IEICE TRANSACTIONS

on Information and Systems

VOL. E97-D NO. 12
DECEMBER 2014

A PUBLICATION OF THE INFORMATION AND SYSTEMS SOCIETY
The Institute of Electronics, Information and Communication Engineers
Kikai-Shinko-Kaikan Bldg., 5-8, Shibakoen 3 chome, Minato-ku, TOKYO, 105-0011 JAPAN

PAPER *Special Section on Parallel and Distributed Computing and Networking*

An Optimal Implementation of the Approximate String Matching on the Hierarchical Memory Machine, with Performance Evaluation on the GPU

Duhu MAN[†], ***Nonmember***, **Koji NAKANO**[† a)], ***and*** **Yasuaki ITO**[†], ***Members***

SUMMARY The Hierarchical Memory Machine (HMM) is a theoretical parallel computing model that captures the essence of computing on CUDA-enabled GPUs. The approximate string matching (ASM) for two strings X and Y of length m and n is a task to find a substring of Y most similar to X. The main contribution of this paper is to show an optimal parallel algorithm for the approximate string matching on the HMM and implement it on GeForce GTX 580 GPU. Our algorithm runs in $O(\frac{n}{w} + \frac{mn}{dw} + \frac{nL}{p} + \frac{mnl}{p})$ time units on the HMM with p threads, d streaming processors, memory band width w, global memory access latency L, and shared memory access latency l. We also show that the lower bound of the computing time is $\Omega(\frac{n}{w} + \frac{mn}{dw} + \frac{nL}{p} + \frac{mnl}{p})$ time units. Thus, our algorithm for the approximate string matching is time optimal. Further, we implemented our algorithm on GeForce GTX 580 GPU and evaluated the performance. The experimental results show that the ASM of two strings of 1024 and 4M ($= 2^{22}$) characters can be done in 419.6ms, while the sequential algorithm can compute it in 27720ms. Thus, our implementation on the GPU attains a speedup factor of 66.1 over the single CPU implementation.

key words: *memory machine models, approximate string matching, edit distance, GPU, CUDA*

1. Introduction

The GPU (Graphics Processing Unit), is a specialized circuit designed to accelerate computation for building and manipulating images [1]–[5]. Latest GPUs are designed for general purpose computing and can perform computation in applications traditionally handled by the CPU. Hence, GPUs have recently attracted the attention of many application developers [1]. NVIDIA provides a parallel computing architecture called *CUDA* (Compute Unified Device Architecture) [6], the computing engine for NVIDIA GPUs. CUDA gives developers access to the virtual instruction set and memory of the parallel computational elements in NVIDIA GPUs. In many cases, GPUs are more efficient than multicore processors [7], since they have hundreds of processor cores and very high memory bandwidth.

NVIDIA GPUs has streaming multiprocessors (SMs) each of which executes multiple threads in parallel. CUDA uses two types of memories in the NVIDIA GPUs: *the shared memory* and *the global memory* [6]. Each SM has the shared memory, an extremely fast on-chip memory with lower capacity, say, 16-48 Kbytes, and low latency. Every SM shares the global memory implemented as an off-chip DRAM, and has large capacity, say, 1.5-6 Gbytes, but its access latency is very large. The efficient usage of the shared memory and the global memory is a key for CUDA developers to accelerate applications using GPUs. In particular, we need to consider *the bank conflict* of the shared memory access and *the coalescing* of the global memory access [3], [7]–[9]. The address space of the shared memory is mapped into several physical memory banks. If two or more threads access the same memory banks at the same time, the access requests are processed in turn. Hence, to maximize the memory access performance, threads of CUDA should access distinct memory banks to avoid the bank conflicts of the memory accesses. To maximize the bandwidth between the GPU and the DRAM chips, the consecutive addresses of the global memory must be accessed at the same time. Thus, CUDA threads should perform coalesced access when they access the global memory.

In our previous paper [10], we have introduced three parallel computing models, *the Discrete Memory Machine (DMM)* and *the Unified Memory Machine (UMM)*, which reflect the essential features of the shared memory and the global memory of CUDA-enabled GPUs. The outline of the architectures of the DMM and the UMM is illustrated in Fig. 1. In both architectures, *a set of threads (Ts)* are connected to *the memory banks (MBs)* through *the memory management unit (MMU)*. Each thread is a Random Access Machine (RAM) [11], which can execute fundamental operations in a time unit. Threads are executed in SIMD [12] fashion, and the processors run on the same program and work on the different data. MBs constitute a single address space of the memory. A single address space of the memory is mapped to the MBs in an interleaved way such that the word of data of address i is stored in the $(i \bmod w)$-th bank, where w is the number of MBs. The main difference of the two architectures is the connection of the address line between the MMU and the MBs, which can transfer an address value. In the DMM, the address lines connect the MBs and the MMU separately, while a single address line from the MMU is connected to the MBs in the UMM. Hence, in the UMM, the same address value is broadcast to every MB, and the same address of the MBs can be accessed in each time unit. On the other hand, different addresses of the MBs can be accessed in the DMM.

Quite recently, we have introduced the Hierarchical

Manuscript received December 26, 2013.
Manuscript revised May 13, 2014.

[†]The authors are with the Department of Information Engineering, Hiroshima University, Higashihiroshima-shi, 739–8527 Japan.
a) E-mail: nakano@cs.hiroshima-u.ac.jp

DOI: 10.1587/transinf.2014PAP0011

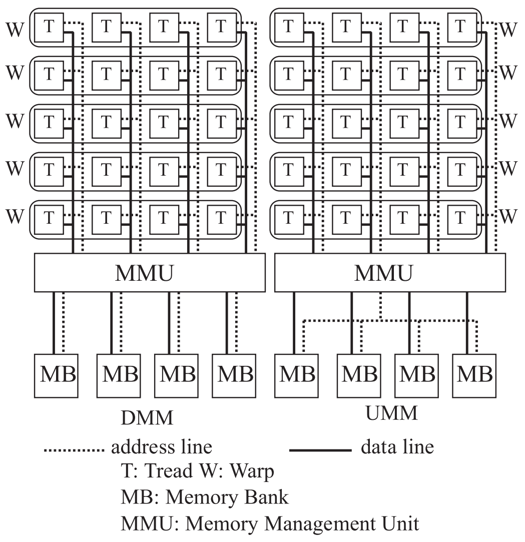


Fig. 1 The architectures of the DMM and the UMM with $w = 4$.

Memory Machine (HMM) [13], which is a hybrid of the DMM and the UMM. The HMM is a more practical parallel computing model that reflects the hierarchical architecture of CUDA-enabled GPUs. Figure 2 illustrates the architecture of the HMM. The HMM consists of d DMMs and a single UMM. Each DMM has w memory banks and the UMM also has w memory banks. We call the memory banks of each DMM *the shared memory* and those of the UMM *the global memory* after CUDA-enabled NVIDIA GPUs. Each DMM can work independently and can perform the computation using its shared memory. Also, all threads of DMMs work as a single UMM and can access to the global memory. While the memory access latency of the shared memory of CUDA-enables GPUs is very low, that of the global memory is several hundred clock cycles [6]. Hence, we use parameters l and L that denote the memory access latencies of the shared memory and the global memory, and assume $l \ll L$.

Suppose that two strings X and Y with length m and n ($m \leq n$), respectively, are given. The approximate string matching (ASM) is a task to find a substring in Y most similar to X. The similarity of two strings is measured by the number of three operations, insertion, deletion, and replacement of characters necessary to change one string into the other. The ASM has a lot of applications in the areas of signal processing, bio-informatics, natural language processing, among others. It is well known that the ASM can be computed in $O(mn)$ time [14] using the dynamic programming technique. Many researchers have been devoted to do research on variations of the ASM. For example, if the problem is to list substrings in Y with similarity no more than k, the computing time can be reduced [15]. Also, if the complicated bit operations of words is allowed, the ASM can be accelerated [16]. Utan *et. al* [17] implemented an approximate regular expression matching algorithm on the FPGA and the GPU.

Although a lot of work of sequential algorithms for the ASM have been published, there is no significant work for parallel algorithms for ASM. Since the computation of the ASM involves long sequential operations, it is very hard to parallelize it to run in poly-logarithmic time. Also, it is not difficult to obtain a cost-optimal linear-time parallel algorithm, which runs in $O(n)$ time using m processors on the PRAM. As a related result, a GPU implementation of k-mismatch ASM has been shown in [18]. However, this string matching is a task to find substrings with Hamming distance no more than k, which is much simpler than ASM. Quite recently, we have published optimal algorithm for the ASM on the DMM and the UMM [19]. This implementation runs in $O(\frac{mn}{w} + ml)$ time units on the DMM and on the UMM using n threads. However, since at most w threads perform computation in every time unit on the DMM and the UMM, it is not possible to accelerate the computation a factor of more than w.

The main contribution of this paper is to present an optimal implementation of the ASM algorithm on the HMM. Our implementation on the HMM achieves more speed-up than our previous work [19] on the DMM and the UMM. Our implementation runs in $O(\frac{n}{w} + \frac{mn}{dw} + \frac{nL}{p} + \frac{mnl}{p})$ time units on the HMM with d DMMs, width w, global memory latency L and shared memory latency l. The dominant factor mn in the running time is accelerated by a factor of dw. We also discuss the lower bound of the computing time for the ASM. More specifically, we prove four lower bounds: $\Omega(\frac{n}{w})$-time *bandwidth limitation*, $\Omega(\frac{mn}{dw})$-time *speed-up limitation*, $\Omega(\frac{nL}{p})$-time *global memory latency limitation*, and $\Omega(\frac{mnl}{p})$-time *shared memory latency limitation*. Thus, our implementation of the ASM on the HMM is time optimal. We also implemented our algorithm for the ASM on GeForce GTX-580 GPU. The experimental results show that the ASM of two strings of 1024 and 4M ($= 2^{22}$) characters can be computed in 419.6ms, while the sequential algorithm can compute it in 27720ms. Thus, our implementation on the GPU attains a speedup factor of 66.1 over the single CPU implementation.

2. Memory Machine Models: The DMM, the UMM, and the HMM

We first define *the Discrete Memory Machine (DMM)* [10], [20], [21] of width w and latency l. Let $m[i]$ ($i \geq 0$) denote a memory cell of address i in the memory. Let $B[j] = \{m[j], m[j+w], m[j+2w], m[j+3w], \ldots\}$ ($0 \leq j \leq w-1$) denote *the j-th bank* of the memory. Clearly, a memory cell $m[i]$ is in the (i mod w)-th memory bank. We assume that memory cells in different banks can be accessed in a time unit, but no two memory cells in the same bank can be accessed in a time unit. Also, we assume that l time units are necessary to complete an access request and continuous requests are processed in a pipeline fashion through the MMU. Thus, it takes $k+l-1$ time units to complete memory access requests to k memory cells in a particular bank.

We assume that p threads are partitioned into $\frac{p}{w}$ groups of w threads called *warps*. More specifically, p threads $T(0)$,

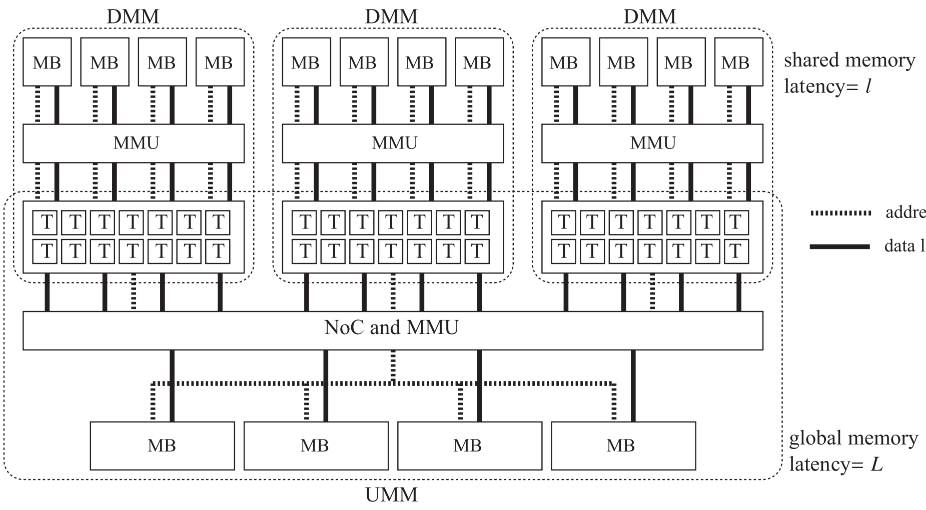


Fig. 2 The architecture of the HMM with $d = 3$ DMMs and width $w = 4$.

$T(1), \ldots, T(p-1)$ are partitioned into $\frac{p}{w}$ warps $W(0), W(1), \ldots, W(\frac{p}{w}-1)$ such that $W(i) = \{T(i\cdot w), T(i\cdot w+1), \ldots, T((i+1)\cdot w-1)\}$ $(0 \le i \le \frac{p}{w}-1)$. Warps are dispatched for memory access in turn, and w threads in a warp try to access the memory at the same time. In other words, $W(0), W(1), \ldots, W(\frac{p}{w}-1)$ are dispatched in a round-robin manner if at least one thread in a warp requests memory access. If no thread in a warp needs memory access, such warp is not dispatched for memory access. When $W(i)$ is dispatched, w threads in $W(i)$ send memory access requests, at most one request per thread, to the memory. We also assume that a thread cannot send a new memory access request until the previous memory access request is completed. Hence, if a thread sends a memory access request, it must wait at least l time units to send a new memory access request.

We next define *the Unified Memory Machine (UMM)* [10], [22] of width w and latency L as follows. Let $A[j] = \{m[j\cdot w], m[j\cdot w+1], \ldots, m[(j+1)\cdot w-1]\}$ denote the j-th address group. We assume that memory cells in the same address group are processed at the same time. However, if they are in the different groups, one time unit is necessary for each of the groups. Also, similarly to the DMM, p threads are partitioned into warps and each warp accesses the memory in turn.

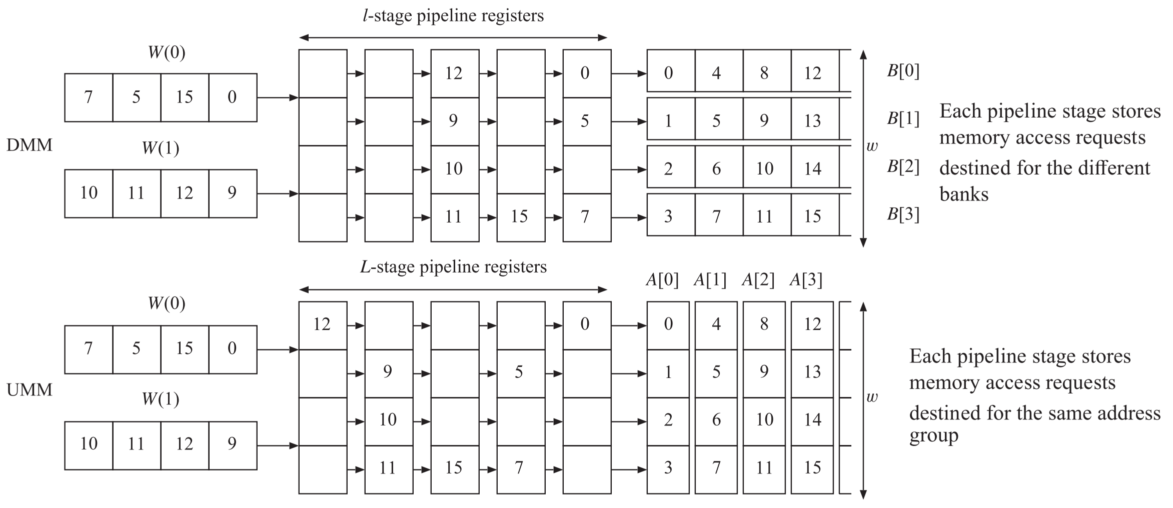
Figure 3 shows examples of memory access on the DMM and the UMM. We assume that each memory access request is completed when it reaches the last pipeline stage. Two warps $W(0)$ and $W(1)$ access to $\langle 7, 5, 15, 0\rangle$ and $\langle 10, 11, 12, 9\rangle$, respectively. In the DMM, memory access requests by $W(0)$ are separated into two pipeline stages, because addresses 7 and 15 are in the same bank $B(3)$. Those by $W(1)$ occupies 1 stage, because all requests are in distinct banks. Thus, the memory requests occupy three stages, it takes $3 + 5 - 1 = 7$ time units to complete the memory access. In the UMM, memory access requests by $W(0)$ are destined for three address groups. Hence the memory requests occupy three stages. Similarly those by $W(1)$ occupy two stages. Hence, it takes $5 + 5 - 1 = 9$ time units to complete the memory access.

Finally, we define *the Hierarchical Memory Machine (HMM)*. The HMM consists of d DMMs and a single UMM as illustrated in Fig. 2. Each DMM has w memory banks and the UMM also has w memory banks. We call the memory banks of each DMM *the shared memory* and those of the UMM *the global memory*. Each DMM works independently. Threads are partitioned into warps of w threads, and each warp are dispatched for the memory access for the shared memory in turn. Further, each warp of w threads in all DMMs can send memory access requests to the global memory. Figure 2 illustrates the architecture of the HMM with $d = 3$ DMMs. Each DMM and the UMM has $w = 4$ memory banks. The shared memory of each DMM and the global memory of the UMM correspond to “the shared memory” of each streaming multiprocessor and “the global memory” of CUDA-enabled GPUs.

3. Coalesced and Conflict-Free Memory Access

This section evaluate the performance of coalesced memory access for the global memory and the conflict-free memory access for the shared memory. These memory access operations are key ingredients of our ASM algorithm.

A round of memory access is an operation such that every thread performs a single memory access to the shared memory or the global memory. A round of memory access by a warp of w threads is *coalesced* if all memory access by a warp destined for the same address group of the global memory. Also, that by a warp is *conflict-free* if all memory access by a warp destined for the distinct memory banks of the shared memory. More specifically, a round of the memory access by a warp is *coalesced* if $\lfloor\frac{a(0)}{w}\rfloor = \lfloor\frac{a(1)}{w}\rfloor = \cdots = \lfloor\frac{a(w-1)}{w}\rfloor$, where $a(i)$ $(0 \le i \le w-1)$ is the address accessed by thread $T(i)$ in the warp. A round of the memory access by a warp is *conflict-free* if, for all pair i and j $(0 \le i < j \le w-1)$, $a(i) = a(j)$ or $a(i) \not\equiv a(j)$ (mod w). We also say that a round of the memory access by



l-stage pipeline registers
$W(0)$
DMM
$W(1)$
$B[0]$
$B[1]$
$B[2]$
$B[3]$
w
Each pipeline stage stores memory access requests destined for the different banks
L-stage pipeline registers
$A[0]$ $A[1]$ $A[2]$ $A[3]$
UMM
Each pipeline stage stores memory access requests destined for the same address group

Fig. 3 Examples of memory access on the DMM and the UMM.

all of the p threads is *coalesced* if memory access by all of the $\frac{p}{w}$ warps is coalesced. Also, that by p threads is *conflict-free* if memory access by every warp is conflict-free.

Let us evaluate the time necessary for coalesced and conflict-free memory access. Suppose that p ($\geq w$) threads perform a round of coalesced memory access to the global memory. Since we have $\frac{p}{w}$ warps each of which sends w memory requests to the same address group, it takes $\frac{p}{w}$ time units to send all p memory requests. After that $L - 1$ time units are necessary to complete the memory requests by the last warp. Thus, it takes $\frac{p}{w} + L - 1$ time units to complete a round of coalesced memory access by p threads. Similarly, a round of conflict-free memory access for the shared memory takes $\frac{p}{w}$ time units to send all memory requests and $l - 1$ time units are necessary to complete the memory requests by the last warp. Thus, a round of coalesced memory access for the global memory and that of conflict-free memory access for the shared memory by p threads take $O(\frac{p}{w} + L)$ time units and $O(\frac{p}{w} + l)$ time units, respectively. Suppose that p threads access to n ($\geq p$) words of the global memory in $\frac{n}{p}$ rounds. If all rounds are coalesced memory access for the global memory, it takes $O(\frac{p}{w} + L) \cdot \frac{n}{p} = O(\frac{n}{w} + \frac{nL}{p})$ time units. Similarly, $\frac{n}{p}$ rounds memory access for the shared memory take $O(\frac{n}{w} + \frac{nl}{p})$ time units. Thus, we have,

Lemma 1: The coalesced memory access to n words of the global memory and the conflict-free memory access to n words of the shared memory take $O(\frac{n}{w} + \frac{nL}{p})$ time units and $O(\frac{n}{w} + \frac{nl}{p})$ time units, respectively, if $n \geq p \geq w$.

4. Approximate String Matching and Edit Distance

The main purpose of this section is to review approximate string matching (ASM) and the edit distance (ED). Please see [14], [19], [23] for the details.

As a preliminary, we first define the edit distance (ED) of two strings. Suppose that source string $X = x_1x_2 \cdots x_m$ of length m and destination string $Y = y_1y_2 \cdots y_n$ of length n are given. Without loss of generality, we can assume that $m \leq n$. We want to change X into Y using the following three operations:

- insertion of a character,
- deletion of a character, and
- replacement of a character.

For example, $X = ababa$ can be changed into $Y = aaabbb$ in five operations as follows: $ababa \xrightarrow{delete} aaba \xrightarrow{delete} aaa \xrightarrow{insert} aaab \xrightarrow{insert} aaabb \xrightarrow{insert} aaabbb$. Alternatively, X can be changed into Y in three operations as follows: $ababa \xrightarrow{replace} aaaba \xrightarrow{replace} aaabb \xrightarrow{insert} aaabbb$. *The ED of two strings* is the minimum number of operations to change one string to the other. For example, the ED of X and Y above is three, because there exists a sequence of three operations to change X into Y, and there exists no sequence of less than three operations to do the same thing. For later reference, let ED(X, Y) denote the ED of X and Y.

The approximate string matching, a more flexible version of the edit distance, is a task to compute the value of ASM(X, Y) defined as follows:

$$\mathrm{ASM}(X, Y) = \min\{\mathrm{ED}(X, Y') \mid Y' \text{ is a substring of } Y\}$$

Clearly, ASM(X, Y) is small if Y has a substring similar to X. It should be clear that ASM(X, Y) is always less than or equal to m, and ED(X, Y) takes a value between $n - m$ and n. For example, if X and Y share no character, then ED$(X, Y) = n$ and ASM$(X, Y) = m$. Also, if the prefix of Y is X then ED$(X, Y) = n - m$ and ASM$(X, Y) = 0$.

We use a matrix c of size $(m + 1) \times (n + 1)$ to compute the ASM. Each $c[i][j]$ $(0 \leq i \leq m, 0 \leq j \leq n)$ is used to store the following value:

$$\min_{1 \leq j' \leq j} \mathrm{ED}(x_1x_2 \cdots x_i, y_{j'}y_{j'+1} \cdots y_j).$$

Note that $x_1x_2 \cdots x_i$ is a null string (i.e. a string with length

			Y							
			a	a	a	b	b	b	a	a
		0	0	0	0	0	0	0	0	0
	a	1	0	0	0	1	1	1	0	0
	b	2	1	1	1	0	1	1	1	1
X	a	3	2	1	1	1	1	2	1	1
	b	4	3	2	2	1	1	1	2	2
	a	5	4	3	2	2	2	2	1	2

Fig. 4 The values of matrix c for the ASM.

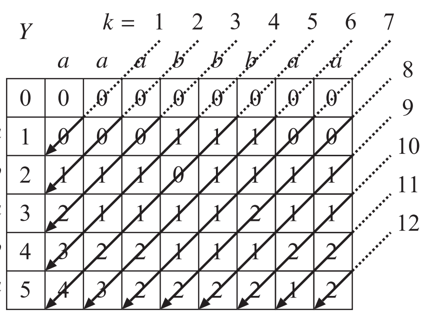


Fig. 5 Illustrating a parallel algorithm for computing matrix c.

0) if $i = 0$. Once all values of c is computed, we can compute the value of ASM(X, Y) by the following formula:

$$\mathrm{ASM}(X, Y) = \min_{0 \le j \le n} c[m][j]$$

Let us show how we compute all values of c. Suppose that $c[i-1][j-1]$, $c[i-1][j]$, and $c[i][j-1]$ are already computed. Let "$x_i \neq y_j$" denote the binary value such that it is 1 if $x_i \neq y_j$ and 0 if $x_i = y_j$. The value of $c[i][j]$ can be computed as follows:

$$\begin{aligned} c[i][j] &= 0 \quad \text{if } i = 0 & (1)\\ &= i \quad \text{if } j = 0 & (2)\\ &= \min(c[i][j-1] + 1, & (3)\\ &\qquad c[i-1][j] + 1, & (4)\\ &\qquad c[i-1][j-1] + (x_i \neq y_j)) & (5)\\ &\qquad \text{if } i > 0 \text{ and } j > 0. \end{aligned}$$

Using this formula, all values of matrix c can be computed as follows:

```
[Algorithm ASM]
for $j \leftarrow 1$ to $n$ do $c[0][j] \leftarrow 0$
for $i \leftarrow 0$ to $m$ do $c[i][0] \leftarrow i$
for $i \leftarrow 1$ to $m$ do
  for $j \leftarrow 1$ to $n$ do
    $c[i][j] \leftarrow \min(c[i][j-1] + 1, c[i-1][j] + 1,$
      $c[i-1][j-1] + (x_i \neq y_j))$
output $\min\{c[m][j] \mid 0 \le j \le n\}$
```

Figure 4 shows the values of c for two strings $X = ababa$ and $Y = aaabbbaa$. From the figure, we can see that the ASM of X and Y is 1.

Usually, the approximate string matching should require algorithms to return the indexes i and j such that ASM$(X, Y) = $ ED$(X, y_i y_{i+1} \cdots y_j)$. The reader should have no difficulty to confirm that once all the values in matrix c is obtained, it is not difficult to compute such values of i and j.

5. A Parallel ASM Algorithm on the DMM

The main purpose of this section is to show a parallel algorithm for computing the ASM on a single DMM of the HMM. We assume that, two input strings X and Y of length m and n are given in the shared memory.

The key idea is to compute the values of the matrix c from the top-left corner to the bottom-right corner as illustrated in Fig. 5. The details of the parallel algorithm are spelled out as follows:

```
[Parallel ASM algorithm]
for $j \leftarrow 1$ to $n$ do in parallel $c[0][j] \leftarrow 0$
for $i \leftarrow 0$ to $m$ do in parallel $c[i][0] \leftarrow i$
for $k \leftarrow 1$ to $n + m - 1$ do
  for $i \leftarrow 1$ to $m$ do in parallel
    begin
      $j \leftarrow k - i + 1$
      if $1 \le j \le n$ then
        $c[i][j] \leftarrow \min(c[i][j-1] + 1, c[i-1][j] + 1,$
          $c[i-1][j-1] + (x[i] \neq y[j]))$
    end
output $\min\{c[m][j] \mid 0 \le j \le n\}$
```

In the third for-loop, for each k $(1 \le k \le n + m - 1)$, the values $c[1][k], c[2][k-1], \ldots, c[m][k-m]$ are computed and stored. It should be clear that this parallel algorithm correctly computes the ASM.

Clearly, when the values of c for k is computed, only those for $k-1$ and $k-2$ are used. Thus, it is sufficient to use a matrix e of size $3 \times (m+1)$ that stores values of c for $k-2$, $k-1$, and k. We assume that $m+1$ is a multiple of w to guarantee that $e[j][i]$ and $e[j'][i']$ are in different banks of the shared memory iff $i \neq i'$. If this is not the case, we use a matrix c of size $3 \times (m'+1)$ such that $m'+1$ is the minimum multiple of w exceeding $m+1$. Let e be a matrix of size $3 \times (m+1)$ such that the value of each $c[i][j]$ $(0 \le i \le m, 0 \le j \le n)$ are stored in $e[j \bmod 3][i]$. The ASM can also be computed using array e as follows:

```
[Improved parallel ASM algorithm]
$minval \leftarrow m$; $e[0][0] \leftarrow 0$; $e[0][1] \leftarrow 1$; $e[1][0] \leftarrow 0$
for $k \leftarrow 1$ to $n + m - 1$ do
  begin
    for $i \leftarrow 0$ to $m$ do in parallel
      begin
        $j \leftarrow k - i + 1$
        if $i = 0$ then $e[j \bmod 3][i] \leftarrow 0$
        else if $j = 0$ then $e[j \bmod 3][i] \leftarrow i$
        else if $1 \le j \le n$ then
          $e[j \bmod 3][i] \leftarrow \min(e[(j-1) \bmod 3][i] + 1,$
            $e[j \bmod 3][i-1] + 1, e[(j-1) \bmod 3][i-1]$
            $+(x[i] \neq y[j]))$
        end
      if $k \geq m$ and $e[j \bmod 3][m] < minval$
      then $minval \leftarrow e[j \bmod 3][m]$
   end
output $minval$
```

Let us evaluate the computing time using p ($\geq w$) threads on the DMM. Initializations of $e[0][0]$, $e[0][1]$, and $e[1][0]$, and can be done in $O(1)$ time using a single thread. The memory access operations to compute the values to be stored in $e[j \bmod 3][i]$ in parallel are conflict-free. For example, parallel reading operation for $e[(j-1) \bmod 3][i-1]$ is done for $e[*][0], e[*][1], \ldots, e[*][m-1]$, where each "*" denotes an integer in $[0, 2]$. Since e is a matrix of size $3\times(m+1)$ and $m+1$ is a multiple of w, each $e[*][i]$ ($0 \leq i \leq m-1$) is in the ($i \bmod w$)-th bank. Hence, this reading operation is conflict-free and the memory access operation to store the values in $e[j \bmod 3][i]$ can be doe in $O(\frac{m}{w} + \frac{ml}{p})$ time units using p threads on the DMM from Lemma 1. Also, it involves the computation of *minval*, which can be computed in $O(1)$ time by evaluating the if statement using a single thread. Thus, the for-loop for a fixed k takes $O(\frac{m}{w} + \frac{ml}{p})$ time units. Since these memory access operations are performed $n+m-1$ times, this algorithm runs in $(n+m-1) \cdot O(\frac{m}{w} + \frac{ml}{p}) = O(\frac{mn}{w} + \frac{mnl}{p})$ time units. Thus, we have,

Lemma 2: The ASM of two strings of length m and n ($m \leq n$) can be computed in $O(\frac{mn}{w} + \frac{mnl}{p})$ time units using p ($w \leq p \leq m$) threads on the DMM.

6. A Parallel ASM Algorithm on the HMM

This section is devoted to show a parallel algorithm for the ASM using d DMMs on the HMM. We assume that X and Y of length m and n each are stored in the global memory of the HMM. Also, we assume that n and p are large enough such that $n \geq wd$ and $p \geq wd$. Since wd threads on the HMM can work at the same time, it makes sense to assume that $p \geq wd$.

Recall that $n - m \leq ED(X,Y) \leq n$. Hence, if $n > 2m$ then $ED(X,Y) > m$. On the other hand, $\text{ASM}(X,Y) \leq m$ always holds. Hence, we can ignore substrings with more than $2m$ characters of Y, when we compute $\text{ASM}(X,Y)$. In other words, we can write

$$\text{ASM}(X,Y) = \min\{\text{ED}(X,Y') \mid Y' \text{ is a substring of } Y \text{ with at most } 2m \text{ characters}\}.$$

Let $Y_0, Y_1, \ldots, Y_{d-1}$ be d substrings of Y such that any substring of Y with at most $2m$ characters is included in at least one of Y_i's. For such Y_i's, we can write

$$\text{ASM}(X,Y) = \min\{\text{ED}(X,Y_i) \mid 0 \leq i \leq d-1\}.$$

Thus, by computing $\text{ASM}(X,Y_i)$ for all i ($0 \leq i \leq d-1$) in parallel, we can obtain $\text{ASM}(X,Y)$. Let $Y_i = y_{is}y_{is+1}\cdots y_{(i+1)s+2m-1}$ for all i ($0 \leq i \leq d-1$), where $s = \frac{n-2m}{d}$. For simplicity, we assume that s is an integer. Clearly, all substrings with at most $2m$ characters in Y are included in at least one of Y_i's. For example, if $n = 1024$, $m = 32$, and $d = 4$ then, $s = 240$, and $Y_0 = y_0y_1\cdots y_{303}$, $Y_1 = y_{240}y_{241}\cdots y_{543}$, $Y_2 = y_{480}y_{481}\cdots y_{783}$, and $Y_3 = y_{720}y_{721}\cdots y_{1023}$. We can confirm that any substring with at most $2m = 64$ characters in Y is included at least one of Y_0, Y_1, Y_2, and Y_3.

We can compute $\text{ASM}(X,Y)$ as follows:

[Parallel ASM algorithm on the HMM]

Step 1: Each DMM(i) reads X and Y_i from the global memory and write them in the shared memory.

Step 2: Each DMM(i) computes $\text{ASM}(X,Y_i)$ in parallel.

Step 3: Each DMM(i) writes the value of $\text{ASM}(X,Y_i)$ in the global memory.

Step 4: Compute the $\min\{\text{ASM}(X,Y_i) \mid 0 \leq i \leq d-1\}$.

We assume that we use $\frac{p}{d}$ threads for each of the d DMMs and evaluate the computing time. In Step 1, to read X by d DMMs, the reading operation for md characters is performed by p threads. Hence, from Lemma 1, it takes $O(\frac{md}{w} + \frac{mdL}{p})$ time units to read X from the global memory. Similarly, the reading of every Y_i from the global memory takes $O(\frac{(s+2m)d}{w} + \frac{(s+2m)dL}{p})$ time units. Also, writing X and Y_i in the shared memory of each DMM is performed independently. From Lemma 1, writing operations of X and Y_i take $O(\frac{m}{w} + \frac{ml}{p})$ time units and $O(\frac{s+2m}{w} + \frac{(s+2m)l}{p})$ time units, respectively. Therefore, Step 1 takes $O(\frac{(s+m)d}{w} + \frac{(s+m)dL}{p}) = O(\frac{n+md}{w} + \frac{(n+md)L}{p})$ time units from $s < \frac{n}{d}$. In Step 2, the computation of each $\text{ASM}(X,Y_i)$ takes $O(\frac{(s+2m)m}{w} + \frac{(s+2m)ml}{\frac{p}{d}}) \leq O(\frac{(n+md)m}{dw} + \frac{(n+md)ml}{p})$ time units from Lemma 2. Note that, $w \leq \frac{p}{d} \leq m$ must be satisfied to use Lemma 2. In Step 3, one thread in DMM(i) writes the value of $\text{ASM}(X,Y_i)$ in the global memory. Since we have d DMMs, Step 3 takes $O(d+L) \leq O(\frac{n}{w} + L)$ time units from $n \geq wd$. Finally, Step 4 computes the minimum of $\text{ASM}(X,Y_i)$ in $O(\frac{d}{w} + \frac{dL}{p} + L)$ time units using p threads on the UMM using the algorithm in [24]. The computing time of the four steps combined, the ASM can be computed in $O(\frac{n+md}{w} + \frac{m(n+md)}{dw} + \frac{(n+md)L}{p} + \frac{m(n+md)l}{p})$ time units.

Lemma 3: The ASM of two strings of length m and n ($m \leq n$) can be computed in $O(\frac{n+md}{w} + \frac{m(n+md)}{dw} + \frac{(n+md)L}{p} + \frac{m(n+md)l}{p})$ using p threads ($wd \leq p \leq md$) on the HMM with d DMMs, width w, shared memory latency l, and global memory latency L.

The parallel ASM algorithm for Lemma 3 uses up to md threads, and the latency overhead $O(\frac{(n+md)L}{p} + \frac{m(n+md)l}{p})$ is minimized when $p = md$. If this is the case, the latency overhead is $O(\frac{nL}{md} + \frac{nl}{d} + L + ml)$, which may be dominant when m and d are not large. To reduce this latency overhead, we should use more than md threads More specifically, we use $p = mD$ threads such that $d < D \leq n$. Since mD threads are available, we use D substrings $Y_0, Y_1, \ldots, Y_{D-1}$ of Y such that each substring Y_i ($0 \leq i \leq D-1$) has $S + 2m$ characters, where $S = \frac{n-2m}{D}$. We assign m threads to compute each

Table 1 The running time (milliseconds) of parallel ASM algorithm on the HMM for $|Y| = 4M$ $(= 2^{22})$.

	GPU CUDA blocks D								CPU	speed-up
$|X| = m$	16	32	64	128	256	512	1024	2048		
32	178.2	89.23	44.92	23.46	23.53	23.64	23.90	24.38	701.8	29.9
64	178.6	89.71	46.74	29.13	29.13	29.18	29.39	30.01	1364	46.8
128	181.1	92.66	55.81	48.40	48.51	48.92	50.16	53.36	2683	55.4
256	187.0	112.2	95.77	93.16	91.83	93.83	100.1	113.3	5295	57.7
512	236.5	191.0	184.6	181.3	185.0	197.9	224.1	277.9	10560	58.2
1024	419.6	423.8	432.3	449.4	483.4	551.5	687.7	960.0	27720	66.1

ASM(X, Y_i). In other words, $p = mD$ threads are arranged in d DMMs, and each DMM computes $\frac{D}{d}$ ASM(X, Y_i)s using $\frac{mD}{d}$ threads. If this is the case, each of the four steps takes the following computing time: Step 1: $O(\frac{n+mD}{w} + \frac{(n+mD)L}{p})$ time units, Step 2: $O(\frac{(n+mD)m}{dw} + \frac{(n+mD)ml}{p}$ time units, Step 3: $O(D + L) < O(\frac{n}{w} + L)$ time units, and Step 4: $O(\frac{D}{w} + \frac{DL}{p} + L)$ time units. Thus, the ASM can be computed in $O(\frac{n+mD}{w} + \frac{m(n+mD)}{dw} + \frac{(n+mD)L}{p} + \frac{m(n+mD)l}{p}) = O(\frac{n+p}{w} + \frac{m(n+p)}{dw} + \frac{nL}{p} + \frac{mnl}{p} + L + ml)$ time units. Thus, we have,

Theorem 4: The ASM of two strings of length m and n $(m \leq n)$ can be computed in $O(\frac{n+p}{w} + \frac{m(n+p)}{dw} + \frac{nL}{p} + \frac{mnl}{p} + L + ml)$ using p threads $(wd \leq p \leq mn)$ on the HMM.

If Y is large enough such that $n \geq p$, we can simplify the computing time as follows:

Corollary 5: The ASM of two strings of length m and n $(m \leq n)$ can be computed in $O(\frac{n}{w} + \frac{mn}{dw} + \frac{nL}{p} + \frac{mnl}{p})$ using p threads on the HMM if $wd \leq p \leq n$.

7. The Lower Bounds of the Computing Time

The main purpose of this section is to show the lower bound of the computing time of implementations for the approximate string matching on the HMM.

To compute the ASM, all of the n characters in Y stored in the global memory must be read at least once. Recall that w memory banks constitute the global memory, and each of the memory banks can handle at most one memory access request in a time unit. Thus, the global memory can process at most w memory access requests in a time unit. Since at least one memory access request must be destined for each character in Y, n memory access requests to the global memory are necessary. Thus, the ASM takes at least $\Omega(\frac{n}{w})$ time units (*bandwidth limitation*).

The HMM has d DMMs, each of which can execute w instructions in a time unit. Thus, the HMM can perform dw instructions in a time unit. Since Algorithm ASM involves at least mn additions, any implementation of Algorithm ASM on the HMM takes $\Omega(\frac{mn}{dw})$ time units (*speed-up limitation*).

Each thread can send at most one memory request in L time units to the global memory. Hence, p threads can send at most $\frac{pt}{L}$ memory requests in t time units. Since every character of Y stored in the global memory must be accessed at least once, $\frac{pt}{L} \geq n$ must be satisfied to access all characters and $t \geq \frac{nL}{p}$ must be satisfied. Hence, $\Omega(\frac{nL}{p})$ time units are necessary (*global memory latency limitation*).

When we implement a dynamic programming based ASM algorithm such as Algorithm ASM or Parallel ASM in the HMM, elements in a matrix c in Algorithm ASM (or a matrix e in Parallel ASM algorithm) must be arranged in the global memory or the shared memory of the HMM. Since we assumed that the shared memory access latency l is much smaller than the global memory access latency L, we can assume that the matrix is arranged in the shared memory for the lower bound discussion. In other words, it takes at least l time units to complete the memory access request to an element of the matrix arranged in the shared memory. On the other hand, a thread can send a new memory access request only after the previous memory access request is completed. Thus, if the HMM has p threads, it cannot have more than p memory access requests being processed. Hence, for any parameter k, all p threads cannot send more than pk memory access requests in lk time units. Since a dynamic programming based ASM algorithm accesses elements in the matrix at least mn times, $pk \geq mn$ is a necessary condition. Thus, it takes at least $lk \geq \frac{mnl}{p}$ time units for p threads to send mn memory access requests to the matrix (*shared memory latency limitation*).

Thus, we have

Theorem 6: Any implementation of the ASM for two strings of length m and n needs at least $\Omega(\frac{n}{w} + \frac{mn}{dw} + \frac{nL}{p} + \frac{mnl}{p})$ time units using p threads on the HMM with d DMMs, width w, global memory latency L, and shared memory latency l.

From Theorem 6, our ASM algorithm on the HMM shown for Corollary 5 is time optimal.

8. Experimental Results

We have implemented our parallel ASM algorithm for the HMM on the GPU and the sequential ASM algorithm on a single CPU. For these experiments, we used a PC with GeForce GTX-580 GPU and Intel Xeon CPU X7460 (2.66GHz). CUDA 4.2 and gcc 4.4.4-14 with option -O2 running on Ubuntu 10.10 (32-bit) have been used. GeForce GTX-580 GPU has 16 streaming multiprocessors. The size w of a warp is 32. Table 1 shows the running time for Y with 4M $(= 2^{22})$ characters and X with 32, 64, 128, 256, 512, 1024, and 2048. We partition the input Y into $D = 16, 32, 64, 128, 256, 512, 1024$ substrings, and

D CUDA blocks of m threads are invoked to compute ASM(X, Y_i) $(0 \leq i \leq D-1)$. Strings of X and Y are stored as arrays of 8-bit unsigned char initialized by random 0/1 values in the global memory. Since X and Y are random 0/1 strings, "$x_i \neq y_j$" is true with probability $\frac{1}{2}$. Such strings are unfavorable for GPUs, because the resulting values of "$x_i \neq y_j$" by all threads in a warp are not the same with high probability. From the table, the ASM of two strings of 1024 and 4M $(= 2^{22})$ characters can be computed in 419.6ms when $D = 16$, while the sequential algorithm can compute it in 27720ms. Thus, our implementation on the GPU attains a speedup factor of 66.1 over the single CPU implementation.

From Theorem 4, our implementation on the GPU runs in $O(\frac{n+p}{w} + \frac{m(n+p)}{dw} + \frac{nL}{p} + \frac{mnl}{p} + L + ml)$ time units. This computing time has two factors: $O(\frac{n+p}{w} + \frac{m(n+p)}{dw})$ time units for memory bandwidth and computation, and $O(\frac{nL}{p} + \frac{mnl}{p} + L + ml)$ time units for memory access latency. Clearly, if p is small, the memory access latency dominates the memory bandwidth and computation. On the other hand, the memory bandwidth and computation dominates the memory access latency for large p. To minimize the computing time, we should select the number p $(= mD)$ threads so that these two factors are balanced. From Table 1, we can see the number p of threads that minimizes the computing time. For example, when $m = 512$, our implementation takes the minimum computing time 181.3ms if 128 CUDA blocks or $512 \times 128 = 64K$ threads are used. This fact implies that the experimental results follow the theoretical analysis of the computing time.

9. Conclusion

We have presented a parallel approximate matching algorithm of two strings of length m and n, which runs $O(\frac{n}{w} + \frac{mn}{dw} + \frac{nL}{p} + \frac{mnl}{p})$ time units using p threads on the Hierarchical Memory Machine (HMM) with d Discrete Memory Machines (DMMs), width w, shared memory access latency l, and global memory access latency L. We also proved that this algorithm is time optimal. Further, we have implemented this algorithm on GeForce GTX-580 GPU. Our implementation achieves a speedup of 66.1 over the single CPU implementation.

References

[1] W.W. Hwu, GPU Computing Gems Emerald Edition, Morgan Kaufmann, 2011.
[2] K. Ogawa, Y. Ito, and K. Nakano, "Efficient Canny edge detection using a GPU," Proc. International Conference on Networking and Computing, pp.279–280, Nov. 2010.
[3] D. Man, K. Uda, Y. Ito, and K. Nakano, "A GPU implementation of computing Euclidean distance map with efficient memory access," Proc. International Conference on Networking and Computing, pp.68–76, Dec. 2011.
[4] A. Uchida, Y. Ito, and K. Nakano, "Fast and accurate template matching using pixel rearrangement on the GPU," Proc. International Conference on Networking and Computing, pp.153–159, Dec. 2011.
[5] Y. Ito, K. Ogawa, and K. Nakano, "Fast ellipse detection algorithm using Hough transform on the GPU," Proc. International Conference on Networking and Computing, pp.313–319, Dec. 2011.
[6] NVIDIA Corporation, "NVIDIA CUDA C programming guide version 5.0," 2012.
[7] D. Man, K. Uda, H. Ueyama, Y. Ito, and K. Nakano, "Implementations of a parallel algorithm for computing euclidean distance map in multicore processors and GPUs," Int. J. Networking and Computing, vol.1, no.2, pp.260–276, July 2011.
[8] NVIDIA Corporation, "NVIDIA CUDA C best practice guide version 3.1," 2010.
[9] K. Nishida, Y. Ito, and K. Nakano, "Accelerating the dynamic programming for the optial poygon triangulation on the GPU," Proc. International Conference on Algorithms and Architectures for Parallel Processing (ICA3PP, LNCS 7439), pp.1–15, Sept. 2012.
[10] K. Nakano, "Simple memory machine models for GPUs," Proc. International Parallel and Distributed Processing Symposium Workshops, pp.788–797, May 2012.
[11] A.V. Aho, J.D. Ullman, and J.E. Hopcroft, Data Structures and Algorithms, Addison Wesley, 1983.
[12] M.J. Flynn, "Some computer organizations and their effectiveness," IEEE Trans. Comput., vol.C-21, pp.948–960, 1972.
[13] K. Nakano, "The hierarchical memory machine model for GPUs," Proc. International Parallel and Distributed Processing Symposium Workshops, pp.591–600, May 2013.
[14] P.H. Sellers, "The theory and computation of evolutionary distances: Pattern recognition," J. Algorithms, vol.1, no.4, pp.359–373, Dec. 1980.
[15] E. Ukkonen, "Algorithms for approximate string matching," Information and Control, vol.64, no.1–3, pp.100–118, Jan.–March 1985.
[16] G. Myers, "A fast bit-vector algorithm for approximate string matching based on dynamic programming," J. ACM, vol.46, no.3, pp.395–415, May 1999.
[17] Y. Utan, M. Inagi, S. Wakabayashi, and S. Nagayama, "A GPGPU implementation of approximate string matching with regular expression operators and comparison with its FPGA implementation," Proc. Int. Conf. Parallel and Distributed Processing Techniques and Applications, July 2012.
[18] Y. Liu, L. Guo, J. Li, M. Ren, and K. Li, "Parallel algorithms for approximate string matching with k mismatches on cuda," Proc. International Parallel and Distributed Processing Symposium Workshops, pp.2414–2422, May 2012.
[19] K. Nakano, "Efficient implementations of the approximate string matching on the memory machine models," Proc. International Conference on Networking and Computing, pp.233–239, Dec. 2012.
[20] A. Kasagi, K. Nakano, and Y. Ito, "An implementation of conflict-free off-line permutation on the GPU," Proc. International Conference on Networking and Computing, pp.226–232, 2012.
[21] K. Nakano, S. Matsumae, and Y. Ito, "The random address shift to reduce the memory access congestion on the discrete memory machine," Proc. International Symposium on Computing and Networking, pp.95–103, Dec. 2013.
[22] K. Nakano, "Sequential memory access on the unified memory machine with application to the dynamic programming," Proc. International Symposium on Computing and Networking, pp.85–94, Dec. 2013.
[23] T.H. Cormen, C.E. Leiserson, and R.L. Rivest, Introduction to Algorithms, MIT Press, 1990.
[24] K. Nakano, "An optimal parallel prefix-sums algorithm on the memory machine models for GPUs," Proc. International Conference on Algorithms and Architectures for Parallel Processing (ICA3PP, LNCS 7439), pp.99–113, Sept. 2012.

Duhu Man received the ME and Ph.D degrees from the Department of Information Engineering, Hiroshima University in 2010 and 2013 respectively. Currently, he is a project assistant professor at the Department of Information Engineering, Hiroshima University.

Koji Nakano received the BE, ME and Ph.D degrees from Department of Computer Science, Osaka University, Japan in 1987, 1989, and 1992 respectively. In 1992–1995, he was a Research Scientist at Advanced Research Laboratory. Hitachi Ltd. In 1995, he joined Department of Electrical and Computer Engineering, Nagoya Institute of Technology. In 2001, he moved to School of Information Science, Japan Advanced Institute of Science and Technology, where he was an associate professor. He has been a full professor at School of Engineering, Hiroshima University from 2003. He has published extensively in journals, conference proceedings, and book chapters. He served on the editorial board of journals including IEEE Transactions on Parallel and Distributed Systems, IEICE Transactions on Information and Systems, and International Journal of Foundations on Computer Science. He has also guest-edited several special issues including IEEE TPDS Special issue on Wireless Networks and Mobile Computing, IJFCS special issue on Graph Algorithms and Applications, and IEICE Transactions special issue on Foundations of Computer Science. He has organized conferences and workshops including International Conference on Networking and Computing, International Conference on Parallel and Distributed Computing, Applications and Technologies, IPDPS Workshop on Advances in Parallel and Distributed Computational Models, and ICPP Workshop on Wireless Networks and Mobile Computing. His research interests includes image processing, hardware algorithms, GPU-based computing, FPGA-based reconfigurable computing, parallel computing, algorithms and architectures.

Yasuaki Ito received B.E. degree from Nagoya Institute of Technology (Japan), M.S. degree from Japan Advanced Institute of Science and Technology in 2003, and D.E. degree from Hiroshima University (Japan), in 2010. From 2004 to 2007 he was a Research Associate at Hiroshima University. Since 2007, Dr. Ito has been with the School of Engineering, at Hiroshima University, where he is working as an Associate Professor. His research interests include reconfigurable architectures, parallel computing, computational complexity and image processing.