2012 IEEE 26th International Parallel and Distributed Processing Symposium Workshops & PhD Forum

Simple Memory Machine Models for GPUs

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

***Abstract*—The main contribution of this paper is to introduce two parallel memory machines, the Discrete Memory Machine (DMM) and the Unified Memory Machine (UMM). Unlike well studied theoretical parallel computational models such as PRAMs, these parallel memory machines are practical and capture the essential feature of memory access of NVIDIA GPUs. As a first step of the development of algorithmic techniques on the DMM and the UMM, we first evaluated the computing time for the contiguous access and the stride access to the memory on these models. We then go on to present parallel algorithms to transpose a two dimensional array on these models. Finally, we show that, for any given permutation, data in an array can be moved along a given permutation both on the DMM and on the UMM. Since the computing time of our permutation algorithms on the DMM and the UMM is equal to the sum of the lower bounds obtained from the memory bandwidth limitation and the latency overhead, they are optimal from the theoretical point of view.**

***Keywords*-memory banks, parallel computing models, parallel algorithms, stride memory access, matrix transpose, array permutation, GPU, CUDA**

I. Introduction

A. Background

The research of parallel algorithms has a long history of more than 40 years. Sequential algorithms have been developed mostly on the RAM (Random Access Machine) [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 PRAM (Parallel Random Access Machine) [2], [3], [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 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 [5], [6], [7], [8]. Latest GPUs are designed for general purpose computing and can perform computation in applications traditionally handled by the CPU. Hence, GPUs have recently attracted the attention of many application developers [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.

CUDA uses two types of memories in the NVIDIA GPUs: *the global memory* and *the shared memory* [10]. The global memory is implemented as a off-chip DRAM, and has large capacity, say, 1.5-4 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 [6], [11], [12]. To maximize the bandwidth between the GPU and the DRAM chips, the consecutive addresses of the global memory must be accessed in the same time. Thus, threads should perform coalesced access when they access to the global memory.

The address space of the shared memory is mapped into several physical memory banks. If two or more processor cores access to the same memory banks in the same time, the access requests are processed sequentially. Hence to maximize the memory access performance, processor cores should access to distinct memory banks to avoid the bank conflicts of the memory access.

B. Our Contribution: Introduction to the Discrete Memory Machine and the Unified Memory Machine

The first contribution of this paper is to introduce simple parallel memory bank machine models that capture the essential features of the coalescing of the global memory access and the bank conflict of the shared memory access. More specifically, we present 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.

978-0-7695-4676-6/12 $26.00 © 2012 IEEE
DOI 10.1109/IPDPSW.2012.98

The architectures of the DMM and the UMM are illustrated in Figure 1. In both architectures, the processing elements (PEs) are connected to the memory banks (MBs) through the memory management unit (MMU). A single address space of the memory is mapped to the MBs in an interleaved way such that the data of address i is stored in the $(i \bmod w)$-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.

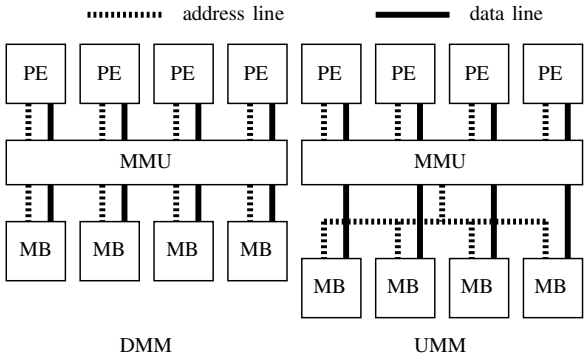


Figure 1. The architectures of the DMM and the UMM

The performance of algorithms of the PRAM is usually evaluated using two parameters: the size n of the problem and the number p of processors. For example, it is well known that the sum of n numbers can be computed in $O(\frac{n}{p}+\log p)$ time on the PRAM [2]. We will use additional two parameters, 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 processors), w (the width of a memory), and l (the latency of a memory). In typical NVIDIA GPUs, the width w of global and shared memory is 16 or 32. Also the latency l of the global memory is 400-800 clock cycles.

We also introduce *the bandwidth limited PRAM* (BPRAM). In the BPRAM of width w, any w processors out of the p processors can access to the memory in a time unit. Clearly, the BPRAM is less powerful than the PRAM and is more powerful than DMM and the UMM. Unlike the DMM and the UMM, the BPRAM has no restriction of the addresses of memory access and 1 memory access latency. We use the BPRAM to show the goodness of the performance of algorithms on the DMM and the UMM. If the computing time of an algorithm to solve some problem on the DMM or the UMM is almost the same as that on the BPRAM, we can say that the algorithm on the DMM or the UMM is close to optimal.

Please note that the DMM and the UMM are theoretical models of parallel computation, that capture the essential feature of the global memory and the shared memory of NVIDIA GPUs. NVIDIA GPUs have other features such as hierarchical architecture grid/block/thread, and the cache of the global memory. However, if these aspects are incorporated in our theoretical parallel models, they will be complicated and need more parameters. The development of algorithms on such complicated model may have too much non-essential and tedious optimizations. Thus, we have introduces two simple parallel models, the DMM and the UMM, which focuses on the memory accesses of the global memory and the shared memory of NVIDIA GPUs.

In [13], the authors have presented a GPU memory model and presented a cache-efficient FFT. However, their model focuses on the cache mechanism and ignores the coalescing and the bank conflict. Also, in [14], acceleration techniques for GPU have been discussed. Although they are taking care of the limited bandwidth of the global memory, the details of the memory architecture are not considered. As far as we know, this paper is the first work that introduces simple theoretical parallel computing models for GPUs. We believe that the development of algorithms on these models are useful to investigate algorithmic techniques for the GPUs.

C. Our Contribution: Fundamental Algorithms on the DMM and the UMM

The second contribution of this paper is to evaluate the performance of two memory access methods, *the contiguous access* and *the stride access* on the DMM and the UMM. The reader should refer to Figure 2 for illustrating these two access methods. We will show that the contiguous access of an array of size n can be done in $O(\frac{n}{w}+\frac{nl}{p})$ time units on the DMM and the UMM. Also, the contiguous access and the stride access can be done in $O(\frac{n}{w})$ time units on the BPRAM. Thus, the contiguous access of the DMM and the UMM is optimal when $wl \leq p$, because $\frac{n}{w} \geq \frac{nl}{p}$ if this is the case. Further, we will show that the stride access of the DMM can be done in $O(\frac{n}{w} \cdot \mathrm{GCD}(\frac{n}{p}, w) + \frac{nl}{p})$ time units on the DMM, where $\mathrm{GCD}(\frac{n}{p}, w)$ is the greatest common divisor of $\frac{n}{p}$ and w. Hence, the stride access of the DMM is optimal if $\frac{n}{p}$ and w are co-prime and $wl \leq p$. The stride access of the UMM can be done in $O(\min(n, \frac{n}{w} \cdot \frac{n}{p} + \frac{nl}{p}))$ time units. Hence, the stride access of the UMM needs an overhead of a factor of $\frac{n}{p}$.

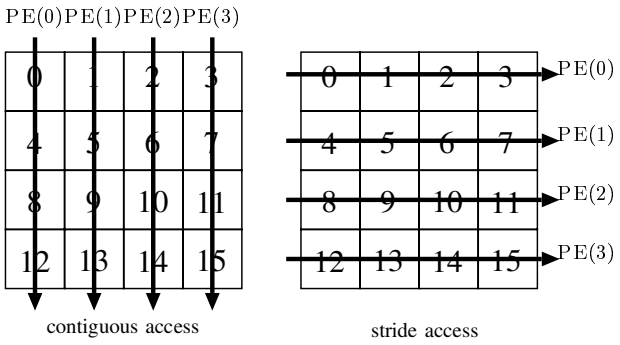


Figure 2. The contiguous access and the stride access for $p = 4$ and $n = 16$.

The third contribution of this paper is to show algorithms for transposing a two dimensional array and for permutation of an array on the DMM and the UMM. We first show that the DMM and the UMM can transpose a two dimensional array of size $\sqrt{n} \times \sqrt{n}$ in $O(\frac{n}{w} + \frac{nl}{p})$ time units. We generalize this result to permute data in an array of the memory. Suppose that a permutation of an array of size n is given. The goal is to move data stored in an array along the given permutation. Quite surprisingly, for any given permutation of an array of size n, data can be moved in $O(\frac{n}{w} + \frac{nl}{p})$ time units both on the DMM and on the UMM.

From the results of the second and the third contributions, we have one important observation as follows. The factor $\frac{n}{w}$ in the computing time comes from *the bandwidth limitation* of the memory. It takes at least $\frac{n}{w}$ time units to access whole data in an array of size n from the memory bandwidth w. Also, the factor $\frac{nl}{p}$ comes from *the latency overhead*. From the memory access latency l, each processor cannot send next access request in l time units. It follows that, each processor can access to the memory once in l time units and each of the l time units can have expected $\frac{p}{l}$ access requests by processors. Hence, $\frac{nl}{p}$ time units are necessary to access all of the elements in an array of size n. Further, to hide the latency overhead factor $\frac{nl}{p}$ from the bandwidth limitation factor $\frac{n}{w}$, the number p of the processors must be no less than wl. We can confirm this fact from a different aspect. We can think that the memory access request are stored in a pipeline buffer of size l for each memory bank. Since we have w memory banks, we have wl pipeline registers to store memory access requests at all. Since at most one memory request per processor are stored in the wl pipeline registers, $wl \leq p$ must be satisfied to fill the pipeline registers full of memory access requests.

This paper is organized as follows. We first define the DMM and the UMM in Section II. In Section III, we evaluate the performance of the DMM and the UMM for the contiguous access and the stride access to the memory. Section IV presents algorithms that perform the transpose of two dimensional array on the DMM and the UMM. In Section V, we show that any permutation on an array can be done efficiently on the DMM. Finally, Section VI presents an permutation algorithm on the UMM.

II. Parallel Memory Machines: DMM and UMM

Let us start with defining *PRAM* (Parallel Random Access Machine), the most popular shared memory parallel machine model. The PRAM consists of p processors and a shared memory. The shared memory is an array of memory cells, each of which can store a word of data. Each of the processors can select a memory cell in the array independently, and can perform read/write operation in a time unit. Please see [2] for the details of the PRAM.

We introduce a memory bandwidth limited PRAM. We assume that w memory cells can be read/written in a time unit. If more than w memory cells are accessed, the w operations are automatically serialized. More specifically, if p memory cells are accessed, w read/write operations performed in each time unit. and it takes $\lceil \frac{p}{w} \rceil$ time to complete the p read/write operations. We call such PRAM *the Bandwidth-limited PRAM (BPRAM).*

We next introduce *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*. 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 memory management unit. Thus, it takes $k + l - 1$ time units to complete k continuous access requests to a particular bank.

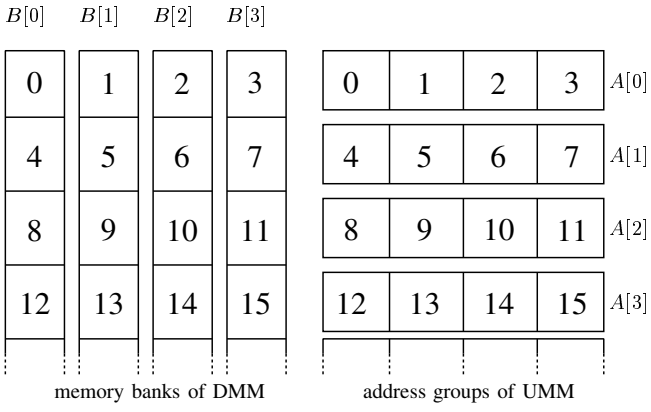


Figure 3. Banks and address groups for $w = 4$

We assume that p processors are partitioned into $\frac{p}{w}$ groups of w processors called *warps*. More specifically,p processors

are partitioned into $\frac{p}{w}$ warps $W(0), W(1), \ldots W(\frac{p}{w}-1)$ such that $W(i) = \{\mathrm{PE}(i \cdot w), \mathrm{PE}(i \cdot w + 1), \ldots, \mathrm{PE}((i+1) \cdot w - 1)\}$ $(0 \leq i \leq \frac{p}{w} - 1)$. Warps are activated for memory access in turn, and w processors in a warp try to access the memory in the same time. In other words, $W(0), W(1), \ldots, W(w-1)$ are activated in a round-robin manner if at least one processor in a warp requests memory access. If no processor in a warp needs memory access, such warp is not activated and is skipped. When $W(i)$ is activated, w processor in $W(i)$ sends memory access requests, one request per processor, to the memory. We also assume that a processor cannot send a new memory access request until the previous memory access request is completed. Hence, if a processor send a memory access request, it must wait for l time units to send a next memory access request.

Let us evaluate the time for memory access using Figure 4 on the DMM for $p = 8$, $w = 4$, and $l = 3$. Suppose that processors in $W(0)$ try to access $m[0], m[1], m[5]$, and $m[10]$, and those in $W(1)$ try to access $m[8], m[9], m[14]$, and $m[15]$. First, memory access requests to $m[0]$, $m[1]$, and $m[10]$ are sent to the banks $B[0], B[1], B[2]$ first, and then access requests to $m[5]$ are sent to the bank $B[1]$. After that, memory access requests to $m[8], m[9], m[14], m[15]$. are sent to the bank $B[0], B[1], B[2], B[3]$. Since we have latency $l = 3$, all of these memory requests are completed in $3 + l - 1 = 5$ time units on the DMM.

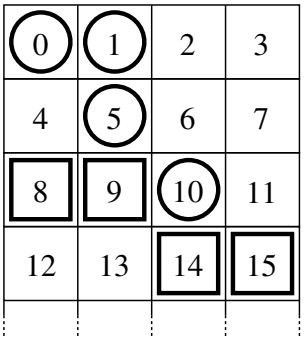


Figure 4. An example of memory access

We next define *the Unified Memory Bank Machine* (*UMM* for short) 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 processors are partitioned into warps and each warp access to the memory in turn.

Again, let us evaluate the time for memory access using Figure 4 on the UMM for $p = 8$, $w = 4$, and $l = 3$. To complete the memory access requests by $W(0)$, those to $A[0]$ is performed for $m[0]$ and $m[1]$. After that, access requests to $A[1]$ is performed for $m[5]$ and then that to $A[2]$ is performed for $m[10]$. Next, memory access requests by $W(1)$ are processed. First, memory access to $A[2]$ is requested for $m[8]$ and $m[9]$, and then that to $A[3]$ is requested for $m[14]$ and $m[15]$. Since we have latency $l = 3$, all of these memory request are processed in $5 + l - 1 = 7$ time units on the UMM.

III. Sequential memory access operations

We begin with simple operations to evaluate the potentiality of the DMM and the UMM. Let p and w be the number of processors and the width of the machines. We assume that an array m of size n is arranged in the memory. Let $m[i]$ $(0 \leq i \leq n-1)$ denote the i-th word of the memory. We assume that $n \gg p > w$. We consider two access operations to the memory such that each of the p processors reads $s = \frac{n}{p}$ memory cells out of the n memory cells. Again, the readers should refer to Figure 2. In the *contiguous access*, the first p memory cells are accessed by the p processors. Next, the second p memory cells are accessed. In this way, all n memory cells are accessed in turn. In the *stride access*, the memory is partitioned into p groups of $\frac{n}{p}$ consecutive memory cells each. A processor is assigned to each group and the assigned processor accesses memory cells in the group sequentially. The contiguous access and the stride access are written as follows.

[Contiguous Access]
for $t \leftarrow 0$ to $s-1$
  for $i \leftarrow 0$ to $p-1$ do in parallel
    PE(i) accesses to $m[i + t \cdot p]$

[Stride Access]
for $t \leftarrow 0$ to $s-1$
  for $i \leftarrow 0$ to $p-1$ do in parallel
    PE(i) accesses to $m[i \cdot s + t]$

In the contiguous access, w processors in each warp access memory cells in different memory banks. Hence, the memory access by a warp takes l time unit. Also, the memory access by a warp is processed in every 1 time unit. Since we have $\frac{p}{w}$ warps, p memory cells can be accessed by p processors in $\frac{p}{w} + l - 1$ time units. Consequently, the contiguous access takes $(\frac{p}{w} + l - 1) \cdot s = O(\frac{n}{w} + \frac{nl}{p})$ time units on the DMM. In the contiguous access on the UMM, each warp access to the memory cells in the same address group. Thus, the memory access by a warp takes l time unit and the whole contiguous access runs in $O(\frac{n}{w} + \frac{nl}{p})$.

The performance analysis of the stride access on the DMM is a bit complicated. Let us start with a simple case: $s = w$. In this case, the p processors access to p memory cells $m[t], m[w+t], m[2w+t], \ldots, m[(p-1)w+t]$ for each t $(0 \leq t \leq w-1)$. Unfortunately, these memory cells are

in the same memory bank $B[t]$. Hence, memory access by a warp takes $w + l - 1$ time units and the memory access to these p memory cells takes $w \cdot \frac{p}{w} + l - 1 = p + l - 1$ time units. Thus, the stride access when $s = w$ takes at least $(p + l - 1) \cdot \frac{n}{p} = O(n + \frac{nl}{p})$ time units.

Next, let us consider general case. The w processors in the first warp access to $m[t], m[s + t], m[2s + t], \ldots, m[(w - 1)s + t]$. for each t ($0 \leq t \leq w - 1$). These w memory cells are allocated in the banks $B[t \bmod w], B[(s + t) \bmod w], B[(2s + t) \bmod w], \ldots, B[((w - 1)s + t) \bmod w]$. Let $L = \mathrm{LCM}(s, w)$ and $G = \mathrm{GCD}(\mathrm{s}, \mathrm{w})$ be the Least Common Multiple and the Greatest Common Divisor of s and w, respectively. From the basic number theory, it should be clear that $t \bmod w = (\frac{L}{s} \cdot s + t) \bmod w$, and the values of $t \bmod w$, $(s + t) \bmod w$, ..., $((\frac{L}{s} - 1) \cdot s + t) \bmod w$ are distinct. Thus, the w memory cells are in the $\frac{L}{s} = \frac{w}{G}$ banks $B[t \bmod w], B[(s + t) \bmod w], B[(2s + t) \bmod w], \ldots, B[((\frac{w}{G} - 1)s + t) \bmod w]$ equally, and each bank has G memory cells of the w memory cells. Hence, the w processor in a warp takes $G + l - 1$ time units for each t, and the p processors takes $G \cdot \frac{p}{w} + l - 1$ time units for each t. Therefore, the DMM takes $(G \cdot \frac{p}{w} + l - 1) \cdot s = O(\frac{nG}{w} + \frac{nl}{p})$ time to complete the stride access. If $s = w$ then $G = w$ and the time for the stride access is $O(n + \frac{nl}{p})$. If s and w are co-prime, $G = 1$ and the stride access takes $O(\frac{n}{w} + \frac{nl}{p})$ time units.

Finally, we will evaluate the computing time of the stride access on the UMM. If $s \geq w$ (i.e. $n \geq pw$), then the w memory cells are accessed by w processors in a warp are in the different address group. Thus, w processors access to w memory cells in $w + l - 1$ time units, and the stride access takes $(w \cdot \frac{p}{w} + l - 1) \cdot \frac{n}{p} = O(n + \frac{nl}{p}))$ time. When $s < w$ (i.e. $n < pw$), the w memory cells accessed by w processors in a warp are in at most $\lceil \frac{(w-1)s+1}{w} \rceil \leq s$ address groups. Hence, the stride access by p processors for each t takes at most $s \cdot \frac{p}{w} + l - 1$ time, and thus, the whole stride access takes $(\frac{sp}{w} + l - 1) \cdot \frac{n}{p} = O(\frac{ns}{w} + \frac{nl}{p})$ time. Consequently, the stride access can be completed in $O(\min(n, \frac{ns}{w}) + \frac{nl}{p}))$ for all $s = \frac{n}{p}$. Finally, we have,

Theorem 1: The contiguous access and the stride access on the PRAM, the BPRAM, the DMM, and the UMM can be done in time units shown in Table I.

IV. Transpose of a two dimensional array

Suppose that two dimensional array a of size $\sqrt{n} \times \sqrt{n}$ is arranged in the memory. We assume that $a[i][j]$ ($0 \leq i, j \leq \sqrt{n} - 1$) is located in the $(i \cdot \sqrt{n} + j)$th memory cell of the memory. The transpose of a two dimensional array is a task to move a word of data stored in $a[i][j]$ to $a[j][i]$ for all ($0 \leq i, j \leq \sqrt{n} - 1$).

Let us start with a straightforward transpose algorithm using the contiguous access and the stride access. The following algorithm transposes a two dimensional array a size $\sqrt{n} \times \sqrt{n}$.

[Straightforward transpose algorithm]
for $t \leftarrow 0$ to $\frac{n}{p} - 1$
  for $i \leftarrow 0$ to $p - 1$ do in parallel
    $j \leftarrow (t \cdot p + i)/\sqrt{n}$
    $k \leftarrow (t \cdot p + i) \bmod \sqrt{n}$
    PE(i) performs exchange $a[j][k] \leftrightarrow a[k][j]$

For each t, the processors performs the contiguous access and the stride access, which takes $O(1)$ time on the PRAM and $O(\frac{p}{w})$ time on the BPRAM. Hence, the PRAM and the BPRAM can transpose a two dimensional array in $O(\frac{n}{p})$ time units and $O(\frac{n}{w})$ time units, respectively.

Since the straightforward algorithm contains the stride access, it is not difficult to see that the DMM and the UMM takes $O(\frac{n}{w} \cdot \mathrm{GCD}(\sqrt{n}, w) + \frac{nl}{p})$ time units and $O(\min(n, \frac{n}{w} \cdot \frac{n}{p}) + \frac{nl}{p})$ time units for transposing a two dimensional array, respectively. On the DMM, $\mathrm{GCD}(\sqrt{n}, w) = w$ if $\sqrt{n}$ is divisible by w. If this is the case, the transpose takes $O(n)$ time units the DMM. We will show that, regardless of the value of n, the transpose can be done in $O(\frac{n}{w} + \frac{nl}{p})$ time both on the DMM and on the UMM.

We first show an efficient transposing algorithm on the DMM. The key idea is to access the array in diagonal fashion. We use a two dimensional array b of size $n \times n$ as a work space.

[Transpose by the diagonal access on the DMM]
for $t \leftarrow 0$ to $\frac{n}{p} - 1$
  for $i \leftarrow 0$ to $p - 1$ do in parallel
    $j \leftarrow (t \cdot p + i)/\sqrt{n}$
    $k \leftarrow (t \cdot p + i) \bmod \sqrt{n}$
    PE(i) performs $b[j][k] \leftarrow a[j][k]$
for $t \leftarrow 0$ to $\frac{n}{p} - 1$
  for $i \leftarrow 0$ to $p - 1$ do in parallel
    $j \leftarrow (t \cdot p + i)/\sqrt{n}$
    $k \leftarrow (t \cdot p + i) \bmod \sqrt{n}$
    PE(i) performs
      $a[(j + k) \bmod \sqrt{n}][k] \leftarrow b[k][(j + k) \bmod \sqrt{n}]$

The readers should refer to Figure 5 for illustrating the indexes of processors reading from memory cells in b and writing from memory cells in a for $n = p = 16$ and $w = 4$. From the figure, we can confirm that processors $\mathrm{PE}(j{\cdot}4{+}0), \mathrm{PE}(j{\cdot}4{+}1), \mathrm{PE}(j{\cdot}4{+}2), \mathrm{PE}(j{\cdot}4{+}3)$ read from memory cells in diagonal location of b and write to memory cells in diagonal location of a for every j ($0 \leq j \leq 3$). Thus, reading and writing memory banks by w processors in a warp are different. Hence p processors can copy p memory cells in $\frac{p}{w} + l$ time units and thus the total computing time is $(\frac{p}{w} + l) \cdot \frac{n}{p} = O(\frac{n}{w} + \frac{nl}{p})$ time. Therefore, we have,

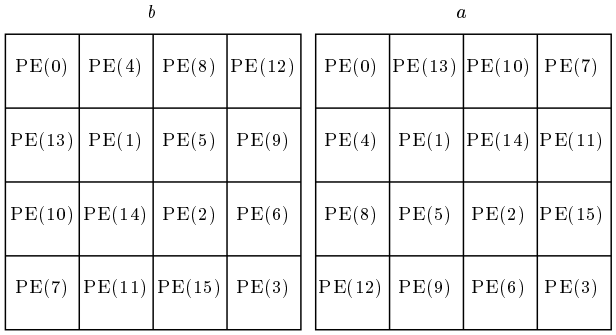
Lemma 2: The transpose of a two dimensional array of size $\sqrt{n} \times \sqrt{n}$ can be done in $O(\frac{n}{w} + \frac{nl}{p})$ time using p processors on the DMM with memory width w and latency l.

Next, we will show that the transpose of a two dimen-

Table I
THE RUNNING TIME FOR THE CONTIGUOUS ACCESS AND THE STRIDE ACCESS

Operation	PRAM	BPRAM	DMM	UMM
Contiguous Access	$O(\frac{n}{p})$	$O(\frac{n}{w})$	$O(\frac{n}{w}+\frac{nl}{p})$	$O(\frac{n}{w}+\frac{nl}{p})$
Stride Access	$O(\frac{n}{p})$	$O(\frac{n}{w})$	$O(\frac{n}{w}\cdot G+\frac{nl}{p})$	$O(\min(n,\frac{n}{w}\cdot\frac{n}{p})+\frac{nl}{p})$

n =#data, p =#processors, w =memory bandwidth, $G=\mathrm{GCD}(\frac{n}{p},w)$



The indeices of processors reading from memory cells of b

The indeices of processors writing in memory cellsof a

Figure 5. Transposing on the DMM

sional array can be also done in $O(\frac{n}{w}+\frac{nl}{p})$ on the UMM if every processor has a local memory that can store w words. As a preliminary step, we will show that the UMM can transpose a two dimensional array of size $w\times w$ in $O(w+l)$ time units using w processors with each processor having a local storage of size w. We assume that each processor has local memory that can store w words. Let l_i denote the local memory of $\mathrm{PE}(i)$, and $l_i[0], l_i[1], \ldots l_i[w-1]$ can store a word of data.

[Transpose by the rotating technique on the UMM]
for $t \leftarrow 0$ to $w-1$
  for $i \leftarrow 0$ to $w-1$ do in parallel
    PE(i) performs $l_i[t] \leftarrow a[t][(t+i) \bmod w]$
for $t \leftarrow 0$ to $w-1$
  for $i \leftarrow 0$ to $w-1$ do in parallel
    PE(i) performs $a[t][(t-i) \bmod w] \leftarrow l_i[(t-i) \bmod w]$

Let (i,j) denote the value stored in $a[i][j]$ initially. The readers should refer to Figure 5 for illustrating how these values are transposed.

Let us confirm that the algorithm above correctly transpose two dimensional array a. In other words, we will show that, when the algorithm terminates, $a[i][j]$ stores (j,i). It should be clear that, the value stored in $l_i[t]$ is $(t,(t+i) \bmod w)$. Since $((t-i) \bmod w, t)$ is stored in $l_i[(t-i) \bmod w]$, it is also stored in $a[t][(t-i) \bmod w]$ when the algorithm terminates. Thus, every $a[i][j]$ $(0 \le i,j \le w-1)$ stores (j,i). This completes the proof of the correctness of our transpose algorithm on the UMM.

Let us evaluate the computing time. In the reading operation $l_i[t] \leftarrow a[t][(t+i) \bmod w]$, w memory cells $a[t][(t+0 \bmod w)], a[t][(t+1 \bmod w)], \ldots, a[t][(t+w-1 \bmod w)]$ are in the different memory banks. Also, in the writing operation $a[t][(t-i) \bmod w] \leftarrow l_i[(t-i) \bmod w]$, w memory cells $a[t][(t-0 \bmod w)], a[t][(t-1 \bmod w)], \ldots, a[t][(t-(w-1) \bmod w)]$ are in the different memory banks. Thus, each reading and writing operation can be done in l time units and this algorithm runs in $O(w+l)$ time units.

The transpose of a larger two dimensional array of size $\sqrt{n}\times\sqrt{n}$ can be done by repeating the transpose of two dimensional array of size $w\times w$. More specifically, the two dimensional array is partitioned into $\frac{\sqrt{n}}{w}\times\frac{\sqrt{n}}{w}$ subarrays of size $w\times w$. Let $A[i][j]$ $(0 \le i,j \le \frac{n}{w}-1)$ denote the subarray of size $w\times w$. First, each subarray $A[i][j]$ are transposed independently using w processors. After that, the corresponding words of $A[i][j]$ and $A[j][i]$ are swapped for all i and j in an obvious way.

Let us evaluate the computing time to complete the transpose of a $\sqrt{n}\times\sqrt{n}$ two dimensional array. Suppose that we have p $(\le \frac{n}{w})$ processors and partition the p processors into $\frac{p}{w}$ groups with w processors each. We assign $\frac{n}{w^2}/\frac{p}{w} = \frac{n}{pw}$ subarrays to each group of w processors. The p processors can transpose $\frac{p}{w}$ subarrays in parallel in $O(w\cdot(\frac{p}{w}+l)) = O(p+wl)$ time units. This transpose of $\frac{p}{w}$ subarrays are repeated $\frac{n}{pw}$ times, the total computing time for the subarray transpose is $\frac{n}{pw}\cdot O(p+wl) = O(\frac{n}{w}+\frac{nl}{p})$ time units. Clearly, the swap operation of subarrays can be also done in $O(\frac{n}{w}+\frac{nl}{p})$ time units. Thus we have,

Lemma 3: The transpose of a two dimensional array of size $\sqrt{n}\times\sqrt{n}$ can be done in $O(\frac{n}{w}+\frac{nl}{p})$ time on the UMM with each processor having local memory of w words.

V. PERMUTATION OF AN ARRAY ON THE DMM

In Section IV, we have shown that the transpose a two dimensional array on the DMM and the UMM can be done in $O(\frac{n}{w}+\frac{nl}{p})$ time units. The main purpose of this section is to show algorithms that perform any permutation of an array. Since a transpose is one of the permutations, the results of this section is a generalization of those presented in Section IV.

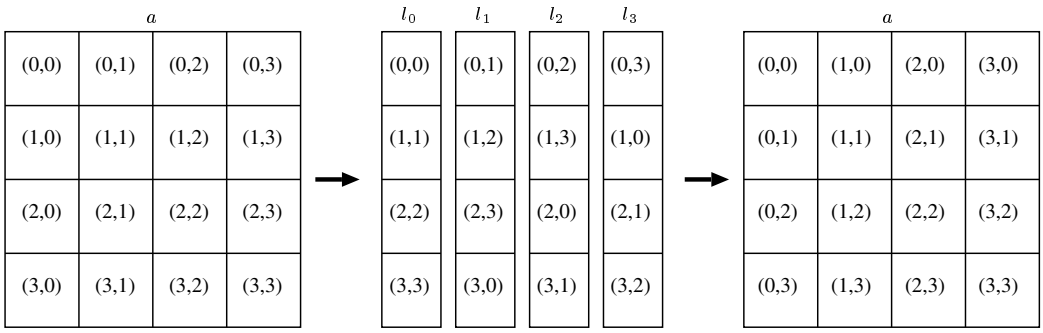


Figure 6. Transposing on the UMM

Let a be a one dimensional array of size n, and P be a permutation of $(0, 1, \ldots, n-1)$. The goal of permutation of an array is to move a word of data stored in $a[i]$ to $a[P(i)]$ for every i $(0 \leq i \leq n-1)$. Note that, permutation is given in offline. We will show that, for given any permutation P, permutation of an array can be done efficiently on the DMM and the UMM.

Let us start with a permutation algorithm on the PRAM and the BPRAM. Suppose we need to do permutation of an array a of size n and permutation P is given. We use an array b of size n as a work space.

[Straightforward permutation algorithm]
for $t \leftarrow 0$ to $\frac{n}{p} - 1$ do
  for $j \leftarrow 0$ to $p-1$ do in parallel
    $i \leftarrow t \cdot p + j$
    $\mathrm{PE}(j)$ performs $b[i] \leftarrow a[i]$
for $t \leftarrow 0$ to $\frac{n}{p} - 1$ do
  for $j \leftarrow 0$ to $p-1$ do in parallel
    $i \leftarrow t \cdot p + j$
    $\mathrm{PE}(j)$ performs $a[P(i)] \leftarrow b[i]$

It should be clear that the above algorithm rums in $O(\frac{n}{p})$ time on the PRAM and in $O(\frac{n}{w})$ time on the BPRAM.

This straightforward permutation algorithm also works correctly on the DMM and the UMM. However, it takes a lot of time to complete the permutation. In the worst case, this straightforward algorithm takes $O(n)$ time on the DMM and the UMM if all writing operation to $a[P(j)]$ are in the same bank on the DMM or in the different address groups on the UMM. We will show that any permutation of an array of size n can be done in $O(\frac{n}{w} + \frac{nl}{p})$ time units on the DMM and the UMM.

If we can schedule reading/writing operations for permutation such that w processors in a warp read from distinct banks and write in distinct banks on the DMM, the permutation can be done efficiently. For such scheduling, we uses an important graph theoretic result [15], [16] as follows:

Theorem 4 (König): A regular bipartite graph with degree ρ is ρ-edge-colorable.

Figure 7 illustrates an example of a regular bipartite graph with degree 4 painted by 4 colors. Each edge is painted by 4 colors such that no node is connected to edges with the same color. The readers should refer to [15], [16] for the proof of Theorem 4.

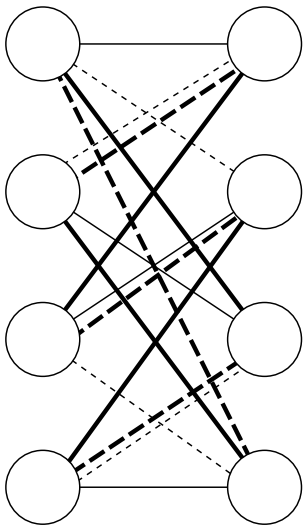


Figure 7. A regular bipartite graph with degree 4

Suppose that a permutation P of $(0, 1, \ldots, n-1)$ on DMM with width w is given. We draw a bipartite graph $G = (U, V, E)$ of P as follows:

- $U = \{0, 1, 2, \ldots, w-1\}$ is a set of nodes,
- $V = \{0, 1, 2, \ldots, w-1\}$ is a set of nodes, and
- for each pair source $a[i]$ and destination $a[P(i)]$, E has an edge connecting $i \bmod w (\in U)$ and $P(i) \bmod w (\in V)$.

From Theorem 4, $G = (U, V, E)$ is $\frac{n}{w}$-colorable. Let $c_0, c_1, \ldots, c_{\frac{n}{w}-1}$ be a set w edges painted by the same color.

Let $p_{i,j}$ $(0 \leq i \leq \frac{n}{w}-1, 0 \leq j \leq w-1)$ denote a node in V such that $(j, p_{i,j})$ in c_i. In other words, each c_i has w edges $(0, p_{i,0}), (1, p_{i,1}), \ldots, (w-1, p_{i,w-1})$, and no two edges of them share a node. Recall that each edge corresponds to a source and its destination. Let $s_{i,j}$ $(0 \leq i \leq \frac{n}{w}-1, 0 \leq j \leq w-1)$ denote a source corresponds to $(j, p_{i,j})$. It should have no difficulty to confirm that, for each i, w values $s_{i,0} \bmod w, s_{i,1} \bmod w, \ldots, s_{i,w-1} \bmod w$ are distinct, because the corresponding w edges do not share a node. Similarly, for each i, w values $P(s_{i,0}) \bmod w, P(s_{i,1}) \bmod w, \ldots, P(s_{i,w-1}) \bmod w$ are distinct. Thus, we have an important lemma as follows:

Lemma 5: Let $s_{i,j}$ be a source defined above. For each i, we have, (1) $a[s_{i,0}], a[s_{i,1}], \ldots a[s_{i,w-1}]$ are in different modules, and (2) $a[P(s_{i,0})], a[P(s_{i,1})], \ldots a[P(s_{i,w-1})]$ are in different modules.

We can perform the bank conflict-free permutation using $s_{i,j}$. We use an array b of size n as a work space. The details are spelled out as follows.

[Permutation algorithm on the DMM]

```
for t ← 0 to n/p − 1 do
  for j ← 0 to p − 1 do in parallel
    i ← t · p + j
    PE(j) performs b[i] ← a[i]
for t ← 0 to n/p − 1 do
  for j ← 0 to p − 1 do in parallel
    i ← t · p + j
    k ← s_{i/w, i mod w}
    PE(j) performs a[P(k)] ← b[k]
```

The first for-loop copies all words in a to b. The second for-loop copies all words in b to a based on P. Thus, a word of data stored in $a[k]$ $(0 \leq k \leq n-1)$ is moved to $a[P(k)]$, and permutation is performed correctly.

We will show that this permutation algorithm terminates in $O(\frac{n}{w} + \frac{nl}{p})$ time units. It should be clear that, in the first for-loop, w words in a read by w processors in a warp are different modules. Similarly, w words in b written by a warp are different modules. Thus, the operation $b[j] \leftarrow a[j]$ by a warp is bank conflict-free and can be done in $O(l)$ time and p processors perform it in $O(\frac{p}{w} + l)$ time. Thus, the first for-loop can be done in $\frac{n}{p} \cdot O(\frac{p}{w} + l) = O(\frac{n}{w} + \frac{nl}{p})$ time. From Lemma 5, the second for-loop is also bank conflict-free. Thus we have,

Theorem 6: Any permutation on an array of size n can be done in $O(\frac{n}{w} + \frac{nl}{p})$ time units on the DMM.

VI. Permutation of an array on the UMM

The main purpose of this section is to show a permutation algorithm on the UMM. Our permutation algorithm uses the transpose algorithm on the UMM presented in Section IV.

We start with a small array. Suppose that we have an array a of size w and permutation P on it. Since all elements in a are in the same address group, they can be read/written in a time unit. Thus, any permutation of an array a of size w can be done in $O(l)$ time.

Next, we show a permutation of an array a of size w^2. We can consider that a permutation is defined on a two dimensional array a. In other words, the goal of permutation is to move a word of data stored in $a[i][j]$ to $a[P(i \cdot w + j)/w][P(i \cdot w + j) \bmod w]$ for every i and j. We first assume that a permutation P is row-wise, that is, $P(i \cdot w + j)/w = i$ for all i and j. We use p processors $(w \leq p \leq w^2)$. In other words, we have $\frac{p}{w}$ warps $W(0), W(1), \ldots, W(\frac{p}{w}-1)$ with w processors each. The details of the row-wise permutation algorithm are as follows.

[Row-wise permutation algorithm]

```
for t ← 0 to w^2/p − 1
  for i ← 0 to p/w do in parallel
    W(i) performs permutation of the (t · p/w + i)-th row.
```

For each t and i, $W(i)$ can perform a permutation of a row in $O(l)$ time. Hence, for each t, $W(0), W(1), \ldots, W(\frac{p}{w})$ can perform row-wise permutation of $\frac{p}{w}$ rows in $O(\frac{p}{w} + l)$ time. Thus, the row-wise permutation algorithm terminates in $\frac{w^2}{p} \cdot (\frac{p}{w} + l) = O(w + \frac{w^2 l}{p})$ time.

We next show any permutation can be done in $O(w + \frac{w^2 l}{p})$ time on the UMM using the row-wise permutation algorithm and Theorem 4. For a given permutation P on a two dimensional array a, we draw a bipartite graph $G = (U, V, E)$ as follows:

- $U = \{0, 1, \ldots, w-1\}$ is a set of w nodes such that each node corresponds to a column of a.
- $V = \{0, 1, \ldots, w-1\}$ is a set of w nodes such that each node corresponds to a row of a.
- E is a set of w^2 edges such that, an edge connecting node i in U and node j in V corresponds to a word of data moved from the i-th column to the j-th row of a by permutation P.

For example if a word of data in $a[1][3]$ is moved to $a[2][4]$ by permutation, an edge is drawn from node 1 in U and node 4 in V. Clearly, G is a regular bipartite graph with degree w. From Theorem 4, this bipartite graph can be painted using w colors such that w edges painted by the same color never share a node.

Suppose that, for a given permutation P on two dimensional array a of size $w \times w$, we have painted edges in w colors $c_0, c_1, \ldots c_{w-1}$. Let (i, j) denote a word of data stored in $a[i][j]$ initially. We can think that (i, j) is assigned one of the w colors. It should be clear that w words of data in each row are painted by distinct w colors. The key idea of permutation is to move words of data using painted colors. The details of the permutation routing on the UMM are spelled out as follows.

[Permutation on the UMM]

Step 1: Move words of data such that a word with color c_i

is moved to the i-th column in the same row by the row-wise permutation.
Step 2: Transpose two dimensional array a.
Step 3: Move words of data such that a word with destination i-th row is stored in the i-th column in the same row.
Step 4: Transpose two dimensional array a.
Step 5: Move words of data such that a word with destination i-th column is stored in the i-th column by the row-wise permutation.

Let us see the data movement of the above permutation algorithm for $w = 4$ using Figure 8 and confirm the correctness of the algorithm. We assume that each $a[i][j]$ is initially storing a word data (x, y) if a data stored in $a[i][j]$ should be moved to $a[x][y]$ along permutation as illustrated in the figure. In the figure, every destination is painted by four colors such that

- four destinations in each row are painted by different colors, and
- four destinations painted by a particular color has four distinct row destinations $(0, *)$, $(1, *)$, $(2, *)$, and $(3, *)$.

After Step 1, four destinations in the i-th column are painted by i-th color, and thus they are four distinct row destinations. After Step 2, each row has four destinations with distinct row destinations. Hence after Step 3, four destinations to be moved to i-th row are in the i-th column. By transposing in Step 4, every destination is in the right row destination. Finally, permutation is completed by row-wise permutation in Step 5.

Clearly, Steps 1, 3, and 5 that involve row-wise permutation can be done in $O(w + \frac{w^2 l}{p})$ time units on the UMM. From Lemma 3, transpose performed in Steps 2 and 4 can be done in $O(w + \frac{w^2 l}{p})$ time on the UMM with each processor having local memory of w words. Thus, we have

Lemma 7: Any permutation of an array of size w^2 can be done in $O(w + \frac{w^2 l}{p})$ time on the UMM with each processor having local memory of w words.

We go on to show permutation algorithm on a larger array a. Suppose we need to perform permutation of array a of size w^4. We can consider that an array a is a two dimensional array of size $w^2 \times w^2$. We use the permutation algorithm for Lemma 7 for row-wise permutation of the two dimensional array of size $w^2 \times w^2$. Similarly to the permutation algorithm for Lemma 7, we generate a bipartite graph with $G = (U, V, E)$ such that U and V has w^2 nodes each and E has w^4 edges corresponding to a word of data. It should be clear that we can perform the permutation on a using G by row-wise permutation and the transpose. First, let us evaluate the computing time for the row-wise permutation. When $w \leq p \leq w^2$, we execute a row permutation in row-by-row. Since each row permutation can be done in $O(w + \frac{w^2 l}{p})$ time, the row-wise permutation can be done in $O(w + \frac{w^2 l}{p}) \cdot w^2 = O(w^3 + \frac{w^4 l}{p})$ time. When $w^2 < p$, we

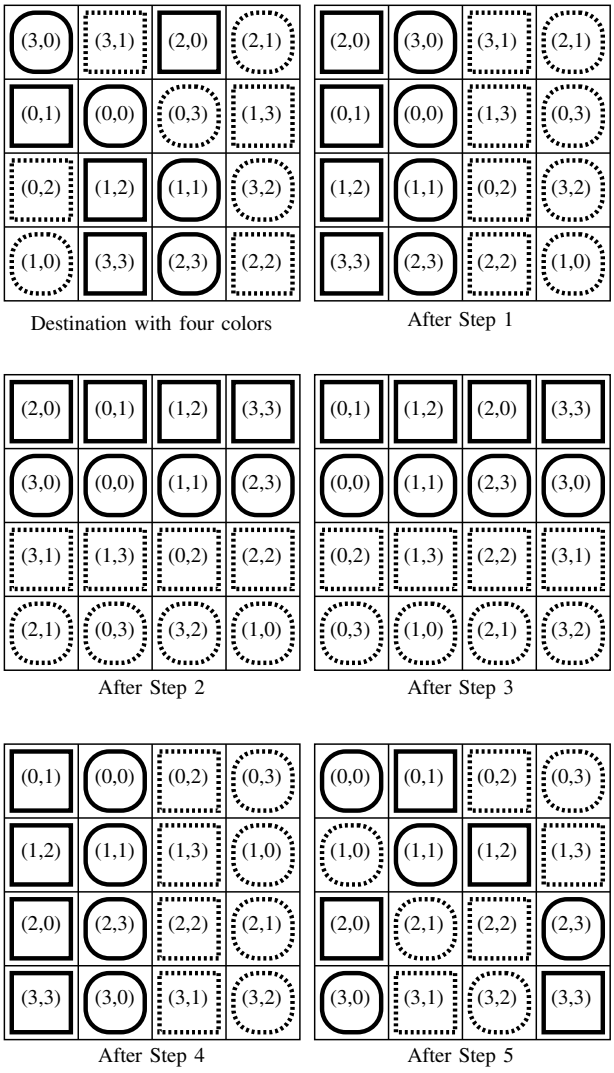


Figure 8. Illustrating a data movement of the permutation algorithm on the UMM

partition the p processors into $\frac{p}{w^2}$ groups of w^2 processors each. Each group of w^2 processors can perform a row permutation in $O(w + l)$ time. Thus, row-wise permutation of $\frac{p}{w^2}$ rows can be done in $O(w \cdot \frac{p}{w^2} + l) = O(\frac{p}{w} + l)$ time using p processors. Since we have w^2 rows, this operation is repeated $\frac{w^4}{p}$ times, the whole permutation can be done in $O(\frac{p}{w} + l) \cdot \frac{w^4}{p} = O(w^3 + \frac{w^4 l}{p})$. Also, the transpose of the two dimensional array of size $w^2 \times w^2$ be done in $O(w^3 + \frac{w^4 l}{p})$ time from Lemma 3. Since the permutation of an array can be done by executing the row-wise permutation three times and the transpose twice, the permutation of an array of size w^4 can be done in $O(w^3 + \frac{w^4 l}{p})$ time.

We can use the same technique for a permutation of an array of size w^8. The readers should have no difficulty to

confirm that any permutation can be done in $O(w^7 + \frac{w^8 l}{p})$ time on the UMM using p processors.

Repeating the same technique, we can obtain a permutation of an array of size $n = w^{2^m}$. In other words, the permutation of an array of size w^{2^m} can be done by executing the row-wise permutation recursively three times and the transpose twice for an array of size $w^{2^{m-1}}$. If the size n of an array satisfies $n \leq w^{O(1)}$, that is, $m = O(1)$, then the depth of the recursion is constant. Thus, we have,

Theorem 8: Any permutation of an array of size n can be done in $O(\frac{n}{w} + \frac{nl}{p})$ time on the UMM with each processor having local memory of w words, provided that $n \leq w^{O(1)}$.

VII. Conclusion

In this paper, we have introduced two parallel memory machines, the Discrete Memory Machine (DMM) and the Unified Memory Machine (UMM). We first evaluated the computing time of the contiguous access and the stride access of the memory on the DMM and the UMM. We then presented an algorithm to transpose a two dimensional array on the DMM and the UMM. Finally, we have shown that any permutation of an array of size n can be done in $O(\frac{n}{w} + \frac{nl}{p})$ time units on the DMM and the UMM with width w and latency l. Since the computing time just involves the bandwidth limitation factor $\frac{n}{w}$ and the latency overhead $\frac{nl}{p}$, the permutation algorithms are optimal.

Although the DMM and the UMM are simple, they capture the characteristic of the shared memory and the global memory of NVIDIA GPUs, Thus, these two parallel computing models are promising for developing algorithmic techniques for NVIDIA GPUs. As a future work, we plan to implement various parallel algorithms developed for the PRAM so far on the DMM and on the UMM. Also, NVIDIA GPUs have small shared memory and large global memory. Thus, it is also interesting to consider a hybrid memory machine such that processors are connected to a small memory of DMM and a large 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] D. Man, K. Uda, Y. Ito, and K. Nakano, "A GPU implementation of computing euclidean distance map with efficient memory access," in *Proc. of International Conference on Networking and Computing*, Dec. 2011, pp. 68–76.

[7] A. Uchida, Y. Ito, and K. Nakano, "Fast and accurate template matching using pixel rearrangement on the GPU," in *Proc. of International Conference on Networking and Computing*, Dec. 2011, pp. 153–159.

[8] Y. Ito, K. Ogawa, and K. Nakano, "Fast ellipse detection algorithm using hough transform on the GPU," in *Proc. of International Conference on Networking and Computing*, Dec. 2011, pp. 313–319.

[9] K. Nishida, Y. Ito, and K. Nakano, "Accelerating the dynamic programming for the matrix chain product on the GPU," in *Proc. of International Conference on Networking and Computing*, Dec. 2011, pp. 320–326.

[10] *NVIDIA CUDA C Programming Guide Version 4.0*, 2011.

[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 CUDA C Best Practice Guide Version 3.1*, 2010.

[13] N. K. Govindaraju, S. Larsen, J. Gray, and D. Manocha, "A memory model for scientific algorithms on graphics processors," in *Proc. of the ACM/IEEE Conference on Supercomputing*, 2006, pp. 6–6.

[14] S. Ryoo, C. I. Rodrigues, S. S. Baghsorkhi, S. S. Stone, D. B. Kirk, and W. mei W. Hwu, "Optimization principles and application performance evaluation of a multithreaded gpu using cuda," in *Proc. of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, 2008, pp. 73–82.

[15] K. Nakano, "Optimal sorting algorithms on bus-connected processor arrays," *IEICE Trans. Fundamentals*, vol. E76-A, no. 11, pp. 2008–2015, Nov. 1993.

[16] R. J. Wilson, *Introduction to Graph Theory, 3rd edition*. Longman, 1985.