ISSN 1330-3651 (Print), ISSN 1848-6339 (Online)
DOI: 10.17559/TV-20150314132647

A FAST PARALLEL ALGORITHM FOR FINDING THE LARGEST COMMON 4-CONNECTED COMPONENT FROM TWO MATRICES

Ying Gao, Haoshen Liu, Jiancong Huang, Jiajie Duan, Lei Mu

Original scientific paper
We describe a new design of parallel algorithm for solving the two-dimensional longest common substring (2D LCS) problem, taking advantage of the multi-core graphic processing unit architecture offered by Compute Unified Device Architecture (CUDA). In this article we also define the 2D LCS problem as finding the largest common 4-connected component from two input matrices and present an algorithm which can exactly solve this problem in *0* (mnst/P) time with a P-core GPU.

Keywords: *CUDA; largest common 4-connected component; parallel algorithm; 2DLCS*

Brzi paralelni algoritam za pronalaženje najveće zajedničke 4-spojene komponente iz dviju matrica

Izvorni znanstveni članak
Opisujemo novi dizajn paralelnog algoritma za rješavanje problema dvodimenzionalnog najduljeg zajedničkog podniza (2D LCS), iskoristivši arhitekturu grafičke obradne jedinice s više jezgri (multi-core graphic processing unit) ponuđene od Compute Unified Device Architecture (CUDA). U ovom radu također definiramo 2D LCS problem pronalaženjem najveće zajedničke 4-spojene komponente iz dvije ulazne matrice i predstavljamo algoritam koji može točno riješiti ovaj problem u *0* (mnst/P) vremenu s P-core GPU.

Ključne riječi: *2DLCS; CUDA; najveća zajednička 4-spojena komponenta; paralelni algoritam*

1 Introduction

Finding the longest string that is a substring common to two given strings is a classic problem in string analysis. This is the LCS [1] problem (different from the longest common subsequence [2] problem, which finds the longest subsequence common to two sequences). Many applications rely on the availability of efficient LCS routines as a basis for their own efficiency. Biological applications such as DNA microarrays [3] often need to compare the DNA sequences of two different genes. File comparison (UNIX diff command, CVS, file fragments predicting [4]) is fundamentally a LCS or longest common subsequence problem. The LCS routine is also a building block of string operations and string matching applications like spell checkers. It is therefore important to continue to explore efficient LCS techniques on emerging computing architecture.

Compute Unified Device Architecture (CUDA) [5] is a parallel computing platform and programming model introduced by NVIDIA. Since GPUs have been at the leading edge of chip-level parallelism for a long time, CUDA provides a general-purpose platform for programmers to exploit the massive parallel computational power of GPU. GPUs have a parallel throughput architecture taking a different approach with CPUs that emphasizes executing many concurrent threads, rather than executing a single thread quickly. Although increasing the CPU clock speed has always been a reliable source for improved performance, the novel GPU approach manages to get rid of the long sophisticated computing pipeline and makes a great breakthrough in multi-core architecture. Current NVIDIA GPU, for example GeForce GTX TITAN, contains up to 2688CUDA cores running at 0,837 GHz, and in contrast to the corresponding consumer product in CPUs which is the Intel® Core™ i7-3970X with 6 cores running at 3,5 GHz, GPUs have a tremendous advantage in parallelism, especially when targeting a data-parallelism problem [6].

In this paper, we present a detailed method with analysis of what we believe is the first algorithm which exactly solves the two-dimensional longest common substring (2D LCS) problem and also benefits from multi-core GPUs using CUDA. There has been an increasing motivation for the 2D LCS operation. For example, computer vision, pattern recognition and computational biology [7, 8] have begun their research in two-dimensional space.

2 Related work

For the sake of clarity, we first give the definition of original LCS problem.

Definition 2.1: Find the longest substring common to two given strings.

Input: Given string $A = a_1 a_2 \cdots a_m$ of length m and string $B = b_1 b_2 \cdots b_n$ of length n.

Output: The result string $C = c_1 c_2 \cdots c_w$ of length w is a common substring of $\boldsymbol{A}$ and $\boldsymbol{B}$ and w is maximized.

The classical method for the LCS problem based on the dynamic programming principle is as follows:

The idea is to find all the lengths of the longest common suffix (LCSuff) for all pairs of prefixes of the strings and store these lengths in a matrix. Construct a score matrix $\boldsymbol{T}$ of size $(m+1) \times (n+1)$, in which $\boldsymbol{T}[i,j](1 \le i \le m, 1 \le j \le n)$ records the length of the longest common suffix for two prefixes $A_{1\ldots i}$ and $B_{1\ldots j}$. The length is filled by the following recurrence equation:

$$LCSuff(A_{1\ldots i}, B_{1\ldots j}) = \begin{cases} LCSuff(A_{1\ldots i-1}, B_{1\ldots j-1}) + 1 & if\ A[i] = B[j] \\ 0 & otherwise \end{cases} \quad (1)$$

The maximum of these lengths is the length of the result string $\boldsymbol{C}$ (LCSubstr):

$$w = |C| = LCSubstr(A,B) = \max LCSuff(A_{1\ldots i}, B_{1\ldots j}), \quad (2)$$

where $1 \le i \le m, 1 \le j \le n$.

After we find out the value of w and point (p,q) corresponding to the particular point (i,j) where can reach its maximum, the result string $\boldsymbol{C}$ can be reconstructed by tracing the diagonal from the corresponding point to upper left in matrix $\boldsymbol{T}$.

$$C_{1\ldots|C|} = A_{p-|C|+1\ldots p} = B_{q-|C|+1\ldots q} \quad (3)$$

Because the main operation of this algorithm is filling the score matrix and each operation takes constant time, its time complexity is $\boldsymbol{O}(mn)$.

Example 1: For strings "EFEF" and "FEEFE", the longest common substring is "EFE". The Fig. 1(a) shows the matrix $\boldsymbol{T}$ of this example.

		E	F	E	F
	0	0	0	0	0
F	0	0	1	0	1
E	0	1	0	2	0
E	0	1	0	1	0
F	0	0	2	0	2
E	0	1	0	3	0

(a)

		E	F	E	F
	0	0	0	0	0
F	0	0	1	0	1
E	0	1	0	1	0
E	0	1	0	1	0
F	0	0	1	0	1
E	0	1	0	1	0

(b)

Figure1 The dynamic programming LCS score matrix $\boldsymbol{T}$ of input strings "EFEF" and "FEEFE"

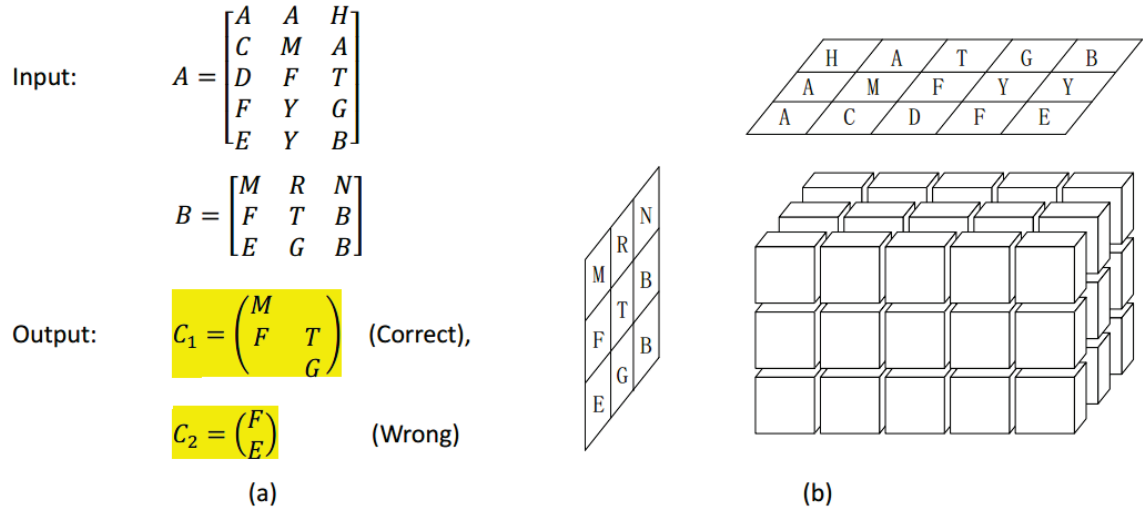

Figure 2 The 2D LCS Data Cube

3 2D LCS Problem

Extending the longest common substring problem to a two dimensional problem, the input should be two matrices which can be not only character matrices [8] but also number matrices (images). Unlike the original problem, the result of the 2D LCS problem does not need to have the same structure as its inputs, which is actually the largest 4-connected component [9] that is common to two input matrices. As we solve the 2D LCS problem by the dynamic programming principle (in contrast to 2D Suffix Tree [10]), the algorithm can be used to process images directly (number matrices). The formal definition of 2D LCS is given below.

Definition 3.1: Find the largest 4-connected component common to two given matrices.

Input: Give matrix $\boldsymbol{A}$ with m rows and n columns and matrix $\boldsymbol{B}$ with s rows and t columns.

Output: The result 4-connected component $\boldsymbol{C}$ of size w is a common component of $\boldsymbol{A}$ and $\boldsymbol{B}$ and w is maximized.

The definition of the 4-connected component is slightly different from its original meaning in binary valued digital imaging. The new definition is given below.

Definition 3.2: A set of matrix elements common to $\boldsymbol{A}$ and $\boldsymbol{B}$, $\boldsymbol{C}$, is a 4-connected component if for every pair of elements c_i and c_j in C, there exists a sequence of elements $c_i, \ldots, c_j$ such that:

(a) All elements in the sequence are in set C i.e. are matrix elements common to $\boldsymbol{A}$ and $\boldsymbol{B}$.

(b) Every 2 elements that are adjacent in the sequence are 4-neighbors.

4 Algorithm

4.1 Method

The main idea of our algorithm for solving 2D LCS problem is based on the dynamic programming principle. We extend the classic method mentioned in the previous section to 2D space and enhance its performance by parallelism. The main data structure of our algorithm is a

data cube instead of a matrix, in which the two input matrices stay outside the top and left faces of the data cube and the data cube itself stores the corresponding comparison result of the two matrices.

However, we cannot find the largest 4-connected component common to $\boldsymbol{A}$ and $\boldsymbol{B}$ by only analysing this data cube. If we view the data cube as a stack of matrices from back to front as in Fig. 2(b), then each matrix is the score matrix of the strings in the corresponding columns of $\boldsymbol{A}$ and $\boldsymbol{B}$. Thus we cannot compare strings from different columns of two matrices, and it would lead to only one optimal substructure of the original problem as C_2 in Fig. 2 because of missing one degree of freedom. Moreover, the data cube cannot deal with the situation that $\boldsymbol{A}$ and $\boldsymbol{B}$ have different number of columns.

Thus, the algorithm needs three basic steps to solve 2D LCS problem:

(i) Shift matrix $\boldsymbol{A}$ through $\boldsymbol{B}$ by columns and extract corresponding matrices $\boldsymbol{C}$, $\boldsymbol{D}$.
(ii) Compare matrix $\boldsymbol{C}$ and $\boldsymbol{D}$ and find the size of the largest 4-connected component common to $\boldsymbol{C}$, $\boldsymbol{D}$ in the exact same columns.
(iii) Repeat last two steps until $\boldsymbol{A}$ and $\boldsymbol{B}$ have no more cross columns, and the 4-connected component common that is the largest component in size among all of them is the result.

4.2 Step (1)

Before going to step (1), we need to ensure the number of columns in matrix $\boldsymbol{A}$ is larger than that in $\boldsymbol{B}$. If $A.cols < B.cols$, exchange matrix $\boldsymbol{A}$ with matrix $\boldsymbol{B}$.

We can now shift matrix $\boldsymbol{A}$ through $\boldsymbol{B}$ like Fig. 3. Assuming matrix $\boldsymbol{B}$ as the stable surface and matrix A as the moving surface that shifts one column upward for each iteration, there are three conditions depending on the relative positions of matrix $\boldsymbol{A}$ and $\boldsymbol{B}$ in 2D space as Fig. 4. Defining h as the iterator, it also represents the relative position of $\boldsymbol{A}$ and $\boldsymbol{B}$, which is approximately the distance between the last column of $\boldsymbol{A}$ and the first column of $\boldsymbol{B}$.

The three conditions are described below:

(a) If $0 < h \leq B.cols$ as Fig. 4(a),

$$C_{i,j} = A_{i,w+j} \quad (4)$$
$$where\ 0 < i \leq A.rows, 0 < j \leq h, w = A.cols - h$$
$$D_{i,j} = B_{i,j} \quad (5)$$
$$where\ 0 < i \leq B.rows, 0 < j \leq h$$

(b) If $B.cols < h \leq A.cols$ as Fig. 4(b),

$$C_{i,j} = A_{i,w+j} \quad (6)$$
$$where\ 0 < i \leq A.rows, 0 < j \leq B.cols, w = A.cols - h$$

The content of matrix $\boldsymbol{D}$ remains unchanged.

(c) If $A.cols < h < B.cols + A.cols$ as Fig. 4(c),

$$C_{i,j} = A_{i,j} \quad (7)$$
$$where\ 0 < i \leq A.rows, 0 < j \leq B.cols + A.cols - h$$
$$D_{i,j} = B_{i,w+j} \quad (8)$$
$$where\ 0 < i \leq B.rows, 0 < j \leq B.cols + A.cols - h, w = h - A.cols$$

Copying one element from one matrix to another takes constant time so that the time for extracting one matrix depends on its size. If we execute step (1) on CPU, at each iteration it takes at most $O(mt + st) = O(t \times \max(m, s))$ time and particularly $O(mt)$ if n is much bigger than t because in condition (b) we only need to extract matrix $\boldsymbol{C}$. Since iterator $h = O(n + t) = O(\max(n, t) = O(n)$ ($A.cols > B.cols$, thus $n > t$), the total running time for step (1) is $O\big(h(mt + st)\big) = O(nt \times \max(m, s))$ and $O(mnt)$ if either n is much bigger than t or s approaches $O(m)$.

We can also execute step (1) on GPU by CUDA programming architecture, and it is very trivial to parallelize step (1). We set up the thread number for running COPY (CUDA kernel for extracting one matrix) as the size of its target matrix, and each thread in the GPU copies one element where its thread ID is corresponding to one in the target matrix. Assuming the GPU has P cores, the total running time on GPU for step (1) is $O(mnt/P)$.

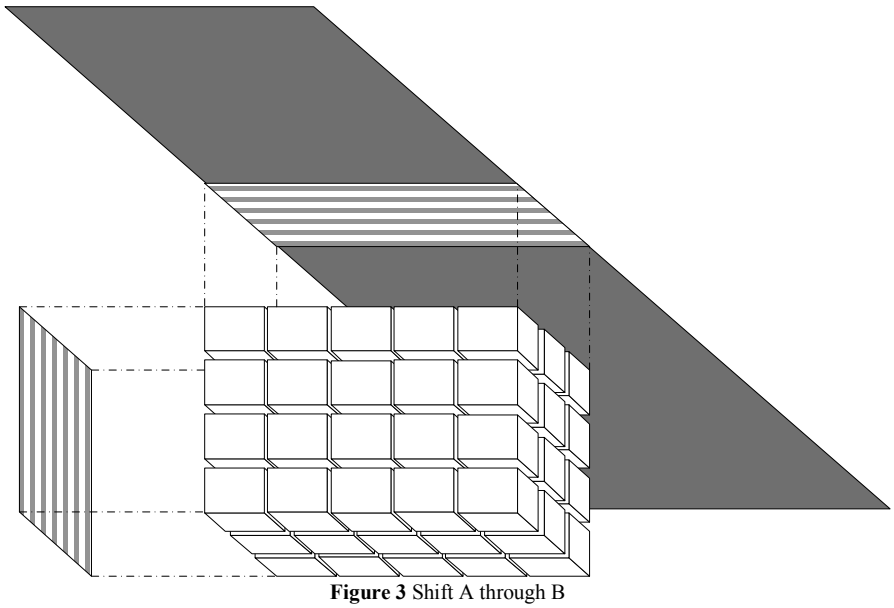

Figure 3 Shift A through B

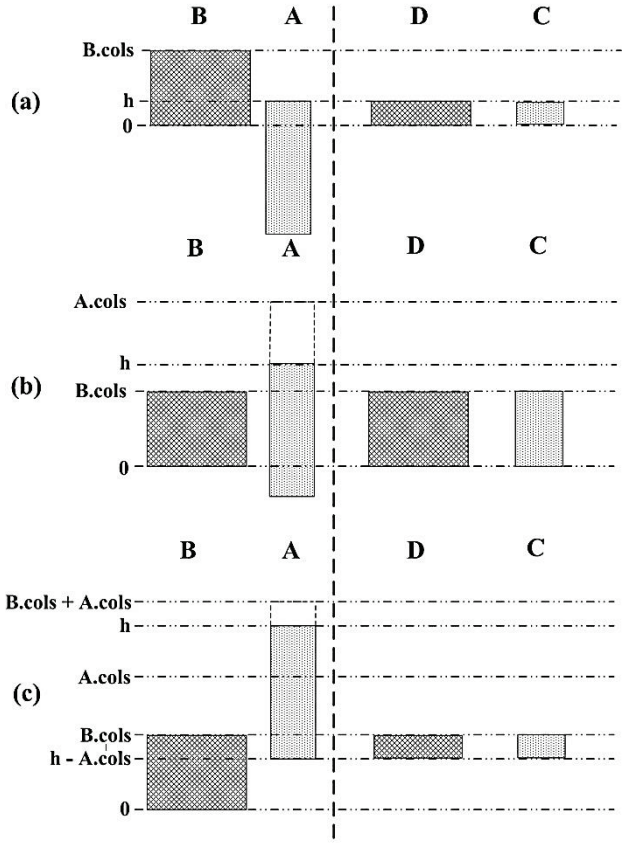

Figure 4 Three conditions of shifting

4.3 Step (2)

We can now extend the original algorithm for LCS problem to 2D space, but for the purpose of easy-to-parallelize, the algorithm would be decomposed into 2 parts.

First part. In this part, the algorithm would compare the temporary matrix $\boldsymbol{C}$ with $\boldsymbol{D}$ and fill the data cube with the comparison result. Define the data cube as $\boldsymbol{T}$ and the top-left-front corner of the data cube as the index origin which $+i$ axis points downwards, $+j$ axis points backwards and $+k$ axis points rightwards. The equation for filling the data cube is described below.

$$T_{i,j,k} = \begin{cases} 1 & if\ D_{i,j} = C_{k,j} \\ 0 & otherwise \end{cases} \quad (9)$$
$$where\ 0 < i \leq B.rows, 0 < j \leq B.cols, 0 < k \leq A.rows$$

The time complexity of this part is $O(mst)$ for each iteration. The total time complexity is $O(hmst) = O(mnst)$. We can also parallelize this part by CUDA. Set up the thread number for running COMPARE (CUDA kernel for the first part algorithm) as $s \times t$. Then each thread starts filling the cube one element a time from left face to right face ($0 < k \leq A.rows$) and manipulates only m elements. Each element in matrix $\boldsymbol{D}$ ejects one thread with a thread ID corresponding to the (i,j) position in $\boldsymbol{D}$. Assuming the GPU has P cores, the total running time on GPU for the first part algorithm is $O(mnst/P)$.

Second part. In the second part, the goal of the algorithm is to find out the largest 4-connected component common to matrix $\boldsymbol{C}$ and $\boldsymbol{D}$. In the original LCS problem, the mirror line as in Fig.1 which contains information about the common substring of two input matrices has become a mirror plane as in Fig. 5 after adding one more dimension to the LCS problem. After comparing and contrasting Eq. (9) and Eq. (1), the data cube has not been calculating the size of each common 4-connected component. Thus we have to finish the statistics on the size of each common 4-connected component in every mirror inclined plane inside the data cube.

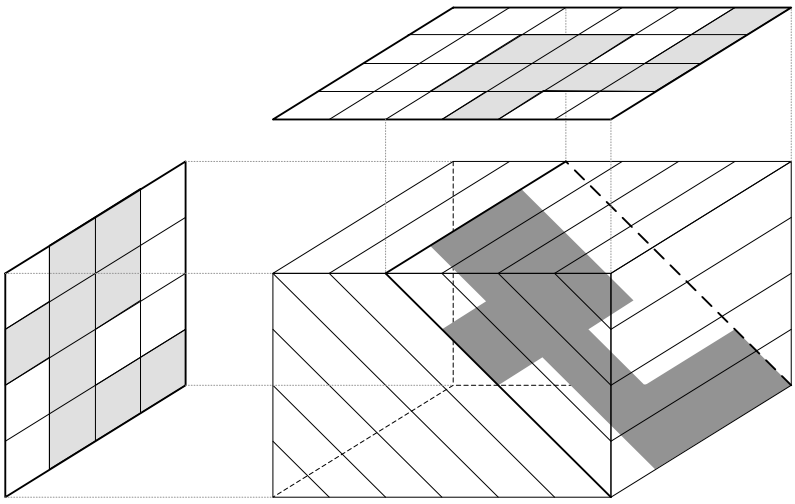

Figure 5 The mirror inclined plane inside the data cube

The statistics work is very similar to the process of connect component labelling. It searches the mirror plane line-to-line, top to bottom, to calculate the size of every connect component it met instead of assigning blob labels. The algorithm consists of two procedures: one is called SEARCH_PIECE which searches the plane pixel by pixel and marks it as "visited", but when it comes a pixel belonged to a connect component and the pixel has not been visited yet, it calls EAT_PIECE; another procedure is called EAT_PIECE which tries to figure out the size of the connect component that the pixel belongs in and marks every pixel in this connect component as "visited".

It also updates the temporary result for the plane that stores the size of connected component if the new size is bigger.

Thus this is a one-pass algorithm. Since it would not visit one pixel for more than two times, it costs $O(st)$ time on one plane and $O\big(st(m+s)\big) = O(st \times \max(m,s))$ on one data cube. For h iterations, it totally costs $O\big(hst(m+s)\big) = O\big(nst(m+s)\big) = O(nst \times \max(m,s))$ time.

4.4 EAT_PIECE

Define the index for the top-left pixel in the mirror plane as (u,v) and the value of the corresponding cell in data cube T as $T(u,v)$. The value of data cube consists of two bits: one stands for visited or unvisited; the other stands for comparison result.

(i) Set the component size S as '1'.
(ii) Enqueue one element (u,v) to a queue Q.
(iii) Dequeue an index (u',v') from Q.
(iv) Check the values of $T(u'+1,v')$, $T(u'-1,v')$, $T(u',v'+1)$, $T(u',v'-1)$. If anyone equals '1', enqueue the corresponding index into Q. And then S increases one.
(v) Set the corresponding bit of $T(u'+1,v')$, $T(u'-1,v')$, $T(u',v'+1)$, $T(u',v'-1)$ as "visited".
(vi) Go back to (3) if Q is not empty.

The process of the whole second part algorithm can also be run on GPU. Set up the thread number for running SEARCH_PIECE (CUDA version) as $(m+s)$. Then each thread searches one mirror plane in the data cube and all mirror planes in the cube can be searched concurrently. Assuming the GPU has P cores, the total running time on GPU for the second part algorithm is $O(nst \times \max(m,s)/P)$.

4.5 Notations and time analysis

At last, the result extraction is trivial. One way is to store the temporary largest component and its size among the components extracted by EAT_PIECE whenever each thread for the mirror plane comes across. Then compare the results among all threads at each iteration by the CUDA REDUCTION scheme [11]. Another method is similar to the first one, but only stores the index that is the trigger element of EAT_PIECE and size of the temporary largest component. After all iterations, we only need to run EAT_PIECE to extract the result component from the last remaining index which belongs to one element of the result largest 4-connected component.

The most time consuming part is the second part of step (2) of the whole 2D LCS algorithm. Thus the algorithm has a time requirement of $O(mnst)$ if s approaches $O(m)$. Assuming the GPU has P cores, the algorithm can run on GPU in $O(mnst/P)$ time. If $P \geq m+s$, the algorithm takes $O(nst)$ time.

5 Experiments

In the experiment, we contrast our algorithm with MatrixMatchMaker (MMMvII [12, 13]) algorithm which finds the largest common submatrices between pairs of phylogenetic distance matrices to detect protein coevolution. Although the MMMvII algorithm has different result structure that is more relaxed than the 2D LCS definition, it is still a good candidate for comparison. After this experiment, our algorithm will be more functional and believable. We believe that it is the first algorithm which can find the largest common 4-connected component from two matrices.

Because we only need to compare the time consumption of two algorithms that actually solve two different problems, the input test data is simplified as two square matrices which are generated randomly. We test ten sizes of square matrix which vary from 110 × 110 to 200 × 200. The MMMvII results are collected by summiting the input test data to the official website [14]. The test platform of our algorithm is NVIDA GeForce GTX 550 Ti with 192 CUDA cores. In Fig. 6, it is easy to figure out that our algorithm has an average 2,27 speedup in contrast to MMMVII even when it is solving a much more complex problem.

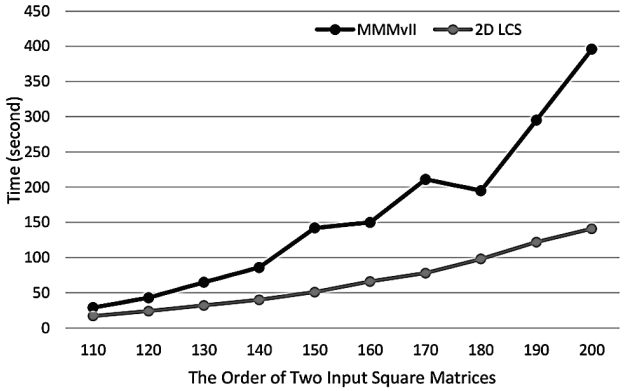

Figure 6 Experiment result of the MMMvII and 2D LCS algorithm

6 A kind of variation

The algorithm can still be improved the degree of parallelism, but the variation version requires more hardware resource. It needs P_1 GPUs and each GPU has P_2 cores. The main idea is to unfold the iteration of the algorithm by multithreading.

In step (1) of the algorithm, we create one thread at the beginning of each iteration and deploy the rest of algorithm to the new thread. This new thread would pick up one idle GPU to finish the rest of the algorithm. Thus this variation algorithm takes $O(mnst/P_1P_2)$ time. If $P_1 \geq$ h and $P_2 \geq m+s$, it takes $O(st)$ time which is also the theoretical limit for the 2D LCS problem.

7 Conclusions

We give the definition of the 2D LCS problem as finding the largest 4-connected component from two matrices and give the first algorithm which can achieve this task in $O(mnst/P)$ time. We also introduce a variation of this algorithm working at $O(mnst/P_1P_2)$ time. This algorithm can be used as a basic building block or parallel scheme of many biological and image problems and when the developer has a well-defined

problem size and enough hardware resource, the variation algorithm can work at $O(st)$ time.

Acknowledgements

This project is supported by the National Natural Science Foundation of China which is under grant No. 51204071, the Fundamental Research Funds for the Central Universities (2013ZZ0047), and the Science and Technology Planning Project of Guangdong Province (2012B010100019).

8 References

[1] Gusfield, D. Algorithms on Strings, Trees, and Sequences: Computer Science and Computational Biology, Cambridge University Press, New York, 1997. DOI: 10.1017/CBO9780511574931

[2] Cormen, T. H.; Leiserson, C. E.; Rivest R. L. Introduction to algorithms. // The MIT Press Cambridge, Massachusetts, London, England, 1989.

[3] Rahmann S. Fast large scale oligonucleotide selection using the longest common factor approach. // J Bioinform Comput Biol. 1, (2003), pp. 343-361. DOI: 10.1142/S0219720003000125

[4] Calhoun, W. C.; Coles, D. Predicting the types of file fragments. // Proceedings of the 2008 DFRWS Conference, Baltimore, MD. (Aug 2008). pp. 146-157. DOI: 10.1016/j.diin.2008.05.005

[5] Sanders, J.; Kandrot, E. CUDA by Example: An Introduction to General-Purpose GPU Programming. Addison-Wesley, (July 2010). ISBN 978-0-13-138768-3.

[6] N. Satish; M. Harris; M. Garland. Designing efficient sorting algorithms for multi-core GPUs. // InIPDPS. (2009), pp. 1-10.

[7] Branden, C.; Tooze, J. Introduction to protein structure. Vol. 2. New York: Garland, 1991.

[8] Maddison, W. P.; Maddison, D. R. Mesquite: a modular system for evolutionary analysis, 2001.

[9] Jung-Me, P.; Carl, G.L.; Hui-Chuan, C. Fast connected component labeling algorithm using a divide and conquer technique. // Conference on computers and their Applications. 2000.

[10] Kim Dong Kyue et al. A simple construction of two-dimensional suffix trees in linear time. // Combinatorial Pattern Matching. Springer Berlin Heidelberg, 2007.

[11] Harris, M. Optimizing parallel reduction in CUDA. // NVIDIA Developer Technology. 6, (2007).

[12] Rodionov, A. et al. A new, fast algorithm for detecting protein coevolution using maximum compatible cliques. // Algorithms Mol Biol. 6, 17(2011). DOI: 10.1186/1748-7188-6-17

[13] Tillier, E. R. M.; Charlebois, R. L. The human protein coevolution network. // Genome research. 19, 10(2009), pp. 1861-1871. DOI: 10.1101/gr.092452.109

[14] Tillier, E. R. M.; Charlebois, R. L. MatrixMatchMaker Web interface. http://www.uhnresearch.ca/labs/tillier/MMMWEBvII/MMMWEBvII.php

Authors' addresses

Ying Gao, Professor
School of Computer Science and Engineering,
South China University of Technology,
Waihuan Dong Road No. 382, Panyu District, Guangzhou, China
E-mail: gaoying@scut.edu.cn

Haoshen Liu, MSc
School of Computer Science and Engineering,
South China University of Technology,
Waihuan Dong Road No. 382, Panyu District, Guangzhou, China
E-mail: sklhs413@sina.com

Jiancong Huang, MSc
School of Computer Science and Engineering,
South China University of Technology,
Waihuan Dong Road No. 382, Panyu District, Guangzhou, China
E-mail: bbb8383@gmail.com

Jiajie Duan, MSc
YunNan Electric Power Test & Research Institute Group CO. Ltd, China

Lei Mu, Senior Engineer
Huanggang Dong Road, Tiaoqiao District, JiNan, Shangdong Province, China