Fast Sequence Alignment Method Using CUDA-enabled GPU

Yeim-Kuan Chang
Department of Computer Science and Information Engineering
National Cheng Kung University
Tainan, Taiwan
ykchang@mail.ncku.edu.tw

De-Yu Chen
Department of Computer Science and Information Engineering
National Cheng Kung University
Tainan, Taiwan
p76981293@mail.ncku.edu.tw

***Abstract*—Sequence alignment is a task that calculates the degree of similarity between two sequences. Given a query sequence, finding a database sequence which is most similar to the query by sequence alignment is the first step in bioinformatics research. The first sequence alignment algorithm was proposed by Needleman and Wunsch. They got the optimal global alignment by using dynamic programming method. Afterwards, Smith and Waterman proposed the optimal local alignment. However, the number of sequences in the database increases exponentially every year and the cost of these two algorithms is expensive in terms of running time and memory space. Thus, accelerating the sequence alignment algorithm has become the trend in recent years. In this paper, we present a fast sequence alignment method using CUDA-enabled GPU. We first redefine the recursive formula of Smith-Waterman algorithm so that one row of the matrix can be calculated in parallel. Then, we use the prefix max scan method to reduce the computation complexity for each row. In addition, only on-chip shared memory is used for reducing the penalty of memory accesses. Experimental results show that the proposed method is in average 50 times faster than implementation of Smith-Waterman based on CPU, 2~4 times faster than other GPU-based versions of Smith-Waterman algorithm.**

Keywords: sequence alignment, GPU, dynamic programming, CUDA, prefix max scan

I. INTRODUCTION

The bioinformatics integrates the applied mathematics, information, statistics and computer science methods to study the biology problem. Such research areas include sequence analysis, genome annotation, computational evolutionary biology, analysis of protein expression and prediction of protein structure etc. The large number of biological data is stored in the form of DNA, RNA and protein sequences. The sequences are printed abutting one another without gaps, and made of their respective letters set shown below:

- DNA : {A, C, G, T}
- RNA : {A, C, G, U}
- Protein : {A, R, N, D, C, Q, E, G, H, I, L, K, M, F, P, S, T, W, Y, V}

Sequence alignment is a task that calculates the degree of similarity between two sequences by arranging sequences of DNA, RNA or proteins, and gaps are allowed to be inserted in sequences for increasing similarity. In other words, letters of one sequence are aligned against letters of another, and only a selected set of operations is allowed at each aligning position: (1) match one letter with another, (2) mismatch one letter with another, (3) align one letter with a gap (denoted by "—") symbol. For example, consider the two sequences S_1 = AGGCCTATG and S_2 = ACG GCCTAATG. One possible alignment is shown in Figure 1 (The dots indicate the match). High sequence similarity means the structure and function will be similar, and they may be derived from a common or similar ancestor. By sorting and classifying these similar structures, we can infer the evolution of a period. Therefore, biologists in the first step for bioinformatics are using sequence alignment tool to search the sequence database.

In addition, sequence alignment can be further classified as global and local alignment. The global alignment that attempts to align every letter of both sequences, on the contrary, local alignment finds the similar region within two given sequences. A first sequence global alignment algorithm was designed by Needleman and Wunsch in 1970 [1], called NW algorithm, which is based on dynamic programming method. Then, Smith and Waterman proposed the local alignment version of NW in 1981 [2]. Both two algorithms produce the optimal alignment, but the cost of two algorithms is expensive in terms of running time and memory space. Thus, heuristic algorithms FASTA [3] and BLAST [4] were proposed in 1985 and 1990, respectively. Unlike the NW or SW algorithm, those heuristic algorithms directly compute the local alignment by producing seeds and extending them rather than using dynamic programming. Unfortunately, heuristic methods loose the optimality guarantee. Many researchers are trying to accelerate the NW or SW algorithm by a variety of different tools. Those tools such as FPGA [5], Cell/BE [6], GPU [7] and OpenMP [8], provides the effectiveness to significantly reduce the runtime

S_1	=	A	—	G	G	C	C	T	A	T	G	—
		.		.	.	.	.	.	.			
S_2	=	A	C	G	G	C	C	T	A	A	T	G

Figure 1. An example of sequence alignment

for sequence alignment algorithm. GPGPU (General Purpose computing on Graphics Processing Units) is a technique of using a GPU to perform computation in applications traditionally handled by the CPU. The general calculation often has nothing to do with the graphics. Because modern graphics processors have parallel processing capability and powerful programmable pipelines, stream processors can handle non-graphics data. Especially in the face of a lot of data that can be processed in parallel, the general purpose graphics processors perform far beyond the traditional central processing applications. CUDA (Compute Unified Device Architecture) [9] is one of GPGPU application. It enables users to write programs to execute on the NVIDIA's GPU. Liu et al [10] first developed the parallelism version of SW algorithm that runs on GPU. Since their work is done before the dawning of CUDA, they employed the OpenGL library, namely a graphics library, to implement the SW algorithm on the GPU. They provided speedups of more than one order of magnitude with respect to optimized CPU implementations. Yongchao Liu et al [11] presented two parallel implementations of SW algorithm by using CUDA, namely CUDA-SW++. CUDASW++ uses one of two kernel functions to align a query sequence to another sequence. Each kernel uses a different strategy to find the optimal local alignment. The first kernel and second kernel are called the inter-task and intra-task, respectively. The length of the database sequence determines which kernel to be used. According to their performance issues, CUDASW++ reaches 9~16 giga cell updates per second (GCUPS).

In this paper, we present a fast sequence alignment method using CUDA-enabled GPU. We first try to redefine the re-cursive formula of SW algorithm so that one row of the matrix can be calculated in parallel. Then, we use the prefix max scan method to reduce the computation complexity. In addition, during the computation, only on-chip shared memory is used for reducing the penalty of memory accesses. Experimental results show that the proposed method is in average 50 times faster than implementation of SW algorithm based on CPU, in average 2~4 times faster than other CUDA-based version of Smith-Waterman algorithm.

The remainder of this paper is organized as follows: Section II reviews the previous sequence alignment algorithms. Section III features the CUDA architecture. Section IV then describes our proposed method and Section VI shows experimental results. Finally, Section VII concludes the paper.

II. RELATED WORK

In this section, we first describe the SW algorithm, and then show the several parallelism version of SW algorithm that has been proposed.

The NW algorithm [1] was modified by Smith and Waterman [2] to deal with local alignment. It guarantees that the alignment is optimal. Like NW, SW algorithm is also based on dynamic programming with quadratic time and space complexity. In order to quantify an alignment, SW algorithm uses a scoring mechanism. Given two sequences S_1 and S_2 of length m and n respectively, based on the relationship between pair of letters in an alignment we mentioned before, there are three types of score as follows:

		C	C	A	C	C	G	G	G	G	C
	0	0	0	0	0	0	0	0	0	0	0
G	0	0	0	0	0	0	6	6	6	6	2
C	0	9	9	5	9	9	5	3	3	3	15
A	0	5	9	13	9	9	9	5	3	3	11
G	0	1	5	9	10	6	15	15	11	9	7
G	0	0	1	5	6	7	12	21	21	17	13
G	0	0	0	1	2	3	13	18	27	27	23
T	0	0	0	0	0	1	9	14	23	25	26
T	0	0	0	0	0	0	5	10	19	21	24
A	0	0	0	4	0	0	1	6	15	19	21
G	0	0	0	0	1	0	6	7	12	21	17

Figure 2. Example of computation of the score matrix for S_1 = GCAGGGTTAG and S_2 = CCACCGGGGC. The gap penalty is −4, substitution matrix is BLOSUM 62, and the alignment score is 27.

- Match score if $S_1[i] = S_2[j]$.
- Mismatch score if $S_1[i] \neq S_2[j]$
- Gap penalty score if $S_1[i]$ = "—" or $S_2[j]$ = "—", or both.

The match and mismatch scores obtained by looking up a substitution matrix. A substitution matrix is a square matrix in which each cell contains a score for the corresponding pair of based. PAM (Percentage of Acceptable point Mutations) series of matrices [12] and BLOSUM (BLOcks Substitution Matrix) series of matrices [13] are two popular substitution matrices. In addition, gap penalty score usually is a negative number.

Let H be a score matrix of size $(m+1)(n+1)$. The recursive formula is defined as:

$$H[i][j] = \max \begin{cases} H[i-1][j-1] + sbt(S_1[i], S_2[j]) \\ H[i-1][j] + g \\ H[i][j-1] + g \end{cases}$$

where $1 \le i \le m$, $1 \le j \le n$, $sbt(S_1[i], S_2[j])$ gets the match or mismatch score, g is the score of gap penalty. The recursive formula are initialized as $H[i][0] = H[0][j] = 0$ for $0 \le i \le m$ and $0 \le j \le n$. If all the matrix cells have been computed, the alignment score is the maximum score in the matrix H. Figure 2 shows an example of computation of the score matrix for S_1 = GCAGGGTTAG and S_2 = CCACCGGGGC. In Figure 2, g=−4, the substitution matrix is BLOSUM 62, and the alignment score is 27.

Since the time and space complexity of SW algorithm are O(mn), in dealing with the long sequence, SW algorithm will not be ideal. Parallel computing is a very common alternative to obtain high performance in execution of SW algorithm. In the following sections, we will describe three existing parallelism mechanisms in the computation of SW algorithm based on GPU.

The first mechanism is to exploit the parallel characteristics of cells in the anti-diagonals. That is, each anti-diagonal cell can be computed independently of the others. As shown in Figure 3, the intermediate results vectors for the (i–

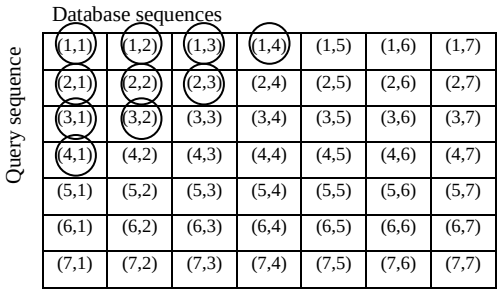

Figure 3. First parallelism mechanism

2)th, (i–1)th and i^{th} anti-diagonals are allocated with size $min(|S_1|,|S_2|)$. When the computation of the i^{th} anti-diagonal is completed, it swaps the intermediate results vectors. However, this mechanism has memory problems due to the non-uniform access pattern of data in memory. Liu et al [10] speeded up the computation of SW algorithm by implementing the first mechanism on the GPU. A peak performance of 0.7 GCUPS at a query of length 4092 is reported.

The second mechanism is shown in Figure 4. The matrix is computed vector by vector in order parallel to the query sequence. To compute a vector, all values from previous one are needed, and stored in the linear memory. More precisely, it uses two memory spaces: one for the previous values and one for the newly computed ones. At the end of each vector, it swaps them and so on. According to the data size read from memory, this mechanism gets 4 or 8 vector values at a time. However, it still has to handle data dependencies within the vector. Rognes and Erling [14] developed the second mechanism on common microprocessors, and their results show a six-fold speed-up relative to the first mechanism. Next, Manavski et al [7] presented a CUDA-based implementation for second mechanism. Their performance reaches a peak of 1.8 GCUPS using a GeForce 8800 GTX card. Yongchao Liu et al [11] presented the best CUDA implementation of first and second mechanisms, namely CUDASW++. According to their performance, they reach 9~16 GCUPS.

The last parallelism mechanism combines the first and second mechanisms. As shown in Figure 5, this mechanism processes the vector in parallel to the query sequence and at the same time it processes the vector from the next column in anti-diagonal direction. However, this method consumes a large amount of memory space. Sanchez et al [15] implemented the third mechanism on GPU, and they claimed that the

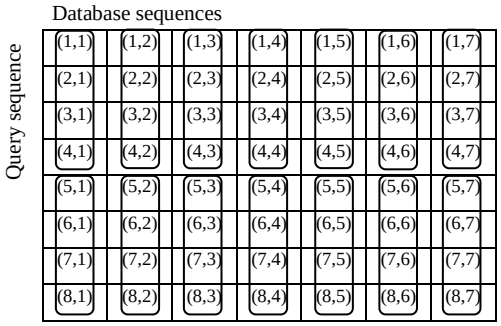

Figure 4. Second parallelism mechanism

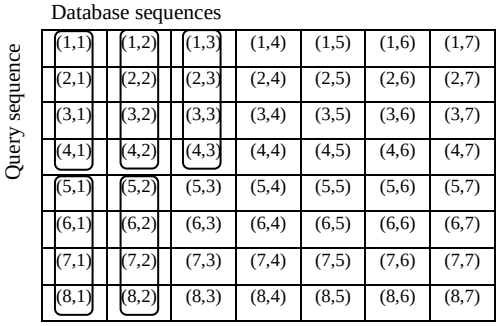

Figure 5. Third parallelism mechanism

implementation obtains a 30% reduction in the execution time relative to the first and second mechanisms.

III. CUDA

CUDA (Compute Unified Device Architecture) [9] is a technique that enables user to write programs to execute on the CUDA-enabled GPU. CUDA was developed by NVIDIA in November 2006. CUDA allows us to directly access their specific graphics hardware and efficiently use massive threads. In this section, we will describe the programing model and hardware model.

A. Programming model

The programming language in CUDA is called CUDA C/C++. It consists of a minimal set of extensions to the C/C++ language and a runtime library. It provides a simple path for users familiar with the C/C++ programming language to easily write programs for execution by the GPU.

Overall, the CUDA program is divided into two parts, the serial codes and parallel codes. The serial codes are executed on the conventional processor while parallel codes are executed on the GPU. The parallel codes are written in *kernel* function that runs in every thread. The kernel function call looks like a C language function call except the execution configuration is added between triple angle brackets "<<<" and ">>>" as shown below.

Kernel<<< *nBlocks, nThreadsPerBlock* >>> (*paramrter list*)

The two parameters between the angle brackets define the number of blocks and the number of threads per block. The total number of simultaneously running threads for a given kernel is the product of these two parameters. There is a limit to the number of threads per block, since all threads of a block are expected to reside on the same processor core and must share the limited memory resources of that core. On current GPUs, a block may contain up to 1024 threads. Blocks are organized into a one-dimensional, two-dimensional, or three-dimensional grid of blocks as illustrated in Figure 6. The number of blocks in a grid is usually dictated by the size of the data being processed or the number of processors in the system, which it can greatly exceed. The programming model also assumes that both the conventional processors and the GPU maintain their own separate memory spaces in DRAM, referred to as host memory and device memory, respectively. Therefore

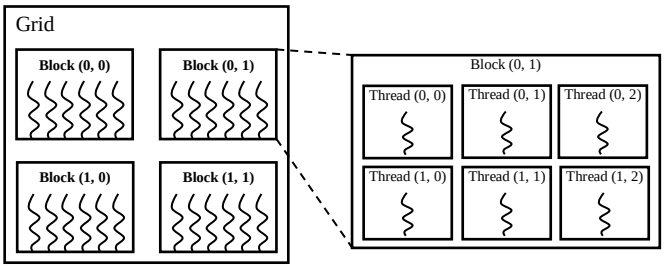

Figure 6. Grid of Thread Blocks

, a CUDA program manages the global, constant, and texture memory spaces visible to kernels through calls to the CUDA runtime. In order to write efficient CUDA programs, it is important to understand the different types of memory modules in CUDA.

(1) *Read-write per thread register*:

For each thread, it has private register. The register is fastest but very limited size.

(2) *Read-write per thread local memory*:

The local memory is another size solution for register limited size. However, access to local memory is as expensive as access to global memory.

(3) *Read-write per block shared memory*:

Shared memory is fast on-chip memory of limited size (16KB per block). Accessing the shared memory is as fast as accessing a register as long as there are no bank conflicts. It is also used to share information between the threads of a block.

(4) *Read-only per grid constant memory*:

Constant memory is a cache that is optimized for the case when all threads read from the same location. Essentially, data can be broadcast from constant memory to all threads with one cycle of latency when there is a cache hit.

(5) *Read-only per grid texture memory*

Texture memory is large (usually depending on the size of global memory) and is cached. It can be read from kernel using texture fetching device function. It is possible, through the judicious use of the caching behavior of texture memory to avoid the bandwidth limitations of global memory and greatly accelerate GPU application performance.

(6) *Read-write per grid global memory*

Global memory is by far the largest memory space, with capacities measured in gigabytes (typically 1GB up). However, since the global memory belongs to off-chip memory; it has high latency, low bandwidth. The effective bandwidth of global memory depends heavily on the memory access pattern, e.g. coalesced access.

B. Hardware model

Figure 7 illustrates an overview of the hardware model in CUDA. This model mainly consists of GPU itself and off-chip device memory. The GPU has a set of Multiprocessors (MPs);

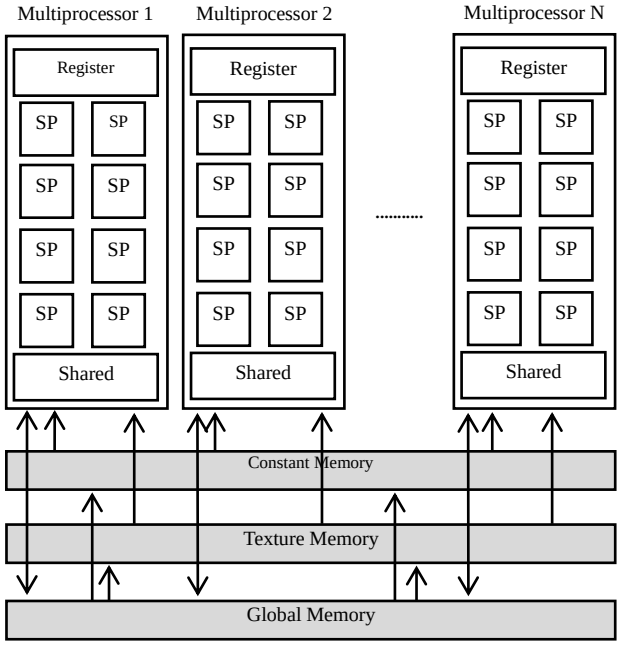

Figure 7. CUDA hardware model

each MP includes a set of stream processors (SPs) and shared memory. During kernel execution, the blocks are assigned to MPs and threads within a block are performed by SPs.

IV. Proposed method

In this section, we will describe the proposed method. Suppose that there are *N* sequences in the database. The mission is to find a database sequence which is mostly similar to the query sequence. The alignment task is defined to be the operations needed for computing the alignment scores of two sequences. Figure 8 shows our idea. Since there is no data dependency between different alignment tasks, we exploit one CUDA block to process each alignment task. It seems possible that more than one processing block can be assigned to compute one alignment task. However, CUDA does not support the synchronization between the blocks. Although the programmer can create synchronization scheme by using global memory, but it will pay a significant performance penalty due to access latency of global memory.

A. Redefine the recursive formula

We first try to redefine the recursive formula of Smith-Waterman (SW) algorithm so that one row of the matrix can be

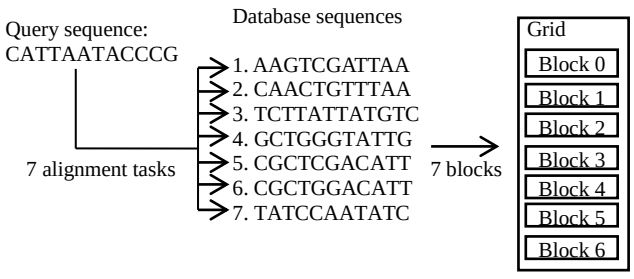

Figure 8. One alignment task is processed by one block

calculated in parallel. Given two sequences S_1 and S_2 of length m and n, respectively, let H be the score matrix of size $(m+1)\times(n+1)$. The recursive formula is as follows:

$$H[i][j] = \max\begin{cases} H[i-1][j-1] + sbt(S_1[i], S_2[j]) \\ H[i-1][j] + g \\ H[i][j-1] + g \end{cases},$$

where $1 \le i \le$ m, $1 \le j \le$ n, $sbt(S_1[i], S_2[j])$ gives the match score when $S_1[i] = S_2[j]$ or the mismatch score when $S_1[i] \neq S_2[j]$, and g is the score of gap penalty. Our target is to parallelize the score computation of one row of the matrix at each step. We first show how to compute the scores of the first row (gray cells) of the matrix below, by assuming the database sequence is of length 8.

		b_1	b_2	b_3	b_4	b_5	b_6	b_7	b_8
	0	0	0	0	0	0	0	0	0
a_1	0								
a_2	0								
a_3	0								
a_4	0								

$$H[1][j] = \max\begin{cases} H[0][j-1] + sbt(a_1, b_j) \\ H[0][j] + g \\ H[1][j-1] + g \\ 0 \end{cases}, \quad for \quad j = 1..8$$

Obviously, there is a data dependency between the cells. In order to compute the scores in parallel, we have the following new formula:

$$H'[1][j] = \max\begin{cases} H[0][j-1] + sbt(a_1, b_j) \\ H[0][j] + g \\ 0 \end{cases}, \quad for \quad j = 1..8$$

and

$$H[1][j] = \max\begin{cases} H'[1][j] \\ H[1][j-1] + g \\ 0 \end{cases}, for \quad j = 1..8.$$

Next, the expanded formula is as follows.

$$H[1][j] = \max\begin{cases} H'[1][j] \\ H'[1][j-1] + g \\ H'[1][j-2] + 2\times g \\ \ldots \\ H'[1][1] + (j-1)\times g \\ H[1][0] + j\times g \end{cases}, \quad for \quad j = 1..8.$$

After expansion, it's clear that there is no data dependency between all 8 of the first row of H matrix cells if H' values can be computed in advance. Based on this observation, the final three formulas are defined as follows. First, the formula (*a*) can be performed by the threads assigned to the cells in parallel. After the first step is finished, formulas (*b*) and (*c*) are then executed. The scores of all the cells in the row are completed.

$$(a)H'[i][j] = \max\begin{cases} H[i-1][j-1] + sbt(S_1[i], S_2[j]) \\ H[i-1][j] + g \\ 0 \end{cases}$$

$$(b)L[i][j] = \max_{0<k\le j}\begin{cases} H'[i][j] \\ H'[i][j-k] + k * g \end{cases}$$

$$(c)H[i][j] = \max\begin{cases} L[i][j] \\ H[i][0] + j * g \end{cases}$$

Thus, the computations of the next row of the matrix can be carried out.

We are now able to compute all the scored in one row of the matrix in parallel. However, the time complexity of this method is the same as the SW algorithm. The main reason is that the computation time of one row of the matrix depends on the last cell. The time complexities of the formulas (*a*) and (*c*) are O(1). The last cell requires O(n) time to perform formula (*b*). The total computation time for the matrix is O(n)×m = O(mn). This does not get any performance improvement. In order to solve this problem, we use prefix max scan to speed up the computation of the formula (*b*). Given the input array[a_1, …, a_n], the *prefix max scan* computes the output array[M(a_1, a_1), M(a_1, a_2), …, M(a_1, a_n)], where M(a_0, a_j) is the maximal value from a_0 to a_j. The prefix max scan can be divided into k steps when $n = 2^k$. For step i = 0 to k – 1, the element array[$j\times2^{i+1}$+1] will be compared with array[$j\times2^{i+1}$–2^i+1] for j = 1 to 2^{k-i-1} in parallel, and store the result in array[$j\times2^{i+1}$+1].

Figure 9 shows the operations performed by the prefix max scan for n = 8. The input is the array of $H'[i][1]+(n-1)*g$, $H'[i][2]+(n-2)*g$, ..., $H'[i][n]$. The output array will be the output array of $H'[i][1]+(n-1)*g$, M($H'[i][1]+(n-1)*g$, $H'[i][2]+(n-2)*g$), M($H'[i][1]+(n-1)*g$, $H'[i][n]$). After adding output array and the array of $-(n-1)*g$, $-(n-2)*g$, ... , $-(n-n)*g$ together, the result $L[i][j]$ of formula (*b*) is obtained. Let us now analyze the execution time again. Suppose that each step of the parallel prefix scan requires O(1) time. The computation time for each row is O($log_2 n$). The total computation time for the matrix is m×O($log_2 n$) = O($m\log_2 n$) which is a great

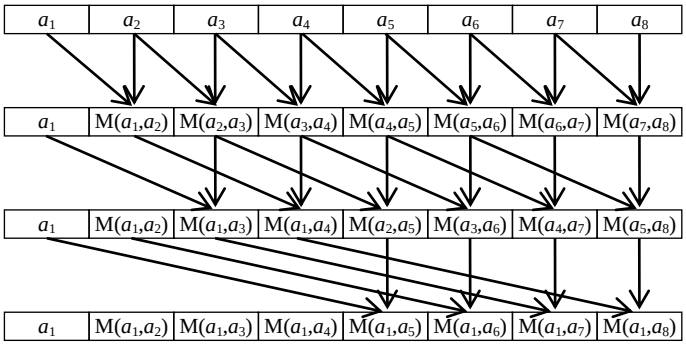

Figure 9. Prefix max scan.

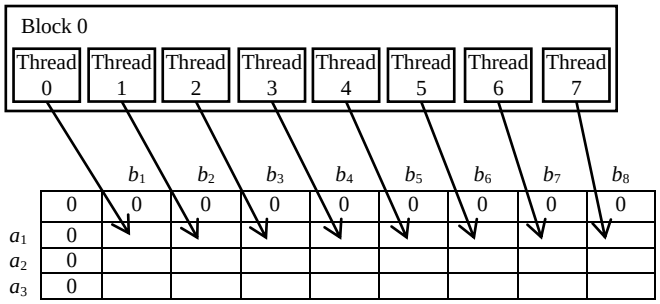

Figure 10. A thread is used to process each cell.

improvement.

B. CUDA implementation

In this section, we describe how to use CUDA to implement the proposed algorithms. As we mentioned before, one alignment task is mapped to one block. Each thread in the block is used to process each cell of one row of the matrix. Figure 10 shows our idea. However, our experiments are carried out on a NVIDIA C1060 graphics card. It only supports 512 threads per block. Thus, the width of the matrix cannot exceed 512. In order to solve this problem, we partition the matrix into slices of equal width when the width of the matrix exceeds 512. The slices are processed iteratively. Figure 11 shows how we partition the matrix into slices of width 512. On the other hand, we avoid storing the entire of matrix during the slice computation. We allocate the minimum amount of memory to compute the matrix. One is a linear amount of shared memory and the other is a linear amount of linear global memory. The shared memory is used to store the i^{th} row when the $(i+1)^{th}$ row is to be computed. The global memory is used to store the last column of the slice. The shared memory of only up to 512×4Bytes (integer type) = 2Kbytes is needed. On the other hand, the database sequences are stored in the global memory. The texture memory is exploited to store the substitution matrix (i.e. $sbt(x,y)$) and the query sequence.

V. Experimental results

The tests of proposed method is executed on NVIDIA C1060 graphics card, with 30 MPs comprising 240 SPs and 4GB RAM, installed in a PC with an Intel Core 2 6420 2.13 GHz processor running the Ubuntu OS. Table I and Table II

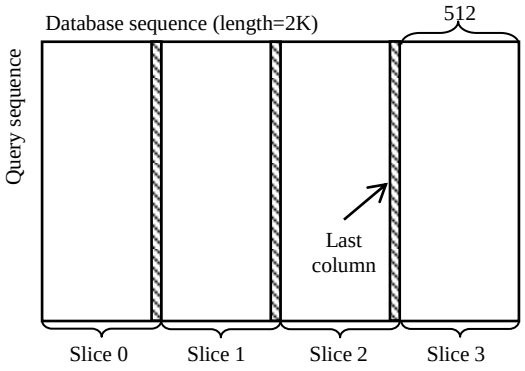

Figure 11. Partition the matrix into slices of equal width

Table I. PC specification

Item	Description
CPU	Intel Core 2 6420 2.13 GHz
RAM	DDR2 4GB
OS	Ubuntu 10.04.2 64bit
GPU	NVIDIA C1060
GPU Driver	270.41.19
PCI Express	2.0

show the detailed information of the graphics card and PC, respectively.

For database sequences used in this paper, five databases are produced by Seq-Gen [16]. Each database includes 100 sequences of equal length, ranging from 128 to 8192. Seq-Gen is a program that can simulate various lengths of DNA or protein sequences. On the other hand, the substitution matrix BLOSUM 62 is used with gap penalty −4.

In order to completely understand the performance of the proposed method, we compare two existing methods as follows:

(1) Single CPU-thread implementation of the Smith-Waterman algorithm. The runtime is measured by sequential executing alignment tasks.

(2) Two kernels of CUDASW++ proposed in [11]. The source codes of CUDASW++ are available at http://www.nvidia.com/object/swplusplus_on_tesla.html

As we mentioned before, CUDASW++ uses one of two kernel functions to find the optimal local alignment between a query sequence and a database sequence. The first and second kernels are the same as the second and first parallelism mechanisms, respectively. The length of the database sequence determines which kernel to be used. In order to compare with two kernels of CUDSASW++, we first increase the threshold to 10,000 so that all sequences in the database are aligned using the first kernel, and then decrease the threshold to 0 so that only the second kernel can be used. Table III shows the runtime comparison.

In the Table III, the length of the query sequence is fixed to 128; we increase the length of the database sequences so that more slices to be processed iteratively. Since we use the shared memory to calculate each slice, it is very useful to accelerate the computation, and there are no bank conflicts in the proposed method. For the same alignment tasks, our proposed method is 50 times faster than that of single CPU-thread, 4 times faster than first kernel of CUDASW++, and 2 times faster than second kernel of CUDASW++ in average. Figure

Table II. Graphics card specification

Item	Description
CUDA driver	3.2
CUDA capability	1.3
Global memory	4 GB
Clock rate	1.3 GHz
Multiprocessors	30
Stream processors per multiprocessor	8
CUDA cores	240
Shared memory per block	16 KB
Maximum number of threads per block	512
Constant memory	65 KB

Table III. Runtime comparison

DBseq length	# of DBseq	Runtime (*ms*)			
		CUDASW++ first kernel	CUDASW++ second kernel	Single CPU-thread	Our method
128	100	9.09	6.18	69.69	1.65
256	100	12.80	6.67	135.71	2.71
512	100	20.36	10.00	233.25	4.93
1k	100	34.14	16.86	482.53	9.66
2k	100	65.35	25.70	872.98	19.14
4k	100	126.00	44.80	2244.11	38.26
8k	100	246.88	85.70	4680.05	76.13

12 summarizes all the compared methods except the single CPU-thread implementation.

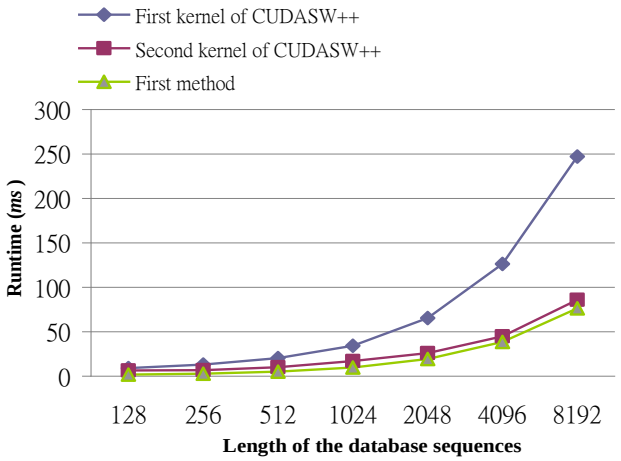

Figure 12. The runtimes of all the compared methods

VI. Conclusion

In recent years, the multi-core system has been widely used. The application can be efficiently calculated in parallel. Focusing on this point, NVIDIA introduces the CUDA technology to provide a user-friendly environment to write programs for GPU's. Due to the rapid growth in biological sequence databases, accelerating sequence alignment by using high-performance device has become a recent trend. In this paper, we have presented a fast sequence alignment scheme using CUDA-enabled GPU. We redefined the recursive formula so that one row of the matrix can be calculated in parallel. Next, we accelerated the computation of each cell of one row by prefix max scan method. In addition, only on-chip shared memory is used for reducing the penalty of memory accesses. Experimental results show the proposed method is in average 50 times faster than implementation of Smith-Waterman based on CPU and 2~4 times faster than other GPU-based version of Smith-Waterman algorithm. Finally, we expect the proposed method can provide researchers as a reference in the study of sequence alignment. Table VI shows the performance comparisons for the sequences of different lengths.

References

[1] S. B. Needleman and C. D. Wunsch, "A general method applicable to the search for similarities in the amino acid sequence of two proteins," *J. Molecular Biology*, vol. 48, pp. 443-453, 1970.

[2] T. F. Smith and M. S. Waterman, "Identification of common molecular subsequence," *J. Molecular Biology*, vol. 147, pp. 195-197, 1981.

[3] D. J. Lipman and W. R. Pearson, "Rapid and Sensitive Protein Similarity Searches," *Science*, vol. 227, pp. 1435-1441, 1985.

[4] S. F. Itschul, W. Gish, W. Miller, E. W. Myers, and D. J. Lipman, "Basic Local Alignment Search Tool," *J. Molecular Biology*, vol. 215, pp. 403-410, 1990.

[5] Oliver T, Schmidt B and Maskell DL, "Reconfigurable architectures for bio-sequence database scanning on FPGAs," *IEEE Trans. Circuit Syst. II* 2005, 52:851–855.

[6] Szalkowski A, Ledergerber C, Krahenbuhl P and Dessimoz C, "SWPS3–fast multi-threaded vectorized Smith-Waterman for IBM Cell/B.E. and x86/SSE2," *BMC Research Notes* 2008, I:107.

[7] Svetlin A Manavski and Giorgio Valle, "CUDA compatible GPU cards as efficient hardware accelerators for Smith-Waterman sequence alignment," *BMC Bioinformatics* 2008, 9(Suppl 2):S10.

[8] S. R. Sathe and D. D. Shrimankar, "Parallelization of DNA Sequence Alignment using Open MP," *ICCCS'*11, February 12–14, 2011.

[9] CUDA zone, http://developer.nvidia.com/category/zone/cuda-zone.

[10] Liu W, Schmidt B, Voss G and Muller-Wittig W, "Streaming algorithms for biological sequence alignment on GPUs," *IEEE Transactions on Parallel and Distributed Systems*, 18(9):1270–1281, 2007.

[11] Yongchao Liu, Bertil Schmidt, Douglas L Maskell, "CUDASW++2.0: enhanced Smith-Waterman protein database search on CUDA-enabled GPUs based on SIMT and virtualized SIMD abstractions," *BMC Research Notes* 2010, 3:93.

[12] David J. States, Warren Gish, and Stephen F. Altschul, "Improved Sensitivity of Nucleic Acid Database Searches Using Application-Specific Scoring Matrices," METHODS: A Companion to Methods in Enzymology Vol. 3, No. 1, August, pp. 66-70, 1991.

[13] Steven Henikoff and Jorja G. Henikoff, "Amino acid substitution matrices from protein blocks," Proc. Natl. Acad. Sci. Vol. 89, pp. 10915-10919, 1992.

[14] Rognes and Seeberg, "Six-fold speed-up of Smith-Waterman sequence database searches using parallel processing on common microprocessors," *BIOINF*: *Bioinformatics*, 16, 2000.

[15] Friman Sánchez, Esther Salamí, Alex Ramirez, Mateo Valero, "Parallel processing in biological sequence comparison using general purpose processors," IEEE International Symposium on Workload *Charact erization*(*IISWC*), 2005.

[16] Seq-Gen web site, http://tree.bio.ed.ac.uk/software/seqgen.