Accelerating the Dynamic Programming for the Optimal Polygon Triangulation on the GPU

Kazufumi Nishida, Koji Nakano, and Yasuaki Ito

Department of Information Engineering, Hiroshima University,
Kagamiyama 1-4-1, Higashi Hiroshima 739-8527, Japan
{nishida,nakano,yasuaki}@cs.hiroshima-u.ac.jp

Abstract. Modern GPUs (Graphics Processing Units) 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). 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 can be solved using the dynamic programming technique in $O(n^3)$ time using a work space of size $O(n^2)$. The main contribution of this paper is to present 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 (granularity adjustment) is to partition the dynamic programming algorithm into many sequential kernel calls of CUDA, and to select the best size and number of blocks and threads for each kernel call. The second idea (sliding and mirroring arrangements) is to arrange the interim 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 16384-gon in 69.1 seconds on the NVIDIA GeForce GTX 580, while a conventional CPU implementation runs in 17105.5 seconds. Thus, our GPU implementation attains a speedup factor of 247.5.

Keywords: Dynamic programming, parallel algorithms, coalesced memory access, GPUGPU, CUDA

1 Introduction

The GPU (Graphical Processing Unit), is a specialized circuit designed to accelerate computation for building and manipulating images [4, 5, 7, 13]. 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 [4, 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 [8], since they have hundreds of processor cores running in parallel.

The dynamic programming is an important algorithmic technique to find an optimal solution of a problem over an exponential number of solution candidates [2]. 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. 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 [1]. 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, 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 [2].

The main contribution of this paper is to implement the dynamic programming to solve *the optimal polygon triangulation problem* [2] 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 [2, 3, 6] 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. 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 (*granularity adjustment*). The dynamic programming algorithm for an n-gon has $n-1$ stages, each of which involves the computation of multiple interim data. Earlier stages of the algorithm are *fine grain* in the sense that we need to compute the values of a lot of interim data but the computation of each interim data is light. On the other hand, later stages of the algorithm are *coarse grain* in the sense that

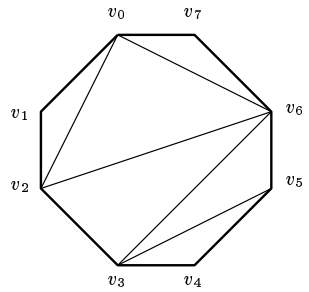

Fig. 1. An example of a triangulation of a convex 8-gon

few interim data are computed but the computation is heavy. Thus, in earlier stages, a single thread is assigned to the computation of each interim data and its value is computed sequentially by the thread (*OneThreadPerEntry*). In middle stages, a block with multiple threads is allocated to the computation for each interim data and the value of the interim data is computed by threads of a block in parallel (*OneBlockPerEntry*). Multiple blocks are allocated to compute each interim data in later stages (*BlocksPerEntry*). Also, the size of each block (i.e. the number of threads), and the number of used blocks affects the performance of algorithms on the GPU. We have tested all of the three methods for various sizes of each block and the number of blocks for every stage, and determined the best way, one of the three methods and the size and the number of blocks for computing the interim data in each stage.

The second innovative idea is to arrange interim data in two dimensional array of the global memory using two types of arrangements: *sliding arrangement* and *mirroring arrangement*. The interim data used in the dynamic programming are stored in a two dimensional array in the global memory of the GPU. The bandwidth of the global memory is maximized when threads repeatedly performs 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 accessed to distant locations in the same time, these access requests need a lot of clock cycles. We use the sliding arrangement for OneThreadPerEntry and the mirroring arrangement for OneBlockPerEntry and BlocksPerEntry. Using these two arrangements, the coalesced access is performed for the interim data.

Our implementation using these two ideas solves the optimal polygon triangulation problem for a convex 16384-gon in 69.1 seconds on the NVIDIA GeForce GTX 580, while a conventional CPU implementation runs in 17105.5 seconds. Thus, our GPU implementation attains a speedup factor of 247.5.

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 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

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 [2].

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 $w_{i,j}$ 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. For this purpose, we define *the parse tree* of a triangulation. Figure 2 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 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 $\frac{(2n)!}{(n+1)!n!}$[12]. 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 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 $(j-i)$-gon with vertices $v_{i-1}, v_i, \ldots, v_j$ as illustrated in Figure 3. Clearly, this $(j-i)$-gon consists of leaves

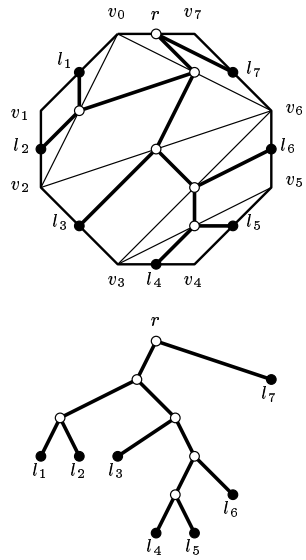

Fig. 2. The parse tree of a triangulation

$l_i, l_{i+1}, \ldots, l_j$ and a chord $v_{i-1}v_j$. Let $m_{i,j}$ be the minimum weight of the $(j-i)$-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 the n-gon is equal to $m_{1,n-1}$.

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 the n-gon.

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

Stage 0 $M_{1,1} = M_{2,2} = \cdots = M_{n-1,n-1} = 0$.

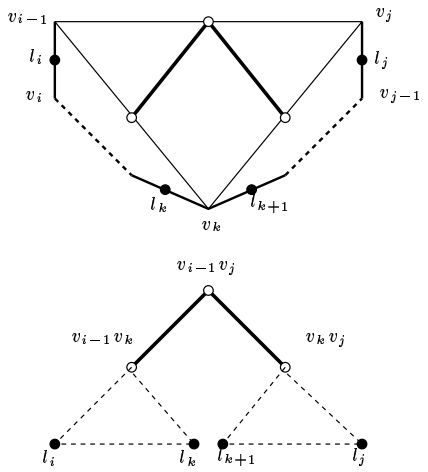

Fig. 3. A $(j-i)$-gon is partitioned into a $(k-i)$-gon and a $(j-k)$-gon

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. 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 over 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.

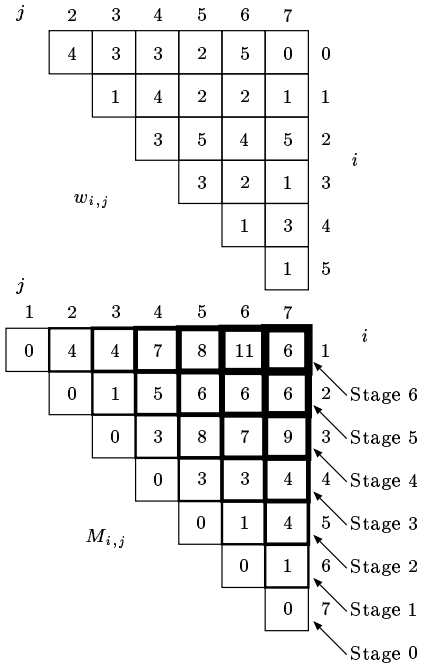

Fig. 4. Examples of $w_{i,j}$ and $M_{i,j}$

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 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 [11, 7, 8]. 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 5 illustrates the CUDA hardware architecture.

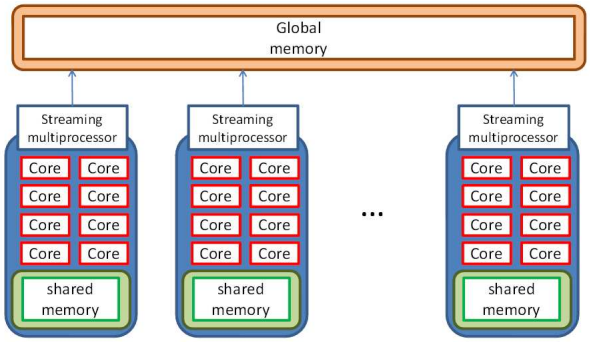

Fig. 5. CUDA hardware architecture

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 5, 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.

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. 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, the coalesced access to the global memory is a key issue to accelerate the computation. As illustrated in Figure 6, 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, the 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 the coalesced access (or the horizontal access) whenever possible.

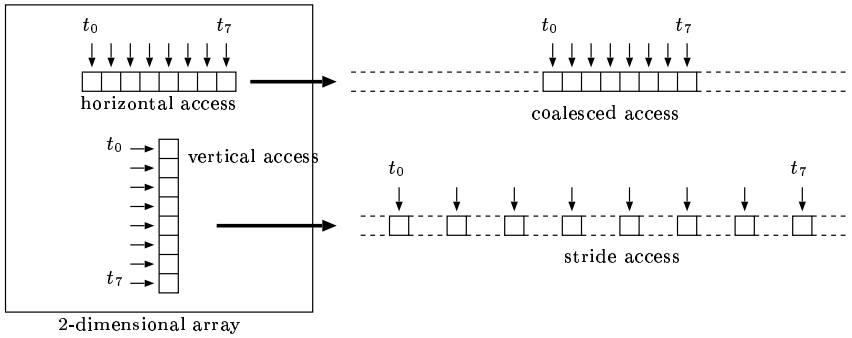

Fig. 6. Coalesced and stride access

4 Our implementation of the dynamic programming 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, granularity adjustment and sliding and mirroring arrangements for accelerating the dynamic programming algorithm.

4.1 Granularity adjustment technique

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 involves the computation of the minimum of p values. We consider three methods, *OneThreadPerEntry*, *OneBlockPerEntry*, and *BlocksPerEntry* to perform the computation of each of the $n-2$ stages. In OneThreadPerEntry, each $M_{i,i+p}$ is computed sequentially by one thread. In OneBlockPerEntry, each $M_{i,i+p}$ is computed by one block with multiple threads in parallel. In BlocksPerEntry, each $M_{i,i+p}$ is computed by multiple blocks in parallel.

Let t be the number of threads in each block and b be the number of blocks. In our implementation of the three methods, t and b can be the parameters that can be changed to get the best performance. The details of the implementation of the three methods are spelled out as follows:

OneThreadPerEntry(t): Each $M_{i,i+p}$ is computed by a single thread sequentially. Thus, we use $(n-p-1)$ threads totally. Since each block has t threads, $\frac{n-p-1}{t}$ blocks are used.

OneBlockPerEntry(t): Each $M_{i,i+p}$ is computed by a block with t threads. The computation of $M_{i,i+p}$ involves the p sums $M_{i,k} + M_{i+p,k+1}$ ($i \leq k \leq i+p-1$). The t threads compute p sums in parallel such that each thread computes $\frac{p}{t}$ sums and their local minimum of the $\frac{p}{t}$ sums is computed. The resulting local t minima are written into the shared memory. After that, a single thread is used to compute the minimum of the t local minima.

BlocksPerEntry(b,t): Each $M_{i,i+p}$ is computed by b blocks with t threads each. The computation of p sums are arranged b blocks equally. Thus, each block computes the $\frac{p}{b}$ sums and their minimum is computed in the same way as OneBlockPerEntry(t). The resulting b minima are written to the global memory. The minimum of the b minima is computed by a single thread.

For each Stage p ($2 \leq p \leq n-2$), we can choose one of the three methods OneThreadPerEntry(t), OneBlockPerEntry(t), and BlocksPerEntry(b,t), independently.

4.2 Sliding and mirroring arrangement

Recall that, each Stage p ($2 \leq p \leq n-2$) of the dynamic programming involves the computation

$$M_{i,i+p} = \min_{i \leq k \leq i+p-1} (M_{i,k} + M_{k+1,i+p}) + w_{i-1,i+p}.$$

Let us first observe *the naive arrangement* which allocates each $M_{i,j}$ to the (i,j) element of the two dimensional array, that is, the element in the i-th row and the j-th column. As illustrated in Figure 7, to compute $M_{i,i+p}$ in Stage p

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

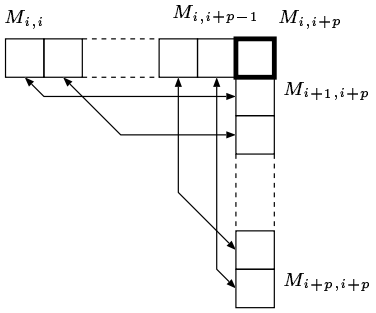

Fig. 7. The computation of $M_{i,i+p}$

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

For the coalesced access of the global memory, we present two arrangements of $M_{i,j}$s in a two 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 two 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 8 for illustrating the sliding and mirroring arrangements. We will use sliding arrangement for OneThreadPerEntry and the mirroring arrangement for OneBlockPerEntry and BlocksPerEntry.

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

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. 8. Sliding and Mirroring arrangements

We will show that the vertical access can be avoided if we use the sliding arrangement for OneThreadPerEntry. 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, the coalesced read is performed. Similarly, we can confirm that the remaining read operations by multiple threads perform the coalesced read.

Next, we will show that the vertical access can be avoided if we use the mirroring arrangement for OneBlockPerEntry and BlocksPerEntry. 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 two dimensional array of the mirroring arrangement, thus, the treads perform the coalesced read. For the 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 two dimensional array of the mirroring arrangement. Clearly, these elements are in the same row and the threads perform the 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 three methods OneThreadPerEntry(t), OneBlockPerEntry(t), and BlocksPerEntry(b,t) for various values of t and b, and find the fastest method and parameters. As we are going to show later, OneThreadPerEntry is the fastest in earlier stages. In middle stages, OneBlockPerEntry is fastest. Finally, BlocksPerEntry is the best in later stages. Thus, we first use the sliding arrangement in earlier stages computed by OneThreadPerEntry. We then convert the two dimensional array with the sliding arrangement into the mirroring arrangement. After that, we execute OneBlockPerEntry and then BlocksPerEntry in the remaining stages.

5 Experimental results

We have implemented our dynamic programming algorithm for the optimal polygon triangulation using CUDA C. We have used NVIDIA GeForce GTX 580 with 512 processing cores (16 Streaming Multiprocessors which has 32 processing cores) running in 1.544GHz and 3GB memory. For the purpose of estimating the speedup of our GPU implementation, we have also implemented a conventional software approach of the 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 the dynamic programming.

Table 1 shows the computing time in seconds for a 16384-gon. Table 1 (a) shows the computing time of OneThreadPerEntry(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 OneThreadPerEntry(64) for all stages

on the naive arrangement, the computing time is 854.8 seconds. OneThreadPerEntry(64) runs in 431.8 seconds on the sliding arrangement and thus, the sliding arrangement can attain a speedup of factor 1.98.

Table 1 (b) shows the computing time of OneBlockPerEntry(t) for $t = 32, 64, 128, 256, 512, 1024$. Let us select t that minimizes the computing time. OneBlockPerEntry(128) takes 604.7 seconds for the naive arrangement and OneBlockPerEntry(128) runs in 73.5 seconds for the mirroring arrangement. Thus, the mirroring arrangement can attain a speedup of factor 8.23.

Table 1 (c) shows the computing time of BlocksPerEntry(b, t) for $b = 2, 4, 8$ and $t = 32, 64, 128, 256, 512, 1024$. Again, let us select b and t that minimize the computing time. BlocksPerEntry(2,128) takes 610.9 seconds for the naive arrangement and BlocksPerEntry(2,128) runs in 97.8 seconds for the mirroring arrangement. Thus, the mirroring arrangement can attain a speedup of factor 6.25.

Table 1. The computing time (seconds) for a 16384-gon using each of the three methods

(a) The computing time of OneThreadPerEntry(t)

t	32	64	128	256	512	1024
naive arrangement	596.8	854.8	863.3	889.2	1202.0	1614.2
sliding arrangement	312.8	431.8	442.2	541.0	668.3	1023.2

(b) The computing time of OneBlockPerEntry(t)

t	32	64	128	256	512	1024
naive arrangement	631.8	606.8	604.7	612.3	678.7	1286.5
mirroring arrangement	169.5	98.5	73.5	80.4	225.0	824.8

(c) The computing time of BlocksPerEntry(b, t)

t		32	64	128	256	512	1024
naive arrangement	$b = 2$	650.2	614.6	610.9	627.3	828.8	2007.8
	$b = 4$	650.5	617.5	624.9	673.1	1174.9	3585.0
	$b = 8$	655.6	630.5	670.0	815.1	1917.8	6779.5
mirroring arrangement	$b = 2$	176.3	110.8	97.8	129.1	422.6	1611.7
	$b = 4$	188.5	136.2	148.2	229.8	820.3	3188.6
	$b = 8$	216.0	189.9	250.5	433.6	1613.7	6337.9

Figure 9 shows the running time of each stage using the three methods. For each of the three methods and for each of the 16382 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 OneThreadPerEntry and the mirroring arrangement is used for OneBlockPerEntry and BlocksPerEntry. Recall that we can use different methods with different parameters can be used for each stage independently. Thus, to attain the minimum computing time we should use

– OneThreadPerEntry for Stages 0-49,
– OneBlockPerEntry for Stages 50-16350, and
– BlocksPerEntry for Stages 16351-16382.

Note that if we use three methods for each stage in this way, we need to convert the sliding arrangement into the mirroring arrangement. This conversion takes only 0.21 mseconds. Including the conversion time, the best total computing time of our implementation for the optimal polygon triangulation problem is 69.1 seconds. The sequential implementation used Intel Core i7 870 runs in 17105.5 seconds. Thus, our best GPU implementation attains a speedup factor of 247.5.

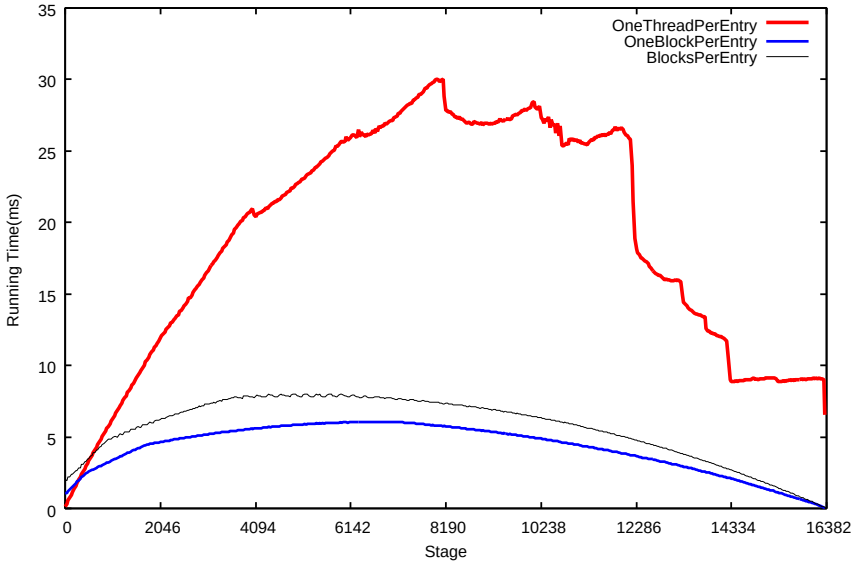

Fig. 9. The running time of each stage using three methods

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 16384-gon in 69.1 seconds on the NVIDIA GeForce GTX 580, while a conventional CPU implementation runs in 17105.5 seconds. Thus, our GPU implementation attains a speedup factor of 247.5.

References

1. Bergroth, L., Hakonen, H., T. Raita, T.: A survey of longest common subsequence algorithms. In: Proc. of International Symposium on String Processing and Information Retrieval (2000)
2. Cormen, T.H., Leiserson, C.E., Rivest, R.L.: Introduction to Algorithms. MIT Press (1990)
3. Gilbert, P.D.: New results on planar Triangulations. In: M.Sc. thesis. pp. Report R–850 (July 1979)
4. Hwu, W.W.: GPU Computing Gems Emerald Edition. Morgan Kaufmann (2011)
5. Ito, Y., Ogawa, K., Nakano, K.: Fast ellipse detection algorithm using Hough transform on the GPU. In: Proc. of International Conference on Networking and Computing. pp. 313–319 (Dec 2011)
6. Klincsek, G.T.: Minimal triangulations of polygonal domains. Annals of Discrete Mathematics 9, 121–123 (July 1980)
7. Man, D., Uda, K., Ito, Y., Nakano, K.: A GPU implementation of computing Euclidean distance map with efficient memory access. In: Proc. of International Conference on Networking and Computing. pp. 68–76 (Dec 2011)
8. Man, D., Uda, K., Ueyama, H., Ito, Y., Nakano, K.: Implementations of a parallel algorithm for computing Euclidean distance map in multicore processors and GPUs. International Journal of Networking and Computing 1(2), 260–276 (July 2011)
9. Nishida, K., Ito, Y., Nakano, K.: Accelerating the dynamic programming for the matrix chain product on the GPU. In: Proc. of International Conference on Networking and Computing. pp. 320–326 (Dec 2011)
10. NVIDIA Corp.: NVIDIA CUDA C Programming Guide Version 4.1 (2011)
11. NVIDIA Corp.: CUDA C Best Practice Guide Version 4.1 (2012)
12. Pólya, G.: On picture-writing. Amer. Math. Monthly 63, 689–697 (1956)
13. Uchida, A., Ito, Y., Nakano, K.: Fast and accurate template matching using pixel rearrangement on the GPU. In: Proc. of International Conference on Networking and Computing. pp. 153–159 (Dec 2011)