Bulk Execution of Oblivious Algorithms on the Unified Memory Machine, with GPU Implementation

Kazuya Tani, Daisuke Takafuji, Koji Nakano, Yasuaki Ito
Department of Information Engineering
Hiroshima University
Kagamiyama 1-4-1, Higashi Hiroshima, 739-8527 Japan

***Abstract*—The Unified Memory Machine (UMM) is a theoretical parallel computing model that captures the essence of the global memory access of GPUs. A sequential algorithm is oblivious if an address accessed at each time does not depend on input data. Many important tasks including matrix computation, signal processing, sorting, dynamic programming, and encryption/decryption can be performed by oblivious sequential algorithms. The bulk execution of a sequential algorithm is to execute it for many different inputs in turn or at the same time. The main contribution of this paper is to show that the bulk execution of an oblivious sequential algorithm can be implemented to run on the UMM very efficiently. More specifically, the bulk execution for p different inputs can be implemented to run $O(\frac{pt}{w} + lt)$ time units using p threads on the UMM with memory width w and memory access latency l, where t is the running time of the oblivious sequential algorithm. We also prove that this implementation is time optimal. Further, we have implemented two oblivious sequential algorithms to compute the prefix-sums of an array of size n and to find the optimal triangulation of a convex n-gon using the dynamic programming technique. The prefix-sum algorithm is a quite simple example of oblivious algorithms, while the optimal triangulation algorithm is rather complicated. The experimental results on GeForce GTX Titan show that our implementations for the bulk execution of these two algorithms can be 150 times faster than that of a single CPU if they have many inputs. This fact implies that our idea for the bulk execution of oblivious sequential algorithms is a potent method to elicit the capability of CUDA-enabled GPUs very easily.**

***Keywords*-Parallel algorithms, oblivious sequential algorithm, 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.

A Graphics Processing Unit (GPU) 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, 16-48 Kbytes. The global memory is implemented as an off-chip DRAM, and thus, it has large capacity, say, 1.5-6 Gbytes, but its access latency is very long. The efficient usage of the shared memory and the global memory is a key for CUDA developers to accelerate applications using GPUs. In particular, we need to consider *the bank conflict* of the shared memory access and *the coalescing* of the global memory access [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 the 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 the 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], [21] 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 [22], [23], [24], [25]. 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 [22]. 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)* [21], 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 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 the memory access in turn, and w threads in a dispatched warp send the 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 the memory access requests are moved to destination memory banks in a pipeline fashion. Note that the DMM and the UMM with width w have 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.

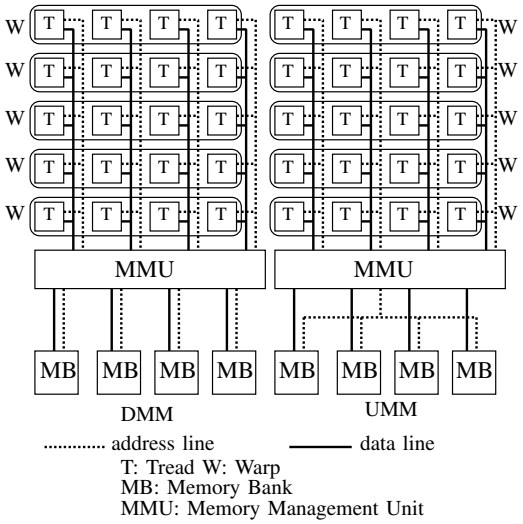


Fig. 1. The architectures of the DMM and the UMM with width $w = 4$

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 at 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, if multiple memory access requests are destined for different address groups, they are processed separately.

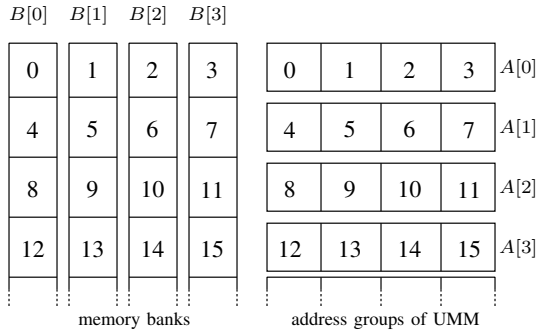


Fig. 2. The memory banks and the address group of the DMM and the UMM for $w = 4$ banks

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 access when we evaluate the performance of algorithms on the DMM and on the UMM. The width w is the number of the 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 [25] 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

Intuitively, a sequential algorithm is *oblivious* if an address accessed at each time unit is independent of the input. For example, the prefix-sums of an array b of size n can be computed by executing $b[i] \leftarrow b[i] + b[i-1]$ for all i $(1 \leq i \leq n-1)$ in turn. This prefix-sum algorithm is oblivious because the address accessed at each time unit is independent of the values stored in b. The readers may think that the oblivious memory access is too restricted, and most useful algorithms are not oblivious. However, many important and complicated tasks including matrix computation, signal processing, sorting, dynamic programming, and encryption/decryption can be performed by oblivious sequential algorithms.

The first contribution of this paper is to introduce the bulk execution of a sequential algorithm. *The bulk execution* of a sequential algorithm is to execute it for many different inputs in turn or at the same time. For example, suppose that we have p arrays $b_0, b_1, \ldots b_{p-1}$ of size n each. We can compute the prefix-sums of each b_j $(0 \leq j \leq p-1)$ by executing the prefix-sum algorithm on a single CPU in turn or on a parallel computer in parallel. The bulk execution has many applications. For example, the conventional FFT algorithm [29] for n points running in $O(n \log n)$ time is oblivious. In practical signal processing, an input stream is equally partitioned into many blocks, and the FFT algorithm is executed for each block in turn or in parallel. This is exactly the bulk execution of the FFT algorithm.

It is very important to avoid the non-coalesced access for high acceleration using the GPU. However, it is not easy to design efficient algorithms that never perform the non-coalesced memory access.

The second contribution of this paper is to show a simple but potent idea to implement algorithms to perform the coalesced memory access to the global memory. More specifically, we show that the bulk execution of an oblivious sequential algorithm can be implemented without performing the non-coalesced memory access. Let t be the running time units of an oblivious sequential algorithm on a single CPU. We show that the bulk execution for p different inputs can be implemented to run $O(\frac{pt}{w} + lt)$ time units using p threads on the UMM with width w and latency l if the inputs are arranged in column-wise. We also prove that any implementation on the UMM takes $\Omega(\frac{pt}{w} + lt)$ time units. Thus, our implementation is time optimal. Further, we have implemented two oblivious sequential algorithms:

- to compute the prefix-sums in $O(n)$ time units on a single CPU, and
- to find the optimal triangulation of a convex n-gon using the dynamic programming technique in $O(n^3)$ time units on a single CPU.

The readers should refer to Figure 3 that shows an example of the optimal triangulation of a convex 8-gon. We selected these two oblivious sequential algorithms because the computation of the prefix-sums are quite simple while the dynamic programming is rather complicated. Our implementations run $O(\frac{pn}{w} + ln)$ time units for the prefix-sums and $O(\frac{pn^3}{w} + ln^3)$ time units for the dynamic programming.

The third contribution of this paper is to implement the bulk execution of these oblivious sequential algorithms on GeForce GTX Titan. The experimental results show that the running

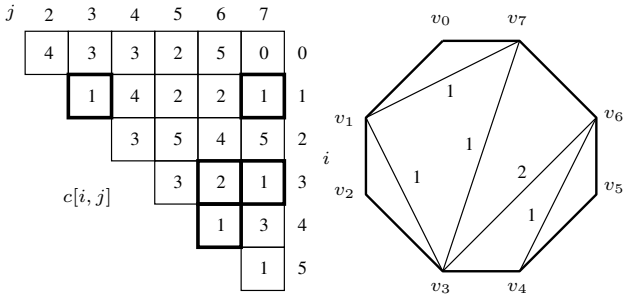


Fig. 3. An example of the optimal triangulation of a convex 8-gon

time on the GPU approximates the theoretical analysis on the GPU. Also, our implementation for the bulk execution for these two algorithm can be more than 150 times faster than the implementation on the convectional CPU. Many researchers have been devoted to implement parallel algorithms on the GPUs [6]. Most efficient implementations use the shared memories on the multistreaming processors of the GPU. More specifically, they first copy a part of input on the global memory to the shared memory, and performs the computation for data on the shared memory. The resulting values are copied from the shared memory to the global memory. This operation is repeated several times. In most implementations, the GPU speedup factor over a single CPU is in the range 10-200. Although our idea for the bulk execution of oblivious sequential algorithms is simple and naive, it can attain a speedup of factor more than 150 in some cases without using the shared memory. Thus, we can say that our idea is a potent method to elicit the capability of CUDA-enabled GPUs very easily.

The rest of this paper is organized as follows: In Section II, we define the Unified Memory Machine (UMM) and the sequential memory access. Section III defines oblivious sequential algorithms and the bulk execution using the prefix-sum algorithm as an example. It also shows that the bulk execution of an oblivious sequential algorithm can be done in $O(\frac{pt}{w} + lt)$ time units on the UMM and prove that it is time optimal. Section IV defines the optimal triangulation problem (OPT problem) and review the dynamic programming for solving this problem. It also shows that the bulk execution can be done in $O(\frac{pn^3}{w} + ln^3)$ time units on the UMM. In Section V, we show experimental results using GeForce GTX Titan. Section VI concludes our work.

II. The Unified Memory Machine (UMM)

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 $m[j \cdot w]$, $m[j \cdot w + 1]$, …, $m[(j + 1) \cdot w - 1]$. 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 Figure 4. 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 the 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 the memory access. If no thread in a warp needs the memory access, such warp is not dispatched for the memory access. When $W(i)$ is dispatched, w threads in $W(i)$ send the memory access requests, one request per thread, to the memory banks.

For the memory access, each warp sends the 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 the 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.

III. Oblivious sequential algorithms and the bulk execution

The main purpose of this section is to introduce oblivious sequential algorithms and the bulk execution of it.

Intuitively, a sequential algorithm is *oblivious* if an address accessed in each time unit is independent of the input. More specifically, there exists a function $a : \{0, 1, \ldots, t-1\} \rightarrow \mathcal{N}$, where t is the running time of the algorithm and $\mathcal{N}$ is a set of all non-negative integers such that, for any input of the algorithm, it accesses address $a(i)$ or does not access the memory at each time i ($0 \leq i \leq t-1$). In other words, at each time i ($0 \leq i \leq t-1$), it never accesses an address other than $a(i)$.

Let us see an example of oblivious algorithms. Suppose that an array b of n integers are given. The prefix-sum computation is a task to store each i-th prefix-sum $b[0] + b[1] + \cdots + b[i]$ in $b[i]$. Let r be a register variable. The following algorithm computes the prefix-sum of n numbers.

[Algorithm Prefix-sums]

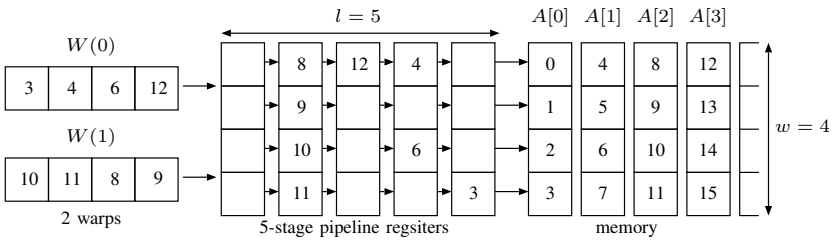


Fig. 4. The UMM with width $w = 4$ and latency $l = 5$

$r \leftarrow 0$
for $i \leftarrow 0$ to $n-1$ do
 $r \leftarrow r + b[i]$
 $b[i] \leftarrow r$

Since $b[0]$, $b[1]$, ..., $b[n-1]$ are added to r in turn, the prefix-sums are stored in b correctly when this algorithm terminates. Let us see the address accessed in each time unit to confirm that this algorithm is oblivious. For simplicity, we ignore access to registers and local computation such as addition and we assume that such operations can be done in zero time unit. Clearly, memory access operations performed in this algorithm are: read $b[0]$, write $b[0]$, read $b[1]$, write $b[1]$, ..., read $b[n-1]$, and write $b[n-1]$. Hence, the memory access function a is $a(2i) = a(2i+1) = i$ for all i $(0 \leq i \leq n-1)$, and thus, this algorithm is oblivious.

Suppose that we need to execute a sequential algorithm for many different inputs on a single CPU in turn or on a parallel machine at the same time. We call such computation *bulk execution*. For example, suppose that we have p arrays $b_0, b_1, \ldots, b_{p-1}$ of size n each on the UMM. The goal of the bulk execution of the prefix-sums is to compute the prefix-sums of every b_j $(0 \leq j \leq p-1)$ on the UMM in parallel. We use p threads and each thread $T(j)$ $(0 \leq j \leq p-1)$ computes the prefix-sums of b_i by Algorithm Prefix-sums. Let r_j $(0 \leq j \leq p-1)$ be a register of thread $T(j)$. The prefix-sums can be computed in parallel by the following algorithm:

[Parallel Algorithm Prefix-sums]
for $j \leftarrow 0$ to $p-1$ do in parallel
 $r_j \leftarrow 0$
 for $i \leftarrow 0$ to $n-1$ do
  $r_j \leftarrow r_j + b_j[i]$
  $b_j[i] \leftarrow r_j$

Let us consider two arrangements of the p arrays of size n each as follows.

row-wise arrangement: They are arranged in a 2-dimensional array with p rows and n columns such that each $b_j[i]$ $(0 \leq i \leq n-1, 0 \leq j \leq p-1)$ is stored in the j-th row and the i-th column, which is allocated in address $j \cdot n + i$.

column-wise arrangement: They are arranged in a 2-dimensional array with n rows and p columns such that each $b_j[i]$ $(0 \leq i \leq n-1, 0 \leq j \leq p-1)$ is stored in the i-th row and the j-th column, which is allocated in address $i \cdot p + j$. The reader should refer to Figure 5 for illustrating row-wise and column-wise arrangements for $p = 4$ arrays of size $n = 6$ each.

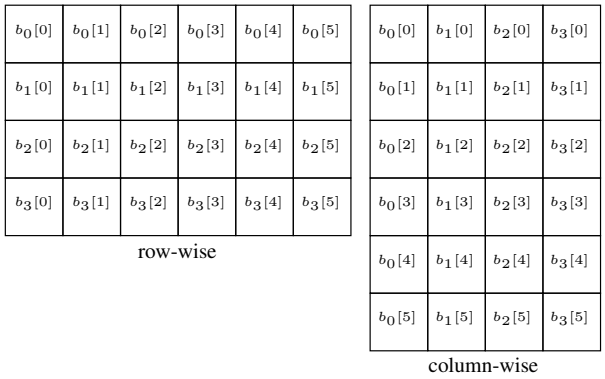


Fig. 5. Row-wise and column-wise arrangements for $p = 4$ arrays of size $n = 6$ each

Let us evaluate the time to execute the prefix-sum algorithm for the row-wise arrangement (row-wise prefix-sums) and column-wise arrangement (column-wise prefix-sums). For simplicity, we assume that p is a multiple of w and n is so large that $n \geq w$. In the row-wise arrangement, for each i $(0 \leq i \leq n-1)$, $b_0[i], b_1[i], \ldots, b_{p-1}[i]$ stored in addresses $i, i+n, \ldots, i+(p-1)n$, are accessed by p threads. They are in p different address groups and corresponding p memory access requests occupy p pipeline stages and thus it takes $p + l - 1$ time units to complete them. Thus, the computation of the row-wise prefix-sums takes $O(p + l - 1) \cdot n = O(np + nl)$ time units. In the column-wise arrangement, for each i $(0 \leq i \leq n-1)$, $b_0[i], b_1[i], \ldots, b_{p-1}[i]$ stored in continuous addresses $i \cdot p, i \cdot p + 1, \ldots, i \cdot p + (p-1)$, are accessed by p threads. They are in $\frac{p}{w}$ address groups. Hence, the computation of the column-wise prefix-sums takes $O(\frac{p}{w} + l - 1) \cdot n = O(\frac{np}{w} + nl)$ time units. Thus, we have

Lemma 1: The row-wise prefix-sums of an array of size $p \times n$ and the column-wise prefix-sums of an array of size $n \times p$ can be computed in $O(np + nl)$ and $O(\frac{np}{w} + nl)$ time units, respectively, using p threads on the UMM with width w and latency l.

We can evaluate the running time of any oblivious algorithm in the same way as Lemma 1. Suppose that we have an oblivious sequential algorithm running t time units. Without loss of generality, the algorithm works on an array of size n. Similarly to the prefix-sum computation above, suppose that the oblivious sequential algorithm is executed for p inputs

of size n each using p threads on the UMM in parallel. We can consider two arrangements, row-wise and column-wise arrangements for the oblivious sequential algorithm. We say that such execution is *a row-wise oblivious computation* if p arrays are arranged in 2-dimensional array of size $p \times n$ such that each row corresponds to an input of the oblivious sequential algorithm. Similarly, it is *a column-wise oblivious computation* if they are arranged in 2-dimensional array of size $n \times p$ such that each column corresponds to an input.

Let us evaluate the computing time on the UMM for the row-wise and the column-wise arrangements as follows. Let $a(j)$ $(0 \leq j \leq t-1)$ denote the address accessed by the oblivious sequential algorithm. In the row-wise arrangement, for each j $(0 \leq j \leq t-1)$, the p threads access $a(j), a(j)+n, \ldots, a(j)+(p-1)n$, which are in p different address groups. Such memory access takes $p+l-1$ time units. Thus, the oblivious algorithm runs $(p+l-1) \cdot t = O(pt+lt)$ time units if we use the row-wise arrangement. In the column-wise arrangement, they access $a(j) \cdot p, a(j) \cdot p+1, \ldots, a(j) \cdot p+(p-1)$, which are in $\frac{p}{w}$ address groups. Since such memory access takes $\frac{p}{w}+l-1$ time units, the oblivious algorithm runs $(\frac{p}{w}+l-1) \cdot t = O(\frac{pt}{w}+lt)$ time units. Thus we have,

Theorem 2: Any row-wise oblivious computation of size $p \times n$ and any column-wise oblivious computation of size $n \times p$ run $O(pt+lt)$ and $O(\frac{pt}{w}+lt)$ time units, respectively, using p threads on the UMM with width w and latency l, where t is the running time of the corresponding oblivious sequential algorithm.

We also prove that column-wise oblivious computation for Theorem 2 is time optimal on the UMM. Since an oblivious algorithm running t time units is executed p times, it may involve pt memory access operations. Since the width of the UMM is w, it takes at least $\frac{pt}{w}$ time units to complete pt memory access operations. Further, since an oblivious algorithm performs t memory access operations in turn, it takes at least lt time units on the UMM. Thus, we have,

Theorem 3: Any implementation of bulk execution of an oblivious algorithm for p inputs takes at least $\Omega(\frac{pt}{w}+lt)$ time units using p threads on the UMM with width w and latency l, where t is the running time of the oblivious sequential algorithm.

Thus, the column-wise oblivious computation for Theorem 2 is time optimal.

IV. 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], [30].

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 3 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

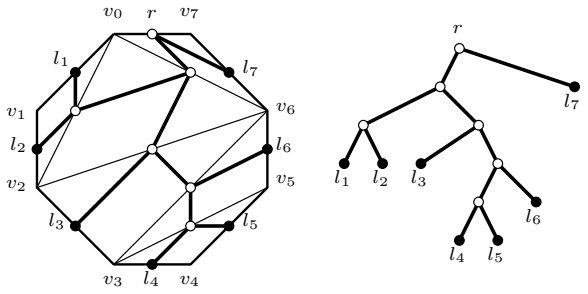


Fig. 6. The parse tree of a triangulation

that a weight $c[i,j]$ of every chord v_iv_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 minimizes the total weight 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 OPT problem asks to find

$$\arg\min_{t \in T} \sum_{v_iv_j \in t} c[i,j].$$

In this paper, for simplicity, we consider just the value of the total weight

$$\min_{t \in T} \sum_{v_iv_j \in t} c[i,j]$$

of the optimal polygon 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 the edge $v_{i-1}v_i$ of a convex n-gon. Also, let r denote the 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 $\frac{(2n)!}{(n+1)!n!}$[31]. 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 a convex n-gon is chopped off by a

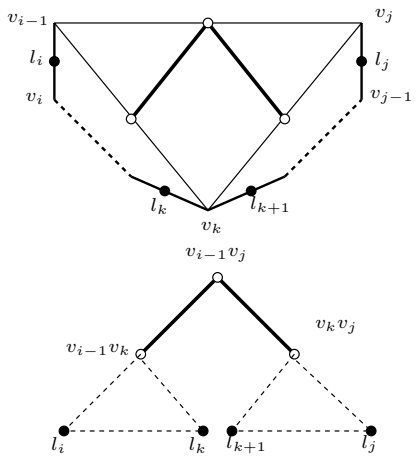


Fig. 7. A $(j-i+2)$-gon is partitioned into a $(k-i+2)$-gon and a $(j-k+1)$-gon

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:

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

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 convex 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:

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

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 convex n-gon. We can evaluate that Figure 8 shows the values of $M[i,j]$'s for a triangle in Figure 3. From the recursive formula, the reader should have no difficulty to confirm that Algorithm OPT below compute the values of all $M[i,j]$'s.

[Algorithm OPT]
for $i \leftarrow 1$ to $n-1$ do
  $M[i,i] \leftarrow 0$
for $i \leftarrow n-2$ downto 1 do
 for $j \leftarrow i+1$ to $n-1$ do
  $s \leftarrow +\infty$
  for $k \leftarrow i$ to $j-1$ do
    $r \leftarrow M[i,k] + M[k+1,j]$
    if $r < s$ then $s \leftarrow r$ else $s \leftarrow s$
  $M[i,j] \leftarrow s + c[i-1,j]$

Note that $s \leftarrow s$ is used to spend the time equal to that for performing $s \leftarrow r$. This redundant operation is necessary to make the algorithm oblivious. Figure 9 illustrates the values of $M[i,j]$ for an 8-gon shown in Figure 3. Clearly, $M[i,j] = \min_{i \leq k \leq j-1}(M[i,k] + M[k+1,j]) + c[i-1,j]$ holds when Algorithm OPT terminates. Thus, Algorithm OPT solves the OPT problem correctly.

j / i	1	2	3	4	5	6	7
1	0	4	4	7	8	11	6
2		0	1	5	6	6	6
3			0	3	8	7	9
4				0	3	3	4
5					0	1	4
6						0	1
7							0

$M[i,j]$

Fig. 8. The resulting values of $M[i,j]$

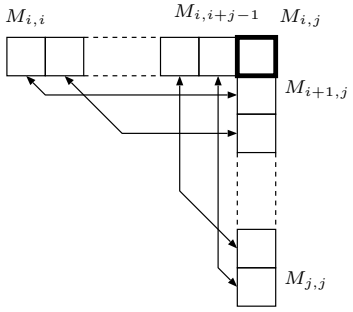


Fig. 9. The computation of $M[i,j]$ by Algorithm OPT

Since Algorithm OPT has a triple for loop using variables i, j and k, it runs $\sum_{i=1}^{n-2}\sum_{j=i+1}^{n-1}\sum_{k=i}^{j-1} O(1) = O(n^3)$ time units. In Algorithm OPT, elements in arrays M and c accessed in each time are independent of the value stored in M and c. Thus, Algorithm OPT is oblivious and we have,

Lemma 4: Oblivious Algorithm OPT runs $O(n^3)$ time units.

Imagine that we execute Algorithm OPT for p convex n-gons using p threads on the UMM in parallel. Similarly to the prefix-sum algorithm, we can consider two arrangements, row-wise arrangement and column-wise arrangement for arrays M and c. The weights of p convex n-gons are stored in p 2-dimensional arrays $c_0, c_1, \ldots, c_{p-1}$ of size $(n-2) \times (n-2)$ each. Also, we use p arrays $M_0, M_1, \ldots, M_{p-1}$ of size $(n-$

$1)\times(n-1)$ each. Although they are two dimensional arrays, we can define the row-wise and column-wise arrangements. For example, Figure 10 illustrates the row-wise and column-wise arrangements of four 2-dimensional arrays M_0, M_1, M_2 and M_3. Each $M_k[i,j]$ is arranged in address $(n-1)^2k+(n-1)i+j$ in the row-wise arrangement. Also, it is arranged in address $(n-1)^2i+(n-1)j+k$ in the column-wise arrangement.

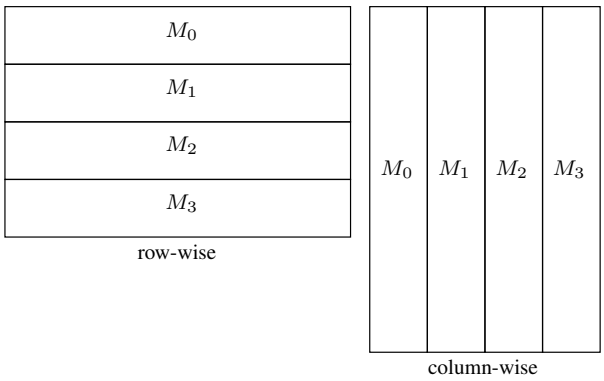


Fig. 10. Row-wise and column-wise arrangements of four 2-dimensional arrays M_0, M_1, M_2 and M_3

Suppose that Algorithm OPT is executed in parallel for the row-wise and the column-wise arrangement. Let r_h and s_h be local registers of thread $T(h)$ $(0 \leq h \leq p-1)$. The following parallel algorithm solves the optimal triangulation problem in parallel.

[Parallel Algorithm OPT]

```
for h ← 0 to p − 1 do in parallel
 for i ← 1 to n − 1 do
   M_h[i, i] ← 0
 for i ← n − 2 downto 1 do
   for j ← i + 1 to n − 1 do
     s_h ← +∞
     for k ← i to j − 1 do
       r_h ← M_h[i, k] + M_h[k + 1, j]
       if r_h < s_h then s_h ← r_h else s_h ← s_h
     M_h[i, j] ← s_h + c_h[i − 1, j]
```

The reader should have no difficulty to confirm that, the memory access by a warp at each time unit for the row-wise arrangement is destined for the distinct memory banks. On the other hand, its memory access for column-wise arrangement is destined for the same memory bank. Hence, Theorem 2 holds for Algorithm OPT. Thus, we have,

Corollary 5: The bulk execution of Algorithm OPT runs in $O(pn^3+ln^3)$ and $O(\frac{pn^3}{w}+ln^3)$ time units, respectively, using p threads on the UMM with width w and latency l.

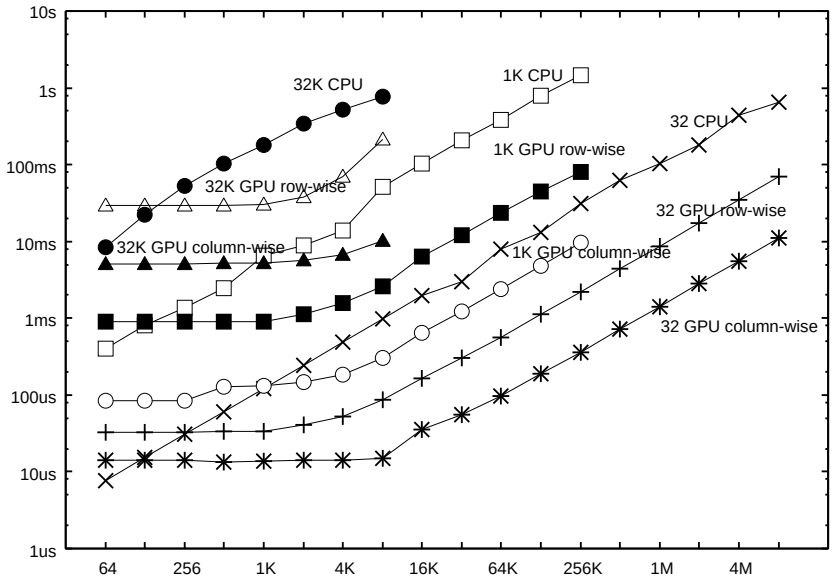
V. Experimental results

We have implemented Parallel Algorithm Prefix-sums and Parallel Algorithm OPT that run in parallel on GeForce GTX Titan [32] using the row-wise arrangement and column-wise arrangement, respectively.

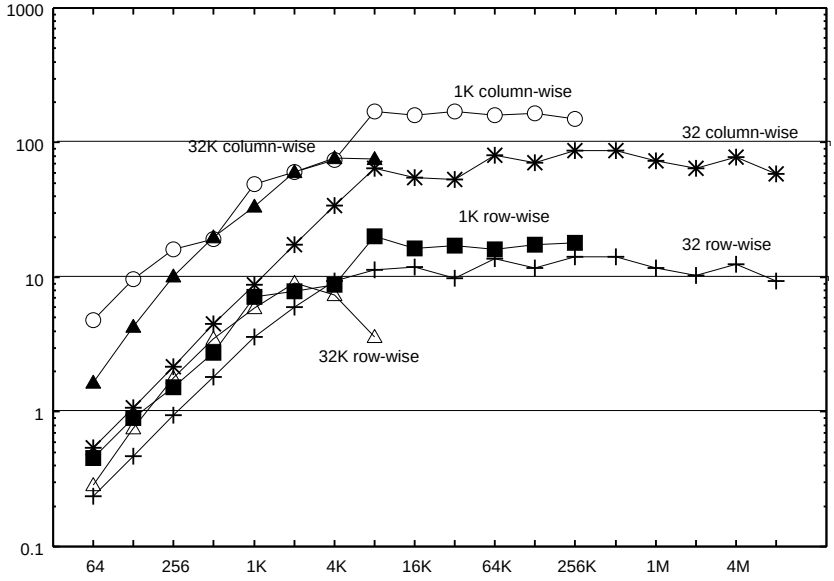
GeForce GTX Titan has 14 streaming multiprocessors with 192 cores each. Hence, it can run 2688 threads in parallel. Note that, a single kernel called to GeForce GTX Titan can run more than 2688 threads in a time sharing manner using CUDA [13] parallel programming platform. All input and output data are stored in the global memory of the GPU and we do not use the shared memory of the streaming multiprocessors.

For the prefix-sums computation, we have executed Parallel Algorithm Prefix-sums for $n = 32$, 1K $(= 1024)$, and 32K $(= 32768)$ float (32-bit) numbers. On the GPU, Parallel Algorithm Prefix-sums is executed for $p = 64, 128, \ldots, 4M$ inputs using p threads in $\frac{p}{64}$ CUDA blocks with 64 threads each. However, due to the global memory capacity, it is executed for up to $p = 256K$ and $p = 8K$ when $n = 1K$ and $n = 32K$, respectively. Also, we have executed Algorithm Prefix-sums p times on the Intel Core i7 CPU (3.5MHz) to see the speedup factor of the GPU over the CPU. We have implemented such algorithm for the row-wise arrangement on the CPU. Figure 11 (1) shows the resulting computing time. Clearly, the computing time by the CPU is proportional to p because it runs $O(pn)$ time. Recall that, from Lemma 1, the row-wise prefix-sums and the column-wise prefix-sums can be computed in $O(np+nl)$ and $O(\frac{np}{w}+nl)$ time units, respectively. The row-wise prefix-sums for $n = 32$ takes about 40μs when $p \leq$1K. Also, the computing time is proportional to p when $p \geq$16K and it runs 67.9ms when $p = 8M$. Thus, we can think that $O(nl) = 37\mu$s and $O(np) = (8.09p)$ns. More specifically, the row-wise prefix-sums for $n = 32$ and p can be computed in approximately 37μs$+(8.09p)$ns. Similarly, the column-wise prefix-sums can be computed in 14μs$+(1.35p)$ns. Figure 11 (2) shows the speedup factor of the row-wise and the column-wise prefix-sums computation using the GPU over the CPU. We can see that the column-wise prefix-sums is much faster than the row-wise prefix-sums and it can achieve a speedup of factor more than 150 when $n =$ 1K and $p \geq$ 8K.

We have also implemented bulk execution of Parallel Algorithm OPT on the same GPU. Similarly to Algorithm Prefix-sums, we have implemented Parallel Algorithm OPT for the row-wise arrangement on the CPU. Figure 12 (1) shows the computing time of the bulk execution on the CPU and the GPU (row-wise and column-wise arrangements) for convex 8-gons, 64-gons, and 512-gons. The computing time is evaluated with p from 64 to the largest possible numbers. For 8-gons, 64-gons, and 512-gons, the maximum values of p are 4M $(= 4194304)$, 64K $(= 65536)$, and 1K $(= 1024)$, respectively. From the figure, we can see that the computing time of the CPU is linear to p. For 8-gons, the computing time of GPU for the row-wise increasing is almost fixed 0.09ms for the size of the bulk execution less than 2K. For the size of the bulk execution larger than 4k, it is linearly increased. Recall that the computing time is $O(pn^3 + ln^3)$ from Corollary 5. Thus, we can think that $O(ln^3)$ =0.09ms and $O(pn^3) = (50.8p)ns$ because it runs 213ms for $p = 4M$ and $\frac{213ms}{4M} \approx 50.8ns$. In other words, the GPU for the row-wise arrangement runs $0.09ms + (50.8p)ns$ for p 8-gons. Similarly, from Corollary 5, the GPU for the



(1) The computing time



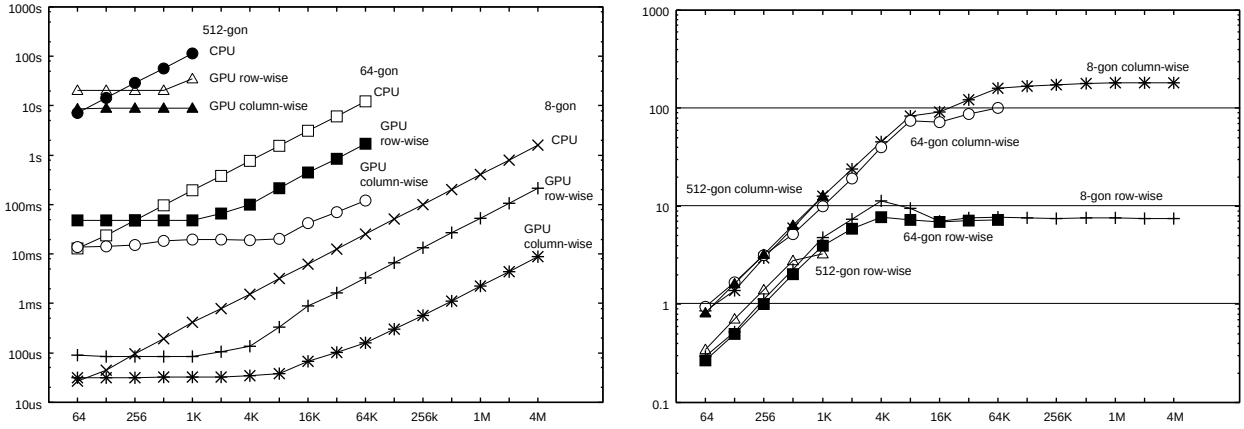
(2) The GPU speedup factor over the CPU

Fig. 11. The computing time of Algorithm Prefix-sums on CPU, GPU (row-wise), and GPU (column-wise), and the speedup for $p = 64, 128, \ldots, 8M$

column-wise arrangement runs in $O(\frac{pn^3}{w} + ln^3)$, and from Figure 12 (1), it runs $0.032ms + (2.11p)ns$ for p 8-gons on the GPU. Figure 12 (2) shows the speedup factor of the GPU over the CPU. The column-wise arrangement can be faster than more than 150 when $p \geq 64$K.

VI. Conclusion

In this paper, we have presented a simple column-wise implementation of the bulk execution of an oblivious sequential algorithm on the UMM. Our column-wise implementation runs $O(\frac{pt}{w} + t)$ time units using p threads on the UMM with width w and latency l, where t is the running time of an oblivious sequential algorithm on a single CPU. Further, we have implemented two oblivious sequential algorithms to compute the prefix-sums on p arrays of size n each and to find the optimal triangulation of p convex n-gons using the dynamic programming technique. The prefix-sum algorithm is a quite simple example of oblivious algorithm while the optimal triangulation algorithm is very complicated. The experimental results on GeForce GTX Titan show that our implementation for the bulk execution for these two algorithm can be more than 150 times faster than the implementation on the conventional CPU. As a further research, we are now developing a conversion system that automatically converts a sequential program written in C language into a CUDA C program for the bulk execution. Using this system, we can get



(1) The computing time (2) The GPU speedup factor over the CPU

Fig. 12. The computing time of Algorithm OPT on CPU, GPU (row-wise), and GPU (column-wise), and the speedup for $p = 64, 128, \ldots, 4M$

the GPU acceleration very easily. We try to implement for the bulk execution of other oblivious algorithms, and can expect to obtain the same results as these oblivious algorithms from our theoretical analysis.

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

[22] 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.

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

[24] ——, "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.

[25] ——, "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.

[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, pp. 79–84.

[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, pp. pp. 1–10.

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

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

[32] NVIDIA Corporation. (2013) NVIDIA GeForce GTX TITAN. [Online]. Available: http://www.geforce.com/hardware/desktop-gpus/geforce-gtx-titan/