IPSJ Transactions on Mathematical Modeling and Its Applications Vol.7 No.2 36–44 (Nov. 2014)

Regular Paper

A GPU Implementation of a Bit-parallel Algorithm for Computing the Longest Common Subsequence

Katsuya Kawanami[1,a)] Noriyuki Fujimoto[1,b)]

Received: February 3, 2014, Revised: March 24, 2014,
Accepted: April 18, 2014

Abstract: The longest common subsequence (LCS) for two given strings has various applications, such as for the comparison of deoxyribonucleic acid (DNA). In this paper, we propose a graphics processing unit (GPU) algorithm to accelerate Hirschberg's LCS algorithm improved with Crochemore et al.'s bit-parallel algorithm. Crochemore et al.'s algorithm includes bitwise logical operators, which can be computed easily in parallel because they have bitwise parallelism. However, Crochemore et al.'s algorithm also includes an operator with less parallelism, i.e., an arithmetic sum. In this paper, we focus on how to implement these operators efficiently in parallel and experimentally show the following results. First, the proposed GPU algorithm with a 2.67 GHz Intel Core i7 920 CPU and GeForce GTX 580 GPU performs a maximum of 12.81 times faster than the bit-parallel CPU algorithm using a single-core 2.67 GHz Intel Xeon X5550 CPU. Subsequently, the proposed GPU algorithm executes a maximum of 4.56 times faster than the bit-parallel CPU algorithm using a four-core 2.67 GHz Intel Xeon X5550 CPU. Furthermore, the proposed algorithm with GeForce 8800 GTX performs 10.9 to 18.1 times faster than Kloetzli et al.'s existing GPU algorithm with the same GPU.

Keywords: longest common subsequence (LCS), bit-parallel algorithm, GPGPU

1. Introduction

There are various metrics for the similarity between two strings, for example, the edit distance and the longest common subsequence (LCS) [6]. LCS can be applied to various problems, for example, comparison of deoxyribonucleic acid (DNA), in exact string matching, and spell checking.

When the lengths of two given strings are m and n, one of the LCSs can be computed by dynamic programming in $O(mn)$ time and $O(mn)$ space [7]. However, m and n can be huge for the comparison of DNA. For example, in Ref. [16], Webster et al. compared genomic sequences of length 5.1 MB from humans and chimpanzees. When m and n are 5.1 MB, algorithms with $O(mn)$ space require more than 26 TB of memory. Hence, $O(mn)$ space is not acceptable in such applications. An algorithm to compute one of the LCSs of two given strings with much less space complexity (and the same time complexity) was proposed by Hirschberg. The algorithm computes an LCS recursively while computing the length of the LCS (LLCS) between various substrings of the two given strings. Hirschberg's algorithm requires $O(mn)$ time and $O(m+n)$ space. A method to compute the LLCS faster with bit-parallelism is well-known. This method requires $O(\lceil m/w \rceil n)$ time and $O(m+n)$ space [2], where w is the word size of a computer. Using this method, Hirschberg's LCS algorithm can be accelerated. However, much faster algorithms are desirable for strings of length of more than one million characters, which are common in the field of the comparison of DNA. Therefore, we consider accelerating the bit-parallel algorithm with a graphics processing unit (GPU). The bit-parallel algorithm includes bitwise logical operations and arithmetic sums. Bitwise logical operations are suitable for GPUs because they have bitwise parallelism. However, arithmetic sums have less parallelism. Therefore, we devise a method to compute them efficiently in parallel.

To the best of our knowledge, with the exception of Refs. [3], [4], [9], [11], [12], and [17], there are no existing studies for solving the LCS problem and/or the related problems using a GPU. However, none of the studies mentioned, with the exception of Ref. [9], address the LCS problem within $O(m+n)$ space: in Ref. [3], Deorowicz solved the LLCS problem only, and not the LCS problem; in Refs. [4] and [17], Dhraief et al. and Yang et al. solved the LCS problem, but their method required $O(mn)$ space; in Refs. [11] and [12], Ozsoy et al. solved the multiple LCS (MLCS) problem (the one-to-many LCS matching problem), which cannot be used to lead to an efficient algorithm to solve the LCS problem [17]; and in Ref. [9], Kloetzli et al. proposed an LCS algorithm for a GPU with $O(m+n)$ space. Thus, in this paper, we compare our proposed algorithm only with Kloetzli et al.'s algorithm.

Kloetzli et al.'s algorithm is a GPU implementation of Chowdhury et al.'s algorithm [1] for CPUs. Kloetzli et al.'s algorithm is based on dynamic programming without bit-parallelism. Their algorithm divides an LCS problem into four subproblems. The division is performed recursively until the size of a subproblem becomes sufficiently small to be executed on a GPU. In the algorithm, one thread block on a GPU executes one subproblem

[1] Osaka Prefecture University, Sakai, Osaka 599–8531, Japan
[a)] mu301005@edu.osakafu-u.ac.jp
[b)] fujimoto@mi.s.osakafu-u.ac.jp

and one thread on a GPU executes 4×4 cells of the table of dynamic programming.

Based on the method proposed in this paper, we implement a bit-parallel LCS algorithm on CUDA [5], [8], [10], [13], and conduct several experiments. In the experiments, our proposed GPU algorithm with 2.67 GHz Intel Core i7 920 CPU and NVIDIA GeForce GTX 580 GPU executes a maximum of 12.81 times faster than our bit-parallel CPU algorithm with a single-core 2.67 GHz Intel Xeon X5550 CPU and a maximum of 4.56 times faster than our bit-parallel CPU algorithm with a four-core 2.67 GHz Intel Xeon X5550 CPU. Another experiment shows that our algorithm is 10.9 to 18.1 times faster than Kloetzli et al.'s GPU algorithm for the same GPU (GeForce 8800 GTX).

The remainder of this paper is organized as follows. In Section 2, we briefly review the definition of LCS and the existing algorithms for a CPU. In Section 3, we present our proposed algorithm. In Section 4, we conduct several experiments to compare our GPU algorithm with our bit-parallel CPU algorithms and the existing GPU algorithm. In Section 5, we provide some concluding remarks and propose future studies. However, owing to limited space, we illustrate neither the architecture nor the programming of GPUs. For readers unfamiliar with these, we recommend the book [8] and the studies [5], [10], and [13].

2. LCS

2.1 The Definition of the LCS

Let C and A be strings $c_1c_2 \cdots c_p$ and $a_1a_2 \cdots a_m$ respectively. In the following, we assume without loss of generality that characters in the same string are different from each other. If there exists a mapping from the indices of C to the indices of A subject to the following conditions, C1 and C2, then C is called a subsequence of A.

C1: $F(i) = k$ if and only if $c_i = a_k$.

C2: If $i < j$, then $F(i) < F(j)$.

However, we define the null string, which is a string of length zero, as a subsequence of any string. We define a string that is a subsequence of both string A and string B as a common subsequence between A and B. The LCS between A and B is the longest of all the common subsequences between A and B. The LCS is not always unique. For example, the LCS between "abcdefghij" and "cfilorux" is "cfi." LCSs between "abcde" and "baexd" are "ad," "ae," "bd," and "be."

2.2 How to Compute the Length of the LCS

The LLCS can be computed using dynamic programming. This algorithm stores the LLCS between A and B in L[m][n] if we fill table L with $(m + 1) \times (n + 1)$ cells based on the following rules, R1 to R3, where m is the length of A and n is the length of B. To fill table L, this algorithm requires $O(mn)$ time and $O(mn)$ space.

R1: If $i = 0$ or $j = 0$, then L[i][j] = 0.

R2: If A[$i - 1$] = B[$j - 1$], then L[i][j] = L[$i - 1$][$j - 1$] + 1.

R3: Otherwise, L[i][j] = max(L[i][$j - 1$], L[$i - 1$][j]).

The rules R2 and R3 imply that the ith row ($1 \le i \le m$) of L can be computed only with the ith and $(i - 1)$th rows. This property leads us to an algorithm that requires less memory, shown in

Listing 1 Hirschberg's LLCS algorithm.

```
1  Input :string A of length m, string B of length n
2  Output:LLCS L[j] of A and B[0..j-1]
3           for all j(0<=j<=n)
4  llcs(A,m,B,n,L){
5    for(j=0 to n)
6      K[1][j] = 0
7    for(i=1 to m) {
8      for(j=0 to n)  K[0][j] = K[1][j]
9      for(j=1 to n) {
10       if(A[i-1] == B[j-1])  K[1][j] = K[0][j-1]+1
11       else  K[1][j] = max(K[1][j-1], K[0][j])
12     }
13   }
14   for(j=0 to n)
15     L[j] = K[1][j]
16 }
```

String B of length 10

String A of length 7

		E	A	B	E	D	C	B	A	A	C
	0	0	0	0	0	0	0	0	0	0	0
B	0	0	0	1	1	1	1	1	1	1	1
C	0	0	0	1	1	1	2	2	2	2	2
A	0	0	1	1	1	1	2	2	3	3	3
E	0	1	1	1	2	2	2	2	3	3	3
D	0	1	1	1	2	3	3	3	3	3	3
A	0	1	2	2	2	3	3	3	4	4	4
C	0	1	2	2	2	3	4	4	4	4	5

Fig. 1 An example of Hirschberg's LLCS algorithm.

Listing 1 [7]. K is a temporary array of size $2 \times (n + 1)$ cells. L is an array for storing output of size $1 \times (n + 1)$ cells. The tenth and eleventh lines in Listing 1 correspond to rules R2 and R3.

Hirschberg's LLCS algorithm shown in Listing 1 stores the LLCS between string A and string B[$0..j - 1$] (the substring of B from the first character to the jth character. When $j = 0$, we regard string B[$0..j - 1$] as the null string) in L[j]. This implementation reduces the required space to $O(m + n)$ with the same time complexity $O(mn)$. In **Fig. 1**, we show an example of Hirschberg's LLCS algorithm when A is "BCAEDAC" and B is "EABEDCBAAC." The result shows that the LLCS between A and B is five.

2.3 Hirschberg's LCS Algorithm

Listing 2 shows the LCS algorithm proposed by Hirschberg [7] where S[$u..l$] ($u \ge l$) represents the reverse of the substring S[$l..u$] of a string S. In the 15th and 16th lines, this algorithm invokes Hirschberg's LLCS algorithm shown in Listing 1. In the 19th and 20th lines, this algorithm invokes itself recursively. The algorithm computes an LCS while computing the LLCS. The algorithm requires $O(mn)$ time and $O(m + n)$ space. The dominant part of the algorithm is Hirschberg's LLCS algorithm llcs().

2.4 Computing the LLCS with Bit-parallelism

There exists an efficient LLCS algorithm with bit-parallelism. The algorithm shown in **Listing 3** is Crochemore et al.'s bit-parallel LLCS algorithm [2], where V is a variable that stores a bit-vector of length m. The notation & represents bitwise AND, | represents bitwise OR, ~ represents the bitwise complement, and

Listing 2 Hirschberg's LCS algorithm.

```
1  Input:string A of length m, string B of length n
2  Output:LCS C of A and B
3  lcs(A,m,B,n,C) {
4    if(n==0)  C = ""(null string)
5    else  if(m==1) {
6      for(j=1 to n)
7        if(A[0]==B[j-1])  {
8          C = A[0]
9          return
10       }
11     C = ""
12   }
13   else {
14     i = m/2
15     llcs(A[0..i-1],i,B,n,L1)
16     llcs(A[m-1..i],m-i,B[n-1..0],n,L2)
17     M = max{j : 0<=j<=n , L1[j]+L2[n-j]}
18     k = min{j : 0<=j<=n , L1[j]+L2[n-j] == M}
19     lcs(A[0..i-1],i,B[0..k-1],k,C1)
20     lcs(A[i..m-1],m-i,B[k..n-1],n-k,C2)
21     C = strcat(C1,C2)
22   }
23 }
```

Listing 3 Crochemore et al.'s bit-parallel LLCS algorithm.

```
1  Input :string A of length m, string B of length n
2  Output:LLCS L[i] of A[0..i-1] and B
3          for all i(0<=i<=m)
4  llcs_bp(A,m,B,n,L){
5    for(c=0 to 255) {
6      for(i=0 to m-1)
7        if(c==A[m-i-1]) PM[c][i] = 1
8        else PM[c][i] = 0
9    }
10   for(i=0 to m-1)
11     V[i] = 1
12   for(j=1 to n)
13     V = (V + (V & PM[B[j]])) | (V & ~(PM[B[j-1]]))
14   L[0] = 0
15   for(i=1 to m)
16     L[i] = L[i-1]+(1-V[i-1])
17 }
```

+ represents the arithmetic sum. Note that, + regards V[0] as the least significant bit. First, Crochemore et al.'s algorithm constructs a pattern match vector (PMV). PMV P of string S with respect to c is the bit-vector of length m that satisfies following conditions, C1 and C2.

C1: If S[i] = c, then P[i] = 1.

C2: Otherwise, P[i] = 0.

For example, the PMV of string "abbacbaacbac" with respect to "a" is 100100110010. In the fifth to ninth lines of Listing 3, the PMV of string A with respect to each character c is constructed. Since we assume one byte character, $0 \leq c \leq 255$. The reverse of the PMV of string A with respect to c is stored in PM[c], where a variable PM is a two-dimensional bit-array of size $256 \times m$.

This algorithm represents the table of dynamic programming as a sequence of bit-vectors such that each bit-vector corresponds to a column of the table. The ith bit of each bit-vector represents the difference between the ith cell and the $(i-1)$th cell of the corresponding column. Repeating bitwise operations, the algorithm performs the computation, which is equal to computing table L from left to right.

The last column of table L output by this algorithm is a bit-vector. However, we can convert it easily into an ordinary array of integers in $O(m)$ time. The converting process is in the 14th to 16th lines in Listing 3. Crochemore et al.'s bit-parallel algorithm requires $O(\lceil m/w \rceil n)$ time and $O(m+n)$ space where w is the word size of a computer.

3. The Proposed Algorithm

3.1 The CPU Algorithm Implemented on a GPU

As mentioned in Section 2.3, the dominant part of the LCS algorithm shown in Listing 2 is the function llcs(), which computes the LLCS. In this paper, we aim to accelerate the LCS algorithm shown in Listing 2, improved with the bit-parallel LLCS algorithm shown in Listing 3 (in other words, we replace the invocation of llcs() in Listing 2 with llcs_bp() in Listing 3). The algorithm requires $O(\lceil m/w \rceil n + m + n)$ time and $O(m+n)$ space. For this purpose, we propose a method to accelerate the LCS algorithm with a GPU. Despite using 64-bit mode on a GPU, the length of every integer register is still 32 bits. Thus, the word size w is 32.

In the 16th line of Listing 2, llcs() computes the LLCS between the reverse of string A and the reverse of string B. However, if we reverse A and B in every invocation of llcs(), the overhead of reversing becomes significant. Thus, we construct a new function llcs'(), which traverses strings from tail to head. The function llcs'() is the same as llcs(), except the order that the string is traversed. We also construct the bit-parallel algorithm llcs_bp'() corresponding to llcs'().

The output of Hirschberg's LLCS algorithm shown in Listing 1 is the mth row of table L. However, Crochemore et al.'s algorithm shown in Listing 3 represents a column of the table as a bit-vector and computes the table from the zeroth column to the nth column. Hence, Crochemore et al.'s algorithm outputs the nth column. Therefore, we change the original row-wise LLCS algorithm shown in Listing 1 into a column-wise algorithm. In addition, we have to change the original LCS algorithm shown in Listing 2 into another form corresponding to the column-wise LLCS algorithm.

Since our algorithm embeds 32 characters into one variable of an unsigned integer, we have to pad string A and ensure that its length is a multiple of 32. For this padding, we can use characters not included in either string A or B (for example, control characters).

3.2 Outline of the Proposed Algorithm

The algorithm shown in Listing 2 has recursive calls of lcs() in the 19th and 20th lines. However, GPUs support recursive calls only within some levels. Hence, we executes only llcs() in the 15th and 16th lines of Listing 2 on a GPU. Other parts of Listing 2 are executed on a CPU.

The LLCS algorithm shown in Listing 3 includes bitwise logical operators (&, |, ~) and arithmetic sums (+) on bit-vectors of length m. Bitwise logical operators are parallelized easily. However, an arithmetic sum has carries. Since carries propagate from the least significant bit to the most significant bit in the worst case, an arithmetic sum has less parallelism. Thus, we have to devise a method in order to extract higher parallelism from the computation of an arithmetic sum.

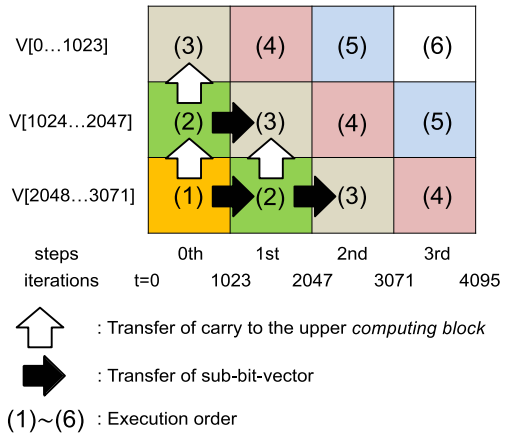

Fig. 2 Block-step partition in the case of $m = 3{,}072$ and $n = 4{,}096$.

We propose processing the bit-vectors of length m in parallel by dividing them into sub-bit-vectors. On a GPU, each bit-vector is represented as an array of unsigned integers of length $\lceil m/32 \rceil$ where the word size is 32. The size of one variable of an unsigned integer on a GPU is 32 bits. In CUDA architecture, 32 threads in the same warp are synchronized at the instruction level (single instruction, multiple data (SIMD) execution). Thus, we set the number of threads in one thread block at 32 so that threads in one thread block can be synchronized with no cost. Since one thread processes one unsigned integer (32 bits), one thread block processes 1,024 bits of the bit-vector of length m.

During one invocation of the kernel function, our algorithm performs only 1,024 iterations of n iterations. We refer to a group of 1,024 iterations as one *step*. For example, the jth step represents 1,024 iterations from $(1{,}024 \times j)$ to $(1{,}024 \times (j+1) - 1)$. We set the number of bits processed in one thread block and the number of iterations in one invocation to the same value so that each thread can transfer carries by reading from or writing to only one variable.

Figure 2 is an example of a block-step partition in the case of $m = 3{,}072$ and $n = 4{,}096$. Each rectangle in the figure represents one step of one block. We refer to it as a *computing block*. No computing block can be executed until its left and lower computing blocks have been executed. Therefore, only the lowest leftmost computing block can be executed in the first invocation of the kernel function. The computing block is the zeroth step of the block covering sub-bit-vector V[2,048..3,071]. In the second invocation of the kernel function, both the first step of the block covering V[2,048..3,071] and the zeroth step of the block covering V[1,024..2,047] can be executed. In each invocation of the kernel function, we execute all computing blocks we are able to execute at that time. Subsequently, computing blocks with the same number in Fig. 2 can be executed in parallel at the same time (the numbers represent the execution order). Based on the above ideas, we invoke the kernel function $(\lceil m/1{,}024 \rceil + \lceil n/1{,}024 \rceil - 1)$ times to obtain the LLCS (for example, we have to invoke the kernel function six times in Fig. 2).

The black arrows in Fig. 2 indicate that the block covering sub-bit-vector V[$i..i+1{,}023$] determines the value of V[$i..i+1{,}023$] to the $(j+1)$th step of itself at the end of the jth step. On the other hand, the white arrows indicate that the block covering sub-bit-vector V[$i..i+1{,}023$] determines carries in each iteration to the jth step of the block covering V[$i-1{,}024..i-1$]. Transfers of values from a computing block to another computing block, shown as black or white arrows in Fig. 2, are performed out of the loop processing the bitwise operations. When we store carries during the loop, we write carries in an array on the shared memory. After the loop, carries on the shared memory are copied into the global memory. Reading occurs in a similar manner. Before the loop, carries on the global memory are loaded into the shared memory. During the loop, we use carries on the shared memory, not on the global memory because the cost of transfer between registers and the global memory on a GPU is larger than the cost of transfer between registers and the shared memory on a GPU.

3.3 The Kernel Function

This section describes the kernel function llcs_kernel() that performs one step and the host function llcs_gpu(), which invokes llcs_kernel(). **Listing 4** is a pseudo code of llcs_kernel() and llcs_gpu(), where llcs_gpu() is a GPU implementation of llcs_bp() as shown in Listing 3. In addition to these functions, we construct llcs_kernel'() and llcs_gpu'(), which are GPU implementations of llcs_bp'(); however, they are considerably similar to llcs_kernel() and llcs_gpu(). Thus, we do not explain them.

First, we explain the kernel function llcs_kernel(). Argument m and argument n represent the length of string A and string B, respectively; dstr2 represents the copy of string B on the global memory. g_V is an array to store the bit-vector V. g_PMV is a two-dimensional array to store the PMV of string A with respect to each character c $(0 \le c \le 255)$. g_PMV[c] is the PMV of string A with respect to c. g_Carry is an array to store carries. When we use g_Carry, we regard it as a two-dimensional array and perform double buffering. Argument num represents the number of invocations of llcs_kernel(). num is used to compute which step the block should process in llcs_kernel(). The for-loop in the ninth to thirteenth lines represents the process of one step. The seventh and fourteenth lines are transfers of values, shown as black arrows in Fig. 2. The eighth and fifteenth lines are transfers of values, shown as white arrows in Fig. 2.

Next, we explain the function llcs_gpu(). llcs_gpu() invokes the kernel function llcs_kernel() $(num\_x + num\_y - 1)$ times in the for-loop of the 29th and 30th lines. The 22nd to 28th lines are pre-processing. The string A is padded in the 22nd line. The number num_x of blocks and the number num_y of steps are computed in the 24th and 25th lines. In the 26th and 27th lines, all bits of the bit-vector V are initialized to one. The 31st and 32nd lines are post-processing, where the bit-vector V is converted into an ordinary array and written in the output array finalOutput.

3.4 Parallelization of an Arithmetic Sum

As we state in Section 3.2, + has less parallelism because it has carries. To parallelize +, we applied Sklansky's method to parallelize the full adder, known as *conditional-sum addition* [14]. Sklansky's method uses the fact that every carry is either zero or one. To compute the addition of n-bit-numbers, each half adder computes a sum and a carry to the upper bit in both cases in advance. Then, carries are propagated in parallel. See Ref. [15] for

Listing 4 A pseudo code of llcs_kernel() and llcs_gpu().

```
__global__ void llcs_kernel(
  int m, n, char *dstr2, num,
  unsigned int *g_V, *g_PMV, *g_Carry) {
  index = global thread ID;
  count = step number of this block;
  cursor = 1024 * count;
  V = g_V[index];
  Load carries from g_Carry on global memory;
  for (j=0 to 1023) {
    if(cursor+j >= n) return;
    PMV = g_PMV[dstr2[cursor+j]][index];
    V = (V & (~PMV)) | (V + (V & PMV));
  }
  g_V[index] = V;
  Save carries to g_Carry on global memory;
}

void llcs_gpu(
  char *A, *B, *dstr1, *dstr2,
  int m, n, *finalOutput,
  unsigned int *g_V, *g_Carry, *g_PMV) {
  dstr1 = padded copy of A;
  dstr2 = B;
  num_x = (m+1023)/1024;
  num_y = (n+1023)/1024;
  for (i=0 to ((m+31)/32)-1)
    g_V[i] = 0xFFFFFFFF;
  Construct pattern match vectors;
  for(i=1 to num_x+num_y-1)
    llcs_kernel() in Parallel on a GPU(gridDim=num_x, blockDim=32);
  for(i=0 to m)
    finalOutput[i] = the amount of zeros from zeroth bit to ith bit in g_V;
}
```

details. In our algorithm, we use 32-bit width half adders rather than one-bit width half adders.

Using the example in **Fig. 3**, we explain the method to parallelize the n-bit full adder. Figure 3 shows a 32-bit addition performed by four full adders of 8-bit width. Note that Fig. 3 is illustrative and our actual implementation uses 1,024-bit full adders realized by 32 full adders of 32-bit width. In Fig. 3, we compute the sum of U, V, and *l_carry* (l_carry represents a carry from the lower sub-bit-vector. Obviously, l_carry is zero or one). In addition, we also compute a carry to the upper sub-bit-vector. To compute U+V+l_carry, first, we compute sums and carries for every byte. $S_0(0)$ represents sums and carries for every byte when a carry from the lower byte is zero. $S_0(1)$ represents them when a carry from the lower byte is one. Next, we consider computing sums and carries for every two bytes (we denote them as $S_1(0)$ and $S_1(1)$) from $S_0(0)$ and $S_0(1)$. To compute $S_1(0)$ and $S_1(1)$, we focus on a carry of the fourth byte of $S_0(1)$. Since the carry is zero, the third byte of $S_1(1)$ must be "0D" ("0D" is the third byte of $S_0(0)$). Therefore, in the case, we copy the third byte of $S_0(0)$ into the third byte of $S_0(1)$. In the same manner, we focus on carries of the fourth byte of $S_0(0)$, the second byte of $S_0(0)$, and the second byte of $S_0(1)$. When a carry of the $(2 \times k)$th byte of $S_0(0)$ is one ($k \in \mathbb{N}$), we copy the $(2 \times k - 1)$th byte of $S_0(1)$ into $S_0(0)$. When a carry of the $(2 \times k)$th byte of $S_0(1)$ is zero, we copy the $(2 \times k - 1)$th byte of $S_0(0)$ into $S_0(1)$. As a result, we can determine $S_1(0)$ and $S_1(1)$. Similarly, we determine the sums and carries for every four bytes from $S_1(0)$ and $S_1(1)$. The results are $S_2(0)$ and $S_2(1)$. $S_2(0)$ is U+V (the sums and carry when a carry from the lower sub-bit-vector is zero). $S_2(1)$ is U+V+1 (the sums and carry when a carry from the lower sub-bit-vector is one). The most important advantage of this method is the ability to execute the computation of $S_t(0)$ and $S_t(1)$ from $S_{t-1}(0)$ and $S_{t-1}(1)$

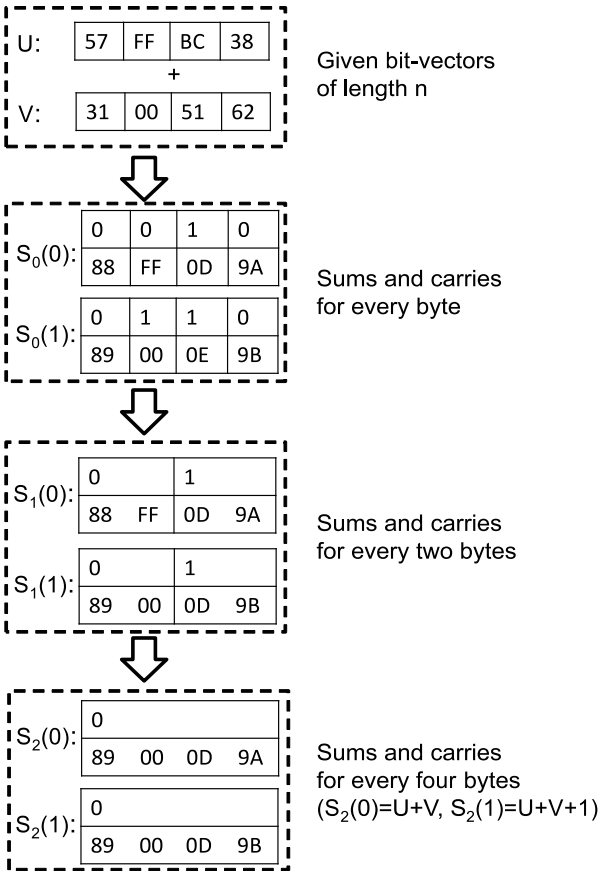

Fig. 3 Parallelization of an n-bit full adder (in the case of $n = 32$).

with 2^t-bytewise parallelism. When the number of elements in U and V is l, we repeat this process ($log_2 l$) times to determine U+V+l_carry.

The implementation is based on the above methods. The inputs are two arrays of unsigned integers, which store sub-bit-vectors of length 1,024, and a carry from the lower sub-bit-vector. The outputs are two arrays of unsigned integers and a carry to the upper sub-bit-vector. The number of elements in one array is 32. In addition, we construct a bool array to store carries to the upper element on the shared memory. First, we compute sums and carries for every 32 bits from two arrays of unsigned integers. When a sum is smaller than two operands, we set a carry to the upper element at one. Next, we obtain sums and carries for every 64 bits from the neighboring two sums and carries for every 32 bits. This process can be performed with one comparison and two substitutions. We use the same method to determine sums and carries for every 128 bits, 256 bits, 512 bits, and finally 1,024 bits. "A carry to the upper element" of the most significant element is "a carry to the upper sub-bit-vector." Thus, we store the carry in the global memory.

3.5 Other Notes

cudaMemcpy() between the host and device and cudaMalloc() of each array are performed before the recursive calls in Listing 2 because the overhead is extremely heavy when cudaMemcpy() and cudaMalloc() are performed in the recursive calls.

When we pad the string A on the device, we need the original A on the host. However, host-to-device transfer is much slower than device-to-device transfer or host-to-host transfer. To reduce the number of host-to-device transfers, we perform host-to-device transfer only once to reproduce a copy of the original A on the global memory. In llcs_gpu() or llcs_gpu'(), while using a string, it is copied into the working memories on the device and is padded on the device.

If the lengths of given strings are shorter than some constant value, the cost of host-to-device transfers becomes larger than the cost to compute the LLCS on a CPU. In such a case, the execution speed becomes slower when we use a GPU. Therefore, we check the lengths of strings before invoking llcs_gpu(). If the sum of the length of string A and B is at least 2,048, we compute the LLCS on a GPU. Otherwise, we compute the LLCS on a CPU. If the length of A is less than 96 or the length of B is less than 256, we compute the LLCS with dynamic programming on a CPU. Otherwise, we compute the LLCS with bit-parallel algorithm on a CPU.

4. Experiments

In this section, we compare the execution times of the proposed algorithm on a GPU with the execution times of our bit-parallel CPU algorithm and Kloetzli et al.'s GPU algorithm. We also break down execution time into three categories: CPU computation, GPU computation, and data transfer between a CPU and a GPU. We execute our GPU program on a 2.67 GHz Intel Core i7 920 CPU, an NVIDIA GeForce GTX 580 GPU, and Windows 7 Professional 64-bit operating system. We compile our GPU program with CUDA 5.0 and Visual Studio 2008 Professional. We execute the CPU programs on a 2.67 GHz Intel Xeon X5550 CPU and Linux 2.6.27.29 (Fedora10 x86_64) operating system. We compile the CPU programs with an Intel C++ compiler 14.0.1 without SSE instructions.

4.1 A Comparison with the Existing CPU Algorithms

We show the results of comparison between the proposed algorithm on a GPU and our bit-parallel algorithm on a CPU in **Fig. 4**, **Fig. 5**, **Fig. 6**, and **Fig. 7**. We execute the bit-parallel algorithm on a CPU with a single core and four cores. To create a multi-core version of the bit-parallel algorithm, we use task parallelism in OpenMP. In Listing 2, we execute the 15th, 16th, 19th, and 20th lines with task parallelism in OpenMP. In OpenMP 3.0 or later, we can use task parallelism with the notation *omp task*. Then, the function llcs() in 15th and 16th lines and lcs() in 19th and 20th lines are executed respectively in parallel. Thus, only two cores are used to execute llcs() in the first invocation of lcs(). In the second or later invocation of lcs(), all of four cores are used.

In Figs. 4 to 7, the green lines show the execution times on a GPU. The red and blue lines show the execution times on a CPU with a single core and four cores, respectively. All of the execution times on the graphs are measured in seconds and shown on the primary y-axis. The orange and purple lines show the speedup ratio of the proposed algorithm on a GPU to the single-core and multi-core version on a CPU, respectively. The speedup ratio is shown on the secondary y-axis. In the four graphs, the lengths of string A are five million, ten million, 15 million, and 20 million, respectively. In the four graphs, the length of string B shown on the x-axis increases from 100 thousand to ten million.

First, we compare the proposed algorithm on a GPU with the bit-parallel algorithm on a CPU with a single core. Figures 4 to 7 show that the proposed algorithm is a maximum of 11.15, 12.81, 12.51, and 12.35 times faster than the bit-parallel algorithm with a single core, respectively. The results show that the speedup ratio is more than ten when lengths of given strings are sufficiently long. In addition, the speedup ratio is more than one when the length of string B is more than 100 thousand in the four graphs.

Next, we compare the proposed algorithm on a GPU with the bit-parallel algorithm on a CPU with four cores. Figures 4 to 7 show that the proposed algorithm is a maximum of 4.02, 4.14, 4.56, and 4.09 times faster than the bit-parallel algorithm with four cores, respectively. The results show that the proposed algorithm on a GPU is a maximum of 4.56 times faster than the multi-core version on a CPU. In addition, the speedup ratio is more than one when the length of string B is more than 700 thousand in the four graphs. This fact implies that the proposed algorithm on a GPU is faster than the single-core version and the multi-core version when lengths of the given strings are sufficiently long.

4.2 A Comparison with the Existing GPU Algorithm

We compared the proposed algorithm on a GPU with Kloetzli et al.'s GPU algorithm. Kloetzli et al. used an AMD Athlon 64 CPU and GeForce 8800 GTX GPU. In order to compare the algorithms over the same GPU, we used a GeForce 8800 GTX GPU. In the experiment, we used a 2.93 GHz Intel Core i3 530 CPU, which is faster than Kloetzli et al.'s CPU. Thus, our environment

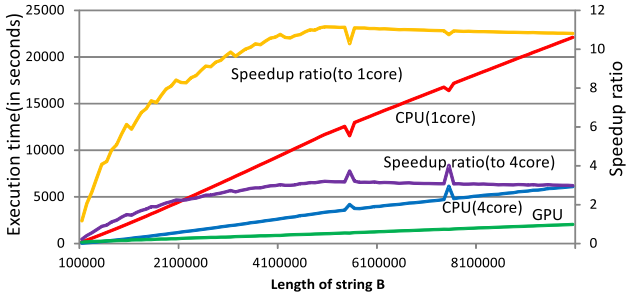

Fig. 4 Execution times for string A of length five million.

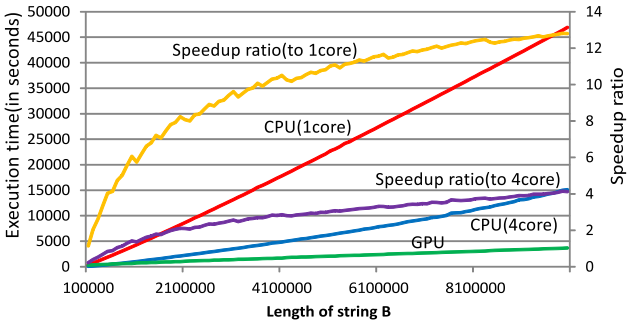

Fig. 5 Execution times for string A of length ten million.

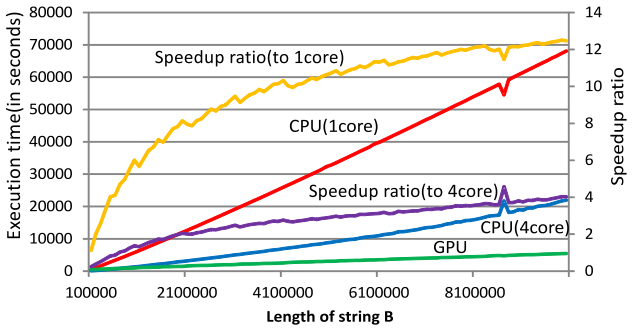

Fig. 6 Execution times for string A of length 15 million.

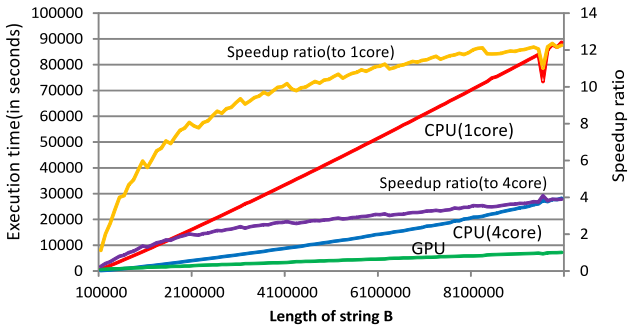

Fig. 7 Execution times for string A of length 20 million.

is not completely similar to that of Kloetzli et al.'s. However, as the CPU offers little contribution to the speed of the algorithms, we can expect the result is not highly affected.

Figure 8 shows the result. The x-axis shows the length of strings A and B measured in millions. The y-axis shows the execution times measured in minutes. In Fig. 8, the blue bars represent the execution times of Kloetzli et al.'s algorithm on a GPU. The red bars represent the execution times of our proposed algorithm on a GPU.

In the shortest case (0.27 million and 1.80 million), the speedup ratio is 12.0. In the longest case (1.51 million and 1.80 million),

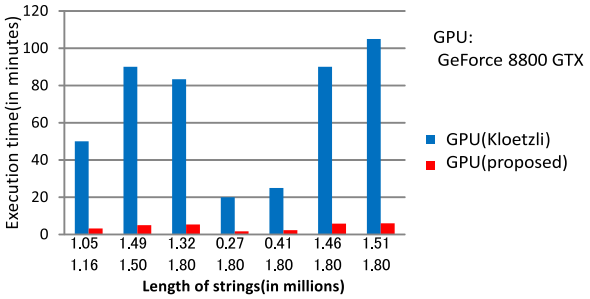

Fig. 8 A comparison with Kloetzli et al.'s GPU algorithm.

the speedup ratio is 17.6. The speedup ratio ranges from 10.9 (0.41 million and 1.80 million) to 18.1 (1.49 million and 1.50 million).

4.3 Breakdown of the Execution Time of the Proposed Algorithm

Table 1 shows breakdown of the execution time of the proposed algorithm for two given strings A and B in the case that the length of string A increases from five million to 20 million by five million and the length of string B increases from one million to ten million by one million. The lengths of given strings are shown in the two leftmost columns in Table 1. Table 1 includes both the execution time and its ratio to the total execution time. The column "GPU" represents the execution time of the GPU kernel functions. The column "transfer" represents the data transfer time between the CPU and the GPU. The column "CPU" represents the execution time of the other part. All of the execution times are measured in seconds.

In Table 1, the execution time of GPU computation accounts for 95.67% to 98.81% of the total execution time. The execution time of CPU computation accounts for 1.05% to 3.63% of the total execution time. The execution time of the data transfer accounts for 0.14% to 0.70% of the total execution time. The execution time of GPU computation shows linear growth in the lengths of given strings. On the other hand, the execution time of CPU computation shows logarithmic growth.

These results imply that GPU computation occupies most of the total execution time. In addition, the longer the lengths of given strings are, the larger the ratio of GPU computation is. Since the proposed algorithm executes GPU computation and CPU computation in serial, parallelizing GPU computation and CPU computation shortens the execution time. However, Table 1 implies that the effect is small because the execution time of CPU computation is at most 3.63%. Even if GPU computation and CPU computation are embarrassingly parallelized, the total execution time is only minorly reduced to 96.37% of the current version.

5. Conclusions

In this paper, we have presented a method to implement the bit-parallel LCS algorithm on a GPU and have conducted several experiments using our program based on the method. As a result, the proposed algorithm performs a maximum of 12.81 times faster than the single-core version of the bit-parallel algorithm on a CPU and a maximum of 4.56 times faster than the multi-

Table 1 Breakdown of the execution time of the proposed algorithm.

Length of string A	Length of string B	Time (Total)	Time (CPU)	Time (GPU)	Time (transfer)	Ratio (CPU)	Ratio (GPU)	Ratio (transfer)
5,000,000	1,000,000	335.10	12.15	320.60	2.35	3.63%	95.67%	0.70%
5,000,000	2,000,000	515.95	17.99	495.51	2.45	3.49%	96.04%	0.47%
5,000,000	3,000,000	697.56	20.25	674.77	2.54	2.91%	96.73%	0.36%
5,000,000	4,000,000	866.04	23.99	839.47	2.58	2.77%	96.93%	0.30%
5,000,000	5,000,000	1,050.03	25.38	1,021.86	2.79	2.41%	97.32%	0.27%
5,000,000	6,000,000	1,251.85	27.64	1,220.62	3.59	2.21%	97.50%	0.29%
5,000,000	7,000,000	1,453.22	29.77	1,419.09	4.36	2.05%	97.65%	0.30%
5,000,000	8,000,000	1,652.43	32.33	1,615.45	4.65	1.96%	97.76%	0.28%
5,000,000	9,000,000	1,850.64	35.38	1,810.26	5.00	1.91%	97.82%	0.27%
5,000,000	10,000,000	2,046.29	38.22	2,003.62	4.45	1.87%	97.91%	0.22%
10,000,000	1,000,000	647.21	13.68	629.01	4.52	2.11%	97.19%	0.70%
10,000,000	2,000,000	998.79	24.33	969.70	4.76	2.43%	97.09%	0.48%
10,000,000	3,000,000	1,354.90	28.19	1,321.85	4.86	2.08%	97.56%	0.36%
10,000,000	4,000,000	1,680.16	35.79	1,639.50	4.87	2.13%	97.58%	0.29%
10,000,000	5,000,000	2,047.44	38.50	2,003.96	4.98	1.88%	97.88%	0.24%
10,000,000	6,000,000	2,341.78	40.64	2,296.13	5.01	1.74%	98.05%	0.21%
10,000,000	7,000,000	2,679.54	46.62	2,627.82	5.10	1.74%	98.07%	0.19%
10,000,000	8,000,000	2,991.43	48.18	2,938.10	5.15	1.61%	98.22%	0.17%
10,000,000	9,000,000	3,357.05	49.75	3,301.87	5.43	1.48%	98.36%	0.16%
10,000,000	10,000,000	3,667.39	51.54	3,610.33	5.52	1.41%	98.44%	0.15%
15,000,000	1,000,000	954.44	16.23	932.39	5.82	1.70%	97.69%	0.61%
15,000,000	2,000,000	1,472.40	27.64	1,438.78	5.98	1.87%	97.72%	0.41%
15,000,000	3,000,000	2,002.67	32.97	1,963.42	6.28	1.65%	98.04%	0.31%
15,000,000	4,000,000	2,485.72	43.44	2,435.76	6.52	1.75%	97.99%	0.26%
15,000,000	5,000,000	3,039.24	45.93	2,986.43	6.88	1.51%	98.26%	0.23%
15,000,000	6,000,000	3,472.54	48.35	3,417.00	7.19	1.39%	98.40%	0.21%
15,000,000	7,000,000	3,980.38	57.51	3,915.33	7.54	1.44%	98.37%	0.19%
15,000,000	8,000,000	4,441.27	58.67	4,374.90	7.70	1.32%	98.51%	0.17%
15,000,000	9,000,000	5,005.08	60.41	4,936.03	8.64	1.21%	98.62%	0.17%
15,000,000	10,000,000	5,462.73	61.82	5,391.97	8.94	1.14%	98.70%	0.16%
20,000,000	1,000,000	1,262.07	18.57	1,234.71	8.79	1.47%	97.83%	0.70%
20,000,000	2,000,000	1,945.03	27.31	1,908.62	9.10	1.40%	98.13%	0.47%
20,000,000	3,000,000	2,650.20	34.55	2,606.29	9.36	1.31%	98.34%	0.35%
20,000,000	4,000,000	3,290.98	48.55	3,232.87	9.56	1.48%	98.23%	0.29%
20,000,000	5,000,000	4,027.68	52.45	3,965.57	9.66	1.30%	98.46%	0.24%
20,000,000	6,000,000	4,599.46	56.33	4,533.42	9.71	1.23%	98.56%	0.21%
20,000,000	7,000,000	5,274.06	69.39	5,194.96	9.71	1.32%	98.50%	0.18%
20,000,000	8,000,000	5,881.29	71.58	5,800.11	9.60	1.22%	98.62%	0.16%
20,000,000	9,000,000	6,637.26	74.17	6,553.24	9.85	1.12%	98.73%	0.15%
20,000,000	10,000,000	7,239.77	76.49	7,153.42	9.86	1.05%	98.81%	0.14%

core version of the bit-parallel algorithm on a CPU. In addition, the proposed algorithm executes 10.9 to 18.1 times faster than Kloetzli et al.'s algorithm on a GPU. Future studies will include optimization to the newest Kepler architecture and measuring execution times on a Kepler GPU.

References

[1] Chowdhury, R.A. and Ramachandran, V.: Cache-Oblivious Dynamic Programming, *The Annual ACM-SIAM Symposium on Discrete Algorithm* (*SODA*), pp.591–600 (2006).

[2] Crochemore, M., Iliopoulos, C.S., Pinzon, Y.J. and Reid, J.F.: A Fast and Practical Bit-Vector Algorithm for the Longest Common Subsequence Problem, *Information Processing Letters*, Vol.80, No.6, pp.279–285 (2001).

[3] Deorowicz, S.: Solving Longest Common Subsequence and Related Problems on Graphical Processing Units, *Software: Practice and Experience*, Vol.40, pp.673–700 (2010).

[4] Dhraief, A., Issaoui, R. and Belghith, A.: Parallel Computing the Longest Common Subsequence (LCS) on GPUs: Efficiency and Language Suitability, *The 1st International Conference on Advanced Communications and Computation* (*INFOCOMP*) (2011).

[5] Garland, M. and Kirk, D.B.: Understanding Throughput-Oriented Architectures, *Comm. ACM*, Vol.53, No.11, pp.58–66 (2010).

[6] Gusfield, D.: *Algorithms on Strings, Trees and Sequences: Computer Science and Computational Biology*, Cambridge University Press (1997).

[7] Hirschberg, D.S.: A Linear Space Algorithm for Computing Maximal Common Subsequences, *Comm. ACM*, Vol.18, No.6, pp.341–343 (1975).

[8] Kirk, D.B. and Hwu, W.W.: *Programming Massively Parallel Processors: A Hands-on Approach*, Morgan Kaufmann (2010).

[9] Kloetzli, J., Strege, B., Decker, J. and Olano, M.: Parallel Longest Common Subsequence Using Graphics Hardware, *The 8th Eurographics Symposium on Parallel Graphics and Visualization* (*EGPGV*) (2008).

[10] Lindholm, E., Nickolls, J., Oberman, S. and Montrym, J.: NVIDIA Tesla: A Unified Graphics and Computing Architecture, *IEEE Micro*, Vol.28, No.2, pp.39–55 (2008).

[11] Ozsoy, A., Chauhan, A. and Swany, M.: Towards Tera-Scale Performance for Longest Common Subsequence Using Graphics Processor, *IEEE Supercomputing* (*SC*) (2013).

[12] Ozsoy, A., Chauhan, A. and Swany, M.: Achieving TeraCUPS on Longest Common Subsequence Problem Using GPGPUs, *IEEE International Conference on Parallel and Distributed Systems* (*ICPADS*), pp.69–77 (2013).

[13] Sanders, J. and Kandrot, E.: *CUDA by Example: An Introduction to General-Purpose GPU Programming*, Addison-Wesley Professional (2010).

[14] Sklansky, J.: Conditional-Sum Addition Logic, *IRE Trans. Electronic Computers*, EC-9, pp.226–231 (1960).

[15] Vai, M.: *VLSI DESIGN*, CRC Press (2000).

[16] Webster, M.T., Smith, N.G., and Ellegren, H.: Microsatellite Evolution Inferred from Human-Chimpanzee Genomic Sequence Alignments, *The National Academy of Sciences of the USA*, Vol.99, No.13, pp.8748–8753 (2002).

[17] Yang, J., Xu, Y. and Shang, Y.: An Efficient Parallel Algorithm for Longest Common Subsequence Problem on GPUs, *The World Congress on Engineering* (*WCE*), Vol.I (2010).

Katsuya Kawanami is a Ph.D. student of the Graduate School of Science at Osaka Prefecture University in Japan. He received his B.Sc. and M.Sc. degrees from Osaka Prefecture University. He is a member of IPSJ and JSCES. His research interests include GPGPU.

Noriyuki Fujimoto is a professor of the Graduate School of Science at Osaka Prefecture University in Japan. He received his B.Eng., M.Eng., and Dr.Eng. degrees from Osaka University in Japan. He is a member of IPSJ, IEICE, IEEE, and ACM. His research interests include high performace computing and combinatorial optimization. In particular, his research efforts now focus on GPGPU for combinatorial optimization.