The Approximate String Matching on the Hierarchical Memory Machine, with Performance Evaluation

Duhu Man, Koji Nakano, and Yasuaki Ito
Department of Information Engineering
Hiroshima University
Kagamiyama 1-4-1, Higashi Hiroshima, 739-8527 Japan

***Abstract*—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 to implement it on a CUDA-enabled GPU. Our algorithm runs in $O(\frac{n}{w}+\frac{mn}{dw}+\frac{nL}{p}+\frac{mnl}{p})$ on the HMM with d streaming processors, memory band width w, global memory access latency L, and shared memory access latency l. Further, we implement our algorithm on GeForce GTX 580 GPU and evaluate the performance. 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.**

***Keywords*-memory machine models, approximate string matching, edit distance, GPU, CUDA**

I. Introduction

The GPU (Graphics Processing Unit), is a specialized circuit designed to accelerate computation for building and manipulating images [1], [2], [3]. 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) [4], 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 [5], 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* [4]. 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 [2], [5], [6], [7]. 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 global memory the consecutive addresses must be accessed at the same time. Thus, CUDA threads should perform coalesced access when they access the global memory.

In our previous paper [8], we have introduced two 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 NVIDIA GPUs. Algorithms on the DMM and the UMM correspond to the computation using the shared memory and the global memory of GPUs, respectively. Later, we have introduced the Hierarchical Memory Machine (HMM), which is a hybrid of the DMM and the UMM [9]. The HMM is a more practical parallel computing model that reflects the hierarchical architecture of CUDA-enabled GPUs. Figure 1 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 [4]. 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 of 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 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

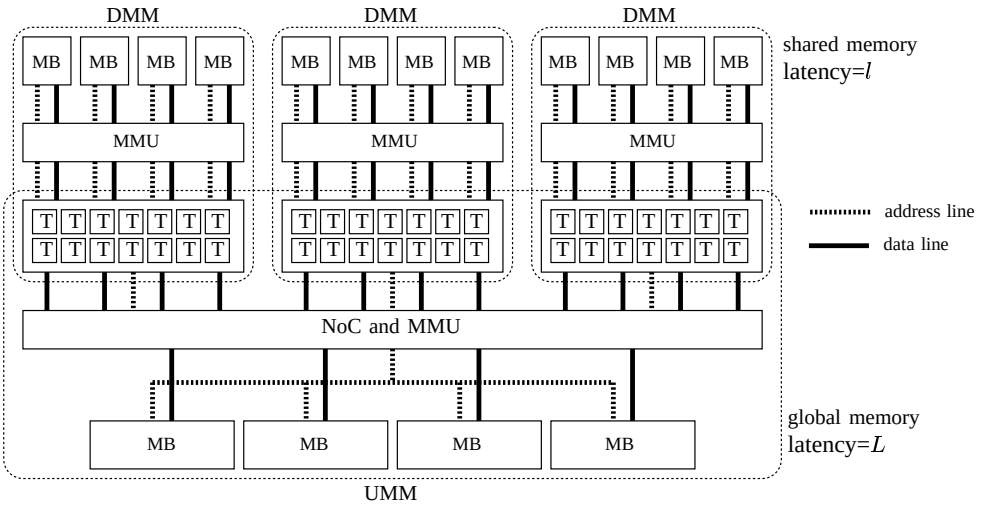

Figure 1. The architecture of the HMM with $d = 3$ DMMs and width $w = 4$

be computed in $O(mn)$ time [10] 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 [11]. Also, if the complicated bit operations of words is allowed, the ASM can be accelerated [12]. Utan *et. al* [13] implemented an approximate regular expression matching algorithm on the FPGA and the GPU. Quite recently, we have published an optimal algorithm for the ASM on the DMM and the UMM [14]. 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 [14] on the DMM and the UMM. It 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. We can prove that this implementation is time optimal in the sense that no other implementation can be faster. However, we omit the proof of the lower bound in this paper, due to the stringent page limitation. 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.

II. Memory Machine Models: the DMM, the UMM, and the HMM

We first define *the Discrete Memory Machine (DMM)* 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 $(i \bmod w)$-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 $T(0)$, $T(1)$, ..., $T(p - 1)$ are partitioned into $\frac{p}{w}$ warps $W(0), W(1), \ldots, W(\frac{p}{w}-1)$ such that $W(i) = \{T(i \cdot w), T(i \cdot w + 1), \ldots, T((i + 1) \cdot w - 1)\}$ ($0 \leq i \leq \frac{p}{w} - 1$). 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)* 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 2 shows examples of memory access on the DMM and the UMM. We assume that each memory access request is completed when it is dequeued from the pipeline. Two warps $W(0)$ and $W(1)$ access to $m[7], m[5], m[15], m[0]$ and $m[10], m[11], m[12], m[9]$, respectively. In the DMM, memory access requests by $W(0)$ are separated into two pipeline stages, because $m[7]$ and $m[15]$ 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, and 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 Figure 1. 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 is dispatched for memory access to the shared memory in turn. Further, each warp of w threads in all DMMs can send memory access requests to the global memory. Figure 1 illustrates the architecture of the HMM with $d = 3$ DMMs. Each DMM and the UMM has $w = 4$ memory banks.

III. Coalesced and conflict-free memory access

This section evaluates 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 all threads perform 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. We also say that a round of the memory access by 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 $n \geq p \geq w$.

IV. Approximate string matching

The main purpose of this section is to define the approximate string matching (ASM).

As a preliminary, we first define the edit distance (ED) of two strings [15]. Suppose that source string $X = x_1 x_2 \cdots x_m$ of length m and destination string $Y = y_1 y_2 \cdots y_n$ 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: (1) insertion of a character, (2) deletion of a character, and (3) replacement of a character. *The ED of two strings* is the minimum number of operations to change one string to the other. For later reference, let $\mathrm{ED}(X, Y)$ 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 $\mathrm{ASM}(X, Y)$ defined as follows:

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

Clearly, $\mathrm{ASM}(X, Y)$ is small if Y has a substring similar to X.

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

$$\min_{1 \leq j' \leq j} \mathrm{ED}(x_1 x_2 \cdots x_i, y_{j'} y_{j'+1} \cdots y_j).$$

Note that $x_1 x_2 \cdots x_i$ is a null string (i.e. string with length 0) if $i = 0$. Once all values of d is computed, we can compute

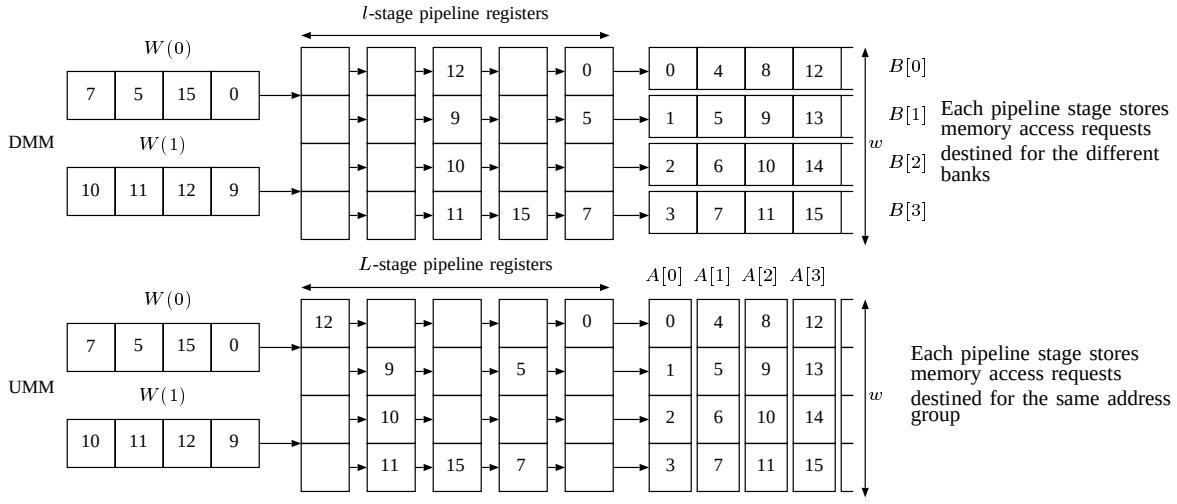

Figure 2. Examples of memory access on the DMM and the UMM

the value of $\mathrm{ASM}(X,Y)$ by the following formula:

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

All values of d and $\mathrm{ASM}(X,Y)$ can be computed by the following parallel algorithm. The key idea is to compute the values of the matrix d from the top-left corner to the bottom-right corner as illustrated in Figure 3. 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 details of the parallel algorithm is spelled out as follows:

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

Please see [14] for the details of this parallel ASM algorithm.

Clearly, when the values of d 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 d 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 d 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 $d[i][j]$

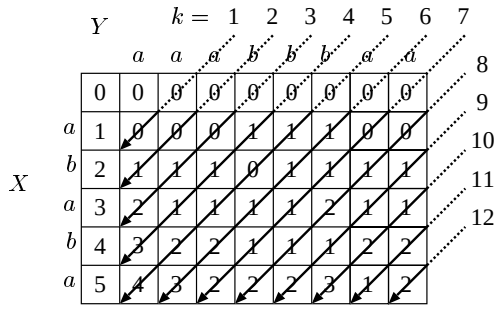

Figure 3. Illustrating a parallel algorithm for computing matrix d

$(0 \le i \le m, 0 \le j \le n)$ are stored in $e[j \bmod 3][i]$. The ASM can also be computed using array e as follows:

[**Improved parallel ASM algorithm**]
$minval \leftarrow m$
for $k \leftarrow 0$ 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 $e[j \bmod 3][m] < minval$ then $minval \leftarrow d[j \bmod 3][m]$
 end
output *minval*

Let us evaluate the computing time using p ($\ge w$) threads

on the DMM. The for-loop for a fixed k involves the following memory access operations:

- reading from $e[k_1][0], e[k_2][1], \ldots, e[k_m][m-1]$,
- reading from $e[k_0][1], e[k_1][2], \ldots, e[k_{m-1}][m]$,
- reading from $e[k_1][1], e[k_2][2], \ldots, e[k_m][m]$, and
- writing in $e[k_0][0], e[k_1][1], \ldots, e[k_{m-1}][m-1]$.

where $k_i = (k-i) \bmod 3$ $(0 \leq i \leq m)$. For simplicity, we consider that the memory access is omitted if the index of arrays above is out of range. It should be clear that each of these memory access operations is conflict-free. Thus, each of them can be done in $O(\frac{m}{w}+\frac{ml}{p})$ time units using p threads on the DMM from Lemma 1. 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.

V. 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 Y 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 $p \geq wd$.

The idea is to partition Y into d substrings and to compute the ASM of X and every substring in parallel. Let $s = \frac{n-2m}{d}$. For simplicity, we assume that s is an integer. We partition Y into d substrings such that $Y_i = y_{is}y_{is+1}\cdots y_{(i+1)s+2m-1}$ for all i $(0 \leq i \leq d-1)$. Clearly, every Y_i $(0 \leq i \leq d-1)$ has $s+2m$ characters. Also, $Y_i \cap Y_{i+1} = y_{(i+1)s}y_{(i+1)s+1}\cdots y_{(i+1)s+2m-1}$ and thus $Y_i \cap Y_{i+1}$ has $2m$ characters. Hence, any substring of length at most $2m$ is included in one of Y_i's, and we have $\mathrm{ASM}(X,Y) = \min\{\mathrm{ASM}(X,Y_i) \mid 0 \leq i \leq d-1\}$.

The idea of parallel processing is to compute each $\mathrm{ASM}(X,Y_i)$ $(0 \leq i \leq d-1)$ by a DMM as follows:

[Parallel ASM algorithm on the HMM]

Step 1: Each $\mathrm{DMM}(i)$ reads X and Y_i from the global memory and writes them in the shared memory.

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

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

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

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 are 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 $\mathrm{ASM}(X,Y_i)$ 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 $\mathrm{DMM}(i)$ writes the value of $\mathrm{ASM}(X,Y_i)$ 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 $\mathrm{ASM}(X,Y_i)$ in $O(\frac{d}{w}+\frac{dL}{p}+L)$ time units using p threads on the UMM using the algorithm in [16], [17]. 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 $(wd \leq p < md)$ 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. If m is too small, the latency overhead $O(\frac{(n+md)L}{p}+\frac{m(n+md)l}{p})$ may be dominant. We need to use more threads to hide this latency overhead. For this purpose, we partition Y into more substrings. Suppose that we partition Y into D $(\geq d)$ substrings $Y_0, Y_1, \ldots, Y_{D-1}$ such that each substring Y_i $(0 \leq i \leq D-1)$ has $S+2m$ characters, where $S = \frac{n-2m}{D}$. We use m threads to compute each $\mathrm{ASM}(X,Y_i)$. More specifically, $p = mD$ threads are arranged in d DMMs, and each DMM computes $\frac{D}{d}$ $\mathrm{ASM}(X,Y_i)$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 $(wd \leq p < mn)$ 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 $n \geq p$.

VI. EXPERIMENTAL RESULTS

We have implemented our parallel ASM algorithm for the HMM on the GPU and the sequential ASM algorithm on a single CPU. Table I shows the experimental results

Table I
THE RUNNING TIME (MILLISECONDS) OF PARALLEL ASM ALGORITHM ON THE HMM FOR $|Y| = 4M$ $(= 2^{22})$

$\|X\| = m$	GPU CUDA blocks D								CPU	speed-up
	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

on GeForce GTX-580 GPU and Intel Xeon CPU X7460 (2.66GHz). GeForce GTX-580 GPU has 16 streaming multiprocessors. The size w of a warp is 32. The table 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 $\mathrm{ASM}(X, Y_i)$ $(0 \leq i \leq D-1)$. 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.

REFERENCES

[1] W. W. Hwu, *GPU Computing Gems Emerald Edition*. Morgan Kaufmann, 2011.

[2] D. Man, K. Uda, Y. Ito, and K. Nakano, "A GPU implementation of computing euclidean distance map with efficient memory access," in *Proc. of International Conference on Networking and Computing*. IEEE CS Press, Dec. 2011, pp. 68–76.

[3] A. Uchida, Y. Ito, and K. Nakano, "Fast and accurate template matching using pixel rearrangement on the GPU," in *Proc. of International Conference on Networking and Computing*. IEEE CS Press, Dec. 2011, pp. 153–159.

[4] NVIDIA Corporation, "NVIDIA CUDA C programming guide version 5.0," 2012.

[5] 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," *International Journal of Networking and Computing*, vol. 1, no. 2, pp. 260–276, July 2011.

[6] NVIDIA Corporation, "NVIDIA CUDA C best practice guide version 3.1," 2010.

[7] K. Nishida, Y. Ito, and K. Nakano, "Accelerating the dynamic programming for the optial poygon triangulation on the GPU," in *Proc. of International Conference on Algorithms and Architectures for Parallel Processing (ICA3PP, LNCS 7439)*. IEEE CS Press, Sept. 2012, pp. 1–15.

[8] K. Nakano, "Simple memory machine models for GPUs," in *Proc. of International Parallel and Distributed Processing Symposium Workshops*. IEEE CS Press, May 2012, pp. 788–797.

[9] ——, "The hierarchical memory machine model for GPUs," in *Proc. of International Parallel and Distributed Processing Symposium Workshops*, May 2013, pp. 591–600.

[10] P. H. Sellers, "The theory and computation of evolutionary distances: Pattern recognition," *Journal of Algorithms*, vol. 1, no. 4, pp. 359–373, December 1980.

[11] E. Ukkonen, "Algorithms for approximate string matching," *Information and Control*, vol. 64, no. 1–3, pp. 100–118, January–March 1985.

[12] G. Myers, "A fast bit-vector algorithm for approximate string matching based on dynamic programming," *Journal of the ACM*, vol. 46, no. 3, pp. 395 – 415, May 1999.

[13] 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," in *Proc. Int. Conf. Parallel and Distributed Processing Techniques and Applications*, July 2012.

[14] K. Nakano, "Efficient implementations of the approximate string matching on the memory machine models," in *Proc. of International Conference on Networking and Computing*, Dec. 2012, pp. 233–239.

[15] T. H. Cormen, C. E. Leiserson, and R. L. Rivest, *Introduction to Algorithms*. MIT Press, 1990.

[16] K. Nakano, "An optimal parallel prefix-sums algorithm on the memory machine models for GPUs," in *Proc. of International Conference on Algorithms and Architectures for Parallel Processing (ICA3PP, LNCS 7439)*. Springer, Sept. 2012, pp. 99–113.

[17] ——, "Asynchronous memory machine models with barrier syncronization," in *Proc. of International Conference on Networking and Computing*, Dec. 2012, pp. 58–67.