Sequential Memory Access on the Unified Memory Machine with Application to the Dynamic Programming

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 Unified Memory Machine (UMM) is a theoretical parallel computing model that captures the essence of the global memory access of GPUs. Although it is a good theoretical model for GPU computing, the performance analysis of parallel algorithms on it is sometimes complicated. The main contribution of this paper is to provide a useful gadget, the sequential memory access, that makes the computing time evaluation easy, and to show its application to the dynamic programming. The sequential memory access has two parameters: length n and fragmentation f. We first show that the sequential memory access of length n with fragmentation f can be done in $O(\frac{n}{w}+\frac{nl}{p}+l+f)$ time units using p threads on the UMM with width w and latency l. We next show that the dynamic programming to solve the optimal polygon triangulation problem can be implemented in the UMM using the sequential memory access. The resulting implementation for a convex n-gon runs in $O(\frac{n^3}{w}+\frac{n^3l}{p}+nl)$ time units using p threads on the UMM with width w and latency l. We also prove that any implementation of the dynamic programming needs $\Omega(\frac{n^3}{w}+\frac{n^3l}{p}+nl)$ time units. Thus, our implementation is time optimal.**

***Keywords*-Dynamic programming, parallel algorithms, memory machine models, coalesced memory access, GPU, CUDA**

I. Introduction

A. Background

The research of parallel algorithms has a long history of more than 40 years. Sequential algorithms have been developed mostly on the Random Access Machine (RAM) [1]. In contrast, since there are a variety of connection methods and patterns between processors and memories, many parallel computing models have been presented and many parallel algorithmic techniques have been shown on them. The most well-studied parallel computing model is the Parallel Random Access Machine (PRAM) [2], [3], [4], [5], which consists of processors and a shared memory. Each processor on the PRAM can access any address of the shared memory in a time unit. The PRAM is a good parallel computing model in the sense that parallelism of each problem can be revealed by the performance of parallel algorithms on the PRAM. However, since the PRAM requires a shared memory that can be accessed by all processors at the same time, it is not feasible.

The GPU (Graphics Processing Unit) is a specialized circuit designed to accelerate computation for building and manipulating images [6], [7], [8]. 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 [6], [9], [10], [11], [12]. NVIDIA provides a parallel computing architecture called *CUDA* (Compute Unified Device Architecture) [13], 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 [14], 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* [13]. 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 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 [11], [14], [15]. 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.

The most well-studied parallel computing model is the Parallel Random Access Machine (PRAM) [2], [3], [4], [5], which consists of processors and a shared memory. Each processor on the PRAM can access any address of the shared memory in a time unit. The PRAM is a good parallel computing model in the sense that parallelism of each problem

can be revealed by the performance of parallel algorithms on the PRAM. Although GPUs have the shared memory and the global memory accessed by multiple threads, parallel algorithms developed for the PRAM may not achieve good performance on GPUs. We should consider the memory access characteristics such as the bank conflicts and the coalescing when we develop efficient parallel algorithms for GPUs. There are several previously published works that aim to present theoretical practical parallel computing models capturing the essence of parallel computers. Many researchers have been devoted to developing efficient parallel algorithms to find algorithmic techniques on such parallel computing models. For example, processors connected by interconnection networks such as hypercubes, meshes, trees, among others [16], bulk synchronous models [17], LogP models [18], reconfigurable models [19], among others. Quite recently, the memory machine models [20] have been presented for theoretical parallel computing models for CUDA-enabled GPUs.

B. Memory Machine Models

In our previous paper [20], 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 CUDA-enabled GPUs. Since the DMM and the UMM are promising as theoretical computing models for GPUs, we have published several efficient algorithms on the DMM and the UMM [21], [22], [23], [24]. For example, in our previous paper [20], we have presented offline permutation algorithms on the DMM and the UMM. We have also implemented the offline permutation algorithm on NVIDIA GeForce GTX-680 and showed that theoretical analysis of the performance on the DMM provides very good approximation of the CUDA C implementation of the offline permutation algorithm [21]. These results imply that the DMM is a good theoretical model for computation using the shared memory on GPUs. Later, we have introduced *the Hierarchical Memory Machine (HMM)* [25], which captures the essence of the hierarchical architecture of the CUDA-enabled GPU. The HMM has multiple DMMs, each of which corresponds to a streaming multiple-processor on a GPU. It also has a global memory which can be accessed by all threads in DMMs. Since all threads share a global memory, we can think it is a UMM. In [26], we have shown an approximate string matching algorithm on the HMM and implemented it on the HMM. In [27], we have presented an offline permutation algorithm on the HMM and evaluated its performance on the CUDA-enabled GPU. The implementation results show that that theoretical analysis of the performance on the HMM provides very good approximation of the actual running time. However, performance analysis of parallel algorithms on the Memory Machine Models including the DMM, the UMM, and the HMM is sometimes complicated and difficult.

The DMM and the UMM have three parameters: the number p of threads, width w, and memory access latency l. Figure 1 illustrates the outline of the architectures of the DMM and the UMM with $p = 20$ threads and width $w = 4$. Each thread is a Random Access Machine (RAM) [1], which can execute fundamental operations in a time unit. Threads are executed in SIMD [28] fashion, and run on the same program and work on the different data. The p threads are partitioned into $\frac{p}{w}$ groups of w threads each called *warp*. The $\frac{p}{w}$ warps are dispatched for memory access in turn, and w threads in a dispatched warp send memory access requests to the memory banks (MBs) through the memory management unit (MMU). We do not discuss the architecture of the MMU, but we can think that it is a multistage interconnection network in which memory access requests are moved to destination memory banks in a pipeline fashion. Note that the DMM and the UMM with width w has w memory banks and each warp has w threads. For example, the DMM and the UMM in Figure 1 have 4 threads in each warp and 4 MBs.

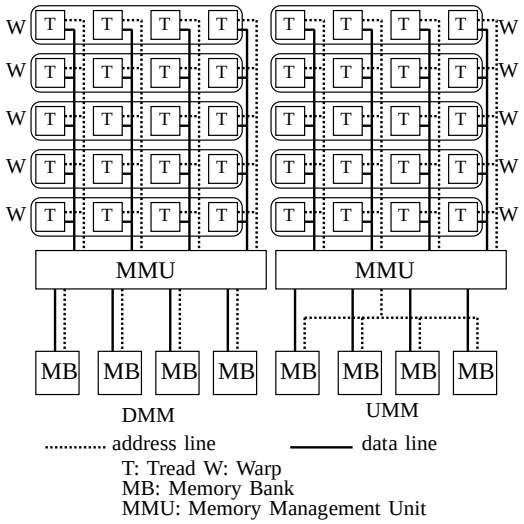

Fig. 1. The architectures of the DMM and the UMM with width $w = 4$

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 $B[i]$, 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 set of address lines 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. Also, we assume that MBs are accessed in a pipeline fashion with latency l. In other words, if a thread sends a memory access request, it takes at least l time units to complete it. A thread can send a new memory access request only after the completion of the previous memory access request and thus,

it can send at most one memory access request in l time units.

Let us clarify the difference of the DMM and the UMM using the bank groups and the address groups. Figure 2 illustrates the memory banks and the address groups. Let $B[j] = \{j, j+w, j+2w, j+3w, \ldots\}$ $(0 \leq j \leq w-1)$ denote the j-th memory bank. Also, let $A[j] = \{j \cdot w, j \cdot w + 1, \ldots, (j+1) \cdot w - 1\}$ denote the j-th address group. In the DMM, if multiple memory access requests are destined for the same memory bank, they are processed sequentially. In the UMM, memory access requests are destined for different address groups, they are processed separately.

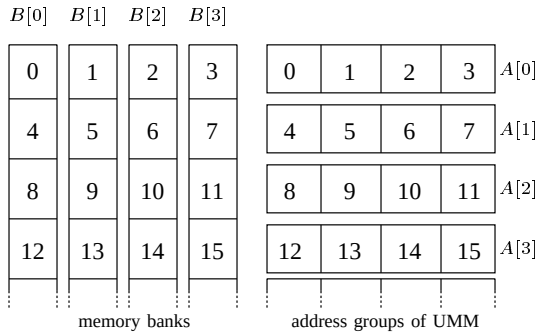

Fig. 2. The memory banks and the address group of the DMM and the UMM for $w = 4$ banks

The performance of algorithms of the PRAM is usually 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 [2]. 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 as well as the number of threads in a warp. The latency l is the number of time units to complete the memory access. For example, we have shown in [24] that the prefix-sums of n numbers can be computed in $O(\frac{n}{w} + \frac{nl}{p} + l \log n)$ time units. In latest CUDA-enabled GPUs, the width w of the shared memory is 32, and that of the global memory is 256-384 bits. Also, the latency l of the shared memory is very small, while that 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 [13]. Thus, the number p of threads can be 65 million.

C. Our contribution

Although the DMM and the UMM are good theoretical models for GPU computing, the performance analysis of parallel algorithms on it is not easy. The main contribution of this paper is to provide a useful gadget, *the sequential memory access* that makes the computing time evaluation on the UMM easy. In the sequential memory access, p threads on the UMM access n addresses such that each thread accesses $\frac{n}{p}$ addresses in turn. The memory access can be either read or write. The sequential memory access has two parameters: length n and fragmentation value f. Figure 3 illustrates an example of sequential memory access of length 12 by 12 threads. We say that a pair of two adjacent memory access requests is a gap, if it is not adjacent in the address space. For example, the sequential memory access in Figure 3 has two gaps: memory access by threads 3 and 4, and that by 6 and 7. *The fragmentation of the sequential memory access* is the number of gaps. We show that the sequential memory access of length n with fragmentation f can be done in $O(\frac{n}{w} + \frac{nl}{p} + l + f)$ time units using p threads on the UMM with width w and latency l.

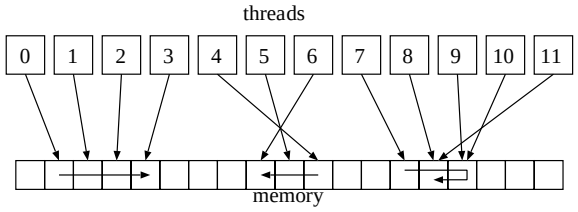

Fig. 3. An example of sequential memory access

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 [29]. A naive solution for such problem needs exponential time. The key idea behind the dynamic programming is to:

- partition a problem into subproblems,
- solve the subproblems independently, and
- combine the solution of the subproblems

to reach an overall solution. 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 [30]. 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 [29].

The second contribution of this paper is to show that the dynamic programming for solving the optimal polygon triangulation problem can be implemented using the sequential memory access. In the optimal polygon triangulation 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 4 shows an example of the input convex 8-gon of nodes $v_0, v_1, \ldots, v_7$ with each chord $v_i v_j$ having weight $c_{i,j}$. It also shows the optimal triangulation with total weight 6. It is known that the optimal polygon triangulation problem for a convex n-gon can be solved in $O(n^3)$ time using the dynamic

programming technique [11], [29], [31], [32]. As far as we know, there is no previously published algorithm running faster than $O(n^3)$ time.

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 implementation of the dynamic programming is not an easy task.

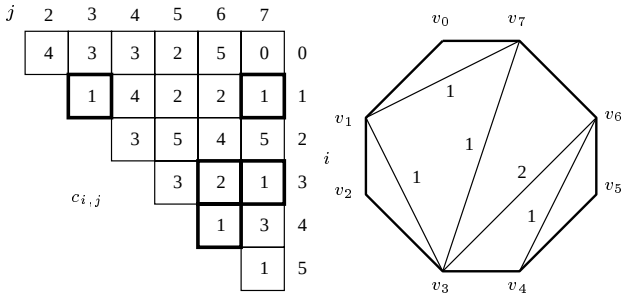

Fig. 4. An example of a triangulation of a convex 8-gon

We will show that the optimal polygon triangulation problem can be solved using the sequential memory access. Our implementation for a convex n-gon runs in $O(\frac{n^3}{w} + \frac{n^3 l}{p} + nl)$ time units using p threads on the UMM with width w and latency l. We also prove that the optimal polygon triangulation problem needs $\Omega(\frac{n^3}{w} + \frac{n^3 l}{p} + nl)$ time units as log as the dynamic programming technique is used. Thus, our implementation is time optimal.

There are several published works on the implementation of the dynamic programming [11], [10], [33], [34], [35]. 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. Our contribution is the first work that presents a parallel implementation for the dynamic programming for the optimal polygon triangulation problem guaranteed to be optimal by the theoretical analysis.

The rest of this paper is organized as follows: In Section II, we define the Unified Memory Machine (UMM) and the sequential memory access. We also evaluate the performance of the sequential memory access on the UMM. Section III defines the optimal triangulation problem (OPT problem) and review the dynamic programming for solving this problem. In Section IV, we show an implementation of the dynamic programming for solving the OPT problem in the UMM. Section V evaluates the computing time of the implementation using the sequential memory access. It also proves the time optimality of our implementation. Section VI concludes our work.

II. The Unified Memory Machine (UMM) and the sequential memory access

The main purpose of this section is to define the Unified Memory Machine (UMM) [20]. The reader should refer to [20] for the details of the the UMM. It also defines the sequential memory access and evaluates its running time on the UMM.

Let us define the UMM with width w and latency l. Let $m[i]$ ($i \geq 0$) denote the memory cell with address i. The memory of the UMM is partitioned into address groups $A[0], A[1], \ldots$ such that each $A[j]$ ($j \geq 0$) stores $m[j \cdot w]$, $m[j \cdot w + 1]$, $\ldots$, $m[(j+1) \cdot w - 1]$. The reader should refer to Figure 2 that illustrates address groups for $w = 4$. Also, the memory access is performed through l-stage pipeline registers as illustrated in 5. Let p be the number of threads of the UMM and $T(0)$, $T(1)$, $\ldots$, $T(p-1)$ be the p threads. We assume that p is a multiple of w. The p threads are partitioned into $\frac{p}{w}$ groups called *warps* with w threads each. More specifically, p threads are partitioned into $\frac{p}{w}$ warps $W(0), W(1), \ldots, W(\frac{p}{w}-1)$ such that $W(i) = \{T(i \cdot w), T(i \cdot w + 1), \ldots, T((i+1) \cdot w - 1)\}$. Warps are dispatched for memory access in turn, and w threads in a warp try to access the memory in the same time. More specifically, $W(0), W(1), \ldots, W(\frac{p}{w} - 1)$ 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 thread in $W(i)$ send memory access requests, one request per thread, to the memory banks.

For the memory access, each warp sends memory access requests to the memory banks through the l-stage pipeline registers. We assume that each stage can store the memory access requests destined for the same address group. For example, since the memory access requests by $W(0)$ are separated in three address groups in the figure, they occupy three stages of the pipeline registers. Also, those by $W(1)$ are in the same address group, they occupy only one stage. In general, if memory access requests by a warp are destined for k address groups, they occupy k stages. For simplicity, we assume that the memory access is completed as soon as the request reaches the last pipeline stage. Thus, all memory access requests by $W(0)$ and $W(1)$ in the figure are completed in $3(\text{address groups}) + 1(\text{address group}) + 5(\text{latency}) - 1 = 8$ time units. 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 also assume that, if two or more threads access the same address, the memory access requests are processed as a single request. If multiple memory read requests are destined for the same address, the value stored in the address is broadcast to the source threads. If multiple memory write requests are sent to

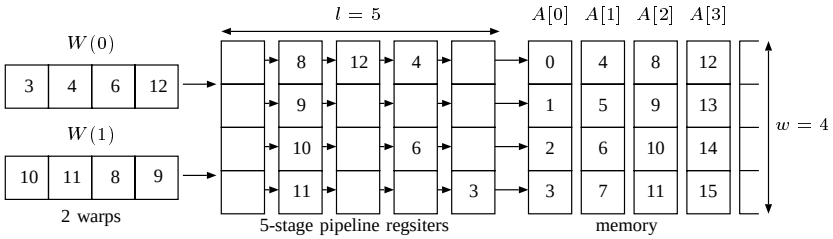

Fig. 5. The UMM with width $w = 4$ and latency $l = 5$

the same address, then one of them is arbitrarily selected and its writing operation is performed. The other writing requests to the same address are removed. Thus, the UMM works as the Concurrent Read Concurrent Write (CRCW) mode with arbitrary resolution of simultaneous writing [2]. From the function of CUDA [13], these assumptions make sense.

Let $a_0, a_1, \ldots, a_{n-1}$ be a sequence of n addresses accessed by threads on the UMM. We say that a pair of two addresses a_i and a_{i+1} $(0 \leq i \leq n-2)$ of adjacent access requests is *a gap* if $|a_i - a_{i+1}| \geq 2$. Clearly, it is not a gap if they are adjacent in the address space or the same address, that is, $|a_i - a_{i+1}| \leq 1$. We say that the number of gaps is *the fragmentation* of the sequence. For example, if $a_i = i$ for all i then its fragmentation is 0. If $a_i = 2i$ then its fragmentation is $n-1$.

The sequential memory access is a memory access operation for a sequence of n addresses such that p threads on the UMM access $\frac{n}{p}$ addresses each as follows:

[**Sequential Memory Access**]
for $t \leftarrow 0$ to $\lceil \frac{n}{p} \rceil - 1$ do
  for $i \leftarrow 0$ to $p-1$ do in parallel
    $T(i)$ accesses $m[a_{t \cdot p+i}]$ if $t \cdot p + i < n$.

Note that "accesses" can be either "reads from" or "writes in". However, "reads from" and "writes in" cannot be mixed. Thus, the sequential memory access must be either the sequential memory read or the sequential memory write. Hence, it is not possible to copy $m[a_j]$ to $m[a_k]$ by executing the sequential memory access once even if $j < k$.

Let us evaluate the time for the sequential memory access of length n with fragmentation f. First, we assume that $p \leq n$. Recall that p threads are partitioned into $\frac{p}{w}$ warps of w threads each. Let us partition n addresses into $\frac{n}{w}$ groups of w addresses each such that each group is accessed by a warp. Let $G_j = \{a_{j \cdot w}, a_{j \cdot w+1}, \ldots, a_{(j+1) \cdot w-1}\}$ $(0 \leq j \leq \frac{n}{w} - 1)$ denote the j-th group. Suppose that group G_j has no gap. Since $|a_i - a_{i+1}| \leq 1$ for all i, $|a - a'| \leq w$ always holds for all $a, a' \in G_j$. Hence, w addresses in G_j must be in one or two address groups. Let k be the integer such that a pair $a_{j \cdot w+k}$ and $a_{j \cdot w+k+1}$ is a gap. We can partition G_j into two subgroups by the gap such that $\{a_{j \cdot w}, a_{j \cdot w+1}, \ldots, a_{(j+1) \cdot w+k}\}$ and $\{a_{j \cdot w+k+1}, a_{j \cdot w+k+2}, \ldots, a_{(i+1) \cdot w-1}\}$ are two subgroups. Since each subgroup has no gap, it is in one or two address groups. Hence, G_j must be in at most four address groups. Let f_j $(0 \leq j \leq \frac{n}{w} - 1)$ be the number of gaps in each G_j. In general, each G_j can be partitioned into $f_j + 1$ subgroups with no gap. Since each group must be in at most two address groups, G_j must be in at most $2f_j + 2$ address groups. Hence, for each t, the memory access by p threads takes,

$$\left(\sum_{j=t \cdot \frac{p}{w}}^{(t+1)\frac{p}{w}-1} (2f_j + 2) \right) + (l-1)$$

time units. Since the sequence has totally f gaps, $f_0 + f_1 + \cdots + f_{\frac{n}{w}-1} \leq f$ holds. Thus, the sequential memory access of length n with fragmentation f takes

$$\begin{aligned} & \sum_{t=0}^{\frac{n}{p}-1} \left(\left(\sum_{j=t \cdot \frac{p}{w}}^{(t+1)\frac{p}{w}-1} (2f_j + 2) \right) + (l-1) \right) \\ = \;& 2f + \sum_{t=0}^{\frac{n}{p}-1} \left(\frac{2p}{w} + l - 1 \right) = O\left(\frac{n}{w} + \frac{nl}{p} + f \right) \end{aligned}$$

time units.

Next, suppose that $p > n$. If this is the case, n threads accesses n addresses. Thus, the sequential memory access takes

$$\left(\sum_{i=0}^{\frac{n}{w}-1} (2f_i + 2) \right) + (l-1) \quad = \quad O\left(\frac{n}{w} + l + f \right)$$

time units.

Combining the two cases $p \leq n$ and $p > n$, we have

Theorem 1: The sequential memory access of length n with fragmentation f takes $O(\frac{n}{w} + \frac{nl}{p} + l + f)$ time units using p threads on the UMM with width w and latency l.

III. The Optimal Polygon Triangulation and the Dynamic Programming

This section defines the optimal polygon triangulation problem (OPT problem) and reviews an algorithm solving this problem by the dynamic programming technique [11], [29].

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 4 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 optimal polygon triangulation problem (OPT problem)* is to find an optimal polygon triangulation that

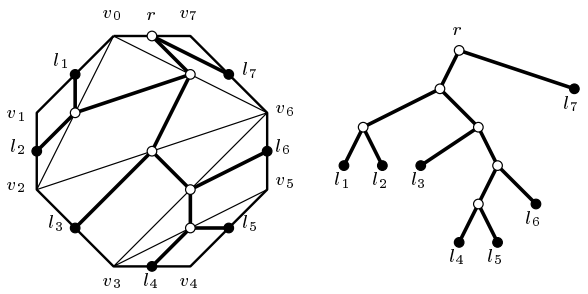

Fig. 6. The parse tree of a triangulation

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.

We will show that the optimal polygon triangulation can be solved by the dynamic programming technique. For this purpose, we define *the parse tree* of a triangulation. Figure 6 illustrates the parse tree of a triangulation. Let l_i ($1 \leq i \leq n-1$) be edge $v_{i-1}v_i$ of a convex n-gon. Also, let r denote edge v_0v_{n-1}. The parse tree is a binary tree of a triangulation, which has the root r and $n-1$ leaves $l_1, l_2, \ldots, l_{n-1}$. It also has $n-3$ internal nodes (excluding the root r), each of which corresponds to a chord of the triangulation. Edges are drawn from the root toward the leaves as illustrated in Figure 6. Since each triangle has three nodes, the resulting graph is a full binary tree with $n-1$ leaves, in which every internal node has exactly two children. Conversely, for any full binary tree with $n-1$ leaves, we can draw a unique triangulation. It is well known that the number of full binary trees with $n+1$ leaves is the Catalan number $\frac{(2n)!}{(n+1)!n!}$[36]. Thus, the number of possible triangulations of convex n-gon is $\frac{(2n-4)!}{(n-1)!(n-2)!}$. Hence, a naive approach, which evaluates the total weights of all possible triangulations, takes an exponential time.

We are now in a position to show an algorithm using the dynamic programming for the optimal polygon triangulation problem. Suppose that an n-gon is chopped off by a chord $v_{i-1}v_j$ ($0 \leq i < j \leq n-1$) and we obtain a $(j-i+2)$-gon with vertices $v_{i-1}, v_i, \ldots, v_j$ as illustrated in Figure 7. Clearly, this $(j-i+2)$-gon consists of leaves $l_i, l_{i+1}, \ldots, l_j$ and a chord $v_{i-1}v_j$. Let $m_{i,j}$ be the minimum weight of the $(j-i+2)$-gon. The $(j-i+2)$-gon can be partitioned into the $(k-i+2)$-gon, the $(j-k+1)$-gon, and the triangle $v_{i-1}v_kv_j$ as illustrated in Figure 7. The values of k can be an integer from i to $j-1$. Thus, we can recursively define $m_{i,j}$ as follows:

$$\begin{aligned} m_{i,j} &= 0 \qquad \text{if } j-i \leq 1, \\ m_{i,j} &= \min_{i \leq k \leq j-1} (m_{i,k} + m_{k+1,j} + c_{i-1,k} + c_{k,j}) \\ &\qquad \text{otherwise.} \end{aligned}$$

The figure also shows its parse tree. The reader should have no

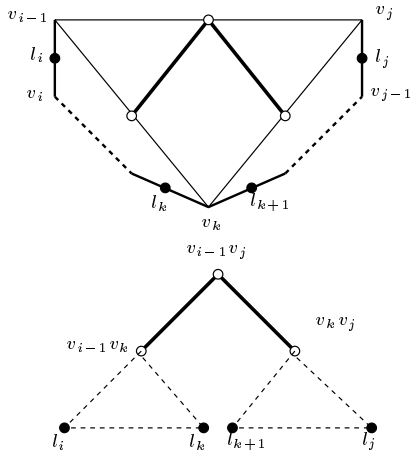

Fig. 7. A $(j-i+2)$-gon is partitioned into a $(k-i+2)$-gon and a $(j-k+1)$-gon

difficulty to confirm the correctness of the recursive formula and the minimum weight of the n-gon is equal to $m_{1,n-1}$.

To reduce the computation, we let $M_{i,j} = m_{i,j} + c_{i-1,j}$ and $c_{0,n-1} = 0$. We can recursively define $M_{i,j}$ as follows:

$$\begin{aligned} M_{i,j} &= 0 \qquad \text{if } j-i \leq 1, \\ M_{i,j} &= \min_{i \leq k \leq j-1} (M_{i,k} + M_{k+1,j}) + c_{i-1,j} \quad \text{otherwise.} \end{aligned}$$

Clearly, from $c_{0,n-1} = 0$, $M_{1,n-1} = m_{1,n-1} + c_{0,n-1} = m_{1,n-1}$ is the minimum weight of the n-gon. Let us start with a simple algorithm for computing all $M_{i,j}$. 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. In other words each $M_{i,j}$ is in diagonal $j-i$. 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 8 shows elements computed in each Stage r for 8-gon illustrated in Figure 4. 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 9 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.

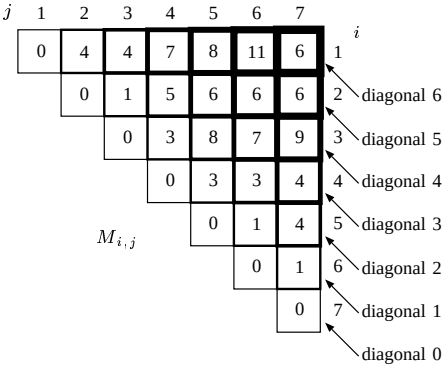

Fig. 8. The computation of $M_{i,i+r}$ and the resulting values of $M_{i,j}$

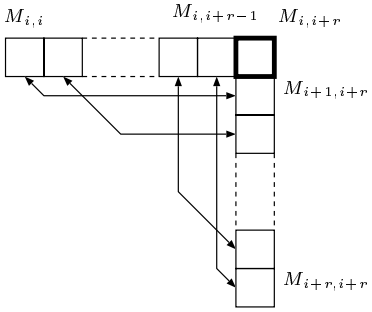

Fig. 9. The computation of $M_{i,i+r}$ by Algorithm DP-OPT

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

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

time.

IV. An algorithm for the OPT problem on the UMM

It is possible to implement Algorithm DP-OPT in the UMM as it is. However, such implementation will perform the sequential memory access with large fragmentation. The main purpose of this section is to modify Algorithm DP-OPT such that its implementation performs the sequential memory access with small fragmentation.

The modified algorithm, Algorithm UMM-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, $r+1$, ..., $\min(n-2, 2r)$. Hence, 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 10 for illustrating diagonals computed by Algorithm UMM-OPT.

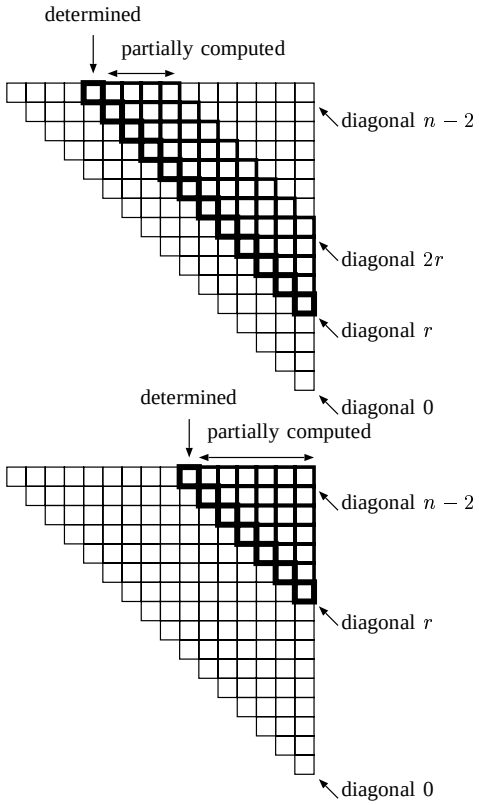

Fig. 10. Diagonals computed by Algorithm UMM-OPT

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 11 illustrates elements updated in Case 1. We can see that sequential memory read is performed for elements in row $i+r$ and sequential memory write is performed for those in row i.

Case 2: Y is in diagonal $r-1$, say, $Y = M_{i+1,i+r}$, $M_{i+2,i+r+1}$, ..., $M_{i+r,\min(n-1,i+2r-1)}$.

We perform $M_{i,j} \leftarrow \min(M_{i,j}, M_{i,j-r} + M_{j-r+1,j})$ for $j = i+r, i+r+1, \ldots, \min(n-1, i+2r-1)$. The reader should have no difficulty to confirm that j takes value up to $\min(n-1, i+2r-1)$ similarly to Case 1. Figure 12 illustrates elements updated in Case 2. We can see that sequential memory read is performed for elements in row i and sequential memory write is performed for those in row i. Also, sequential memory read is performed for elements in diagonal $r-1$. The fragmentation of this sequential memory read is very large, because elements are not in the same row. Hence, we store all elements in diagonal $r-1$ in a 1-dimensional array b at the beginning of Stage r to reduce the fragmentation.

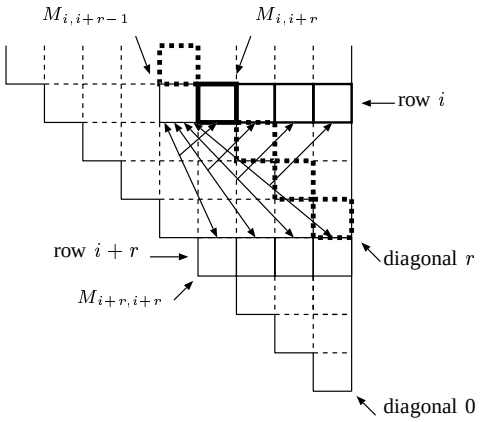

Fig. 11. The computation of Algorithm UMM-OPT for Case 1

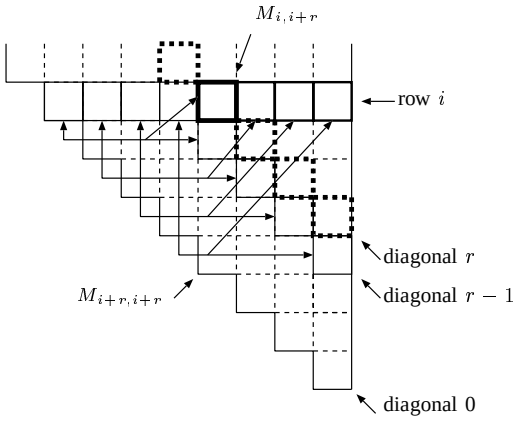

Fig. 12. The computation of Algorithm UMM-OPT for Case 2

We are now in a position to write Algorithm UMM-OPT. Let $b_1, b_2, \ldots, b_{n-1}$ denote elements of 1-dimensional array b. The details are spelled out as follows:

[**Algorithm UMM-OPT**]
for $i \leftarrow 1$ to $n-2$ do in parallel // 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 in parallel //Loop C
    $b_i \leftarrow M_{i,i+r-1}$
  for $i \leftarrow 1$ to $n-r-1$ do in parallel //Loop D (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 E (Case 2)
    for $j \leftarrow i+r$ to $\min(n-1, i+2r-1)$ do in parallel
      $M_{i,j} \leftarrow \min(M_{i,j}, M_{i,j-r} + b_{j-r+1})$
  for $i \leftarrow 1$ to $n-r-1$ do in parallel //Loop F
    $M_{i,i+r} \leftarrow M_{i,i+r} + c_{i-1,i+r}$

Loops A and B are the same as those of Algorithm DP-OPT. Loop C copies all elements in diagonal $r-1$ to 1-dimensional array b. Loops D and E correspond to Cases 1 and 2 respectively. Finally, Loop F adds $c_{i-1,i+r}$ to $M_{i,i+r}$ to complete the computation of all elements in diagonal r.

V. The computing time of Algorithm UMM-OPT on UMM

This section is devoted to evaluating the running time of Algorithm UMM-OPT on the UMM with p threads, width w and latency l. We use Theorem 1 to evaluate the time for the sequential memory access performed by Algorithm UMM-OPT. For this purpose, we will evaluate the length and the fragmentation of the sequential memory access performed in each loop of Algorithm UMM-OPT. Once we have the length and the fragmentation, the running time can be evaluated using Theorem 1. The reader should refer to Table I that summarizes the computing time of each loop of Algorithm UMM-OPT.

In Loop A, the sequential memory write to a sequence $M_{i,i+1}, M_{i,i+2}, \ldots, M_{i,n-1}$ is performed for each i ($1 \leq i \leq n-2$). Clearly, this sequence has no gap. Thus, Loop A can be done by the sequential memory write of length $\sum_{i=1}^{n-2}(n-i-1) < n^2$ with fragmentation $n-3$. Loop B performs the sequential memory write to a sequence $M_{1,1}, M_{2,2}, \ldots, M_{n-1,n-1}$. Since every adjacent pair is a gap, this sequence of length $n-1$ has $n-2$ gaps.

Next, let us evaluate the length and the fragmentation of sequential memory access performed in each loop of each Stage r ($1 \leq r \leq n-2$). In Loop C, $n-r-1$ elements in diagonal $r-1$ of M are read. This corresponds to the sequential read of length $n-r-1$ and fragmentation $n-r-2$. It also performs the sequential memory write for 1-dimensional array b with length $n-2$ with fragmentation 0. Loop D performs three types of sequential memory access: $M_{i,j}$, $M_{i,i+r-1}$, and $M_{i+r,j}$. For each i ($1 \leq i \leq n-r-1$), the sequential memory access to $M_{i,j}$ has no gap. Hence, the sequential memory access to $M_{i,j}$ combined for all i is of

TABLE I
THE ANALYSIS OF RUNNING TIME OF ALGORITHM UMM-OPT

		accessed array	length	fragmentation	running time	remark
Loop A		$M_{i,j}$	$< n^2$	$n-3$	$O(\frac{n^2}{w}+\frac{n^2l}{p}+l+n)$	initialization by $+\infty$
Loop B		$M_{i,i}$	$n-1$	$n-2$	$O(\frac{n}{w}+\frac{nl}{p}+l+n)$	initialization by 0
Stage r	Loop C	$M_{i,i+r-1}$	$n-r-1$	$n-r-2$	$O(\frac{n}{w}+\frac{nl}{p}+l+n)$	reading from diagonal $r-1$
		b_i	$n-r-1$	0	$O(\frac{n}{w}+\frac{nl}{p}+l)$	writing in a 1-d array
	Loop D	$M_{i,j}$	$< n^2$	$n-r-2$	$O(\frac{n^2}{w}+\frac{n^2l}{p}+l+n)$	reading/writing row i
		$M_{i,i+r-1}$	$< n^2$	$n-r-2$	$O(\frac{n}{w}+\frac{nl}{p}+l)$	reading from diagonal $r-1$
		$M_{i+r,j}$	$< n^2$	$n-r-2$	$O(\frac{n^2}{w}+\frac{n^2l}{p}+l+n)$	reading from row $i+r$
	Loop E	$M_{i,j}$	$< n^2$	$n-r-2$	$O(\frac{n^2}{w}+\frac{n^2l}{p}+l+n)$	reading/writing row i
		$M_{i,j-r}$	$< n^2$	$n-r-2$	$O(\frac{n^2}{w}+\frac{n^2l}{p}+l+n)$	reading from row i
		m_{j-r+1}	$< n^2$	$n-r-2$	$O(\frac{n^2}{w}+\frac{n^2l}{p}+l+n)$	reading from a 1-d array
	Loop F	$M_{i,i+r}$	$n-r-1$	$n-r-2$	$O(\frac{n}{w}+\frac{nl}{p}+l+n)$	reading/writing diagonal r
		$c_{i-1,i+r}$	$n-r-1$	$n-r-2$	$O(\frac{n}{w}+\frac{nl}{p}+l+n)$	reading/writing diagonal $r-1$ of w
All Stages					$O(\frac{n^3}{w}+\frac{n^3l}{p}+nl+n^2)$	

length

$$\sum_{i=1}^{n-r-1}(\min(n-1,i+2r-1)-(i+r)+1) \quad < \quad n^2$$

with fragmentation $n-r-2$. Similarly, the sequential memory access to $M_{i,i+r-1}$ and $M_{i+r,j}$ is of length $< n^2$ with fragmentation $n-r-2$. Loop D performs three types of sequential memory access: $M_{i,j}$, $M_{i,j-r}$, and b_{j-r+1}. The reader should have no difficulty to confirm that each sequential memory access is of length $< n^2$ with fragmentation $n-r-2$. Loop E performs two memory access operations: $M_{i,i+r}$ and $c_{i-1,i+r}$. These sequential memory access are of length $n-r-1$ with fragmentation $n-r-2$. Thus, each Stage r takes $O(\frac{n^2}{w}+\frac{n^2l}{p}+l+n)$ time units. It follows that Stages 1 to $n-2$ takes $O(\frac{n^3}{w}+\frac{n^3l}{p}+nl+n^2)$ time units.

We will show that $O(n^2)$ in the computing time can be removed and Algorithm UMM-OPT runs in $O(\frac{n^3}{w}+\frac{n^3l}{p}+nl)$ time units. We consider three cases: $w \leq n$, $w \geq n^2$, and $n < w < n^2$.

Case 1: $w \leq n$

From $\frac{n^3}{w} \geq n^2$, we have $O(\frac{n^3}{w}+\frac{n^3l}{p}+nl+n^2) = O(\frac{n^3}{w}+\frac{n^3l}{p}+nl)$.

Case 2: $w \geq n^2$

Tables M and c can be stored in one address group each. Thus, all memory access requests by a warp occupy one pipeline stage. Hence, the sequential memory access of length $< n^2$ for tables M and c can be done in $O(\frac{n^2}{w}+\frac{n^2l}{p}+l)$ time units. Hence, every loop of Algorithm UMM-OPT takes $O(\frac{n^2}{w}+\frac{n^2l}{p}+l)$ time units, and Algorithm UMM-OPT runs in $O(\frac{n^3}{w}+\frac{n^3l}{p}+nl)$ time units.

Case 3: $n < w < n^2$

For simplicity, we assume that n is a multiple of w and let $s = \frac{n^2}{w}$. Clearly, table M are partitioned into s address groups. Hence consecutive s rows are in the same address group. In Algorithm UMM-OPT, row i of M is accessed for each value of loop variable i. Hence, every loop accesses M from the top row to the bottom row. We can think that a pair of two adjacent memory accesses is a gap, only if they are in different address groups. Under this definition of the gap, the sequential memory access of each loop has $s-1$ gaps. Also, it should have no difficulty to confirm that Theorem 1 holds under this definition. Hence, each sequential memory access can be done in $O(\frac{n^2}{w}+\frac{n^2l}{p}+l+s) = O(\frac{n^2}{w}+\frac{n^2l}{p}+l)$ time units and Algorithm UMM-OPT runs in $O(\frac{n^3}{w}+\frac{n^3l}{p}+nl)$ time units.

Finally, we have,

Theorem 2: The optimal polygon triangulation problem of a convex n-gon can be solved in $O(\frac{n^3}{w}+\frac{n^3l}{p}+nl)$ time units using p threads on the UMM with width w and latency l.

Let us discuss the time lower bound for Theorem 2. Note that, we will prove the lower bound of implementations of the OPT problem that uses the dynamic programming technique. It may be possible that $o(n^3)$-time algorithm for the optimal polygon triangulation problem exists, and the problem can be solved more efficiently than Theorem 2. What we prove is Theorem 2 is time optimal, as long as the dynamic programming technique is used.

Clearly, $O(n^3)$ numbers must be read at least once, and the w memory banks on the UMM accept at most w access requests in a time unit. Hence, it takes at least $\Omega(\frac{n^3}{w})$ time units to solve the OPT problem. Also, p threads can access at most p numbers in l time units, and thus they can access at most $\frac{pt}{l}$ numbers in t time units. Since $\frac{pt}{l} \geq n^3$ must be satisfied, it takes at least $t = \Omega(\frac{n^3l}{p})$ time units. Also, to compute the values of M in diagonal r, those in diagonal $r-1$ are necessary. In other words, the values of $r-1$ must be read before those in diagonal r are written. This takes at least $\Omega(l)$ time units. Since we need to compute the value in diagonal $n-2$, it takes at least $\Omega(nl)$ time units to solve the OPT problem.

From the discussion above, we have,

Theorem 3: Any implementation of the dynamic programming based parallel algorithm for the OPT problem for an n-gon needs $\Omega(\frac{n^3}{w} + \frac{n^3 l}{p} + nl)$ time units using p threads on the UMM with width w and latency l.

This theorem implies that Algorithm UMM-OPT for Theorem 2 is time optimal has no extra overhead.

VI. Conclusion

In this paper, we have presented the sequential memory access on the Unified Memory Machine (UMM) that makes the computing time evaluation easy, and shown an optimal algorithm for the optimal triangulation problem (OPT problem) using the sequential memory access on the UMM. We have shown that the sequential memory access of length n with fragmentation f can be done in $O(\frac{n}{w} + \frac{nl}{p} + l + f)$ time units using p threads on the UMM with width w and latency l. Also, we have shown that the dynamic programming to solve the optimal polygon triangulation problem can be implemented using the sequential memory access on the UMM. The resulting implementation for a convex n-gon runs in $O(\frac{n^3}{w} + \frac{n^3 l}{p} + nl)$ time units using p threads on the UMM with width w and latency l. We have also proved that any implementation of the dynamic programming based parallel algorithm for the OPT problem for an n-gon needs $\Omega(\frac{n^3}{w} + \frac{n^3 l}{p} + nl)$ time units. Thus, our implementation is time optimal.

In our previous paper [25], we have presented the Hierarchical Memory Machine (HMM), a hybrid of the UMM and the DMM. The HMM is a more realistic model of CUDA-enabled GPUs, which has multiple streaming processors with the global memory. It is very interesting future work to show a more efficient implementation on the HMM for the dynamic programming, that runs in $o(\frac{n^3}{w})$ time units.

References

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

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

[3] J. JáJá, *An Introduction to Parallel Algorithms*. Addison-Wesley, 1992.

[4] A. Grama, G. Karypis, V. Kumar, and A. Gupta, *Introduction to Parallel Computing*. Addison Wesley, 2003.

[5] M. J. Quinn, *Parallel Computing: Theory and Practice*. McGraw-Hill, 1994.

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

[7] 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.

[8] 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.

[9] K. Ogawa, Y. Ito, and K. Nakano, "Efficient canny edge detection using a gpu," in *Proc. of International Conference on Networking and Computing*, Nov. 2010, pp. 279–280.

[10] 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.

[11] ——, "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.

[12] 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*, Dec. 2012, pp. 94–102.

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

[14] 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.

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

[16] F. T. Leighton, *Introduction to Parallel Algorithms and Architectures: Arrays, Trees, Hypercubes*. Morgan Kaufmann, 1991.

[17] R. H. Bisseling, *Parallel Scientific Computation: A Structured Approach using BSP and MPI*. Oxford University Press, 2004.

[18] D. Culler, R. Karp, D. Patterson, A. Sahay, K. E. Schauser, E. Santos, R. Subramonian, and T. von Eicken, "LogP: towards a realistic model of parallel computation," in *Proceedings of the fourth ACM SIGPLAN symposium on Principles and practice of parallel programming*. ACM, 1993, pp. 1–12.

[19] R. Vaidyanathan and J. L. Trahan, *Dynamic Reconfiguration: Architectures and Algorithms*. Kluwer Academic/Plenum Publishers, 2004.

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

[21] 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.

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

[23] ——, "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.

[24] ——, "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.

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

[26] D. Man, K. Nakano, and Y. Ito, "The approximate string matching on the hierarchical memory machine, with performance evaluation," in *Proc. of International Symposium on Embedded Multicore/Many-core System-on-Chip*, Sept. 2013.

[27] 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*, Oct. 2013.

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

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

[30] 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.

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

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

[33] 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.

[34] 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.

[35] 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.

[36] G. Pólya, "On picture-writing," *Amer. Math. Monthly*, vol. 63, pp. 689–697, 1956.