$ and $S_{unmatch} = -2$.

Figure 5 shows the measured alignment throughputs with different lengths m of sequences ranging from 1 Kbp to 33 Mbp. Our method was faster than the previous 32-bit scan-based method [13] when $m > 524$K. For such short sequences, i.e., short vs short sequences, the overhead of data transfer limited the alignment throughput of our method. However, the speedup over the previous method reached a factor of 3.1 when $n = 256$ and $m = 33$M. Furthermore, the proposed method was 21.2 times faster than the antidiagonal-based method [3]. The highest throughput of 12.3 GCUPS was obtained when $n = 256$ and $m = 33$M.

TABLE 1. BREAKDOWN OF EXECUTION TIME FOR PROPOSED AND PREVIOUS METHODS [13] WITH $n = 256$. t_1, t_2, AND t_3 REPRESENT CPU TIME, GPU TIME, AND CPU-GPU TRANSFER TIME IN MILLISECONDS, RESPECTIVELY.

Breakdown	Short ($m = 100$K)		Long ($m = 33$M)	
	Proposed	Previous	Proposed	Previous
t_1	37.0	44.0	670.0	2217.0
t_2	3.3	1.2	7.1	6.2
t_3	13.0	0.3	30.0	2.3
Total	53.3	45.5	707.1	2225.5

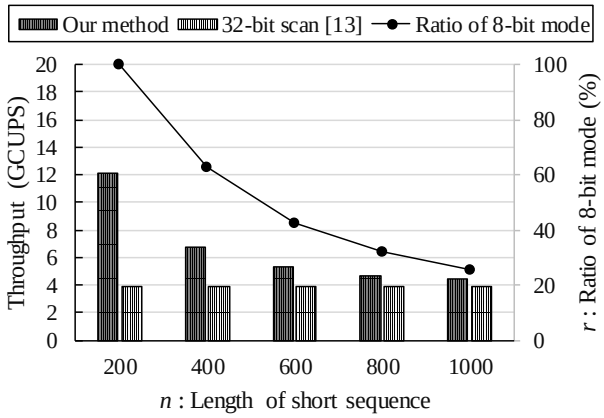


Figure 6. Alignment throughput with different ratio r of 8-bit mode over 32-bit mode. $r = 0$ and $r = 100$ mean that the alignment is fully processed with 32-bit mode and 8-bit mode, respectively.

We next investigated the breakdown of execution time t for short and long sequences (Table 1). When $m = 100$K and $n = 256$, i.e., for short sequences, our method was slower than the scan-based previous method. In contrast, our method was faster than the previous method for long sequences. For both cases, our method achieved shorter GPU time t_1. However, the CPU-GPU transfer time t_3 increased from 2.3 ms to 30.0 ms when $m = 33$M and from 0.3 ms to 13.0 ms when $m = 100$K. This increase was due to the data transfer occurred at line 6 in Algorithm 1. That is, the proposed method transferred the highest scores to switch the data structure according to Eqs. (9) and (10) computed on the CPU.

Finally, we counted the number l of matrix elements processed with the 8-bit mode. According to this number l, we computed $r = l/mn$, i.e., the rate of the 8-bit mode over the 32-bit mode (Fig. 6). To investigate the impact of the 8-bit mode, we varied the rate r by aligning the genome shotgun sequence of Homo sapiens chromosome 21 with itself; we used the sequence as a long sequence while its subsequence as a short sequence of 200–1000 bp. In Fig. 6, the proposed method shows a linear relationship between the ratio r and the throughput. That is, the achieved acceleration over the previous method comes from the 8-bit data structure, which reduces the amount of memory access. In particular, the proposed method is effective when n is small or the sequences are not highly similar.

5. Conclusion

We have presented a mixed-word-length method capable of accelerating the SW algorithm for a pair of short and long sequences. The key idea for realizing acceleration is to integrate an 8-bit data structure into a scan-based parallel scheme [13]. Our method not only saves the memory bandwidth but also avoids integer overflow by appropriately switching 8-bit mode and 32-bit mode.

Experimental results show that the proposed method is three times and twenty times faster than the scan-based method and the antidiagonal-based method, respectively. Therefore, the proposed method is useful for accelerating SW alignment for a pair of short and long sequences.

Future work includes further optimization such as pruning of scoring computation and reduction of memory access.

ACKNOWLEDGMENTS

This study was supported in part by the Japan Society for the Promotion of Science KAKENHI Grant Numbers 15H01687, 16H02801 and 15K12008.

References

[1] T. F. Smith and M. S. Waterman, "Identification of common molecular subsequences," *J. Molecular Biology*, vol. 147, no. 1, pp. 195–197, Mar. 1981.

[2] F. Ino, Y. Munekawa, and K. Hagihara, "Sequence homology search using fine grained cycle sharing of idle GPUs," *IEEE Trans. Parallel and Distributed Systems*, vol. 23, no. 4, pp. 751–759, Apr. 2012.

[3] M. Korpar and M. Šikić, "SW#–GPU-enabled exact alignments on genome scale," *Bioinformatics*, vol. 29, no. 19, pp. 2494–2495, Oct. 2013. [Online]. Available: https://sourceforge.net/swsharp/code/

[4] E. F. de O. Sandes, G. Miranda, A. C. M. A. de Melo, X. Martorell, and E. Ayguadé, "CUDAlign 3.0: Parallel biological sequence comparison in large GPU clusters," in *Proc. 14th IEEE/ACM Int'l Symp. Cluster, Cloud and Grid Computing (CCGrid'14)*, May 2014, pp. 160–169.

[5] D. Okada, F. Ino, and K. Hagihara, "Accelerating the Smith-Waterman algorithm with an interpair pruning method for all-pairs comparison of base sequences," *BMC Bioinformatics*, vol. 16, no. 321, Oct. 2015, 15 pages.

[6] E. Rucci, C. Garcia, G. Botella, A. D. Giusti, M. R. Naiouf, and M. Prieto-Matias, "First experiences accelerating Smith-Waterman on Intel's Knights Landing processor," in *Proc. 17th Int'l Conf. Algorithms and Architectures for Parallel Processing (ICA3PP'17)*, Aug. 2017, pp. 569–579.

[7] J. D. Owens, M. Houston, D. Luebke, S. Green, J. E. Stone, and J. C. Phillips, "GPU computing," *Proceedings of the IEEE*, vol. 96, no. 5, pp. 879–899, May 2008.

[8] K. Ikeda, F. Ino, and K. Hagihara, "Efficient acceleration of mutual information computation for nonrigid registration using CUDA," *IEEE J. Biomedical and Health Informatics*, vol. 18, no. 3, pp. 956–968, May 2014.

[9] Y. Mitani, F. Ino, and K. Hagihara, "Parallelizing exact and approximate string matching via inclusive scan on a GPU," *IEEE Trans. Parallel and Distributed Systems*, vol. 28, no. 7, pp. 1989–2002, Jul. 2017.

[10] NVIDIA Corporation, "CUDA C Programming Guide Version 8.0," Jan. 2017. [Online]. Available: http://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf

[11] W. Liu, B. Schmidt, G. Voss, and W. Müller-Wittig, "Bio-sequence database scanning on a GPU," in *Proc. 20th IEEE Int'l Parallel and Distributed Processing Symp. (IPDPS'06)*, Apr. 2006.

[12] Y. Munekawa, F. Ino, and K. Hagihara, "Design and implementation of the Smith-Waterman algorithm on the CUDA-compatible GPU," in *Proc. 8th IEEE Int'l Conf. Bioinformatics and Bioengineering (BIBE'08)*, Oct. 2008, 6 pages (CD-ROM).

[13] A. Khajeh-Saeed, S. Poole, and B. Perot, "Acceleration of the Smith-Waterman algorithm using single and multiple graphics processors," *J. Computational Physics*, vol. 229, no. 11, pp. 4247–4258, Jun. 2010.

[14] W. Liu, B. Schmidt, G. Voss, and W. Müller-Wittig, "Streaming algorithms for biological sequence alignment on GPUs," *IEEE Trans. Parallel and Distributed Systems*, vol. 18, no. 9, pp. 1270–1281, Sep. 2007.

[15] N. Bell and J. Hoberock, *Thrust: A Productivity-Oriented Library for CUDA*. San Mateo, CA: Morgan Kaufmann, Jan. 2011, ch. 26, http://thrust.github.io/.