Memory Access Optimal Algorithms for the GPU
(GPU 用のメモリアクセスの最適なアルゴリズム)

by

Akihiko Kasagi

A dissertation submitted

in partial fulfillmen of the requirements for the degree of

Doctor of Engineering

in Information Engineering

Under Supervision of

Professor Koji Nakano

Department of Information Engineering,

Graduate School of Engineering,

Hiroshima University

March, 2015

SUMMARY

A GPU (Graphics Processing Unit) is a specialized circuit designed to accelerate computation for building and manipulating images. 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. The GPU has two types of memories: the shared memory and the global memory. The performance of applications using a GPU depends greatly on the usage of these memories. Because of the above background, we consider memory access optimal algorithms on a single GPU. This dissertation shows theoretical memory machine models and memory access optimal algorithms which are as follows:

We introduce simple parallel memory machine models that capture the essential features of NVIDIA GPUs. The Discrete Memory Machine (DMM) and the Unifie Memory Machine (UMM) reflec the essential features of the shared memory and the global memory of NVIDIA GPUs. In both architectures, a sea of threads are connected to the memory banks (MBs) through the memory management unit (MMU). Each thread is a Random Access Machine (RAM), which can execute fundamental operations in a time unit. Threads are executed in SIMD fashion, and the processors run on the same program and work on the different data. The Hierarchical Memory Machine (HMM) is a hybrid of the DMM and the UMM. The HMM is a more practical parallel computing model that reflect the hierarchical architecture of CUDA-enabled GPUs.

Offline permutation is a task to move data along a permutation P given beforehand. The conventional algorithm of offline permutation performs $b[P(i)]$ ′ $a[i]$ for all i ($0 \geq i \geq n$ 1). We present conflict-fre offline permutation algorithm on the DMM and implement it to run on the shared memory in the GPU. Our idea is to use two

permutations S and D which can be obtained from P. Using these two permutations our conflict-fre permutation algorithm performs $b[D(i)] \leftarrow a[S(i)]$ for all i. In the experimental results, our conflict-fre permutation algorithm runs in 167ns for any permutation including the random permutation and the worst permutation. We also present optimal offline permutation algorithm on the HMM. This algorithm has no stride access requests and no bank-conflicts We evaluate these algorithms using several parameters and implement these algorithms to the GPU. The experimental results show that our optimal offline permutation algorithm on the HMM is faster than the conventional permutation algorithm for most cases.

The summed area table (SAT) of a matrix is a data structure frequently used in the area of computer vision which can be obtained by computing the column-wise prefix-sum and then the row-wise prefix-sums The previously published best algorithm (2R1W SAT algorithm) performs 2 read operations and 1 write operation per element. We present a more efficient algorithm (1R1W SAT algorithm) which performs 1 read operation and 1 write operation per element. Clearly, since every element in a matrix must be read at least once, and all resulting values must be written, our 1R1W SAT algorithm is optimal in terms of the global memory access. We also show a combined algorithm ((1+r)R1W SAT algorithm) of 2R1W and 1R1W SAT algorithms that may have better performance. The experimental results show that our (1+r)R1W SAT algorithm runs faster than any other SAT algorithms for large input matrices. Also, it runs more than 100 times faster than the best SAT algorithm using a single CPU.

Contents

List of Figures

Chapter 1

Introduction

1.1 Background and Motivation

The Graphics Processing Unit (GPU) is a specialized circuit designed to accelerate computation for building and manipulating images [13, 18, 35]. GPU consists of thousands of processing cores designed for handling multiple tasks simultaneously. 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 [13, 8, 32]. NVIDIA provides a parallel computing architecture called CUDA (Compute Unifie Device Architecture) [5], 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 [19, 14], since they have hundreds of processor cores and very high memory bandwidth. Since the GPU also has a high energy efficiency which denotes performance per watt, the GPU attracts notice as a computational accelerator for high performance computing (HPC).

NVIDIA GPUs can perform a data communication between the CPU and the GPU through a PCI express. Then, the data from the main memory in the CPU is stored into the global memory, off-chip DRAM in the GPU. However, the PCI express has low bandwidth, say, 2–16 Gbytes/sec. On the other hand, the global memory has high bandwidth, say, 180–336 Gbytes/sec. The bandwidth of the global memory is much higher than the bandwidth of the PCI express. Developers should avoid the data communication between CPU and GPU as possible [15]. GPU architecture has two types of memories: the shared memory and the global memory. The performance of GPU applications depends greatly on the usage of these memories. Especially, to maximize the memory bandwidth of the global memory is a key to accelerate the GPU application. Because of the above background, we treat independent jobs running on a single GPU and accelerate algorithms in terms of memory access. For this purpose, this dissertation shows theoretical memory machine models and memory access optimal algorithms which are as follows:

Parallel Memory Machine Models

In this chapter, we introduce simple parallel memory machine models that capture the essential features of NVIDIA GPUs. *the Discrete Memory Machine (DMM)* and *the Unifie Memory Machine (UMM)*, which reflec the essential features of the shared memory and the global memory of NVIDIA GPUs. In both architectures, a sea of threads are connected to the memory banks (MBs) through the memory management unit (MMU). Each thread is a Random Access Machine (RAM), which can execute fundamental operations in a time unit. Threads are executed in SIMD fashion, and the processors run on the same program and work on the different data. The HMM is a more practical parallel computing model that reflect the hierarchical architecture of CUDA-enabled GPUs. The asynchronous Hierarchical Memory Machine is more realistic model for GPUs In the asynchronous HMM. In this dissertation, we evaluate several algorithms using these parallel memory machine models.

Offline Permutation Algorithms on the Discrete Memory Machine

Offline permutation is a task to move data along a permutation P given beforehand. It is known that the computation of FFT can be done by a multistage network in which each stage involves permutation. Sorting network such as bitonic sorting also involves permutation in each stage. The conventional algorithm of offline permutation performs $b[P(i)] \leftarrow a[i]$ for all i $(0 \geq i \geq n \quad 1)$. In this chapter, we present conflict-fre offline permutation algorithm on the DMM and implement it to run on the shared memory in the GPU. Our idea is to use two permutations S and D which can be obtained from P. Using these two permutations our conflict-fre permutation algorithm performs $b[D(i)] \leftarrow a[S(i)]$ for all i. These two permutations S and D can be determined using a graph theoretic result about bipartite graph coloring. We evaluate the several permutation algorithms and compare these performances of read/write access. Experimental results for 1024 double (64-bit) numbers on NVIDIA GeForce GTX-680 show that the straightforward permutation algorithm takes 247.8ns for the random permutation and 1684ns for the worst permutation that involves the maximum bank conflicts Our conflict-fre permutation algorithm runs in 167ns for any permutation including the random permutation and worst permutation. It follows that our conflict-fre permutation is 1.48 times faster than random permutation and 10.0 times faster for the worst permutation.

An Optimal Offline Permutation Algorithm on the Hierarchical Memory Machine

In this chapter, we present optimal offline permutation algorithm on the HMM. This permutation algorithm uses conflict-fre offline permutation algorithm in chapter 4. Our scheduled offline permutation algorithm on the HMM performs three step permutations, row-wise permutation, column-wise permutation, and row-wise permutation, each of which is performed in DMMs of the HMM in parallel. These three permutation can be determined using a graph theoretic result about bipartite graph coloring. This algorithm has no stride access requests on the HMM and no bank-conflict in DMMs. We also present an offline permutation algorithm for any permutation running in $16\frac{n}{w} + 16\frac{n}{kw} + 16L \quad 16$ time units on the HMM with k DMMs. Quite surprisingly, our offline permutation algorithm on the GPU achieves better performance that the conventional algorithm in random permutation, although the running time has a large constant factor. We can say that the experimental results provide a good example of GPU computation showing that a complicated but ingenious implementation with a larger constant factor in computing time can outperform a much simpler conventional algorithm.

Parallel Algorithms for the Summed Area Table on the Asynchronous Hierarchical Memory Machine

The summed area table (SAT) of a matrix is a data structure frequently used in the area of computer vision which can be obtained by computing the column-wise prefix sums and then the row-wise prefix-sums A straightforward algorithm (2R2W SAT algorithm), which computes the column-wise prefix-sum and then the row-wise prefix sums, performs 2 read operations and 2 write operations per element of a matrix. The best known algorithm (2R1W SAT algorithm) so far performs 2 read operations and 1 write operation per element [28]. We present a more efficient algorithm (1R1W SAT algorithm) which performs 1 read operation and 1 write operation per element. Clearly, since every element in a matrix must be read at least once, and all resulting values must be written, our 1R1W SAT algorithm is optimal in terms of the global memory access. We show a combined algorithm (1+r)R1W SAT algorithm of 2R1W and 1R1W SAT algorithms that may have better performance. We have also implemented these algorithms on GeForce GTX 780 Ti. Our best algorithm, $(1 + r)$R1W SAT algorithm, runs faster than any other algorithms for large input. It also runs at least 100 times faster than the sequential algorithm running on a single CPU.

1.2 Dissertation Organization

This doctoral dissertation is organized as follows: The background with motivation and the introduction of this dissertation are presented in Chapter 1. In Chapter 2, we briefl introduce NVIDIA GPUs and CUDA programming model. Chapter 3 describes simple parallel memory machine models that capture the essential features of NVIDIA GPUs. Chapter 4 describes an offline permutation algorithm on DMMs. Chapter 5 describes an offline permutation algorithm on the HMM. In Chapter 6, we propose 1R1W SAT algorithm and (1+r)R1W SAT algorithm which are optimal in terms of the global memory access. Finally, this dissertation is concluded in Chapter 7.

Chapter 2

GPU and CUDA

In this chapter, we briefl introduce NVIDIA GPUs and CUDA programming model. NVIDIA GPUs have streaming multiprocessors (SMs) each of which executes multiple threads in parallel. For example, a multithreaded program is partitioned into each SM that execute independently from each other, so that a GPU with more multiprocessors will automatically. In 2006, NVIDIA introduced CUDA, a general purpose parallel computing platform and programming model that leverages the parallel compute engine in NVIDIA GPUs to solve many complex computational problems in a more efficient way than on a CPU. CUDA comes with a software environment that allows developers to use C as a high-level programming language. CUDA uses two types of memories of the NVIDIA GPUs: the shared memory and the global memory [5]. As shown in Figure 2.1, each SM has the shared memory, an extremely fast on-chip memory with lower capacity, say, 16–48 Kbytes, and low latency. Every SM shares the global memory implemented as an off-chip DRAM, and has large capacity, say, 1.5–6 Gbytes, but its access latency is high. 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 conflic* of the shared memory access and *the coalescing* of the global memory access [18, 19, 4, 29, 30].

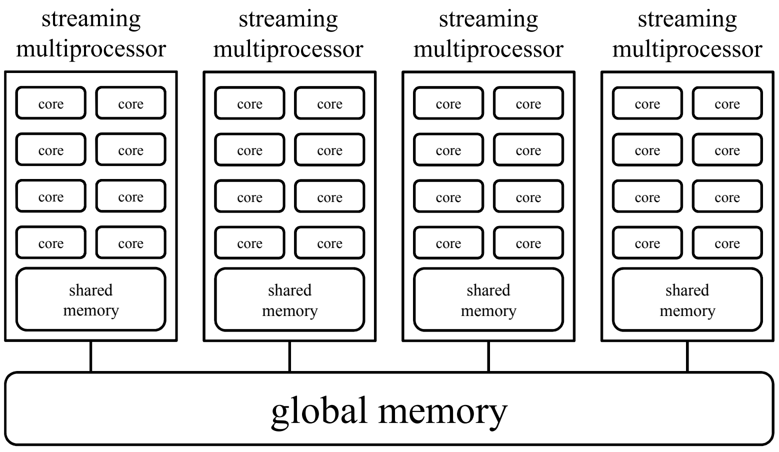

Figure 2.1: High-level GPU architecture

2.1 CUDA programming model

CUDA C extends C by allowing the programmer to defin C function called *kernel*. CUDA parallel programming model has a hierarchy of thread groups called *grid*, *block* and *thread*. A single grid is organized by multiple blocks, each of which has equal number of threads as illustrated Figure 2.2. When a kernel function is called by the host CPU, CUDA generates hierarchy of thread groups that are define by kernel call. Each thread that executes the kernel is given a unique thread ID that is accessible within the kernel through the built-in **threadIdx** variable. Each block is also given a unique block ID through the built-in **blockIdx** variable. These variable provides a way to invoke computation across the elements in a domain such as a vector, matrix. CUDA blocks are allocated to streaming processors such that all threads in a block are executed by the same streaming processor in parallel. All threads can access to the global memory. However, threads in a block can access to the shared memory of the streaming processor to which the block is allocated. Since blocks are arranged to multiple streaming processors, threads in different blocks cannot share data in the shared memories. In the execution, 32 threads in a block are split into groups of thread called *warps*. Each of these warps contains the same number of threads and is executed independently. When a warp is selected for execution, all threads in the warp execute the same instruction concurrently.

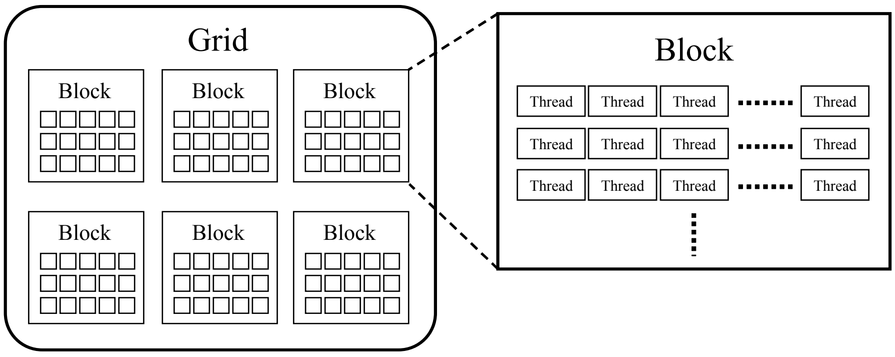

Figure 2.2: Hierarchy of thread groups

Coalescing

Since the global memory in a GPU has very high latency, efficient global memory access is important to improve the performance of GPU applications. To maximize a bandwidth of the global memory, we need to consider the coalescing of the global memory access. If memory access requests in a warp are consecutive address, this access is called coalesced access and maximize the memory bandwidth. On the other hand, if memory access requests in a warp are not consecutive address, called stride accesses has large overhead to process the memory requests. Figure 2.3 shows two patterns of memory requests.

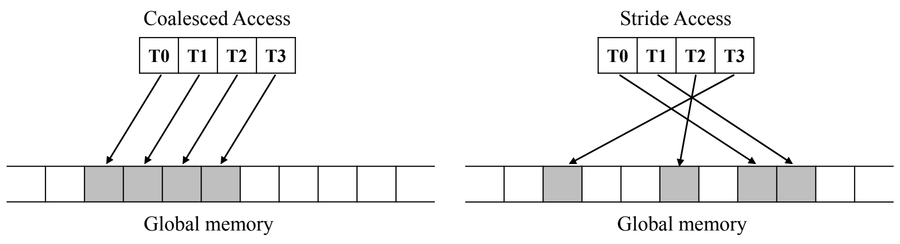

Figure 2.3: Coalesced access and stride access to the global memory

Bank-Conflic

The address space of the shared memory is mapped into several physical memory banks. Successive 32-bit words are assigned to successive banks. Each bank can only address one dataset at a time. If two or more threads in a warp access the same memory banks at the same time, the access requests have to be serialized, called a bank-conflict Hence, to maximize the memory access performance, threads of CUDA should access distinct memory banks to avoid the bank conflict of memory access. For example in Figure 2.4, memory access requests of *m*[0], *m*[5], *m*[10] and *m*[11] has no bank conflict On the other hand, memory access requests of *m*[0], *m*[4], *m*[9] and *m*[15] involve the bank conflic because *m*[0] and *m*[4] are stored in the same bank.

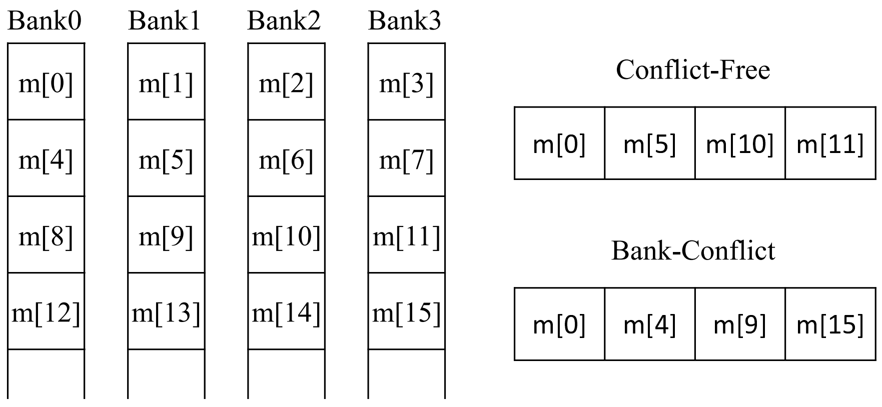

Figure 2.4: Conflict-fre access and bank conflic in the shared memory

Chapter 3

Parallel Memory Machine Models

The firs contribution of this chapter is to introduce simple parallel memory machine models that capture the essential features of the bank conflic of the shared memory access and the coalescing of the global memory access. More specificall , we present two models, *the Discrete Memory Machine (DMM)* and *the Unifie Memory Machine (UMM)* [24, 27], which reflec the essential features of the shared memory and the global memory of NVIDIA GPUs.

The outline of the architectures of the DMM and the UMM are illustrated in Figure 3.1. 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 chapter 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 [9] fashion, and the processors run on the same program and work on the different data.

MBs constitute a single address space of the memory. A single address space of the memory is mapped to the MBs in an interleaved way such that the word of data of address i is stored in the (i mod w)-th bank, where w is the number of MBs. The main difference of the two architectures is the connection of the address line between the MMU and the MBs, which can transfer an address value. In the DMM, the address lines connect the MBs and the MMU separately, while a single address line from the MMU is connected to the MBs in the UMM. Hence, in the UMM, the same address value is broadcast to every MB, and the same address of the MBs can be accessed in each time unit. On the other hand, different addresses of the MBs can be accessed in the DMM. The DMM and the UMM capture the essence of the shared memory access and the global memory access of current GPUs.

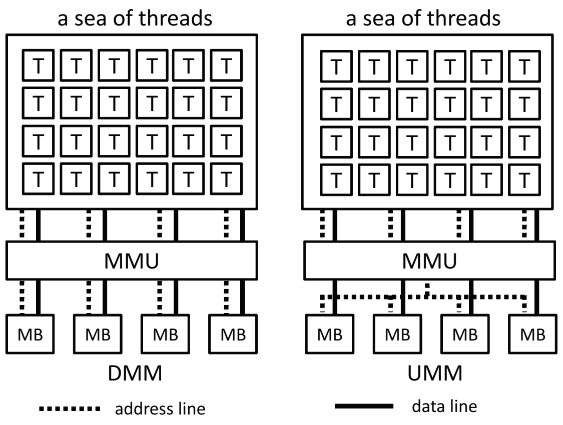

Figure 3.1: The architectures of the DMM and the UMM

3.1 The Discrete Memory Machine

The main contribution of this sections is to show *the Discrete Memory Machine (DMM)*. We defin the Discrete Memory Machine (DMM) of width w and latency l. Let $m[i]$ ($0 \geq i$) denote a memory cell of address i in the memory. Let $B[j] = \}j, j+w, j+2w, j+3w, \ldots\langle$

$(0 \geq j \geq w \quad 1)$ be a set of address of *the j-th memory bank* of the memory. In other words, address i is in the (i mod w)-th memory bank. Clearly, a memory cell $m[i]$ is in the (i mod w)-th memory bank. Figure 3.2 illustrates memory banks of DMM for w = 4. 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.

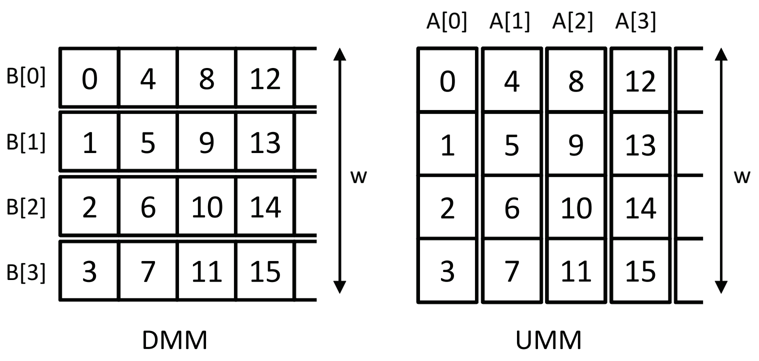

Figure 3.2: DMM and UMM with w = 4

Also, we assume that addresses in different banks can be accessed in a time unit, but no two addresses 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. Thus, it takes $k + l \quad 1$ time units to complete memory access requests to k addresses in a particular bank.

We assume that p threads are partitioned into $\frac{p}{w}$ groups of w threads called *warps*. More specificall , p threads $T(0)$, $T(1)$, ..., $T(p \quad 1)$ are partitioned into $\frac{p}{w}$ warps $W(0), W(1), \ldots, W(\frac{p}{w} \quad 1)$ such that $W(i) = \}T(i \times w), T(i \times w + 1), \ldots, T((i + 1) \times w \quad 1)\langle$ $(0 \geq i \geq \frac{p}{w} \quad 1)$. Warps are dispatched for memory access in turn, and w threads in a warp try to access the memory at the same time. In other words, $W(0), W(1), \ldots, W(\frac{p}{w} \quad 1)$ are dispatched in a round-robin manner if at least one thread in a warp requests memory access. If no thread in a warp needs memory access, such warp is not dispatched for memory access. When $W(i)$ is dispatched, w threads in $W(i)$ send memory access requests, at most one request per thread, to the memory. We also assume that a thread cannot send a new memory access request until the previous memory access request is completed. Hence, if a thread sends a memory access request, it must wait at least l time units to send a new memory access request.

3.2 The Unifie Memory Machine

The main contribution of this sections is to show *the Unifie Memory Machine (UMM)*. We defin the Unifie Memory Machine (UMM) of width w and latency l as follows. Let $A[j] = \} j \times w, j \times w + 1, \ldots, (j+1) \times w \quad 1 \langle$ denote a set of addresses in *the j-th address group*. We assume that addresses in the same address group are processed at the same time. However, if they are in the different groups, one time unit is necessary for each of the groups. Also, similarly to the DMM, p threads are partitioned into warps and each warp accesses the memory in turn.

Figure 3.3 shows examples of memory access on the DMM and the UMM. We assume that each memory access request is completed when it reaches the last pipeline stage. Two warps $W(0)$ and $W(1)$ access to $\rangle 7, 5, 15, 0|$ and $\rangle 10, 11, 12, 9|$, respectively. In the DMM, memory access requests by $W(0)$ are separated into two pipeline stages, because addresses 7 and 15 are in the same bank $B(3)$. Those by $W(1)$ occupies 1 stage, because all requests are in distinct banks. Thus, the memory requests occupy three stages, it takes $3 + 5 \quad 1 = 7$ time units to complete the memory access. In the UMM, memory access requests by $W(0)$ are destined for three address groups. Hence the memory requests occupy three stages. Similarly those by $W(1)$ occupy two stages.

Hence, it takes 5 + 5   1 = 9 time units to complete the memory access.

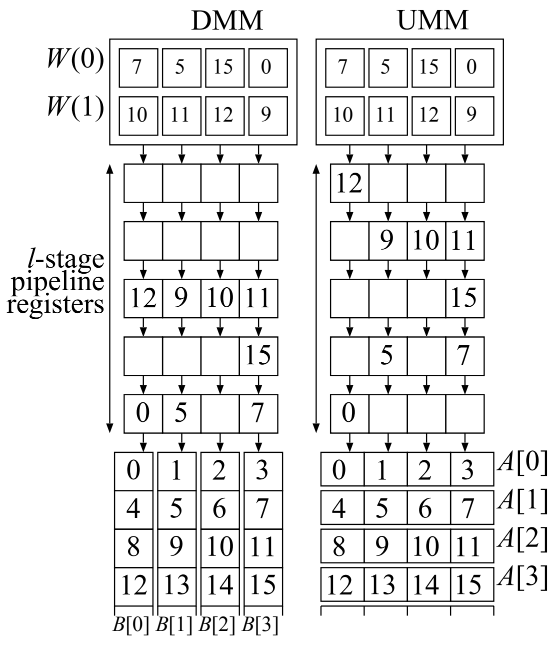

Figure 3.3: Examples of memory access on the DMM and the UMM

3.3 The Hierarchical Memory Machine

The main contribution of this sections is to show *the Hierarchical Memory Machine (HMM)* [25]. The HMM consists of d DMMs and a single UMM as illustrated in Figure 3.4. Each DMM has w memory banks and the UMM also has w memory banks. We call the memory banks of each DMM *the shared memory* and those of the UMM *the global memory*. Each DMM works independently. Threads are partitioned into warps of w threads, and each warp are dispatched for the memory access for the shared memory in turn. Further, each warp of w threads in all DMMs can send memory access requests to the global memory. Figure 3.4 illustrates the architecture of the HMM with $d = 2$ DMMs. Each DMM and the UMM have $w = 4$ memory banks. The shared memory of each DMM and the global memory of the UMM correspond to “the shared memory” of each streaming multiprocessor and “the global memory” of GPUs. We also assume that the shared memory in each DMM of the HMM can store up to $O(w^2)$ numbers. The capacity of the shared memory of latest CUDA-enabled GPUs is up to 48KBytes and the number w of the banks is 32 [5]. Since an array of 32^2 double (64-bit) numbers occupy 8KBytes, each shared memory can store at most 6 such matrices. Thus, it is reasonable to assume that DMM can store $O(w^2)$ numbers in the shared memory.

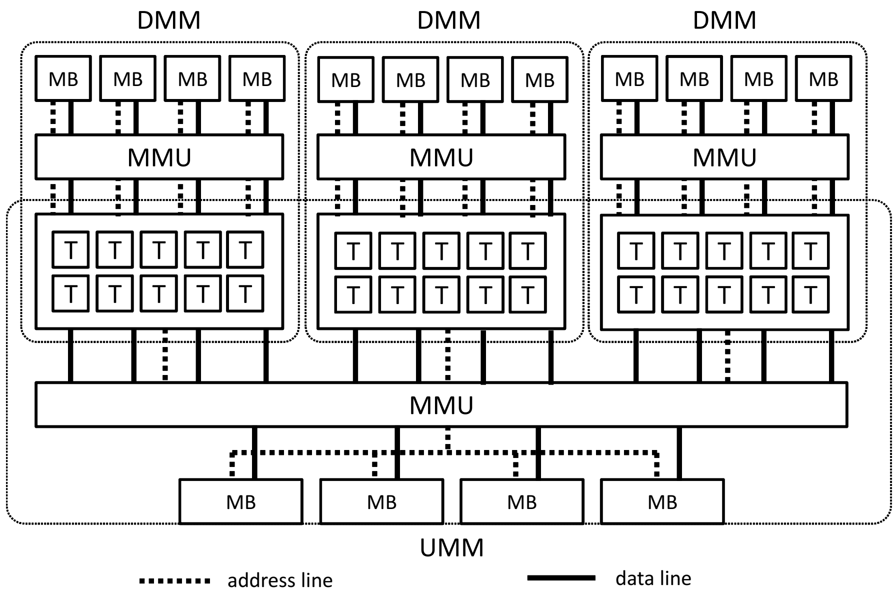

Figure 3.4: The architecture of the HMM with d = 3 DMMs and width w = 4

3.4 The Asynchronous Hierarchical Memory Machine

The main contribution of this section is to show *the asynchronous Hierarchical Memory Machine*. For more realistic model for GPUs, we introduce the asynchronous Hierarchical Memory Machine (asynchronous HMM). In the asynchronous HMM, DMMs work asynchronously in the sense that some DMMs may work slightly slower or faster than the others. Instead, all threads in all DMMs can execute a barrier synchronization instruction. If a thread in a DMM executes the barrier synchronization instruction, it must wait until all the other threads in all DMMs execute it [22]. Also, after all threads execute the barrier synchronization instruction, all DMMs are *reset*, that is, the shared memory of all DMMs are initialized and all data stored in it are lost. The reader may think that this reset assumption of all DMMs is not reasonable. However, this assumption is mandatory for program scalability of DMMs in the HMM. More specificall , suppose that a programmer writes a program of the HMM with d DMMs. It may be possible to execute this program in the HMM with $d^{\in}$ DMMs such that $d^{\in} < d$. If this is the case, the program of $d^{\in}$ DMMs are executed until a barrier synchronization instruction is executed. After that, the $d^{\in}$ DMMs are reset and the program of next $d^{\in}$ DMMs are executed. The same procedure is repeated until all threads execute the barrier synchronization instruction. Hence, it makes sense to assume that all DMMs are reset after each barrier synchronization step. The previous DMMs are responsible for copying the data stored in the shared memory to the global memory before barrier synchronization if they are used after the synchronization. Actually, we need to terminate a CUDA kernel call for a GPU when barrier synchronization of all threads is necessary [5]. When a CUDA kernel call is terminated for barrier synchronization, the data stored in the shared memory by a CUDA block are lost. This is because CUDA blocks are executed in streaming multiprocessors with small shared memory one by one in turn.

Chapter 4

Offline Permutation Algorithms on the Discrete Memory Machine

Offline permutation is a task to move data along a permutation given beforehand. Accelerating offline permutation is very important, because it has many applications. For example, matrix transpose, which is one of the important permutations, is frequently used in matrix computation. It is known that the computation of FFT can be done by a multistage network in which each stage involves permutation [31]. Sorting network such as bitonic sorting [10, 3, 12] also involves permutation in each stage. Further, communication on processor networks such as hypercubes, meshes, and so on can be emulated by permutation on the shared memory. Thus, parallel algorithms on processor networks can be simulated on the shared memory machine by data permutations.

The main contribution of this chapter is to present a conflict-fre offline permutation algorithm on the DMM and implement it to run on the shared memory in the GPU. Suppose that we have two arrays a and b of size n each. Let P be a permutation of $(0, 1, \ldots, n \quad 1)$. In other words, $P(0), P(1), \ldots, P(n \quad 1)$ take distinct integer values in the range $[0, n \quad 1]$. Offline permutation along P is a task to copy $a[i]$ to $b[P(i)]$ for all i $(0 \geq i \geq n \quad 1)$. The destination-designated (D-designated) algorithm just performs $b[P(i)]$ ′ $a[i]$ for all i. However, writing operation in array b may involve bank conflicts Our idea is to use two permutations S and D which can be obtained from P. Using these two permutations our conflict-fre permutation algorithm performs $b[D(i)]$ ′ $a[S(i)]$ for all i. Two permutations S and D are determined so that memory access operations to arrays a and b have no bank conflict Two permutations S and D can be determined using a graph theoretic result about bipartite graph coloring. This idea is originally shown in [24]. Our main contribution is to actually implement permutation algorithms including the D-designated and our conflict-fre permutation algorithms on the shared memory of the latest GPU, NVIDIA GeForce GTX-680.

The experimental results for 1024 double (64-bit) numbers on NVIDIA GeForce GTX-680 show that the straightforward permutation algorithm takes 247.8ns for the random permutation and 1684ns for the worst permutation that involves the maximum bank conflicts Our conflict-fre permutation algorithm runs in 167ns for any permutation including the random permutation and the worst permutation, although it performs more memory accesses. It follows that our conflict-fre permutation is 1.48 times faster for the random permutation and 10.0 times faster for the worst permutation. Further, we show a conflict-fre in-place permutation method that computes S and D in place. Quite surprisingly, for the transpose, the shuffle, and the bit reversal permutations, it runs in 105.4-109.0ns. Since the simple copy operation of two arrays takes 102.8ns, our conflict-fre in-place permutation method has very small overhead for permutation. We also present the experimental results for 1024 floa (32-bit) numbers.

This chapter is organized as follows. In Section 4.1, we defin off-line permutation and show straightforward algorithms. Section 4.2 shows our conflict-fre permutation algorithm and Section 4.3 describes the details of the implementation. In Section 4.4, we defin several important permutations used for our experiment, and present an in-place permutation method. In Section 4.5, experimental results using GeForce GTX-680 are shown. Section 6.7 concludes our work.

4.1 Offline Permutation and Conventional Algorithms

The main purpose of this section is to defin offline permutation and show conventional algorithms for this task.

Suppose that we have two arrays a and b of size n each. Let P be a permutation of $(0, 1, \ldots, n \quad 1)$. In other words, $P(0), P(1), \ldots, P(n \quad 1)$ take distinct integer values in the range $[0, n \quad 1]$. Offline permutation along P is a task to copy $a[i]$ to $b[P(i)]$ for all i $(0 \geq i \geq n \quad 1)$ as shown in Figure 4.1.

P = **(13, 10, 7, 4, 2, 0, 6, 3, 8, 1, 9, 15, 11, 5, 12, 14)**

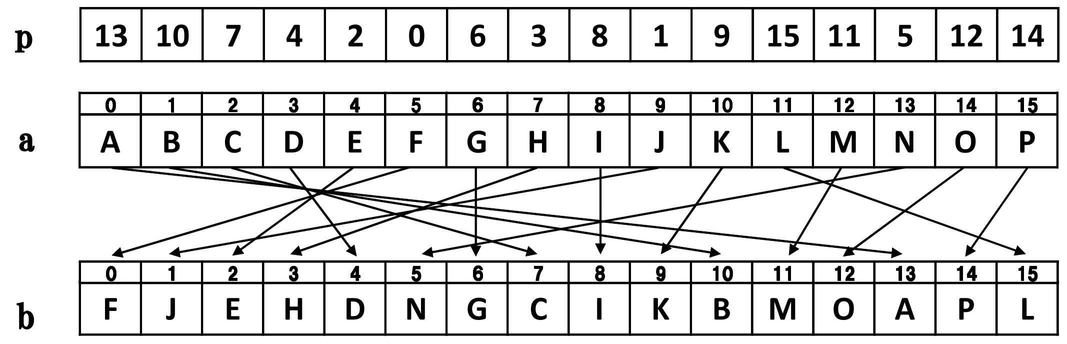

Figure 4.1: Conventional permutation algorithm

Suppose that we have n threads for the task of offline permutation. We assume that $P(0), P(1), \ldots, P(n \quad 1)$ are stored in an array p of size n, such that $p[i] = P(i)$ for all

i ($0 \geq i \geq n \quad 1$). Let $T(i)$ ($0 \geq i \geq n \quad 1$) denote a thread. The following algorithm, Destination-designated permutation algorithm, performs the offline permutation along P.

[Destination-designated permutation algorithm]

```
for i ′  0 to n   1 do
  T(i) performs b[p[i]] ′  a[i]
```

Clearly, reading operations from arrays a and p have no bank conflict However, writing operation in array b may have bank conflict

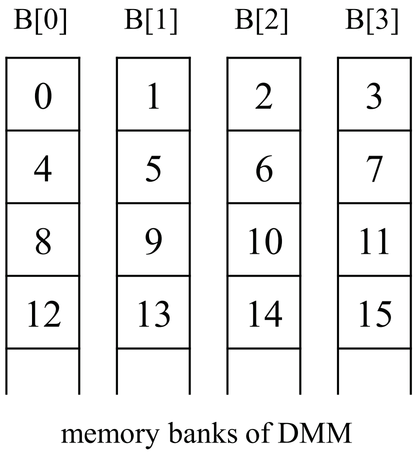

Figure 4.2: Memory banks for $w = 4$

For example, if $P = (0, 4, 8, 12, 1, 5, 9, 13, 2, 6, 10, 14, 3, 7, 11, 15)$ and $w = 4$, then the firs warp $W(0)$ performs writing operation to $b[0], b[4], b[8]$, and $b[12]$, which are in the same bank $B[0]$ (Figure 4.2). Hence, writing operations by $W(0)$ have bank conflict

We can avoid writing bank conflic if we use the Source-designated permutation Q. Let $P^{\ 1}$ be the inverse of P, that is, $P^{\ 1}(P(i)) = i$ for all i ($0 \geq i \geq n \quad 1$). We assume that $P\ ^{1}(0), P\ ^{1}(1), \ldots, P\ ^{1}(n \quad 1)$ are stored in an array q of size n, such that each $q[i]$ stores $P\ ^{1}(i)$. The following algorithm performs the offline permutation along P.

[Source-designated permutation algorithm]

```
for $i$ ′  0 to $n$  1 do
  $T(i)$ performs $b[i]$ ′  $a[q[i]]$
```

Let us show that this algorithm performs the offline permutation along P correctly. The goal of the permutation along P is to satisfy $b[P(i)] = a[i]$ for all i $(0 \geq i \geq n \quad 1)$. Hence, it is sufficient to satisfy $b[P\ ^{1}(P(i))] = a[Q(i)]$ for all i $(0 \geq i \geq n \quad 1)$. From $P\ ^{1}(P(i)) = i$, it is also sufficient to satisfy $b[i] = a[P\ ^{1}(i)]$. Thus, the Source-designated permutation algorithm performs the offline permutation along P correctly.

It should be clear that writing operations in b and reading operations from q have no bank conflict However, reading operations from a may have bank conflict For example, for P define above, we have $P = P\ ^{1}$. Hence, reading operations have always bank conflicts

We will show that, bank conflict-fre permutation is possible if we use two arrays s and d determined from P appropriately. Let S and D be permutations over $(0, 1, \ldots, n$ $1)$. Suppose that $S\ ^{1}(D(i)) = P(i)$ for all i $(0 \geq i \geq n \quad 1)$, where $S\ ^{1}$ denotes the inverse of S. Let s and d be arrays of size n storing the values of S and D respectively. The following algorithm performs permutation along P:

[Conflict-f ee permutation algorithm]

```
for $i$ ′  0 to $n$  1 do
  $T(i)$ performs $b[d[i]]$ ′  $a[s[i]]$
```

Let us see the correctness of the algorithm. When the algorithm terminates, $b[D(i)]$ is storing $a[S(i)]$ for all i $(0 \geq i \geq n \quad 1)$. In other words, $b[S\ ^{1}(D(i))]$ is storing

$a[S^{\ 1}(S(i))]$ for all i. Thus, $b[P(i)] = a[i]$ is satisfie and permutation along P is performed correctly.

Clearly, reading operations for array s and d are conflict-free However, access to arrays a and b may have bank conflicts If we defin S and D appropriately, access to arrays s and d can be conflict-free Let P be a permutation define above. We defin S and D as follows: S=(0, 5, 10, 15, 1, 6, 11, 12, 2, 7, 8, 13, 3, 4, 9, 14) and D=(0, 5, 10, 15, 4, 9, 14, 3, 8, 13, 2, 7, 12, 1, 6, 11). For such S, we have $S^{\ 1}$=(0, 4, 8, 12, 13, 1, 5, 9, 10, 14, 2, 6, 7, 11, 15, 3). Hence, $S^{\ 1} \times D$ = (0, 4, 8, 12, 1, 5, 9, 13, 2, 6, 10, 14, 3, 7, 11, 15) = P. Thus, after our conflict-fre permutation algorithm using S and D are executed, permutation along P can be completed. Also, reading operations from a and writing operations in b are conflict-free For example, warp $W(1)$ reads from $a[1], a[6], a[11], a[12]$ which are in banks $B[1], B[2], B[3], B[0]$, respectively. It also writes in $b[4], b[9], b[14], b[3]$ which are in banks $B[0], B[1], B[2], B[3]$, respectively.

Let us evaluate the computing time of our conflict-fre permutation algorithm. We assume that n threads are used to permute an array of size n. Since we have $\frac{n}{w}$ warps of w threads each and reading from array s involve no bank conflict reading from array s takes $O(\frac{n}{w} + l)$ time units. Similarly, reading from array a and d, and writing in array b also take $O(\frac{n}{w} + l)$ time units. On the other hand, in the worst case, the Destination-designated and Source-designated permutation algorithms take $O(n + l)$ time units if memory access by a warp is performed to the same bank.

4.2 Graph coloring based conflict-f ee permutation

This chapter is devoted to show how S and D are determined from P to guarantee that the conflict-fre permutation using S and D involves no bank conflict The same idea is used in [24].

We use an important graph theoretic result [21, 36] as follows:

Theorem 4.2.1 (König) *A regular bipartite graph with degree ρ is ρ-edge-colorable.*

Figure 4.3 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. In other words, no two edges with the same color share a node. The readers should refer to [21, 36] for the proof of Theorem 4.2.1.

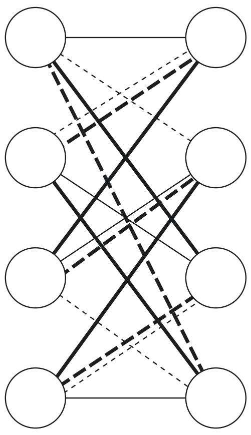

Figure 4.3: A regular bipartite graph with degree 4

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

- ⊆ $U = \}B[0], B[1], B[2], \ldots, B[w \quad 1]\langle$ is a set of nodes each of which corresponds to a bank of a.

- ⊆ $V = \}B[0], B[1], B[2], \ldots, B[w \quad 1]\langle$ 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 connecting $B[i \bmod w](\emptyset\ U)$ and $B[P(i) \bmod w](\emptyset\ V)$.

Clearly, an edge $(B[u], B[v])$ $(0 \geq u, v \geq w \quad 1)$ 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 4.2.1. Suppose that all of the n edges in E are painted by $\frac{n}{w}$ colors 0, 1, . . ., $\frac{n}{w} \quad 1$. We determine value $c_{i,j}$ $(0 \geq i \geq \frac{n}{w} \quad 1, 0 \geq j \geq w \quad 1, 0 \geq c_{i,j} \geq n \quad 1)$ such that an edge $(B[c_{i,j} \bmod w], B[P(c_{i,j}) \bmod w])$ with color i corresponds to a pair of source $a[c_{i,j}]$ and destination $b[P(c_{i,j})]$. It should have no difficulty to confir that, for each i,

⊆ w banks $B[c_{i,0} \bmod w]$, $B[c_{i,1} \bmod w]$, . . ., $B[c_{i,w \quad 1} \bmod w]$ are distinct, and

⊆ w banks $B[P(c_{i,0}) \bmod w]$, $B[P(c_{i,1}) \bmod w]$, . . ., $B[P(c_{i,w \quad 1}) \bmod w]$ are distinct.

Thus, we have the following important lemma:

Lemma 4.2.2 *Let $c_{i,j}$ $(0 \geq i \geq \frac{n}{w} \quad 1, 0 \geq j \geq w \quad 1, 0 \geq c_{i,j} \geq n \quad 1)$ denote a source define above. For each i, we have, (1) $a[c_{i,0}]$, $a[c_{i,1}]$, . . ., $a[c_{i,w \quad 1}]$ are in different banks, and (2) $b[P(c_{i,0})]$, $b[P(c_{i,1})]$, . . ., $b[P(c_{i,w \quad 1})]$ are in different banks.*

We defin permutation S and D using $c_{i,j}$ as follows:

$$S(i \times w + j) = c_{i,j}$$

$$D(i \times w + j) = P(c_{i,j})$$

Suppose that the conflict-fre permutation algorithm using S and D above is executed. Since the copy operation is performed from $a[c_{i,j}]$ to $b[P(c_{i,j})]$, the permutation along P is completed correctly. Also, each warp $W(i)$ $(0 \geq i \geq \frac{n}{w} \quad 1)$ performs copy operation from $a[c_{i,0}], a[c_{i,1}], \ldots, a[c_{i,w \quad 1}]$ to $b[P(c_{i,0})], b[P(c_{i,1})], \ldots, b[P(c_{i,w \quad 1})]$. From Lemma 4.2.2, reading from a and writing in b by warp $W(i)$ are conflict-free

4.3 Implementation of conflict-f ee permutation algorithm

The main purpose of this section is to show an implementation of the conflict-fre permutation algorithm to the GPU using CUDA.

Suppose that a permutation P of $(0, 1, \ldots, n \quad 1)$ is given. We firs draw a bipartite graph $G = (U, V, E)$ of P shown in the previous chapter and fin an edge coloring. Recall that edges are painted by $\frac{n}{w}$ colors so that no two edge with the same color shares a node. Clearly, the edge coloring can be done by repeating a bipartite graph matching $\frac{n}{w}$ times. Also, it is known that a maximum bipartite graph matching, which is a subset of edges sharing no node, can be found in polynomial time.

For the reader's benefits we briefl explain how a maximum bipartite graph matching can be found. Please see [11] for the details. Let $G = (U, V, E)$ be a bipartite graph and M ($\leq E$) is a matching. Note that M may not be a maximal. A path A of G is called *an augmenting path* if

⊆ two terminals of A are not connected to M, and

⊆ edges of M and $\overline{M}(= E \quad M)$ appear alternatively in A.

Figure 4.4 shows examples of augmenting paths.

Clearly, the firs and the last edges are in $\overline{M}$. Also, in an augmenting path A, the number of edges of $\overline{M}$ is exactly one larger than that of M. In other words, $\sqrt{A} \lfloor \overline{M} \sqrt{=}$ $\sqrt{A} \lfloor M \sqrt{+}$ 1 holds.

Let us consider *the flippin operation* for an augmenting path as follows:

⊆ $M \prime \quad M \quad (A \lfloor M)$, that is, remove edges in $A \lfloor M$ from M.

⊆ $M \prime \quad M \cap (A \lfloor \overline{M})$, that is, add edges in $A \lfloor \overline{M}$ to A.

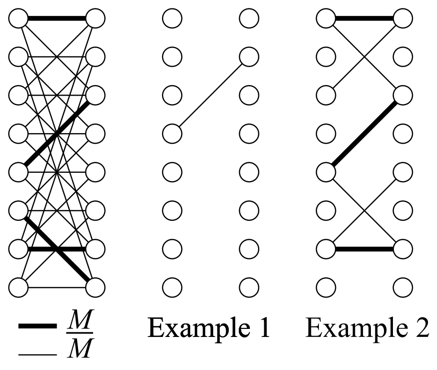

Figure 4.4: Examples of augmenting paths

The reader should refer to Figure 4.5 for illustrating the resulting bipartite matching after the flippin operation. Clearly, the resulting M is a matching and the number of edges in M increases by one.

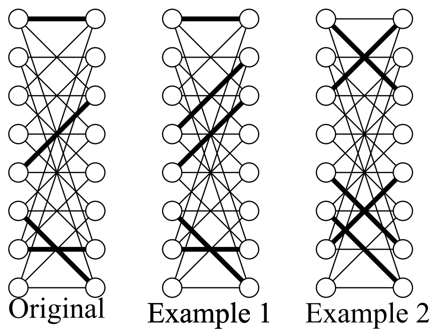

Figure 4.5: The resulting bipartite matching after flippin operation

An augmenting path can be found in polynomial time if it exists. Pick a node connected to no edge in M. Construct a shortest path tree from the picked node such that, in all paths from the root (or the picked node) to the leaves, edges $\overline{M}$ and M appears alternatively. If we can fin a non-root node connected to no edge in M, then the path from the root to the non-root node is an augmenting path.

From these observation, we can fin a maximum matching of a bipartite graph G as follows. Initially, let $M = C$. Find an augmenting path with respect to G and M and performs flippin operation. This task is repeated until we can fin no augmenting path with respect to G and M. The resulting matching M is a maximum matching.

For graph coloring, we repeat findin the maximum matching. First, fin the maximum matching M, paint edges in M with color 0, and remove edges in M from G. In this way, we can fin a bipartite graph coloring in polynomial time.

Note that, we perform a bipartite graph coloring in offline. So, it is not necessary to fin a bipartite graph coloring using a GPU. Actually, we have implemented a bipartite graph coloring to run on a convectional Linux PC.

We have implemented permutation algorithms using CUDA. Arrays a and b are define as arrays of n 32-bit floa (or 64-bit double) numbers in the shared memory of the GPU and arrays p, q, s, and d are define arrays of n int numbers in the shared memory as follows:

```
__shared__ floa  a[n], b[n];
__shared__ int p[n], q[n], s[n], d[n];
```

Also, three permutation algorithms are implemented by CUDA device functions as follows:

[Destination-designated permutation algorithm]

```
__device__ d-designated(floa  *a, floa  *b, int *p)}
  b[p[threadIdx.x]]=a[threadIdx.x];
〈
```

[Source-designated permutation algorithm]

```
__device__ s-designated(floa  *a, floa  *b, int *q)}
```

```
  b[threadIdx.x]=a[q[threadIdx.x]];
〈
```

[Conflict-f ee permutation algorithm]

```
__device__ conflict-free(flo   *a, floa  *b, int *s, int *d)}
  b[d[threadIdx.x]]=a[s[threadIdx.x]];
〈
```

Each of the above codes is executed by every thread with a unique ID represented by threadIdx.x such that threadIdx.x = i for $T(i)$.

To clarify the overhead of permutation, we also use a simple copy CUDA device function as follows:

[Copy algorithm]

```
__device__ copy(floa  *a, floa  *b)}
  b[threadIdx.x]=a[threadIdx.x];
〈
```

In other words, the copy algorithm performs identical permutation such that $P(i) = i$ for all i.

Since the permutation algorithms use one or two arrays of p, q, s, and d, we call *the array-use* methods. Table 4.1 summarizes memory access operations performed by each of the permutation algorithms. For example, the Destination-designated permutation algorithm performs read operations for arrays a and p, and write operations for array b. Hence, it performs $2n + n = 3n$ memory access operations. Our conflict-fre permutation algorithm performs $4n$ memory access operations. Thus, if each memory access operation have the same access time, the conflict-fre permutation algorithm is $\frac{4n}{3n} = \frac{4}{3}$ times slower than the Destination-designated and Source-designated permutation algorithms. However, as we are going to show later, our conflict-fre permutation algorithm can be much faster than the Destination-designated and Source-designated permutation algorithms.

Table 4.1: Memory access by each algorithm

Algorithms	a	b	p	q	s	d	read	write
Copy	r	w					n	n
D-designated	r	w	r				$2n$	n
S-designated	r	w		r			$2n$	n
Our conflict-fre	r	w			r	r	$3n$	n

4.4 Important permutations and in-place permutation method

This chapter firs introduces several important permutations used to evaluate the performance of permutation algorithms later. Also, we introduce the in-place permutation method which is the most efficient if a permutation is simple.

We use several widely-used important permutations as follows:

Identical: Permutation such that $P(i) = i$ for every i as Figure 4.6.

Random: One of all possible $n!$ permutations is selected uniformly at random as Figure 4.7.

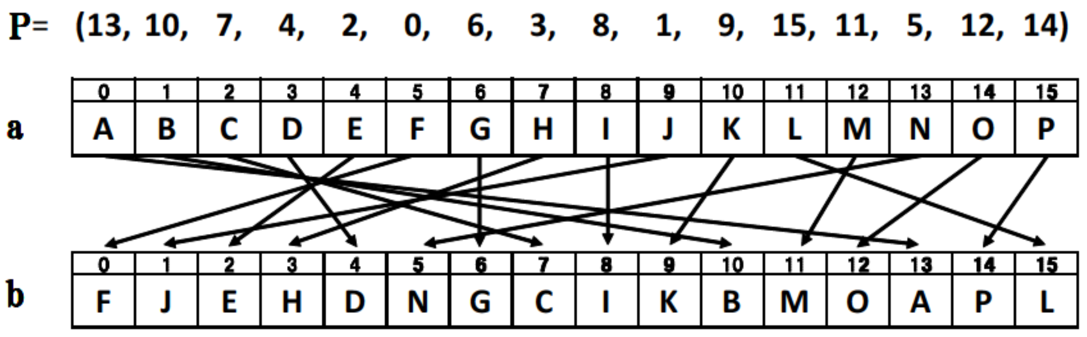
P= (13, 10, 7, 4, 2, 0, 6, 3, 8, 1, 9, 15, 11, 5, 12, 14)
a
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
A B C D E F G H I J K L M N O P
b
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
F J E H D N G C I K B M O A P L

Figure 4.6: Identical permutation

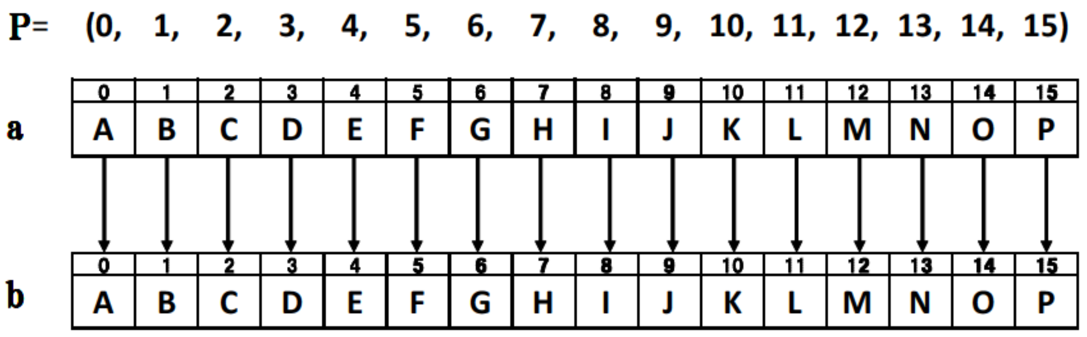
P= (0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15)
a
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
A B C D E F G H I J K L M N O P
b
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
A B C D E F G H I J K L M N O P

Figure 4.7: Random permutation

Transpose: Suppose that a and b are matrix with dimension $\overline{n} \bullet \overline{n}$. Transpose corresponds to the data movement such that a is read in row-major order and b is written in column-major order as Figure 4.8. That is, $P(i \times \overline{n} + j) = j \times \overline{n} + i$ for every i and j $(0 \geq i \geq \overline{n}\ \ 1, 0 \geq j \geq \overline{n}\ \ 1)$.

Shuffle: Let $i_m i_{m\ 1} \times\times\times i_1$ be the binary representation of i. As shown in Figure 4.9, the shuffle permutation is define as $P(i_m i_{m\ 1} \times\times\times i_1) = i_{m\ 1} \times\times\times i_1 i_m$. Shuffle permutation is used for shuffle exchanging in sorting networks [10, 3].

Bit-reversal: The bit-reversal permutation is define as $P(i_m i_{m\ 1} \times\times\times i_1) = i_1 \times\times\times i_{m\ 1} i_m$. Bit-reversal is used for data reordering in the FFT algorithms [31].

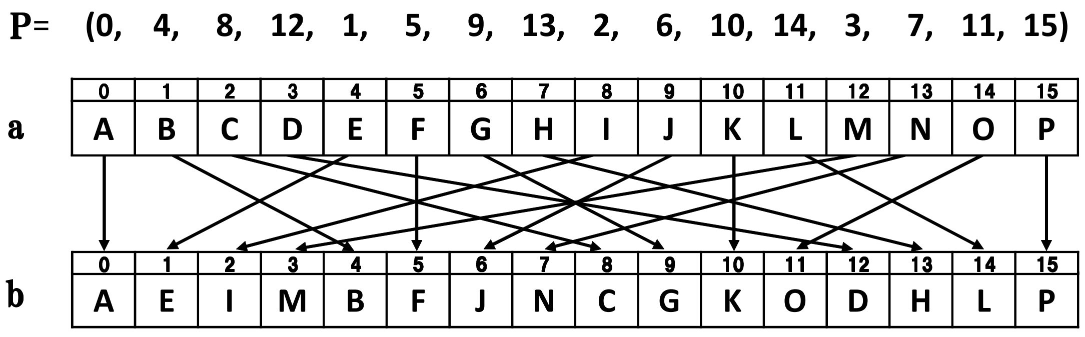
P= (0, 4, 8, 12, 1, 5, 9, 13, 2, 6, 10, 14, 3, 7, 11, 15)
a
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
A B C D E F G H I J K L M N O P
b
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
A E I M B F J N C G K O D H L P

Figure 4.8: Transpose permutation

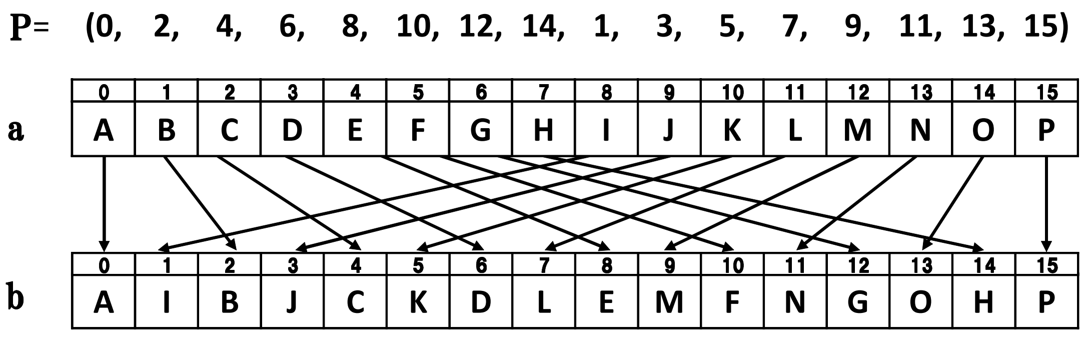
P= (0, 2, 4, 6, 8, 10, 12, 14, 1, 3, 5, 7, 9, 11, 13, 15)
a
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
A B C D E F G H I J K L M N O P
b
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
A I B J C K D L E M F N G O H P

Figure 4.9: Shuffle permutation

If a permutation P is simple and regular, it may be possible to compute the value of $P(i)$ for every i $(0 \geq i \geq n\ \ 1)$ easily. If this is the case, it is not necessary to use array p to store the value P. Instead, each thread computes the value of P(`threadIdx.x`) in place. For simplicity, we assume $n = 1024$ and explain how the values of P(`threadIdx.x`) for the transpose, the shuffle, and the bit-reversal permutations are computed. Let p denote a local integer variable of a thread to store the destination. The values of P(`threadIdx.x`) for the transpose permutation can be computed by the following formula:

```
p = (threadIdx.x >> 5) |
        ((threadIdx.x & 0x1f)<< 5);
```

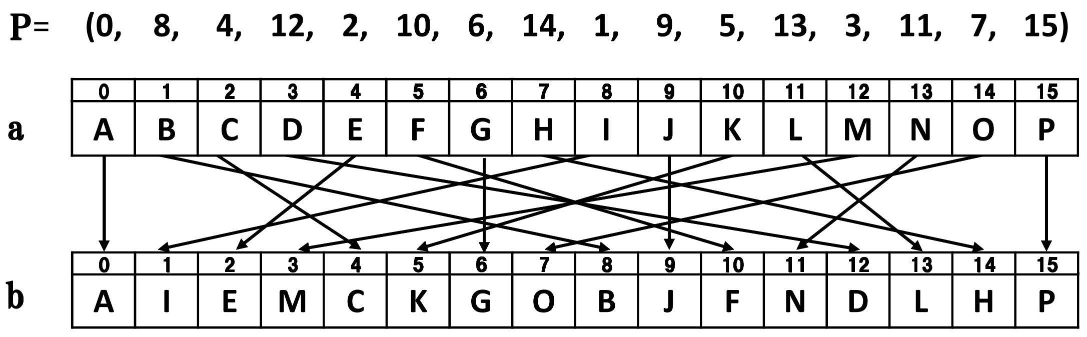
P= (0, 8, 4, 12, 2, 10, 6, 14, 1, 9, 5, 13, 3, 11, 7, 15)
a
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
A B C D E F G H I J K L M N O P
b
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
A I E M C K G O B J F N D L H P

Figure 4.10: Bit-reversal permutation

After the value *p* above is computed, the destination-designated permutation can be done by executing the following assignment in parallel.

```
b[p]=a[threadIdx.x];
```

The value of *P*(`threadIdx.x`) for the shuffle permutation can be obtained by the following assignment:

```
p = (threadIdx.x >> 9) |
        ((threadIdx.x & 0x1ff)<< 1);
```

The following three assignments can perform the bit-reversal permutation. In these formulas, two local variables *u* and *v* are used to store temporal integers.

```
u = (threadIdxIdx.x >> 5) |
        ((threadIdxIdx.x & 0x1f)<< 5);
v = ((u & 0x318) >> 3) | ((u & 0x63)<< 3);
p= ((v & 0x252) >> 1) | ((v & 0x129)<< 1) |
        (u & 0x84);
```

Next, let us consider the Source-designated permutation for the three permutations. Clearly, $P^{\ 1} = P$ for the transpose and the bit-reversal permutations. Hence, the same assignments can be used for these two permutations. Also, the source index of the shuffle permutation can be obtained by the following assignment.

```
q = (threadIdx.x >> 1) |
      ((threadIdx.x & 0x1)<< 9);
```

Thus, the Source-designated permutation method can be done in the same manner as the Destination-designated permutation.

We can apply the same technique to the conflict-fre permutation. In other words, the values of $S(i)$ and $D(i)$ can be computed in place without using arrays s and d. Let s and d denote local integer variables to store the source and the destination. Quite surprisingly, for $n = 1024$, the value of s for the three permutations above can be computed by the following formula:

```
s = threadIdx.x ˆ ((threadIdx.x & 0x1f) << 5);
```

The value d can be computed using the formulas to compute p. For example, the value d for the transpose can be computed using s as follows:

```
d = (s>> 5) | ((s & 0x1f)<< 5);
```

After the values of s and d are computed, the conflict-fre permutation can be done by executing the following assignment in parallel.

```
b[d]=a[s];
```

For the shuffle and the bit-reversal permutations, we can use the above formula for computing s as it is to obtain the conflict-fre permutation.

Note that the in-place permutation approach can be used only for simple permutations such that the values p, q, s, and d can be computed by simple formulas without using arrays to store the pre-computed values. As we have shown, the transpose, the shuffle, and the bit-reversal permutations are examples of simple permutations. However, in general, it may not be possible to compute the values p, q, s, and d by simple formulas if the permutation has no regularity. In particular, there is no simple way to compute these values for the random permutation. One obvious program is to use the switch statement "switch(threadIdx.x)" with n cases. Clearly, this obvious program occupies more space than the permutation methods using arrays of size n to store the source or the destination. Actually, from the Kolmogorov complexity theory [17], the length of programs to compute these values for the random permutation must be proportional to n. It follows that, there is no better way than the program using the switch statement for most of the randomly generated permutations.

4.5 Experimental results

This section is devoted to show the experimental results using GeForce GTX-680 with CUDA Compute Capability 3.0 [5]. The shared memory has $w = 32$ memory banks with access latency $l = 1$. It has two modes: *64-bit mode* and *32-bit mode*. In the 64-bit mode, the word size of each of the 32 banks is 64. In the 32-bit mode, the word size is 32. We have evaluated the performance three permutation algorithms, the Destination-designated permutation algorithm, the Source-designated permutation algorithm, and our conflict-fre permutation algorithm for both of the two modes. The computing time for f ve permutations, the identical, the random, the transpose, the shuffle, and the bit-reversal is evaluated. Further, the in-place permutation method are evaluated for the three permutations, the transpose, the shuffle, and the bit-reversal. Also, to estimate the overhead of these three permutation algorithms, we have evaluated the performance of the simple copy algorithm. Since any permutation algorithm cannot be faster than the copy algorithm, its computing time is the lower bound of that for all permutation algorithms. Hence, we can see the overhead of the computation and/or the memory access performed by permutation algorithms. The performance has been evaluated for arrays of size $n = 1024$. We used the 64-bit mode to permute 64-bit (double) numbers and the 32-bit mode to permute 32-bit (float numbers. A CUDA kernel with a single block of 1024 threads was invoked from the host.

Table 4.2 shows the execution time to permute an array of 64-bit (double) numbers of size $n = 1024$. Since the execution time of each algorithm for $n = 1024$ is too short to measure, each algorithm has been executed for each permutation 100 million times and we have taken its average. The simple copy operation takes 102.8ns, which is the lower bound of the execution time of all permutation algorithm. Our conflict free algorithm runs in 166.7-167.1 ns for all permutations. We can clarify the fact that our conflict-fre algorithm runs in the same time units for any permutation. Also, if the in-place computation is used, our conflict-fre algorithm runs in 105.4-109.0 ns. Since the in-place computation of the bit-reversal is more complicated than the others, it takes a bit more time. However, compared with the simple copy, the overhead of the in-place computation is less than 10% of the total execution time. Thus, if the in-place computation of a required permutation is enough simple, then we should select the in-place conflict-fre permutation algorithm.

The Destination-designated and the Source-designated permutation algorithms both for the transpose and for the bit-reversal permutations involve *the full bank-conflic* , in

Table 4.2: The execution time (ns) of the three algorithms for an 64-bit (double) array of size 1024.

Permutations	Algorithms (array-use)			Algorithms (in-place)			Copy
	D-designated	S-designated	Conflict-fre	D-designated	S-designated	Conflict-fre	
Identical	135.4	124.4	167.1	-	-	-	102.8
Random	247.8	275.5	166.7	-	-	-	
Transpose	1684	1696	167.1	1626	1633	105.4	
Shuffle	178.4	183.4	166.9	160.0	161.3	105.4	
Bit-reversal	1684	1697	166.9	1668	1677	109.0	

the sense that all memory access requests by a warp are destined for the same memory bank. For example, the firs warp of the Destination-designated permutation algorithm read from $a[0]$, $a[1]$, . . ., $a[31]$ and write in $b[0 \times 32]$, $b[1 \times 32]$, . . ., $b[31 \times 32]$ for the transpose permutation. Clearly, the write operations are performed to the same bank of b. We can see this fact that the Destination-designated and the Source-designated permutation algorithms for the transpose and the bit-reversal permutation are 10 times slower than our conflict-fre algorithm. In the shuffle permutation, a pair of memory requests is destined for the same same memory bank. For example, the firs warp of the Destination-designated permutation algorithm read from $a[0], a[1], \ldots, a[31]$ and write in $b[0], b[2], \ldots, b[62]$. Each of 16 pairs $(b[0], b[32])$, $(b[2], b[34])$, . . ., $(b[30], b[62])$ are in the same memory bank. Hence, every two memory bank receives two write requests. Thus, the Destination-designated and the Source-designated permutation algorithms are bit slower than our conflict-fre permutation algorithm for the shuffle permutation.

Table 4.3 summarizes the the cost of memory access requests for arrays a (read) / b (write) and the total costs of the array-use and the in-place methods. For example, the cost of the Destination-designated permutation algorithm for the transpose permutation is 1/32, because reading of *a* has no bank conflic and 32 write requests to *b* are destined for the same memory bank. Also, its total cost for the in-place method is 33. For the array-use method, the Destination-designated permutation algorithm need to access array *p* and its cost is 1. Thus, the total cost for the array-use method is 34. We can see that more the permutation algorithm with more total costs takes more execution time.

Table 4.3: The cost of memory access requests for arrays a (read) / b (write) and the total cost (array-use/in-place)

	64-bit(double)			32-bit(float		
	D-designated	S-designated	Conflict-fre	D-designated	S-designated	Conflict-fre
Identical	1/1 (3)	1/1 (3)	1/1 (4)	1/1 (3)	1/1(3)	1/1(4)
Random	1/3.46 (5.46)	3.46/1 (5.46)	1/1 (4)	1/3.37 (5.37)	3.37/1(5.37)	1/1(4)
Transpose	1/32 (34/33)	32/1 (34/33)	1/1(4/2)	1/16 (18/17)	16/1 (18/17)	1/1 (4/2)
Shuffle	1/2 (4/3)	2/1 (4/3)	1/1 (4/2)	1/1 (3/2)	2/1 (4/3)	1/1 (4/2)
Bit-reversal	1/32 (34/33)	32/1 (34/33)	1/1 (4/2)	1/16 (18/17)	16/1 (18/17)	1/1(4/2)

Table 4.4 shows the execution time to permute an array of 32-bit (float numbers of size $n = 1024$. Each execution time for 32-bit (float numbers is almost equal to the corresponding execution time of 64-bit (double) numbers except the underlined. Each underlined execution time for 32-bit numbers is much smaller than that for 64-bit numbers. This is because the 32-bit mode of the shared memory has some exception of the bank conflict If two memory requests are destined for different 32-bit words of the same bank and these different 32-bit words are aligned in the same 64-bit word, they can be accessed at the same time. For example, two 32-bit words *b*[0] and *b*[32] are in the same bank, but they are aligned in the same 64-bit word. Thus, *b*[0] and *b*[32] can be access at the same time without bank conflict The reader should refer to Figure 4.11

for illustrating the word alignment of the 64-bit and the 32-bit mode. Please see chapter F.5.3 in [5] for the details. From the word alignment of the 32-bit mode, the cost of each permutation algorithm is evaluated as shown in Table 4.3. For example, in the Destination-designated permutation algorithm of the shuffle permutation, the firs warp writes in $b[0]$, $b[2]$, ..., $b[62]$. Since each of 16 pairs $(b[0], b[32])$, $(b[2], b[34])$, ..., $(b[30], b[62])$ are aligned in a 64-bit word, the writing operations by a warp are conflict free. From Tables 4.3 and 4.4, we can see that if more requests are destined for the same bank, a permutation takes more time.

Table 4.4: The execution time (ns) of the three algorithms for an 32-bit (float array of size 1024.

	Algorithms (array-use)			Algorithms (in-place)			
Permutations	D-designated	S-designated	Conflict-fre	D-designated	S-designated	Conflict-fre	Copy
Identical	135.5	123.6	164.9	-	-	-	102.8
Random	245.9	265.8	164.9	-	-	-	
Transpose	<u>876.3</u>	<u>891.0</u>	164.7	<u>839.3</u>	<u>847.5</u>	105.5	
Shuffle	<u>135.3</u>	183.2	164.9	<u>104.0</u>	161.3	105.0	
Bit-reversal	<u>876.3</u>	<u>891.2</u>	164.8	<u>862.0</u>	<u>870.5</u>	108.9	

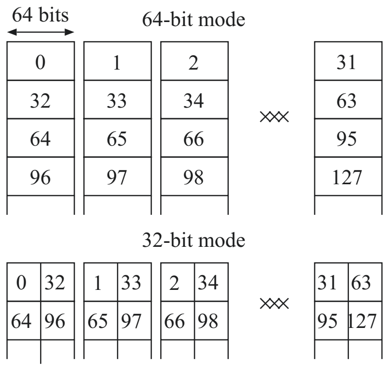

Figure 4.11: The word alignments of the 64-bit and 32-bit modes

By comparing Tables 4.2, 4.3, and 4.4, we can see that the execution time is almost proportional to the total cost. More specificall , the total cost multiplying by 50ns is a moderately good estimation of the execution time. For example, the total cost of the Destination-designated permutation algorithm (array-use) for the 64-bit transpose is 34. Hence, we can estimate that the execution time is 1700ns, while the experimental result shows that the execution time is 1684ns. Thus, we can say that the DMM is a good theoretical model of GPUs.

Suppose that some new permutation is given and we need to write a program for it. We can use the Destination-designated or the Source-designated permutation algorithms if the execution time is not dominant in the whole application program. If we want to minimize the execution time we should use the conflict-fre permutation algorithm. If the permutation is so simple that we can write a simple program to compute the values of $s(i)$ and $d(i)$ of the conflict-fre permutation, we should choose the in-place conflict free permutation algorithm. If this is the case, the execution time is almost the same as the simple copy program. If we cannot fin such simple program, we should use graph-coloring based conflict-fre permutation algorithm using two additional arrays s and d.

4.6 Conclusion

The main contribution of this chapter is to implement several permutation algorithms including our conflict-fre permutation algorithm on the shared memory of NVIDIA GeForce GTX-680. The experimental results for 1024 64-bit numbers on NVIDIA GeForce GTX-680 show that the destination-designated permutation algorithm takes 247.8 ns for the random permutation and 1684ns for the worst permutation that involves the maximum bank conflicts Our conflict-fre   permutation algorithm runs in 167ns for any permutation including the random permutation and the worst permutation, although it performs more memory accesses.

Chapter 5

An Optimal Offline Permutation Algorithm on the Hierarchical Memory Machine

Offline permutation is a task to move numbers along a permutation given beforehand. More specificall , for given two arrays *a* and *b* of size *n*, and a permutation *P*, the value of each $a[i]$ $(0 \geq i \geq n \quad 1)$ is copied to $b[P(i)]$. A conventional algorithm can complete the offline permutation by executing $b[p[i]]$ ′ $a[i]$ for all i $(0 \geq i \geq n \quad 1)$ in parallel, where an array *p* stores the permutation *P*.

The offline permutation has many applications in the area of parallel computing. For example, matrix transpose, which is one of the important permutations, is frequently used in matrix computation. It is known that the computation of the FFT can be done by a multistage network in which each stage involves permutation [31]. Sorting networks such as bitonic sorting [10], [3] also involve permutation in each stage. Further, communication on processor networks such as hypercubes, meshes, and so on can be emulated by permutation. Further, random permutation is very helpful for randomized algorithms [20, 34].

On the PRAM, the conventional permutation algorithm achieves the optimal running time. Since $b[p[i]] \leftarrow a[i]$ can be done in parallel using n processors, the conventional permutation algorithm runs in $O(1)$ time on the PRAM. However, the running time of the conventional algorithm on GPUs depends on the permutation. As we will show in this chapter, the conventional algorithm for permutation P takes a lot of time for most of all possible permutations.

In chapter 4, we have presented a conflict-fre offline permutation algorithm running in $O(\frac{n}{w} + \frac{nl}{p} + l)$ time units using p threads on the DMM with width w and latency l. Later, we have implemented the conventional offline permutation algorithm and this conflict-fre permutation algorithm on a single SM of GeForce GTX-680 GPU and evaluated the performance. The experimental results showed that the conventional permutation algorithm and the conflict-fre permutation algorithm run in 246ns and in 165ns, respectively, for the random permutation of 1024 floa (32-bit) numbers. Hence, the conflict-fre permutation algorithm is 1.5 times faster. However, since the shared memory has only 48Kbits, it is not possible to permute larger arrays than 4096 floa (32-bit) numbers. It is also shown in chapter 4 an offline permutation algorithm running in $O(3^{\log \frac{\log n}{\log w}}(\frac{n}{w}+\frac{nl}{p}+l))$ time units using p threads on the UMM with width w and latency l. This algorithm is time optimal only for small n such that $n \geq w^{O(1)}$. This permutation algorithm has large overhead for large n.

The main contribution of this chapter is to present an optimal permutation algorithm for larger arrays on the global memory of the HMM. Our scheduled offline permutation algorithm performs three step permutations, row-wise permutation, column-wise permutation, and row-wise permutation, each of which is performed in DMMs of the HMM in parallel. Our scheduled offline permutation runs in $32\frac{n}{w} + 16L \quad 16$ time units using n threads on the HMM with width w and global memory latency L. This algorithm is time optimal in the sense that permutation takes at least $\Omega(\frac{n}{w} + L)$ time units. We also show that the conventional algorithm runs in $D_w(P) + 2\frac{n}{w} + 3L \quad 3$ time units, where $D_w(P)$ is the distribution of P, which takes a value between $\frac{n}{w}$ and n. Intuitively, $D_w(P)$ is large if the distribution of contiguous w values in P is large. Hence the computing time of the conventional algorithm is between $3\frac{n}{w} + 3L \quad 3$ and $n + 2\frac{n}{w} + 3L \quad 3$ time units.

The readers may think that, our scheduled permutation algorithm is not practically fast on GPUs, although it is time optimal from the theoretical point of view. The constant factors 32 and 16 in the running time seem too large to achieve better performance than the conventional algorithm with small constant factors in the computing time. However, contrary to this instinct, our scheduled permutation algorithm can run faster than the conventional algorithm. To show this fact, we have implemented our scheduled offlin permutation algorithm on GeForce GTX-680 GPU and evaluate the performance for various permutations. The experimental results show that, the running time of our scheduled offline permutation algorithm terminates in constant time for any permutation of the same size. In other words, the computing time depends on the size of the input array, but is independent of permutation P.

On the other hand, the computing time of the conventional algorithm depends on the permutation. The experimental results also show that, for permutations with large distribution, our scheduled permutation algorithm runs faster than the conventional algorithm whenever $n \approx 256\text{K}$ $(= 2^{18})$. For example, our offline permutation algorithm runs in 780ms for any permutation of 4M (= 2^{22}) floa (32-bit) numbers. The conventional algorithm takes 2328ms for the bit-reversal permutation.

We also show that, for almost all of the permutations over all possible n! permutations, our scheduled permutation algorithm is faster than the conventional algorithm. To show this fact, we pick 1000 permutations from all possible n! permutations at random for n =4M(= 2^{22}). The conventional algorithm takes 424.87-426.39ms, while our scheduled permutation algorithm takes 173.50-173.92ms. Thus, our scheduled permutation algorithm is 2.45 time faster than the conventional algorithm for almost all permutations over all possible n! permutations.

This chapter is organized as follows. In Section 5.1, we defin three memory access operations, casual memory access, coalesced memory access, and conflict-fre memory access and evaluate the running time. Section 5.2 define the offline permutation and show two conventional permutation algorithms, destination-designated permutation algorithm and source-designated permutation algorithm. Section 5.3 presents an algorithm for transposing a matrix, and Section 5.4 shows algorithms for row-wise permutation and column-wise permutation of a matrix. In Section 5.5, we present our scheduled permutation algorithm and show the optimality. Finally, Section 5.6 shows experimental results for comparing the conventional permutation algorithms and our scheduled permutation algorithm. Section5.7 concludes our work.

5.1 Coalesced, Conflict-F ee, and Casual Memory Access

This section firs define a round of memory access by threads. We also defin offline permutation and show conventional algorithms for this task.

We can evaluate the performance of algorithms on the HMM by the number of rounds of memory access. *A round of memory access* is an operation such that all threads perform a single memory access to the shared memory or the global memory. For example, the conventional permutation algorithm performing $b[p[i]] \prime\ \ a[i]$ involves one reading round for a and p each, and one writing round for b.

Next, we defin coalesced and conflict-fre memory access rounds. A round of memory access by a warp of w threads is *coalesced* if all memory access by a warp destined for the same address group of the global memory. Also, that by a warp is *conflict-f ee* if all memory access by a warp destined for the distinct memory banks of the shared memory. More specificall , a round of the memory access by a warp is *coalesced* if $\rfloor\frac{m(0)}{w}\{ = \rfloor\frac{m(1)}{w}\{ = \times\times\times = \rfloor\frac{m(w\ \ 1)}{w}\{$, where $m(i)(0 \geq i \geq w\ \ \ 1)$ is the address accessed by thread $T(i)$ in the warp. A round of the memory access by a warp is *conflict-f ee* if, for all pair i and j $(0 \geq i < j \geq w\ \ \ 1)$, $m(i) = m(j)$ or $m(i) \neq m(j) \pmod{w}$. We also say that a round of the memory access by all of the n threads is coalesced if memory access by all of the $\frac{n}{w}$ warps is coalesced. Also, that by n threads is *conflict-f ee* if memory access by every warp is conflict-free For example, in the conventional permutation algorithm, a round of the memory access to a and p are coalesced. However, that to b may not be coalesced or conflict-free Clearly, the memory access is conflict-fre if it is coalesced. We also say that a round of memory access is casual if it is not guaranteed to be coalesced or conflict-free For example, a round of access to b in the conventional permutation algorithm is casual because it may not be coalesced.

Let us evaluate the time necessary for coalesced and conflict-fre memory access. Suppose that n threads perform a round of coalesced memory access to the global memory. Since we have $\frac{n}{w}$ warps each of which sends w memory requests to the same address group, it takes $\frac{n}{w}$ time units to send all n memory requests, after that $L \quad 1$ time units are necessary to complete the memory requests by the last warp. Thus, it takes $\frac{n}{w} + L \quad 1$ time units to complete a round of coalesced memory access by n threads. Similarly, a round of conflict-fre memory access for the shared memory takes $\frac{n}{w}$ time units to send all memory requests. Since the latency of the shared memory on the HMM is 1, the memory access is completed in $\frac{n}{w}$ time units. Thus, we have,

Lemma 5.1.1 *A round of coalesced memory access for the global memory and that of conflict-f ee memory access for the shared memory by n threads take* $\frac{n}{w} + L \quad 1$ *time units and* $\frac{n}{w}$ *time units, respectively.*

Note that casual memory access by n threads may be destined for the different address group or the same memory bank. If this is the case, it takes n time units to send n memory requests. Thus, the casual memory access to the global memory and the shared memory may take $n + L \quad 1$ time units and n time units, respectively.

5.2 Offline Permutation and Conventional Algorithms

Let us defin the permutation of an array as follows. Suppose that we have two arrays a and b of size n. Let P be a permutation of $(0, 1, ..., n \quad 1)$. In other words,

$(P(0), P(1), ..., P(n \quad 1))$ take distinct integers in the range $[0, n \quad 1]$. Offline permutation along P is a task to copy $a[i]$ to $b[P(i)]$ for all i ($0 \geq i \geq n \quad 1$). We assume that $(P(0), P(1), ..., P(n \quad 1))$ are stored in an array p of size n, such that $p[i] = P(i)$ for all i ($0 \geq i \geq n \quad 1$). The following algorithm can perform the offline permutation:

[Destination-designated permutation algorithm]

```
for $i$ ′ $0$ to $n \quad 1$ do
  $T(i)$ performs $b[p[i]]$ ′ $a[i]$
```

The Destination-designated (D-designated) permutation algorithm involves three rounds of memory access: one round of coalesced reading from a, one round of coalesced reading from p, and one round of casual writing in b. Thus, we have

Lemma 5.2.1 *The D-designated permutation algorithm performs the offline permutation by memory access rounds in Table 5.1.*

Table 5.1: The number of rounds and the running time of algorithms on the HMM

	global memory				shared memory		running time
	casual reading	casual writing	coalesced reading	coalesced writing	conflict-fre reading	conflict-fre writing	
D-designated permutation	-	1	2	-	-	-	$D_w(P) + 2\frac{n}{w} + 3L \quad 3$
S-designated permutation	1	-	1	1	-	-	$D_w(P^{\ 1}) + 2\frac{n}{w} + 3L \quad 3$
Transpose	-	-	1	1	1	1	$4\frac{n}{w} + 2L \quad 2$
Row-wise permutation	-	-	3	1	2	2	$8\frac{n}{w} + 4L \quad 4$
Column-wise permutation	-	-	5	3	4	4	$16\frac{n}{w} + 8L \quad 8$
Our scheduled permutation	-	-	11	5	8	8	$32\frac{n}{w} + 16L \quad 16$

We can design the Source-designated (S-designated) permutation algorithm using the inverse permutation $P^{\ 1}$ of P such that $P^{\ 1}(P(i)) = i$ for all i ($0 \geq i \geq n \quad 1$). Suppose that $P^{\ 1}(0)$, $P^{\ 1}(1)$, ..., $P^{\ 1}(n \quad 1)$ are stored in an array q of size n, such that $q[i] = P^{\ 1}(i)$ for all i ($0 \geq i \geq n \quad 1$). The following algorithm can perform the offline permutation:

[Source-designated permutation algorithm]

```
for $i$ ′  0 to $n$   1 do
  $T(i)$ performs $b[i]$ ′   $a[q[i]]$
```

Clearly, memory access to b and q are coalesced, while that to a may not. Thus, we have

Lemma 5.2.2 *The S-designated permutation algorithm performs the offline permutation by memory access rounds in Table 5.1.*

Let us defin several important permutations that will be used to evaluate the performance of permutation algorithms by experiments on the GPU.

Identical: Permutation such that $P(i) = i$ for every i.

Random: One of all possible $n!$ permutations is selected uniformly at random.

Transpose: Suppose that a and b are matrix with dimension $\quad \overline{n} \bullet \quad \overline{n}$. Transpose corresponds to the data movement such that a is read in row-major order and b is written in column-major order. That is, $P(i \times \quad \overline{n} + j) = j \times \quad \overline{n} + i$ for every i and j ($0 \geq i \geq \quad \overline{n} \quad 1, 0 \geq j \geq \quad \overline{n} \quad 1$).

Shuffle: Let $i_m i_{m \ 1} \times\times\times i_1$ be the binary representation of i. The shuffle permutation is define as $P(i_m i_{m \ 1} \times\times\times i_1) = i_{m \ 1} \times\times\times i_1 i_m$.

Shuffle permutation is used for shuffle exchanging in sorting networks [10, 3].

Bit-reversal: The bit-reversal permutation is define as $P(i_m i_{m \ 1} \times\times\times i_1) = i_1 \times\times\times i_{m \ 1} i_m$.

Bit-reversal is used for data reordering in the FFT algorithms [31].

For later reference, we defin *the distribution of a permutation* for conventional permutation algorithms. The distribution of a permutation P is the total number of address groups of b accessed by all warps in D-designated permutation algorithm. We can defin the distribution $D_w(P)$ of a permutation P with respect to width w as follows:

$$D_w(P) = \sum_{j=0}^{\frac{n}{w}\quad 1} \surd \}\rfloor \frac{P(j \times w)}{w} \{, \rfloor \frac{P(j \times w + 1)}{w} \{, \ldots, \rfloor \frac{P((j+1) \times w \quad 1)}{w} \{\langle \; \surd$$

where x denote the number of unique elements in a set x. It should be clear that the D-designated permutation algorithm for P occupies $D_w(P)$ pipeline registers for writing in b. Hence, the casual writing in b takes $D_w(P) + L \quad 1$ time units. Similarly, the S-designated permutation algorithm for P takes $D_w(P^{\ 1}) + L \quad 1$ time units for reading from a. Thus, we have,

Lemma 5.2.3 *The D-designated permutation algorithm and the S-designated permutation algorithm for a permutation P take time units shown in Table 5.1.*

Clearly, $D_w(\text{identical}) = \frac{n}{w}$ and $D_w(\text{shuffle}(k)) = D_w(\text{shuffle}(k)^{\ 1}) = 2\frac{n}{w}$. Further, the values of $D_w(\text{bit-reversal})$, $D_w(\text{bit-reversal}^{\ 1}$, $D_w(\text{transpose})$, and $D_w(\text{transpose}^{\ 1})$ are n. Since the random permutation is not a fi ed permutation, $D_w(\text{random})$ is not a constant value. However, we can say that, for enough large n, there exists small $\epsilon > 0$, such that $n \quad \epsilon < D_w(\text{random}) \geq n$ with high probability.

5.3 Transpose of a Matrix on the HMM

This section is devoted to show that the transpose of a matrix a of size $\quad \overline{n} \bullet \quad \overline{n}$ stored in the global memory of the HMM can be done by four memory access rounds. For simplicity, we assume that $\overline{n}$ is a multiple of w. We assume that elements in a matrix a are arranged in the row-major order in the memory space, that is, each $a[i][j]$ in the i-th row and j-th column is allocated in address $(i \times \overline{n} + j)$ of the global memory.

We firs show that a matrix a of size $w \bullet w$ on the global memory can be transposed using one DMM with w^2 threads. We use an array α of size $w \bullet w$ on the shared memory. We write each element in α such that $\alpha[i, j](0 \geq i, j \geq w \quad 1)$, which is allocated in address $i \times w + (i + j) \bmod w$. We call such allocation *the diagonal arrangement*. Figure 5.1 illustrates the diagonal arrangement of a $4 \bullet 4$ matrix. The advantage of the diagonal arrangement is:

⊆ all elements $\alpha[i, 0], \alpha[i, 1], ..., \alpha[i, w \quad 1]$in the same row are arranged in different memory banks, and

⊆ all elements $\alpha[0, j], \alpha[1, j], ..., \alpha[w \quad 1, j]$ in the same column are arranged in different memory banks.

Hence, access to the same row or the same column of α is conflict-free Thus, we can transpose a matrix a using α as follows.

[Transpose of a matrix of size $w \bullet w$]

```
for $i$ ′  0 to $w$  1 do in parallel
  for $i$ ′  0 to $w$  1 do in parallel
    Step 1: $T(i \times w + j)$ performs $\alpha[i, j]$ ′  $a[i][j]$
    Step 2: $T(i \times w + j)$ performs $a[i][j]$ ′  $\alpha[j, i]$
```

Since each $a[i][j]$ is copied to $a[j][i]$ through $\alpha[i, j]$, the transposing can be done correctly. Every element of a in the global memory is read once and written once.

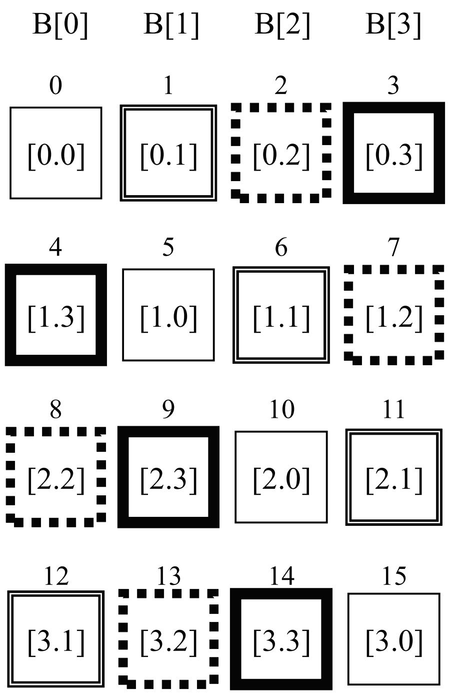

Figure 5.1: Diagonal arrangement with $w = 4$

Also, every element of α in the shared memory is read once and written once. Clearly, memory access to a is coalesced, and that to *alpha* is conflict-free

Next, we will show that the transpose of a matrix a of size $\overline{n} \bullet \overline{n}$ can be done using that of size $w \bullet w$. We assume that $\overline{n}$ is a multiple of w. We partition a into $\frac{\overline{n}}{w} \bullet \frac{\overline{n}}{w}$ submatrices of size $w \bullet w$. Let $A(i, j)$ $(0 \geq i, j \geq \frac{\overline{n}}{w}\quad 1)$ denote a submatrix of elements $a[i^{\in}][j^{\in}]$ $(i \times w \geq i^{\in} \geq (i + 1) \times w \quad 1, j \times w \geq j^{\in} \geq (j + 1) \times w \quad 1)$. The transpose can be done by storing the transpose of each $A(i, j)$ in $A(j, i)$ for all i and j $(0 \geq i, j \geq \frac{\overline{n}}{w}\quad 1)$. This can be done by the transposing algorithm for a $w \bullet w$ matrix. Thus, we have,

Lemma 5.3.1 *The transpose of a matrix of size* $\overline{n} \bullet \overline{n}$ *can be done by memory access rounds and running time in Table 5.1.*

5.4 Row-wise and Column-wise permutation

The main purpose of this section is to show efficient row-wise permutation and column-wise permutation algorithms, which are key ingredients of our scheduled permutation algorithm on the HMM.

Suppose that we have matrices a and b of size $\overline{n} \bullet \overline{n}$ each stored in the global memory. Also, $\overline{n}$ permutations $P_0, P_1, ..., P_{\overline{n}\ 1}$ of $(0, 1, ..., \overline{n}\ \ 1)$ are given. The goal of the row-wise permutation is to copy the value of each $a[i][j]$ $(0 \geq i, j \geq \overline{n})$ to $b[i][P_i(j)]$.

Let D_i and S_i $(0 \geq i \geq \overline{n}\ \ 1)$ be permutations such that a $P_i(S_i(j)) = D_i(j)$ is satisfie for all i and j $(0 \geq i, j \geq \overline{n}\ \ 1)$. We show how D_i and S_i are determined from P_i later. We assume that matrices s and d such that each $s[i][j] = S_i(j)$ and $d[i][j] = D_i(j)$ are also stored in the global memory. We use n threads, which are partitioned into $\overline{n}$ blocks of $\overline{n}$ threads each. Let $B_0, B_1, ..., B_{\overline{n}\ 1}$ denote the $\overline{n}$ blocks. Also, let $T_i(j)$ $(0 \geq i, j \geq \overline{n})$ denote the j-th thread of block B_i. Each B_i $(0 \geq i \geq \overline{n}\ \ 1)$ is assigned to a row $a[i]$ of a and works for the permutation of $a[i]$. Since we have d DMMs, each DMM has $\frac{\overline{n}}{d}$ blocks. We assume that each block B_i $(0 \geq i \geq \overline{n}\ \ 1)$ has two arrays $\alpha_i$and β_i of size $\overline{n}$ each in the shared memory of the DMM. Further, each $T_i(j)$ $(0 \geq i, j \geq \overline{n})$ has two local (register) variables $S_{i,j}$ and $D_{i,j}$. The details of the row-wise permutation are spelled out as follows:

[Row-wise permutation]

```
for $i$ ← 0 to $\overline{n}$ 1 do in parallel
  for $i$ ← 0 to $\overline{n}$ 1 do in parallel
    Step 1: $T_i(j)$ performs $\alpha_i[j]$ ← $a[i][j]$
    Step 2: $T_i(j)$ performs $S_{i,j}$ ← $s[i][j]$ and $D_{i,j}$ ← $d[i][j]$
    Step 3: $T_i(j)$ performs $\beta_i[D_{i,j}]$ ← $\alpha_i[S_{i,j}]$
    Step 4: $T_i(j)$ performs $b[i][j]$ ← $\beta_i[j]$
```

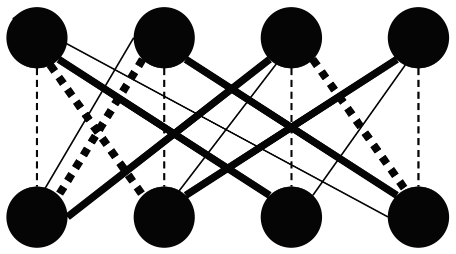

Figure 5.2: A regular bipartite graph with degree 4 painted by 4 colors

It should be clear that $b[i][D_i(j)]$ stores $a[i][D_i(j)]$. Hence, $b[i][D_i(S_i^{\ 1}(j))]$ stores $a[i][S_i(S_i^{\ 1}(j))]$. From $P_i(S_i(j)) = D_i(j)$, we have $P_i(j) = D_i(S_i^{\ 1}(j))$, and thus $b[i][P(j)]$ stores $a[i][j]$. Hence, this algorithm performs the row-wise permutation correctly. We will show that D_i and S_i can be determined from P_i such that $P_i(S_i(j)) = D_i(j)$holds and memory access to α_i and β_i is conflict-free

We use the following graph theoretic result [21], [36]:

Theorem 5.4.1 *A regular bipartite graph with degree ρ is ρ-edge-colorable.*

Figure 5.2 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 [21], [36] for the proof of Theorem 5.5.1.

We will show how D_i and S_i are determined from permutation P_i. We draw a bipartite graph $G = (U, V, E)$ from P_i as follows:

$\subseteq$ $U = B[0], B[1], ..., B[w \quad 1]$ is a set of nodes each of which corresponds to a bank of α_i

$\subseteq$ $V = B[0], B[1], ..., B[w \quad 1]$ is a set of nodes each of which corresponds to a bank of β_i

$\subseteq$ For each pair source $\alpha_i[j]$ and destination $\beta_i[P(j)]$, E has a corresponding edge connecting $B[j \bmod w](\emptyset\ U)$ and $B[P_i(j) \bmod w](\emptyset\ V)$.

Clearly, an edge $(B[u], B[v])$ $(0 \geq u, v \geq w \quad 1)$ corresponds to a number to be copied from bank $B[u]$ of α_i to $B[v]$ of β_j. Also, $G = (U, V, E)$ is a regular bipartite graph with degree $\frac{\bar{n}}{w}$. Hence, $G$is $\frac{\bar{n}}{w}$-colorable from Theorem 5.5.1. Suppose that all of the $\quad \bar{n}$ edges in E are painted by $\frac{\bar{n}}{w}$ colors 0, 1, ..., $\frac{\bar{n}}{w} \quad 1$. We can determine integer values $f_i(j, k)$ $(0 \geq j \geq \frac{\bar{n}}{w} \quad 1, 0 \geq k \geq w \quad 1, 0 \geq f_i(j, k) \geq \quad \bar{n} \quad 1)$ such that an edge $(B[f_i(j, k) \bmod w], B[P(f_i(j, k) \bmod w)])$ with color j corresponds to a pair of source $\alpha_i[f_i(j, k)]$ and destination $\beta_i[P(f_i(j, k))]$. It should have no difficulty to confir that, for each j, (1) w banks $B[f_i(j, 0) \bmod w], B[f_i(j, 1) \bmod w], ..., B[f_i(j, w \quad 1) \bmod w]$are distinct, and (2) w banks $B[P(f_i(j, 0)) \bmod w]$, $B[P(f_i(j, 1)) \bmod w]$, ..., $B[P(f_i(j, w \quad 1)) \bmod w]$ are distinct. It follows that, (1) $\alpha_i[f_i(j, 0)], \alpha_i[f_i(j, 1)], ..., \alpha_i[f_i(j, w \quad 1)]$ are in different banks, and (2) $\beta_i[f_i(j, 0)], \beta_i[f_i(j, 1)], ..., \beta_i[f_i(j, w \quad 1)]$are in different banks. Hence, we defin S_i and D_i from $f_i(j, k)$ such that $S_i(j \times w + k) = f_i(j, k)$ and

$D_i(j \times w + k) = P(f_i(j, k))$ for all j and k ($0 \geq j \geq \frac{n}{w}, 0 \geq k \geq w \quad 1$). For such S_i and D_i, $P(S_i(j)) = D_i(j)$ holds and the memory access to α_i and β_i is conflict-free

Let us evaluate the number of memory access rounds. Step 1 performs one round of coalesced reading from α and one round of coalesced (conflict-free writing in α. Step 2 performs one round of coalesced reading from s and d each. Step 3 involves one round of conflict-fre reading from α and one round of conflict-fre writing in β. Finally, Step 4 performs one round of coalesced (conflict-free reading from β and one round of coalesced writing in b. Note that a, b, s, and d are in the global memory, and α and β are in the shared memory. Thus, we have,

Lemma 5.4.2 *The row-wise permutation can be done by memory access rounds and running time in Table 5.1.*

It should be clear that, the column-wise permutation can be done in three steps: transpose, row-wise permutation, and transpose. Thus, from Lemmas 5.3.1 and 5.4.2 we have,

Lemma 5.4.3 *The column-wise permutation can be done by memory access rounds and running time in Table 5.1.*

5.5 Our Scheduled permutation Algorithm

The main purpose of this section is to show our scheduled offline permutation algorithm on the HMM. The scheduled permutation algorithm uses the row-wise permutation and the column-wise permutation.

Suppose that arrays a and b of size n each are given. Let P be a permutation of $(0, 1, ..., n \quad 1)$. For convenience, we can think that both a and b are matrices of size

$\overline{n} \bullet \ \overline{n}$. For simplicity, we assume that $\overline{n}$ is a multiple of w. The goal of permutation is to move a number stored in $a[i][j]$ to $b[\rfloor P(i \times w + j)\ \ \overline{n}\{][P(i \times w + j) \bmod \ \overline{n}]$ for every i and j $(0 \geq i, j \geq w \ \ 1)$. Note that, the permutation is define for a 1-dimensional array and our scheduled permutation algorithm is not restricted to a square matrix.

Our scheduled permutation has three steps, row-wise permutation (Step 1), column-wise permutation (Step 2), and row-wise permutation (Step 3). We will show how we determine three permutations performed in the three steps. For a given permutation P on a matrix a, we draw a bipartite graph $G = (U, V, E)$ as follows:

⊆ $U = R[0], R[1], ..., R[w \ \ 1]$ is a set of nodes each of which corresponds to a row of a

⊆ $V = R[0], R[1], ..., R[w \ \ 1]$ is a set of nodes each of which corresponds to a row of b

⊆ For each pair source $a[i][j]$ and destination $b[\rfloor P(i \times w + j)\ \ \overline{n}\{][P(i \times w + j) \bmod \ \overline{n}]$, E has a corresponding edge connecting $R[i](\emptyset\ U)$ and $R[\rfloor P(i \times w + j)\ \ \overline{n}\{](\emptyset\ V)$.

Clearly, G is a regular bipartite graph with degree $\overline{n}$. From Theorem 5.5.1, the bipartite graph G thus obtained can be painted using $\overline{n}$ colors such that $\overline{n}$ edges painted by the same color never share a node. Thus, we have that (1) numbers in the same row are painted by different colors, and (2) numbers painted by the same color have different row destination. The readers should refer to Figure 5.3 for illustrating how input numbers are painted.

In Step 1, row-wise permutation is performed such that a number with color i $(0 \geq i \geq \ \ \overline{n} \ \ 1)$ in each row is transferred to the -th column. From (1) above, $\overline{n}$ numbers in each row are painted by $\overline{n}$ colors and thus, Step 1 is possible. Step 2 uses column-wise permutation to move numbers to the fina row destinations. From (2) above, $\overline{n}$ numbers in each column has different $\overline{n}$ row destinations and Step 2 is possible. Finally, in Step 3, row-wise permutation is performed to move numbers to the fina column destinations. The readers should refer to Figure 5.3 for illustrating how numbers are routed by the permutation algorithm for $\overline{n} = 4$. In this figure $(\rfloor P(i \times w + j)\ \ \overline{n}\{][P(i \times w + j) \bmod \overline{n})$ is stored in $a[i][j]$ initially, and after the permutation algorithm terminates, (i, j) is stored in $b[i][j]$.

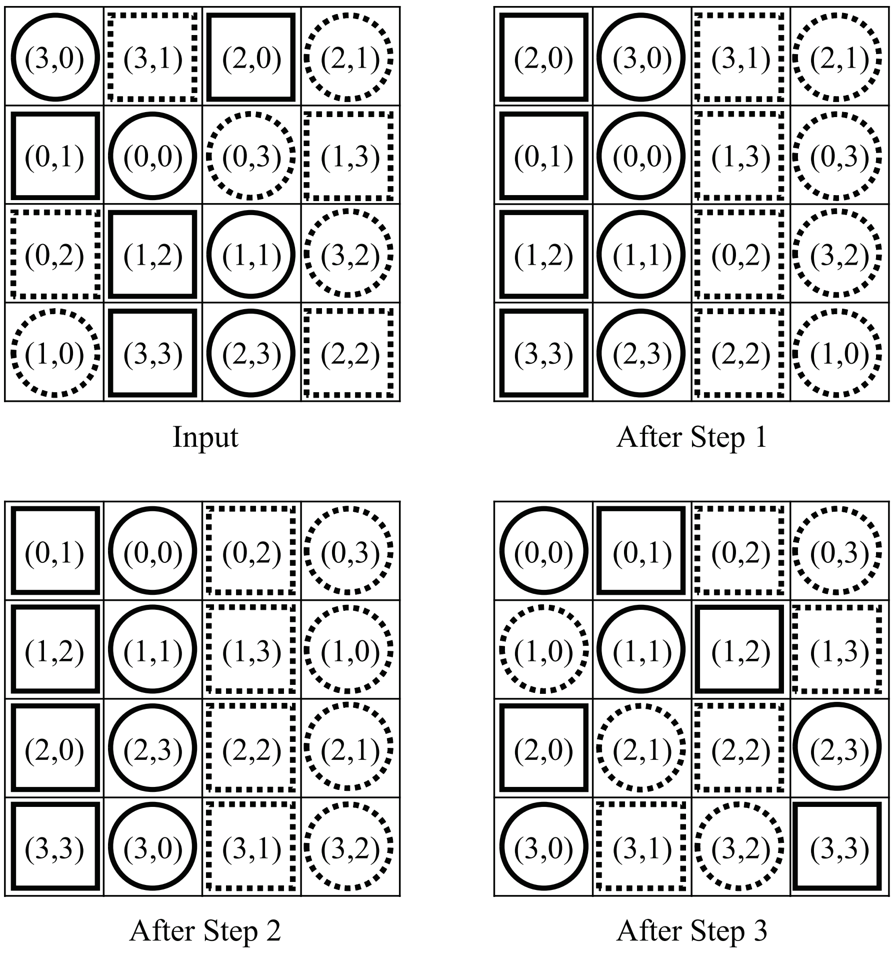

Figure 5.3: Illustrating how numbers are routed by the permutation algorithm

Since the scheduled permutation algorithm on the HMM performs row-wise permutation twice and the column-wise permutation once, we have,

Theorem 5.5.1 *Our scheduled permutation algorithm on the HMM can be done by memory access rounds and running time in Table 5.1.*

We can prove $\Omega(\frac{n}{w} + L)$-time lower bound for the permutation on the HMM. Since all of the n elements in a must be read at least once and w elements can be read in a time unit, $\frac{n}{w}$ time units are necessary. Also, reading of one element takes L time units. Thus, $\Omega(\frac{n}{w}+L)$ time units are necessary for permutation of n elements and our scheduled permutation algorithm is optimal from the theoretical point of view.

5.6 Experimental Result

The main purpose of this section is to show experimental results on GeForce GTX-680. We have implemented D-designated, S-designated, and our scheduled algorithm and evaluate the performance for $\overline{n} = 256, 512, 1024, 2048$ and 4096. The experiment is performed for an array a both of floa (32-bit) numbers and of double (64-bit) numbers. Also, f ve permutations, identical, shuffle, random, bitreversal, and transpose permutations are used to evaluate the performance.

We have invoked $\frac{n}{1024}$ 1024 CUDA blocks [5] of 1024 threads each for D-designated and S-designated permutation algorithms. In the D-designated algorithm, each block is assigned to a row of a and works for the copy of the assigned row. Similarly, in the S-designated algorithm, each block is assigned to a row of b. Also, arrays p and q used in D-designated and S-designated are arrays of int (32-bit) numbers, since at most $\log 4096^2 = 24$ bits are necessary

Recall that scheduled permutation algorithm involves three steps, row-wise permutation, column-wise permutation, and row-wise permutation. Also, column-wise permutation has three substeps, transpose, row-wise permutation, and transpose. Thus, it has essentially f ve steps, three for rowwise permutation and two for transpose. The implementation of our scheduled algorithms performs f ve sequential kernel calls for these f ve steps. For the row-wise permutation, $\overline{n}$ CUDA blocks are invoked. However, since each CUDA block can have up to 1024 threads [5], each block is assigned 1024 threads when $\overline{n} \approx 1024$. If this is the case, each thread works for $\frac{n}{1024}$ numbers. Also, arrays s and d used in our scheduled permutation algorithms are 2-dimensional arrays of n short int (16-bit) numbers in the global memory, since at most $\log 4096 = 12$ bits are necessary.

Table 5.2 shows the running time of the three permutation algorithms for f ve permutations. Since the shared memory of GeForce GTX680 has up to 48Kbytes, it is not possible to implement our scheduled algorithm for $4096 \bullet 4096$ double (64-bit) numbers. Thus, we evaluate the performance up to $2048 \bullet 2048$ double (64-bit) numbers. Clearly, for the D-designated and S-designated permutation algorithms, the identical permutation is fastest, because it is just a copy between two arrays.

From Table 5.2, we can see that D-designated and S-designated permutation algorithms take more time for permutation with larger distribution, while our scheduled permutation algorithm takes almost the same running time for each value of $\overline{n}$ Since the identical and the shuffle permutation have very small distribution, our scheduled permutation algorithm cannot be better than the D-designated and S-designated permutation algorithms. Since the random, the bit-reversal, and the transpose permutations have large distribution, our scheduled permutation algorithm runs faster when $\overline{n} \approx 512$. However, our scheduled permutation algorithm is slower when $\overline{n} = 256$. We can presume that the L2 cache of size 512Kbytes [6] on GeForce GTX-680 decreases the overhead of the casual memory access performed by the D-designated and S-designated permutation algorithms efficiently for small n. Also, in most cases, the S-designated permutation algorithm is more efficient that the D-designated. This is because the casual writing takes more running time than the casual reading due to the overhead of cache coherency in writing.

Table 5.3 shows the running time of the three permutation algorithms for double (64-bit) numbers and the values of $\frac{D_w(P)}{n}$. We have selected 1000 permutations P of size 4M at random. The table shows the minimum, the average, and the maximum values for 1000 permutations. We can see that the values of $D_w(P)$ are very close to n for all permutations. Also, the variance of the computing time of each algorithm is very small. Hence, we can say that, for most of all possible permutations, our scheduled permutation is faster than the D-designated and the S-designated permutation algorithms. The identical and the shuffle permutations are examples of few exceptions.

5.7 Conclusion

In this chapter, we have presented an optimal offline permutation algorithm on the HMM, a theoretical model of CUDA-enabled GPUs. We have implemented the optimal offline algorithm and the conventional algorithms on GeForce GTX-680 GPU and evaluate their performance. The experimental results showed that our optimal offline permutation algorithm is faster than the conventional permutation algorithm for most cases.

Table 5.2: The running time (milliseconds) of D-designated, S-designated and Our scheduled algorithm

(a) Permutation for float(32-bit numbers

	D-designated					S-designated					Our scheduled				
$\overline{n}$	256	512	1024	2048	4096	256	512	1024	2048	4096	256	512	1024	2048	4096
identical	0.86	2.48	9.06	33.2	130	0.86	2.49	9.13	33.1	129	3.87	11.7	46.9	173	780
shuffle	0.94	3.05	11.5	44.7	186	0.84	2.47	9.09	33.6	133	3.87	11.7	46.9	174	780
random	1.55	15.1	93.9	425	1756	3.3	15.7	89.8	398	1644	3.87	11.7	47	173	780
bit-reversal	1.6	15.6	95.3	459	2328	3.12	20.8	96.6	414	1870	3.87	11.7	47	173	780
transpose	1.44	21.2	127	636	2850	2.72	17.8	87	370	2037	3.87	11.7	46.9	173	780

(b) Permutation for double(64-bit) numbers

	D-designated				S-designated				Our scheduled			
$\overline{n}$	256	512	1024	2048	256	512	1024	2048	256	512	1024	2048
identical	1.07	3.57	13.5	54.6	1.07	3.60	13.8	54.6	5.07	16.9	66.6	275
shuffle	1.44	5.14	19.7	82.2	1.08	3.57	13.6	54.6	5.09	17.0	66.7	275
random	2.98	21.6	104	452	3.4	21.3	100	424	5.09	17.0	66.6	275
bit-reversal	3.00	22.0	108	559	3.36	25.0	104	498	5.09	17.0	66.6	275
transpose	2.07	22.2	134	638	2.99	15.4	80.3	358	5.12	17.0	66.6	275

Table 5.3: The Running time(milliseconds) of the three permutations and the values of $D_w(P)$ for permutation P of size 4M

D-designated	S-designated	Scheduled	$\frac{D_w(P)}{n}$
424.87	397.89	173.5	0.99987
425.52	398.27	173.66	0.99989
426.39	398.77	173.92	0.99990

Chapter 6

Parallel Algorithms for the Summed Area Table on the Asynchronous Hierarchical Memory Machine

The summed area table (SAT) of a matrix is a data structure frequently used in the area of computer vision which can be obtained by computing the column-wise prefix-sum and then the row-wise prefix-sum [23, 2, 33]. The main contribution of this chapter is to show a global-memory-access-optimal parallel algorithm for computing the summed area table stored in the global memory of the asynchronous HMM.

In the chapter 3, we have introduced the Hierarchical Memory Machine (HMM) [25], which is a hybrid of the DMM and the UMM. The HMM is a more practical parallel computing model that reflect the hierarchical architecture of CUDA-enabled GPUs. Figure 3.4 illustrates the architecture of the HMM. The HMM consists of d DMMs and a single UMM. Each DMM has w memory banks and the UMM has w memory banks. We call the memory banks of each DMM *the shared memory* and those of the UMM *the global memory* after CUDA-enabled GPUs. Each DMM can work independently and can perform the computation using its shared memory. Also, all threads of DMMs work as a single UMM and can access to the global memory. While the memory access latency of the shared memory of GPUs is very low, that of the global memory is several hundred clock cycles [5]. Hence, we assume that the latency of the shared memory is 1, and we use parameter l to denote the latency of the global memory in the HMM.

Suppose that a matrix a of size $\overline{n} \bullet \overline{n}$ is given. The summed area table (SAT) [7] is a matrix b of the same size such that

$$b[i][j] = \sum_{0 \geq i' \geq i, 0 \geq j' \geq j} a[i'][j'].$$

It should have no difficulty to confir that the SAT can be obtained by computing the column-wise prefix-sum and the row-wise prefix-sum as illustrated in Figure 6.1. Once we have the summed area table, the sum of any rectangular area of a can be computed by evaluating

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

Thus, the sum of a rectangular area can be computed using four elements of the summed area table b. Since the sum of any rectangular area can be computed in $O(1)$ time the summed area table has a lot of applications in the are of image processing and computer vision [16]. In [26], They have presented a parallel algorithm that computes the SAT in $O(\frac{n}{w}+\frac{nl}{p}+l\log n)$ time units using p threads on the UMM with width w and latency l. This algorithm is optimal in the sense that any SAT algorithm takes at least $\Omega(\frac{n}{w}+\frac{nl}{p}+l\log n)$ time units. However, this algorithm repeats pairwise addition and has a large constant factor in the computing time and it is not practically efficient.

A straightforward algorithm (2R2W SAT algorithm) on the asynchronous HMM,

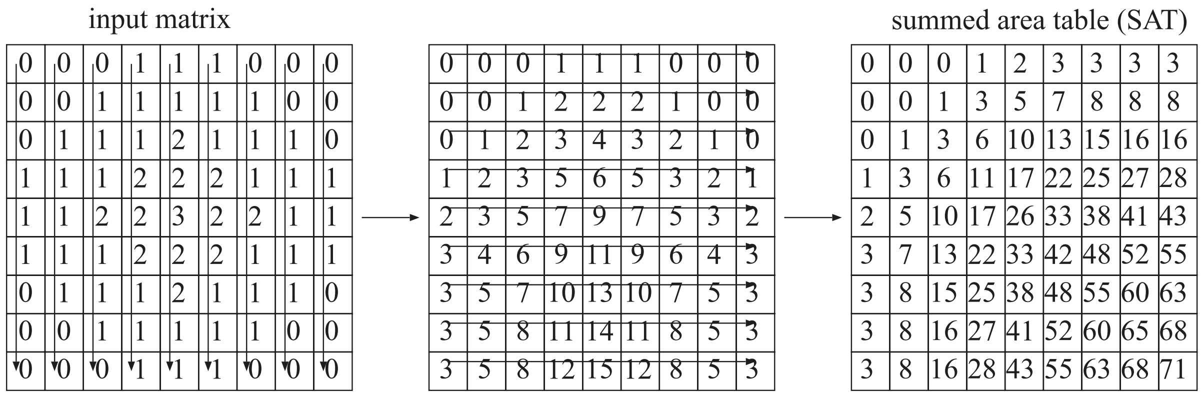

Figure 6.1: The summed area table (SAT) of a 9 • 9 matrix and 2R2W SAT algorithm

which computes the column-wise prefix-sum and then the row-wise prefix-sums performs 2 read operations and 2 write operations per element of a matrix. The best known algorithm (2R1W SAT algorithm) so far performs 2 read operations and 1 write operation per element [28]. We present a more efficient algorithm (1R1W SAT algorithm) on the asynchronous HMM, which performs only 1 read operation and 1 write operation per element. Clearly, since every element in a matrix must be read at least once and all resulting values must be written, our 1R1W SAT algorithm is optimal in terms of the global memory access. We also show a combined algorithm ($(1 + r)$R1W SAT algorithm) of 2R1W and 1R1W SAT algorithms that can run faster than any other algorithms for large matrices. Table 6.1 shows the total number of memory access operations to the global memory and the shared memory, the number of barrier synchronization steps, and the global memory access cost. The global memory access cost, which is computed from the number of global memory access operations and the number of barrier synchronization steps, approximates the computing time on the HMM.

For simplicity, in the table, we omit small terms to focus on dominant terms. For example, 2R2W SAT algorithm performs n $\overline{n}$ coalesced write operations, but we simply write n in the corresponding entry.

Table 6.1: The performance of SAT algorithms on the HMM

SAT algorithms	global memory access		shared memory access	barrier	global memory
	Coalesced	Stride		synchronization	access cost
	Read/Write	Read/Write	Read/Write	steps	
2R2W	n / n	n / n	-	1	$2n + 2\frac{n}{w} + 2l$
4R4W	$4n$ / $4n$	-	n / n	3	$8\frac{n}{w} + 4l$
4R1W	-	$4n$ / n	-	$2 \quad \overline{n}$	$5n + 2 \quad \overline{n}l$
2R1W	$2n$ / n	-	$4n$ / $4n$	$2d + 2$	$3\frac{n}{w} + (2d + 3)l$
1R1W	n / n	-	$2n$ / $2n$	$2\frac{\overline{n}}{w}$	$2\frac{n}{w} + 2\frac{\overline{n}}{w}l$
1.25R1W	$1.25n$ / n	-	$2.5n$ / $2.5n$	$\frac{\overline{n}}{w} + 4d + 4$	$1.25\frac{n}{2} + (\frac{\overline{n}}{w} + 4d + 5)l$
$(1 + r)$R1W	$(1 + r)n$ / n	-	$(2 + \quad \overline{r})n$ / $(2 + \quad \overline{r})n$	$2\frac{(1 \quad \overline{r}) \quad \overline{n}}{w} + 4d + 4$	$(1 + r)\frac{n}{w} + (2\frac{(1 \quad \overline{r}) \quad \overline{n}}{w} + 4d + 5)l$

d is the depth of recursion, which takes value no more than 1 from the practical point of view.

6.1 The global memory access cost on the HMM and the diagonal arrangement on the DMM

Let C, S, and B be the total number of coalesced global memory access operations, the total number of stride global memory access operations, and the number of barrier synchronization steps performed on the HMM. *The global memory access cost* is define to be $\frac{C}{w} + S + (B + 1)(l \quad 1)$. We will show that the global memory access cost approximates the computing time on the HMM if the computation performed in each DMM is negligible.

Suppose that an algorithm performs n coalesced memory access operations and two barrier synchronization steps. Clearly, by two barrier synchronization steps, the memory access is partitioned into three stages as illustrated in Figure 6.2. Let n_0, n_1, and n_2 such

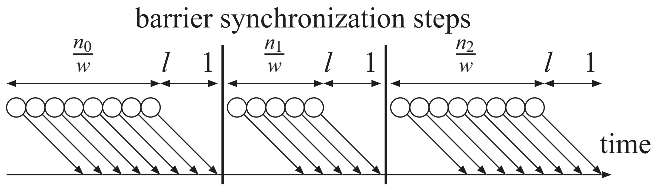

Figure 6.2: Timing chart of coalesced memory access to the global memory with two barrier synchronization steps

that $n = n_0 + n_1 + n_2$ be the numbers of memory access operations performed in the three stages. Since w coalesced memory access operations by a warp of w threads occupy one pipeline stage, the three stages takes $\frac{n_0}{w} + l \quad 1$, $\frac{n_1}{w} + l \quad 1$, and $\frac{n_2}{w} + l \quad 1$ time units respectively. Hence, the algorithm runs $\frac{n}{w} + 3(l \quad 1)$ time units. Also, if n memory access operations are stride, each memory access operation occupy one pipeline stage, the algorithm runs in $n + 3(l \quad 1)$ time units. In general, if an algorithm performs C coalesced memory access operations, S stride memory access operations, and B barrier synchronization steps, it runs $\frac{C}{w} + S + (B + 1)(l \quad 1)$, which is the global memory access cost.

Suppose that we have a matrix of size $w \bullet w$ in the shared memory of a DMM in the HMM. Since a column of the matrix is in the same bank, column-wise access by w threads has bank conflicts while row-wise access is conflict-free In [24], we have presented *a diagonal arrangement* of a matrix such that each (i, j) element is arranged in $a[i][(i + j) \bmod w]$. Figure 6.3 illustrates the diagonal arrangement of a $4 \bullet 4$ matrix. We can confir that both a row-wise access to $(1, 0), (1, 1), (1, 2), (1, 3)$ and a column-wise access to $(0, 1), (1, 1), (2, 1), (3, 1)$ are conflict-free Thus, we have,

Lemma 6.1.1 *In the diagonal arrangement of a $w \bullet w$ matrix, both a row-wise access and a column-wise access are conflict-f ee.*

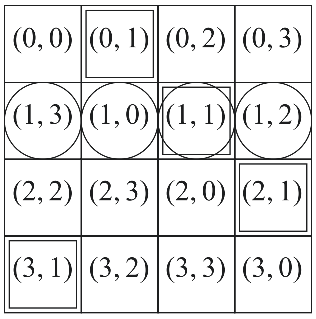

Figure 6.3: Diagonal arrangement of a $4 \bullet 4$ matrix

The diagonal arrangement is used to compute the SAT of a $w \bullet w$ matrix in a shared memory and transpose of a matrix in the global memory of the HMM.

6.2 2R2W and 4R4W SAT algorithms

Let s_i denote a local register of thread $T(i)$ $(0 \geq i \geq n \quad 1)$. As illustrated in Figure 6.1, the summed area table (SAT) of a $\overline{n} \bullet \overline{n}$ matrix a can be computed by the column-wise prefix-sum and the row-wise prefix-sums

[2R2W SAT algorithm]

```
for $i \leftarrow 0$ to $\overline{n}$ do in parallel // column-wise prefix-sum
    $T(i)$ performs $s_i \leftarrow a[0][i]$
  for $j \leftarrow 1$ to $\overline{n} \quad 1$ do
    $T(i)$ performs $s_i \leftarrow s_i + a[j][i]$
    $T(i)$ performs $a[j][i] \leftarrow s_i$
barrier_synchronization
for $i \leftarrow 0$ to $\overline{n}$ do in parallel // row-wise prefix-sum
    $T(i)$ performs $s_i \leftarrow a[i][0]$
```

```
for $j \leftarrow 1$ to $\sqrt{n} - 1$ do
    $T(i)$ performs $s_i \leftarrow s_i + a[i][j]$
    $T(i)$ performs $a[i][j] \leftarrow s_i$
```

In the computation of the column-wise prefix-sums $a[0][0], a[0][1], \ldots, a[0][\sqrt{n} - 1]$ are read. After that, for each j ($1 \geq j \geq \sqrt{n} - 1$), $a[j][0], a[j][1], \ldots, a[j][\sqrt{n} - 1]$ are read and written. Clearly, memory access to these elements are coalesced. In the computation of the column-wise prefix-sums $a[0][0], a[1][0], \ldots, a[\sqrt{n} - 1][0]$ are read. After that, for each j ($1 \geq j \geq \sqrt{n} - 1$), $a[0][j], a[1][j], \ldots, a[\sqrt{n} - 1][j]$ are read and written. Memory access to these elements are coalesced. Hence, 2R2W SAT algorithm performs $2n \sqrt{n}$ coalesced memory access operations and $2n \sqrt{n}$ stride memory access operations. Since 2R2W SAT algorithm has one barrier synchronization step, we have,

Lemma 6.2.1 *The global memory access cost of 2R2W SAT algorithm is at most* $2n + 2\frac{n}{w} + 2(l - 1)$.

We can avoid stride memory access when we compute the row-wise prefix-sum by transposing a matrix. More specificall , the row-wise prefix-sum can be obtained by transpose, column-wise prefix-sums and transpose. It has been shown in chapter 4 that transpose of a matrix of size $\sqrt{n} \bullet \sqrt{n}$ in the global memory of the HMM can be done in $2n$ coalesced memory access operations with no barrier synchronization step. The idea of the transpose it to partition the matrix into $\frac{\sqrt{n}}{w} \bullet \frac{\sqrt{n}}{w}$ blocks with $w \bullet w$ elements each. We can transpose a block via a $w \bullet w$ matrix with diagonal arrangement in a shared memory of a DMM efficiently. First, a block in the global memory is read in row-wise and it is written in a $w \bullet w$ matrix with diagonal arrangement in row-wise. After that, the

$w \bullet w$ matrix is read in column-wise and it is written in a a block in the global memory in row-wise. The reader should refer to Figure 6.4 illustrating transpose of a block using a $4 \bullet 4$ matrix with diagonal arrangement. By executing this block transpose for all blocks in parallel so that a pair of corresponding two blocks are swapped appropriately, the transpose of a $\overline{n} \bullet \overline{n}$ can be done. The reader should refer to chapter 5 for the details of the transpose.

$$\underset{a}{\begin{array}{|c|c|c|c|c|}\hline 0 & 1 & 2 & 3 & 4 \\ \hline 5 & 6 & 7 & 8 & 9 \\ \hline 10 & 11 & 12 & 13 & 14 \\ \hline \end{array}} \rightarrow \underset{b}{\begin{array}{|c|c|c|c|c|}\hline 0 & 3 & 6 & 9 & 12 \\ \hline 1 & 4 & 7 & 10 & 13 \\ \hline 2 & 5 & 8 & 11 & 14 \\ \hline \end{array}}$$

Figure 6.4: Transpose of a block using a $4 \bullet 4$ matrix with diagonal arrangement

By executing matrix transpose twice, we can design 4R4W SAT algorithm as follows:

[4R4W SAT algorithm]

Step 1: Compute the column-wise prefix-sum

Step 2: Transpose

Step 3: Compute the column-wise prefix-sum

Step 4: Transpose

After Steps 1, 2, and 3, barrier synchronization is necessary. Also, each step needs no more than $2n$ coalesced global memory access. Thus, we have,

Lemma 6.2.2 *The global memory access cost of 4R4W SAT algorithm is at most* $8\frac{n}{w} + 4(l\quad 1)$.

6.3 2R1W SAT algorithm

The main purpose of this chapter is to review a SAT algorithm for GPU shown in [28]. Since this SAT algorithm performs $2n$ read and n write operations to the global memory, we call it 2R1W SAT algorithm. 2R1W SAT algorithm that we will explain is slightly different from that in [28] for easy understanding of the algorithm.

Suppose that a $\overline{n}\bullet\quad\overline{n}$ matrix a is partitioned into $\frac{\overline{n}}{w}\bullet\frac{\overline{n}}{w}$ blocks of $w\bullet w$ elements each. 2R1W SAT algorithm has three steps as follows:

[2R1W SAT algorithm]

Step 1: Each DMM reads a block in the global memory and write it in the shared memory. The column-wise sums, the row-wise sums, and the sum of the block are computed. More specificall , for a block $a^{\in}$ of size $w\bullet w$,

⊆ column-wise sums: $C[i] = \sum_{j=0}^{w\quad 1} a^{\in}[j][i]$ for all i $(0 \geq i \geq w\quad 1)$,

⊆ row-wise sums: $R[i] = \sum_{j=0}^{w\quad 1} a^{\in}[i][j]$ for all i $(0 \geq i \geq w\quad 1)$, and

⊆ sum: $S = \sum_{i=0}^{w\quad 1}\sum_{j=0}^{w\quad 1} a^{\in}[i][j]$.

The column-wise sums of all blocks excluding the bottom blocks are written in the global memory such that they constitute a matrix of size $(\frac{\overline{n}}{w}\quad 1)\bullet\quad\overline{n}$ in the global memory. Similarly, the row-wise sums of all blocks constitute a matrix of size $\overline{n}\bullet(\frac{\overline{n}}{w}\quad 1)$, and the sums constitute a matrix of size $(\frac{\overline{n}}{w}\quad 1)\bullet(\frac{\overline{n}}{w}\quad 1)$. The reader should refer to Figure 6.5 for illustrating the resulting values for a matrix in Figure 6.1 with $w = 3$. Let $\mathcal{R}$, $\mathcal{S}$, and $\cup$ denote matrices of the resulting values for the column-wise sums, the row-wise sums, and the sums, respectively. In Figure 6.5, the sizes of $\mathcal{R}$, $\mathcal{S}$, and $\cup$ are $2\bullet 9$, $9\bullet 2$, and $2\bullet 2$, respectively.

Step 2: The column-wise prefix-sum of $\mathcal{R}$ and the row-wise prefix-sum of $\mathcal{S}$ are computed in the same way as 2R2W SAT algorithm. If $\cup$ is no larger than $w \bullet w$ then we compute the SAT of $\cup$ using a single DMM. Otherwise, we execute 2R1W SAT algorithm recursively for $\cup$. The reader should refer to Figure 6.5 for the resulting values of $\mathcal{R}$, $\mathcal{S}$, and $\cup$.

Step 3-1: Each DMM reads a block from the global memory and write it in the shared memory. Let $a^{\in}$ denote a block read by a particular DMM. It reads w elements in $\mathcal{R}$, and adds them to the top row of $a^{\in}$ so that each of the resulting sums is the sum of all elements above it, inclusive, in the same column. Similarly, it reads w elements in $\mathcal{S}$, and adds them to the leftmost column of $a^{\in}$ so that each of the resulting sums is the sum of all elements to the left-side of it, inclusive, in the same row. Further, an element in $\cup$ is added to the top left corner of $a^{\in}$ so that the resulting sum is the the sum of all blocks above and to the left of it, inclusive. The reader should refer to Figure 6.6 illustrating these operations for a block. Also, Figure 6.5 illustrates the resulting values of all blocks.

Step 3-2: Each DMM computes the SAT of a block obtained in Step 3-1 and the resulting values are written in the global memory. Figure 6.6 illustrates the values of a block before and after this step. The reader should have no difficulty to confir that the block thus obtained stores the SAT of the input matrix correctly.

Let us evaluate the global memory access cost. In Step 1, all elements in a are read, and $\mathcal{R}$, $\mathcal{S}$, and $\cup$ are written. Thus, n elements are read from the global memory and less than $2\frac{n}{w} + \frac{n}{w^2}$ elements are written in the global memory. Note that we should write $\mathcal{S}^T$, that is, transposed $\mathcal{S}$ in the global memory for the purpose of coalesced memory access for the row-wise prefix-sum computation in Step 2. If this is the case, the row-

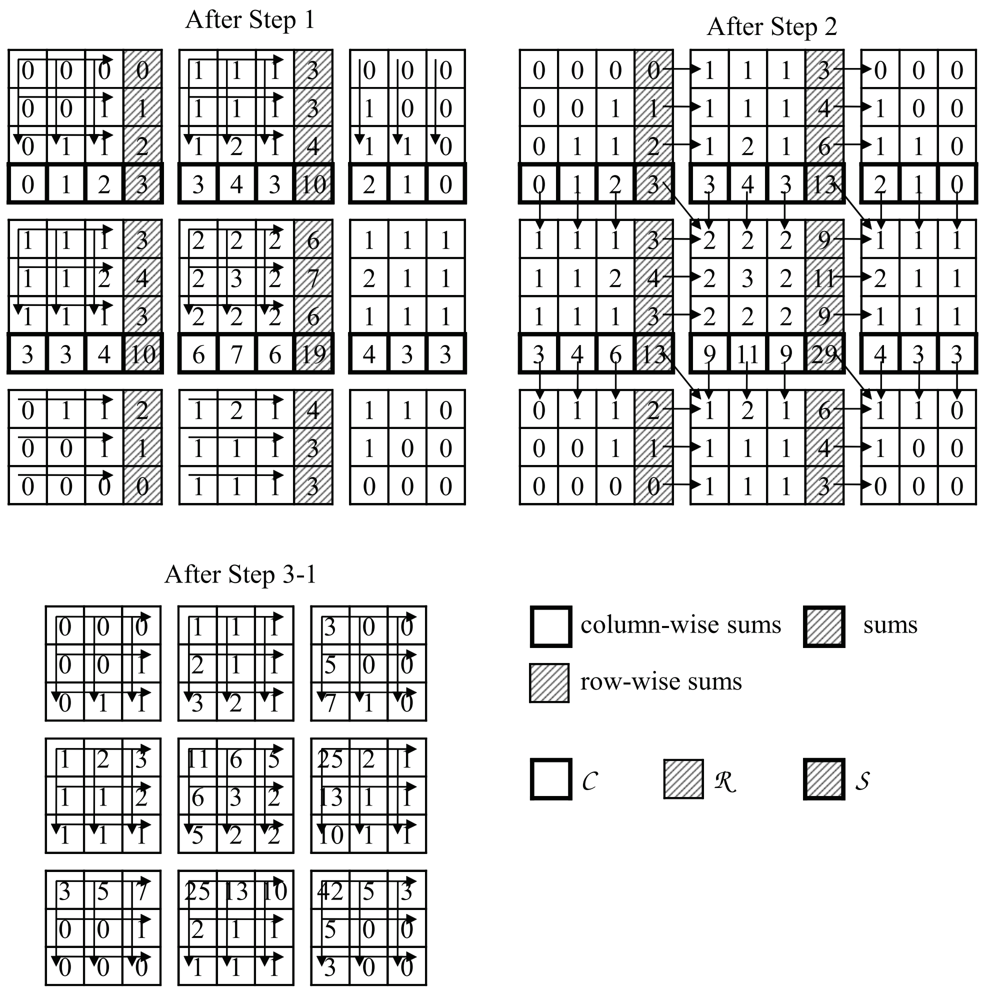

Figure 6.5: 2R1W SAT algorithm executed for a matrix in Figure 6.1 with $w = 3$

wise prefix-sum of $\mathcal{S}$ corresponds to the column-wise prefix-sum of $\mathcal{S}^T$, which can be computed by coalesced memory access to the global memory. Also, in Step 1, the column-wise sums, the row-wise sums, and the SAT of a block in a shared memory are computed. This computation can be done without bank conflict by diagonal arrangement of a block. Each of the d DMMs performs the computation of the column-wise sums for $\frac{n}{w^2 d}$ blocks of size $w \bullet w$. Since the memory access is conflict-free this takes only $\frac{n}{wd}$ time units, which is so small that it can be hidden by latency overhead.

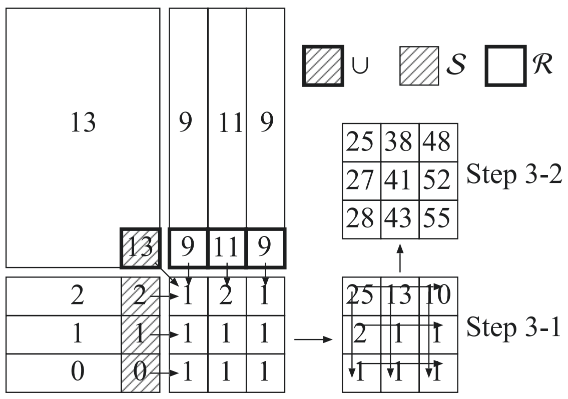

Figure 6.6: Step 3 of 2R1W SAT algorithm

Step 2 performs the computation of the the column-wise prefix-sum and the row-wise prefix-sum for matrices of $\frac{n}{w}$ elements. It also computes the SAT of $\frac{n}{w^2}$ elements, recursively, which performs global memory access to $O(\frac{n}{w^2})$ elements. In Step 3-1 all elements in a and the resulting values of $\mathcal{R}$, $\mathcal{S}$, and $\cup$ are read from the global memory. In Step 3-2 the resulting SAT is written in the global memory. Thus, 2R1W SAT algorithm performs at most $3n + 8\frac{n}{w} + O(\frac{n}{w^2})$ coalesced memory access operations. This includes the memory access by recursive computation of $\cup$.

Let us evaluate the number of barrier synchronization steps. Barrier synchronization step is necessary after Steps 1 and 2 if the SAT of $\cup$ is computed without recursion. If the SAT of $\cup$ is computed recursively, additional two barrier synchronization steps is necessary for each recursion. Hence, if 2R1W SAT algorithm involves d recursions, it performs $2d + 2$ barrier synchronization steps. Thus, we have,

Lemma 6.3.1 *The global memory access cost of 2R1W SAT algorithm with recursion depth d is* $3\frac{n}{w} + 8\frac{n}{w^2} + O(\frac{n}{w^3}) + (2d + 3)(l \quad 1)$.

Since $w = 32$ in current GPUs, $d = 0$ if $n \geq 2^{20}$ and $d = 1$ if $n \geq 2^{30}$. Thus, d is no more than 1 from the practical point of view.

6.4 Our 1R1W SAT algorithm

The main purpose of this chapter is to show our novel SAT algorithm called 1R1W SAT algorithm. This algorithm performs only $n + O(\frac{n}{w})$ read operations and $n + O(\frac{n}{w})$ write operations to the global memory. Before showing 1R1W SAT algorithm, we present 4R1W SAT algorithm. By combining techniques used in 4R1W SAT and 2R1W SAT algorithms, we can obtain 1R1W SAT algorithm.

Let b be the SAT of an input $\overline{n} \bullet \overline{n}$ matrix a. Suppose that the values of $b[i\quad 1][j\quad 1]$, $b[i][j\quad 1]$, and $b[i\quad 1][j]$ are already computed. We can obtain the value of $b[i][j]$ by evaluating the following formula:

$$b[i][j] = a[i][j] + b[i][j\quad 1] + b[i\quad 1][j]\quad b[i\quad 1][j\quad 1] \tag{6.1}$$

From this formula, 4R1W SAT algorithm computes the SAT in a diagonal scan order from the top left to the bottom right. More specificall , 4R1W algorithm has $2\ \overline{n}\quad 1$ stages and each Stage k ($0 \geq k \geq 2\ \overline{n}\quad 2$) computes Formula (6.1) for all i and j such that $i + j = k$. Figure 6.7 illustrates the computation performed in Stage 7.

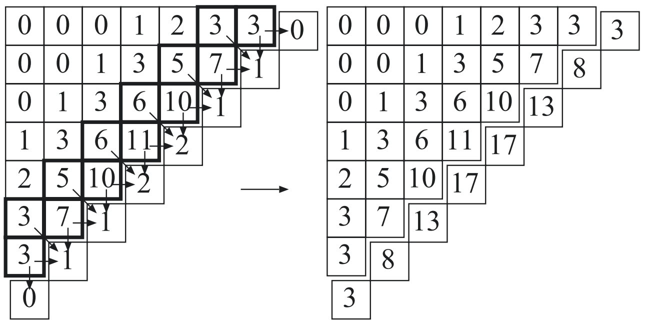

Figure 6.7: Stage 7 of 4R1W SAT algorithm

Let us evaluate the performance of 4R1W SAT algorithm. To compute each $b[i][j]$, 3 elements in b and 1 element in a are read. Also, the resulting value is written in b.

Thus, $4n$ reading operations and n writing operations are performed. Unfortunately, all memory access are stride. Further, barrier synchronization step is necessary after every stage from 0 to $2\quad\overline{n}\quad 3$. Thus, we have,

Lemma 6.4.1 *The global memory access cost of 4R1W SAT algorithm is* $5n+(2\quad\overline{n}\quad 1)l$.

We are now in a position to show our new 1R1W SAT algorithm. The idea is to extend 4R1W SAT algorithm to perform SAT computation in block-wise. In each block-wise computation, a similar computation to 2R1W SAT algorithm is performed. Again, an input matrix a of size $\quad\overline{n} \bullet \quad\overline{n}$ is partitioned into $\frac{\overline{n}}{w} \bullet \frac{\overline{n}}{w}$ blocks of size $w \bullet w$ each. Let $A(i, j)$ $(0 \geq i, j \geq \frac{\overline{n}}{w}\quad 1)$ denote a block in the i-th row and in the j-th column. 1R1W SAT algorithm has $2\frac{\overline{n}}{w}\quad 1$ stages. Each Stage k $(0 \geq k \geq 2\frac{\overline{n}}{w}\quad 2)$ computes the SAT of block $A(i, j)$ with $i + j = k$. The reader should refer to Figure 6.8 for illustrating the computation performed in Stage 3. In Stage 3, $A(2, 1)$ and $A(1, 2)$ are computed using the resulting values in $A(2, 0)$, $A(1, 1)$, and $A(0, 2)$. From these resulting values, we can obtain $\mathcal{R}$, $\mathcal{S}$, and $\cup$ for each of $A(2, 1)$ and $A(1, 2)$. For example, in Figure 6.8, $\mathcal{R}$ for $A(2, 1)$ is $(9, 11, 9)$. This can be obtained by the resulting value 13 of the bottom top corner in $A(1, 0)$ and the resulting values $(22, 33, 42)$ in the bottom row of $A(1, 1)$. More specificall , we can obtain $\mathcal{R}$ by computing pairwise subtraction $(22, 33, 42)\quad(13, 22, 33) = (9, 11, 9)$. After the values of $\mathcal{R}$, $\mathcal{S}$, and $\cup$, these values are added to $A(2, 1)$ in the same way as Step 3-1 of 2R1W SAT algorithm. Finally, we compute the SAT of $A(2, 1)$ in the same way as Step 3-2 of 2R1W SAT algorithm.

Let us evaluate the performance of 1R1W SAT algorithm. In each stage, the values of a block in the global memory are read and the resulting values are written to the global memory. Also, the values necessary to compute $\mathcal{R}$, $\mathcal{S}$, and $\cup$ are read and written to the global memory. For each block, $2w + 1$ elements are read from the global memory for

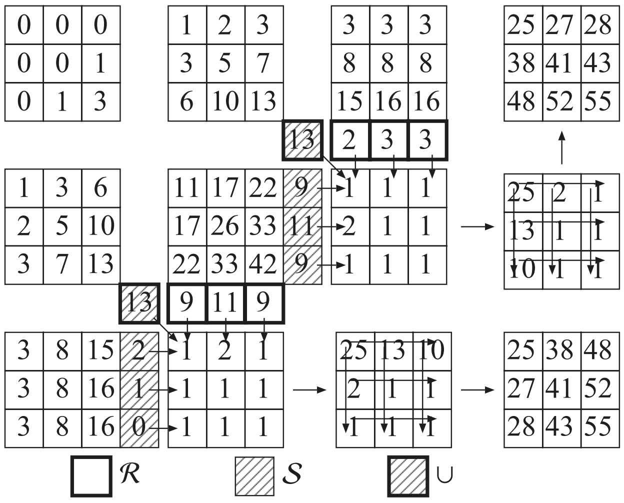

Figure 6.8: Stage 3 of 1R1W SAT algorithm for a matrix in Figure 6.1

this task. Since we have $\frac{n}{w^2}$ blocks, $n+(2w+1)\times\frac{n}{w^2}$ elements are read and $n+(2w+1)\times\frac{n}{w^2}$ elements are written in all stages. Barrier synchronization is necessary after each of Stages from 0 to $2\frac{\overline{n}}{w}\quad 3$. Thus, we have,

Theorem 6.4.2 *The global memory access cost of 4R1W SAT algorithm is* $2\frac{n}{w}+O(\frac{n}{w^2})+(2\frac{\overline{n}}{w}\quad 2)l$.

6.5 (1 + *r*)R1W SAT algorithm

The main purpose of this chapter is to accelerate the SAT computation further by combining 1R1W and 2R1W SAT algorithms. The idea of further acceleration is to use 2R1W SAT algorithm in early and late stages of 1R1W SAT algorithms to reduce the latency overhead.

Again, suppose that a $\overline{n} \bullet \quad \overline{n}$ matrix a is partitioned into $\frac{\overline{n}}{w} \bullet \frac{\overline{n}}{w}$ blocks of size $w \bullet w$ each. As illustrated in Figure 6.9, for any fi ed parameter r ($0 < r < 1$), we partition blocks into (A) top left triangle, (B) bottom right triangle, and (C) remaining blocks. Clearly, (A) and (B) have $\frac{\sqrt{r}\sqrt{n}}{w} + (\frac{\sqrt{r}\sqrt{n}}{w} \quad 1) + \times\times\times + 1 = \frac{\sqrt{r}\sqrt{n}}{w} \times (\frac{\sqrt{r}\sqrt{n}}{w} + 1)/2 \leftarrow \frac{rn}{2w^2}$ blocks each. We firs use 2R1W SAT algorithm to compute the SAT of (A). After that, we use 1R1W SAT algorithm for (C). Finally, 2R1W SAT algorithm is used for the computation of the SAT in (B).

Let us evaluate the performance. Since (A) and (B) have approximately $\frac{rn}{2}$ elements in $\frac{rn}{2w^2}$ blocks each, 2R1W SAT algorithm for (A) and (B) performs $rn + O(\frac{rn}{w})$ read operations and $\frac{rn}{2} + O(\frac{rn}{w})$ write operations each. Also, since (C) has $(1 \quad r)n$ elements, 1R1W SAT algorithm for (C) performs $(1 \quad r)n + O(\frac{(1 \ r)n}{w})$ read operations and $(1 \quad r)n + O(\frac{(1 \ r)n}{w})$ write operations. Hence, this SAT algorithm performs $(1+r)n + O(\frac{n}{w})$ read operations and $n + O(\frac{n}{w})$ write operations. Thus, we call this SAT algorithm $(1+r)$R1W SAT algorithm. Further, 2R1W SAT algorithm for (A) and (B) needs $2+2d$ barrier synchronization steps each, where d is the depth of the recursion of 2R1W SAT algorithm. Since 1R1W SAT algorithm for (C) has $2\frac{\sqrt{(1 \ r)}\sqrt{n}}{w} \quad 1$ stages, it needs $2\frac{\sqrt{(1 \ r)}\sqrt{n}}{w} \quad 2$ barrier synchronization steps. Also, after the computation of the SAT for (A) and (B), 1 barrier synchronization steps each is necessary. Totally, $(1 + r)$R1W SAT algorithm executes $2\frac{\sqrt{(1 \ r)}\sqrt{n}}{w} + 4 + 4d$ barrier synchronization steps. Thus, we have,

Theorem 6.5.1 *The global memory access cost of* $(1 + r)$*R1W SAT algorithm is* $(2 + r)\frac{n}{w} + O(\frac{n}{w^2}) + (2\frac{\sqrt{(1 \ r)}\sqrt{n}}{w} + 5 + 4d)l$.

When $r = 0.25$, the global memory access cost of 1.25R1W SAT algorithm is $2.5\frac{n}{w} + (\frac{\sqrt{n}}{w} + 5 + 4d)l$ time units. Since 2R1W and 1R1W SAT algorithms run approximately $3\frac{n}{w} + (3 + 2d)l$ and $2\frac{n}{w} + 2\frac{\sqrt{n}}{w}l$ time units, respectively, 1.25R1W SAT algorithm may run faster than these algorithms. Further, we can select the best value r that minimize the running time of $(1 + r)$R1W SAT algorithm.

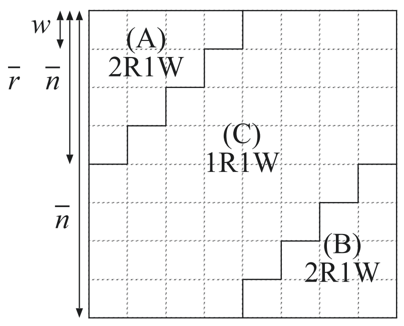

Figure 6.9: Partition of a matrix for $(1 + r)$R1W SAT algorithm

6.6 Experimental results

We have implemented all SAT algorithms presented so far in this chapter on GeForce GTX 780 Ti.

Since the number of memory banks and the number of threads in a warp is 32 [5], we have implemented SAT algorithms with $w = 32$. Barrier synchronization of all threads is implemented by invoking separated CUDA kernel calls. For example, one CUDA kernel call is invoked for each of $2\frac{\overline{n}}{w}$ 1 stages of 4R1W SAT algorithm.

We have tested several configuratio in terms of the number of threads in a CUDA block, and selected the best configuration For example, in 2R2W, 4R4W, and 4R1W SAT algorithms, CUDA blocks with 64 threads each are invoked. Since each Stage i and each Stage $2\frac{\overline{n}}{w}$ 1 i $(0 \geq i \geq \frac{\overline{n}}{w}$ 1) of 4R1W SAT algorithm computes $i + 1$ values of the SAT, it uses $i + 1$ threads in $\frac{i+1}{64}$ CUDA blocks. In 2R1W and 1R1W SAT algorithm, a $32 \bullet 32$ block in a matrix is copied to the shared memory in a streaming multiprocessor and the column-wise sums, the row-wise sums, and/or the SAT of it is computed. After that, the resulting values are copied to the global memory. For this operation, we use one CUDA block with 128 threads for each $32 \bullet 32$ block. All 128 threads in a CUDA block are used to copy a $32 \bullet 32$ block in the shared memory and

32 threads out of 128 threads are used to compute the column-wise sums, the row-wise sums, and/or the SAT.

Table 6.2 shows the running time of SAT algorithms for a double (64-bit) matrix of size from 1K• 1K (= 1024 • 1024) to 18K• 18K (= 18432 • 18432). Since a 18K• 18K 64-bit matrix uses 2.53GBytes, it is hard to store a matrix larger than it in the global memory of GeForce GTX 780 Ti of size 3GBytes. The running time of the best SAT algorithm for each value of $\overline{n}$ is highlighted in boldface. Since 4R1W SAT algorithm performs a lot of kernel calls and stride memory access, and has large memory access latency overhead, it needs much more computing time than the other algorithms. Recall that 4R4W SAT algorithm corresponds to 2R2W SAT algorithm with transpose and 4R4W SAT algorithm performs much more memory access operations than 2R2W SAT algorithm. Since 2R2W SAT algorithm performs stride memory access, it is much slower than 4R4W SAT algorithm. These experimental results imply that stride memory access imposes a large penalty on the computing time.

Recall that 2R1W and 1R1W SAT algorithms are block-based algorithms, that perform $3n + O(\frac{n}{w})$ and $2n + O(\frac{n}{w})$ global memory access operations, respectively. Hence, they are faster than 4R4W SAT algorithm, which performs approximately $8n$ global memory access operations. Although 1R1W SAT algorithm performs fewer global memory access operations than 2R1W SAT algorithm, it runs slower when $\overline{n} \geq 6$K. The reason is that 1R1W SAT algorithm has a larger latency overhead than 2R1W SAT algorithm and the latency overhead dominates the bandwidth overhead when the size of input is small. 1.25R1W SAT algorithm runs faster than both 2R1W and 1R1W SAT algorithms whenever $\overline{n} \approx 5$K. We have evaluated the computing time for all possible values of r to fin the best value r that minimize the running time of $(1 + r)$R1W.

Table 6.2: The running time of SAT algorithm (in milliseconds) and the value of r that minimize the running time of $(1 + r)$R1W SAT algorithm for matrices of sizes from 1K● 1K to 18K● 18K

SAT Algorithms	1K	2K	3K	4K	5K	6K	7K	8K	10K	12K	14K	16K	18K
2R2W	1.47	3.28	5.71	9.53	13.6	23.9	27.1	47.8	90.8	163	160	234	401
4R4W	1.07	2.52	4.48	6.77	9.67	13.7	17.2	22.2	33.9	50.4	64.2	83.1	117
4R1W	11.5	22.9	36.4	50.1	113	104	173	252	315	597	437	742	1600
2R1W	**0.332**	**0.850**	**1.83**	**3.09**	4.79	6.78	9.25	12.3	18.9	27.2	36.8	48.7	61
1R1W	0.902	1.46	2.43	3.65	5.05	6.81	8.71	10.9	16.2	22.6	29.7	38	53.8
1.25R1W	0.453	1.05	1.96	3.25	4.71	6.41	8.47	10.8	16.5	23	31.2	40.7	57.6
fastest $(1 + r)$R1W	0.365	0.958	1.94	3.16	**4.58**	**6.32**	**8.25**	**10.5**	**15.7**	**22.0**	**29.1**	**37.5**	**53.1**
r $(0 < r < 1)$	0.168	0.174	0.172	0.159	0.136	0.123	0.0876	0.103	0.0963	0.0710	0.0835	0.0694	0.0725
2R2W(CPU)	25.9	107	241	427	670	966	1310	1690	2670	3850	5250	6760	8670
4R1W(CPU)	18.0	73.2	165	293	459	660	904	1160	1830	2660	3600	4590	5950

Table 6.2 also shows the values of r $(0 < r < 1)$ that minimize the running time of $(1 + r)$R1W SAT algorithm. From the table, we can see that $(1 + r)$R1W SAT algorithm attain the best performance when $\overline{n} \approx$5K. Also, the value of r that gives the best performance decreases as the size of a matrix increases. This is because the memory bandwidth overhead of 1R1W SAT algorithm dominates the latency overhead for larger matrices and 1R1W SAT algorithm has better performance than 2R1W algorithm. We can conjecture that 1R1W SAT algorithm could be the best if an input matrix was much larger than 18K● 18K.

To see a speed-up factor of SAT algorithms running on the GPU over a conventional CPU, we have evaluated the performance of several sequential SAT algorithms on Intel Xeon X7460 (2.66GHz). Table 6.2 shows the running time of top two sequential algorithms as follows:

2R2W(CPU): The column-wise prefix-sum are computed in a raster scan order from the top row to the bottom row. More specificall , $a[i+1][j]$ ′ $a[i+1][j] + a[i][j]$ is executed in a raster scan order of (i, j). The row-wise prefix-sum are also computed in a raster scan order, that is, $a[i][j+1]$ ′ $a[i][j+1] + a[i][j]$ is executed in a raster scan order of (i, j).

4R1W(CPU): Formula (6.1) is evaluated in a raster scan order of (i, j).

From the table, we can see that 4R1W(CPU) SAT algorithm runs faster than 2R2W(CPU) SAT algorithm, because of the memory access locality.

Also, $(1+r)$R1W SAT algorithm runs more than 100 times faster than 4R1W(CPU) SAT algorithm when $\overline{n} \approx$5K.

6.7 Conclusion

The main contribution of this chapter is to propose SAT algorithms optimized for GPU. We have also presented a global-memory-access-optimal parallel algorithm for computing the summed area table on the asynchronous HMM. The experimental results on GeForce GTX 780 Ti show that our best algorithm, $(1+r)$R1W SAT algorithm, runs faster than any other algorithms for an input matrix of size 5K• 5K or larger. It also runs at least 100 times faster than the best sequential algorithm running on a single CPU.

Chapter 7

Conclusions

7.1 Summary

In this dissertation, we have presented several algorithms which are optimal from the theoretical point of view. These algorithms have no bank-conflic on the shared memory and perform coalesced access to the global memory to maximize the bandwidth. We have also evaluated several parallel algorithms on the GPU and compared performance of algorithm.

Offline permutation is a task to move data along a permutation given beforehand. The conventional algorithm which performs $b[P(i)]$ ′ $a[i]$ for all i $(0 \geq i \geq n \quad 1)$. Since this algorithm involves many bank conflict on the shared memory and performs a lot of stride access, this algorithm has lower performance for the shared memory and the global memory. In chapter 4, we evaluate the several permutation algorithms and compare these performances of read/write access. Our algorithms are scheduled memory access using bipartite graph coloring to avoid the bank-conflict We have implemented several permutation algorithms including our conflict-fre permutation algorithm on the shared memory of NVIDIA GeForce GTX-680. The experimental results for 1024 64-bit numbers on NVIDIA GeForce GTX-680 show that the destination-designated permutation algorithm takes 247.8 ns for the random permutation and 1684ns for the worst permutation that involves the maximum bank conflicts Our conflict-fre permutation algorithm runs in 167ns for any permutation including the random permutation and the worst permutation, although it performs more memory accesses.

Similarly, we have presented an optimal offline permutation algorithm on the HMM in chapter 5. Our scheduled offline permutation algorithm on the HMM performs three step permutations, row-wise permutation, column-wise permutation, and row-wise permutation, each of which is performed in DMMs of the HMM in parallel. We have also implemented the optimal offline permutation algorithm and the conventional algorithms on GeForce GTX-680 GPU and evaluated their performance. The experimental results showed that our optimal offline permutation algorithm is faster than the conventional permutation algorithm for most cases.

Finally, We have presented a global-memory-access-optimal parallel algorithm for computing the summed area table on the asynchronous HMM. The experimental results on GeForce GTX 780 Ti showed that our best algorithm, $(1 + r)$R1W SAT algorithm, runs faster than any other algorithms for an input matrix of size 5K$\bullet$5K or larger. It also runs at least 100 times faster than the best sequential algorithm running on a single CPU.

References

[1] A. V. Aho, J. D. Ullman, and J. E. Hopcroft. *Data Structures and Algorithms*. Addison Wesley, 1983.

[2] J. Bai, Q. Song, O. Veksler, and X. Wu. Fast dynamic programming for labeling problems with ordering constraints. In *Computer Vision and Pattern Recognition (CVPR), 2012 IEEE Conference on*, pages 16–21, June 2012.

[3] K. E. Batcher. Sorting networks and their applications. In *Proc. of AFIPS Spring Joint Computer Conference*, pages 307–314, 1968.

[4] N. Corporation. *NVIDIA CUDA C best practice guide version 5.0*. 2012.

[5] N. Corporation. *NVIDIA CUDA C programming guide version 5.0*. 2012.

[6] N. Corporation. *NVIDIA GeForce GTX680 GPU whitepaper*. 2012.

[7] F. Crow. Summed-area tables for texture mapping. In *Proc. of the 11th annual conference on Computer graphics and interactive techniques*, number 207–212, 1984.

[8] M. Daga, T. Scogland, and F. Wu. Architecture-aware mapping and optimization on a 1600-core gpu. In *Parallel and Distributed Systems (ICPADS), 2011 IEEE 17th International Conference on*, pages 316–323, December 2011.

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

[10] A. Gibbons and W. Rytter. *Efficient Parallel Algorithms*. Cambridge University Press, 1988.

[11] J. E. Hopcroft and R. M. Karp. An $n^{5/2}$ algorithm for maximum matchings in bipartite graphs. *SIAM Journal on Computing*, 2(4):225–231, 1973.

[12] S. H. Hsiao and C. Y. R. Chen. Performance evaluation of circuit switched multistage interconnection networks using a hold strategy. *IEEE Transactions on Parallel and Distributed Systems*, pages 632–640, September 1992.

[13] W. W. Hwu. *GPU Computing Gems Emerald Edition*. Morgan Kaufmann, 2011.

[14] Y. Ito and K. Nakano. A GPU implementation of dynamic programming for the optimal polygon triangulation. *IEICE Transactions on Information and Systems*, E96-D(12):2596–2603, December 2013.

[15] R. Landaverde, T. Zhang, A. K. Coskun, and M. Herbordt. An investigation of unifie memory access performance in cuda. In *IEEE High Performance Extreme Computing Conference (HPEC)*, 2014.

[16] A. Lauritzen. *"Chapter 8: Summed-area variance shadow maps" in GPU Gems 3*. Addison-Wesley, 2007.

[17] M. Li and P. M. Vitanyi. *An Introduction to Kolmogorov Complexity and Its Applications, 3rd Edition*. Springer, 2008.

[18] D. Man, Y. I. K. Uda, and K. Nakano. A GPU implementation of computing euclidean distance map with efficient memory access. In *Proc. of International Conference on Networking and Computing*, pages 68–76, December 2011.

[19] 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(2):260–276, July 2011.

[20] R. Motwani and P. Raghavan. *Randomized Algorithms*. Cambridge University Press, 1995.

[21] K. Nakano. Optimal sorting algorithms on bus-connected processor arrays. *IEICE Transaction Fundamentals*, E76-A(11):2008–2015, November 1993.

[22] K. Nakano. Asynchronous memory machine models with barrier syncronization. In *Proc. of International Conference on Networking and Computing*, pages 58–67, December 2012.

[23] K. Nakano. An optimal parallel prefix-sum algorithm on the memory machine models for GPUs. In *Proc. of International Conference on Algorithms and Architectures for Parallel Processing (ICA3PP, LNCS7439)*, pages 99–113, September 2012.

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

[25] K. Nakano. The hierarchical memory machine model for GPUs. In *Proc. of International Parallel and Distributed Processing Symposium Workshops*, pages 591–600, May 2013.

[26] K. Nakano. Optimal parallel algorithms for computing the sum, the prefix-sums and the summed area table on the memory machine models. *IEICE Transaction on Information and Systems*, E96-D(12), 2013. 2626–2634.

[27] K. Nakano. Simple memory machine models for GPUs. *International Journal of Parallel, Emergent and Distributed Systems*, 29(1):17–37, 2014.

[28] D. Nehab, A. Maximo, R. S. Lima, and H. Hoppe. GPU-efficient recursive filterin and summed-area tables. *ACM Transaction Graphics*, 2011.

[29] N. 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*, pages 320–326, December 2011.

[30] K. Ogawa, Y. Ito, and K. Nakano. Efficient canny edge detection using a GPU. In *Proc. of International Conference on Networking and Computing*, pages 279–280, 2010.

[31] J. D. S. Parker. Notes on shuffle/exchange-type switching networks. *IEEE Transaction on Computers*, C-29(3):213–222, March 1980.

[32] D. P. Playne and K. A. Hawick. Job parallelism using graphical processing unit individual multi-processors and localised memory. In *Proc. 19th Int. Conf. on Parallel and Distributed Processing Techniques and Applications (PDPTA)*, July 2013.

[33] M. Poostchi, K. Palaniappan, F. Bunyak, M. Becchi, and G. Seetharaman. Efficient gpu implementation. In *ACCV'12 Proceedings of the 11th international conference on Computer Vision*, volume 1, pages 266–278, 2012.

[34] H. S. Stone. Parallel processing with the perfect shuffle. *IEEE Transaction on Computers*, C-20(2):153–161, February 1971.

[35] 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*, pages 153–159, December 2011.

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

Acknowledgment

First and foremost, I would like to express my sincere gratitude to my adviser, Professor Koji Nakano for his continuous encouragement, advice and support. His knowledge and research experience are in value through the whole period of my Ph.D. study. He is acknowledged, in particular, for his kindness, generous guidance and illuminating discussions. As an advisor, he taught me practices and skills that will benefi my future academic career.

I also wish to express my heartful thanks to Associate Professor Yasuaki Ito for his invaluable help throughout the study. My heartiest thanks go to all members of computer system laboratory. They were always kind and very keen to help.

I am grateful to all the faculty members and staffs of the Department of Information Engineering, Hiroshima University.

Last but not least, I wish to express my thanks to my parents who has always supported me.

List of publications

Journals

[J-1] Akihiko Kasagi, Koji Nakano, and Yasuaki Ito, Offline Permutation Algorithms on the Discrete Memory Machine with Performance Evaluation on the GPU, IEICE Transactions on Information and Systems, Vol. E96-D, No. 12, pp. 2617–2625, December 2013.

[J-2] Akihiko Kasagi, Koji Nakano, and Yasuaki Ito, Offline Permutation on the CUDA-enabled GPU, IEICE Transactions on Information and Systems Vol. E97-D, No. 12, pp. 3052–3062, December 2014.

International Conferences

[I-1] Akihiko Kasagi, Koji Nakano and Yasuaki Ito, An Implementation of Conflict Free Offline Permutation on the GPU, Proc. of International Conference on Networking and Computing (ICNC), pp. 226–232, December 2012.

[I-2] Akihiko Kasagi, Koji Nakano, and Yasuaki Ito, An Optimal Offline Permutation Algorithm on the Hierarchical Memory Machine, with the GPU implementation, Proc. of International Conference on Parallel Processing, pp. 1–10, October, 2013.

[I-3] Akihiko Kasagi, Koji Nakano, Yasuaki Ito, Parallel Algorithms for the Summed Area Table on the Asynchronous Hierarchical Memory Machine, with GPU implementations, Proc. of International Conference on Parallel Processing, pp.251–260, Septembe, 2014.