A Time Optimal Parallel Algorithm for the Dynamic Programming on the Hierarchical Memory Machine

Koji Nakano
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 architecture of CUDA-enabled GPUs. The main contribution of this paper is to present an efficient implementation of the $O(n^3)$-time dynamic programming algorithm for solving the optimal triangulation problem for a convex n-gon in the HMM. Although the HMM can run a lot of threads in parallel, it is very hard to accelerate computation involving complicated memory access such as the dynamic programming for the optimal triangulation problem. It is often the case that the acceleration rate is limited to the bandwidth w of the global memory for problems involving complicated stride memory access. Quite surprisingly, our implementation of the dynamic programing algorithm for solving the optimal triangulation problem runs $O(\frac{n^3}{w^2})$ time units using $\max(wL, w^2l)$ threads on the HMM with bandwidth w, global memory latency L and shared memory latency l. Hence, this parallel algorithm achieves the acceleration rate of more than w although the dynamic programming algorithm involves complicated stride memory access. Also, we prove that this parallel algorithm is time optimal when $L = O(wl)$.**

***Keywords*-Dynamic programming, parallel algorithms, memory machine models, 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], [4], [5], [6], [7]. NVIDIA provides a parallel computing architecture called *CUDA* (Compute Unified Device Architecture) [8], 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 [9], since they have thousands 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* [8]. The shared memory is an extremely fast on-chip memory with lower capacity, say, 16-48 Kbytes. The global memory is implemented as an off-chip DRAM, and thus, it 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 [6], [9], [10], [11]. 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 papers [12], [13], we have introduced three models, *the Discrete Memory Machine (DMM)*, *the Unified Memory Machine (UMM)*, and *the Hierarchical Memory Machine (HMM)* which reflect the essential features of computation performed by CUDA-enabled GPUs. The DMM is a theoretical parallel computing model of a streaming multiprocessor in a CUDA-enabled GPU. It has a shared memory with w memory banks, and a warp of w threads can access the shared memory. The UMM is a model for the parallel computation using the global memory of a CUDA-enabled GPU. It also has a global memory with w memory banks. The difference of the shared memory of the DMM and the global memory of the UMM is the restriction of access to memory banks. The same address of memory banks of the global memory must be accessed in each time unit, while different addresses of memory banks of the shared memory can be accessed. The HMM is a hybrid of the DMM and the UMM, which can be used to design and evaluate parallel algorithms using multiple streaming multiprocessors in a CUDA-enabled GPU. The HMM has multiple DMMs each of which has a shared memory. It also has a global memory which can be accessed by all threads in DMMs. The reader should refer to Figure 1 illustrating the HMM with width 4 and 3 DMMs. We use parameters w, L, and l to denote the number of memory banks, the global memory access latency, and the shared memory access latency. By using the HMM, we can give a theoretical analysis of performance of algorithms developed for CUDA-enabled GPUs using these parameters. Theoretical analysis of algorithms on the HMM approximates the performance of them on CUDA-enabled GPUs. For example, we have shown in [14] an offline permutation algorithm on the HMM and implemented it in GeForce GTX-680. The experimental results

978-1-4799-4152-0/14 $31.00 © 2014 IEEE
DOI 10.1109/CANDAR.2014.14

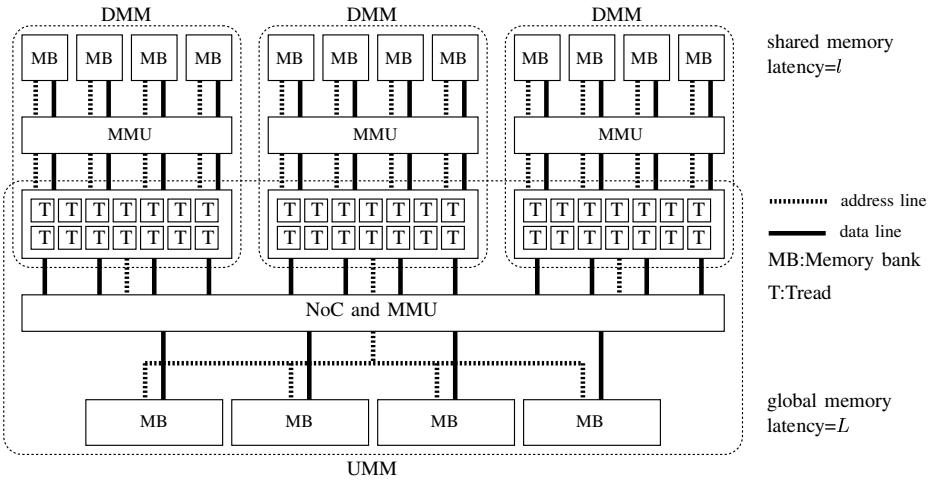

Fig. 1. The Hierarchical Memory Machine (HMM) with 3 DMMs.

showed that theoretical analysis approximates the performance on the GPU.

The dynamic programming is an important and sophisticated algorithmic technique to find an optimal solution of a problem over an exponential number of solution candidates [15]. A naive solution for such problem needs exponential time. The dynamic programming enables us to solve such problems in polynomial time. For example, the longest common subsequence problem, which requires finding the longest common subsequence of given two sequences, can be solved by the dynamic programming [16]. Since a sequence has an exponential number of subsequences, a straightforward algorithm takes an exponential time to find the longest common subsequence. However, it is known that this problem can be solved in $O(mn)$ time by the dynamic programming, where m and n are the lengths of two sequences. Many important problems including the edit distance problem, the matrix chain product problem, and the optimal polygon triangulation problem can be solved by the dynamic programming [15].

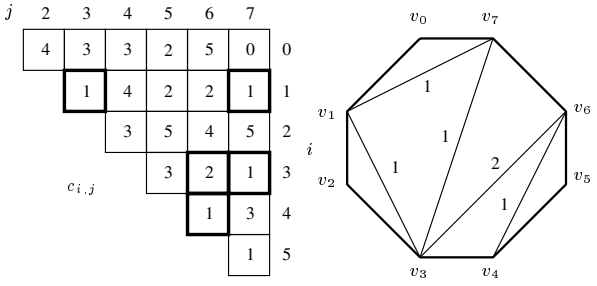

Fig. 2. An example of a triangulation of a convex 8-gon

The main contribution of this paper is to show an efficient implementation of the dynamic programming algorithm solving the optimal polygon triangulation problem (OPT) on the HMM. In the OPT problem, a convex n-gon with each chord being assigned a weight is given. The problem is to find a triangulation (i.e. a set of $n-3$ non-crossing chords) with minimum total weight. Figure 2 shows an example of a convex 8-gon with nodes $v_0, v_1, \ldots, v_7$ with each chord v_iv_j having weight $c_{i,j}$. It also shows the optimal triangulation with total weight 6. It is known that the OPT problem for a convex n-gon can be solved in $O(n^3)$ time using the dynamic programming technique [6], [15], [17], [18], [19]. Since a straightforward algorithm takes an exponential time, this problem is often used to introduce the dynamic programming technique. Although this algorithm is efficient, the memory access operation is complicated. To find an efficient parallel implementation of the dynamic programming is not easy. It is often the case that the acceleration rate is limited to the bandwidth w of the global memory for problems involves complicated stride memory access. Quite surprisingly, our implementation of the dynamic programing technique for solving the optimal triangulation problem achieves the acceleration rate of more than w. It runs $O(\frac{n^3}{w^2})$ time units using $\max(wL, w^2l)$ threads on the HMM. Since the sequential algorithm runs $O(n^3)$ time units, our implementation achieves a speed-up factor of w^2.

In our previous paper [19], we have shown that the OPT problem for a convex n-gon can be solved in $O(\frac{n^3}{w})$ using wL threads on the UMM with width w and latency L. Note that the UMM has the global memory with bandwidth w, but it does not have a shared memory. Hence, this implementation is optimal in the sense that n^3 memory access operations to the global memory with bandwidth w take at least $\Omega(\frac{n^3}{w})$ and it is not possible to solve the OPT in less than $O(\frac{n^3}{w})$ time units. Our new implementation running $O(\frac{n^3}{w^2})$ time units on the HMM implies that the bandwidth limitation of a factor of w can be broken even if the number of threads is still $\max(wL, w^2l)$. We also prove that this algorithm is time optimal whenever $L = O(w^2l)$.

There are several published works on the implementa-

tion of the dynamic programming [5], [6], [20], [21], [22]. Their implementations have been optimized mainly by the developer's experience. Hence, these implementations are very complicated and they have no concrete theoretical analysis of the performance. Although the experimental results have been presented, the optimality of the implementation has not been shown. Actually, it can be proved that some of the presented implementations are not optimal from the theoretical point of view. The performance of the implementation on the GPUs depends on a lot of factors, say, programmer's skill, compiler version and optimization option, GPU model numbers, host PC performance, etc. It is very hard to compare the experimental results and hence the theoretical analysis independent of them is very important.

The rest of this paper is organized as follows. Section II introduces three memory machines, the Discrete Memory Machine (DMM), the Unified Memory Machine (UMM), and the Hierarchical Memory Machine (HMM), which are theoretical parallel computing models for CUDA-enabled GPUs and shows several fundamental memory access operations. In Section III, we review the dynamic programing algorithm for solving the OPT problem in $O(n^3)$ time for a convex n-gon. Section IV shows an implementation of the dynamic programming algorithm on the DMM. The resulting algorithm runs $O(\frac{n^3}{w} + \frac{n^3 l}{p})$ time units using p ($w \leq p \leq n^2$) threads on the DMM with width w and latency l. Using this implementation, we show that the OPT problem can be solved in $O(\frac{n^3}{w^2} + \frac{n^2 L}{w})$ time units on the HMM using wL threads. The latency overhead $O(\frac{n^2 L}{w})$ is dominant if $n \leq wL$. Section VI reduces the latency overhead and shows that the OPT problem can be solved in $O(\frac{n^3}{w^2})$ on the HMM using $\max(wL, w^2 l)$ threads. It also proves the time optimality. Section VII concludes our work.

II. The DMM, the UMM, and the HMM

The main purpose of this section is define three memory machine models: the Discrete Memory Machine (DMM), the Unified Memory Machine (UMM), and the Hierarchical Memory Machine (HMM), which capture the essence of parallel computing on CUDA-enabled GPUs.

We first define *the Discrete Memory Machine (DMM)* [12], [23] 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 $W(0), W(1)$, ..., $W(\frac{p}{w}-1)$ 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)* [12], [19] of width w and latency L. 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 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*. All DMMs work in parallel. 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 1 illustrates the architecture of the HMM with $d = 3$ DMMs and $w = 4$. 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. Also, it makes sense to assume that the shared memory in each DMM can store up to $O(w^2)$ words of data because CUDA enabled-GPUs can store $12w^2$ words of data. The capacity of the shared memory in a streaming multiprocessor of CUDA enabled-GPUs is up to 48Kbytes [8].

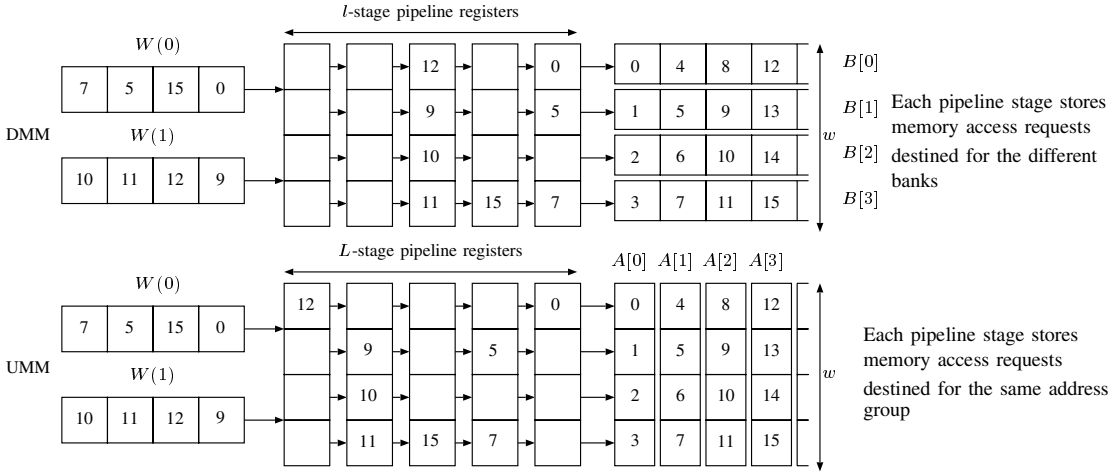

Fig. 3. Examples of memory access on the DMM and the UMM

Since the number of memory banks and the number of threads in a warp is $w = 32$, an array of w^2 32-bit integers occupies 4K bytes. Thus, a streaming multiprocessor cannot have more than 12 such arrays. We use l and L to denote the memory access latencies of the shared memory in a DMM and the global memory of the UMM. The memory access latency of the global memory of CUDA-enabled GPUs is several hundred clock cycles, while that of the shared memory of a streaming multiprocessor is several clock cycles [8]. Hence, it makes sense to assume that $l \ll L$.

For later reference, we will evaluate the time necessary to complete fundamental memory access operations. Suppose that p ($\geq w$) threads in $\frac{p}{w}$ warps in a DMM of the HMM access n elements in the shared memory, $\frac{n}{p}$ elements each in turn. If all memory access requests by w threads in all warps are conflict-free, the first p elements can be accessed in $\frac{p}{w} + l - 1$ time units. Since this access operation is repeated $\frac{n}{p}$ times, it takes $\frac{n}{p} \cdot (\frac{p}{w} + l - 1) = O(\frac{n}{w} + \frac{nl}{p})$ time units to access n elements. Hence, we have,

Lemma 1: If n memory access requests by p ($\geq w$) threads in a DMM with width w and latency l are conflict-free, then they can be completed in $O(\frac{n}{w} + \frac{nl}{p})$ time units.

Suppose that the HMM has d DMMs with p threads each and they copy n elements in the global memory to the shared memory of the DMMs such that each shared memory has $\frac{n}{d}$ elements. Each thread works for copying $\frac{n}{pd}$ elements and we assume that memory access to the global memory is coalesced and that to the shared memory is conflict-free. Hence, the first pd elements can be read from the global memory in $\frac{pd}{w} + L - 1$ time units. After that, they can be written in the shared memory in $\frac{p}{w} + l - 1$ time units. Thus, the first pd elements can be copied in $O(\frac{pd}{w} + L)$ time units. This operation is repeated $\frac{n}{pd}$ times, we have

Lemma 2: The task of copying n elements in the global memory to the shared memory of d DMMs with p threads each takes $O(\frac{n}{w} + \frac{nL}{pd})$ time units of the HMM.

Clearly, the task of copying n elements in the shared memory of d DMMs $\frac{n}{d}$ elements each to the global memory can be done in the same time units.

III. The Optimal Polygon Triangulation and the Dynamic Programming

This section defines the optimal polygon triangulation (OPT) problem and reviews an algorithm solving this problem by the dynamic programming technique [6], [15], [19].

Let $v_0, v_1, \ldots, v_{n-1}$ be vertices of a convex n-gon. Clearly, the convex n-gon can be divided into $n - 2$ triangles by a set of $n - 3$ non-crossing chords. We call a set of such $n - 3$ non-crossing chords *a triangulation*. Figure 2 shows an example of a triangulation of a convex 8-gon. The convex 8-gon is separated into 6 triangles by 5 non-crossing chords. Suppose that a weight $c_{i,j}$ of every chord $v_i v_j$ in a convex n-gon is given. The goal of the OPT problem is to find an optimal polygon triangulation that minimizes the total weight of selected chords for the triangulation. Actually, the corresponding optimal triangulation (i.e. a set of $n - 3$ non-crossing chords) can be obtained by a few extra bookkeeping steps to obtain the actual triangulation, using the data structure to compute the minimum total weight.

Let M be a matrix such that $M_{i,j}$

$$M_{i,j} = 0 \quad \text{if } j - i \leq 1, \tag{1}$$

$$M_{i,j} = \min_{i \leq k \leq j-1} (M_{i,k} + M_{k+1,j}) + c_{i-1,j} \quad \text{otherwise.} \tag{2}$$

By computing all $M_{i,j}$s, we can obtain the optimal triangulation. Please see [6], [19] to confirm that we can obtain the optimal triangulation by M. We say that the $n - r - 1$ elements $M_{1,r+1}, M_{2,r+2}, \ldots, M_{n-r-1,n-1}$ of M constitute *diagonal* r. Clearly, elements in diagonal r can be computed if all elements in diagonals 0, 1, ..., $r - 1$ are computed. Using this idea, the simple parallel algorithm, Algorithm DP-OPT computes all values in M in $n - 1$ stages. Each Stage r

$(0 \leq r \leq n-2)$ computes the values in diagonal r using the recursive formula for $M_{i,j}$. Since Stages 0, 1, ..., $r-1$ have computed elements in diagonal 0, 1, ..., $r-1$, this is possible. Figure 4 shows elements computed in each Stage r for 8-gon illustrated in Figure 2. The details of Algorithm DP-OPT is spelled out as follows:

[**Algorithm DP-OPT**]
for $i \leftarrow 1$ to $n-2$ do // Loop A
 for $j \leftarrow i+1$ to $n-1$ do
  $M_{i,j} \leftarrow +\infty$
for $i \leftarrow 1$ to $n-1$ do // Loop B (Stage 0)
 $M_{i,i} \leftarrow 0$
for $r \leftarrow 1$ to $n-2$ do //(Stage r)
 for $i \leftarrow 1$ to $n-r-1$ do // Loop C
  for $j \leftarrow i$ to $i+r-1$ do
   $M_{i,i+r} \leftarrow \min(M_{i,i+r}, M_{i,j} + M_{j+1,i+r})$
 for $i \leftarrow 1$ to $n-r-1$ do // Loop D
  $M_{i,i+r} \leftarrow M_{i,i+r} + c_{i-1,i+r}$

In Loop A of Algorithm DP-OPT, all elements in M are initialized by $+\infty$. Loop B corresponds to Stage 0, which stores 0 in all $M_{i,i}$ $(1 \leq i \leq n-1)$. Loop C computes $\min(M_{i,i+r}, M_{i,j} + M_{j+1,i+r})$ for all j and stores it in $M_{i,i+r}$. Figure 5 illustrates how $M_{i,i+r}$ is computed. Clearly, $M_{i,i+r} = \min_{i \leq j \leq i+r-1}(M_{i,j} + M_{j+1,i+r}) + c_{i-1,i+r}$ holds at the end of Stage r. Thus, Algorithm DP-OPT solves the OPT problem correctly.

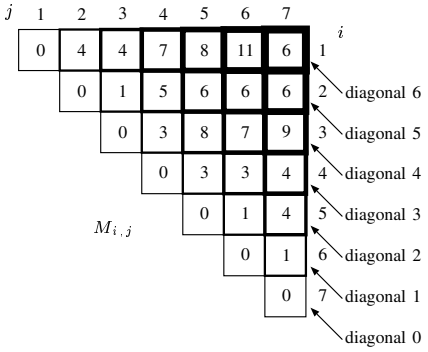

Fig. 4. The computation of $M_{i,i+r}$ and the resulting values of $M_{i,j}$

Let us evaluate the computing time. Each Stage r $(2 \leq r \leq n-2)$ performs

- $(n-r-1)$ $M_{i,j}$'s, $M_{1,r+1}, M_{2,r+2}, \ldots, M_{n-r-1,n-1}$ are computed, and
- the computation of each $M_{i,j}$ involves the computation of the minimum over r values, each of which is the sum of two $M_{i,j}$'s.

Thus, each Stage r takes

$$(n-r-1) \cdot O(r) \quad = \quad O(nr - r^2)$$

time. Therefore, Algorithm DP-OPT runs in

$$\sum_{0 \leq r \leq n-2} O(nr - r^2) \quad = \quad O(n^3)$$

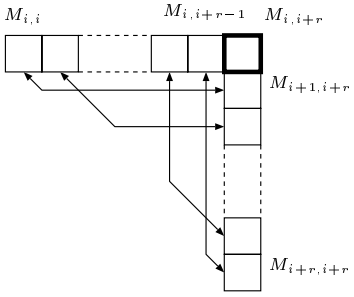

Fig. 5. The computation of $M_{i,i+r}$ by Algorithm DP-OPT

time.

IV. A PARALLEL DYNAMIC PROGRAMMING ALGORITHM FOR THE DMM

It is possible to implement Algorithm DP-OPT in the DMM as it is. However, such implementation involves bank conflicts. The main purpose of this section is to modify Algorithm DP-OPT such that its implementation performs conflict-free memory access.

The modified algorithm, Algorithm DMM-OPT, has Stages 0, 1, ..., $n-2$. Each Stage r $(1 \leq r \leq n-2)$ performs $M_{i,j} \leftarrow \min(M_{i,j}, X+Y)$ for X and Y such that

- one of X and Y is in diagonal $r-1$, and
- the other is in diagonals 0, 1, ..., $r-1$.

Hence, the operation $M_{i,j} \leftarrow \min(M_{i,j}, X+Y)$ is performed for elements $M_{i,j}$ in diagonals r, $r+1$, ..., $\min(n-2, 2r)$. Thus, in each Stage r, the values of elements in diagonal r are determined. Also, the values of elements in diagonals $r+1$, $r+2$, ..., $\min(n-2, 2r)$ are partially computed. The reader should refer to Figure 6 for illustrating diagonals computed by Algorithm DMM-OPT.

We will show how $M_{i,j} \leftarrow \min(M_{i,j}, X+Y)$ is performed in Stage r.

Case 1: X is in diagonal $r-1$, say, $X = M_{i,i+r-1}$.
We perform $M_{i,j} \leftarrow \min(M_{i,j}, M_{i,i+r-1} + M_{i+r,j})$ for $j = i+r, i+r+1, \ldots, \min(n-1, i+2r-1)$. Let us verify the reason why j takes value up to $\min(n-1, i+2r-1)$. Clearly, $M_{i+r,j}$ in j-th column, and thus, $j \leq n-1$. Also, since $M_{i+r,j}$ is in diagonal $j-(i+r)$ and it must be in diagonal $r-1$ or smaller, $j-(i+r) \leq r-1$, that is, $j \leq i+2r-1$ be satisfied. Thus, $j \leq \min(n-1, i+2r-1)$ holds. Figure 7 illustrates elements updated in Case 1.

Case 2: Y is in diagonal $r-1$, say, $Y = M_{i+1,i+r}$, $M_{i+2,i+r+1}, \ldots, M_{i+r,\min(n-1,i+2r-1)}$.
We perform $M_{i,j} \leftarrow \min(M_{i,j}, M_{i,j-r} + M_{j-r+1,j})$ for $j = i+r, i+r+1, \ldots, \min(n-1, i+2r-1)$. Figure 8 illustrates elements updated in Case 2.

We are now in a position to write Algorithm DMM-OPT. The details are spelled out as follows:

[**Algorithm DMM-OPT**]
for $i \leftarrow 1$ to $n-2$ do in parallel // Loop A

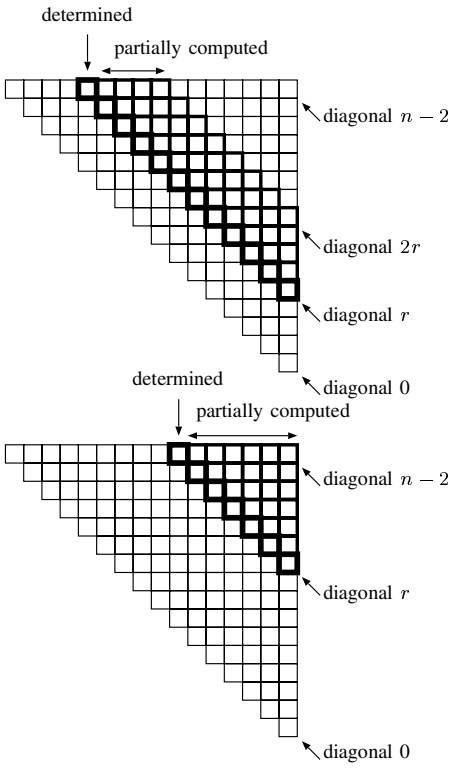

Fig. 6. Diagonals computed by Algorithm DMM-OPT

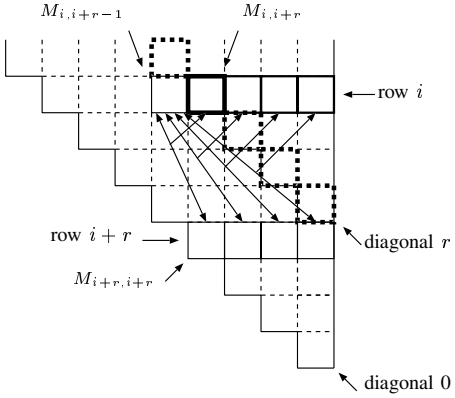

Fig. 7. The computation of Algorithm UMM-OPT for Case 1

for $j \leftarrow i+1$ to $n-1$ do in parallel
  $M_{i,j} \leftarrow +\infty$
for $i \leftarrow 1$ to $n-1$ do in parallel // Loop B (Stage 0)
  $M_{i,i} \leftarrow 0$
for $r \leftarrow 1$ to $n-2$ do // (Stage r)
  for $i \leftarrow 1$ to $n-r-1$ do in parallel //Loop C (Case 1)
    for $j \leftarrow i+r$ to $\min(n-1, i+2r-1)$ do in parallel
      $M_{i,j} \leftarrow \min(M_{i,j}, M_{i,i+r-1} + M_{i+r,j})$
  for $i \leftarrow 1$ to $n-r-1$ do in parallel //Loop D (Case 2)
    for $j \leftarrow i+r$ to $\min(n-1, i+2r-1)$ do in parallel

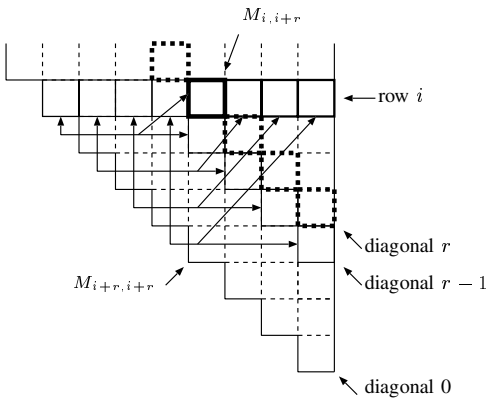

Fig. 8. The computation of Algorithm DMM-OPT for Case 2

      $M_{i,j} \leftarrow \min(M_{i,j}, M_{i,j-r} + M_{j-r+1,j})$
  for $i \leftarrow 1$ to $n-r-1$ do in parallel //Loop E
    $M_{i,i+r} \leftarrow M_{i,i+r} + c_{i-1,i+r}$

Loops A and B are the same as those of Algorithm DP-OPT. Loops C and D correspond to Cases 1 and 2, respectively. Finally, Loop E adds $c_{i-1,i+r}$ to $M_{i,i+r}$ to complete the computation of all elements in diagonal r. We use p ($\geq w$)

Let us implement Algorithm DMM-OPT in the DMM. We arrange arrays M and c such that $M_{i,j}$ and $c_{i,j}$ are arranged in bank $B[j \bmod w]$ of the shared memory on the DMM with width w. Thus, if a warp of w thread performs incremental column access such that $M_{i_0,j}$, $M_{i_1,j+1}$, ..., $M_{i_{w-1},j+w-1}$, for any integer $i_0, i_1, \ldots, i_{w-1}$ and j, then these elements are in distinct memory banks and the memory access is conflict-free. Also, if a warp of w threads access to the same element, it is also conflict-free.

Let us evaluate the computing time of Algorithm DMM-OPT on the DMM with width w and latency l using p ($w \leq p \leq n^2$) threads. Loop A performs conflict-free writing operations for less than n^2 elements. Since the memory access is conflict-free, Loop A takes at most $O(\frac{n^2}{w} + \frac{n^2 l}{p})$ time units from Lemma 1. Since the memory access is conflict-free, Loop B takes $O(\frac{n}{w} + \frac{nl}{p})$ time units. Let us evaluate the time of each Stage r ($1 \leq r \leq n-2$). Each thread $T(j)$ ($0 \leq j \leq n-1$) accesses $M_{i,j}$, $M_{i,i+r-1}$, $M_{i+r,j}$, $M_{i,j-r}$, and $M_{j-r+1,j}$ for each i ($1 \leq i \leq n-r-1$) for Loops C and D. Clearly, the memory access operations are conflict-free. Since less than $O(n^2)$ elements are accessed, this takes $O(\frac{n^2}{w} + \frac{n^2 l}{p})$ time units. Since the memory access operations are conflict-free, Loop E takes $O(\frac{n}{w} + \frac{nl}{p})$ time units. Hence, each Stage r takes $O(\frac{n^2}{w} + \frac{n^2 l}{p})$ time units. Thus, we have,

Lemma 3: Algorithm DMM-OPT runs $O(\frac{n^3}{w} + \frac{n^3 l}{p})$ time units using p ($w \leq p \leq n^2$) threads on the DMM with width w and latency l.

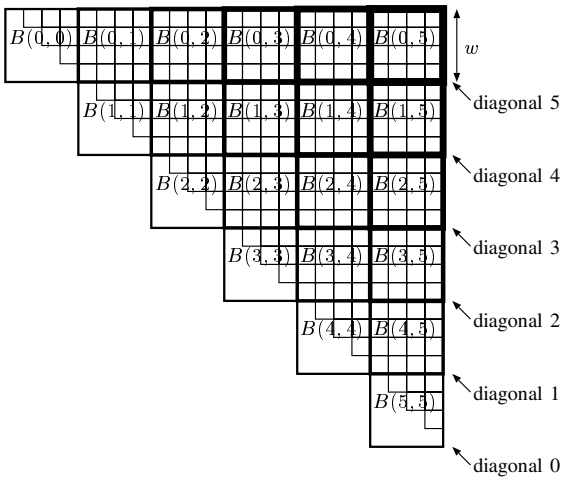

Fig. 9. Partition M into blocks of size $w \times w$ each when $w = 4$

V. A Parallel Dynamic Programming Algorithm for the HMM

This section is devoted to show a parallel algorithm that computes array M for the OPT on the HMM. We assume that arrays M and c are stored in the global memory of the HMM. For simplicity, we assume that the number n of nodes is a multiple of width w.

The idea is to partition M into blocks of size $w \times w$ each as illustrated in Figure 9. In the figure, M for a convex 24-gon is partitioned into blocks of size 4×4. In general, M for a convex n-gon has $n-1$ rows and $n-1$ columns and thus, $\frac{n}{w}$ rows and $\frac{n}{w}$ columns of blocks. Let $s = \frac{n}{w}$. As illustrated in the figure, let $B(i, j)$ $(0 \leq i \leq j \leq s-1)$ denote a block of M. As before, we can think that for each r $(0 \leq r \leq s-1)$, $B(0, r), B(1, r+1), \ldots, B(s-r-1, s-1)$ are in diagonal r. Since we assumed n is a multiple of w and M has $n-1$ rows and $n-1$ columns, each block in diagonal 0 has $w-1$ rows and $w-1$ columns. For later reference, we partition array c with $n-2$ rows and $n-2$ columns into s rows and s columns in the same way. Similarly, let $C(i, j)$ $(0 \leq i \leq j \leq s-1)$ denote a block of c.

If all elements of M in blocks of diagonal 0, 1, ..., $r-1$ are computed, we can compute those in blocks of diagonal r. Thus, all elements in M can be computed by the following algorithm:

[Algorithm HMM-OPT]
for $r \leftarrow 0$ to $s-1$ do // (Stage r)
  each block of M in diagonal r is assigned a DMM
  and the DMM compute the values of the block;

In Stage r, $s-r$ DMMs are used to compute the values of $s-r$ blocks. Let us show how each stage is performed.

Stage 0 of Algorithm HMM-OPT

In Stage 0, all values of each $B(i, i)$ $(0 \leq i \leq s-1)$ in diagonal 0 are computed by a DMM. This can be done by executing Algorithm DMM-OPT in each DMM. Let us see how all values in $B(0, 0)$ is computed by a DMM. The other blocks can be computed in the same manner. To compute all elements M in $B(0, 0)$, the values of $c_{i,j}$ in $C(0, 0)$ are necessary. The DMM copies all elements of $C(0, 0)$ from the global memory to the shared memory. After that, Algorithm DMM-OPT is executed to compute all values in $B(0, 0)$. The resulting values in $B(0, 0)$ are copied from the shared memory to the global memory.

Stage 1 of Algorithm HMM-OPT

In Stage 1, all values of each $B(i, i+1)$ $(0 \leq i \leq s-2)$ in diagonal 1 are computed by a DMM. Let us see how all values in $B(0, 1)$ is computed. To compute $B(0, 1)$, the values of $B(0, 0)$, $B(1, 1)$, and $C(0, 1)$ are necessary. A DMM copies these values from the global memory to the shared memory. After that, Algorithm DMM-OPT is executed to compute all values in $B(0, 1)$. The resulting values of $B(0, 1)$ are copied from the shared memory to the global memory.

Stage 2 of Algorithm HMM-OPT

In Stage 2, all values of each $B(i, i+2)$ $(0 \leq i \leq s-3)$ in diagonal 2 are computed by a DMM. Let us see how all values in $B(0, 2)$ is computed. Note that, $B(0, 0)$, $B(0, 1)$, $B(1, 2)$, $B(2, 2)$, and $C(0, 2)$ are used to compute the values in $B_{0,2}$. The reader should refer to Figure 10. Stage 1 consists of 2 substages. In the first substage, for every $M_{i,j}$ in $B(0, 2)$,

$$\min\{M_{i,k} + M_{k+1,j} \mid M_{i,k} \text{ is in } B(0,1) \text{ and } M_{k+1,j} \text{ is in } B(1,2)\}$$

is computed and stored it in $M_{i,j}$. In other words, $M_{i,j}$s in $B(0, 2)$ are partially computed. For this purpose, $B(0, 1)$ and $B(1, 2)$ are copied from the global memory to the shared memory of the DMM. After that, the DMM executes the following algorithm:

[The first substage of Stage 2 of Algorithm HMM-OPT]
for $i \leftarrow 1$ to w do in parallel
  for $j \leftarrow 2w$ to $3w-1$ do in parallel
    $M_{i,j} \leftarrow +\infty$
for $k \leftarrow w$ to $2w-1$ do
  for $i \leftarrow 1$ to w do in parallel
    for $j \leftarrow 2w$ to $3w-1$ do in parallel
      $M_{i,j} \leftarrow \min(M_{i,j}, M_{i,k} + M_{k+1,j})$

The reader should have no difficulty to confirm that, in this algorithm, all $M_{i,j}$s are in $B(0, 0)$, all $M_{i,k}$s are in $B(0, 1)$, and all $M_{k+1,j}$s are in $B(1, 2)$.

The second stage determines the final values of $M_{i,j}$ in $B(0, 2)$ using $B(0, 0)$, $B(2, 2)$, (the current values of) $B(0, 2)$ and $C(0, 2)$ using a DMM. Thus, these values are copied from the global memory to the shared memory. After that, the final values of $B(0, 2)$ are computed in a similar way to Algorithm DMM-OPT as follows. Elements $M_{i,j}$ in $B(0, 2)$ are partitioned in $2w-1$ diagonals from 0 to $2w-2$ as illustrated in Figure 11. The values of $M_{i,j}$s are determined from diagonal 0 to $2w-2$. Note that, the value of $M_{2w,w}$ in

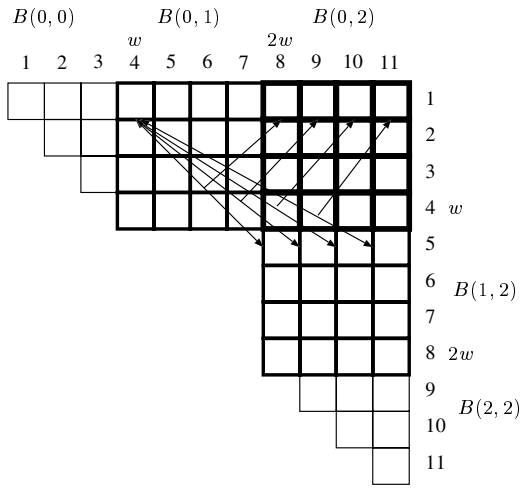

Fig. 10. The first substage of Stage 2 of Algorithm HMM-OPT

diagonal 0 has been already determined when the first stage terminates. When $M_{i,j}$s in diagonal r ($1 \leq r \leq 2w-2$) are determined, those in diagonal $r+1$, $r+2$, ..., $2w-2$ are partially computed. Similarly to two cases of DMM-OPT illustrated in Figures 7 and 8, $M_{i,j}$s are updated by two cases as illustrated in Figure 11.

Stages 3, 4, ..., $s-1$ of Algorithm HMM-OPT

In Stage r ($3 \leq r \leq s-1$), all values of each $B(i, i+r)$ ($0 \leq i \leq s-r-1$) in diagonal r are computed by a DMM. The values of $B(i, i+r)$ are determined in r substages. In the first substage, $B(i, i+1)$ and $B(i+1, i+r)$ are used to compute the values of $B(i, i+r)$ partially. The second substage partially computes the values of $B(i, i+r)$ using $B(i, i+2)$ and $B(i+2, i+r)$. The same operation are repeated until the substage $r-1$ partially computes the values of $B(i, i+r)$ using $B(i, i+r-1)$ and $B(i+r-1, i+r)$. After that, the substage r determines the values of $B(i, i+r)$ using $B(i, i)$, $B(i+r, i+r)$, and $C(i, i+r)$ in a similar way to the second substage of Stage 2.

The reader should have no difficulty to confirm that memory access performed in all stages are conflict-free. Let us evaluate the computing time of each stage. We evaluate the computing time of Algorithm HMM-OPT if we execute it on the HMM with $s = \frac{n}{w}$ DMMs with $p = wl$ threads each. Hence, the HMM has totally $ps = \frac{n}{l}$ threads.

In Stage 0, we assign one DMM to compute $B(i, i)$ ($0 \leq i \leq s-1$). First, $C(i, i)$ is copied from the global memory to the shared memory. Since each $C(i, i)$ is arranged in a matrix of size $w \times w$, at most $w^2 s = nw$ elements are read from the global memory. This task takes $O(\frac{nw}{w} + \frac{nwL}{ps}) = O(n + wL)$ time units from Lemma 2. After that, the value of $B(i, i)$ is computed in a DMM in parallel. From Lemma 3, this takes $O(\frac{w^3}{w} + \frac{w^3 l}{p}) = O(w^2)$ time units. Finally, the resulting values of all $B(i, i)$ are written in the global memory in $O(n + wL)$ time units. Hence, Stage 0 takes $O(n + wL)$ time units.

Stage 1 has two substages. In the first substage, $B(i, i)$, $B(i+1, i+1)$, and $C(i, i+1)$ are copied from the global memory to compute each $B(i, i+1)$ by a DMM. This copy operations takes $O(n + wL)$ time units. After that, the values of each $B(i, i+1)$ are computed by a DMM in $O(w^2)$ time units. Finally, the resulting values of all $B(i, i+1)$ are written in the global memory in $O(n+wL)$ time units. Hence, Stage 1 also takes $O(n + wL)$ time units.

Let us evaluate the computing time of Stage r ($3 \leq r \leq s-1$). Stage r has r substages. Each of the first $r-1$ substage can be done in the same way as the first substage of Stage 2. The last substage can be done in the same way as the second substage of Stage 2. Since each substage takes at most $O(n + wL)$ time units. Thus, Stage r takes $O(r(n+wL))$ time units.

By summing the computing time of all stages from 0 to $s-1$, we have that Algorithm HMM-OPT runs $O(s^2(n + wL)) = O(\frac{n^3}{w^2} + \frac{n^2 L}{w})$ time units on the HMM. Thus, we have,

Lemma 4: Algorithm HMM-OPT runs $O(\frac{n^3}{w^2} + \frac{n^2 L}{w})$ time units on the HMM using $\frac{n}{w}$ DMMs with wl threads each.

If $n < wL$, the latency overhead $O(\frac{n^2 L}{w})$ is dominant.

VI. Reducing the latency overhead of Algorithm HMM-OPT

We can reduce $O(\frac{n^2 L}{w})$-time latency overhead of Algorithm HMM-OPT by computing every block $B(i, j)$ in parallel.

Intuitively, we can write the computation performed by Algorithm HMM-OPT as follows:

$$B(i,j) = C(i,j) \quad \text{if } j - i \leq 1, \tag{3}$$

$$B(i,j) = \min_{i \leq k \leq j}(B(i,k) + B(k,j)) + C(i,j) \quad \text{otherwise.} \tag{4}$$

Note that Formulas (3) and (4) are informal. Formula (3) implies that the values of $B(i, j)$ can be determined only from $C(i, j)$. Formula (4) means that the values of $B(i, j)$ can be computed by taking the minimum of pairwise sums of elements one from $B(i, k)$ and the other from $B(k, j)$. We can obtain each value of $B(i, j)$ by adding the corresponding value of $C(i, j)$ and the minimum of pairwise sums. These formulas are essentially the same as Formulas (1) and (2). Thus, we can apply partial computation technique used in Algorithm DMM-OPT to Algorithm HMM-OPT. As illustrated in Figure 6, Algorithm DMM-OPT determines the values in diagonal r and partially computes the values in diagonals $r+1$, $r+2$, ..., $\min(n-2, 2r)$ in each Stage r ($0 \leq r \leq n-2$). We apply this technique for blocks in Algorithm HMM-OPT. More specifically, each Stage r ($0 \leq r \leq s-1$) determines the values in block diagonal r and partially computes the values in block diagonals $r+1$, $r+2$, ..., $\min(s-1, 2r)$, where $s = \frac{n}{w}$. Using this idea, all blocks can be computed as follows:

[Algorithm HMM-OPT2]

```
for i ← 0 to s − 1 do in parallel // Loop A
  for j ← i to s − 1 do in parallel
    B(i, j) ← +∞
for i ← 0 to s − 1 do in parallel // Loop B (Stage 0)
```

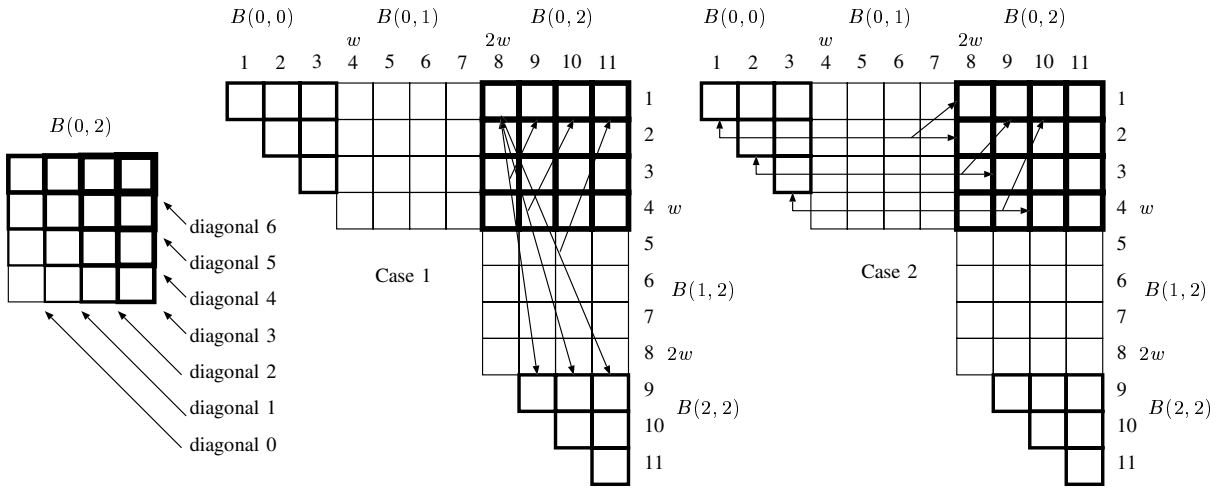

Fig. 11. The second stage of Algorithm HMM-OPT

$B(i,i) \leftarrow C(i,i)$
for $r \leftarrow 1$ to $s-1$ do // (Stage r)
  for $i \leftarrow 0$ to $s-1$ do in parallel // Loop C
    for $j \leftarrow i+r$ to $\min(s-1, i+2r)$ do in parallel
      $B(i,j) \leftarrow \min(B(i,j), B(i,i+r) + B(i+r,j),$
        $B(i,j-r) + B(j-r,j))$
  for $i \leftarrow 0$ to $s-r-1$ do in parallel // Loop D
    $B(i,i+r) \leftarrow \min(B(i,i) + B(i,i+r) + C(i,i+r),$
      $B(i,i+r) + B(i+r,i+r) + C(i,i+r))$

Loop A initializes all $B(i,j)$s by $+\infty$. Loop B, which corresponds to Stage 0, computes each $B(i,i)$ using $C(i,i)$. In Loop C, all $B(i,j)$ are updated using elements in diagonal r. Figure 12 illustrates how $B(i,j)$ are updated. To update $B(i,j)$, the values of $B(i,i+r) + B(i+r+1,j)$ and $B(i,j-r-1)+B(j-r,j)$ are computed. The values of $B(i,j)$ are updated by their minima. Note that both $B(i,i+r)$ and $B(j-r,j)$ are in diagonal r and both $B(i+r,j)$ and $B(i,j-r)$ are in diagonals 0, 1, ..., $r-1$. Hence, all values in $B(i,i+r)$, $B(i+r,j)$, $B(i,j-r)$, $B(j-r,j)$ have been already computed. Loop D, which corresponds to the second substage of Stage 2 of Algorithm HMM-OPT, is illustrated in Figure 11. The computation of $B(i,i)+B(i,i+r)+C(i,i+r)$ corresponds to Case 1, and that of $B(i,i+r)+B(i+r,i+r)+C(i,i+r)$ corresponds to Case 2. Hence, $B(i,i)$, $B(i,i+r)$, $B(i+r,i+r)$, and $C(i,i+r)$ are copied from the global memory to the shared memory, and the final values of $B(i,j)$ are computed using them.

Let us evaluate the computing time of this algorithm. We assign one DMM with $p = wl$ threads to update the value of each $B(i,j)$. Thus, we use $d = \frac{s(s+1)}{2}$ DMMs which has totally $pd \approx \frac{n^2 l}{2w}$ threads. In Loop A, p threads in each DMM write 0 to the global memory. Loop A takes $O(\frac{n^2}{w} + \frac{n^2 L}{pd}) = O(\frac{n^2}{w} + \frac{wL}{l})$ time units from Lemma 2. In Loop B, a DMM is assigned each $B(i,i)$ and copies $C(i,i)$ from the global memory to the shared memory and computes the values of $B(i,i)$ using Algorithm DMM-OPT. The resulting values of $B(i,i)$ are copied from the shared memory to the global memory. Since $O(n^2)$ elements are copied, the copy operations take $O(\frac{n^2}{w} + \frac{n^2 L}{pd}) = O(\frac{n^2}{w} + \frac{wL}{l})$ time units from Lemma 2. The computation of $B(i,i)$ takes $O(\frac{w^3}{w} + \frac{w^3 l}{p}) = O(w^2)$ time units from Lemma 3. Loop B takes $O(\frac{n^2}{w} + \frac{wL}{l} + w^2)$ time units. For each value r in Loop C, a DMM is assigned each $B(i,j)$ and updates the value of $B(i,j)$. For this purpose, it copies $B(i,i+r)$, $B(i+r,j)$, $B(i,j-r)$, and $B(j-r,j)$ from the global memory to the shared memory. After that, each DMM partially computes $B(i,j)$ in a similar way to a substage of Algorithm HMM-OPT. After that, the resulting values of $B(i,j)$ are written in the global memory. Since $O(n^2)$ elements are

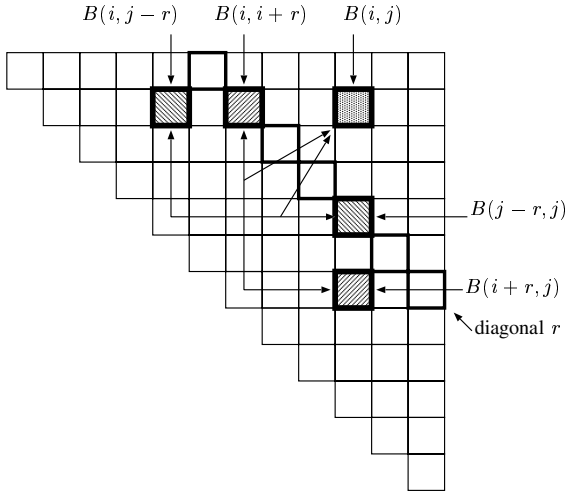

Fig. 12. The update of $B(i,j)$ using elements in diagonal r of Algorithm HMM-OPT2

copied, the copy operations take $O(\frac{n^2}{w} + \frac{wL}{l})$ time units. The computation of $B(i, j)$ is performed in parallel, which takes $O(w^2)$ time units. Hence, all stages of Loop C take $(s-1) \cdot O(\frac{n^2}{w} + \frac{wL}{l} + w^2) = O(\frac{n^3}{w^2} + \frac{nL}{l} + nw)$ time units. Since Loop D is almost the same as the second substage of Algorithm HMM-OPT, it can be done in $O(\frac{n^2}{w} + \frac{wL}{l} + w^2)$ time units. Thus, we have,

Lemma 5: Algorithm HMM-OPT2 runs $O(\frac{n^3}{w^2} + \frac{nL}{l} + nw)$ time units on the HMM using less than $\frac{n^2}{w^2}$ DMMs with wl threads each.

Next let us consider the case that we have fewer DMMs. Let d be the number of available DMMs and assume that $d \leq \frac{s(s+1)}{2}$. Since Loop C dominates the total computing time, we will evaluate the running time of Loop C. For each value of loop variable r in Loop C, the computation of d $B(i, j)$s are performed using d DMMs in parallel. This computation is repeated $O(\frac{s^2}{d}) = O(\frac{n^2}{w^2 d})$ times. Since each iteration of the computation by d $B(i, j)$s performs the copy operation for $O(dw^2)$ elements, each iteration takes $O(\frac{dw^2}{w} + \frac{dw^2 L}{dwl}) = O(dw + \frac{wL}{l})$ time units from Lemma 2. Also, the computation of $B(i, j)$ takes $O(w^2)$ time units from Lemma 3. Hence, for each r of Loop C, each iteration by d DMMs takes $O(dw + \frac{wL}{l} + w^2)$ time units. Thus, for each value of r, Loop C takes $O(\frac{n^2}{w^2 d}) \cdot O(dw + \frac{wL}{l} + w^2) = O(\frac{n^2}{w} + \frac{n^2 L}{wdl} + \frac{n^2}{d})$ time units. Therefore, Loop C of Algorithm HMM-OPT2 runs $s \cdot O(\frac{n^2}{w} + \frac{n^2 L}{wdl} + \frac{n^2}{d}) = O(\frac{n^3}{w^2} + \frac{n^3 L}{w^2 dl} + \frac{n^3}{wd})$ time units. Thus, we have,

Theorem 6: Algorithm HMM-OPT2 runs $O(\frac{n^3}{w^2} + \frac{n^3 L}{w^2 dl} + \frac{n^3}{wd})$ time units on the HMM d ($\leq \frac{n^2}{w^2}$) DMMs with wl threads each.

From Theorem 6, Algorithm HMM-OPT2 runs $O(\frac{n^3}{w^2})$ time units if $d \geq \frac{L}{l}$ and $d \geq w$. Hence, we have,

Corollary 7: Algorithm HMM-OPT2 runs $O(\frac{n^3}{w^2})$ time units on the HMM using $\max(\frac{L}{l}, w)$ DMMs with wl threads each.

Hence, if the total number of threads is $\max(wL, w^2 l)$ the OPT can be solved in $O(\frac{n^3}{w^2})$ time units on the HMM. When $L = O(wl)$, the total number of threads is $\max(wL, w^2 l) = O(w^2 l)$. Since each thread can send at most one memory request in l time units, it takes at least $\Omega(\frac{n^3}{w^2})$ time units for $O(w^2 l)$ threads to send $O(n^3)$ memory requests. Thus, the parallel implementation shown for Corollary 7 is time optimal.

VII. Conclusion

The main contribution of this paper is to show an efficient implementation of the dynamic programming on the HMM, which is a theoretical parallel computing model of CUDA-enabled GPUs. Our implementation runs $O(\frac{n^3}{w^2})$ time units on the HMM using $\max(wL, w^2 l)$ threads. Since the sequential algorithm takes $O(n^3)$ time, our implementation achieves a speed up factor of w^2. Further, we have proved that it is time optimal when $L = O(wl)$.

References

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

[2] 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.

[3] Y. Takeuchi, D. Takafuji, Y. Ito, and K. Nakano, "Ascii art generation using the local exhaustive search on the GPU," in *Proc. of International Symposium on Computing and Networking*, Dec. 2013, pp. 194–200.

[4] K. Ogawa, Y. Ito, and K. Nakano, "Efficient Canny edge detection using a GPU," in *Proc. of International Conference on Networking and Computing*. IEEE CS Press, Nov. 2010, pp. 279–280.

[5] K. Nishida, Y. Ito, and K. Nakano, "Accelerating the dynamic programming for the matrix chain product on the GPU," in *Proc. of International Conference on Networking and Computing*, Dec. 2011, pp. 320–326.

[6] ——, "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.

[7] A. Uchida, Y. Ito, and K. Nakano, "An efficient GPU implementation of ant colony optimization for the traveling salesman problem," in *Proc. of International Conference on Networking and Computing*. IEEE CS Press, Dec. 2012, pp. 94–102.

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

[9] 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.

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

[11] K. Nakano, "Optimal parallel algorithms for computing the sum, the prefix-sums, and the summed area table on the memory machine models," *IEICE Trans. on Information and Systems*, vol. E96-D, no. 12, pp. 2626–2634, 2013.

[12] ——, "Simple memory machine models for GPUs," *International Journal of Parallel, Emergent and Distributed Systems*, vol. 29, no. 1, pp. 17–37, 2014.

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

[14] A. Kasagi, K. Nakano, and Y. Ito, "An optimal offline permutation algorithm on the hierarchical memory machine, with the GPU implementation," in *Proc. of International Conference on Parallel Processing*. IEEE CS Press, Oct. 2013, pp. 1–10.

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

[16] L. Bergroth, H. Hakonen, and T. T. Raita, "A survey of longest common subsequence algorithms," in *Proc. of International Symposium on String Processing and Information Retrieval*, 2000.

[17] P. D. Gilbert, "New results on planar Triangulations," in *M.Sc. thesis*, July 1979, pp. Report R–850.

[18] G. T. Klincsek, "Minimal triangulations of polygonal domains," *Annals of Discrete Mathematics*, vol. 9, pp. 121–123, July 1980.

[19] K. Nakano, "Sequential memory access on the unified memory machine with application to the dynamic programming," in *Proc. of International Symposium on Computing and Networking*, Dec. 2013, pp. 85–94.

[20] P. Steffen, R. Giegerich, and M. Giraud, "GPU parallelization of algebraic dynamic programming," in *Proc. of International Conference on Parallel Processing and Applied Mathematics: Part II*, Sept. 2009, pp. 290–299.

[21] C.-C. Wu, J.-Y. Ke, H. Lin, and W. chun Feng, "Optimizing dynamic programming on graphics processing units via adaptive thread-level parallelism," in *Proc. of International Conference on Parallel and Distributed Systems*, Dec. 2011.

[22] S. Xiao, A. M. Aji, and W. chun Feng, "On the robust mapping of dynamic programming onto a graphics processing unit," in *Proc. of International Conference on Parallel and Distributed Systems*, Dec. 2009, pp. 26–33.

[23] A. Kasagi, K. Nakano, and Y. Ito, "An implementation of conflict-free off-line permutation on the GPU," in *Proc. of International Conference on Networking and Computing*, 2012, pp. 226–232.