Proceedings of the World Congress on Engineering 2010 Vol I
WCE 2010, June 30 - July 2, 2010, London, U.K.

An Efficient Parallel Algorithm for Longest Common Subsequence Problem on GPUs

Jiaoyun Yang, Yun Xu*, Yi Shang* *

Abstract—**Sequence alignment is an important problem in computational biology and finding the longest common subsequence (LCS) of multiple biological sequences is an essential and effective technique in sequence alignment. A major computational approach for solving the LCS problem is dynamic programming. Several dynamic programming methods have been proposed to have reduced time and space complexity. As databases of biological sequences become larger, parallel algorithms become increasingly important to tackle large size problems. In the meantime, general-purpose computing on graphics processing units (GPGPU) has emerged as a promising technology for cost-effective high performance computing. In this paper, we develop an efficient parallel algorithm on GPUs for the LCS problem. We propose a new technique that changes the data dependency in the score table used by dynamic programming algorithms to enable higher degrees of parallelism. The algorithm takes advantage of the large number of processing units and the unique memory-accessing properties of GPUs to achieve high performance. The algorithm was implemented on Nvidia 9800GT GPUs and tested on randomly generated sequences of different lengths. The experiment results show that the new algorithm is about 6 times faster on GPUs than on typical CPUs and is 3 times faster than an existing efficient parallel algorithm, the diagonal parallel algorithm.**

Keywords: Parallel algorithms, GPUs, dynamic programming, bioinformatics

1 Introduction

Sequence alignment is a fundamental technique for biologists to investigate the similarity of different species, as high sequence similarity often implies molecular structural and functional similarity. In computational methods, biological sequences are represented as strings and finding the longest common subsequence (LCS) is a widely used method for sequence alignment. The LCS problem commonly refers to finding the longest common subsequence of two strings, whereas for three or more strings it is called multiple longest common subsequence problem (MLCS) [1][2][3].

Dynamic programming is a classical approach for solving LCS problem, in which a score matrix is filled through a scoring mechanism. The best score is the length of the LCS and the subsequence can be found by tracing back the table. Let m and n be the lengths of two strings to be compared. The time and space complex of dynamic programming is $O(mn)$. Many algorithms have been proposed to improve the time and space complex. In [4], Myers and Miller applied a divide-and-conquer technique [5] to solve the LCS problem and the space complexity of their algorithm is $O(m+n)$ while the time complex remains to be $O(mn)$. In [6], Masek and Paterson presented some new techniques to reduce the time complex to $O(n^2/logn)$.

Parallel algorithms have been developed to reduce execution time through parallelization in diagonal direction [7] and bit-parallel algorithms [8][9][10]. Bit-parallel algorithms depends on machine word size and are not good for general processors. Aluru, Futamura and Mehrotra [15] proposed a parallel algorithm using prefix computation for general sequence alignment problem with time complex $O(mn/p+\tau(m+p)\log p+\mu m\log p)$. Efficient parallel algorithms based on dominant point approaches has also been proposed for general MLCS problem [11][12][13][14]. However, these algorithms are usually slow for two-string LCS problems.

In recent years, graphics processing units (GPUs) have become a cost-effective means in parallel computing and have been widely used in bioinformation as hardware accelerators. Some traditional algorithms have been implemented on GPUs and achieved significant speedups [16][17].

In this paper, we present an efficient parallel algorithm on GPUs for the LCS problem. Based on the dynamic programming algorithm, we change the data dependence

*This work is supported in part by the Key Project of The National Nature Science Foundation of China, under the grants No. 60533020 and No. 60970085, and by the Key Subproject of The National High Technology Research and Development Program of China, under the grant No. 2009AA01A134.
J.Y. Yang, Y. Xu, and Y. Shang are with the Key Laboratory on High Performance Computing, Anhui Province, and the College of Computer Science and Technology, University of Science and Technology of China, Heifei, Anhui, China. Y. Shang is also with the Department of Computer Science, University of Missouri, Columbia, Missouri, USA.
E-mail: jiaoyun@mail.ustc.edu.cn,xuyun@ustc.edu.cn,and shangy@missouri.edu

ISBN: 978-988-17012-9-9
ISSN: 2078-0958 (Print); ISSN: 2078-0966 (Online)

in the dynamic programming table so that the cells in the same row or the same column of the dynamic table can be computed in parallel. The algorithm uses $O(max(m, n))$ processors and its time complex is $O(n)$. A nice property of the algorithm is that each GPU processor has balanced workload. The algorithm was implemented on Nvidia graphics cards and achieved a good speedup on test problems of random sequences.

This paper is organized as follows. In Section 2, classical dynamic programming algorithm for LCS and some related works are introduced. In Section 3, we present a new parallel algorithm on GPUs and performance analysis. In Section 4, we discuss implementation considerations on NVidia GPU platform CUDA and preliminary experimental results. Finally, in Section 5, we summary the paper.

2 Related work

The classical method for the LCS problem is the Smith-Waterman algorithm which is based on the dynamic programming principle. Given two input strings $a_1a_2 \ldots a_n$ and $b_1b_2 \ldots b_m$, the algorithm is as follows:

Construct a score matrix S of size $(n + 1) * (m + 1)$, in which $S[i, j]$ $(1 \leq i \leq n, 1 \leq j \leq m)$ records the length of the longest common subsequence for substrings $a_1a_2 \ldots a_i$ and $b_1b_2 \ldots b_j$. Fill the entries by the following recurrence formula:

$$S[i,j] = \begin{cases} 0 & i \text{ or } j \text{ is } 0 \\ S[i-1,j-1]+1 & a_i = b_j \\ max(S[i-1,j], S[i,j-1]) & \text{o.w.} \end{cases} \quad (1)$$

Fig. 1 shows an example of the LCS score table of strings TGCATA and ATCTGA.

		T	G	C	A	T	A
	0	0	0	0	0	0	0
A	0	0	0	0	1	1	1
T	0	1	1	1	1	2	2
C	0	1	1	2	2	2	2
T	0	1	1	2	2	3	3
G	0	1	2	2	2	3	3
A	0	1	2	2	3	3	4

Figure 1: The dynamic programming LCS score table of strings TGCATA and ATCTGA.

The main operation of the dynamic programming algorithm is filling the score table and its time complex is $O(m * n)$.

The diagonal parallel algorithm [7] is the best existing parallelization of dynamic programming for the LCS problem. It takes advantage of the independence of diagonal cells, i.e., the cells on the same diagonal in the score table doesn't depend on each other, so that they can be computed in parallel.

General Purpose computing on Graphics Processing Units (GPGPU) is a rapidly developing area in high performance computing. Although GPUs were originally designed for graphics computations, Due to rapid improvement of hardware performance and improved programmability, GPUs are becoming a cost-effective option for high performance desktop computing. Compared to CPUs, GPUs' performance increases two and a half times a year in recent years, much faster than the Moore's law for CPUs. A modern GPU usually contains large numbers of computing units. GPUs are usually used as a hardware accelerator or a co-processor, and more and more computational intensive tasks have been moved from CPUs to GPUs, such as for scientific computing [18], image processing [19] and bioinformatics [16][17]. Svetlin A. Manavski and Giorgio Valle also implemented the Smith-Waterman algorithm on the GPU [16]. GPUs are also appearing in personal computing devices. For example, the recently announced Apple iPad tablet computer has tightly integrated CPUs and GPUs to achieve very high performance while consuming very low power.

3 A New Parallel LCS algorithm

In this section, we first introduce some basic concepts of dynamic programming for LCS. Then we analyze data dependency in the score matrix constructed by dynamic programm. We present a new parallel algorithm based on rearranging the entries in the score matrix for greater parallelism. We analyze the time complex of the parallel algorithm and compare it with the diagonal parallel algorithm to show it advantages.

Let $A = a_1a_2 \ldots a_n$ and $B = b_1b_2 \ldots b_m$ be two strings over a finite alphabet Σ.

Definition 1 *$C = a_{j1}a_{j2} \ldots a_{jk}$ is a subsequence of A, if $\forall i$, $1 \leq i \leq k$:*

$$1 \leq j_i \leq n;$$

and for all s and t, $1 \leq s < t \leq k$:

$$j_s < j_t.$$

Definition 2 *$C = a_{j1}a_{j2} \ldots a_{jk}$ is a common subsequence of A and B, if C is a subsequence of A and a subsequence of B.*

ISBN: 978-988-17012-9-9
ISSN: 2078-0958 (Print); ISSN: 2078-0966 (Online)

Definition 3 *$C = a_{j1}a_{j2}\dots a_{jk}$ is the longest common subsequence of A and B, if*

1. *C is a common subsequence of A and B.*
2. *There is no common subsequence D, the length of D is larger than C.*

Definition 4 *In the dynamic table, if $a_i = b_j$, then the point $S[i,j]$ is a match point.*

The LCS problem is to find the longest common subsequence C of two sequences A and B.

3.1 Data Dependency Analysis

In the score matrix that dynamic program algorithm constructs according to Eq. (1), for all i and j ($1 \le i \le n, 1 \le j \le m$), $S[i,j]$ depends on three entries, $S[i-1,j-1]$, $S[i-1,j]$, and $S[i,j-1]$, as shown in Fig. 2. In another word, $S[i,j]$ depends on the data in the same row and the same column and thus the same row or same column data can't be computed in parallel.

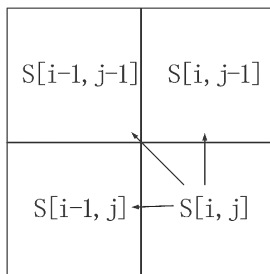

Figure 2: Data dependency in the score table of a dynamic programming algorithm for solving LCS problem.

An apparent way to parallelize the dynamic programming algorithm is to compute the score table in the diagonal direction. Its disadvantage is that the workload of different processors are different and unbalanced.

In order to compute the same row data or column data in parallel, the data dependence needs to be changed.

From the data dependence in the score table, the following observation can be made.

Observation 1 *For all s and t:*

If $1 \le i \le n$ and $1 \le s < t \le m$, then $S[i,s] \le S[i,t]$.

If $1 \le j \le m$ and $1 \le s < t \le n$, then $S[s,j] \le S[t,j]$.

Based on the observation, we modify Eq. (1) so that the same row data in the score table can be computed in parallel. Eq. (1) is divided into three parts:

1. If $i = 0$ or $j = 0$, then $S[i,j] = 0$. The data dependence is not changed.
2. If $a_i = b_j$, the $S[i,j] = S[i-1,j-1]+1$. The data dependence is not changed.
3. Otherwise, $S[i,j] = max(S[i-1,j], S[i,j-1])$. The data dependence is to be changed.

For condition (3), $S[i,j]$ can be made independent of the ith row data as $S[i,j-1]$ can be replaced by the following recurrence equation:

$$S[i,j-1] = \begin{cases} 0 & \text{if } i = 0 \text{ or } j-1 = 0 \\ S[i-1,j-2]+1 & \text{if } a_i = b_{j-1} \\ max(S[i-1,j-1], S[i,j-2]) & \text{o.w.} \end{cases} \quad (2)$$

Again, only in the third condition, $S[i,j-1]$ depends on the ith row data. Thus $S[i,j-2]$ can be re-formulated in a similar way, and so on. This process ends when $j-k = 0$ at the kth step, or $a_i = b_{j-k}$ at the kth step. Assume that the process stops at the kth step, and the k must be the minimum number that makes $a_i = b_{j-k}$ or $j-k = 0$. Then the recurrence equation Eq. (1) can be replaced by the following recurrence equation:

$$S[i,j] = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0 \\ S[i-1,j-1]+1 & \text{if } a_i = b_j \\ max(S[i-1,j], S[i-1,j-k-1]+1) & \text{if } a_i = b_{j-k} \\ max(S[i-1,j], 0) & \text{if } j-k = 0 \end{cases} \quad (3)$$

From Eq. (3), we have the following theorem.

Theorem 1 *In the score table of dynamic programming, the ith row data can be calculated just based on the $(i-1)th$ row data.*

Similar to Theorem 1, the same idea also works for columns if $S[i-1,j]$ is replaced in Eq. (1). For example, Fig. 3 shows the result of changing the data dependence in the third row of the score table for sequences ATTG and TAGC. The arrows represent data dependency.

From the re-formulation process, it can be shown that $S[i,j-k]$ is a match point if the process ends at $S[i,j-k]$. $S[i,j-k]$ is the first match point before $S[i,j]$ in the ith row. The first character a_i in sequence B can be found through preprocessing. A table P of size $l*(m+1)$ is constructed, where l is the size of the finite alphabet Σ and m is the length of sequence B. Let $C[1\dots l]$ be the finite alphabet and $P[i,j]$ represent the maximum number before j that makes $b_{p[i,j]} = C[i]$. $P[i,j]$ can be computed as follows:

$$P[i,j] = \begin{cases} 0 & \text{if } j = 0 \\ j-1 & \text{if } b_{j-1} = C[i] \\ P[i,j-1] & \text{o.w.} \end{cases} \quad (4)$$

ISBN: 978-988-17012-9-9
ISSN: 2078-0958 (Print); ISSN: 2078-0966 (Online)

		A	T	T	G
	0	0	0	0	0
T	0	0	1	1	1
A	0	1	1	1	1

Figure 3: An example of re-formulated data dependency in the score table of a dynamic programming algorithm for solving LCS problem.

As an example, Table 1 shows the table P for sequence ATTGCCA.

We can re-formulate Eq. (3) by replacing $j - k$ with $P[c, j]$:

$$S[i,j] = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0 \\ S[i-1, j-1] + 1 & \text{if } a_i = b_j \\ max(S[i-1,j], 0) & \text{if } P[c,j] = 0 \\ max(S[i-1,j], S[i-1, P[c,j]-1] + 1) & \\ & \text{o.w.} \end{cases} \quad (5)$$

where c is the number of character a_i in array C.

3.2 A New Technique for Efficient Parallelization

According to Theorem 1, the ith row data can be calculated by the $(i - 1)th$ row data. The parallel algorithm calculate the ith row data in parallel as follows:

1. For $i = 1$ to l par-do
   For $j = 1$ to m
   Calculate $P[i, j]$ according to Eq. (4)
   End for
   End for

2. For $i = 1$ to n
   For $j = 1$ to m par-do
   Calculate S[i,j] according to Eq. (5)
   End for
   End for

Assuming max(m, n) processors, in step 1, l is the size of the alphabet and is usually smaller than the length of the sequences. For example the alphabet size is 4 for genes and 20 for proteins. Step 1 takes $O(m)$ time and step 2 takes $O(n)$ time. Overall the algorithm's time complex is $O(max(m, n))$.

Existing diagonal parallel algorithms calculate the score table one diagonal after another. Although the data on the same diagonal can be computed in parallel, the number of data points on different diagonals is different, which goes from 1 to $max(m, n)$. Given $max(m, n)$ processors, some processors are idle when the number of data point on a diagonal is less than $max(m, n)$, which leads to reduced parallelism.

The calculation of $S[i, j]$ depends on branching conditions, which makes efficient parallelization difficult. We can re-write Eq. (4) as

$$P[i,j] = \begin{cases} 0 & \text{if } j = 0 \\ j & \text{if } b_{j-1} = C[i] \\ P[i, j-1] & \text{o.w.} \end{cases} \quad (6)$$

And re-write Eq. (5) as

$$S[i,j] = \begin{cases} 0 & \text{if } i \text{ or } j = 0 \\ max(S[i-1,j], 0) & \text{if } P[c,j] = 0 \\ max(S[i-1,j], S[i-1, P[c,j]-1] + 1) & \\ & \text{o.w.} \end{cases} \quad (7)$$

where c is the number of character a_i in array C.

Now the revised algorithm without branching is as follows:

1. For $i = 1$ to l par-do
   For $j = 1$ to m
   Calculate $P[i, j]$ according to Eq. (6)
   End for
   end for

2. For $i = 1$ to n
   For $j = 1$ to m par-do
   $t =$ the sign bit of $(0 - P[c, j])$.
   $s =$ the sign bit of $(0 - (S[i - 1, j] - t * S[i - 1, P[c, j] - 1]))$.
   $S[i, j] = S[i - 1, j] + t * (s \oplus 1)$.
   End for
   End for

4 Implementation and Experimental Results

General-purpose computing on GPUs (GPGPU) has great potentials for high performance computing. Nowadays a general GPU usually has a large number of computing units organized as a streaming architecture. In the GPU, every computing unit contains 8 scalar processor cores that execute the program concurrently. Due to GPU's high performance on scientific computing, NVidia and AMD have developed their new GPGPU platforms CUDA and CTM, respectively, in order to make general purpose GPU programming easier. We used the CUDA platform in our implementation and experiments.

CUDA uses threads to manage the computing units. The creation, management, and execution of the threads are handled by hardware and there is no software overhead

ISBN: 978-988-17012-9-9
ISSN: 2078-0958 (Print); ISSN: 2078-0966 (Online)

Table 1: The table P generated by preprocessing according to Eq. (4) for string ATTGCCA.

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

in managing threads. Threads are organized into block and blocks are then grouped into grid. CUDA uses SIMT (Single Instruction, Multiple Threads) technique and executes the same instruction in multiple threads. Every block uses computing units in turns to hide the delays of memory-access and synchronization. It is important to create enough threads to use the GPU resources efficiently. For the LCS problem, since the sequences to be compared with is usually very long, we change the data dependence so that every cell in the same row can be computed in parallel. Our parallel algorithm invokes a large numbers of threads and achieves improved parallelism on GPUs.

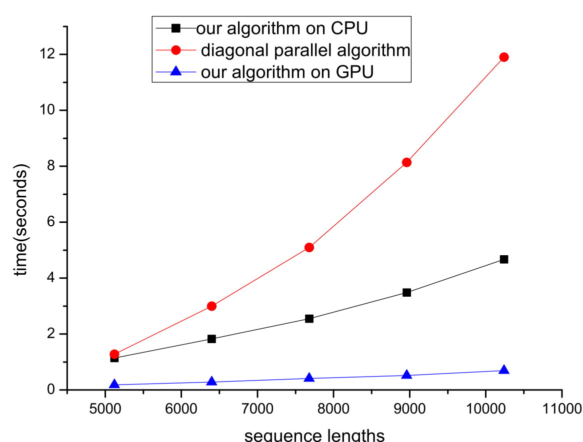

Figure 4: Performance comparison of our algorithm and a diagonal parallel algorithm.

In experiments, our parallel algorithm was implemented on Nvidia 9800GT graphics card running at 1.4 GHz on Windows. To compare with the execution times on CPUs and the time of previous parallel algorithm, we also implemented our algorithm and the diagonal parallel algorithm using OpenMP on a typical PC with Intel Dual-Core E2140 processor running at 1.6 GHz. We experimented with random gene sequences (alphabet size 4) of lengths from 5120 to 10240. Experiment results are shown in Fig. 4. The results show that on GPU our algorithm is 6 times faster than on CPU. On the same CPU platform, our algorithm is 3 times faster than the diagonal parallel algorithm.

5 Summary

In this paper, we present an efficient parallel algorithm for solving LCS problems on GPUs. By changing the data dependency in the score table used by dynamic programming, the algorithm enables higher degree of parallelism and achieves a good speedup on GPUs. The technique of data dependency re-ordering and re-formulation can be used in many other dynamic programming algorithms for their parallel implementation on GPUs.

References

[1] D.S. Hirschberg, "Algorithms for the longest common subsequence problem," *J. ACM*, V24, N4, pp. 664-675, 10/77

[2] W. J. Hsu, M. W. Du. "Computing a longest common subsequence for a set of strings," *BIT*, V24, N1, pp. 45-59, 3/84

[3] Y. Robert, M. Tchuente, "A Systolic Array for the Longest Common Subsequence Problem", *Inform. Process. Lett.*, V21, N4, pp. 191-198, 10/85

[4] E.W. Myers, W. Miller, "Optimal Alignments in Linear Space", *Computer Applications in the Biosciences*, V4, N1, pp. 11-17, 3/88

[5] D.S. Hirschberg, "A linear space algorithm for computing maximal common subsequences", *Communications of the ACM*, V18, N6, pp. 341-343, 6/75

[6] W. Masek, M. Paterson, "A faster algorithm for computing string edit distances", *J. of Computer and System Sciences*, V20, N1, pp. 18-31, 2/80

[7] N. Ukiyama, H. Imai, "Parallel multiple alignments and their implementation on CM5", *Genome Informatics*, Yokohama, Japan, pp. 103-108, 12/93

[8] L. Allison, T.L. Dix, "A bit-string longest common subsequence algorithm", *Inform. Process. Lett.*, V23, N6, pp. 305-310, 12/86

[9] M. Crochemore, C. S. Iliopoulos, Y. J. Pinzon, J. F. Reid, "A fast and practical bit vector algorithm for the longest common subsequence problem", *Inform. Process. Lett.*, V80, N5, pp. 12/01

ISBN: 978-988-17012-9-9
ISSN: 2078-0958 (Print); ISSN: 2078-0966 (Online)

[10] Hyyrö, "Bit-parallel LCS-length computation revisited", *In Proc. 15th Australasian Workshop on Combinatorial Algorithms AWOCA*, NSW, Australasion, pp. 16-27, 7/04

[11] W. Liu, L. Chen, "A Parallel Algorithm For Solving LCS Of Multiple Bioseqences", *In: Proceedings of the Fifth International Conference on Machine Learning and Cybernetics*, Dalian, China, pp. 4316-4321, 8/06

[12] D. Korkin, Q. Wang, Y. Shang, "An Efficient Parallel Algorithm for the Multiple Longest Common Subsequence (MLCS) Problem", *In: 37th International Conference on Parallel Processing*, Portland, America, pp. 354-363, 9/08

[13] Q. Wang, D. Korkin, Y. Shang, "Efficient Dominant Point Algorithms for the Multiple Longest Common Subsequence (MLCS) Problem", *Proceedings of the Twenty-First International Joint Conference on Artificial Intelligence*, Pasadena, America, pp. 1494-1499, 7/09

[14] K. Hakata, H. Imai, "Algorithms for the longest common subsequence problem for multiple strings based on geometric maxima", *Optimization Methods and Software*, V10, N2, pp. 233-260, 1998

[15] Y. Lin, C. Su, "Faster optimal parallel prefix circuits: new algorithmic construction", *Journal of Parallel and Distributed Computing*, V65, N12, pp. 1585-1595, 12/05

[16] S.A. Manavski, G. Valle, "CUDA Compatible GPU Cards as Efficient Hardware Accelerators for Smith-Waterman Sequence Alignment", *BMC Bioinformatics* V9(Suppl 2), S10, 2008

[17] W. Liu, B. Schmidt, G. Voss, "Streaming algorithms for biological sequence alignment on GPUs", *IEEE Trans. Parallel and Distributed Systems*, V18, N9, pp. 1270-1281, 9/07

[18] J. Kruger, R. Westermann, "Linear Algebra Operators for GPU Implementation of Numerical Algorithms", *ACM Trans. Graphics*, V22, N3, pp. 908-916, 7/03

[19] F. Xu, K. Mueller, "Ultra-Fast 3D Filtered Backprojection on Commodity Graphics Hardware", *2nd IEEE International Symposium on Biomedical Imaging: Macro to Nano*, Arlington, America, pp. 571-574, 4/04

ISBN: 978-988-17012-9-9
ISSN: 2078-0958 (Print); ISSN: 2078-0966 (Online)