PAPER

A GPU Implementation of Dynamic Programming for the Optimal Polygon Triangulation

Yasuaki ITO[†] *and* **Koji NAKANO**[†]**,** *Members*

SUMMARY This paper presents a GPU (Graphics Processing Units) implementation of dynamic programming for the optimal polygon triangulation. Recently, GPUs can be used for general purpose parallel computation. Users can develop parallel programs running on GPUs using programming architecture called CUDA (Compute Unified Device Architecture) provided by NVIDIA. The optimal polygon triangulation problem for a convex polygon is an optimization problem to find a triangulation with minimum total weight. It is known that this problem for a convex n-gon can be solved using the dynamic programming technique in $O(n^3)$ time using a work space of size $O(n^2)$. In this paper, we propose an efficient parallel implementation of this $O(n^3)$-time algorithm on the GPU. In our implementation, we have used two new ideas to accelerate the dynamic programming. The first idea (adaptive granularity) is to partition the dynamic programming algorithm into many sequential kernel calls of CUDA, and to select the best parameters for the size and the number of blocks for each kernel call. The second idea (sliding and mirroring arrangements) is to arrange the working data for coalesced access of the global memory in the GPU to minimize the memory access overhead. Our implementation using these two ideas solves the optimal polygon triangulation problem for a convex 8192-gon in 5.57 seconds on the NVIDIA GeForce GTX 680, while a conventional CPU implementation runs in 1939.02 seconds. Thus, our GPU implementation attains a speedup factor of 348.02.

key words: *Dynamic programming, parallel algorithms, coalesced memory access, GPGPU, CUDA*

1. Introduction

Dynamic programming is an important algorithmic technique to find an optimal solution of a problem over an exponential number of solution candidates [1]. A naive solution for such problem needs exponential time. The key idea behind 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. 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 approach [2]. Since a sequence have 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 $O(nm)$ time by the dynamic programming approach, where n and m 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 approach [1].

The GPU (Graphics Processing Unit), is a specialized circuit designed to accelerate computation for building and manipulating images [3]–[7]. 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 [3], [8], [9]. NVIDIA provides a parallel computing architecture called *CUDA* (Compute Unified Device Architecture) [10], 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 [11], since they have hundreds of processor cores running in parallel.

The main contribution of this paper is to implement the dynamic programming approach to solve *the optimal polygon triangulation problem* [1] on the GPU. Suppose that a convex n-gon is given and we want to triangulate it, that is, to split it into $n-2$ triangles by $n-3$ non-crossing chords. Figure 1 illustrates an example of a triangulation of an 8-gon. In the figure, the triangulation has 6 triangles separated by 5 non-crossing chords. We assume that each of the $\frac{n(n-3)}{2}$ chords is assigned a weight. The goal of the optimal polygon triangulation is to select $n-3$ non-crossing chords that triangulate a given convex n-gon such that the total weight of selected chords is minimized. A naive approach, which evaluates the total weights of all possible $\frac{(2n-4)!}{(n-1)!(n-2)!}$ triangulations, takes an exponential time. On the other hand, it is known that the dynamic programming technique can be applied to solve the optimal polygon triangulation in $O(n^3)$ time [1], [12], [13] using work space of size $O(n^2)$. As far as we know, there is no previously published algorithm running faster than $O(n^3)$ time.

In our implementation, we have used two new ideas to accelerate the dynamic programming algorithm. The first idea is to partition the dynamic programming algorithm into a lot of sequential kernel calls of CUDA, and to select the best method and the numbers of blocks and threads for each kernel calls (*Adaptive granularity*). The dynamic programming algorithm for an n-gon has $n-1$ stages, each of which involves the computation of multiple working data. Earlier stages of the algorithm are *fine grain* in the sense that we

Manuscript received January 1, 2011.
Manuscript revised January 1, 2011.
[†]The authors are with Department of Information Engineering, Hiroshima University, Kagamiyama 1-4-1, Higashi-Hiroshima, 739-8527, Japan
DOI: 10.1587/trans.E0.??.1

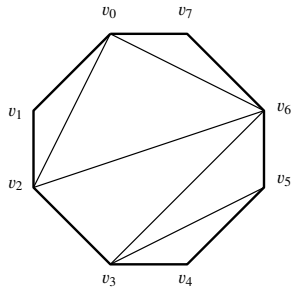


Fig. 1 An example of a triangulation of a convex 8-gon

need to compute the values of a lot of working data but the computation of each working data is light. On the other hand, later stages of the algorithm are *coarse grain* in the sense that few working data are computed but the computation is heavy. Thus, in earlier stages, a single thread is assigned to the computation of each working data and its value is computed sequentially by a thread (*thread-based method*). In later stages, one or more blocks with multiple threads are allocated to the computation for each working data and the value of the working data is computed by threads of blocks in parallel (*block-based method*). Also, the size of each block (i.e. the number of threads), and the number of used blocks affect the performance of algorithms on the GPU. We have tested these approaches for various parameters and determined the best way in each stage. Each stage selects the corresponding best way to attain the ultimate performance.

The second idea is to arrange working data in a 2-dimensional array of the global memory using two types of arrangements: *sliding arrangement* and *mirroring arrangement*. The working data used in the dynamic programming algorithm are stored in a 2-dimensional array in the global memory of the GPU. The bandwidth of the global memory is maximized when threads repeatedly perform coalesced access to it. In other words, if threads accessed to continuous locations of the global memory, these access requests can be completed in minimum clock cycles. On the other hand, if threads access distant locations in the same time, these access requests need a lot of clock cycles. We use the sliding arrangement for the thread-based method and the mirroring arrangement for the block-based method. Using these two arrangements, the coalesced access is performed for the working data.

Our implementation using these two ideas solves the optimal polygon triangulation problem for a convex 8192-gon in 5.57 seconds on the NVIDIA GeForce GTX 680, while a conventional CPU implementation runs in 1939.02 seconds. Thus, our GPU implementation attains a speedup factor of 348.02.

The rest of this paper is organized as follows; Section 2 introduces the optimal polygon triangulation problem and reviews the dynamic programming approach solving it. In Section 3, we show the GPU and CUDA architectures to understand our new idea. Section 4 proposes our two new ideas to implement the dynamic programming approach on the GPU. The experimental results are shown in Section 5. Finally, Section 6 offers concluding remarks.

2. The optimal polygon triangulation and the dynamic programming approach

The main purpose of this section is to define the optimal polygon triangulation problem and to review an algorithm solving this problem by the dynamic programming approach [1].

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 1 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 $w_{i,j}$, which is a real number, of every chord v_iv_j in a convex n-gon is given. The goal of *the optimal polygon triangulation problem* is to find an optimal triangulation that minimizes the total weights of selected chords for the triangulation. More formally, we can define the problem as follows. Let T be a set of all triangulations of a convex n-gon and $t \in T$ be a triangulation, that is, a set of $n-3$ non-crossing chords. The optimal polygon triangulation problem requires finding the total weight of a minimum weight triangulation as follows:

$$\min\{\sum_{v_iv_j \in t} w_{i,j} \mid t \in T\}.$$

We will show that the optimal polygon triangulation can be solved by the dynamic programming approach. For this purpose, we define *the parse tree* of a triangulation. Figure 2 illustrates the parse tree of a triangulation. Let l_i $(1 \le i \le n-1)$ 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 such that a node and its two children construct a triangle in a triangulation as illustrated in Figure 2. 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 $\frac{(2n)!}{(n+1)!n!}$[14]. 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 approach for the optimal polygon triangulation problem. Suppose that an n-gon is chopped off by a chord $v_{i-1}v_j$ $(0 \le i < j \le n-1)$ and we obtain a $(j-i)$-gon with vertices $v_{i-1}, v_i, \ldots, v_j$ as illustrated in Figure 3. Clearly, this $(j-i)$-gon consists of leaves $l_i, l_{i+1}, \ldots, l_j$

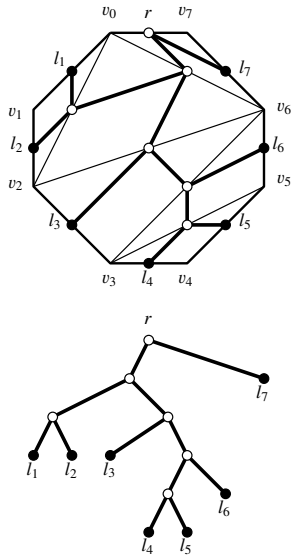


Fig. 2 The parse tree of a triangulation

and a chord $v_{i-1}v_j$. Let $m_{i,j}$ be the minimum weight of triangulation on the $(j-i)$-gon. The $(j-i)$-gon can be partitioned into the $(k-i)$-gon, the $(j-k)$-gon, and the triangle $v_{i-1}v_kv_j$ as illustrated in Figure 3. The values of k can be an integer from i to $j-1$. Thus, we can recursively define $m_{i,j}$ as follows:

$$m_{i,j} = 0 \qquad \text{if } j-i \leq 1,$$
$$m_{i,j} = \min_{i \leq k \leq j-1} (m_{i,k} + m_{k+1,j} + w_{i-1,k} + w_{k,j}) \quad \text{otherwise.}$$

The figure also shows its parse tree. The reader should have no difficulty to confirm the correctness of the recursive formula and the minimum weight of triangulation on the n-gon is equal to $m_{1,n-1}$.

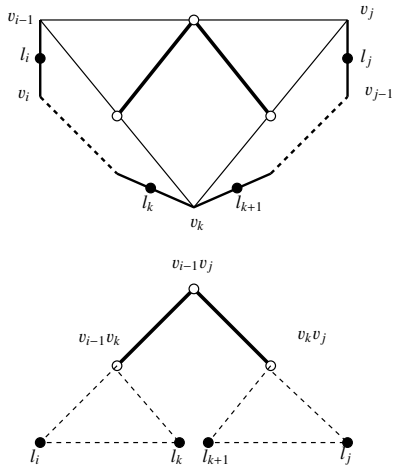


Fig. 3 A $(j-i)$-gon is partitioned into a $(k-i)$-gon and a $(j-k)$-gon

Let $M_{i,j} = m_{i,j} + w_{i-1,j}$ and $w_{0,n-1} = 0$. We can recursively define $M_{i,j}$ as follows:

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

It should be clear that $M_{1,n-1} = m_{1,n-1} + w_{0,n-1} = m_{1,n-1}$ is the minimum weight of triangulation on the n-gon.

Using the recursive formula for $M_{i,j}$, all the values of $M_{i,j}$'s can be computed in $n-1$ stages by the dynamic programming algorithm as follows:

Stage 0 $M_{1,1} = M_{2,2} = \cdots = M_{n-1,n-1} = 0$.
Stage 1 $M_{i,i+1} = w_{i-1,i+1}$ for all i $(1 \leq i \leq n-2)$
Stage 2 $M_{i,i+2} = \min_{i \leq k \leq i+1}(M_{i,k} + M_{k+1,i+2}) + w_{i-1,i+2}$ for all i $(1 \leq i \leq n-3)$
$\vdots$
Stage p $M_{i,i+p} = \min_{i \leq k \leq i+p-1}(M_{i,k} + M_{k+1,i+p}) + w_{i-1,i+p}$ for all i $(1 \leq i \leq n-p-1)$
$\vdots$
Stage $n-3$ $M_{i,n+i-3} = \min_{i \leq k \leq n+i-4}(M_{i,k} + M_{k+1,n+i-3}) + w_{i-1,n+i-3}$ for all i $(1 \leq i \leq 2)$
Stage $n-2$ $M_{1,n-1} = \min_{1 \leq k \leq n-2}(M_{i,k} + M_{k+1,n-1}) + w_{0,n-1}$

Figure 4 shows examples of $w_{i,j}$ and $M_{i,j}$ for a convex 8-gon. It should be clear that each stage computes the values of table $M_{i,j}$ in a particular diagonal position. Also, Figure 5 illustrates how $M_{i,j}$ is computed. To obtain the values of $M_{i,j}$, the sum of each of p pairs are computed and then the minimum of the p sums is computed. In other words, *the sum-minimum* of p pairs is computed for each $M_{i,j}$. Let us analyze the computation performed in each Stage p $(2 \leq p \leq n-2)$.

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

Thus, Stage p takes $(n-p-1)\cdot O(p) = O(n^2-p^2)$ time. Therefore, this algorithm runs in $\sum_{2 \leq p \leq n-2} O(n^2 - p^2) = O(n^3)$ time.

From this analysis, we can see that earlier stages of the algorithm is *fine grain* in the sense that we need to compute the values of a lot of $M_{i,j}$'s but the computation of each $M_{i,j}$ is light. On the other hand, later stages of the algorithm is *coarse grain* in the sense that few $M_{i,j}$'s are computed but its computation is heavy.

3. GPU and CUDA architectures

CUDA uses two types of memories in the NVIDIA GPUs: *the global memory* and *the shared memory* [10]. The global memory is implemented as an off-chip DRAM of the GPU, and has large capacity, say, 1.5-6 Gbytes, but its access latency is very long. The shared memory is an extremely fast on-chip memory with lower capacity, say, 16-48 Kbytes. The efficient usage of the global memory and the shared

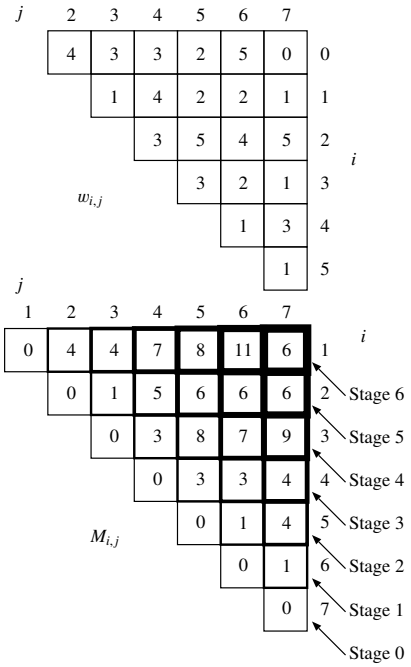


Fig. 4 Examples of $w_{i,j}$ and $M_{i,j}$

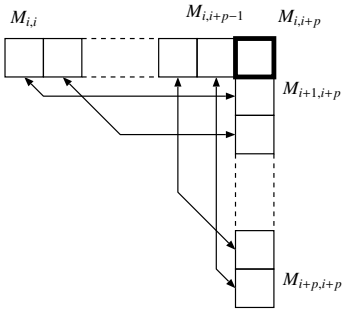


Fig. 5 The computation of $M_{i,i+p}$

memory is a key for CUDA developers to accelerate applications using GPUs. In particular, we need to consider *the coalescing* of the global memory access and *the bank conflict* of the shared memory access [5], [11], [15]. To maximize the bandwidth between the GPU and the DRAM chips, the consecutive addresses of the global memory must be accessed in the same time. Thus, threads should perform coalesced access when they access to the global memory. Figure 6 illustrates the CUDA hardware architecture.

CUDA parallel programming model has a hierarchy of thread groups called *grid*, *block* and *thread*. A single grid is organized by multiple blocks, each of which has equal number of threads. The blocks are allocated to streaming processors such that all threads in a block are executed by the same streaming processor in parallel. All threads can access to the global memory. However, as we can see in Figure 6, threads in a block can access to the shared memory of the streaming processor to which the block is allocated. Since blocks are arranged to multiple streaming processors, threads in different blocks cannot share data in shared memories.

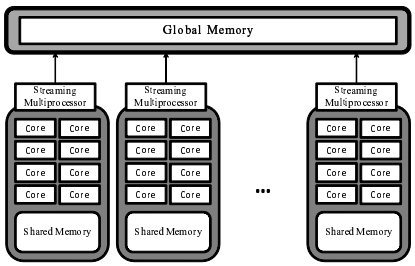


Fig. 6 CUDA hardware architecture

CUDA C extends C language by allowing the programmer to define C functions, called *kernels*. By invoking a kernel, all blocks in the grid are allocated in streaming processors, and threads in each block are executed by processor cores in a single streaming processor. In the execution, threads in a block are split into groups of thread called *warps*. Each of these warps contains the same number of threads and is executed independently. When a warp is selected for execution, all threads execute the same instruction. Any flow control instruction can significantly impact the effective instruction throughput by causing threads of the same warp to diverge, that is, to follow different execution paths. If this happens, the different execution paths have to be serialized. When all the different execution paths have completed, the threads back to the same execution path.

The kernel calls terminates, when threads in all blocks finish the computation. Since all threads in a single block are executed by a single streaming processor, the barrier synchronization of them can be done by calling CUDA C `syncthreds()` function. However, there is no direct way to synchronize threads in different blocks. One of the indirect methods of inter-block barrier synchronization is to partition the computation into kernels. Since continuous kernel calls can be executed such that a kernel is called after all blocks of the previous kernel terminates, execution of blocks is synchronized at the end of kernel calls. Thus, we arrange a single kernel call to each of $n-1$ stages of the dynamic programming algorithm for the optimal polygon triangulation problem.

As we have mentioned, coalesced access to the global memory is a key issue to accelerate the computation. As illustrated in Figure 7, when threads access to continuous locations in a row of a two-dimensional array (*horizontal access*), the continuous locations in address space of the global memory are accessed in the same time (*coalesced access*). However, if threads access to continuous locations in a column (*vertical access*), the distant locations are accessed in the same time (*stride access*). From the structure of the global memory, coalesced access maximizes the bandwidth of memory access. On the other hand, the stride access needs a lot of clock cycles. Thus, we should avoid the stride access (or the vertical access) and perform coalesced access (or the horizontal access) whenever possible. We note that since all threads in a warp execute the same

instruction, memory access by threads in the same warp is performed at the same timing. Therefore, the threads in a warp can access continuous memories, that is, coalesced access can be performed by appropriately addressing.

4. Our implementation of the dynamic programming approach for the optimal polygon triangulation

The main purpose of this section is to show our implementation of dynamic programming for the optimal polygon triangulation in the GPU. We focus on our new ideas, adaptive granularity and sliding and mirroring arrangements for accelerating the dynamic programming algorithm.

4.1 Adaptive granularity

Recall that each Stage p ($2 \leq p \leq n-2$) consists of the computation of $(n-p-1)$ $M_{i,j}$'s, each of which is computed by the sum-minimum of p pairs. To compute the value of $M_{i,j}$'s, we consider two methods *thread-based* (TB) and *block-based* (BB).

In the TB method, one thread is used to compute the value of each of the $(n-p-1)$ $M_{i,j}$'s. More specifically, one thread computes the sum-minimum of p pairs by an obvious sequential computation. Clearly, we use $n-p-1$ threads if the TB method is used for Stage p. Also, if we arrange t threads in a block, $\frac{n-p-1}{t}$ blocks are used. For later reference, we write TB(t) if we use t threads for each block.

In the BB approach, one or more blocks are used to compute the value of each $M_{i,j}$. We write BB(t,b) if we use b blocks with t threads to compute each $M_{i,j}$. Clearly, BB(t,b) uses $(n-p-1)b$ blocks for the computation of Stage p. Since we use b blocks with t threads each to compute $M_{i,j}$, we arrange $\frac{p}{b}$ pairs to each block, which is responsible for computing their sum-minimum. After that, the minimum of the b resulting sum-minimum values are computed.

Let us explain how we compute the value of $\min_{i\leq k\leq i+p-1}(M_{i,k}+M_{k+1,i+p})$ using the BB method. In this method each block uses $\frac{t}{2}$ words in the shared memory for communication between threads. For simplicity, let a_{i-k} and b_{i-k} denote $M_{i,k}$ and $M_{k+1,i+p}$, and will show how we compute $\min_{0\leq i\leq p-1}(a_i+b_i)$. We first show the computation of $\min_{0\leq i\leq p-1}(a_i+b_i)$ by BB$(t,1)$. For simplicity, we assume that p is a multiple of t. First, t threads read $a_0, a_1, \ldots, a_{t-1}$ and $b_0, b_1, \ldots, b_{t-1}$, and compute $a_0+b_0, a_1+b_1, \ldots, a_{t-1}+b_{t-1}$. Next, they read $a_t, a_{t+1}, \ldots, a_{2t-1}$ and $b_t, b_{t+1}, \ldots, b_{2t-1}$, and compute $a_t+b_t, a_{t+1}+b_{t+1}, \ldots, a_{2t-1}+b_{2t-1}$. By repeating the same procedure, the i-th thread ($0 \leq i \leq p-1$) computes $m_i = \min_{0\leq j\leq \frac{p}{t}-1} a_{j\cdot t+i} + b_{j\cdot t+i}$. Finally, we compute $m_0, m_1, \ldots m_{t-1}$ by the binary reduction technique used for computing the sum [16]–[18] as follows. The threads partitioned into two groups of $\frac{t}{2}$ threads each. The $\frac{t}{2}$ threads in the second group sends $m_{\frac{t}{2}}, m_{\frac{t}{2}+1}, \ldots, m_{t-1}$ to the first group via the shared memory. The $\frac{t}{2}$ threads in the first group compute $\min(m_0, m_{\frac{t}{2}}), \min(m_1, m_{\frac{t}{2}+1}), \ldots, \min(m_{\frac{t}{2}-1}, m_{t-1})$. The same procedure is recursively executed for the first $\frac{t}{2}$ threads until the first thread computes $\min_{0\leq k\leq \frac{t}{2}} m_t$. Clearly, this value is equal to $\min_{0\leq i\leq p-1}(a_i+b_i)$. Figure 8 illustrates how the algorithm works.

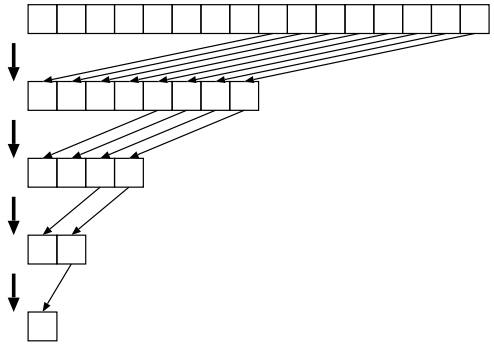


Fig. 8 Illustrating the algorithm for the binary reduction

If more than one block is used, the computation of $\min_{0\leq i\leq p-1}(a_i+b_i)$ is equally partitioned and assigned to each block. Each block compute the sum-minimum independently, and the minimum of them is computed by the atomicMin function supported by CUDA [10]. More specifically, the first thread of each of b block writes the minimum value to the same address of the global memory using atomicMin() function. After all atomic writing operation are terminated, we can obtain the minimum of the written values by b threads.

4.2 Sliding and mirroring arrangement

Let us first observe *the naive arrangement* which allocates each $M_{i,j}$ to the (i,j) element of the 2-dimensional array, that is, the element in the i-th row and the j-th column. As illustrated in Figure 5, to compute $M_{i,i+p}$ in Stage p

- p working data $M_{i,i}, M_{i,i+1}, \ldots, M_{i,i+p-1}$ in the same row and
- p working data $M_{i+1,i+p}, M_{i+2,i+p}, \ldots, M_{i+p,i+p}$ in the same column

are accessed. Hence, the naive arrangement involves the vertical access (or the stride access), which decelerates the computing time.

For coalesced access of the global memory, we present two arrangements of $M_{i,j}$s in a 2-dimensional array, *the sliding arrangement* and *the mirroring arrangement* as follows:

Sliding arrangement: Each $M_{i,j}$ ($0 \leq i \leq j \leq n-1$) is allocated to $(i-j+n, j)$ element of the 2-dimensional array of size $n \times n$.

Mirroring arrangement Each $M_{i,j}$ ($0 \leq i \leq j \leq n-1$) is allocated to (i,j) element and (j,i) element.

The reader should refer to Figure 9 for illustrating the sliding and mirroring arrangements. We will use the sliding

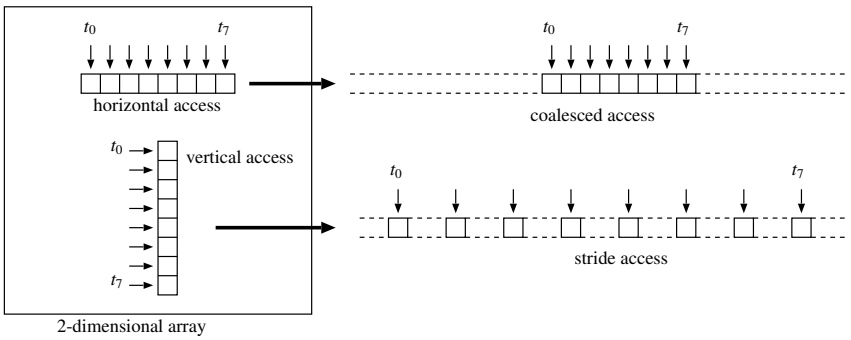


Fig. 7 Coalesced and stride access

					$M_{0,5}$
				$M_{0,4}$	$M_{1,5}$
			$M_{0,3}$	$M_{1,4}$	$M_{2,5}$
		$M_{0,2}$	$M_{1,3}$	$M_{2,4}$	$M_{3,5}$
	$M_{0,1}$	$M_{1,2}$	$M_{2,3}$	$M_{3,4}$	$M_{4,5}$
$M_{0,0}$	$M_{1,1}$	$M_{2,2}$	$M_{3,3}$	$M_{4,4}$	$M_{5,5}$

Sliding arrangement

$M_{0,0}$	$M_{0,1}$	$M_{0,2}$	$M_{0,3}$	$M_{0,4}$	$M_{0,5}$
$M_{0,1}$	$M_{1,1}$	$M_{1,2}$	$M_{1,3}$	$M_{1,4}$	$M_{1,5}$
$M_{0,2}$	$M_{1,2}$	$M_{2,2}$	$M_{2,3}$	$M_{2,4}$	$M_{2,5}$
$M_{0,3}$	$M_{1,3}$	$M_{2,3}$	$M_{3,3}$	$M_{3,4}$	$M_{3,5}$
$M_{0,4}$	$M_{1,4}$	$M_{2,4}$	$M_{3,4}$	$M_{4,4}$	$M_{4,5}$
$M_{0,5}$	$M_{1,5}$	$M_{2,5}$	$M_{3,5}$	$M_{4,5}$	$M_{5,5}$

Mirroring arrangement

Fig. 9 Sliding and Mirroring arrangements

arrangement for TB and the mirroring arrangement for BB.

We will show that the vertical access can be avoided if we use the sliding arrangement for the TB. Suppose that each thread i computes the value of $M_{i,i+p}$. First, each thread i reads $M_{i,i}$ in parallel and then read $M_{i+1,i+p}$ in parallel. Thus, $M_{0,0}, M_{1,1}, \ldots$ are read in parallel and then $M_{1,1+p}, M_{2,2+p}, \ldots$ are read in parallel. Clearly, $M_{0,0}, M_{1,1}, \ldots$ are in the same row of the sliding arrangement. Also, $M_{1,1+p}, M_{2,2+p}, \ldots$ are also in the same row. Thus, coalesced read is performed. Similarly, we can confirm that the remaining read operations by multiple threads perform coalesced read.

Next, we will show that the vertical access can be avoided if we use the mirroring arrangement for the BB. Suppose that a block computes the value of $M_{i,i+p}$. Threads in the block read $M_{i,i}, M_{i,i+1}, \ldots, M_{i,i+p-1}$ in parallel, and then read $M_{i+1,i+p}, M_{i+2,i+p}, \ldots, M_{i+p,i+p}$ in parallel. Clearly, $M_{i,i}, M_{i,i+1}, \ldots, M_{i,i+p-1}$ are stored in $(i, i), (i, i+1), \ldots, (i, i+p-1)$ elements in the 2-dimensional array of the mirroring arrangement, and thus, threads perform coalesced read. For coalesced read, threads read $M_{i+1,i+p}, M_{i+2,i+p}, \ldots, M_{i+p,i+p}$ stored in $(i+p, i+1), (i+p, i+2), \ldots, (i+p, i+p)$ elements in the 2-dimensional array of the mirroring arrangement. Clearly, these elements are in the same row and the threads perform coalesced read.

4.3 Our algorithm for the optimal polygon triangulation

Our algorithm for the optimal polygon triangulation is designed as follows: For each Stage p ($2 \leq p \leq n-2$), we execute two methods, TB(t) and BB(t, b) for various values of t and b, and find the fastest method and parameters. As we are going to show later, the TB method is faster in earlier stages and the BB method is faster in the remaining stages. Thus, we first use the sliding arrangement in earlier stages computed by the TB method. We then convert the 2-dimensional array with the sliding arrangement into the mirroring arrangement. After that, we execute the BB method in the remaining stages. Note that the computing time of our algorithm depends only on the number of vertices, i.e., it is independent from the weights of edges. Therefore, given the number of vertices, we can find and determine the best method with optimal parameters.

5. Experimental results

We have implemented our dynamic programming algorithm for the optimal polygon triangulation using CUDA C. We have used NVIDIA GeForce GTX 680 with 1536 processing cores (8 Streaming Multiprocessors which have 192 processing cores) running in 1.058GHz and 2GB memory. For the purpose of estimating the speedup of our GPU implementation, we have also implemented a conventional software approach of dynamic programming for the optimal polygon triangulation using GNU C. We have used Intel Core i7 870 running in 2.93GHz and 8GB memory to run the sequential algorithm for dynamic programming.

Table 1 shows the computing time in seconds for an

8192-gon. Table 1 (a) shows the computing time of TB(t) for t = 32, 64, 128, 256, 512, 1024. The computing time is evaluated for the naive arrangement and the sliding arrangement. For example, if we execute TB(32) for all stages on the naive arrangement, the computing time is 113.14 seconds. TB(64) runs in 13.92 seconds on the sliding arrangement and thus, the sliding arrangement can attain a speedup of factor 8.13.

Table 1 (b) shows the computing time of BB(b,t) for $b = 1, 2, 4, 8$ and $t = 32, 64, 128, 256, 512, 1024$. Again, let us select b and t that minimize the computing time. BB(1, 32) takes 28.33 seconds for the naive arrangement and BB(1, 256) runs in 5.94 seconds for the mirroring arrangement. Thus, the mirroring arrangement can attain a speedup of factor 4.77.

For each of the two methods and for each of the stages, we select best values of the number t of threads in each block and the number b of blocks. Also, the sliding arrangement is used for TB and the mirroring arrangement is used for BB. Recall that we can use different methods with different parameters for each stage independently. To attain the minimum computing time, we should use two methods shown in Table 2.

Table 2 The optimal combination of two methods for different size of polygons

n	TB	BB
128	—	Stages 0-126
256	Stages 0-7	Stages 8-254
512	Stages 0-19	Stages 20-510
1024	Stages 0-43	Stages 44-1022
2048	Stages 0-82	Stages 83-2046
4096	Stages 0-199	Stages 200-4094
8192	Stages 0-923	Stages 924-8190

Table 3 shows the computing time using the optimal combination of two methods. Note that if we use two methods for each stage in this way, we need to convert the sliding arrangement into the mirroring arrangement. The computing time in the table includes the conversion time if the conversion is necessary. This conversion takes only 7.80 mseconds for an 8192-gon. Readers can find that the conversion time is much smaller than the total execution time. Also, smaller polygons denote the same tendency of it. For an 8192-gon, the best total computing time of our implementation for the optimal polygon triangulation problem is 5.57 seconds. The sequential implementation used Intel Core i7 870 runs in 1939.02 seconds. Thus, our best GPU implementation attains a speedup factor of 348.02. Recall that the computing time does not depend on edge weights shown in the above section. Therefore, for another polygon whose weights are different, we can obtain almost the same speedup factor as that of the above experiment.

We note that since the timing of change of the methods depends on the number of sides of a polygon and the execution environment, if the number of sides of a polygon is known, once the methods are executed for each stage, the appropriate timing can be determined by the running time.

6. Concluding remarks

In this paper, we have proposed an implementation of the dynamic programming algorithm for an optimal polygon triangulation on the GPU. Our implementation selects the best methods, parameters, and data arrangement for each stage to obtain the best performance. The experimental results show that our implementation solves the optimal polygon triangulation problem for a convex 8192-gon in 5.57 seconds on the NVIDIA GeForce GTX 680, while a conventional CPU implementation runs in 1939.02 seconds. Thus, our GPU implementation attains a speedup factor of 348.02.

In the future work, we plan to apply our proposed approach to other problems of dynamic programming. Our proposed approach basically can be applied to other problems that utilize memorization such as the matrix-chain multiplication problem, the longest common subsequence problem, etc [1]. Applying our idea to them, optimal granularity of parallelism should be obtained. However, since memory access pattern and utilized memory size differ for each problem, in order to obtain the further acceleration, it is necessary to consider programming issues of the GPU system as mentioned in Section 3.

Also, we plan to apply our proposed method to *the algebraic dynamic programming* (ADP) [19]. ADP is a new technique for dynamic programming. Compared to the usual style of dynamic programming, ADP provides a higher level of abstraction, helping to solve more sophisticated problems. Therefore, applying our idea to the ADP, we will accelerate the computation of the various problems based on the ADP.

Acknowledgement

The authors would like to thank to Mr. Kazufumi Nishida, who has developed a preliminary version of CUDA C programs of this work.

References

[1] T.H. Cormen, C.E. Leiserson, and R.L. Rivest, Introduction to Algorithms, first ed., MIT Press, 1990.
[2] L. Bergroth, H. Hakonen, and T. T. Raita, "A survey of longest common subsequence algorithms," Proc. of International Symposium on String Processing and Information Retrieval, 2000.
[3] W.W. Hwu, GPU Computing Gems Emerald Edition, Morgan Kaufmann, 2011.
[4] Y. Ito, K. Ogawa, and K. Nakano, "Fast ellipse detection algorithm using Hough transform on the GPU," Proc. of International Conference on Networking and Computing, pp.313–319, Dec. 2011.
[5] D. Man, K. Uda, Y. Ito, and K. Nakano, "A GPU implementation of computing Euclidean distance map with efficient memory access," Proc. of International Conference on Networking and Computing, pp.68–76, Dec. 2011.
[6] K. Ogawa, Y. Ito, and K. Nakano, "Efficient Canny edge detection using a GPU," International Workshop on Advances in Networking and Computing, pp.279–280, Nov. 2010.
[7] A. Uchida, Y. Ito, and K. Nakano, "Fast and accurate template

Table 1 The computing time (seconds) for an 8192-gon using each of the two methods

(a) The computing time of TB(t)

t	32	64	128	256	512	1024
naive arrangement	113.14	116.67	117.07	118.39	130.61	167.34
sliding arrangement	16.88	13.92	13.96	13.96	14.04	14.31

(b) The computing time of BB(b, t)

t		32	64	128	256	512	1024
naive arrangement	$b = 1$	28.33	45.27	68.02	85.48	92.13	102.41
	$b = 2$	34.15	47.79	71.42	89.37	122.08	179.62
	$b = 4$	51.41	67.17	77.92	98.71	137.35	242.05
	$b = 8$	56.84	80.97	83.42	109.61	155.55	278.01
mirroring arrangement	$b = 1$	10.70	6.64	6.98	5.94	6.38	9.92
	$b = 2$	10.89	6.88	6.45	6.17	8.37	16.25
	$b = 4$	11.68	7.59	6.19	7.90	13.07	27.92
	$b = 8$	12.98	8.94	7.38	12.02	23.02	53.52

Table 3 The computing time of the optimal combination of two methods for different size of polygons

n	128	256	512	1024	2048	4096	8192
CPU	4.42 [ms]	34.92 [ms]	279.42 [ms]	2397.80 [ms]	20206.62 [ms]	211.61 [s]	1939.02 [s]
GPU	0.56 [ms]	1.50 [ms]	5.04 [ms]	22.00 [ms]	111.28 [ms]	0.73 [s]	5.57 [s]
Speed-up	7.42	23.33	55.49	108.19	181.58	289.24	348.02

matching using pixel rearrangement on the GPU," Proc. of International Conference on Networking and Computing, pp.153–159, Dec. 2011.

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

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

[10] NVIDIA Corporation, "NVIDIA CUDA C programming guide version 4.2," 2012.

[11] D. Man, K. Uda, H. Ueyama, Y. Ito, and K. Nakano, "Implementations of a parallel algorithm for computing Euclidean distance map in multicore processors and GPUs," International Journal of Networking and Computing, vol.1, pp.260–276, July 2011.

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

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

[14] G. Pólya, "On picture-writing," The American Mathematical Monthly, vol.63, no.10, Dec. 1956.

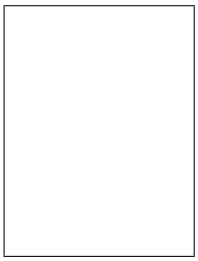
[15] NVIDIA Corporation, "NVIDIA CUDA C best practice guide version 4.1," 2012.

[16] M. Harris, S. Sengupta, and J.D. Owens, "Chapter 39. parallel prefix sum (scan) with CUDA," in GPU Gems 3, Addison-Wesley, 2007.

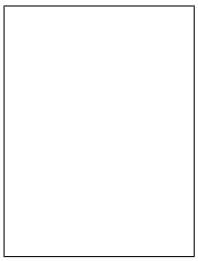
[17] K. Nakano, "An optimal parallel prefix-sums algorithm on the memory machine models for GPUs," Proc. of International Conference on Algorithms and Architectures for Parallel Processing (ICA3PP, LNCS 7439), pp.99–113, Sept. 2012.

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

[19] R. Giegerich and C. Meyer, "Algebraic dynamic programming," Algebraic Methodology and Software Technology, pp.349–364, Springer, 2002.



Yasuaki Ito received the B.E. degree from Nagoya Institute of Technology (Japan) in 2001, the M.S. degree from Japan Advanced Institute of Science and Technology in 2003, and the D.E. degree from Hiroshima University (Japan), in 2010. From 2004 to 2007 he was a Research Associate at Hiroshima University. Since 2007, Dr. Ito has been with the School of Engineering, at Hiroshima University, where he is working as an Assistant Professor. His research interests include reconfigurable architectures, parallel computing, computational complexity and image processing.



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