The International Journal of Parallel, Emergent and Distributed Systems
Vol. 00, No. 00, Month 2011, 1–22

RESEARCH ARTICLE

Simple Memory Machine Models for GPUs

Koji Nakano[a*]

[a]*Department of Information Engineering, Hiroshima University*
Kagamiyama 1-4-1, Higashi Hiroshima, 739-8527 Japan
(*Received 00 Month 200x; in final form 00 Month 200x*)

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 GPU memory accesses. As a first step of the development of algorithmic techniques on the DMM and the UMM, we first evaluate 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 2-dimensional array on these models and evaluate their performance. Finally, we show that, for any permutation given in off-line, data in an array can be moved efficiently along the 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 limitation, they are optimal from the theoretical point of view. We believe that the DMM and the UMM can be good theoretical platforms to develop algorithmic techniques for GPUs.

Keywords: Memory banks, Parallel computing models, Parallel algorithms, Matrix transpose, Array permutation, GPU, CUDA

1. Introduction

1.1 Background

The research of parallel algorithms has a long history of more than 40 years. Sequential algorithms have been developed mostly on the Random Access Machine (RAM) [1]. In contrast, since there are a variety of connection methods and patterns between processors and memories, many parallel computing models have been presented and many parallel algorithmic techniques have been shown on them. The most well-studied parallel computing model is the Parallel Random Access Machine (PRAM) [2–4], which consists of processors and a shared memory. Each processor on the PRAM can access any address of the shared memory in a time unit. The PRAM is a good parallel computing model in the sense that parallelism of each problem can be revealed by the performance of parallel algorithms on the PRAM. However, since the PRAM requires a shared memory that can be accessed by all processors at the same time, it is imaginary and impractical.

The GPU (Graphical Processing Unit), is a specialized circuit designed to accelerate computation for building and manipulating images [5–8]. Latest GPUs are designed for general purpose computing and can perform computation in applications traditionally handled by the CPU. Hence, GPUs have recently attracted the attention of many application developers [5, 9, 10]. NVIDIA provides a parallel

*Corresponding author. Email: nakano@cs.hiroshima-u.ac.jp

ISSN: 1744-5760 print/ISSN 1744-5779 online

DOI: 10.1080/17445760.YYYY.CATSid
http://www.informaworld.com

computing architecture called *CUDA* (Compute Unified Device Architecture) [11], 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 [12], 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* [11]. 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 [6, 12, 13]. 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.

There are several previously published works that aim to present theoretical practical parallel computing models capturing the essence of parallel computers. Many researchers have been devoted to developing efficient parallel algorithms to find algorithmic techniques on such parallel computing models. For example, processors connected by interconnection networks such as hypercubes, meshes, trees, among others [14], bulk synchronous models [15], LogP models [16], reconfigurable models [17], among others. As far as we know, no sophisticated and simple parallel computing model for GPUs has been presented. Since GPUs are attractive parallel computing devices for many developers, it is challenging work to introduce a theoretical parallel computing model for GPUs.

1.2 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 machine models that capture the essential features of the bank conflict of the shared memory access and the coalescing of the global 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.

The outline of the architectures off the DMM and the UMM are illustrated in Figure 1. In both architectures, *a sea of threads (Ts)* are connected to *the memory banks (MBs)* through *the memory management unit (MMU)*. Each thread is a Random Access Machine (RAM) [1], which can execute fundamental operations in a time unit. We do not discuss the architecture of the sea of threads in this paper, but we can imagine that it consists of a set of multi-core processors which can execute many threads in parallel. Threads are executed in SIMD [18] fashion, and the processors run on the same program and work on the different data. In principle, each thread is assigned a local memory (or local registers) that can access $O(1)$ words of data. However, sometimes, we assume that each thread has more than $O(1)$ local registers, if many registers are very useful to accelerate the computation. If this is the case, we assume that each thread has r local registers

to store words of data. In either cases, we assume that each thread can access a local register in 1 time unit.

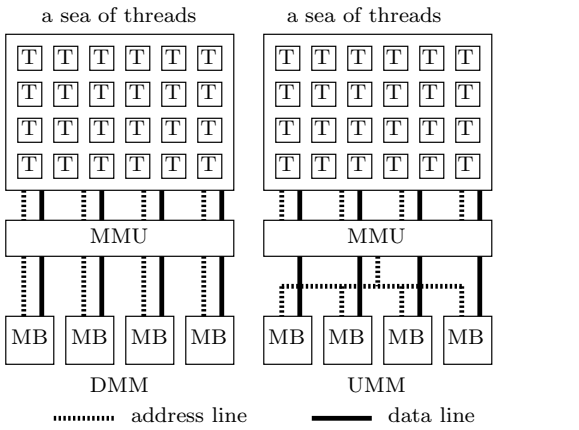

Figure 1. The architectures of the DMM and the UMM

MBs constitute a single address space of the memory. A single address space of the memory is mapped to the MBs in an interleaved way such that the word of data of address i is stored in the $(i \bmod w)$-th bank, 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.

The performance of algorithms on the PRAM is usually evaluated using two parameters: the size n of the input and the number p of processors. For example, it is well known that the sum of n numbers can be computed in $O(\frac{n}{p} + \log p)$ time on the PRAM [2]. We will use four parameters, the size n of the input, the number p of threads, the width w and the latency l of the memory when we evaluate the performance of algorithms on the DMM and on the UMM. The width w is the number of memory banks and the latency l is the number of time units to complete the memory access. Hence, the performance of algorithms on the DMM and the UMM is evaluated as a function of n (the size of a problem), p (the number of threads), w (the width of a memory), and l (the latency of a memory). Further, r (the number of local registers used by each thread) may be additionally used.

In NVIDIA GPUs, the width w of the shared memory and the global memory is 16 or 32. Also, the latency l of the global memory is several hundreds clock cycles. In CUDA, a grid can have at most 65535 blocks with at most 1024 threads each [11]. Thus, the number p of threads can be 65 million.

1.3 Position and Role of Memory Machine Models, the DMM and the UMM

The DMM and the UMM are theoretical models of parallel computation, that capture the essential feature of the shared memory and the global memory of GPUs. The architecture of the GPUs are more complicated. It is a hybrid of the DMM

and the UMM. Also, when we develop efficient programs running on the GPUs, we need to consider several issues. NVIDIA GPUs have other features such as hierarchical architecture grid/block/thread. All threads are partitioned into equal sized blocks. Synchronization of all threads in each block can be done by calling barrier synchronization function `__syncthreads()`, which has fairly low overhead. On the other hand, no direct way is provided for synchronization of all threads in all blocks. There are several indirect ways of synchronization of all threads, but they have rather high overhead. It follows that, local barrier synchronization is acceptable while global barrier synchronization should be avoid. This fact is not incorporated in the DMM and the UMM. It may be possible to incorporate many features of GPUs and introduce a more exact parallel computing model for GPUs. If all features of GPUs are incorporated in our theoretical parallel models, they will be too complicated and need more parameters. The development of algorithms on such complicated models may have too much non-essential and tedious optimizations. Thus, we focus on just memory access features on the current GPUs, and introduce parallel computing models, the DMM and the UMM. Actually, efficient memory access is a key issue to develop high performance programs on the GPUs [13, 19]. Thus, we have introduced two simple parallel models, the DMM and the UMM, which focus on the memory access to the shared memory and the global memory of NVIDIA GPUs. Sometimes, direct implementation of efficient algorithms on the DMM and the UMM may not be efficient on an actual GPU. However, we believe that algorithmic techniques on the DMM and the UMM are useful for developing algorithms on GPUs.

In [20], a GPU memory model has been shown and a cache-efficient FFT has been presented. However, their model focuses on the cache mechanism and ignores the coalescing and the bank conflict. Also, in [21], 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.

Further, the parallel architecture of our memory machines make senses not only for GPUs, but also for a class of all parallel machines that support a uniform shared address space designed using a set of off-chip memory chips or on-chip memory blocks. Usually, DRAMs [22] are used to constitute an off-chip memory. An on-chip memory block can be implemented in a rectangular block of a VLSI chip. For example, modern FPGAs has a lot of block RAMs, each of which can store 18kbit data [23], can be used as a memory bank. To increase the capacity and the bandwidth, we should use multiple on-chip memory chips or on-chip memory blocks. To connect a set of processor cores with these memory elements though the MMU, the architecture of the UMM and the DMM make a whole lot of sense.

1.4 Our Contribution: Fundamental Data Movement 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 by four threads $T(0), T(1), T(2)$, and $T(3)$. It is well-known that the contiguous access is much more efficient than the stride access on the GPUs [13]. We will show that, the contiguous access is also more efficient on the DMM and on the UMM. More specifically, we first show that the contiguous access of an array of size n

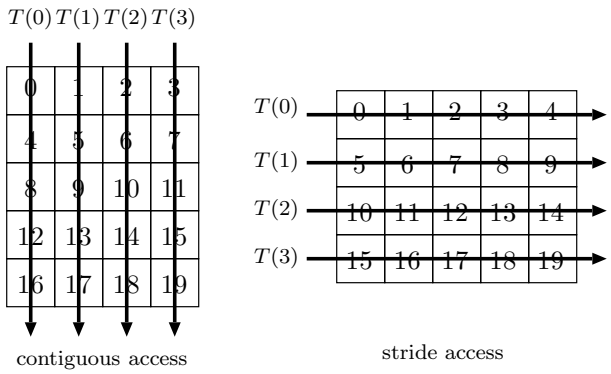

Figure 2. The contiguous access and the stride access for $p = 4$ and $n = 16$.

can be done in $O(\frac{n}{w} + \frac{nl}{p})$ time units on the DMM and the UMM. We also show two lower bounds, $\Omega(\frac{n}{w})$ time units by *the bandwidth limitation* and $\Omega(\frac{nl}{p})$ time units by *the latency limitation* to access all of data in an array of size n. Thus, the contiguous access on the DMM and the UMM is optimal. Further, we will show that the stride access on 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 on the DMM is optimal if $\frac{n}{p}$ and w are co-prime. The stride access on 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 on the UMM needs an overhead of a factor of $\frac{n}{p}$.

From these memory access results, 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 limitation*. From the memory access latency l, each thread cannot send a new access request in l time units. It follows that, each thread can access the memory once in l time units and any consecutive l time units can have at most p access requests by p threads. 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 threads must be no less than wl. We can confirm this fact from a different aspect. We can think that *the memory access requests* 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 thread are stored in the wl pipeline registers, $wl \leq p$ must be satisfied to fill the pipeline registers full of memory access requests.

1.5 Our Contribution: Transpose and Permutation on the DMM and the UMM

The third contribution is to show optimal off-line permutation algorithms on the DMM and the UMM.

As a preliminary step, we show transposing algorithms for a 2-dimensional array of size $\sqrt{n} \times \sqrt{n}$. In [19], several techniques are presented for transposing a 2-dimensional array stored in the shared memory and the global memory on GPUs. We have adapted these techniques on the DMM and the UMM. The resulting transposing algorithms run in $O(\frac{n}{w} + \frac{nl}{p})$ time units and in $O((\frac{n}{w} + \frac{nl}{p})\sqrt{\frac{w}{r}})$ time units on the DMM and the UMM, respectively.

We next show a permutation algorithm on the DMM. We use a graph theoretic result of bipartite graph edge-coloring to schedule data routing. The resulting algorithm runs in $O(\frac{n}{w} + \frac{nl}{p})$ time units on the DMM.

Finally, we show a permutation algorithm on the UMM. This algorithm repeatedly performs transposing and row-wise permutation. The resulting algorithm runs in $O((\frac{n}{w} + \frac{nl}{p})\sqrt{\frac{w}{r}})$ time units on the UMM, respectively.

This paper is organized as follows. We first define the DMM and the UMM in Section 2. In Section 3, we evaluate the performance of the DMM and the UMM for the contiguous access and the stride access to the memory. Section 4 discusses lower bounds obtained by *the bandwidth limitation* and *the latency limitation*. Section 5 presents algorithms that perform the transpose of 2-dimensional array on the DMM and the UMM. In Section 6, we show that any permutation on an array can be done efficiently on the DMM. Finally, Section 7 presents a permutation algorithm on the UMM.

2. Parallel Memory Machines: DMM and UMM

We first 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* of the memory. Clearly, a memory cell $m[i]$ is in the $(i \bmod w)$-th memory bank. We assume that memory cells in different banks can be accessed in a time unit, but no two memory cells in the same bank can be accessed in a time unit. Also, we assume that l time units are necessary to complete an access request and continuous requests are processed in a pipeline fashion through the MMU. Thus, it takes $k+l-1$ time units to complete k continuous access requests to a particular bank.

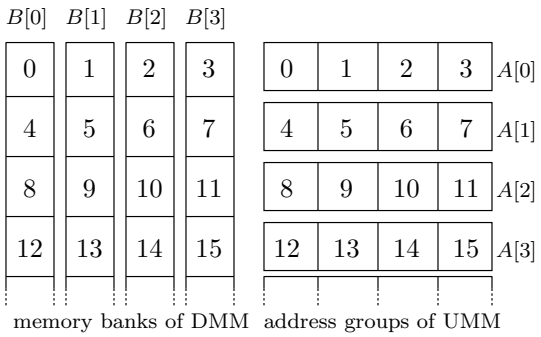

Figure 3. Banks and address groups for $w = 4$

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

previous memory access request is completed. Hence, if a thread send a memory access request, it must wait l time units to send a new memory access request.

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

Note that, the architecture of pipeline registers illustrated in Figure 4 are imaginary, and it is used only for evaluating the computing time. The actual architecture should involves a multistage interconnection network [24, 25] or sorting network [26, 27], to route memory access requests.

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

We next define *the Unified Memory Machine* (*UMM* 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 at the same time. However, if they are in the different groups, one time unit is necessary for each of the groups. Also, similarly to the DMM, p threads are partitioned into warps and each warp 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 $l = 3$. The memory access requests by $W(0)$ are in three address groups. Thus, three time units are necessary to send them to the first stage. Next, two time units are necessary to send memory access requests by $W(1)$, because they are in two address groups. After that, it takes $l - 1 = 2$ time units to process the memory access requests. Hence, totally $3 + 2 + 2 = 7$ time units are necessary to complete all memory accesses.

3. Sequential memory access operations

We begin with simple operations to see the potentiality of the DMM and the UMM. Let p and w be the number of threads and the width of the memory 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 $w \leq p$ and n is divisible by p. We consider two access operations to the memory such that each of the p threads accesses the $\frac{n}{p}$ memory cells out of the n memory cells. Suppose that array m is arranged in a 2-dimensional array m_c of size $\frac{n}{p} \times p$ (i.e. $\frac{n}{p}$ rows and p columns) such that $m_c[i][j] = m[i \cdot p + j]$ for all i and j ($0 \leq i \leq \frac{n}{p} - 1$ and $0 \leq j \leq p - 1$). Similarly, let m_s be a 2-dimensional array of size $p \times \frac{n}{p}$ (i.e. p rows and $\frac{n}{p}$ columns) such that $m_s[i][j] = m[i \cdot \frac{n}{p} + j]$ for all i and j ($0 \leq i \leq p - 1$ and $0 \leq j \leq \frac{n}{p} - 1$).

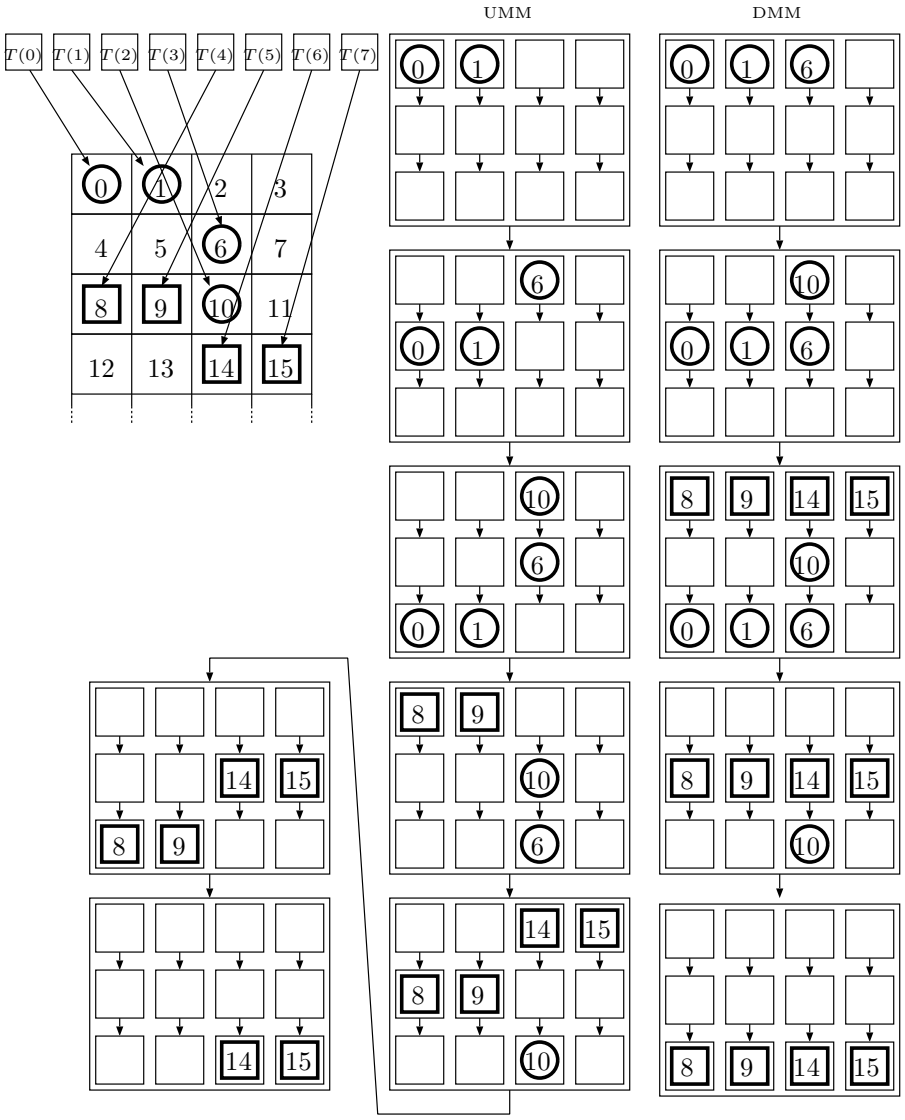

Figure 4. An example of memory access

The contiguous access and the stride access can be written as follows:

[Contiguous Access]
for $t \leftarrow 0$ to $\frac{n}{p} - 1$
  for $i \leftarrow 0$ to $p - 1$ do in parallel
    $T(i)$ accesses to $m_c[t][i] \quad (= m[t \cdot p + i])$

[Stride Access]
for $t \leftarrow 0$ to $\frac{n}{p} - 1$
  for $i \leftarrow 0$ to $p - 1$ do in parallel
    $T(i)$ accesses to $m_s[i][t] \quad (= m[i \cdot \frac{n}{p} + t])$

The readers should refer to Figure 2 for illustrating the contiguous and stride accesses for $n = 20$, $p = 4$, and $\frac{n}{p} = 5$. At time $t = 0$, p threads access contiguous locations $m[0], m[1], m[2]$, and $m[3]$ in the contiguous access, while they access distant locations $m[0], m[5], m[10]$, and $m[15]$ in the stride access.

Let us evaluate the time necessary to complete the contiguous access and the stride access. In the contiguous access, w threads in each warp access memory cells in different memory banks. Hence, the memory access by a warp takes l time units. Also, *the memory access requests* by a warp is sent in every 1 time unit. Since we have $\frac{p}{w}$ warps, the access to p memory cells by p threads can be completed in $\frac{p}{w}+l-1$ time units. Since this access operation is repeated $\frac{n}{p}$ times, the contiguous access takes $(\frac{p}{w}+l-1)\cdot\frac{n}{p} = O(\frac{n}{w}+\frac{nl}{p})$ 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 is completed in $O(\frac{n}{w}+\frac{nl}{p})$ time units.

The performance analysis of the stride access on the DMM is a bit complicated. Let us start with a simple case: $\frac{n}{p} = w$. In this case, the p threads access 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, the memory access by a warp takes $w+l-1$ time units and the memory access to the p memory cells takes $w\cdot\frac{p}{w}+l-1 = p+l-1$ time units. Thus, the stride access when $\frac{n}{p} = w$ 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 threads in the first warp access $m[t], m[\frac{n}{p}+t], m[2\frac{n}{p}+t], \ldots, m[(w-1)\frac{n}{p}+t]$ for each t $(0 \leq t \leq w-1)$. These w memory cells are allocated in the banks $B[t \bmod w], B[(\frac{n}{p}+t) \bmod w], B[(2\frac{n}{p}+t) \bmod w], \ldots, B[((w-1)\frac{n}{p}+t) \bmod w]$. Let $L = \mathrm{LCM}(\frac{n}{p}, w)$ and $G = \mathrm{GCD}(\frac{n}{p}, w)$ be the Least Common Multiple and the Greatest Common Divisor of $\frac{n}{p}$ and w, respectively. From the basic number theory, it should be clear that $t \bmod w = (\frac{L}{\frac{n}{p}}\cdot\frac{n}{p}+t) \bmod w$, and the values of $t \bmod w$, $(\frac{n}{p}+t) \bmod w$, …, $((\frac{L}{\frac{n}{p}}-1)\cdot\frac{n}{p}+t) \bmod w$ are distinct. Thus, the w memory cells are in the $\frac{L}{\frac{n}{p}} = \frac{w}{G}$ banks $B[t \bmod w], B[(\frac{n}{p}+t) \bmod w], B[(2\frac{n}{p}+t) \bmod w], \ldots, B[((\frac{w}{G}-1)\frac{n}{p}+t) \bmod w]$ equally, and each bank has G memory cells of the w memory cells. Hence, the w threads in a warp take $G+l-1$ time units for each t, and the p threads take $G\cdot\frac{p}{w}+l-1$ time units for each t. Therefore, the DMM takes $(G\cdot\frac{p}{w}+l-1)\cdot\frac{n}{p} = O(\frac{nG}{w}+\frac{nl}{p})$ time units to complete the stride access. If $\frac{n}{p} = w$ then $G = w$ and the time for the stride access is $O(n+\frac{nl}{p})$. If $\frac{n}{p}$ 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 $\frac{n}{p} \geq w$ (i.e. $n \geq pw$), then the w memory cells are accessed by w threads in a warp are in the different address group. Thus, w threads access 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 units. When $\frac{n}{p} < w$ (i.e. $n < pw$), the w memory cells accessed by w threads in a warp are in at most $\lceil\frac{(w-1)\frac{n}{p}+1}{w}\rceil \leq \frac{n}{p}$ address groups. Hence, the stride access by p threads for each t takes at most $\frac{n}{p}\cdot\frac{p}{w}+l-1 = \frac{n}{w}+l-1$ time units, and thus, the whole stride access takes $(\frac{n}{w}+l-1)\cdot\frac{n}{p} = O(\frac{n^2}{pw}+\frac{nl}{p})$ time units. Consequently, the stride access can be completed in $O(\min(n, \frac{n^2}{pw})+\frac{nl}{p}))$ time units for all values of $\frac{n}{p}$. Thus, we have,

THEOREM 3.1 . *The contiguous access and the stride access on the DMM and the UMM can be completed in time units shown in Table 1.*

Suppose that we have two arrays a and b of size n each. The copy operation from a and b can be done by the contiguous read and the contiguous write in an obvious way. Since both the DMM and the UMM can perform the contiguous access in

Table 1. The running time for the contiguous access and the stride access

	DMM	UMM
Contiguous Access	$O(\frac{n}{w}+\frac{nl}{p})$	$O(\frac{n}{w}+\frac{nl}{p})$
Stride Access	$O(\frac{n}{w}\cdot \mathrm{GCD}(\frac{n}{p},w)+\frac{nl}{p})$	$O(\min(n,\frac{n^2}{pw})+\frac{nl}{p})$

n =#data, p =#threads, w =memory bandwidth, l =memory latency

$O(\frac{n}{w}+\frac{nl}{p})$ time units from Theorem 3.1, we have,

COROLLARY 3.2. *The copy between two arrays of size n each can be done in $O(\frac{n}{w}+\frac{nl}{p})$ time units using p threads on the DMM and on the UMM with width w and latency l.*

4. The lower bounds of the computing time and the latency hiding

Let us discuss the lower bound of the computing time of the DMM and the UMM for non-trivial problems, which require to access all words in an input array of size n.

Since the bandwidth of the memory is w, at most w words in the memory can be accessed in a time unit. Thus, it takes at least $\Omega(\frac{n}{w})$ time to solve a non-trivial problem. We call the $\Omega(\frac{n}{w})$-time lower bound *the bandwidth limitation*.

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

From the discussion above, we have,

THEOREM 4.1. *Both the DMM and the UMM with p threads, width w, and latency l takes at least $\Omega(\frac{n}{w}+\frac{nl}{p})$ time units to solve a non-trivial problem of size n.*

From Theorem 4.1, the copy operation for Corollary 3.2 is optimal. In the following sections, we will show algorithms for data movement running in $O(\frac{n}{w}+\frac{nl}{p}$ time. Since data movements are non-trivial problems, they have a lower bound of $\Omega(\frac{n}{w}+\frac{nl}{p})$ time units. Hence, the algorithms for data movement are optimal.

Let us discuss two factors, $\frac{n}{w}$ for bandwidth limitation and $\frac{nl}{p}$ for latency limitation. If $\frac{n}{w}\geq\frac{nl}{p}$, that is, $wl\leq p$, then the bandwidth limitation dominates the latency limitation. As illustrated in Figure 4, both the DMM and the UMM have wl imaginary pipeline registers. Each thread can occupy one of the wl imaginary pipeline registers for memory access. Thus, we need at least wl threads to fill all the pipeline registers with memory access requests. Otherwise, that is, if $wl>p$, then a set of wl pipeline registers always has an empty one. It follows that, for the purpose of hiding the latency overhead, the number p of threads must be at least the number wl of the pipeline registers.

5. Transpose of a 2-dimensional array

Suppose that a 2-dimensional array a and b of size $\sqrt{n}\times\sqrt{n}$ is arranged in the memory. The transpose of the 2-dimensional array is a task to move a word of data stored in $a[i][j]$ to $b[j][i]$ 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 2-dimensional array a of size $\sqrt{n} \times \sqrt{n}$.

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

On the PRAM, simultaneous reading and simultaneous writing by processors can be done in $O(1)$ time. Hence, this straightforward transposing algorithm runs in $O(\frac{n}{p})$ time on the PRAM. Also, it takes at least $\Omega(\frac{n}{p})$ time to access n words by p processors on the PRAM. Thus, this straightforward transposing algorithm is time optimal for the PRAM.

Since the straightforward algorithm involves the stride access, it is not difficult to see that the DMM and the UMM take $O(\frac{n}{w} \cdot \mathrm{GCD}(\sqrt{n}, w) + \frac{nl}{p})$ time units and $O(\min(n, \frac{n^2}{pw}) + \frac{nl}{p})$ time units for transposing a 2-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 units both on the DMM and on the UMM.

We first show an efficient transposing algorithm on the DMM. The technique used in this algorithm is essentially the same as the diagonal block reordering presented in [19]. The key idea is to access the array in diagonal fashion. The details of the algorithm are spelled out as follows:

[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
  $j \leftarrow (t \cdot p + i)/\sqrt{n}$
  $k \leftarrow (t \cdot p + i) \bmod \sqrt{n}$
  $T(i)$ performs $b[(j + k) \bmod \sqrt{n}][k] \leftarrow a[k][(j + k) \bmod \sqrt{n}]$

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

LEMMA 5.1. *The transpose of a 2-dimensional array of size $\sqrt{n} \times \sqrt{n}$ can be done in $O(\frac{n}{w} + \frac{nl}{p})$ time units using p threads on the DMM with memory width w and latency l.*

Next, we will show that the transpose of a 2-dimensional array can be also done in $O(\frac{n}{w} + \frac{nl}{p})$ on the UMM if every thread has w local registers. As a preliminary step, we will show that the UMM can transpose a 2-dimensional array of size $w \times w$ in wl time units using w threads with each thread having a local storage of size w. We assume that each thread has w local registers. Let $r_i[0], r_i[1], \ldots r_i[w-1]$ denote w local registers of $T(i)$.

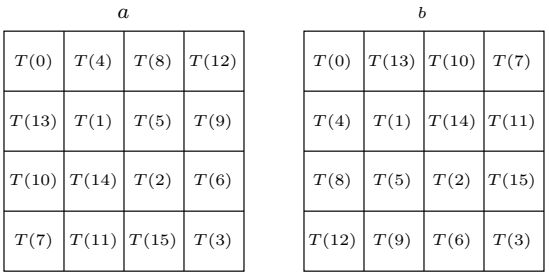

Figure 5. Transposing on the DMM

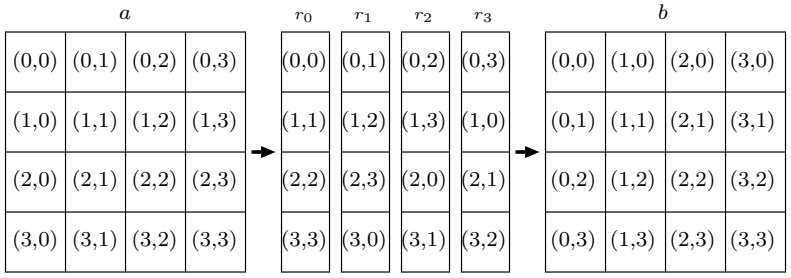

Figure 6. Transposing of a 2-dimensional array of size $w \times w$ on the UMM

[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
    $T(i)$ performs $r_i[t] \leftarrow a[t][(t+i) \bmod w]$
for $t \leftarrow 0$ to $w-1$
  for $i \leftarrow 0$ to $w-1$ do in parallel
    $T(i)$ performs $b[t][(t-i) \bmod w] \leftarrow r_i[(t-i) \bmod w]$

Let (i, j) denote the value stored in $a[i][j]$ initially. The readers should refer to Figure 6 for illustrating how these values are transposed.

Let us confirm that the algorithm above correctly transpose the 2-dimensional array a. In other words, we will show that, when the algorithm terminates, $b[i][j]$ stores (j, i). It should be clear that, the value stored in $r_i[t]$ is $(t, (t+i) \bmod w)$. Since $((t-i) \bmod w, t)$ is stored in $r_i[(t-i) \bmod w]$, it is also stored in $b[t][(t-i) \bmod w]$ when the algorithm terminates. Thus, every $b[i][j]$ $(0 \leq i, j \leq w-1)$ 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 $r_i[t] \leftarrow a[t][(t+i) \bmod w]$, 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 $b[t][(t-i) \bmod w] \leftarrow r_i[(t-i) \bmod w]$, w memory cells $b[t][(t-0 \bmod w)], b[t][(t-1 \bmod w)], \ldots, b[t][(t-(w-1) \bmod w)]$ are in the different memory banks. Thus, each reading and writing operation can be done in $O(l)$ time units and this algorithm runs in $O(wl)$ time units.

The transpose of a larger 2-dimensional array of size $\sqrt{n} \times \sqrt{n}$ can be done by repeating the transpose of a 2-dimensional array of size $w \times w$. The algorithm has two steps. More specifically, the 2-dimensional array is partitioned into $\frac{\sqrt{n}}{w} \times \frac{\sqrt{n}}{w}$ subarrays of size $w \times w$. Let $A[i][j]$ $(0 \leq i, j \leq \frac{n}{w} - 1)$ denote the subarray of size $w \times w$. First, each subarray $A[i][j]$ is transposed independently using w threads

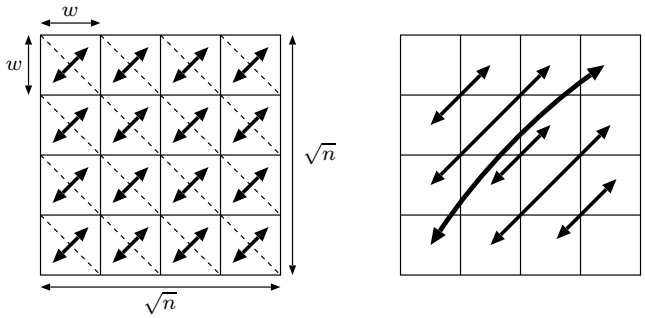

Figure 7. Transposing on the UMM

(local transpose). After that, the corresponding words of $A[i][j]$ and $A[j][i]$ are swapped for all i and j in an obvious way (global transpose). Figure 7 illustrates the transposing algorithm on the UMM.

Let us evaluate the computing time to complete the transpose of a $\sqrt{n} \times \sqrt{n}$ 2-dimensional array. Suppose that we have p ($\leq \frac{n}{w}$) threads and partition the p threads into $\frac{p}{w}$ groups with w threads each. We assign $\frac{n}{w^2}/\frac{p}{w} = \frac{n}{pw}$ subarrays to each warp of w threads. Each of the $\frac{p}{w}$ warps transposes each of the $\frac{p}{w}$ subarrays in parallel. It takes $O(w \cdot (\frac{p}{w} + l)) = O(p + wl)$ time units. The transposing of $\frac{p}{w}$ subarrays is repeated $\frac{n}{pw}$ times, the total computing time for transposing all subarrays is $\frac{n}{pw} \cdot O(p + wl) = O(\frac{n}{w} + \frac{nl}{p})$ time units. It should have no difficulty to confirm that the global transpose can be also done in $O(\frac{n}{w} + \frac{nl}{p})$ time units. Thus we have,

LEMMA 5.2. *The transpose of a 2-dimensional array of size $\sqrt{n} \times \sqrt{n}$ can be done in $O(\frac{n}{w} + \frac{nl}{p})$ time using p ($w \leq p \leq \frac{n}{w}$) threads on the UMM with each thread having w local registers.*

Finally, we will show the case that each thread of the UMM has r ($< w$) local registers. We first show how we transpose a 2-dimensional array a of size $\sqrt{rw} \times \sqrt{rw}$ using w threads. We first partition w threads into $\sqrt{rw}$ groups of $\sqrt{\frac{w}{r}}$ threads each. Each group has totally $\sqrt{\frac{w}{r}} \cdot r = \sqrt{rw}$ local registers and works as a single thread with $\sqrt{rw}$ local registers. Each group i ($0 \leq i \leq \sqrt{\frac{w}{r}}$) with $\sqrt{rw}$ local registers can read and store $\sqrt{rw}$ data $a[0][(i+0) \bmod \sqrt{rw}], a[1][(i+1) \bmod \sqrt{rw}], \ldots, a[\sqrt{rw}-1][(i+\sqrt{rw}-1) \bmod \sqrt{rw}]$ in the local registers. After that, they are written into $b[(i+0) \bmod \sqrt{rw}][0], a[(i+1) \bmod \sqrt{rw}][1], \ldots, a[(i+\sqrt{rw}-1) \bmod \sqrt{rw}][\sqrt{rw}-1]$. All groups read and write the arrays in turn, the transpose of a 2-dimensional array a of size $\sqrt{rw} \times \sqrt{rw}$ can be done in $O(l\sqrt{rw})$ time units.

Similarly to Lemma 5.2, we perform the transpose of a 2-dimensional array a of size $\sqrt{n} \times \sqrt{n}$. For this purpose, we partition a into $\sqrt{\frac{n}{rw}} \times \sqrt{\frac{n}{rw}}$ subarrays of size $\sqrt{rw} \times \sqrt{rw}$. Let us evaluate the computing time. The p threads can transpose $\frac{p}{w}$ subarrays in parallel in $O(\sqrt{rw} \cdot (\frac{p}{w} + l)) = O(p\sqrt{\frac{r}{w}} + l\sqrt{rw})$ time. Since we have $\frac{n}{rw}$ subarrays, this transpose operation is repeated $\frac{n}{rw}/\frac{p}{w} = \frac{n}{rp}$ times. Thus, the local transpose can be done in $O(p\sqrt{\frac{r}{w}} + l\sqrt{rw}) \cdot \frac{n}{rp} = O(\frac{n}{\sqrt{rw}} + \frac{nl}{p} \cdot \sqrt{\frac{w}{r}}) = O((\frac{n}{w} + \frac{nl}{p}) \cdot \sqrt{\frac{w}{r}})$ time units. The global transpose is just a copy of data, it can be done in $O(\frac{n}{w} + \frac{nl}{p})$ time units. Hence, we have,

LEMMA 5.3. *The transpose of a 2-dimensional array of size $\sqrt{n} \times \sqrt{n}$ can be done in $O((\frac{n}{w} + \frac{nl}{p}) \cdot \sqrt{\frac{w}{r}})$ time using p ($w \leq p \leq \frac{n}{r}$) threads on the UMM with each thread having r ($r \leq w$) local registers.*

Lemma 5.3 implies that the transpose by the UMM with r local registers has a overhead of factor $\sqrt{\frac{w}{r}}$.

6. Permutation of an array on the DMM

In Section 5, we have presented algorithms to transpose a 2-dimensional array on the DMM and the UMM. 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 5.

Let a and b be one dimensional arrays of size n each, and P be a permutation of $(0, 1, \ldots, n-1)$. The goal of permutation of an array is to copy a word of data stored in $a[i]$ to $b[P(i)]$ for every i $(0 \leq i \leq n-1)$. We assume that, permutation P 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 evaluating the performance of the straightforward permutation algorithm. Suppose we need to do permutation of an array a of size n and permutation P is given.

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

Clearly each t takes $O(1)$ time unit on the PRAM. Hence, the straightforward algorithm runs in $O(\frac{n}{p})$ time units on the PRAM.

This straightforward permutation algorithm also works correctly on the DMM and the UMM. However, it may take a lot of time to complete the permutation. In the worst case, this straightforward algorithm takes $O(n)$ time units on the DMM and the UMM if all writing operation to $b[P(i)]$ 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 threads 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 use the following important graph theoretic result [28, 29]:

THEOREM 6.1 König *A regular bipartite graph with degree ρ is ρ-edge-colorable.*

Figure 8 illustrates an example of a regular bipartite graph with degree 4 painted by 4 colors. Each edge is painted by one of the 4 colors such that no node is connected to edges with the same color. In other words, no two edges with the same color share a node. The readers should refer to [28, 29] for the proof of Theorem 6.1.

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

- $U = \{B[0], B[1], B[2], \ldots, B[w-1]\}$ is a set of nodes each of which corresponds to a bank of a.
- $V = \{B[0], B[1], B[2], \ldots, B[w-1]\}$ is a set of nodes each of which corresponds to a bank of b.
- For each pair source $a[i]$ and destination $b[P(i)]$, E has a corresponding edge

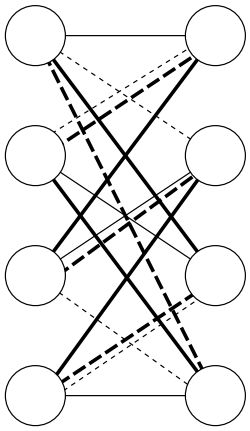

Figure 8. A regular bipartite graph with degree 4

connecting $B[i \bmod w] (\in U)$ and $B[P(i) \bmod w] (\in V)$.

Clearly, an edge $(B[u], B[v])$ $(0 \leq u, v \leq w-1)$ corresponds to a word of data to be copied from bank $B[u]$ of a to $B[v]$ of b. Also, $G = (U, V, E)$ is a regular bipartite graph with degree $\frac{n}{w}$. Hence, G is $\frac{n}{w}$-colorable from Theorem 6.1. Suppose that all of the n edges in E are painted by $\frac{n}{w}$ colors 0, 1, ..., $\frac{n}{w} - 1$. We determine value $s_{i,j}$ $(0 \leq i \leq \frac{n}{w} - 1, 0 \leq j \leq w - 1, 0 \leq s_{i,j} \leq n - 1)$ such that an edge $(B[s_{i,j} \bmod w], B[P(s_{i,j}) \bmod w])$ with color i corresponds to a pair of source $a[s_{i,j}]$ and destination $b[P(s_{i,j})]$. It should have no difficulty to confirm that, for each i,

- w banks $B[s_{i,0} \bmod w], B[s_{i,1} \bmod w], \ldots, B[s_{i,w-1} \bmod w]$ are distinct, and
- w banks values $B[P(s_{i,0}) \bmod w], B[P(s_{i,1}) \bmod w], \ldots, B[P(s_{i,w-1}) \bmod w]$ are distinct.

Thus, we have an important lemma as follows:

LEMMA 6.2. *Let $s_{i,j}$ denote a source defined above. For each i, we have, (1) $a[s_{i,0}]$, $a[s_{i,1}]$, ..., $a[s_{i,w-1}]$ are in different banks, and (2) $b[P(s_{i,0})]$, $b[P(s_{i,1})]$, ..., $b[P(s_{i,w-1})]$ are in different banks.*

We can perform the bank conflict-free permutation using $s_{i,j}$. The details are spelled out as follows.

[Permutation algorithm on the DMM]
for $t \leftarrow 0$ to $\frac{n}{p} - 1$ do
  for $j \leftarrow 0$ to $p - 1$ do in parallel
    $i \leftarrow t \cdot p + j$
    $k \leftarrow s_{i/w, i \bmod w}$
    $T(j)$ performs $b[P(k)] \leftarrow a[k]$

Since $b[P(k)] \leftarrow a[k]$ are performed for all k $(0 \leq k \leq n - 1)$, this algorithm performs data movement along permutation P correctly. We will show that this permutation algorithm terminates in $O(\frac{n}{w} + \frac{nl}{p})$ time units. For $t = 0$, warp $W(q)$ $(0 \leq q \leq \frac{p}{w} - 1)$ with w threads $T(wq), T(wq + 1), \ldots, T(w(q + 1) - 1)$ performs $b[P(s_{q,0})] \leftarrow a[s_{q,0}]$, $b[P(s_{q,1})] \leftarrow a[s_{q,1}]$, ..., $b[P(s_{q,w-1})] \leftarrow a[s_{q,w-1}]$ in parallel. From Lemma 6.2, these w threads read from different banks in a and write to different banks in b. Thus, p threads complete operations for $t = 0$ in $O(\frac{p}{w} + l)$ time units. Similarly, we can prove that the operation for every t can be done in $O(\frac{p}{w} + l)$ time units. Thus the total running time is $\frac{n}{p} \cdot O(\frac{p}{w} + l) = O(\frac{n}{w} + \frac{nl}{p})$ time units. Thus, we have,

THEOREM 6.3 . *Any permutation on an array of size n can be done in $O(\frac{n}{w} + \frac{nl}{p})$ time units using p threads on the DMM with width w and latency l.*

7. 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 5.

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 units.

Next, we show a permutation algorithm for an array a of size w^2. We can consider that a permutation is defined on a 2-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 $(0 \leq i, j \leq w-1)$. We first show an algorithm for the row-wise permutation which is a permutation satisfying $P(i \cdot w + j)/w = i$ for all i and j. Figure 9 shows an example of row-wise permutation. In this figure, we assume that each $a[i][j]$ is initially storing $(P(i \cdot w + j)/w, P(i \cdot w + j) \bmod w]) = (i, P(i \cdot w + j) \bmod w])$. After the permutation, it is copied to $a[i][P(i \cdot w + j) \bmod w]$ and thus, each $a[i][j]$ stores (i, j).

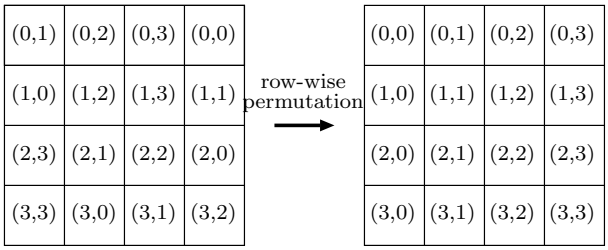

Figure 9. Row-wise permutation

We use p threads $(w \leq p \leq w^2)$ partitioned into $\frac{p}{w}$ warps $W(0), W(1), \ldots, W(\frac{p}{w} - 1)$ with w threads each. The details of the row-wise permutation algorithm are as follows.

[Row-wise permutation algorithm]
for $t \leftarrow 0$ to $\frac{w^2}{p} - 1$
  for $i \leftarrow 0$ to $\frac{p}{w}$ do in parallel
    $W(i)$ performs permutation of the $(t \cdot \frac{p}{w} + i)$-th row.

Clearly, each row of an array a of size w^2 corresponds to an address group. For each t and i, $W(i)$ can perform a permutation of a row in $O(l)$ time units. Hence, for each t, $W(0), W(1), \ldots, W(\frac{p}{w} - 1)$ can perform the row-wise permutation of $\frac{p}{w}$ rows in $O(\frac{p}{w} + l)$ time units. 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 units. Hence we have,

LEMMA 7.1 . *Any row-wise permutation of a two-dimensional array of size $w \times w$ can be done in $O(w + \frac{w^2 l}{p})$ time units using p threads $(w \leq p \leq w^2)$ on the UMM with width w and latency l.*

We next show an algorithm for the column-wise permutation , which is a permutation satisfying $P(i \cdot w + j) \bmod w = j$ for all i and j. This can be done by three

steps as follows:

[Column-wise permutation on the UMM]
Step 1: Transpose the two-dimensional array
Step 2: Row-wise permute the two-dimensional array
Step 3: Transpose the two-dimensional array

Figure 10 illustrates the data movement of the three steps. Again, in this figure, we assume that each $a[i][j]$ is initially storing $(P(i\cdot w+j)/w, P(i\cdot w+j) \bmod w) = (P(i\cdot w+j) \bmod w, j)$. After the transpose in Step 1, $a[j][i]$ stores $(P(i\cdot w+j) \bmod w, j)$. The row-wise permutation is performed such that $a[j][i]$ stores (i, j). Finally, by transposing in Step 3, $a[i][j]$ stores (i, j).

Since column-wise permutation can be done by transposing and row-wise permutation, from Lemma 5.2 and Lemma 7.1, we have,

LEMMA 7.2. *Any column-wise permutation of a two-dimensional array of size $w \times w$ can be done in $O(wl)$ time units using w threads on the UMM with each thread having w local registers.*

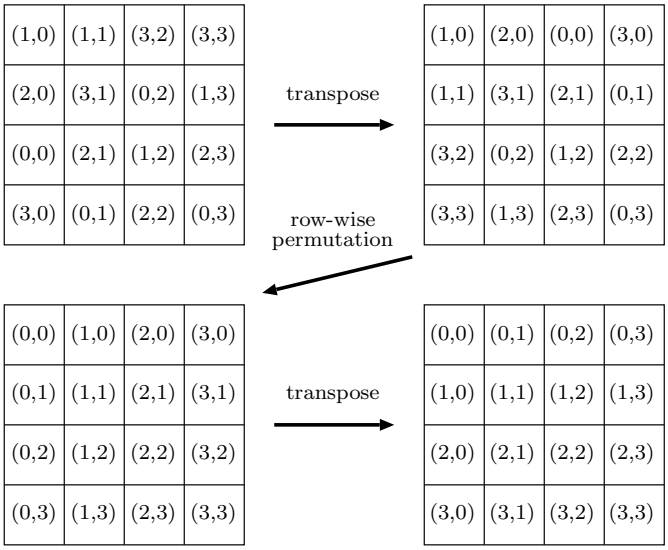

Figure 10. Column-wise permutation

We next show any permutation of a 2-dimensional array of size $w\times w$ can be done in $O(wl)$ time units using w threads on the UMM by the row-wise permutation and the column-wise permutation. For a given permutation P on a 2-dimensional array a, we draw a bipartite graph $G = (U, V, E)$ as follows:

- $U = \{A[0], A[1], A[2], \ldots, A[w-1]\}$ is a set of nodes each of which corresponds to an address group of source.
- $V = \{A[0], A[1], A[2], \ldots, A[w-1]\}$ is a set of nodes each of which corresponds to an address group of destination.
- For each pair source $a[i][j]$ and destination $a[P(i\cdot w+j)/w][P(i\cdot w+j) \bmod w]$, E has a corresponding edge connecting $A[i](\in U)$ and $A[P(i\cdot w+j)/w](\in V)$.

For example if a word of data in $a[1][3]$ is copied to $a[2][4]$ by permutation P, an edge is drawn from node $A[1]$ in U and node $A[2]$ in V. Clearly, G is a regular bipartite graph with degree w. From Theorem 6.1, 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 a 2-dimensional array a of size $w \times w$, we have painted edges in w colors $0, 1, \ldots, w-1$. Since each edge corresponds to a data stored in a, we can think that data is painted by the same color as the corresponding edge. Permutation can be done by three steps as follows:

[Permutation on the UMM]
Step 1: Row-wise permute the 2-dimensional array.
Step 2: Column-wise permute the 2-dimensional array.
Step 3: Row-wise permute the 2-dimensional array.

Let us see how permutation of each step is determined by edge coloring. As before, we assume that $a[i][j]$ is storing $(P(i \cdot w + j)/w, P(i \cdot w + j) \bmod w)$ and show that after the permutation algorithm is executed $a[i][j]$ stores (i, j). The readers should refer to Figure 11 for illustrating the data movement of the permutation algorithm for $w = 4$. From the figure we can confirm the following lemma:

LEMMA 7.3. *Suppose that data stored in a 2-dimensional array of $w \times w$ are painted by w colors using edge coloring of the corresponding bipartite graph above. We have: (1) data in the same row are painted by different colors, and (2) data painted by the same color has different row destination.*

Since nodes in U are connected to w edges painted by different colors, we have (1) above. Also, since w edges painted by the same color connected to different nodes in V, we have (2) above.

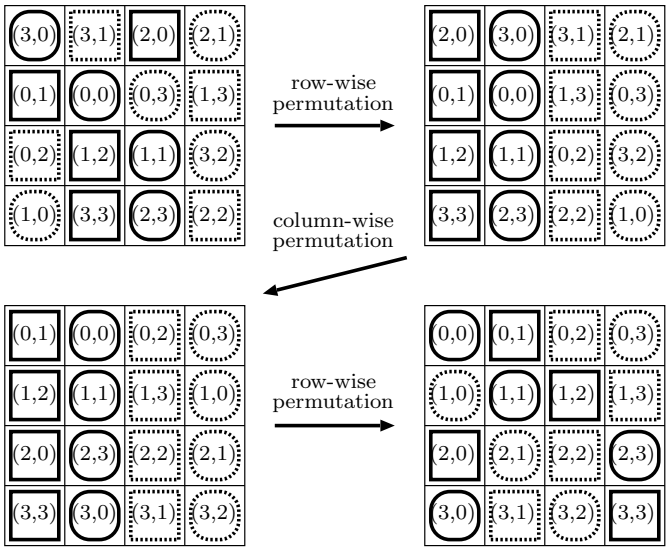

Figure 11. Illustrating a data movement of the permutation algorithm on the UMM

In Step 1, row-wise permutation is performed such that data with color i ($0 \leq i \leq w-1$) are stored in the i-th column. From Lemma 7.3 (1), w data in each row are painted by w colors, Step 1 is possible. Step 2 uses column-wise permutation to move data to the final row destination. From Lemma 7.3 (2), w data in each column has different w row destination, Step 2 is possible. Finally, in Step 3, row-wise permutation is performed to move data to the final column destination.

Since the permutation algorithm on the UMM performs the row-wise permutation and the column-wise permutation, from Lemma 7.1 and Lemma 7.2, we have,

LEMMA 7.4. *Any permutation of an array of size w^2 can be done in $O(wl)$ time units using w threads on the UMM with each thread having local memory of w words.*

We go on to show a 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 2-dimensional array of size $w^2 \times w^2$. We use the permutation algorithm for Lemma 7.4 to perform the row-wise permutation of the 2-dimensional array of size $w^2 \times w^2$. Similarly to the permutation algorithm for Lemma 7.4, we generate a bipartite graph with $G = (U, V, E)$ such that

- $U = \{0, 1, 2, \ldots, w^2 - 1\}$ is a set of nodes each of which corresponds to a row of source.
- $V = \{0, 1, 2, \ldots, w^2 - 1\}$ is a set of nodes each of which corresponds to a row of destination.
- For each pair source $a[i][j]$ and destination $a[P(i \cdot w+j)/w^2][P(i \cdot w+j) \bmod w^2]$, E has a corresponding edge connecting $i(\in U)$ and $P(i \cdot w + j)/w(\in V)$.

Similarly to the permutation algorithm for Lemma 7.4, any permutation of a 2-dimensional array of size $w^2 \times w^2$ can be done in three steps, row-wise permutation, column-wise permutation, and then row-wise permutation. The key idea is to use the permutation algorithm for Lemma 7.4 to perform the row-wise permutation and the column-wise permutation. We will discuss the details of the row-wise permutation and the column-wise permutation of a 2-dimensional array of size $w^2 \times w^2$

We show that the row-wise permutation of a 2-dimensional array of size $w^2 \times w^2$ can be done in $O(w^3 + \frac{w^4 l}{p})$ time units using p threads on the UMM. The p threads are partitioned into $\frac{p}{w}$ warps. First, each of the $\frac{p}{w}$ warps assigned a row of the first $\frac{p}{w}$ rows performs the row-wise permutation of the first $\frac{p}{w}$ row in parallel. This can be done by the permutation algorithm for Lemma 7.4, which runs $O(wl)$ time units. Note that, each of the w threads of a warp requests at most $O(w)$ memory access in the permutation algorithm for Lemma 7.4. The first memory access requests by the p threads in $\frac{p}{w}$ warps are completed $\frac{p}{w} + l$ time units. Since the memory access requests by p threads are repeated $O(w)$ times, the row-wise permutation of the first $\frac{p}{w}$ rows is completed in $O((\frac{p}{w} + l) \cdot w) = O(p + wl)$ time units. Since we have w^2 rows, this operation is repeated $w^2/\frac{p}{w} = \frac{w^3}{p}$ times. Thus, the row-wise permutation can be done in $O((p + wl) \cdot \frac{w^3}{p}) = O(w^3 + \frac{w^4 l}{p})$ time units on the UMM.

Similarly to the row-wise permutation of a 2-dimensional array of size $w \times w$ shown in Figure 10, the column-wise permutation of a 2-dimensional array of size $w^2 \times w^2$ can be done by transpose, row-wise permutation, and transpose. The transpose of a 2-dimensional array of size $w^2 \times w^2$ can be done in $O(w^3 + \frac{w^4 l}{p})$ time units on the UMM from Lemma 5.2. Also, the row-wise permutation can be done in $O(w^3 + \frac{w^4 l}{p})$ time units. Thus, the column-wise permutation can be done in $O(w^3 + \frac{w^4 l}{p})$ time units.

We are now in a position to show our permutation algorithm for a 2-dimensional array of size $w^2 \times w^2$. Similarly to permutation of a 2-dimensional array of size $w \times w$, permutation of a 2-dimensional array of size $w^2 \times w^2$ can be done in three steps, row-wise permutation, column-wise permutation and row-wise permutation. Since each step can be done in $O(w^3 + \frac{w^4 l}{p})$ time on the UMM, any permutation of a 2-dimensional array of size $w^2 \times w^2$ can be done in $O(w^3 + \frac{w^4 l}{p})$ time units on the UMM.

We can use the same technique for a permutation of an array of size $w^4 \times w^4$. The readers should have no difficulty to confirm that any permutation can be done

in $O(w^7 + \frac{w^8 l}{p})$ time units on the UMM using p threads.

Repeating the same technique, we can obtain a permutation algorithm for an array of size $n = w^c \times w^c$. Permutation of a 2-dimensional array of size $w^c \times w^c$ can be done by executing the row-wise permutation recursively three times and the transpose for an array of size $w^{c/2} \times w^{c/2}$ twice. If the size n of an array satisfies $n \leq w^{O(1)}$, that is, $c = O(1)$, then the depth of the recursion is constant. If this is the case, the computing time is $O(w^{2c-1} + \frac{m^{2c} l}{p}) = O(\frac{n}{w} + \frac{nl}{p})$. Thus, we have,

LEMMA 7.5. *Any permutation of an array of size n can be done in $O(\frac{n}{w} + \frac{nl}{p})$ time units ($w \leq p \leq \frac{n}{w}$) on the UMM with each thread having w local registers provided that $n \leq w^{O(1)}$.*

Finally, if each register has only r ($\leq w$) local registers, we can use the transpose algorithm for Lemma 5.3. If this is the case, we have,

THEOREM 7.6. *Any permutation of an array of size n can be done in $O((\frac{n}{w} + \frac{nl}{p}) \cdot \sqrt{\frac{w}{r}})$ time units ($w \leq p \leq \frac{n}{r}$) on the UMM with each thread having r ($r \leq w$) local registers provided that $n \leq w^{O(1)}$.*

8. 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 2-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 $\frac{n}{w}$ and the latency limitation $\frac{nl}{p}$, 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 threads are connected to a small memory of DMM and a large memory of 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] K. Nishida, Y. Ito, and K. 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)*, Sept., 2012, pp. 1–15.

[11] NVIDIA Corporation, *NVIDIA CUDA C programming guide version 4.0* (2011).

[12] 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 1 (2011), pp. 260–276.

[13] NVIDIA Corporation, *NVIDIA CUDA C best practice guide version 3.1* (2010).

[14] F.T. Leighton, *Introduction to Parallel Algorithms and Architectures: Arrays, Trees, Hypercubes*, Morgan Kaufmann, 1991.

[15] R.H. Bisseling, *Parallel Scientific Computation: A Structured Approach using BSP and MPI*, Oxford University Press, 2004.

[16] D. Culler, R. Karp, D. Patterson, A. Sahay, K.E. Schauser, E. Santos, R. Subramonian, and T. Eickenvon , *LogP: towards a realistic model of parallel computation*, in *Proceedings of the fourth ACM SIGPLAN symposium on Principles and practice of parallel programming*, 1993, pp. 1–12.

[17] R. Vaidyanathan and J.L. Trahan, *Dynamic Reconfiguration: Architectures and Algorithms*, Kluwer Academic/Plenum Publishers, 2004.

[18] M.J. Flynn, *Some computer organizations and their effectiveness*, IEEE Transactions on Computers C-21 (1972), pp. 948–960.

[19] G. Ruetsch and P. Micikevicius, *Optimizing matrix transpose in CUDA* (2009).

[20] 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.

[21] S. Ryoo, C.I. Rodrigues, S.S. Baghsorkhi, S.S. Stone, D.B. Kirk, and W. W. Hwumei , *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.

[22] D.T. Wang, *Modern dram memory systems:performance analysis and a high performance, power-constrained dram scheduling algorithm*, Ph.D. thesis, University of Maryland, USA, 2005.

[23] Xilinx Inc., *Virtex-5 FPGA users guide* (2009).

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

[25] A. Gottlieb, R. Grishman, C.P. Kruskal, K.P. McAuliffe, L. Rudolph, and M. Snir, *The nyu ultracomputer – designing an MIMD shared memory parallel*

computer, IEEE Trans. on Computers C-32 (1983), pp. 175 – 189.
[26] S.G. Akl, *Parallel Sorting Algorithms*, Academic Press, 1985.
[27] K.E. Batcher, *Sorting networks and their applications*, in *Proc. AFIPS Spring Joint Comput. Conf.*, Vol. 32, 1968, pp. 307–314.
[28] K. Nakano, *Optimal sorting algorithms on bus-connected processor arrays*, IEICE Trans. Fundamentals E76-A (1993), pp. 2008–2015.
[29] R.J. Wilson, *Introduction to Graph Theory, 3rd edition*, Longman, 1985.