PAPER Special Section on Foundations of Computer Science

Asynchronous Memory Machine Models with Barrier Synchronization

Koji NAKANO[†], ***Member***

SUMMARY The Discrete Memory Machine (DMM) and the Unified Memory Machine (UMM) are theoretical parallel computing models that capture the essence of the shared memory and the global memory of GPUs. It is assumed that warps (or groups of threads) on the DMM and the UMM work synchronously in a round-robin manner. However, warps work asynchronously in real GPUs, in the sense that they are randomly (or arbitrarily) dispatched for execution. The first contribution of this paper is to introduce asynchronous versions of these models in which warps are arbitrarily dispatched. In addition, we assume that threads can execute the "syncthreads" instruction for barrier synchronization. Since the barrier synchronization operation may be costly, we should evaluate and minimize the number of barrier synchronization operations executed by parallel algorithms. The second contribution of this paper is to show a parallel algorithm to the sum of n numbers in optimal computing time and few barrier synchronization steps. Our parallel algorithm computes the sum of n numbers in $O(\frac{n}{w} + l \log n)$ time units and $\boldsymbol{O(\frac{\log l}{\log w} + \log\log w)}$ barrier synchronization steps using wl threads on the asynchronous UMM with width w and latency l. Since the computation of the sum takes at least $\Omega(\frac{n}{w} + l \log n)$ time units, this algorithm is time optimal. Finally, we show that the prefix-sums of n numbers can also be computed in $O(\frac{n}{w} + l \log n)$ time units and $\boldsymbol{O(\frac{\log l}{\log w} + \log\log w)}$ barrier synchronization steps using $\boldsymbol{wl}$ threads.

key words: *memory machine models, parallel algorithms, contiguous memory access, asynchronous models, GPU, CUDA*

1. Introduction

The GPU (Graphics Processing Unit), is a specialized circuit designed to accelerate computation for building and manipulating images [9], [10], [16], [25], [27]. 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 [9], [21], [22]. NVIDIA provides a parallel computing architecture called *CUDA* (Compute Unified Device Architecture) [24], 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 [17], 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* [24]. The shared memory is an extremely fast on-chip memory with lower capacity, say, 16-64 Kbytes. 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 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 [16], [17], [23]. 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 of CUDA should access 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 coalesced access when they access the global memory.

In our previous paper [20], we have introduced two parallel computing 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 is 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 multicore processors which can execute multiple threads in parallel and/or in time-sharing manner. Threads are executed in SIMD [5] fashion, and the threads 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 $(i \bmod w)$-th MB, 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,

Manuscript received January 1, 2010.
Manuscript revised January 1, 2010.

[†]The authors are with Department of Information Engineering, Hiroshima University, Kagamiyama 1-4-1, Higashi-Hiroshima, 739-8527, JAPAN

DOI: 10.1587/trans.E0.??.1

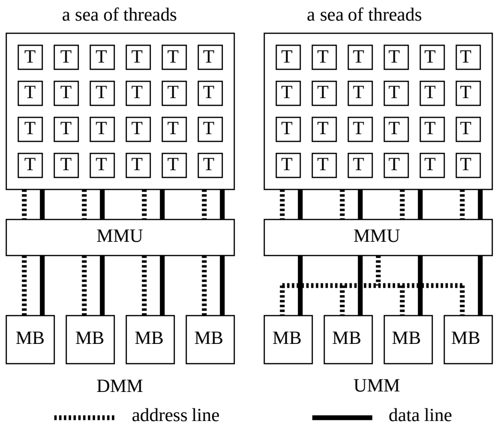

Fig. 1 The architectures of the DMM and the UMM

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.

Since we believe that the memory machine models are promising as computing models for GPUs, we have published several efficient algorithms on the DMM and the UMM [12], [18], [19]. For example, we have shown in [19] that the sum and 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 both on the DMM and the UMM and have proved that the computation of the sum takes at least $\Omega(\frac{n}{w} + \frac{nl}{p} + l\log n)$ time units. Thus, the algorithms for the sum and the prefix-sums are time optimal. Further, in our previous paper [20], we have presented offline permutation algorithms on the DMM and the UMM. We also implemented the offline permutation algorithm on the DMM and showed that theoretical analysis of the performance on the DMM provides very good approximation of the CUDA C implementation of offline permutation algorithm [12]. This fact implies that the DMM is a good theoretical model for computation using the shared memory on GPUs.

Suppose that we use p threads $T(0), T(1), \ldots, T(p-1)$. Threads on the DMM and the UMM are partitioned into $\frac{p}{w}$ groups of w threads called *warps*. Let $W(0), W(1), \ldots, W(\frac{p}{w}-1)$ denote the $\frac{p}{w}$ groups. In our previous papers [12], [18], [19], it is assumed that threads on the DMM and the UMM works *synchronously* in the sense that warps are dispatched for memory access from $W(0)$ to $W(\frac{p}{w}-1)$ in turn by a round-robin manner. Note that a parameter w is used to denote the number of memory banks and the number of threads in a warp. Since each warp has 32 threads and the shared memory are implemented in 32 memory banks in CUDA compute capability 3.x [24], this assumption makes sense.

The first contribution of this paper is to extend memory machine models presented in our previous paper [20] for more realistic parallel computing models. More specifically, we assume that threads work *asynchronously* in the sense that warps are dispatched for memory access arbitrarily. The scheduler arbitrarily selects one of the warps and dispatches it for memory access. In addition, we assume that threads can execute a special instruction *syncthreads* for the purpose of barrier synchronization. In NVIDIA GPUs, `__syncthreads()` instruction is supported for synchronization of all threads in a CUDA block. However, it takes $\frac{p}{16}$ clock cycles to synchronize p threads [24]. Also, for the purpose of synchronization of threads in multiple blocks, we need to separate an algorithm into different kernel calls [24]. Hence, barrier synchronization may be costly. Thus, it is very important to evaluate the number of barrier synchronization steps executed by algorithms and to minimize it. Note that, parallel algorithms on the asynchronous models must work correctly for any worst choice of warps by a malicious scheduler. Also, the performance including the computing time should be evaluated for the case of worst choice of warps.

The second contribution of this paper is to show a time-optimal summing algorithm on the asynchronous version of the UMM. Our summing algorithm computes the sum of n numbers in $O(\frac{n}{w} + l\log n)$ time units and $\boldsymbol{O(\frac{\log l}{\log w} + \log\log w)}$ barrier synchronization steps using wl threads on the asynchronous UMM with width w and latency l. Quite surprisingly, the number of barrier synchronization steps and the number of threads are independent of n. Even if the input size n is quite large, our parallel algorithm computes the sum in optimal time units and a fixed number of syncthreads using a fixed number of threads. Our summing algorithm uses many intermediate algorithms. Table 1 summarizes our summing algorithms presented in this paper.

The prefix-sum computation is a task to compute $a[0] + a[1] + \cdots + a[i]$ for all i $(0 \le i \le n-1)$ for a given array a of n numbers. The third contribution of this paper is to extend our parallel algorithm for the sum to compute the prefix-sums. We modify each algorithm Simple, Tree, Simple-Tree, and Hybrid in Table 1 to compute the prefix-sums. Also, since the asynchronous UMM is less powerful than the asynchronous DMM, our summing and prefix-summing algorithms on the asynchronous UMM are also work for the asynchronous DMM.

The computation of the sum and the prefix-sums is very important, in particular, in the area of parallel computing, although it is trivial in sequential computation. For example, the sum computation is frequently used in linear algebra such as matrix multiplication. The prefix-sums computation are used in many parallel algorithms including graph algorithms [6], geometric algorithms [3], discrete optimization problem [13], among others. Thus, parallel algorithms for the sum and the prefix-sums on various parallel computing models are explained in the first or the second chapters of most text books of parallel computing [11], [15], [26]. On the GPUs, the prefix-sums computations are used for interactive rendering of glossy environment reflections and refractions [8]. Also, we have shown parallel algorithms of ant colony optimization for the traveling salesman problem on the GPU [28]. Our parallel algorithms uses the prefix-

Table 1 Performance of parallel algorithm for computing the sum and the prefix-sums

algorithms	time units	threads	syncthreads	time optimality
Simple	$O(\frac{n}{w} + l\log n)$	$\frac{n}{2}$	$O(\log \frac{n}{w})$	optimal
Tree	$O(\frac{n}{w}\log w + l\log n)$	n	$O(\frac{\log n}{\log w})$	overhead of factor $\log w$
Simple-Tree	$O(\frac{n}{w} + l\log n)$	$\frac{n}{2}$	$O(\frac{\log n}{\log w} + \log\log w)$	optimal
Hybrid	$O(\frac{n}{w} + l\log n)$	wl	$O(\frac{\log l}{\log w} + \log\log w)$	optimal

sums computation, which dominates the total computing time. Since many parallel algorithms involves the computation of the sum and the prefix-sums, it is very important to show efficient parallel algorithms for them on the memory machine models.

Note that our algorithms for the sums and the prefix-sums are designed for the asynchronous UMM. Although the asynchronous UMM is inspired by the architecture of CUDA-enabled GPUs, it is a simple abstract parallel computing model and independent of actual implementation of architectures. If the cost of barrier synchronization is small enough, then our algorithm runs in $O(\frac{n}{w} + l\log n)$ time. This computing time is optimal, and thus, no algorithm can run faster than this computing time. If the barrier synchronization cost is very large and the time for the barrier synchronization dominates the computing time, our algorithm runs in $O(\frac{\log l}{\log w} + \log\log w)$ time. The global memory access latency of CUDA-enabled GPUs is several hundred clock cycles and the number of threads in a warp is 32 [23]. The value of $\frac{\log l}{\log w} + \log\log w$ for $l = 10000$ and $w = 32$ is no more than 6. Also, this value is independent of the size n input. Hence, we can say that the algorithm performs a small fixed number of barrier synchronization steps and even if the barrier synchronization is costly, the time for it in our algorithm is negligible for enough large n.

2. The Asynchronous Unified Memory Machine

We first define *the Unified Memory Machine (UMM)* of width w and latency l. 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 accessed at the same time. However, if they are in different address groups, one time unit is necessary for each of the groups. 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 access requests destined for k address groups. 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 at least l time units to send a new memory access request.

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 $W(i) = \{T(i\cdot w), T(i\cdot w+1), \ldots, T((i+1)\cdot w - 1)\}$ $(0 \le i \le \frac{p}{w} - 1)$. Warps are dispatched for memory access in turn and w threads in a warp try to access the memory at the same time. We define two assumptions *synchronous manner* and *asynchronous manner* in terms of the warp dispatch. In the synchronous manner, $W(0), W(1), \ldots, W(\frac{p}{w}-1)$ are dispatched in a round-robin manner if at least one thread in a warp requests memory access. More specifically, suppose that every thread executes $\mathcal{T}$ instructions. In the synchronous manner, warps work equally as follows:

```
[Synchronous Model]
for $t \leftarrow 0$ to $\mathcal{T} - 1$ do
  for $i \leftarrow 0$ to $\frac{p}{w} - 1$ do
    Every thread in $W(i)$ executes an instruction.
```

On the other hand, in asynchronous operations, one of the warps is dispatched and executed as follows:

```
[Asynchronous Model]
for $t \leftarrow 0$ to $\frac{\mathcal{T}p}{w} - 1$ do
  Arbitrarily select a warp $W(i)$ to be executed.
  Every thread in $W(i)$ executes an instruction.
```

Note that, in the asynchronous model, if all threads in a warp $W(i)$ have no instruction to be executed, such warp $W(i)$ is not selected. For example, if threads in $W(i)$ have just sent memory access requests and they are waiting for completion of memory access, $W(i)$ is not selected. Such warp $W(i)$ will be selected after the completion of memory access.

We also assume that, for the purpose of barrier synchronization, all threads can execute the *syncthreads* instruction. Suppose that at least one of the p threads executes *syncthreads*. After that, all threads executing *syncthreads* have been blocked until all threads execute *syncthreads*. Once all threads execute *syncthreads*, they restart executing instructions.

For the reader's benefit, let us evaluate the time for memory access on the UMM for $p = 8$, $w = 4$, and $l = 5$ using Figure 2. 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 $m[7], m[5], m[15]$, and $m[0]$, and those in $W(1)$ try to access $m[10], m[11], m[12]$, and $m[9]$. The time for the memory access is evaluated under the assumption that memory access are processed by imaginary l pipeline stages with w registers each as illustrated in the figure. We assume that the memory access completes when the request reaches the last stage of the pipeline register. Note that, the architecture of pipeline registers illustrated in Figure 2 are imaginary, and it is used only for evaluating the computing time. The actual architecture should involves a multistage interconnection network [7], [14], sorting network [2], [4], or Network-on-Chip (NoC) to route memory access requests.

Let us evaluate the time for memory access on the UMM with width $w = 4$ and latency $l = 5$. First, access request for $m[7], m[5], m[15]$ and $m[0]$ by $W(0)$ are sent to the first stage. Memory requests by $W(0)$ are partitioned into address groups, the memory requests by $W(0)$ occupy 3 stages. After that, those by $W(1)$ occupies 2 stages. Finally, after $l - 1 = 4$ time units, these memory requests are processed. Hence, the UMM takes $3 + 2 + 4 = 9$ time units to complete the memory access.

3. Contiguous Memory Access

The main purpose of this section is to show the contiguous memory access on the asynchronous UMM. The evaluation of the computing time for the contiguous access on the synchronous DMM and the synchronous UMM is not difficult [19], [20]. However, that for the asynchronous version is complicated.

Suppose that an array a of size n ($\geq p$) is given. We use p threads to access all of n memory cells in a such that each thread accesses $\frac{n}{p}$ memory cells. Note that "accessing" can be "reading from" or "writing in." The contiguous memory 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
    $T(i)$ accesses $\boldsymbol{a[t \cdot p + i]}$
```

Let evaluate the computing time. Each warp $W(j)$ $(0 \leq j \leq \frac{p}{w} - 1)$ with w threads accesses w memory cells in the same address group. Note that p threads are partitioned into $\frac{p}{w}$ warps, and each thread sends a memory access request $\frac{n}{p}$ times. We will evaluate the computing time for the following two cases:

Case 1: $\boldsymbol{\frac{p}{w} < l}$ First, one of the warps is randomly dispatched and sends memory access requests. After a warp sends requests, it will not be selected at least l time units. Thus, all of the $\frac{p}{w}$ warps are dispatched in the first $\frac{p}{w}$ time units. Each warp takes l time units to complete the memory access, Thus, the second memory access is started at time l. Figure 3 illustrates how contiguous memory access is performed when $\frac{p}{w} < l$. Contiguous memory access requests by $\frac{p}{w}$ warps are repeatedly sent $\frac{n}{p}$ times. Thus, it takes $\frac{p}{w} + l \cdot \frac{n}{p} = O(\frac{p}{w} + \frac{nl}{p})$ time units for the contiguous memory access.

Case 2: $\boldsymbol{\frac{p}{w} \geq l}$ Each of the $\frac{p}{w}$ warps sends memory access requests $\frac{n}{p}$ times. Hence, totally they send memory access requests $\frac{n}{p} \cdot \frac{p}{w} = \frac{n}{w}$ times. Clearly, if at least l warps have not completed memory access, they can send memory access requests continuously. On the other hand, if no warp send memory access requests in a particular time unit, then less than l warps still have memory access requests to be sent. If this is the case, less than l such warps can send memory access requests exactly once in l time units. Since each warp sends memory access $\frac{n}{p}$ times, it takes at most $l \cdot \frac{n}{p} = O(\frac{nl}{p})$ time units for less than l such warps to complete the memory access requests. Therefore, the contiguous memory access can be completed in $O(\frac{n}{w} + \frac{nl}{p})$ time units.

Thus, we have,

Lemma 1: The contiguous access to an array of size n can be done in $O(\frac{n}{w} + \frac{nl}{p})$ time units with 0 barrier synchronization step using p threads on the asynchronous UMM with width w and latency l.

4. Summing algorithms on the UMM

4.1 A simple summing algorithm

The main purpose of this subsection is to show a simple parallel algorithm for computing the sum on the memory machine models. The summing algorithm presented in this subsection is the essentially same as one presented in [19] on the synchronous DMM and the synchronous UMM.

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 using $\frac{n}{2}$ threads. The details are spelled out as follows:

```
[**Summing Algorithm Simple**]
for $t \leftarrow m - 1$ down to 0 do
  begin
    for $i \leftarrow 0$ to $2^t - 1$ do in parallel
      $T$(i) performs $a[i] \leftarrow a[i] + a[i + 2^t]$
    if($2^t > w$) *syncthreads*
  end
```

Let us evaluate the computing time. 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 $O(\frac{2^t}{w} + \frac{2^t l}{2^t}) = O(\frac{2^t}{w} + l)$ time using 2^t threads on the UMM with width w and latency l from Lemma 1. Thus, the total computing time is

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

Barrier synchronization *syncthreads* is executed $m - \log w - 1 = O(\log \frac{n}{w})$ times. Thus, we have,

Lemma 2: Summing Algorithm Simple computes the sum of n numbers in $O(\frac{n}{w} + l \log n)$ time units and $O(\log \frac{n}{w})$ barrier synchronization steps using $\frac{n}{2}$ threads on the UMM with

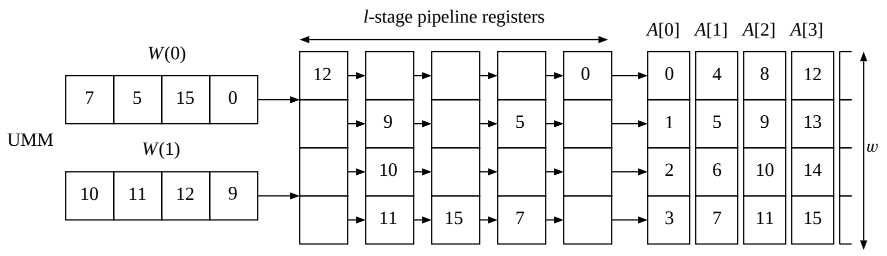

Fig. 2 Examples of memory access on the UMM

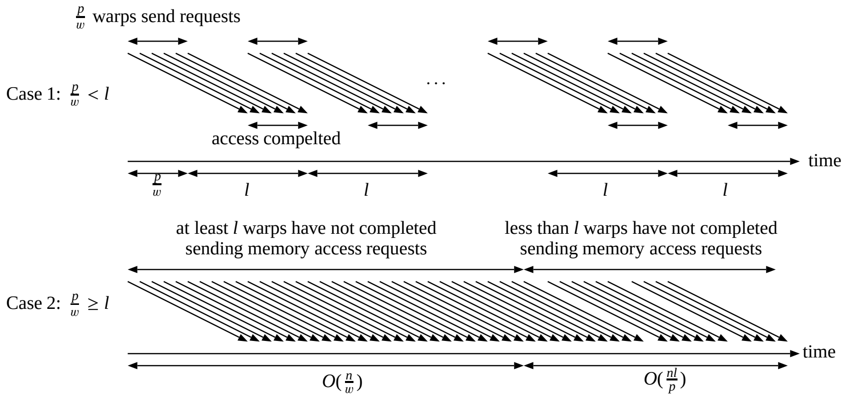
p/w warps send requests
Case 1: p/w < l
access compelted
time
at least l warps have not completed sending memory access requests
less than l warps have not completed sending memory access requests
Case 2: p/w ≥ l
O(n/w)
O(nl/p)

Fig. 3 Contiguous memory access

width w and latency l.

Suppose that Summing Algorithm Simple is executed for w numbers. If this is a case, the sum can be computed in $O(l \log w)$ time units and 0 barrier synchronization step using one warp with $\frac{w}{2}$ threads on the UMM. For later reference, we call this algorithm Summing Algorithm One-Warp.

4.2 A summing algorithm based on a w-ary tree

We need to use more than w threads to obtain a time-optimal summing algorithm. However, if we use more than w threads, barrier synchronization is necessary. This subsection shows a summing algorithm using more than w threads. The goal of the summing algorithm shown in this subsection is to minimize the number of barrier synchronization steps.

For simplicity, we assume that $n = w^k$ for some integer k. We can build a w-ary tree with n leaves, each of which corresponds to an input number. The n leaves are partitioned into $\frac{n}{w}$ groups and each group is connected to an first-level internal node. Thus, we have $\frac{n}{w}$ first-level internal nodes. The first-level internal nodes are partitioned into $\frac{n}{w^2}$ groups and each group is connected to an second-level internal node. Continuing similarly, we can build a w-ary tree with k-levels.

The computation of the sum is performed from leaves to the root. The sum of each group of the leaves is computed by a warp. The resulting sum is stored in a first level internal node. After that, the sum of each group in the first level internal nodes is computed by a warp, and the resulting sum is stored in a second level internal node. Continuing similarly,

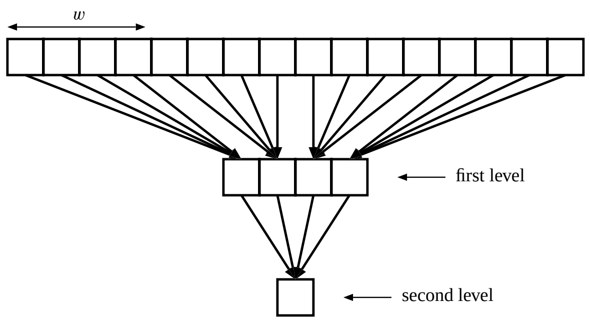

Fig. 4 A summing algorithm based on a w-ary tree

we can obtain the sum at the root.

Let a_0 denote the input array, and $a_1, a_2, \ldots, a_k$ be working space each of which corresponds to internal nodes of the tree. Each a_i ($1 \leq i \leq k$) can store $\frac{n}{w^i}$ numbers. Summing Algorithm Tree computes the resulting sum in $a_k[0]$ as follows:

```
[Summing Algorithm Tree]
for $t \leftarrow 1$ to $k$ do
  for $i \leftarrow 0$ to $\frac{n}{w^t} - 1$ do in parallel
    begin
      $W(i)$ computes $\boldsymbol{a_t[i] \leftarrow a_{t-1}[i \cdot w] + a_{t-1}[i \cdot w + 1]}$
        $+ \cdots + a_{t-1}[(i+1) \cdot w - 1]$
        using Summing Algorithm One-Warp
      syncthreads
    end
```

Let us evaluate the computing time for each value of t.

First, when $t = k$, Summing Algorithm One-Warp is used to compute the sum of w numbers using one warp. This takes $O(l \log w)$ time units. When $t = k - 1$, each of w warps executes Summing Algorithm One-Warp to compute the sum of w numbers independently. In Summing Algorithm One-Warp $\frac{w}{2}$ threads in a single warp performs memory access to one address group $O(\log w)$ times. Hence, we can simply think that w^2 threads in w warps perform the contiguous memory access of w^2 numbers $O(\log w)$ times. From Lemma 1, it takes $O(\frac{w^2}{w} + \frac{w^2 l}{w^2}) \cdot O(\log w) = O((w + l) \log w)$ time units. Let us consider the general case for $t = k - j$ $(0 \leq j \leq k - 1)$. If this is the case, each of w^j warps executes Summing Algorithm One-Warp to compute the sum of w numbers independently. Similarly, we can think that w^{j+1} threads in w^j warps perform the contiguous memory access to w^{j+1} numbers $O(\log w)$ times. Hence, it takes $O(\frac{w^{j+1}}{w} + \frac{w^{j+1} l}{w^{j+1}}) \cdot O(\log w) = O((w^j + l) \log w)$ time units. Therefore, the total computing time of Summing Algorithm Tree is:

$$\boldsymbol{\sum_{j=0}^{k-1} O((w^j + l) \log w) = O((w^{k-1} + kl) \log w) = O(\frac{n}{w} \log w + l \log w)}$$

from $k = \frac{\log n}{\log w}$. Also, Summing Algorithm Tree performs *syncthreads* $k = \frac{\log n}{\log w}$ times. Thus, we have,

Lemma 3: Summing Algorithm Tree compute the sum of n numbers in $O(\frac{n}{w} \log w + l \log n)$ time units with $O(\frac{\log n}{\log w})$ barrier synchronization steps using n threads on the UMM with width w and latency l.

4.3 A time-optimal algorithm for computing the sum using few barrier synchronization steps

We can obtain time optimal summing algorithms with compensation of few additional barrier synchronization steps. This subsection is devoted to show such time optimal summing algorithms.

Suppose that Summing Algorithm Simple is executed for $t = m - 1, m - 2, \ldots, m - \log\log w$. It should be clear that the interim sum are stored in $a[0], a[1], \ldots, a[\frac{n}{\log w} - 1]$. After that, the sum of these $\frac{n}{\log w}$ numbers are computed by Summing Algorithm Simple. The details are spelled out as follows:

[**Summing Algorithm Simple-Tree**]
```
for $t \leftarrow m - 1$ down to $m - \log\log w$ do
  begin
    for $i \leftarrow 0$ to $2^t - 1$ do in parallel
      $T(i)$ performs $a[i] \leftarrow a[i] + a[i + 2^t]$
    syncthreads
  end
Use Summing Algorithm Tree to compute the sum $a[0]+$
  $a[1]+ \cdots + a[\frac{n}{\log w} - 1]$.
```

Let us evaluate the computing time. As we have discussed, Summing Algorithm Simple takes $O(\frac{2^t}{w} + l)$ time units for each t. Thus, the execution of Summing Algorithm Simple for $t = m - 1, m - 2, \ldots, m - \log\log w$ takes

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

Also, it has $\log\log w$ barrier synchronization steps. After that, Summing Algorithm Tree is executed for $\frac{n}{\log w}$ numbers. From Lemma 3, it takes

$$O(\frac{\frac{n}{\log w}}{w} \log w + l \log \frac{n}{\log w})) = O(\frac{n}{w} + l \log n)$$

time units. Further, it has $O(\frac{\log \frac{n}{\log w}}{\log w}) = O(\frac{\log n}{\log w})$ synchronization steps. Thus, we have,

Theorem 4: Summing Algorithm Simple-Tree computes the sum of n numbers in $O(\frac{n}{w} + l \log n)$ time units with $O(\frac{\log n}{\log w} + \log\log w)$ barrier synchronization steps using n threads on the UMM with width w and latency l.

We will show that, the number of syncthreads can be independent of the number n of input numbers. Suppose that input n $(\geq wl)$ numbers are stored in an array a of size $\frac{n}{wl} \times wl$ ($\frac{n}{wl}$ rows and wl columns). First, we assign one thread to each column and compute the column-wise sum. After that, the sum of the column-wise sums using Summing Algorithm Simple-Tree. The details are spelled out as follows:

[**Summing Algorithm Hybrid**]
```
for $t \leftarrow 1$ to $\frac{n}{wl} - 1$ do
  for $i \leftarrow 0$ to $wl - 1$ do in parallel
    $T(i)$ performs $a[0][i] \leftarrow a[0][i] + a[t][i]$
Use Summing Algorithm Simple-Tree to compute the sum
  $a[0][0] + a[0][1] + \cdots + a[0][wl - 1]$.
```

Let us evaluate the computing time. The computation of the column-wise sums performs contiguous access. Thus, from Lemma 1, it takes $O(\frac{n}{w} + \frac{nl}{wl}) = O(\frac{n}{w} + l)$ time units. After that, Summing Algorithm Simple-Tree is executed for wl numbers. From Theorem 4, it takes $O(\frac{wl}{w} + l \log(wl)) = O(l \log n)$ time units using $O(\frac{\log(wl)}{\log w} + \log\log w) = O(\frac{\log l}{\log w} + \log\log w)$ barrier synchronization steps. Thus, we have,

Theorem 5: Summing Algorithm Hybrid computes the sum of n numbers in $O(\frac{n}{w} + l \log n)$ time units with $O(\frac{\log l}{\log w} + \log\log w)$ barrier synchronization steps using wl threads on the UMM with width w and latency l.

Since the computation of the sum of n numbers takes at least $\Omega(\frac{n}{w} + \frac{nl}{p} + l \log n)$ time units using p threads on the UMM with width w and latency l [20], Summing Algorithm Hybrid is optimal from $p = wl$.

5. Prefix-sum algorithms on the memory machine models

5.1 A simple prefix-summing algorithm

This subsection shows an algorithm for the prefix-sums running in $O(\frac{n}{w} + l \log n)$ time units using $\frac{n}{2}$ threads. The used technique is essentially the same as one shown in [19]. We use m arrays $a_0, a_1, \ldots a_{m-1}$ as work space, where $n = 2^m$. 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 $2^0 + 2^1 + \cdots + 2^{m-1} = 2^m - 1 = n - 1$. 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 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
 begin
   for $i \leftarrow 0$ to $2^t - 1$ do in parallel
     $T(i)$ performs $a_t[i] \leftarrow a_{t+1}[2 \cdot i] + a_{t+1}[2 \cdot i + 1]$
   if($2^t > w$) syncthreads
 end
```

Figure 5 illustrates how the interval sums are computed. When this program terminates, each $a_t[i]$ $(0 \leq t \leq m-1, 0 \leq i \leq 2^t - 1)$ 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 obtained 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
 begin
   for $i \leftarrow 0$ to $2^t - 1$ do in parallel
     $T(i)$ performs $a_{t+1}[2 \cdot i + 1] \leftarrow a_t[i]$
   for $i \leftarrow 0$ to $2^t - 2$ do in parallel
     $T(i)$ performs $a_{t+1}[2 \cdot i + 2] \leftarrow a_{t+1}[2 \cdot i + 2] + a_t[i]$
   if($2^t > w$) syncthreads
 end
```

Figure 6 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 $a_m[i]$ $(0 \leq i \leq 2^t - 1)$ stores the prefix-sum $a_m[0] + a_m[1] + \cdots + a_m[i]$. We assume that $\frac{n}{2}$ threads are available and evaluate the computing time. The first stage involves the following memory access operations for each t $(0 \leq t \leq m-1)$:

- 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 $O(\frac{2^{t+1}}{w} + \frac{2^{t+1}l}{2^t}) = O(\frac{2^t}{w} + l)$ time units using 2^t threads. Therefore, the total computing time of the first stage is

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

Also, *syncthreads* is executed for $t = m-1, m-2, \ldots,$ and $\log w + 1$. Thus, it is executed at most $m - \log w - 1 = O(\log \frac{n}{w})$ times. The second stage consists of the following memory access operations for each t $(0 \leq t \leq m-1)$:

- 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 $O(\frac{2^{t+1}}{w} + l)$ time units. Hence, the total computing time of the second stage is also $O(\frac{n}{w} + l \log n)$. Further, *syncthreads* is executed $O(\log \frac{n}{w})$ times. Thus, we have,

Lemma 6: The prefix-sums of n numbers can be computed in $O(\frac{n}{w} + l \log n)$ time units with $O(\log \frac{n}{w})$ barrier synchronization steps using $\frac{n}{2}$ threads on the UMM with width w and latency l.

For later reference we call the algorithm for Lemma 6 Prefix-summing Algorithm Simple analogously to Summing Algorithm Simple. Further, we can design Prefix-summing Algorithm One-Warp as a special case of Prefix-summing Algorithm Simple, analgously to Summing Algorithm One-Warp. In other words, Prefix-summing Algorithm One-Warp computes the prefix-sums of w numbers using one warp of $\frac{w}{2}$ threads in $O(l \log w)$ time and 0 barrier synchronization step.

5.2 A prefix-summing algorithm based on a w-ary tree

We will show that a prefix-summing algorithm based on a w-ary tree, which is analogous to the summing algorithm based on it. Similarly, we assume that $n = w^k$ for some integer k. We can build a rooted w-ary tree with n leaves, each of which corresponds to an input number as before.

Let a_k denote the input array, and $a_1, a_2, \ldots, a_{k-1}$ be working space each of which corresponds to internal nodes of the tree. Each a_i $(1 \leq i \leq k)$ can store w^i numbers. Figure 7 illustrates these arrays for $n = 64$, $w = 4$ and $k = 3$. We store the sum of the leaves in each internal node as illustrated in the figure. We use Summing Algorithm One-Warp to compute the sum to be stored in each array. During the computation of the sum, no barrier synchronization is necessary. After the sum computation by a warp terminates, *syncthreads* instruction is executed. The reader should have no difficulty to confirm that the computing time and the number of barrier synchronization steps are equal to those of Summing Algorithm Tree. Thus, the interval sums of n numbers can be computed in $O(\frac{n}{w} \log w + l \log n)$ time units with $O(\frac{\log n}{\log w})$ barrier synchronization steps using n threads on the UMM with width w and latency l.

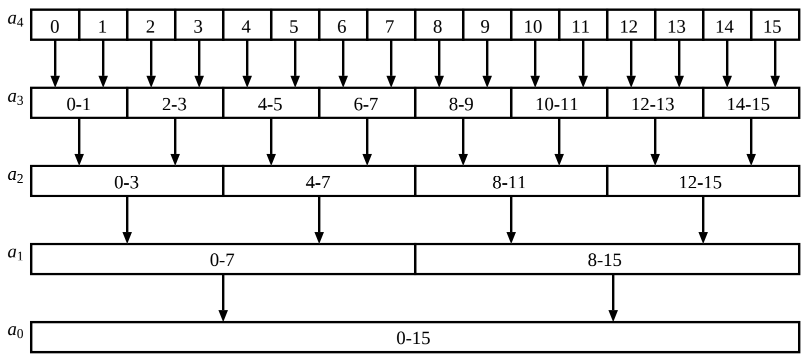

Fig. 5 Illustrating the computation of interval sums in *m* arrays.

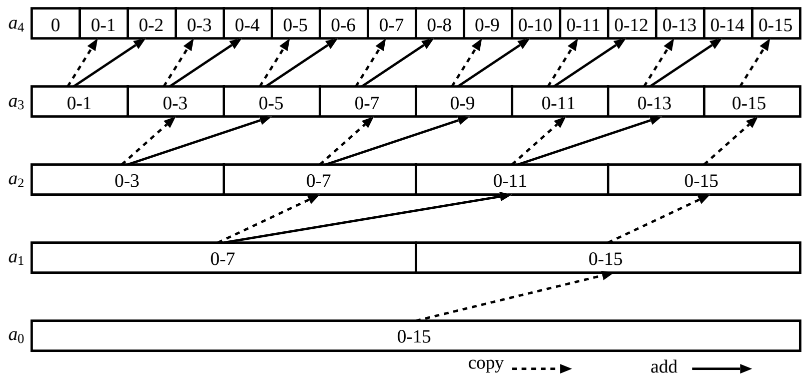

Fig. 6 Illustrating the computation of the sums of the interval sums in *m* arrays.

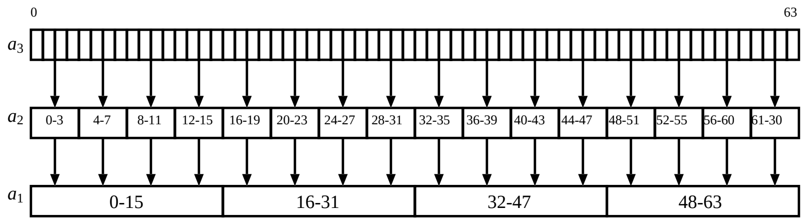

Fig. 7 The interval sums for the prefix-summing based on 4-ary tree

Next, we will show that the prefix-sums can be computed using the interval sums using Figure 8. First, the prefix sums of $w = 4$ sums in a_1 are computed. We can think that a_2 is partitioned into $w = 4$ groups of $w = 4$ sums each, which corresponds to a node of the 4-ary tree. The prefix-sums in a_1 are added to the first number of each group in a_2. Next, the prefix-sums of each group in a_2 are computed independently. Again, we can think that a_3 is partitioned into 16 groups of 4 numbers each. The prefix-sums in a_2 are added to the first number of each group in a_3 similarly. Finally, the prefix-sums within each group of a_3 are computed. In this way, the prefix-sums are computed in array a_3.

Let us evaluate the computing time. We use Prefix-summing Algorithm One-warp to compute the prefix-sums within each group. Similarly to Summing Algorithm Tree, we can prove that the prefix-sums of a_1 can be computed in $O(l \log w)$ time units. Also, we can say that the prefix-sums of a_{j+1} $(0 \leq j \leq m-1)$ within each group can be computed in $O((w^j + l) \log w)$ time units. Hence, the total computing time is $O(\frac{n}{w} \log w + l \log n)$ time units. Also, the barrier synchronization is necessary after computing the prefix-sums of each a_{j+1}. Further, it should be clear that the computing time for adding the prefix-sums to the first element of each group is dominated by the computation of the prefix-sums within each group. Hence, we can ignore this computing time. Thus, we have,

Lemma 7: The prefix-sums of *n* numbers can be computed in $O(\frac{n}{w} \log w + l \log n)$ time units with $O(\frac{\log n}{\log w})$ barrier synchronization steps using *n* threads on the UMM with width *w* and latency *l*.

Again, for later reference, we call the algorithm for

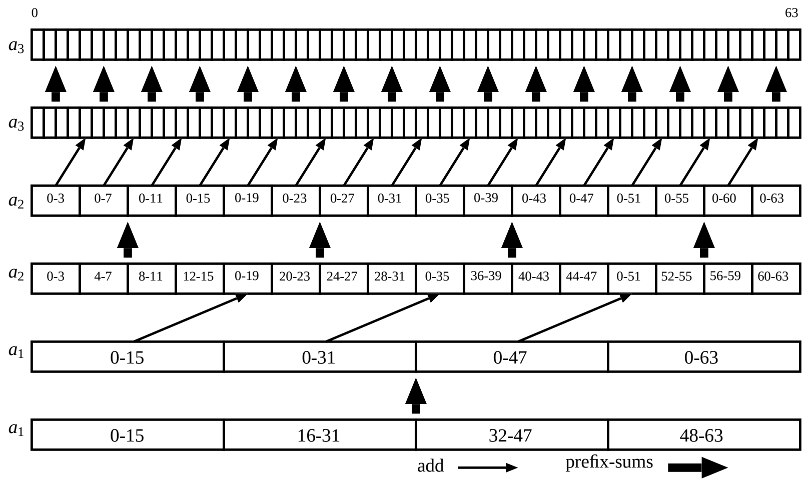

Fig. 8 Computation of the prefix-sums of the interval sums

Lemma 7 Prefix-summing Algorithm Tree.

From Prefix-summing Algorithm Tree, we can get a general idea for computing the prefix-sums. By hierarchical partitioning of the input, we can build a rooted tree data structure such that the leaves are the input numbers and each internal node corresponds to the sum of the children. This can be done by computing the sum of the children for each internal node toward the root. After that, the prefix sums of the children are computed for each internal nodes from the root to the leaves. Clearly, the computation of the prefix-sums involves that of the sum. Hence, it is sufficient to show how we compute the prefix-sums of children of each internal node for every level of the rooted tree. We use this idea to obtain Prefix-summing Algorithm Simple-Tree and Prefix-summing Algorithm Hybrid shown in the following subsection.

5.3 The prefix-summing algorithm using few barrier synchronization steps

We first show Prefix-summing Algorithm Simple-Tree analogously to Summing Algorithm Simple-Tree. We use a rooted tree of height 2. The root has $\frac{n}{\log w}$ children with $\log w$ leaves each.

It should have no difficulty to confirm that the prefix-sums can be computed by the following two computations:

(A) the prefix sums of $\log w$ leaves of every children, and
(B) the prefix sums of $\frac{n}{\log w}$ children.

We first show (A). Similarly to Summing Algorithm Simple-Tree, we execute the first $\log\log w$ steps of the first stage of Prefix-summing Algorithm Simple. Figure 9 illustrates the computation of the interval sums for $\log w = 4$ and $\log\log w = 2$. After that, we execute the last $\log\log w$ steps of the second stage of Prefix-summing Algorithm Simple. Recall that the second stage performs "copy" and "add" operations as illustrated in Figure 6. Note that "add" operations to the neighbor's children are omitted, because we need the prefix-sums within $\log w$ children. For example, the "add" operation from 0-3 in a_3 to 4-5 in a_4 is omitted in the figure. Similarly to Algorithm Simple-Tree, the two stages of $\log\log w$ steps each take $O(\frac{n}{w} + l\log n)$ time units and $O(\log\log w)$ barrier synchronization steps.

Next, we show (B). The prefix sums of the $\frac{n}{\log w}$ children of the root are computed by Prefix-summing Algorithm Tree, which takes $O(\frac{\frac{n}{\log w}}{w}\log w + l\log w) = O(\frac{n}{w} + l\log w)$ time units and $O(\frac{\log\frac{n}{\log w}}{\log w}) = O(\frac{\log n}{\log w})$ barrier synchronization steps. Thus, we have,

Theorem 8: Prefix-Summing Algorithm Simple-Tree computes the prefix-sums of n numbers in $O(\frac{n}{w} + l\log n)$ time units with $O(\frac{\log n}{\log w} + \log\log w)$ barrier synchronization steps using $\frac{n}{2}$ threads on the UMM with width w and latency l.

Next, we show Prefix-summing Algorithm Hybrid analogous to Summing Algorithm Hybrid. We use a rooted tree which has wl children with $\frac{n}{wl}$ leaves each. Similarly, it is sufficient to show the following two computations:

(A) the prefix sums of $\frac{n}{wl}$ leaves of every children, and
(B) the prefix sums of wl children.

We can consider that the input is a 2-dimensional array a with wl rows and $\frac{n}{wl}$ columns. Clearly, (A) corresponds to the row-wise prefix-sums of a. However, the straightforward row-wise prefix-sums algorithm using wl threads requires non-coalesced memory access. Thus, we transpose the 2-dimensional array a as illustrated in Figure 11. After the transpose, (A) corresponds to the column-wise prefix-sums which can be computed as follows:

```
[Column-wise prefix-sums]
for $t \leftarrow 1$ to $\frac{n}{wl} - 1$ do
  for $i \leftarrow 0$ to $wl - 1$ do in parallel
    $T$(i) performs $a[t][i] \leftarrow a[t][i] + a[t-1][i]$
```

For each t, wl threads access to the t-th and the $(t-1)$-th rows. Thus, these threads performs contiguous memory access for

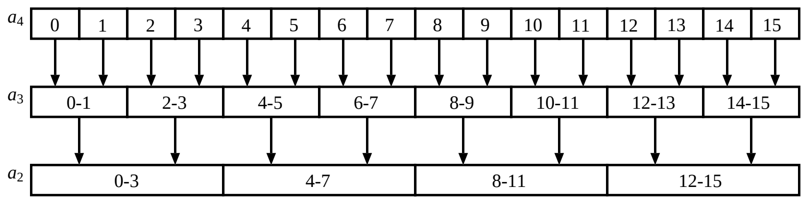

Fig. 9 The computation of the interval sums for Prefix-summing Algorithm Simple-Tree for $\log w = 4$

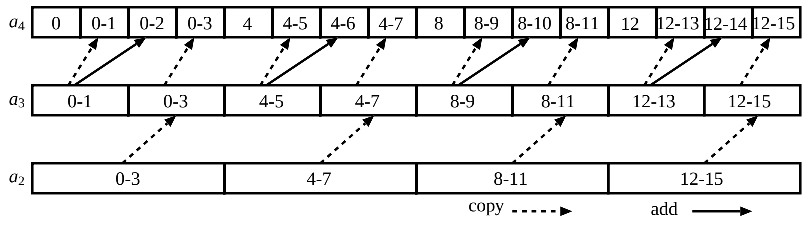

Fig. 10 The computation of the sums of the interval sums for Prefix-summing Algorithm Simple-Tree for $\log w = 4$

each t and the computing time is $O(\frac{wl}{w} + \frac{wll}{wl}) \cdot \frac{n}{wl} = O(\frac{n}{w})$ time units and 0 barrier synchronization step from Lemma 1. Also transpose can be done in $O(\frac{n}{w} + \frac{nl}{wl} + l) = O(\frac{n}{w} + l)$ time units using wl threads on the UMM using the technique for transposing a matrix shown in [20]. After transposing, 1 barrier synchronization step is necessary to confirm that transpose is completed before the computation of the column-wise prefix-sums. Thus, (A) can be done in $O(\frac{n}{w} + l)$ time units and 1 barrier synchronization step.

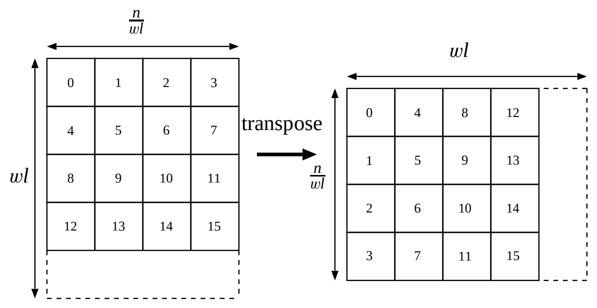

Fig. 11 Transpose of a of size $wl \times \frac{n}{wl}$

We use Prefix-summing Algorithm Simple-Tree for (B). From Theorem 8, (B) can be done in $O(\frac{wl}{wl} + l\log(wl)) = O(l\log n)$ time units and $O(\frac{\log l}{\log w} + \log\log w)$ barrier synchronization steps. Thus, we have,

Theorem 9: Prefix-summing Algorithm Hybrid computes the prefix-sums of n numbers in $O(\frac{n}{w} + l\log n)$ time units with $O(\frac{\log l}{\log w} + \log\log w)$ barrier synchronization steps using wl threads on the UMM with width w and latency l.

6. Conclusion

The main contribution of this paper is to introduce the asynchronous version of the UMM. We have also presented time-optimal parallel summing algorithm running in $O(\frac{n}{w} + l\log n)$ time units and $O(\frac{\log l}{\log w} + \log\log w)$ barrier synchronization steps. Further, we have shown that the prefix-sums can be computed in the same time units and the same barrier synchronization steps. It is an interesting open problem to further reduce the number of barrier synchronization steps of time-optimal parallel summing computation. Finding a good lower bound of the number of barrier synchronization steps is also a challenging open problem.

References

[1] Alfred V. Aho, Jeffrey D. Ullman, and John E. Hopcroft. *Data Structures and Algorithms*. Addison Wesley, 1983.
[2] Selim G. Akl. *Parallel Sorting Algorithms*. Academic Press, 1985.
[3] Selim. G. Akl and Kelly. A. Lyons. *Parallel Computational Geometry*. Prentice-Hall, 1993.
[4] K. E. Batcher. Sorting networks and their applications. In *Proc. AFIPS Spring Joint Comput. Conf.*, volume 32, pages 307–314, 1968.
[5] Michael J. Flynn. Some computer organizations and their effectiveness. *IEEE Transactions on Computers*, C-21:948–960, 1972.
[6] A. Gibbons and W. Rytter. *Efficient Parallel Algorithms*. Cambridge University Press, 1988.
[7] ALLAN Gottlieb, RALPH Grishman, CLYDE P. Kruskal, KEVIN P. McAuliffe, LARRY Rudolph, and MARC Snir. The nyu ultracomputer – designing an MIMD shared memory parallel computer. *IEEE Trans. on Computers*, C-32(2):175 – 189, Feb. 1983.
[8] Mark Harris, Shubhabrata Sengupta, and John D. Owens. Chapter 39. parallel prefix sum (scan) with CUDA. In *GPU Gems 3*. Addison-Wesley, 2007.
[9] Wen-mei W. Hwu. *GPU Computing Gems Emerald Edition*. Morgan Kaufmann, 2011.
[10] Yasuaki Ito, Kouhei Ogawa, and Koji Nakano. Fast ellipse detection algorithm using Hough transform on the GPU. In *Proc. of International Conference on Networking and Computing*, pages 313–319, Dec. 2011.
[11] Joseph JáJá. *An Introduction to Parallel Algorithms*. Addison-Wesley, 1992.

[12] Akihiko Kasagi, Koji Nakano, and Yasuaki Ito. An implementation of conflict-free off-line permutation on the GPU. In *Proc. of International Conference on Networking and Computing*, pages 226–232, 2012.

[13] Vipin Kumar, Ananth Grama, Anshul Gupta, and George Karyapis. *Introduction to Parallel Computing: Design and Analysis of Algorithms*. The Benjamin/Cumming Publishing, 1994.

[14] Duncan H. Lawrie. Access and alignment of data in an array processor. *IEEE Trans. on Computers*, C-24(12):1145– 1155, Dec. 1975.

[15] Frank Thomson Leighton. *Introduction to Parallel Algorithms and Architectures: Arrays, Trees, Hypercubes*. Morgan Kaufmann, 1991.

[16] Duhu Man, Kenji Uda, Yasuaki Ito, and Koji Nakano. A GPU implementation of computing euclidean distance map with efficient memory access. In *Proc. of International Conference on Networking and Computing*, pages 68–76. IEEE CS Press, Dec. 2011.

[17] Duhu Man, Kenji Uda, Hironobu Ueyama, Yasuaki Ito, and Koji Nakano. Implementations of a parallel algorithm for computing euclidean distance map in multicore processors and GPUs. *International Journal of Networking and Computing*, 1(2):260–276, July 2011.

[18] Koji Nakano. Efficient implementations of the approximate string matching on the memory machine models. In *Proc. of International Conference on Networking and Computing*, pages 233–239, Dec. 2012.

[19] Koji Nakano. 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)*, pages 99–113. Springer, Sept. 2012.

[20] Koji Nakano. Simple memory machine models for GPUs. In *Proc. of International Parallel and Distributed Processing Symposium Workshops*, pages 788–797. IEEE CS Press, May 2012.

[21] Kazufumi Nishida, Yasuaki Ito, and Koji Nakano. Accelerating the dynamic programming for the matrix chain product on the GPU. In *Proc. of International Conference on Networking and Computing*, pages 320–326, Dec. 2011.

[22] Kazufumi Nishida, Yasuaki Ito, and Koji Nakano. 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)*, pages 1–15. IEEE CS Press, Sept. 2012.

[23] NVIDIA Corporation. NVIDIA CUDA C best practice guide version 3.1, 2010.

[24] NVIDIA Corporation. NVIDIA CUDA C programming guide version 5.0, 2012.

[25] Kohei Ogawa, Yasuaki Ito, and Koji Nakano. Efficient canny edge detection using a gpu. In *Proc. of International Conference on Networking and Computing*, pages 279–280, Nov. 2010.

[26] Michael J. Quinn. *Parallel Computing: Theory and Practice*. McGraw-Hill, 1994.

[27] Akihiro Uchida, Yasuaki Ito, and Koji Nakano. Fast and accurate template matching using pixel rearrangement on the GPU. In *Proc. of International Conference on Networking and Computing*, pages 153–159. IEEE CS Press, Dec. 2011.

[28] Akihiro Uchida, Yasuaki Ito, and Koji Nakano. An efficient GPU implementation of ant colony optimization for the traveling salesman problem. In *Proc. of International Conference on Networking and Computing*, pages 94–102, Dec. 2012.

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.