An Optimal Parallel Prefix-sums Algorithm on the Memory Machine Models for GPUs

Koji Nakano

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

Abstract. The main contribution of this paper is to show optimal algorithms computing the sum and the prefix-sums 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, 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. Finally, 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.

Keywords: 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) [5, 7, 19], 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 in the same time, it is not feasible.

The GPU (Graphical Processing Unit), is a specialized circuit designed to accelerate computation for building and manipulating images [10, 11, 13, 20]. 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 [10, 16]. NVIDIA provides a parallel computing architecture called *CUDA* (Compute Unified Device Architecture) [18], 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 global memory* and *the shared memory* [18]. 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 long. 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 [13, 14, 17]. To maximize the bandwidth between the GPU and the DRAM chips, the consecutive addresses of the global memory must be accessed in the same time. Thus, threads of CUDA should perform coalesced access when they access to the global memory. The address space of the shared memory is mapped into several physical memory banks. If two or more threads access to the same memory banks in the same time, the access requests are processed sequentially. Hence to maximize the memory access performance, threads should access to distinct memory banks to avoid the bank conflicts of the memory access.

In our previous paper [15], 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. The outline of the architectures of the DMM and the UMM are illustrated in Figure 1. In both architectures, *a sea of threads (Ts)* is connected to *the memory banks (MBs)* through *the memory management unit (MMU)*. Each thread is a Random Access Machine (RAM) [1], which can execute one of the 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 processors which can execute many threads in parallel and/or in time-sharing manner. Threads are executed in SIMD [4] fashion, and the processors run on the same program and work on the different data.

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. 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 address line from the MMU is connected to the MBs in the UMM. Hence, in the UMM, the same address value is broadcast to every MB, and the same address of the MBs can be accessed in each time unit. On the other hand,

different addresses of the MBs can be accessed in the DMM. Since the memory access of the UMM is more restricted than that of the DMM, the UMM is less powerful than the DMM.

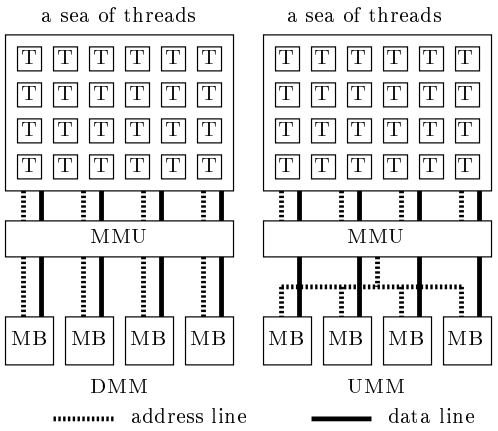

Fig. 1. The architectures of the DMM and the UMM

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 [5]. We will use four parameters, the size n of the input, the number p of threads, the width w and the latency l of the memory when we evaluate the performance of algorithms on the DMM and on the UMM. The width w is the number of memory banks 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. In CUDA, a grid can have at most 65535 blocks with at most 1024 threads each [18]. Thus, the number p of threads can be 65 million.

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 $a[0] + a[1] + \cdots + a[i]$. Clearly, a sequential algorithm can compute the prefix sums by executing $a[i+1] \leftarrow a[i+1] + a[i]$ for all i $(0 \leq i \leq n-1)$. 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-sums algorithms as a subroutine. In particular, many parallel algorithms uses a parallel prefix-sums algorithm. 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 [5]. So, it is very important to develop efficient parallel algorithms for the prefix-sums.

The main contribution of this paper is to show an optimal prefix-sums algorithm 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 limitation, and $\Omega(l \log n)$-time reduction limitation. From this discussion, the computation of the sum and the prefix-sums 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 [8, 19]. 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 prefix-sums 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 $\frac{nl}{p}$. Finally, 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.

Several techniques for computing the prefix-sums on GPUs have been shown in [8]. 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 to 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 [9] 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 performance of the algorithm has not been evaluated.

This paper is organized as follows. Section 2 reviews the Discrete Memory Machine (DMM) and the Unified Memory Machine (UMM) introduced in our previous paper [15]. In Section 3, we evaluate the computing time of the contiguous memory access to 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 prefix-sums algorithm, which runs in $O(\frac{n \log n}{w} + \frac{nl \log n}{p} + l \log n)$ time units. Finally, we show an optimal parallel prefix-sums algorithm running in $O(\frac{n}{w} + \frac{nl}{p} + l \log n)$ time units. Section 8 offers conclusion of this paper.

2 Parallel Memory Machines: DMM and UMM

The main purpose of this section is to review the Discrete Memory Machine (DMM) and the Unified Memory Machine (UMM). introduced in our previous paper [15].

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, it takes $k+l-1$ time units to complete k access requests to a particular bank.

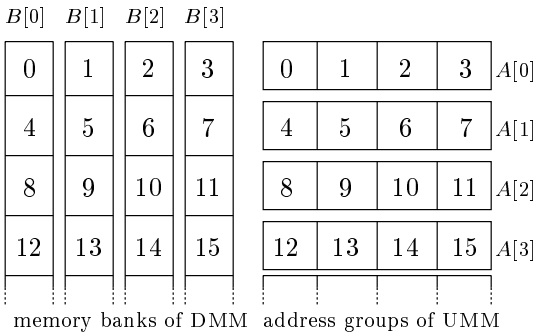

Fig. 2. Banks and address groups for $w = 4$

We assume that p threads are partitioned into $\frac{p}{w}$ groups of w threads called *warps*. More specifically, p threads are partitioned into $\frac{p}{w}$ warps $W(0), W(1), \ldots, W(\frac{p}{w}-1)$ such that $W(i) = \{T(i \cdot w), T(i \cdot w+1), \ldots, T((i+1) \cdot w-1)\}$ $(0 \leq i \leq \frac{p}{w}-1)$. Warps are dispatched for memory access in turn, and w threads in a warp try to access the memory in the same time. In other words, $W(0), W(1), \ldots, W(w-1)$ are dispatched in a round-robin manner if at least one thread in a warp requests memory access. If no thread in a warp needs memory access, such warp is not dispatched for memory access. When $W(i)$ is dispatched, w thread in $W(i)$ 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.

For the reader's benefit, let us evaluate the time for memory access using Figure 3 on the DMM for $p = 8$, $w = 4$, and $l = 3$. In the figure, $p = 8$ threads are partitioned into $\frac{p}{w} = 2$ warps $W(0) = \{T(0), T(1), T(2), T(3)\}$ and $W(1) = \{T(4), T(5), T(6), T(7)\}$. As illustrated in the figure, 4 threads in $W(0)$ try to access $m[0], m[1], m[6]$, and $m[10]$, and those in $W(1)$ try to access $m[8], m[9], m[14]$, and $m[15]$. The time for the memory access are evaluated under the assumption that memory access are processed by imaginary l pipeline stages with w registers each as illustrated in the figure. Each pipeline register in the first stage receives memory access request from threads in an dispatched warp. Each i-th $(0 \leq i \leq w-1)$ pipeline register receives the request to the i-th memory bank. In each time unit, a memory request in a pipeline register is moved to the next one. We assume that the memory access completes when the request reaches the last pipeline register.

Note that, the architecture of pipeline registers illustrated in Figure 3 are imaginary, and it is used only for evaluating the computing time. The actual architecture should involves a multistage interconnection network [6, 12] or sorting network [2, 3], to route memory access requests.

Let us evaluate the time for memory access on the DMM. First, access request for $m[0], m[1], m[6]$ are sent to the first stage. Since $m[6]$ and $m[10]$ are in the same bank $B[2]$, their memory requests cannot be sent to the first stage in the same time. Next, the $m[10]$ is sent to the first stage. After that, memory access requests for $m[8], m[9], m[14], m[15]$ are sent in the same time, because they are in different memory banks. Finally, after $l - 1 = 2$ time units, these memory requests are processed. Hence, the DMM takes 5 time units to complete the memory access.

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 in 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 access to the memory in turn.

Again, let us evaluate the time for memory access using Figure 3 on the UMM for $p = 8$, $w = 4$, and $l = 3$. The memory access requests by $W(0)$ are in three address groups. Thus, three time units are necessary to send them to the first stage. Next, two time units are necessary to send memory access requests by $W(1)$, because they are in two address groups. After that, it takes $l - 1 = 2$ time units to process the memory access requests. Hence, totally $3 + 2 + 2 = 7$ time units are necessary to complete all memory access.

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 [15]. Suppose that an array a of size n $(\geq p)$ is given. We use p threads to access to all of n memory cells in a such that each thread accesses to $\frac{n}{p}$ memory cells. Note that "accessing to" can be "reading

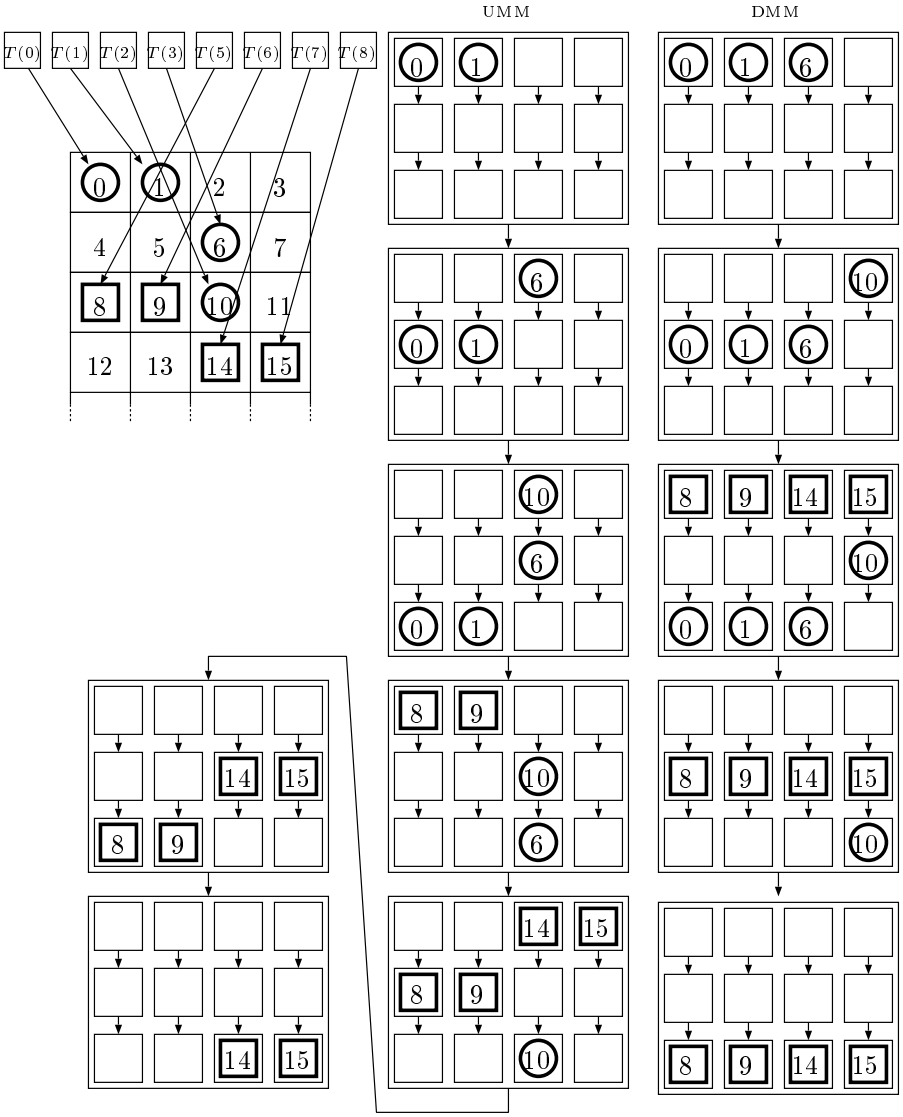

Fig. 3. An example of memory access

from" or "writing in." Let $a[i]$ $(0 \leq i \leq n-1)$denote the i-th memory cells 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 to $a[p \cdot t+i]$

We will evaluate the computing time. For each t $(0 \leq t \leq \frac{n}{p}-1)$, p threads access to p memory cells $a[pt], a[pt+1], \ldots, a[p(t+1)-1]$. This memory access is performed by $\frac{p}{w}$ warps in turn. More specifically, first, w threads in $W(0)$ access to $a[pt], a[pt+1], \ldots, a[pt+w-1]$. After that, p threads in $W(1)$ access to $a[pt+w], a[pt+w+1], \ldots, a[pt+2w-1]$, and the same operation is repeatedly performed. In general, p threads in $W(j)$ $(0 \leq j \leq \frac{p}{w}-1)$ accesses to $a[pt+jw], a[pt+jw+1], \ldots, a[pt+(j+1)w-1]$. Since w memory cells 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 memory cells 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 $\frac{p}{w}$ 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 to p memory cells $a[pt], a[pt+1], \ldots, a[p(t+1)-1]$ 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 using p threads on the UMM and the DMM with width w and latency l.*

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 sums of pairs. We implement this technique to perform contiguous memory access. The details are spelled out as follows:

[**Optimal algorithm for computing the sum**]
for $t \leftarrow m-1$ down to 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.

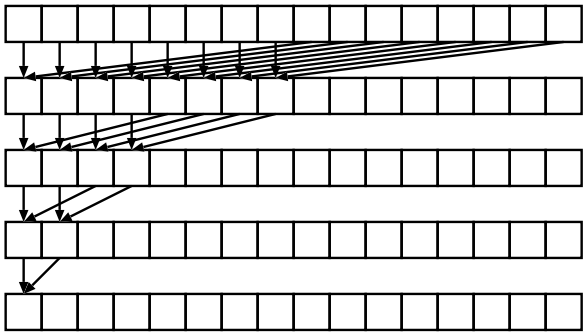

Fig. 4. Illustrating the summing algorithm for n numbers

We assume that p threads 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 $a[0], a[1], \ldots, a[2^t-1]$,

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. We will 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.

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*.

Each thread can perform a binary operation such as addition in a time unit. If at least one of the two operands of a binary operation is stored in the shared memory, it takes at least l time units to obtain the resulting value. Clearly, addition operation must be performed $n-1$ times to compute the sum of n numbers. The computation of the sum using addition is represented using a binary tree with n leaves and $n-1$ internal nodes. The root of the binary tree corresponds to the sum. From basic graph theory results, there exists a path from the root to a leaf, which has at least $\log n$ internal nodes. The addition corresponds to each internal node takes l time units. Thus, it takes at least $\Omega(l \log n)$ time to compute the sum, regardless of the number p of threads. We call the $\Omega(l \log n)$-time lower bound *the reduction limitation*.

From the discussion above, we have,

Theorem 1. *Both the DMM and the UMM with p threads, width w, and latency l takes at least $\Omega(\frac{n}{w} + \frac{nl}{p} + l \log n)$ time units to compute the sum of n numbers.*

From Theorem 1, the parallel algorithm for commuting the sum shown for Lemma 2 is optimal.

Let us discuss about 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 is of the sum algorithm for Lemma 2 is $O(\frac{n}{w})$. Note that the memory machine models have wl imaginary registers. Since more than one memory requests by a thread

can not be stored in imaginary pipeline registers, $wl \leq p$ 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 imaginary 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 [8, 9], 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 $p-1$ 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.

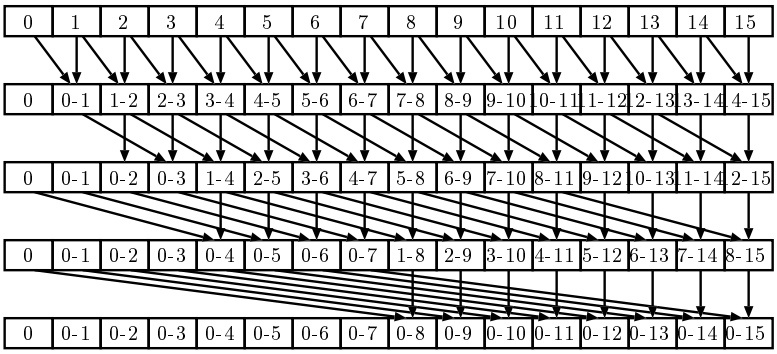

Fig. 5. Illustrating the naive prefix-sums algorithm for n numbers

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 $(0 \leq t \leq p-1)$: can be done by

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

Each of the three operations can be done by contiguous memory access for $n-2^t$ memory cells. 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:

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

Thus, we have,

Lemma 3. *The naive prefix-sums algorithm runs in $O(\frac{n\log n}{w}+\frac{nl\log n}{p})$ time units using p threads on the DMM and on the UMM with width w and latency l.*

Clearly, from Theorem 1, the naive algorithm is not optimal.

7 Our optimal prefix-sums algorithm

This section shows an optimal prefix-sums algorithm running in $O(\frac{n\log n}{w}+\frac{nl}{p}+l\log n)$ time units. We use $m-1$ arrays $a_1, a_2, \ldots a_{m-1}$ as work space. Each a_t $(1 \le t \le m-1)$ can store 2^t-1 numbers. Thus, the total size of the $m-1$ arrays is no more than $(2^1-1)+(2^2-1)+\cdots+(2^{m-1}-1) = 2^m-m < n$. 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 $m-1$ arrays. The second stage uses interval sums in the $m-1$ arrays to compute the resulting prefix-sums. The details of the first stage is spelled out as follows.

[Compute the interval sums]
for $t \leftarrow m-1$ down to 1 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 illustrated how the interval sums are computed. When this program terminates, each $a_t[i]$ $(1 \le t \le m-1, 0 \le i \le 2^t-2)$ 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 $t \leftarrow 1$ to $m-1$ do
  for $i \leftarrow 0$ to 2^t-2 do in parallel
    begin
      $a_{t+1}[2\cdot i+1] \leftarrow a_t[i]$
      $a_{t+1}[2\cdot i+2] \leftarrow a_{t+1}[2\cdot i+2] + a_t[i]$
    end
$a_m[n-1] \leftarrow a_m[n-2] + a_m[n-1]$

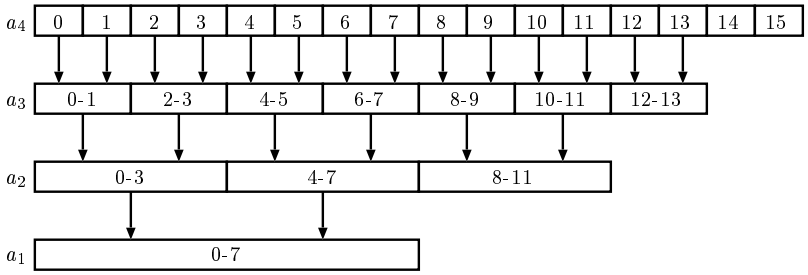

Fig. 6. Illustrating the computation of interval sums in $m-1$ arrays.

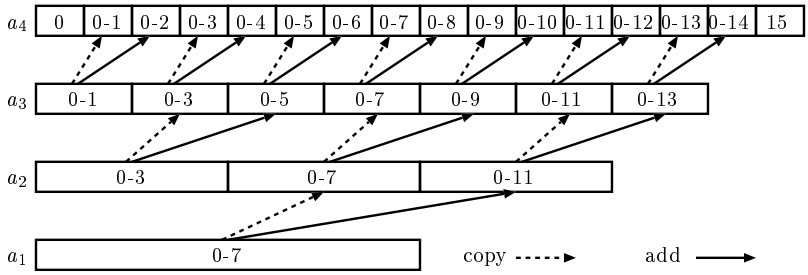

Fig. 7. Illustrating the computation of interval sums in $m-1$ arrays.

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

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

Since every two addresses is accessed, these four 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

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

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

- reading from $a_t[0], a_t[1], \ldots, a_t[2^t-2]$,

- reading from $a_{t+1}[2], a_{t+1}[4], \ldots, a_{t+1}[2^{t+1}-2]$,
- writing in $a_{t+1}[1], a_{t+1}[3], \ldots, a_{t+1}[2^{t+1}-3]$, and
- writing in $a_{t+1}[2], a_{t+1}[4], \ldots, a_{t+1}[2^{t+1}-2]$.

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 2. *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 1, 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 $p \leq k \leq n$. We partition the input a with n numbers into $\frac{n}{k}$ groups with k ($\geq p$) numbers each. Each t-th group ($0 \leq t \leq \frac{n}{k} - 1$) 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($t > 0$) $a[tk] \leftarrow a[tk] + a[tk-1]$
  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 $O(\frac{k}{w} + \frac{kl}{p} + l \log k)$. 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 1. *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.*

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

8 Conclusion

The main contribution of this paper is to show that an optimal parallel prefix-sums algorithm that runs in $O(\frac{n}{w} + \frac{nl}{p} + l \log n)$ time units. This algorithm uses work space of size $\min(n, p \log p, wl \log(wl))$.

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. Aho, A.V., Ullman, J.D., Hopcroft, J.E.: Data Structures and Algorithms. Addison Wesley (1983)
2. Akl, S.G.: Parallel Sorting Algorithms. Academic Press (1985)
3. Batcher, K.E.: Sorting networks and their applications. In: Proc. AFIPS Spring Joint Comput. Conf. vol. 32, pp. 307–314 (1968)
4. Flynn, M.J.: Some computer organizations and their effectiveness. IEEE Transactions on Computers C-21, 948–960 (1872)
5. Gibbons, A., Rytter, W.: Efficient Parallel Algorithms. Cambridge University Press (1988)
6. Gottlieb, A., Grishman, R., Kruskal, C.P., McAuliffe, K.P., Rudolph, L., Snir, M.: The nyu ultracomputer – designing an MIMD shared memory parallel computer. IEEE Trans. on Computers C-32(2), 175 – 189 (Feb 1983)
7. Grama, A., Karypis, G., Kumar, V., Gupta, A.: Introduction to Parallel Computing. Addison Wesley (2003)
8. Harris, M., Sengupta, S., Owens, J.D.: Chapter 39. parallel prefix sum (scan) with CUDA. In: GPU Gems 3. Addison-Wesley (2007)
9. Hillis, W.D., Steele, Jr., G.L.: Data parallel algorithms. Commun. ACM 29(12), 1170–1183 (Dec 1986), http://doi.acm.org/10.1145/7902.7903
10. Hwu, W.W.: GPU Computing Gems Emerald Edition. Morgan Kaufmann (2011)
11. Ito, Y., Ogawa, K., Nakano, K.: Fast ellipse detection algorithm using hough transform on the GPU. In: Proc. of International Conference on Networking and Computing. pp. 313–319 (Dec 2011)
12. Lawrie, D.H.: Access and alignment of data in an array processor. IEEE Trans. on Computers C-24(12), 1145– 1155 (Dec 1975)
13. Man, D., Uda, K., Ito, Y., Nakano, K.: A GPU implementation of computing euclidean distance map with efficient memory access. In: Proc. of International Conference on Networking and Computing. pp. 68–76 (Dec 2011)
14. Man, D., Uda, K., Ueyama, H., Ito, Y., Nakano, K.: Implementations of a parallel algorithm for computing euclidean distance map in multicore processors and GPUs. International Journal of Networking and Computing 1, 260–276 (July 2011)
15. Nakano, K.: Simple memory machine models for GPUs. In: Proc. of International Parallel and Distributed Processing Symposium Workshops. pp. 788–797 (May 2012)
16. Nishida, K., Ito, Y., Nakano, K.: Accelerating the dynamic programming for the matrix chain product on the GPU. In: Proc. of International Conference on Networking and Computing. pp. 320–326 (Dec 2011)
17. NVIDIA Corporation: NVIDIA CUDA C best practice guide version 3.1 (2010)
18. NVIDIA Corporation: NVIDIA CUDA C programming guide version 4.0 (2011)
19. Quinn, M.J.: Parallel Computing: Theory and Practice. McGraw-Hill (1994)
20. Uchida, A., Ito, Y., Nakano, K.: Fast and accurate template matching using pixel rearrangement on the GPU. In: Proc. of International Conference on Networking and Computing. pp. 153–159 (Dec 2011)