PAPER *Special Section on Special Section on Parallel and Distributed Computing and Networking*

Optimal Parallel Algorithms for Computing the Sum, the Prefix-sums, and the Summed Area Table on the Memory Machine Models

Koji NAKANO[†]**, *Member***

SUMMARY The main contribution of this paper is to show optimal parallel algorithms to compute the sum, the prefix-sums, and the summed area table on two memory machine models, the Discrete Memory Machine (DMM) and the Unified Memory Machine (UMM). The DMM and the UMM are theoretical parallel computing models that capture the essence of the shared memory and the global memory of GPUs. These models have three parameters, the number p of threads, and the width w of the memory, and the memory access latency l. We first show that the sum of n numbers can be computed in $O(\frac{n}{w} + \frac{nl}{p} + l\log n)$ time units on the DMM and the UMM. We then go on to show that $\Omega(\frac{n}{w} + \frac{nl}{p} + l\log n)$ time units are necessary to compute the sum. We also present a parallel algorithm that computes the prefix-sums of n numbers in $O(\frac{n}{w} + \frac{nl}{p} + l\log n)$ time units on the DMM and the UMM. Finally, we show that the summed area table of size $\sqrt{n} \times \sqrt{n}$ can be computed in $O(\frac{n}{w} + \frac{nl}{p} + l\log n)$ time units on the DMM and the UMM. Since the computation of the prefix-sums and the summed area table is at least as hard as the sum computation, these parallel algorithms are also optimal.

key words: *Memory machine models, prefix-sums computation, parallel algorithm, GPU, CUDA*

1. Introduction

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]–[4], which consists of processors and a shared memory. Each processor on the PRAM can access any address of the shared memory in a time unit. The PRAM is a good parallel computing model in the sense that parallelism of each problem can be revealed by the performance of parallel algorithms on the PRAM. However, since the PRAM requires a shared memory that can be accessed by all processors at the same time, it is not feasible.

The GPU (Graphics Processing Unit), is a specialized circuit designed to accelerate computation for building and manipulating images [5]–[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 [5], [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 and very high memory bandwidth.

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, and has large capacity, say, 1.5-6 Gbytes, but its access latency is very large. The shared memory is an extremely fast on-chip memory with lower capacity, say, 16-64 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 [7], [11], [12]. 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, threads of CUDA should perform coalesced access when they access the global memory. 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 sequentially. Hence to maximize the memory access performance, threads should access distinct memory banks to avoid the bank conflicts of the memory access.

In our previous paper [13], 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 NVIDIA GPUs. We have presented that an off-line permutation algorithm developed for the DMM runs efficiently on an GPU [14]. The outline of the architectures of the DMM and the UMM are illustrated in Figure 1. In both architectures, *a sea of threads (Ts)* are connected to *the memory banks (MBs)* through *the memory management unit (MMU)*. Each thread is a Random Access Machine (RAM) [1], which can execute fundamental operations in a time unit. We do not discuss the architecture of the sea of threads in this paper, but we can imagine that it consists of a set of multi-core pro-

Manuscript received January 1, 2011.
Manuscript revised January 1, 2011.
[†]The author is with the Department of Information Engineering, Hiroshima University.
DOI: 10.1587/transinf.E0.D.1

cessors which can execute many threads in parallel. Threads are executed in SIMD [15] fashion, and the processors run on the same program and work on the different data.

The DMM and the UMM have three parameters: *width* w, *latency* l, and the number p of threads. The width w is the number of MBs. Also, they can dispatch w threads in p threads and the w threads can send memory access requests. The latency l is the number of clock cycles to complete the memory access requests.

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 mod w)-th bank, where w is the number of MBs. We can think that w addresses $w \cdot j$, $w \cdot j + 1$, ..., $(w+1) \cdot j - 1$ for each $j \geq 0$ constitute *an address group*. The main difference of the two architectures is the connection of the address line between the MMU and the MBs, which can transfer an address value. In the DMM, the address lines connect the MBs and the MMU separately, while a single set of address lines from the MMU is connected to the MBs in the UMM. Hence, in the UMM, the same address value is broadcast to every MB, and the same address of the MBs can be accessed in each time unit. On the other hand, different addresses of the MBs can be accessed in the DMM. Hence, the DMM allows to access any address group of each MB at the same time, while the UMM only allows to access the same address group of all MBs. Since the memory access of the UMM is more restricted than that of the DMM, the UMM is less powerful than the DMM.

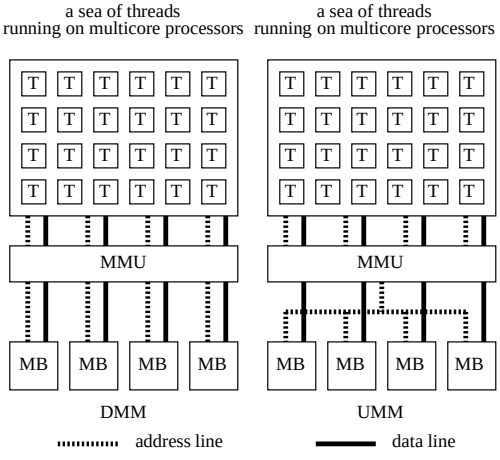


Fig. 1 The architectures of the DMM and the UMM

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 use four parameters, the size n of the input, the number p of threads, the width w and the latency l of the memory when we evaluate the performance of algorithms on the DMM and on the UMM. The width w is the number of memory banks and the latency l is the number of time units to complete the memory access. Hence, the performance of algorithms on the DMM and the UMM is evaluated as a function of n (the size of a problem), p (the number of threads), w (the width of a memory), and l (the latency of a memory). In NVIDIA GPUs, the width w of global and shared memory is 16 or 32. Also, the latency l of the global memory is several hundreds clock cycles. A grid can have at most 65535 blocks with at most 512 threads each for CUDA Version 1.x and $2^{31} - 1$ blocks with at most 1024 threads each for CUDA Version 3.0 or later [10]. Thus, the number p of threads can be 33 million for CUDA Version 1.x and much more for CUDA Version 3.0 or later.

Suppose that an array a of n numbers is given. The prefix-sums of a is the array of size n such that the i-th ($0 \leq i \leq n-1$) element is $a[0] + a[1] + \cdots + a[i]$. In this paper, we consider the addition '+' as a binary operator for the prefix-sums, but it can be generalized for any associative binary operator. Clearly, a sequential algorithm can compute the prefix-sums by executing $a[i+1] \leftarrow a[i+1] + a[i]$ for all i ($0 \leq i \leq n-1$) in turn. The computation of the prefix-sums of an array is one of the most important algorithmic procedures. Many algorithms such as graph algorithms, geometric algorithms, image processing and matrix computation call prefix-sum algorithms as a subroutine. For example, the prefix-sums computation is used to obtain the pre-order, the in-order, and the post-order of a rooted binary tree in parallel [2]. So, it is very important to develop efficient parallel algorithms for the prefix-sums.

Suppose that a matrix a of size $\sqrt{n} \times \sqrt{n}$ is given. Let $a[i][j]$ ($\mathbf{0 \leq i, j \leq \sqrt{n}-1}$) denote the element at the i-th row and j-th column. The summed area table [16] is a matrix b of the same size such that $b[i][j] = \sum_{0 \leq i' \leq i, 0 \leq j' \leq j} a[i'][j']$. It should have no difficulty to confirm that the summed area table can be obtained by computing the column-wise prefix-sums of the row-wise prefix-sums as illustrated in Figure 2. Alternatively, we can compute the row-wise prefix-sums of the column-wise prefix-sums to obtain the same matrix.

Once we have the summed area table, the sum of any rectangular area of a can be computed in $O(1)$ time. More specifically, we have the following equation:

$$\sum_{u<i\leq d, l<j\leq r} a[i][j] = b[d][r] + b[u][l] - b[d][l] - b[u][r].$$

Thus, the sum of a rectangular area can be computed using four elements of the summed area table b. Since the sum of a rectangular area can be computed in $O(1)$ time the summed area table has many applications in the are of image processing [17].

The main contribution of this paper is to show optimal parallel algorithms computing the sum and the prefix-sums on the DMM and the UMM. We first show that the sum of n numbers can be computed in $O(\frac{n}{w} + \frac{nl}{p} + l \log n)$ time units using p threads on the DMM and the UMM with width w and latency l. We then go on to discuss the lower bound of the time complexity and show three lower bounds, $\Omega(\frac{n}{w})$-time bandwidth limitation, $\Omega(\frac{nl}{p})$-time latency

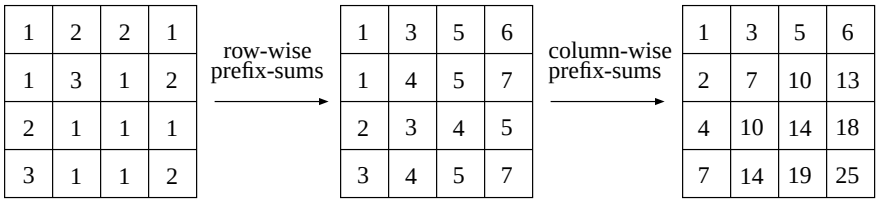


Fig. 2 Summed area table

limitation, and $\Omega(l \log n)$-time reduction limitation. From this discussion, the computation of the sum takes at least $\Omega(\frac{n}{w} + \frac{nl}{p} + l \log n)$ time units on the DMM and the UMM. Thus, the sum algorithm is optimal. For the computation of the prefix-sums, we first evaluate the computing time of a well-known naive algorithm [4], [17]. We show that a naive prefix-sums algorithm runs in $O(\frac{n \log n}{w} + \frac{nl \log n}{p} + l \log n)$ time. Hence, this naive algorithm is not optimal and it has an overhead of factor $\log n$ both for the bandwidth limitation $\frac{n}{w}$ and for the latency limitation $\frac{nl}{p}$. Next, we show an optimal parallel algorithm that computes the prefix-sums of n numbers in $O(\frac{n}{w} + \frac{nl}{p} + l \log n)$ time units on the DMM and the UMM. However, this algorithm uses work space of size n and it may not be acceptable if the size n of the input is very large. We also show that the prefix-sums can also be computed in the same time units, even if work space can store only $\min(p \log p, wl \log(wl))$ numbers. Finally, we show that the summed area table can be computed in $O(\frac{n}{w} + \frac{nl}{p} + l \log n)$ time units on the DMM and the UMM.

Several techniques for computing the prefix-sums on GPUs have been shown in [17]. They have presented a complicated data routing technique to avoid the bank conflict in the computation of the prefix-sums. However, their algorithm performs memory access distant locations in parallel and it performs non-coalesced memory access. Hence it is not efficient for the UMM, that is, the global memory of GPUs. In [17] a work-efficient parallel algorithm for prefix-sums on the GPU has been presented. However, the algorithm uses work space of $n \log n$, and also the theoretical analysis of the performance has not been presented.

This paper is organized as follows. Section 2 briefly defines the Discrete Memory Machine (DMM) and the Unified Memory Machine (UMM) introduced in our previous paper [13]. In Section 3, we evaluate the computing time of the contiguous memory access the memory of the DMM and the UMM. The contiguous memory access is a key ingredient of parallel algorithm development on the DMM and the UMM. Using the contiguous access, we show that the sum of n numbers can be computed in $O(\frac{n}{w} + \frac{nl}{p} + l \log n)$ time units in Section 4. We then go on to discuss the lower bound of the time complexity and show three lower bounds, $\Omega(\frac{n}{w})$-time bandwidth limitation, $\Omega(\frac{nl}{p})$-time latency limitation, and $\Omega(l \log n)$-time reduction limitation in Section 5. Section 6 shows a naive algorithm for the prefix-sums, which runs in $O(\frac{n \log n}{w} + \frac{nl \log n}{p} + l \log n)$ time units. We then present an optimal parallel algorithm for the prefix-sums running in $O(\frac{n}{w} + \frac{nl}{p} + l \log n)$ time units in Section 7. Finally, we show that the summed area table can be computed in $O(\frac{n}{w} + \frac{nl}{p} + l \log n)$ time units in Section 8. Section 9 offers conclusion of this paper.

2. Parallel Memory Machines: DMM and UMM

The main purpose of this section is to define the Discrete Memory Machine (DMM) and the Unified Memory Machine (UMM) introduced in our previous paper [13]. Please see [13] for the details.

We first define *the Discrete Memory Machine (DMM)* of width w and latency l. Let $m[i]$ $(i \geq 0)$ denote a memory cell of address i in the memory. Let $B[j] = \{m[j], m[j + w], m[j + 2w], m[j + 3w], \ldots\}$ $(0 \leq j \leq w - 1)$ denote *the j-th bank* of the memory. Clearly, a memory cell $m[i]$ is in the $(i \bmod w)$-th memory bank. We assume that memory cells in different banks can be accessed in a time unit, but no two memory cells in the same bank can be accessed in a time unit. Also, we assume that l time units are necessary to complete an access request and continuous requests are processed in a pipeline fashion through the MMU. Thus, if only one request is sent to a bank, l time units are necessary to complete it. In general, it takes $k + l - 1$ time units to complete k (≥ 1) access requests to a particular bank.

We assume that p threads on the DMM and the UMM with width w are partitioned into $\frac{p}{w}$ groups of w threads called *warps*. It makes sense to assume that $p \geq w$ and p is a multiple of w. More specifically, p threads are partitioned into $\frac{p}{w}$ warps $W(0), W(1), \ldots, W(\frac{p}{w} - 1)$ such that $W(i) = \{T(i \cdot w), T(i \cdot w + 1), \ldots, T((i + 1) \cdot w - 1)\}$ $(0 \leq i \leq \frac{p}{w} - 1)$. In other words, the number of MBs and the number of threads in a warp are equal. Warps are activated for memory access in turn, and w threads in a warp try to access the memory at the same time. In other words, $W(0), W(1), \ldots, W(\frac{p}{w} - 1)$ are activated in a round-robin manner if at least one thread in a warp requests memory access. If no thread in a warp needs memory access, such warp is not activated for memory access. When $W(i)$ is activated, w threads in $W(i)$ sends memory access requests, one request per thread, to the memory. We also assume that a thread cannot send a new memory access request until the previous memory access request is completed. Hence, if a thread send a memory access request, it must wait l time units to send a new memory access request.

We next define *the Unified Memory Machine* (*UMM*) of width w as follows. Let $A[j] = \{m[j \cdot w], m[j \cdot w +$

$1], \ldots, m[(j+1)\cdot w-1]\}$ denote the j-th address group. We assume that memory cells in the same address group are processed at the same time. However, if they are in the different groups, one time unit is necessary for each of the groups. Also, similarly to the DMM, p threads are partitioned into warps and each warp accesses the memory in turn.

Figure 3 shows examples of memory access on the DMM and the UMM. We can think that the memory access is performed through l-stage pipeline registers as illustrated in the figure. We assume that each memory access request is completed when it reaches the last pipeline stage. Two warps $W(0)$ and $W(1)$ access to $\langle m[7], m[5], m[15], m[0]\rangle$ and $\langle m[10], m[11], m[12], m[9]\rangle$, respectively. In the DMM, memory access requests by $W(0)$ are separated into two pipeline stages, because addresses $m[7]$ and $m[15]$ are in the same bank $B(3)$. Those by $W(1)$ occupies 1 stage, because all requests are in distinct banks. Thus, the memory requests occupy three stages, it takes $3+5-1=7$ time units to complete the memory access. In the UMM, memory access requests by $W(0)$ are destined for three address groups. Hence the memory requests occupy three stages. Similarly those by $W(1)$ occupy two stages. Hence, it takes $5+5-1=9$ time units to complete the memory access.

Throughout of this paper, without losing generality, we assume that the width w of the DMM and the UMM, the number p of the threads, and the size n of problem input are powers of two. The reader should have no difficulty to modify algorithms presented in this paper to work correctly even if some of these values are not powers of two using standard algorithmic techniques.

3. Contiguous Memory Access

The main purpose of this section is to review the contiguous memory access on the DMM and the UMM shown in [13].

Suppose that an array a of size n ($\geq p$) is given. We use p threads to access all of n elements in a such that each thread accesses $\frac{n}{p}$ elements. Note that "accessing" can be "reading from" or "writing in." Let $a[i]$ $(0 \leq i \leq n-1)$ denote the i-th element in a. When $n \geq p$, the contiguous access can be performed as follows:

[**Contiguous memory access**]
for $t \leftarrow 0$ to $\frac{n}{p}-1$ do
  for $i \leftarrow 0$ to $p-1$ do in parallel
    $T(i)$ access $a[p\cdot t+i]$

We will evaluate the computing time. For each t $(0 \leq t \leq \frac{n}{p}-1)$, p threads access p elements $a[pt], a[pt+1], \ldots, a[p(t+1)-1]$. This memory access is performed by $\frac{p}{w}$ warps in turn. More specifically, first, w threads in $W(0)$ access $a[pt], a[pt+1], \ldots, a[pt+w-1]$. After that, w threads in $W(1)$ access $a[pt+w], a[pt+w+1], \ldots, a[pt+2w-1]$, and the same operation is repeatedly performed. In general, w threads in $W(j)$ $(0 \leq j \leq \frac{p}{w}-1)$ accesses $a[pt+jw], a[pt+jw+1], \ldots, a[pt+(j+1)w-1]$. Since w elements are accessed by a warp are in the different bank, the access can be completed in l time units on the DMM. Also, these w elements are in the same address group, and thus, the access can be completed in l time units on the UMM.

Recall that the memory access are processed in pipeline fashion such that w threads in each $W(j)$ send w memory access requests in one time unit. Hence, p threads in $\frac{p}{w}$ warps send p memory access requests in $\frac{p}{w}$ time units. After that, the last memory access requests by $W(\frac{p}{w}-1)$ are completed in $l-1$ time units. Thus, p threads access p elements $a[pt], a[pt+1], \ldots, a[p(t+1)-1]$ in $\frac{p}{w}+l-1$ time units. Since this memory access is repeated $\frac{n}{p}$ times, the contiguous access can be done in $\frac{n}{p}\cdot(\frac{p}{w}+l-1) = O(\frac{n}{w}+\frac{nl}{p})$ time units.

If $n < p$ then, the contiguous memory access can be simply done using n threads out of the p threads. If this is the case, the memory access can be done by $O(\frac{n}{w}+l)$ time units. Therefore, we have,

Lemma 1: The contiguous access to an array of size n can be done in $O(\frac{n}{w}+\frac{nl}{p}+l)$ time units using p threads on the UMM and the DMM with width w and latency l.

We can consider that memory access is contiguous if w threads in every warp access the contiguous addresses of the shared memory on the DMM or the global memory of the UMM. It should have no difficulty to confirm that, if all of the p threads performs such contiguous memory access to n numbers, $\frac{n}{p}$ numbers per thread, the memory access is also completed in $O(\frac{n}{w}+\frac{nl}{p}+l)$ time units.

4. An optimal parallel algorithm for computing the sum

The main purpose of this section is to show an optimal parallel algorithm for computing the sum on the memory machine models.

Let a be an array of $n=2^m$ numbers. Let us show an algorithm to compute the sum $a[0]+a[1]+\cdots+a[n-1]$. The algorithm uses a well-known parallel computing technique which repeatedly computes the pairwise sums. We implement this technique to perform contiguous memory access. The details are spelled out as follows:

[**Optimal algorithm for computing the sum**]
for $t \leftarrow m-1$ downto 0 do
  for $i \leftarrow 0$ to 2^t-1 do in parallel
    $a[i] \leftarrow a[i]+a[i+2^t]$

Figure 4 illustrates how the sums of pairs are computed. From the figure, the reader should have no difficulty to confirm that this algorithm compute the sum correctly.

We assume that p threads are used to compute the sum. For each t $(0 \leq t \leq m-1)$, 2^t operations "$a[i] \leftarrow a[i]+a[i+2^t]$" are performed. These operation involve the following memory access operations:

- reading from $a[0], a[1], \ldots, a[2^t-1]$,
- reading from $a[2^t], a[2^t+1], \ldots, a[2\cdot 2^t-1]$, and
- writing in $a[0], a[1], \ldots, a[2^t-1]$.

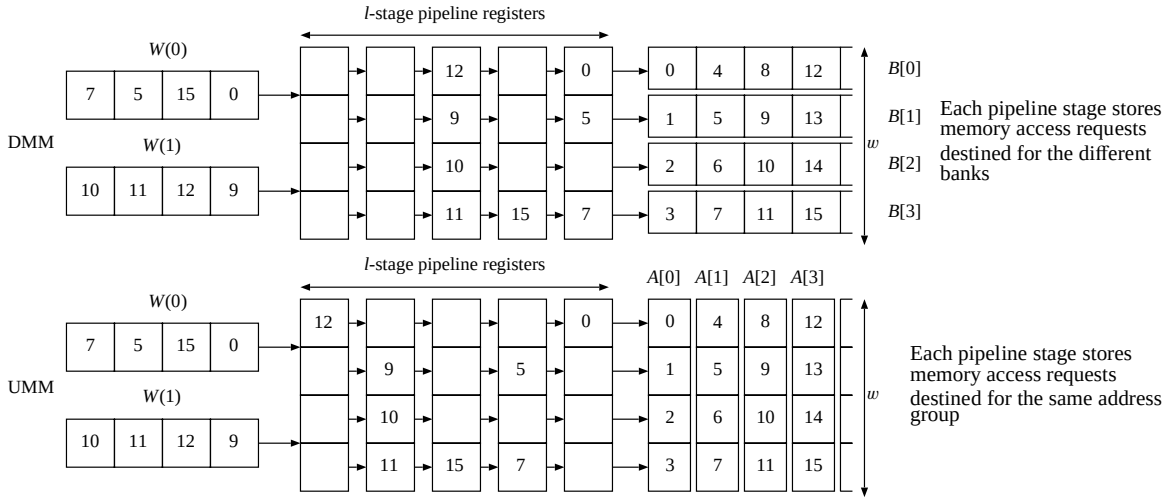


Fig. 3 Examples of memory access on the DMM and the UMM

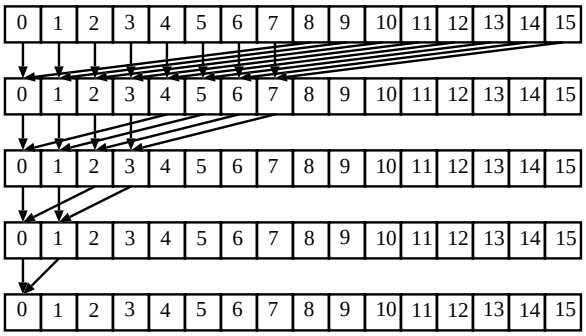


Fig. 4 Illustrating the summing algorithm for n numbers

Since these memory access operations are contiguous, they can be done in $O(\frac{2^t}{w} + \frac{2^t l}{p} + l)$ time using p threads both on the DMM and on the UMM with width w and latency l from Lemma 1. Thus, the total computing time is

$$\sum_{t=0}^{m-1} O(\frac{2^t}{w} + \frac{2^t l}{p} + l) = O(\frac{2^m}{w} + \frac{2^m l}{p} + lm)$$
$$= O(\frac{n}{w} + \frac{nl}{p} + l\log n)$$

and we have,

Lemma 2: The sum of n numbers can be computed in $O(\frac{n}{w} + \frac{nl}{p} + l\log n)$ time units using p threads on the DMM and on the UMM with width w and latency l.

5. The lower bound of the computing time and the latency hiding

Let us discuss the lower bound of the time necessary to compute the sum on the DMM and the UMM to show that our parallel summing algorithm for Lemma 2 is optimal. Since the sum is the last value of the prefix-sums and the summed area table, this lower bound discussion for the sum can be applied to that for the prefix-sums and the summed area table. We will show three lower bounds of the sum, $\Omega(\frac{n}{w})$-time bandwidth limitation, $\Omega(\frac{nl}{p})$-time latency limitation, and $\Omega(l\log n)$-time reduction limitation.

Since the width of the memory is w, at most w numbers in the memory can be read in a time unit. Clearly, all of the n numbers must be read to compute the sum. Hence, $\Omega(\frac{n}{w})$ time units are necessary to compute the sum. We call the $\Omega(\frac{n}{w})$-time lower bound *the bandwidth limitation*.

Since the memory access takes latency l, a thread can send at most $\frac{t}{l}$ memory read requests in t time units. Thus, p threads can send at most $\frac{pt}{l}$ total memory requests in t time units. Since at least n numbers in the memory must be read to compute the sum, $\frac{pt}{l} \geq n$ must be satisfied. Thus, at least $t = \Omega(\frac{nl}{p})$ time units are necessary. We call the $\Omega(\frac{nl}{p})$-time lower bound *the latency limitation*.

Next, we will show *the reduction limitation*, the $\Omega(l\log n)$-time lower bound. The formal proof is more complicated than those for the bandwidth limitation and the latency limitation.

Imagine that each of n input numbers stored in the shared memory (or the global memory) is a token and each thread is a box. Whenever two tokens are placed in a box,

they are merged into one immediately. We can move tokens to boxes and each box can accept at most one token in l time units. Suppose that we have n tokens outside boxes. We will prove that it takes at least $l \log n$ time units to merge them into one token. For this purpose, we will prove that, if we have n' tokens at some time, we must have at least $\frac{n'}{2}$ tokens l time units later. Suppose that we have n' tokens such that k of them are in k boxes and the remaining $n'-k$ tokens are out of boxes. If $k \leq n'-k$ then we can move k tokens to k boxes and can merge k pairs of tokens in l time units. After that, $n'-k$ tokens remain. If $k > n'-k$ then we can merge $n'-k$ pairs of tokens and we have k tokens after l time units. Hence, after l time units, we have at least $\max(n'-k, k) \geq \frac{n'}{2}$ tokens. Thus, we must have at least $\frac{n'}{2}$ tokens l time units later. In other words, it is not possible to reduce the number of tokens by half or less. Thus, it takes at least $l \log n$ time units to merge n tokens into one. It should be clear that, reading a number by a thread from the shared memory (or the global memory) corresponds to a token movement to a box. Therefore, it takes at least $\Omega(l \log n)$ time units to compute the sum of n numbers.

From the discussion above, we have,

Theorem 3: Both the DMM and the UMM with p threads, width w, and latency l take at least $\Omega(\frac{n}{w} + \frac{nl}{p} + l \log n)$ time units to compute the sum of n numbers.

From Theorem 3, the parallel algorithm for commuting the sum shown for Lemma 2 is optimal.

Let us discuss the relation of three limitations. From a practical point of view, width w and latency l are constant values that cannot be changed by parallel computer users. These values are fixed when a parallel computer based on the memory machine models is manufactured. Also, the size n of the input are variable. Programmers can adjust the number p of threads to obtain the best performance. Thus, the value of the latency limitation $\frac{nl}{p}$ can be changed by programmers.

Let us compare the values of three limitations.

$\boldsymbol{wl \leq p}$: From $\frac{n}{w} \geq \frac{nl}{p}$, the bandwidth limitation dominates the latency limitation.

$\boldsymbol{wl \leq \frac{n}{\log n}}$: From $\frac{n}{w} \geq l \log n$, the bandwidth limitation dominates the reduction limitation.

$\boldsymbol{p \leq \frac{n}{\log n}}$: From $\frac{nl}{p} \geq l \log n$, the latency limitation dominates the reduction limitation.

Thus, if both $wl \leq p$ and $wl \leq \frac{n}{\log n}$ are satisfied, the computing time of the sum algorithm for Lemma 2 is $O(\frac{n}{w})$. As illustrated in Figure 3, the memory machine models have l-stage pipeline registers such that each stage has w registers. Since more than one memory requests by a thread can not be stored in each pipeline register, $wl \leq p$ must be satisfied to fill all the pipeline registers with memory access requests by p threads. Since the sum algorithm has $\log n$ stages and expected $\frac{n}{\log n}$ memory access requests are sent to the pipeline registers, $wl \leq \frac{n}{\log n}$ must also be satisfied to fill all the pipeline registers with $\frac{n}{\log n}$ memory access requests. From the discussion above, to hide the latency, the number p of threads must be at least the number wl of pipeline registers and the size n of input must be at least $wl \log(wl)$.

6. A naive prefix-sums algorithm

We assume that an array a with $n = 2^m$ numbers is given. Let us start with a well-known naive prefix-sums algorithm for array a [17], [18], and show it is not optimal. The naive prefix-sums algorithm is written as follows:

[A naive prefix-sums algorithm]
for $t \leftarrow 0$ to $m-1$ do
  for $i \leftarrow 2^t$ to $n-1$ do in parallel
    $a[i] \leftarrow a[i] + a[i-2^t]$

Figure 5 illustrates how the prefix-sums are computed.

We assume that p threads are available and evaluate the computing time of the naive prefix-sums algorithm. The following three memory access operations are performed for each t ($0 \leq t \leq m-1$) by:

- reading from $a[0], a[1], \ldots, a[n-2^t-1]$,
- reading from $a[2^t], a[2^t+1], \ldots, a[n-1]$, and
- writing in $a[2^t], a[2^t+1], \ldots, a[n-1]$.

Each of the three operations can be done by contiguous memory access for $n-2^t$ elements. Hence, the computing time of each t is $O(\frac{n-2^t}{w} + \frac{(n-2^t)l}{p} + l)$ from Lemma 1. The total computing time is:

$$\sum_{t=0}^{m-1} O(\frac{n-2^t}{w} + \frac{(n-2^t)l}{p} + l)$$
$$= O(\frac{n \log n}{w} + \frac{nl \log n}{p} + l \log n).$$

Thus, we have,

Lemma 4: The naive prefix-sum algorithm runs in $O(\frac{n \log n}{w} + \frac{nl \log n}{p} + l \log n)$ time units using p threads on the DMM and on the UMM with width w and latency l.

If the computing time of Lemma 4 matches the lower bound shown in Theorem 3, the prefix-sum algorithm is optimal. However, it does not match the lower bound. In the following section, we will show an optimal prefix-sum algorithm whose running time matches the lower bound.

7. Optimal prefix-sum algorithm

This section shows an optimal algorithm for the prefix-sums running in $O(\frac{n}{w} + \frac{nl}{p} + l \log n)$ time units. We use m arrays $a_0, a_1, \ldots a_{m-1}$ as work space. Each a_t ($0 \leq t \leq m-1$) can store 2^t numbers. Thus, the total size of the m arrays is no more than $2^0 + 2^1 + \cdots + 2^{m-1} - 1 = 2^m - 1 = n - 1$. We assume that the input of n numbers are stored in array a_m of size n.

The algorithm has two stages. In the first stage, interval sums are stored in the m arrays. The second stage uses interval sums in the m arrays to compute the resulting

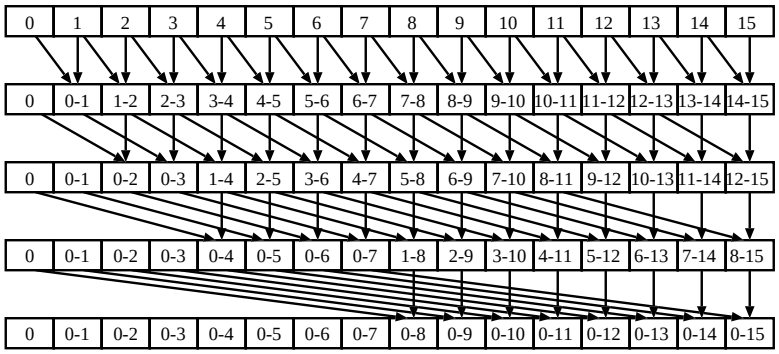


Fig. 5 Illustrating the naive prefix-sums algorithm for n numbers

prefix-sums. The details of the first stage are spelled out as follows.

[Compute the interval sums]
for $t \leftarrow m-1$ downto 0 do
  for $i \leftarrow 0$ to 2^t-1 do in parallel
    $a_t[i] \leftarrow a_{t+1}[2\cdot i] + a_{t+1}[2\cdot i+1]$

Figure 6 illustrates how the interval sums are computed. When this program terminates, each $a_t[i]$ $(0 \le t \le m-1, 0 \le i \le 2^t-1)$ stores $a_t[i\cdot\frac{n}{2^t}]+a_t[i\cdot\frac{n}{2^t}+1]+\cdots+a_t[(i+1)\cdot\frac{n}{2^t}-1]$.

In the second stage, the prefix-sums are computed by computing the sums of the interval sums as follows:

[Compute the sums of the interval sums]
for $t \leftarrow 0$ to $m-1$ do
  for $i \leftarrow 0$ to 2^t-1 do in parallel
    $a_{t+1}[2\cdot i+1] \leftarrow a_t[i]$
  for $i \leftarrow 0$ to 2^t-2 do in parallel
    $a_{t+1}[2\cdot i+2] \leftarrow a_{t+1}[2\cdot i+2] + a_t[i]$

Figure 7 shows how the prefix-sums are computed. In the figure, "$a_{t+1}[2\cdot i+1] \leftarrow a_t[i]$" and "$a_{t+1}[2\cdot i+2] \leftarrow a_{t+1}[2\cdot i+2] + a_t[i]$" correspond to "copy" and "add", respectively.

When this algorithm terminates, each $a_m[i]$ $(0 \le i \le 2^t-1)$ stores the prefix-sum $a_m[0]+a_m[1]+\cdots+a_m[i]$. We assume that p threads are available and evaluate the computing time. The first stage involves the following memory access operations for each t $(0 \le t \le m-1)$:

- reading from $a_{t+1}[0], a_{t+1}[2], \ldots, a_{t+1}[2^{t+1}-2]$,
- reading from $a_{t+1}[1], a_{t+1}[3], \ldots, a_{t+1}[2^{t+1}-1]$, and
- writing in $a_t[0], a_t[1], \ldots, a_t[2^t-1]$.

Every two addresses is accessed in the reading operations. Thus, these three memory access operations are essentially contiguous access and they can be done in $O(\frac{2^t}{w}+\frac{2^t l}{p}+l)$ time units. Therefore, the total computing time of the first stage is

$$\sum_{t=1}^{m-1} O(\frac{2^t}{w}+\frac{2^t l}{p}+l) = O(\frac{n}{w}+\frac{nl}{p}+l\log n).$$

The second stage consists of the following memory access operations for each t $(0 \le t \le m-1)$:

- reading from $a_t[0], a_t[1], \ldots, a_t[2^t-1]$,
- reading from $a_{t+1}[2], a_{t+1}[4], \ldots, a_{t+1}[2^{t+1}-2]$, and
- writing in $a_{t+1}[0], a_{t+1}[1], \ldots, a_{t+1}[2^{t+1}-1]$.

Similarly, these operations can be done in $O(\frac{2^t}{w}+\frac{2^t l}{p}+l)$ time units. Hence, the total computing time of the second stage is also $O(\frac{n}{w}+\frac{nl}{p}+l\log n)$. Thus, we have,

Theorem 5: The prefix-sums of n numbers can be computed in $O(\frac{n}{w}+\frac{nl}{p}+l\log n)$ time units using p threads on the DMM and on the UMM with width w and latency l if work space of size n is available.

From Theorem 3, the lower bound of the computing time of the prefix-sums is $\Omega(\frac{n}{w}+\frac{nl}{p}+l\log n)$.

Suppose that n is very large and work space of size n is not available. We will show that, if work space no smaller than $\min(p\log p, wl\log(wl))$ is available, the prefix-sums can also be computed in $O(\frac{n}{w}+\frac{nl}{p}+l\log n)$. Let k be an arbitrary number such that $p \le k \le n$. We partition the input a with n numbers into $\frac{n}{k}$ groups with k $(\ge p)$ numbers each. Each t-th group $(0 \le t \le \frac{n}{k}-1)$ has k numbers $a[tk], a[tk+1], \ldots, a[(t+1)k-1]$. The prefix-sums of every group is computed using p threads in turn as follows.

[Sequential-parallel prefix-sums algorithm]
for $t \leftarrow 0$ to $\frac{n}{k}-1$ do
  begin
    if$(t \neq 0)$ $a[tk] \leftarrow a[tk] + a[tk-1]$
    compute the prefix-sums of k numbers
      $a[tk], a[tk+1], \ldots, a[(t+1)k-1]$
  end

It should be clear that this algorithm computes the prefix-sums correctly. The prefix-sums of k numbers can be computed in $O(\frac{k}{w}+\frac{kl}{p}+l\log k)$. Since the computation of the prefix-sums is repeated $\frac{n}{k}$ times, the total computing time is $O(\frac{k}{w}+\frac{kl}{p}+l\log k)\cdot\frac{n}{k} = O(\frac{n}{w}+\frac{nl}{p}+\frac{nl\log k}{k})$. Thus, we have,

Corollary 6: The prefix-sums of n numbers can be computed in $O(\frac{n}{w}+\frac{nl}{p}+\frac{nl\log k}{k})$ time units using p threads on the DMM and on the UMM with width w and latency l if work space of size k is available.

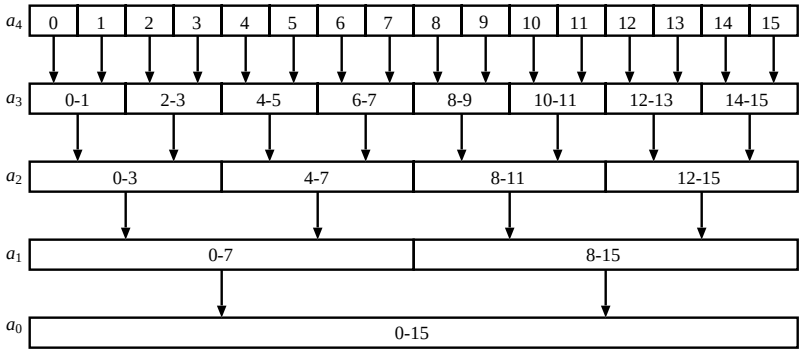


Fig. 6 Illustrating the computation of interval sums in m arrays.

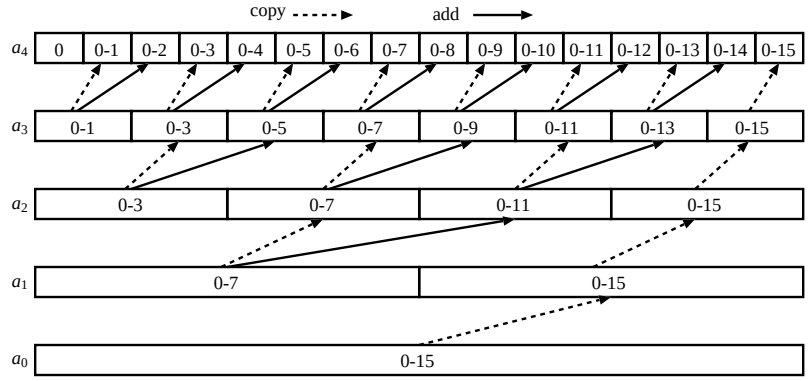


Fig. 7 Illustrating the computation of the sums of the interval sums in m arrays.

If $k \geq p \log p$ then, $\frac{nl \log k}{k} \leq \frac{nl \log(p \log p)}{p \log p} = O(\frac{nl}{p})$. If $k \geq wl \log(wl)$ then $\frac{nl \log k}{k} \leq \frac{nl \log(wl \log(wl))}{wl \log(wl)} = O(\frac{n}{w})$. Thus, if $k \geq \min(p \log p, wl \log(wl))$ then the computing time is $O(\frac{n}{w} + \frac{nl}{p})$.

8. Optimal algorithms for the summed area table

The main purpose of this section is to show optimal algorithms for computing the summed area table. It should be clear that the column-wise prefix-sums can be obtained by the transpose, the row-wise prefix-sums computation, and the transpose. In our previous paper [13], we have presented that the transpose of a matrix of size $\sqrt{n} \times \sqrt{n}$ can be done in $O(\frac{n}{w} + \frac{nl}{p})$ time units both on the DMM and the UMM with width w and latency l using p threads. As illustrated in Figure 2 the summed area table can be computed by the row-wise prefix-sums and the column-wise prefix-sums. Thus, if the row-wise prefix-sums of a matrix of size $\sqrt{n} \times \sqrt{n}$ can be computed in $O(\frac{n}{w} + \frac{nl}{p} + l \log n)$ time units, the summed area table can be computed in the same time units.

For simplicity, we assume that $\sqrt{n}$ is a power of two. Recall that the algorithm for Theorem 5 computes the prefix-sums of n numbers. Imagine that the this algorithm is executed on a matrix of size $\sqrt{n} \times \sqrt{n}$ such that numbers in a matrix arranged in a 1-dimensional array in row-major order. We will show that the row-wise prefix-sums can be computed by modifying the algorithm for Theorem 5, which has following two stages:

Stage 1 the interval sums are computed as illustrated in Figure 6, and

Stage 2 the sums of the interval sums are computed as illustrated in Figure 7.

The idea is to omit several operations performed in these two stages such that the interval sums of a row is not propagated to the next row. We can think that a_t ($\frac{\log n}{2} \leq t \leq \log n$) in these stages is a matrix of size $\sqrt{n} \times \frac{2^t}{\sqrt{n}}$ such that $a_t[i \cdot \frac{2^t}{\sqrt{n}} + j]$ ($0 \leq i \leq \sqrt{n}-1, 0 \leq j \leq \frac{2^t}{\sqrt{n}}-1$) is an element in the i-th row and j-th column. For example, the sizes of a_4, a_3, and a_2 in Figures 6 and 7 are 4×4, 4×2 and 4×1, respectively. Hence, $a_4[0], a_4[1], a_4[2], a_4[3], a_3[0], a_3[1]$, and $a_2[0]$ are in row 0. We modify Stage 1 such that the interval sums within each row is computed. For this purpose, we execute the first $\frac{\log n}{2}$ ($= \frac{m}{2}$) steps of Stage 1 and the remaining $\frac{m}{2}$ steps are omitted as follows:

[Compute the interval sums in row-wise]
for $t \leftarrow m-1$ downto $\frac{m}{2}$ do
  for $i \leftarrow 0$ to $2^t - 1$ do in parallel
    $a_t[i] \leftarrow a_{t+1}[2 \cdot i] + a_{t+1}[2 \cdot i + 1]$

For example, in Figure 6, a_3 and a_2 are computed, but the computation of a_1 and a_0 is omitted.

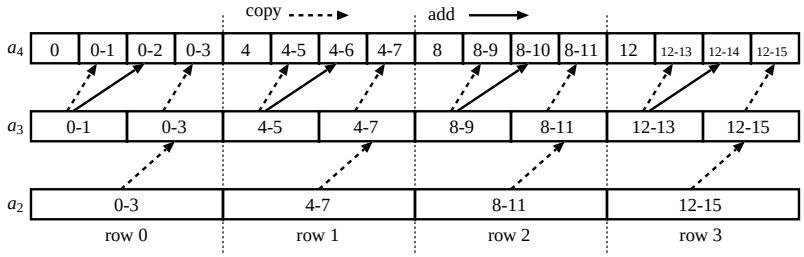


Fig. 8 Illustrating the computation of the sums of the interval sums of every row

Next, we modify Stage 2 such that the computation is performed within each row. First, we omit the first $\frac{m}{2}$ steps and execute the remaining $\frac{m}{2}$ steps. In the remaining $\frac{m}{2}$ steps, operations from a row to the next row are omitted. The reader should compare Figure 7 and Figure 8 to see how Stage 2 is modified. Note that all "copy" operations are always performed within the same row. However, "add" operations may be performed to next rows. For example, "add" operation $a_3[2] \leftarrow a_3[2] + a_2[0]$ is omitted because $a_3[2]$ is in row 1 and $a_2[0]$ is in row 0. The details are spelled out as follows:

[Compute the sums of the interval sums in row-wise]
for $t \leftarrow \frac{m}{2}$ to $m-1$ do
  for $i \leftarrow 0$ to $2^t - 1$ do in parallel
    $a_{t+1}[2 \cdot i + 1] \leftarrow a_t[i]$
  for $i \leftarrow 0$ to $2^t - 2$ do in parallel
    if $a_{t+1}[2 \cdot i + 2]$ and $a_t[i]$ are in the same row then
      $a_{t+1}[2 \cdot i + 2] \leftarrow a_{t+1}[2 \cdot i + 2] + a_t[i]$

Clearly the computation of modified Stages 1 and 2 does not exceed their original stages shown for Theorem 5. Thus, we have,

Theorem 7: The summed area table of a matrix of size $\sqrt{n} \times \sqrt{n}$ can be computed in $O(\frac{n}{w} + \frac{nl}{p} + l \log n)$ time units using p threads on the DMM and on the UMM with width w and latency l, if work space of size n is available.

From Theorem 3, the parallel algorithm for Theorem 7 is optimal.

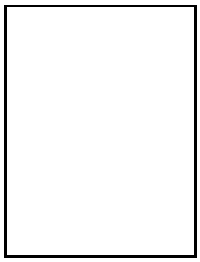
9. Conclusion

The main contribution of this paper is to show parallel algorithms to compute the sum, the prefix-sums, and the summed area table in $O(\frac{n}{w} + \frac{nl}{p} + l \log n)$ time units using p threads on the DMM and the UMM with width w and latency l. We have also shown any algorithm to compute these values takes $\Omega(\frac{n}{w} + \frac{nl}{p} + l \log n)$ time units.

We believe that two memory machine models, the DMM and the UMM are promising as platforms of development of algorithmic techniques for GPUs. We plan to develop efficient algorithms for graph-theoretic problems, geometric problems, and image processing problems on the DMM and the UMM.

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] A. Grama, G. Karypis, V. Kumar, and A. Gupta, Introduction to Parallel Computing, Addison Wesley, 2003.
[4] M.J. Quinn, Parallel Computing: Theory and Practice, McGraw-Hill, 1994.
[5] W.W. Hwu, GPU Computing Gems Emerald Edition, Morgan Kaufmann, 2011.
[6] 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.
[7] 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.
[8] A. Uchida, Y. Ito, and K. Nakano, "Fast and accurate template matching using pixel rearrangement on the GPU," Proc. of International Conference on Networking and Computing, pp.153–159, Dec. 2011.
[9] 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.
[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] NVIDIA Corporation, "NVIDIA CUDA C best practice guide version 3.1," 2010.
[13] K. Nakano, "Simple memory machine models for GPUs," Proc. of International Parallel and Distributed Processing Symposium Workshops, pp.788–797, May 2012.
[14] A. Kasagi, K. Nakano, and Y. Ito, "An implementation of conflict-free off-line permutation on the GPU," to appear in Proc. of International Conference on Networking and Computing, pp.226–232, 2012.
[15] M.J. Flynn, "Some computer organizations and their effectiveness," IEEE Transactions on Computers, vol.C-21, pp.948–960, 1972.
[16] F. Crow, "Summed-area tables for texture mapping," Proc. of the 11th annual conference on Computer graphics and interactive techniques, pp.207–212, 1984.
[17] M. Harris, S. Sengupta, and J.D. Owens, "Chapter 39. parallel prefix sum (scan) with CUDA," in GPU Gems 3, Addison-Wesley, 2007.
[18] W.D. Hillis and G.L. Steele, Jr., "Data parallel algorithms," Commun. ACM, vol.29, no.12, pp.1170–1183, Dec. 1986.



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 also guest-edited several special issues including IEEE TPDS Special issue on Wireless Networks and Mobile Computing, IJFCS special issue on Graph Algorithms and Applications, and IEICE Transactions special issue on Foundations of 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 includes image processing, hardware algorithms, GPU-based computing, FPGA-based reconfigurable computing, parallel computing, algorithms and architectures.