Efficient Implementations of the Approximate String Matching on the Memory Machine Models

Koji Nakano
Department of Information Engineering
Hiroshima University
Kagamiyama 1-4-1, Higashi Hiroshima, 739-8527 Japan
Email: nakano@cs.hiroshima-u.ac.jp

***Abstract*—The Discrete Memory Machine (DMM) and the Unified Memory Machine (UMM) are theoretical parallel computing models that capture the essence of the shared memory access and the global memory access of GPUs. The approximate string matching for two strings X and Y is a task to find a substring of Y most similar to X. The main contribution of this paper is to show efficient implementations of approximate string matching on the memory machine models. Our best implementation for strings X and Y with length m and n ($m \leq n$), respectively, runs in $O(\frac{mn}{w} + ml)$ time units using n threads both on the DMM on the UMM with width w and latency l.**

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

I. Introduction

The GPU (Graphical 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.

CUDA uses two types of memories in the NVIDIA GPUs: *the shared memory* and *the global memory* [4]. The shared memory is an extremely fast on-chip memory with lower capacity, say, 16-64 Kbytes. The global memory is implemented as an off-chip DRAM, and has large capacity, say, 1.5-6 Gbytes, but its access latency is very long. 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 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 [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. The outline of the architectures of the DMM and the UMM is illustrated in Figure 1. In both architectures, *a sea of threads (Ts)* are connected to *the memory banks (MBs)* through *the memory management unit (MMU)*. Each thread is a Random Access Machine (RAM) [9], which can execute fundamental operations in a time unit. We do not discuss the architecture of the sea of threads in this paper, but we can imagine that it consists of a set of multi-core processors which can execute many threads in parallel. Threads are executed in SIMD [10] 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. Since the memory access of the UMM is more restricted than that of the DMM, the UMM is less powerful than the DMM.

The performance of algorithms of the PRAM is usually

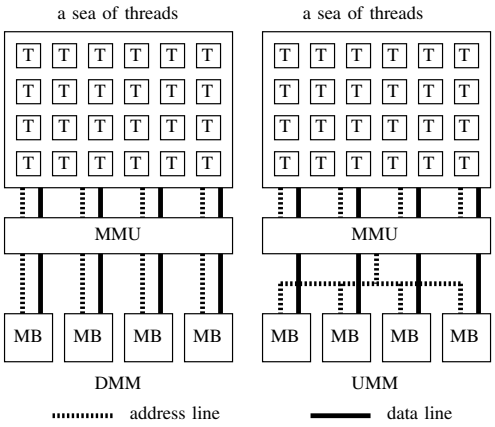

Figure 1. The architectures of the DMM and the UMM

evaluated using two parameters: the size n of the input and the number p of processors. For example, it is well known that the sum of n numbers can be computed in $O(\frac{n}{p}+\log n)$ time on the PRAM [11]. We will use four parameters, the size n of the input, the number p of threads, the width w and the latency l of the memory when we evaluate the performance of algorithms on the DMM and on the UMM. The width w is the number of memory banks and the latency l is the number of time units to complete the memory access. Hence, the performance of algorithms on the DMM and the UMM is evaluated as a function of n (the size of a problem), p (the number of threads), w (the width of a memory), and l (the latency of a memory). In NVIDIA GPUs, the width w of global and shared memory is 16 or 32. Also, the latency l of the global memory is several hundred clock cycles. In CUDA, a grid can have at most 65535 blocks with at most 1024 threads each [4].

Suppose that two strings X and Y with length m and n ($m \leq n$), respectively, are given. The approximate string matching is a task to find a substring Y' 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 approximate string matching has a lot of applications in the areas of signal processing, bio-informatics, natural language processing, among others. It is well known that the approximate string matching can be computed in $O(mn)$ time [12] using the dynamic programming technique. Many researchers have been devoted to do research on variations of the approximate string matching. For example, if the problem is to list substrings in Y with similarity no more than k, the computing time can be reduced [13]. Also, if the complicated bit operations of words is allowed, the approximate string matching can be accelerated [14].

Although a lot of work of sequential algorithms for the approximate string matching have been published, there is no significant work for parallel algorithms for approximate string matching. Since the computation of the approximate string matching involves long sequential operations, it is very hard to parallelize it to run in poly-logarithmic time. Also, it 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 approximate string matching has been shown in [15]. However, this string matching is a task to find substrings with Hamming distance no more than k, which is much simpler than our approximate string matching.

The main contribution of this paper is to show optimal implementation of approximate string matching algorithm shown in [12] on the memory machine models. We first show that the approximate string matching for X and Y with length m and n ($m \leq n$), respectively, can be computed in $O(\frac{mn}{w}+nl)$ time units using m threads on the DMM. We then go on to presents the matrix sliding technique, which allows us to perform the approximate string matching on the UMM in the same computing time.

From the practical point of view, n can be very large while m is small. Also, the latency l of current GPUs is several hundred. Thus, the factor $O(nl)$ in the latency overhead in the computation of the approximate string matching dominates the computing time. Hence, we show that the latency overhead can be reduced to $O(ml)$. In other words, we show that the approximate string matching for X and Y with length m and n ($m \leq n$) respectively can be computed in $O(\frac{mn}{w}+ml)$ time units using n threads both on the DMM and the UMM.

This paper is organized as follows. In Section II, we review the memory machine models presented in our previous paper [8], that capture the essence of the shared memory access and the global memory access of GPUs. Next, we evaluate the performance of the contiguous memory access on the memory machine models in Section III. Section IV defines the edit distance (ED) and the approximate string matching (ASM) of two strings, and shows their conventional sequential algorithms. We then show parallel algorithms for the DMM and the UMM in Sections V and VI, respectively. Finally, in Section VII, we show a better algorithm on the UMM that optimizes the latency overhead. Section VIII concludes our work.

II. Parallel Memory Machines: DMM and UMM

The main purpose of this section is to define the Discrete Memory Machine (DMM) and the Unified Memory Machine (UMM) introduced in our previous paper [8]. The reader should refer to [8], [16] for the details of the DMM and the UMM.

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. However, we assume that multiple memory access requests destined for the same address in the same bank have no extra overhead. For example, if two or more threads read from the same address, it can be read at the same time. Also, if two or more threads write in the same address, one of them is arbitrary selected and succeeds in writing.

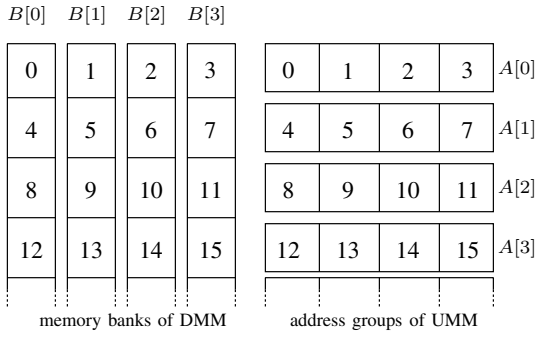

Figure 2. Banks and address groups for $w = 4$

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 $W(0), W(1)$, ..., $W(\frac{p}{w}-1)$ 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 and is skipped. When $W(i)$ is dispatched, w threads in $W(i)$ sends 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* for short) of width w 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.

III. Contiguous memory access and contiguous bank access

The main purpose of this section is to review the contiguous memory access on the DMM and the UMM shown in [8], [16]. We also show the contiguous bank access for the DMM.

The contiguous memory access is a key technique for accelerating the computation. Suppose that an array a of size n $(\geq p)$ is given. We use p threads to access all of n memory cells in a such that each thread accesses $\frac{n}{p}$ memory cells. Note that "accessing" can be "reading from" or "writing in." Let $a[i]$ $(0 \leq i \leq n-1)$ denote the i-th memory cells in a. When $n \geq p$, *the contiguous access* can be performed as follows:

[Contiguous memory access]
for $t \leftarrow 0$ to $\frac{n}{p} - 1$ do
  for $i \leftarrow 0$ to $p - 1$ do in parallel
    $T(i)$ access $a[p \cdot t + i]$

Let us evaluate the computing time. First, we assume that $w \leq p$. For each t $(0 \leq t \leq \frac{n}{p} - 1)$, p threads access p memory cells $a[pt], a[pt+1], \ldots, a[p(t+1)-1]$. This memory access is performed by $\frac{p}{w}$ warps in turn. More specifically, first, w threads in $W(0)$ access $a[pt], a[pt+1], \ldots, a[pt+w-1]$. After that, p threads in $W(1)$ access $a[pt+w], a[pt+w+1], \ldots, a[pt+2w-1]$, and the same operation is repeatedly performed. In general, p threads in $W(j)$ $(0 \leq j \leq \frac{p}{w} - 1)$ access $a[pt+jw], a[pt+jw+1], \ldots, a[pt+(j+1)w-1]$. Since w memory cells accessed by a warp are in different banks, the access can be completed in l time units on the DMM. Also, these w memory cells are in the same address group, and thus, the access can be completed in l time units on the UMM. Recall that the memory access are processed in pipeline fashion such that w threads in each $W(j)$ send w memory access requests in one time unit. Hence, p threads in $\frac{p}{w}$ warps send p memory access requests in $\frac{p}{w}$ time units. After that, the last memory access requests by $W(\frac{p}{w} - 1)$ are completed in $l - 1$ time units. Thus, p threads access p memory cells $a[pt], a[pt+1], \ldots, a[p(t+1)-1]$ in $\frac{p}{w} + l - 1$ time units. Since this memory access is repeated $\frac{n}{p}$ times, the contiguous access can be done in $\frac{n}{p} \cdot (\frac{p}{w} + l - 1) = O(\frac{n}{w} + \frac{nl}{p})$ time units.

Next, let us consider the case that $p \leq w$. If this is the case p threads is in a single warp. This warp performs memory access $\frac{n}{p}$ times each of which takes l time units. Thus, the contiguous access can be done in $O(\frac{nl}{p})$ time.

Therefore, we have,

Lemma 1: The contiguous memory access to an array of size n can be done in $O(\frac{n}{w} + \frac{nl}{p})$ time using p threads $(p \leq n)$ on the DMM and the UMM with width w and latency l.

For later reference, we also define *the contiguous bank access*, which takes in the same time units as the contiguous access on the DMM. Let $b_0, b_1, \ldots b_{n-1}$ be a sequence of integers such that $b_i \bmod w = i \bmod w$ for all i ($0 \leq i \leq n-1$). Hence, each $a[b_i]$ is in bank $i \bmod w$. The contiguous bank access is spelled out as follows:

[Contiguous bank access]
for $t \leftarrow 0$ to $\frac{n}{p} - 1$ do
 for $i \leftarrow 0$ to $p-1$ do in parallel
  $T(i)$ access $a[b_{p \cdot t+i}]$

If $p \geq w$ then p threads are partitioned into $\frac{p}{w}$ warps. In the contiguous bank access, w threads in each warp access the different banks. Hence, p threads access p memory cells $a[b_{pt}], a[b_{pt+1}], \ldots, a[b_{p(t+1)-1}]$ in $\frac{p}{w} + l - 1$ time units. Thus, the contiguous bank access can be done in $\frac{n}{p} \cdot (\frac{p}{w} + l - 1) = O(\frac{n}{w} + \frac{nl}{p})$ time units. If $p \leq w$ then the contiguous bank access takes $O(\frac{nl}{p})$ time units similarly to the contiguous memory access. Hence, we have,

Lemma 2: The contiguous access to an array of size n can be done in $O(\frac{n}{w} + \frac{nl}{p})$ time using p threads ($p \leq n$) on the DMM with width w and latency l.
Note that, the UMM takes a lot of time for the contiguous bank access, because threads in warps may access different address groups.

IV. 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 [12], [17] for the details.

As a preliminary, we first define the edit distance (ED) of two strings. 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:

- 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: $ababa \overset{delete}{\rightarrow} aaba \overset{delete}{\rightarrow} aaa \overset{insert}{\rightarrow} aaab \overset{insert}{\rightarrow} aaabb \overset{insert}{\rightarrow} aaabbb$. Alternatively, X can be changed into Y in three operations as follows: $ababa \overset{replace}{\rightarrow} aaaba \overset{replace}{\rightarrow} aaabb \overset{insert}{\rightarrow} aaabbb$. *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 $\mathrm{ED}(X, Y)$ denote the edit distance 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. Usually, the approximate string matching should require algorithms to return the indexes i and j such that $\mathrm{ASM}(X, Y) = \mathrm{ED}(X, y_i y_{i+1} \cdots y_j)$. Due to the stringent page limitation, we assume that an approximate string matching algorithm just computes the values of $\mathrm{ASM}(X, Y)$ in this paper, and is not required to output such i and j. Actually, once the values of $\mathrm{ASM}(X, Y)$ is obtained, it is not difficult to compute such i and j.

Let m and n be the length of X and Y, and assume that $m \leq n$. It should be clear that $\mathrm{ASM}(X, Y)$ is always less than or equal to m, and $\mathrm{ED}(X, Y)$ takes a value between $n - m$ and $n + m$. For example, if X and Y shares no character, then $\mathrm{ED}(X, Y) = n + m$ and $\mathrm{ASM}(X, Y) = m$. Also, the prefix of Y is X then $\mathrm{ED}(X, Y) = n - m$ and $\mathrm{ASM}(X, Y) = 0$.

We use a matrix d of size $(m+1) \times (n+1)$. 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 the value of $\mathrm{ASM}(X, Y)$ the following formula:

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

Let us show how we compute all values of d. Suppose that $d[i-1][j-1]$, $d[i-1][j]$, and $d[i][j-1]$ 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 $d[i][j]$ can be computed as follows:

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

Let ϕ denote a null string or a string with length 0. We can confirm the correctness of the formula above as follows:
(1) $d[0][j] = 0$ from $\mathrm{ASM}(\phi, y_1 y_2 \cdots y_j) = 0$,
(2) $d[i][0] = i$ from $\mathrm{ASM}(x_1 x_2 \cdots x_i, \phi) = i$,
(3) $d[i][j] \leq d[i][j-1] + 1$ from $\mathrm{ASM}(x_1 \cdots x_i, y_1 \cdots y_j) \leq \mathrm{ASM}(x_1 \cdots x_i, y_1 \cdots y_{j-1}) + 1$,
(4) $d[i][j] \leq d[i-1][j] + 1$ from $\mathrm{ASM}(x_1 \cdots x_i, y_1 \cdots y_j) \leq \mathrm{ASM}(x_1 \cdots x_{i-1}, y_1 \cdots y_j) + 1$,
(5) if $x_i = y_j$, then $d[i][j] \leq d[i-1][j-1]$,

			Y							
			a	a	a	b	b	b	a	a
		0	0	0	0	0	0	0	0	0
	a	1	1	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	3	1	2

Figure 3. The values of matrix d for the ASM

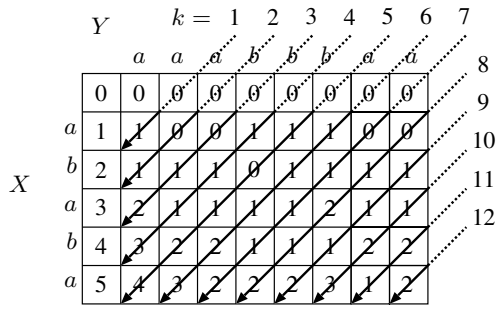

Figure 4. Illustrating a parallel algorithm for computing matrix d

and if $x_i \neq y_j$, then $d[i][j] \leq d[i-1][j-1]+1$ from $\mathrm{ASM}(x_1 \cdots x_i, y_1 \cdots y_j) \leq \mathrm{ASM}(x_1 \cdots x_{i-1}, y_1 \cdots y_{j-1}) + 1$.

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

[Sequential algorithm for the ASM]
for $j \leftarrow 0$ to n do $d[0][j] \leftarrow 0$
for $i \leftarrow 0$ to m do $d[i][0] \leftarrow i$
for $i \leftarrow 1$ to m do
 for $j \leftarrow 1$ to n do
 $d[i][j] \leftarrow \min(d[i][j-1]+1, d[i-1][j]+1,$
 $d[i-1][j-1] + (x_i \neq y_j)$
output $\min\{d[m][j] \mid 0 \leq j \leq n\}$

Figure 3 shows the values of d for two strings $X = ababa$ and $Y = aaabbbaa$. From the figure, we can see that the ASM of X and Y is 1.

V. A PARALLEL ALGORITHM ON THE DMM

The main purpose of this section is to show a parallel algorithm for computing the ASM.

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 4. The details of the parallel algorithm is spelled out as follows:

[Parallel algorithm for the ASM]
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 $1 \leq j \leq n$ 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 $\min\{d[m][j] \mid 0 \leq j \leq n\}$

In the third for-loop, for each k ($1 \leq k \leq n+m-1$), the values $d[1][k], d[2][k-1], \ldots d[m][k-m]$ are computed and stored. Clearly, when the values of d for k is computed, only those for $k-1$ and $k-2$ are used. Thus, this parallel algorithm correctly computes the ASM.

Let us evaluate the computing time on the DMM. We assume that $n+1$, the row size of matrix d, is a multiple of w. If this is not a case, we can choose the minimum integer n' exceeding $n+1$ such that n' is a multiple of w, and use a matrix d of size $m \times n'$. We also assume that we use m threads on the UMM.

The first for-loop performs "$d[0][j] \leftarrow 0$" in parallel. Since writing in n elements $d[0][1], d[0][2], \ldots, d[0][n]$ is contiguous memory access, it takes $O(\frac{n}{w} + \frac{nl}{m})$ time units from Lemma 1. The second for-loop performs "$d[i][0] \leftarrow i$" in parallel. Since $d[0][0], d[1][0], \ldots, d[m][0]$ are in the same bank $B(0)$, all writing operations are performed in turn. Hence, the second for-loop takes $m+l$ time units.

The evaluation of the computing time for the third for-loop is a little complicated. The third for-loop for a fixed k involves the following memory access operations:

- reading from $x[1], x[2], \ldots, x[m]$,
- reading from $y[k], y[k-1], \ldots, y[k-m+1]$,
- reading from $d[1][k-1], d[2][k-2], \ldots, d[m][k-m]$,
- reading from $d[0][k], d[1][k-1], \ldots, d[m-1][k-m+1]$,
- reading from $d[0][k-1], d[1][k-2], \ldots, d[m-1][k-m]$, and
- writing in $d[1][k], d[2][k-1], \ldots, d[m][k-m+1]$.

For simplicity, we consider that the memory access is omitted if the index of arrays above is out of range. Since the first two reading operations are the contiguous memory access for m elements, they take $O(\frac{m}{w} + \frac{ml}{m}) = O(\frac{m}{w} + l)$ time units from Lemma 1. The remaining four reading operations are not contiguous. However, they are contiguous bank access, and they also take $O(\frac{m}{w}+l)$ time units from Lemma 2. Thus, the third for-loop runs in $O(\frac{m}{w}+l)\cdot(n+m-1) = O(\frac{nm}{w}+nl)$ time from $m \leq n$.

Finally, we need to compute $\min\{d[m][j] \mid 0 \leq j \leq n\}$. The minimum of $n+1$ numbers can be computed using a single thread in $(n+1)l = O(nl)$ time in an obvious way.

Therefore, we have,

Lemma 3: The ASM of two sequences of length m and n ($m \leq n$) can be computed in $O(\frac{nm}{w}+nl)$ time units using

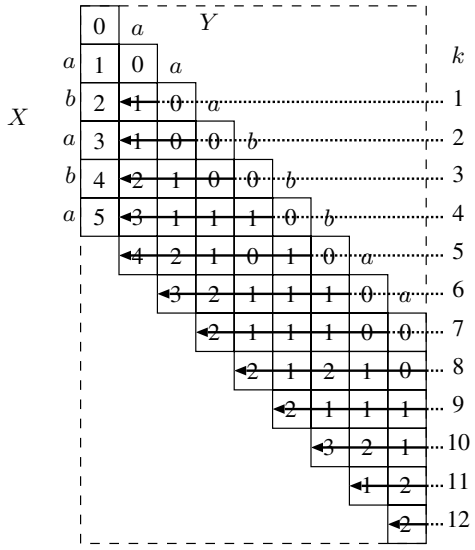

Figure 5. Illustrating a parallel algorithm for computing slided matrix s

m threads on the DMM with width w and latency l.

VI. A PARALLEL ALGORITHM ON THE UMM

This section is devoted to show how we implement the parallel algorithm for computing the approximate string matching on the UMM.

If the parallel algorithm in the previous section is implemented on the UMM as it is, the memory access is performed for different address groups. If this is the case, the memory access request is processed in turn, the UMM takes at least $O(nm)$ time units to compute the values of d.

For the purpose of the contiguous memory access in the third for-loop, we slide the matrix d as illustrated in Figure 5. More specifically, we use a matrix s of size $(n+m+1)\times(n+1)$ such that the value of each $d[i][j]$ is stored in $s[i+j][j]$.

Let us evaluate the computing time on the UMM if the parallel algorithm is executed for the matrix s. Again, we assume that we use m threads. The first for-loop and the second for-loop performs writing operations for $s[0][0], s[1][1], \ldots, s[n][n]$ and for $s[1][0], s[2][0], \ldots, s[m][0]$, respectively. Hence, the memory access by a warp is performed for the different address groups both in the first for-loop and in the second for-loop. Thus, the first for-loop and the second for-loop take $m + l$ time units and $n + l$ time units, respectively.

The third for-loop for a fixed k involves the following memory access operations:

- reading from $x[1], x[2], \ldots, x[m]$,
- reading from $y[k], y[k-1], \ldots, y[k-m+1]$,
- reading from $s[k][k-1], s[k][k-2], \ldots, s[k][k-m]$,
- reading from $s[k][k], s[k][k-1], \ldots, s[k][k-m+1]$,
- reading from $s[k-1][k-1], s[k-1][k-2], \ldots, s[k-1][k-m]$, and
- writing in $s[k+1][k], s[k+1][k-1], \ldots, s[k+1][k-m+1]$.

Since the first two reading operations are the contiguous access, they take $O(\frac{m}{w} + \frac{ml}{m}) = O(\frac{m}{w} + l)$ time units from Lemma 1. Since the remaining four memory access operations are performed by the contiguous access, they also take $O(\frac{m}{w} + l)$ time units. Thus, the third for-loop runs in $O(\frac{m}{w} + l) \cdot (n + m) = O(\frac{nm}{w} + nl)$ time units. Finally, we need to compute $\min\{s[m + j][j] \mid 0 \leq j \leq n\}$, which can be done in $O(nl)$ time units. Therefore, we have,

Lemma 4: The ASM of two sequences of length m and n ($m \leq n$) can be computed in $O(\frac{nm}{w} + nl)$ time units using m threads on the UMM with width w and latency l.

Clearly, the implementation for Lemma 4 also works on the DMM.

The implementation for Lemma 4 uses an array s of size $(n + m + 1) \times (n + 1)$. However, only three rows $s[k - 1][*], s[k][*], s[k+1][*]$ of s are accessed for each k. Further, $m + 2$ elements are used in each row, and two of them are just initialized values. Thus, we can reduce the size of array s to $3 \times m$. For this purpose, we use array s' of size $3 \times m$ to handle the values in s as follows:

- return 0 when $s[j][j]$ is read,
- return i when $s[i][0]$ is read, and
- access $s'[i \bmod 3][j \bmod m]$ when $s[i][j]$ is accessed.

Thus, we have,

Theorem 5: The ASM of two sequences of length m and n ($m \leq n$) can be computed in $O(\frac{nm}{w} + nl)$ time units using m threads and working space of size $3m$ on the DMM and the UMM with width w and latency l.

VII. FURTHER IMPROVEMENT OF THE PARALLEL ASM

In practical applications, $\mathrm{ASM}(X, Y)$ for a very long Y need to be computed. In other words, we need to compute $\mathrm{ASM}(X, Y)$ such that $m \ll n$. If this is the case, the latency overhead $O(nl)$ dominates the computing time. The main purpose of this section is to show that the latency overhead can be reduced to $O(ml)$. This hides the latency overhead and achieves a significant improvement when $m \ll n$.

We first show that, we can restrict the length of substrings of Y to at most $2m$ when we compute the ASM. If a substring Y' of Y is longer than $2m$, then $\mathrm{ED}(X, Y') > 2m - m = m$. However, since $\mathrm{ASM}(X, Y) \leq m$ always hold, we can ignore such Y' when we compute $\mathrm{ASM}(X, Y)$.

Using this fact, we can further parallelize the computation of $\mathrm{ASM}(X, Y)$. We partition Y into $\frac{n}{m}$ substrings $Y_0, Y_1, \ldots, Y_{\frac{n}{m}-1}$ of length m each such that $Y_i = y_{im+1}y_{im+2}\cdots y_{(i+1)m}$ ($0 \leq i \leq \frac{n}{m} - 1$). Further, let $Z_0, Z_1, \ldots, Z_{\frac{n}{m}-3}$ be the sequences of adjacent three Y_i's, that is, $Z_i = Y_iY_{i+1}Y_{i+2}$ ($0 \leq i \leq \frac{n}{m} - 3$). Clearly, the length of each Z_i is $3m$ and any substring Y' with at most

$2m$ characters is a substring of one of $Z_0, Z_1, \ldots, Z_{\frac{n}{m}-3}$. Thus, the following algorithm can compute $\mathrm{ASM}(X, Y)$.

[Improved parallel algorithm for the ASM]
for $i \leftarrow 0$ to $\frac{n}{m} - 3$ do in parallel
 compute $\mathrm{ASM}(X, Z_i)$
output $\min\{\mathrm{ASM}(X, Z_i) \mid 0 \leq i \leq \frac{n}{m} - 3\}$

Let us evaluate the computing time. We use the implementation for Theorem 5 to compute each $\mathrm{ASM}(X, Z_i)$ $(0 \leq i \leq \frac{n}{m} - 3)$ in parallel. In this implementation, a matrix s' of size $3 \times m$ is used to compute each $\mathrm{ASM}(X, Z_i)$. Since $m \cdot (\frac{n}{m} - 3) < n$, we can consider that we have a large combined matrix of size $3 \times n$ and the contiguous memory access is performed using n threads. From Lemma 1, the contiguous memory access to an array of $3n$ can be done using n threads in $O(\frac{3n}{w} + \frac{3nl}{n}) = O(\frac{n}{w} + l)$ time units. Since X and Z_i have m and $3m$ characters, respectively, this contiguous memory access is repeated $m+3m-1 = 4m-1$ times. Thus, the values of all $\mathrm{ASM}(X, Z_i)$ $(0 \leq i \leq \frac{n}{m} - 3)$ can be computed in $O(\frac{n}{w} + l) \cdot (4m - 1) = O(\frac{nm}{w} + ml)$ time using n threads.

Next, we need to compute the minimum of $\frac{n}{m} - 2$ numbers. We can compute the minimum in $(\frac{n}{m} - 2)l = O(\frac{nl}{m})$ time units using a single thread. However, when n is very large, this computing time dominates the total computing time. We want to compute the minimum of $\frac{n}{m} - 2$ numbers in no more than $O(\frac{nm}{w} + ml)$ time. We use the fact that these numbers are no more than m. Let $s_0, s_1, \ldots, s_{\frac{n}{m}-3}$ denote $\frac{n}{m} - 2$ numbers. We use $\frac{n}{m} - 2$ threads and an array v of size $m + 1$ initialized by 0. A thread assigned to each s_i $(0 \leq i < \frac{n}{m} - 2)$ performs $v[s_i] \leftarrow 1$. Each of the $\frac{\frac{n}{m}-2}{w}$ warp accesses an array of size $m + 1$, this operation takes $(\frac{\frac{n}{m}-2}{w} + l) \cdot \frac{m+1}{w} = O(\frac{n}{w^2} + \frac{ml}{w})$ time units. After that, we find the minimum j such that $v[j] = 1$. Such minimum j, which is equal to the minimum of $\frac{n}{m} - 2$ numbers, can be found in $(m + 1)l = O(ml)$ time using a single thread in an obvious way. Thus, the minimum of $\frac{n}{m} - 2$ numbers can be computed in $O(\frac{n}{mw} + \frac{ml}{w}) < O(\frac{nm}{w} + ml)$ time units. Finally, we have,

Theorem 6: The ASM of two sequences of length m and n $(m \leq n)$ can be computed in $O(\frac{nm}{w} + ml)$ time units using n threads and working space of size $3n$ on the DMM and the UMM with width w and latency l.

VIII. Conclusion

In this paper, we have presented efficient implementations for the approximate string matching on the memory machine models. Our best implementation for strings X and Y with length m and n $(m \leq n)$, respectively, runs in $O(\frac{mn}{w} + ml)$ time units using n threads both on the DMM on the UMM with width w and latency l.

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*, 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*, Dec. 2011, pp. 153–159.

[4] NVIDIA Corporation, "NVIDIA CUDA C programming guide version 4.0," 2011.

[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, 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)*, Sept. 2012, pp. 1–15.

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

[9] A. V. Aho, J. D. Ullman, and J. E. Hopcroft, *Data Structures and Algorithms*. Addison Wesley, 1983.

[10] M. J. Flynn, "Some computer organizations and their effectiveness," *IEEE Transactions on Computers*, vol. C-21, pp. 948–960, 1972.

[11] A. Gibbons and W. Rytter, *Efficient Parallel Algorithms*. Cambridge University Press, 1988.

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

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

[14] 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.

[15] Y. Liu, L. Guo, J. Li, M. Ren, and K. Li, "Parallel algorithms for approximate string matching with k mismatches on cuda," in *Proc. of International Parallel and Distributed Processing Symposium Workshops*, May 2012, pp. 2414 –2422.

[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)*, Sept. 2012, pp. 99–113.

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