IEICE TRANSACTIONS

on Information and Systems

VOL. E97-D NO. 12
DECEMBER 2014

A PUBLICATION OF THE INFORMATION AND SYSTEMS SOCIETY
The Institute of Electronics, Information and Communication Engineers
Kikai-Shinko-Kaikan Bldg., 5-8, Shibakoen 3 chome, Minato-ku, TOKYO, 105-0011 JAPAN

IEICE TRANS. INF. & SYST., VOL.E97–D, NO.12 DECEMBER 2014

PAPER Special Section on Parallel and Distributed Computing and Networking

Offline Permutation on the CUDA-enabled GPU

Akihiko KASAGI[†], ***Student Member***, **Koji NAKANO**[†a)], ***and*** **Yasuaki ITO**[†], ***Members***

SUMMARY The Hierarchical Memory Machine (HMM) is a theoretical parallel computing model that captures the essence of computation on CUDA-enabled GPUs. The offline permutation is a task to copy numbers stored in an array a of size n to an array b of the same size along a permutation P given in advance. A conventional algorithm can complete the offline permutation by executing $b[p[i]] \leftarrow a[i]$ for all i in parallel, where an array p stores the permutation P. We first present that the conventional algorithm runs $D_w(P) + 2\frac{n}{w} + 3L - 3$ time units using n threads on the HMM with width w and latency L, where $D_w(P)$ is the distribution of P. We next show that important regular permutations including transpose, shuffle, and bit-reversal permutations run $2\frac{n}{w} + 2\frac{n}{kw} + 2L - 2$ time units on the HMM with k DMMs. We have implemented permutation algorithms for these regular permutations on GeForce GTX 680 GPU. The experimental results show that these algorithms run much faster than the conventional algorithm. We also present an offline permutation algorithm for any permutation running in $16\frac{n}{w} + 16\frac{n}{kw} + 16L - 16$ time units on the HMM with k DMMs. Quite surprisingly, our offline permutation algorithm on the GPU achieves better performance than 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.

key words: *memory machine models, offline permutation, GPU, CUDA*

1. Introduction

Offline permutation is a task to move numbers along a permutation given beforehand. More specifically, for given two arrays a and b of size n, and a permutation P, the value of each $a[i]$ $(0 \leq i \leq n-1)$ is copied to $b[P(i)]$. A conventional algorithm can complete the offline permutation by executing $b[p[i]] \leftarrow a[i]$ for all i $(0 \leq i \leq n-1)$ in parallel, where an array p stores the permutation P. 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 [1]. Sorting network such as bitonic sorting [2], [3] 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. If a parallel algorithm performs offline permutations frequently, the acceleration of offline permutations has a large impact. Some algorithms frequently execute offline permutation. For example, bitonic merging [3] can be implemented using the perfect shuffle permutation [4] and the compare-exchange of adjacent values. The implementation repeatedly performs the alternation of data movement along the perfect shuffle permutation and the compare-exchange. Since the compare-exchange of adjacent values is a light-weight task with conflict-free memory access, the acceleration of perfect shuffle permutation will give a large impact on the running time of the bitonic sorting.

The main purose of this paper is to show an optimal algorithm for offline permutation on the GPU. *The GPU* (Graphics Processing Unit), is a specialized circuit designed to accelerate computation for building and manipulating images [5]–[7]. Latest GPUs are designed for general purpose computing and can perform computation in applications traditionally handled by the CPU. Hence, GPUs have recently attracted the attention of many application developers [5]. NVIDIA provides a parallel computing architecture called *CUDA* (Compute Unified Device Architecture) [8], 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 [9], since they have hundreds of processor cores and very high memory bandwidth.

In our previous paper [10], we have presented a conflict-free offline permutation algorithm running in $O(\frac{n}{w} + \frac{nl}{p} + l)$ time units using p threads on the Discrete Memory Machine (DMM) with width w and latency l, which is a theoretical model for computation using the shared memory of a streaming processor on a CUDA-enabled GPU. Later, we have implemented the conventional offline permutation algorithm and this conflict-free permutation algorithm on a single streaming multiprocessor of GeForce GTX 680 GPU and evaluated the performance [11]. The experimental results showed that the conventional permutation algorithm and the conflict-free permutation algorithm run in 246ns and in 165ns, respectively, for the random permutation of 1024 float (32-bit) numbers. Hence, the conflict-free 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 float (32-bit) numbers.

We first show that an optimal permutation algorithm for larger arrays on the global memory of the Hierachical Mem-

Manuscript received December 26, 2013.
Manuscript revised May 10, 2014.

[†]The authors are with the Department of Information Engineering, Hiroshima University, Higashihiroshima-shi, 739–8527 Japan.
a) E-mail: nakano@cs.hiroshima-u.ac.jp

DOI: 10.1587/transinf.2014PAP0010

ory Machine (HMM), which is a theoretial model for computation using all streaming processors of a CUDA-enabled GPU. It 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. Each step uses a conflict-free offline permutation algorithm presented in our previous paper [11]. It runs $16\frac{n}{w} + 16\frac{n}{kw} + 16L - 16$ 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 $D_w(P) + 2\frac{n}{w} + 3L - 3$ 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 $3\frac{n}{w} + 3L - 3$ and $n + 2\frac{n}{w} + 3L - 3$ 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 factor 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 offline 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 $n \geq 256\text{K}$ ($= 2^{18}$). For example, our offline permutation algorithm runs in 780ms for any permutation of 4M ($= 2^{22}$) float (32-bit) numbers. The conventional algorithm takes 2328ms for the bit-reversal permutation.

We also show that some regular permutations can be done efficiently. More specifically, we present permutation algorithms for transpose, shuffle, and bit-reversal permutations on the HMM, which run $4\frac{n}{w} + 2L - 2$ time units. We have implemented these algorithms on the GPU and evaluate the performance on GeForce GTX 680. Our implementation runs in 157ms for bit-reversal permutations of 4M ($= 2^{22}$) float (32-bit) numbers, which is 14.8 times faster than the conventional algorithm.

This paper is organized as follows. First, we briefly explain CUDA architecture and define three memory machines, DMM, UMM, and HMM in Sect. 2. In Sect. 3, we define three memory access operations, casual memory access, coalesced memory access, and conflict-free memory access and evaluate the running time. Section 4 defines the offline permutation and shows two conventional permutation algorithms, destination-designated permutation algorithm and source-designated permutation algorithm. Section 5 presents permutation algorithms for transpose, shuffle, and bit-reversal permutations. Section 6 shows algorithms for row-wise permutation and column-wise permutation of a matrix. In Sect. 7, we present our scheduled permutation algorithm and show the optimality. Finally, Sect. 8 shows experimental results for comparing the conventional permutation algorithms and our scheduled permutation algorithm. Section 9 concludes our work.

2. CUDA Architecture and the Memory Machine Models

The main purpose of this section is to explain CUDA architecture [8] and to define three memory machine models: the Discrete Memory Machine (DMM), the Unified Memory Machine (UMM), and the Hierarchical Memory Machine (HMM).

NVIDIA GPUs have streaming multiprocessors (SMs) each of which executes multiple threads in parallel. CUDA uses two types of memories of the NVIDIA GPUs: *the shared memory* and *the global memory* [8]. 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 conflict* of the shared memory access and *the coalescing* of the global memory access [6], [9], [12], [13]. The address space of the shared memory is mapped into several physical memory banks. If two or more threads access the same memory banks at the same time, the access requests are processed in turn. Hence, to maximize the memory access performance, threads of CUDA should access distinct memory banks to avoid the bank conflicts of the memory accesses. To maximize the bandwidth between the GPU and the DRAM chips, the consecutive addresses of the global memory must be accessed at the same time. Thus, CUDA threads should perform coalesced access when they access the global memory.

In our previous paper [10], we have introduced two models, *the Discrete Memory Machine (DMM)* and *the Unified Memory Machine (UMM)*, which extract the essential features of the shared memory and the global memory of CUDA-enabled GPUs. Since the DMM and the UMM are promising as theoretical computing models for GPUs, we have published several efficient algorithms on the DMM and the UMM [14]–[17]. The DMM and the UMM have three parameters: the number p of threads, width w, and memory access latency l. Figure 1 illustrates the outline of the architectures of the DMM and the UMM with $p = 20$ threads and width $w = 4$. The p threads are partitioned into $\frac{p}{w}$ groups of w threads each called *warp*. The $\frac{p}{w}$ warps are dispatched for memory access in turn, and w threads in a dispatched warp

send memory access requests to the memory banks (MBs) through the memory management unit (MMU). We do not discuss the architecture of the MMU, but we can think that it is a multistage interconnection network [18] in which memory access requests are moved to destination memory banks in a pipeline fashion. Note that the DMM and the UMM with width w has w memory banks and each warp has w threads. For example, the DMM and the UMM in Fig. 1 have 4 threads in each warp and 4 MBs.

Let us define *the Discrete Memory Machine (DMM)* of width w and latency l. Let $m[i]$ $(i \geq 0)$ denote a memory cell of address i in the memory. Let $B[j] = \{m[j], m[j + w], m[j + 2w], m[j + 3w], \ldots\}$ $(0 \leq j \leq w - 1)$ denote *the j-th bank* of the memory. Clearly, a memory cell $m[i]$ is in the (i mod w)-th memory bank. We assume that memory cells in different banks can be accessed in a time unit, but no two memory cells in the same bank can be accessed in a time unit. Also, we assume that l time units are necessary to complete an access request and continuous requests are processed in a pipeline fashion through the MMU.

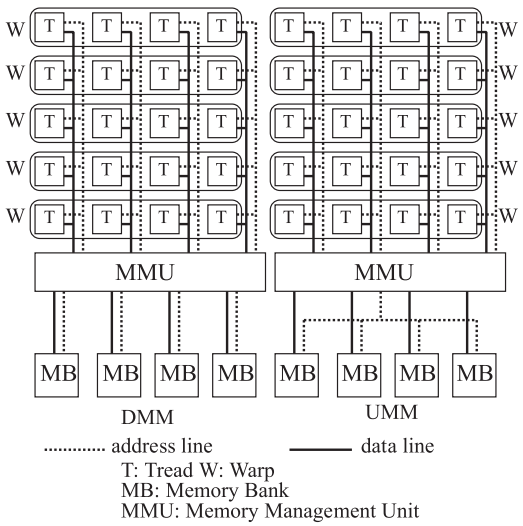

Fig. 1 The architectures of the DMM and the UMM with width $w = 4$.

Recall that p threads are partitioned into $\frac{p}{w}$ groups of w threads called warps. More specifically, p threads $T(0)$, $T(1)$, ..., $T(p-1)$ are partitioned into $\frac{p}{w}$ warps $W(0), W(1)$, ..., $W(\frac{p}{w}-1)$ such that $W(i) = \{T(i \cdot w), T(i \cdot w+1), \ldots, T((i+1) \cdot w - 1)\}$ $(0 \leq i \leq \frac{p}{w} - 1)$. Warps are dispatched for memory access in turn, and w threads in a warp try to access the memory at the same time. In other words, $W(0), W(1), \ldots, W(\frac{p}{w} - 1)$ are dispatched in a round-robin manner if at least one thread in a warp requests memory access. 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.

We next define *the Unified Memory Machine (UMM)* of width w as follows. Let $A[j] = \{m[j \cdot w], m[j \cdot w + 1], \ldots, m[(j+1) \cdot w - 1]\}$ denote the j-th address group. We assume that memory cells in the same address group are processed 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 2 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 $\langle m[7], m[5], m[15], m[0]\rangle$ and $\langle m[10], m[11], m[12], m[9]\rangle$, respectively. In the DMM, memory access requests by $W(0)$ are separated into two pipeline stages, because $m[7]$ and $m[15]$ are in the same bank $B(3)$. Those by $W(1)$ occupy one stage, because all requests are in distinct banks. Thus, the memory requests occupy three stages, it takes $3 + 5 - 1 = 7$ time units to complete the memory access. In the UMM,

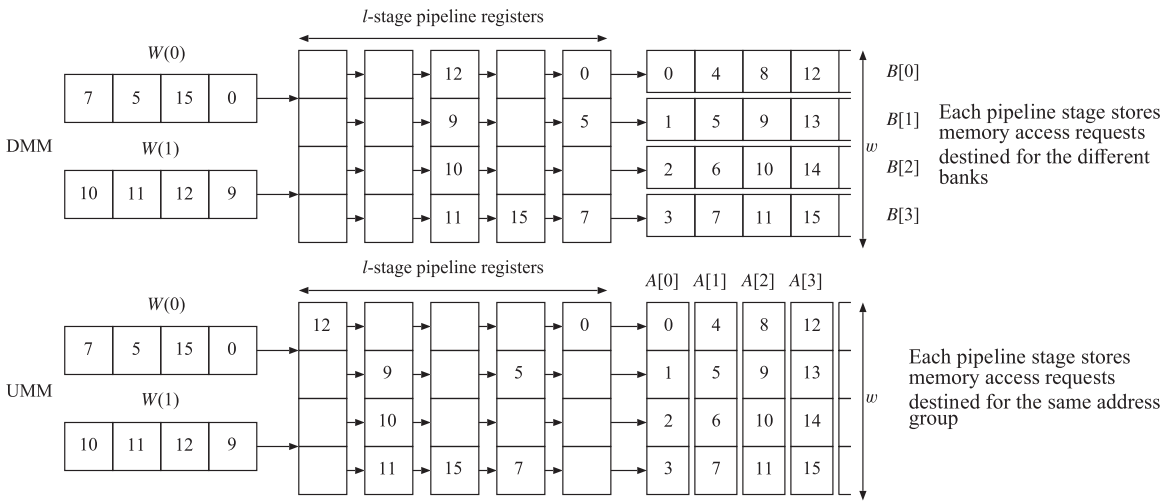

Fig. 2 Examples of memory access on the DMM and the UMM.

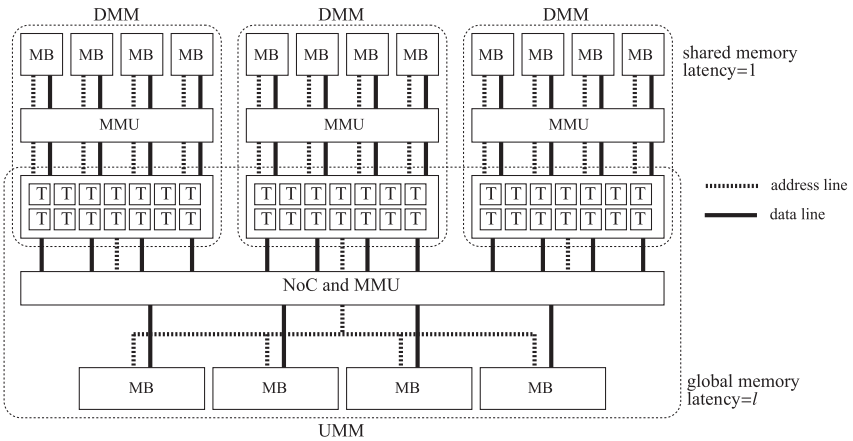

Fig. 3 The architecture of the HMM with $k = 3$ DMMs and width $w = 4$.

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.

Quite Recently, we have introduced *the Hierarchical Memory Machine (HMM)* [19], [20], which is a hybrid of the DMM and the UMM. The HMM is a more practical parallel computing model that extracts the architecture of GPUs. Figure 3 illustrates the architecture of the HMM. The HMM consists of multiple DMMs and a single UMM. 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* after those of 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. If multiple DMMs try to access the global memory, they are dispatched in turn. Thus, it makes sense that the global memory also has w banks. The latency of the shared memory of NVIDIA GPUs is several clock cycles. On the other hand, that of the global memory is several hundred clock cycles [8]. Hence, for simplicity, we assume that those of the HMM are 1 and L, respectively, although we may use parameter l to denote the latency of the shared memory access [20].

3. Coalesced, Conflict-Free, and Casual Memory Access

This section first defines a round of memory access by threads. We also define 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 $b[p[i]] \leftarrow a[i]$ involves one reading round for a and p each, and one writing round for b.

Next, we define coalesced and conflict-free 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-free* if all memory access by a warp destined for the distinct memory banks of the shared memory. More specifically, a round of the memory access by a warp is *coalesced* if $\lfloor \frac{m(0)}{w} \rfloor = \lfloor \frac{m(1)}{w} \rfloor = \cdots = \lfloor \frac{m(w-1)}{w} \rfloor$, where $m(i)$ $(0 \leq i \leq w-1)$ is the address accessed by thread $T(i)$ in the warp. A round of the memory access by a warp is *conflict-free* if, for all pair i and j $(0 \leq i < j \leq w-1)$, $m(i) = m(j)$ or $m(i) \not\equiv m(j) \pmod{w}$. 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-free* 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-free 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-free 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 $L-1$ time units are necessary to complete the memory requests by the last warp. Thus, it takes $\frac{n}{w} + L - 1$ time units to complete a round of coalesced memory access by n threads. Similarly, a round of conflict-free 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 1: A round of coalesced memory access for the global memory and that of conflict-free memory access for

Table 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-free reading	conflict-free writing	
D-designated permutation	-	1	2	-	-	-	$D_w(P)+2\frac{n}{w}+3L-3$
S-designated permutation	1	-	1	1	-	-	$D_w(P^{-1})+2\frac{n}{w}+3L-3$
Transpose/Shuffle/Bit-reversal	-	-	1	1	1	1	$2\frac{n}{w}+2\frac{n}{kw}+2L-2$
Row-wise permutation	-	-	3	1	2	2	$4\frac{n}{w}+4\frac{n}{kw}+4L-4$
Column-wise permutation	-	-	5	3	4	4	$8\frac{n}{w}+8\frac{n}{kw}+8L-8$
Our scheduled permutation	-	-	11	5	8	8	$16\frac{n}{w}+16\frac{n}{kw}+16L-16$

the shared memory by n threads take $\frac{n}{w}+L-1$ 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 $n+L-1$ time units and n time units, respectively.

4. Offline Permutation and Conventional Algorithms

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

[Destination-designated permutation algorithm]
for $i\leftarrow 0$ to $n-1$ do
  $T(i)$ performs $b[p[i]]\leftarrow a[i]$

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 2: The D-designated permutation algorithm performs the offline permutation by memory access rounds in Table 1.

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 $(0\le i\le n-1)$. Suppose that $P^{-1}(0),P^{-1}(1),\ldots,P^{-1}(n-1)$ are stored in an array q of size n, such that $q[i]=P^{-1}(i)$ for all i $(0\le i\le n-1)$. The following algorithm can perform the offline permutation:

[Source-designated permutation algorithm]
for $i\leftarrow 0$ to $n-1$ do
  $T(i)$ performs $b[i]\leftarrow a[q[i]]$

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

Lemma 3: The S-designated permutation algorithm performs the offline permutation by memory access rounds in Table 1.

Let us define several important permutations that will be used to evaluate the performance of permutation algorithms by experiments on the GPU.
Let $n=2^m$ and $u_mu_{m-1}\cdots u_1$ be the binary representation of an integer u.
Identical: Permutation such that $P(u)=u$ for every u $(0\le u\le n-1)$.
Random: One of all possible $n!$ permutations is selected uniformly at random.
Shuffle: Shuffle permutation is defined as $P(u_mu_{m-1}\cdots u_1)=u_{m-1}\cdots u_1u_m$. Shuffle permutation is used for shuffle exchanging in sorting networks [2], [3].
Bit-reversal: Bit-reversal permutation is defined as $P(u_mu_{m-1}\cdots u_1)=u_1\cdots u_{m-1}u_m$. Bit-reversal is used for data reordering in the FFT algorithms [1], [21]
Transpose: Suppose that a and b are matrix with dimension $\sqrt{n}\times\sqrt{n}$. 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, $P(i\cdot\sqrt{n}+j)=j\cdot\sqrt{n}+i$ for every i and j $(0\le i,j\le\sqrt{n}-1)$. In other words, $P(u_mu_{m-1}\cdots u_1)=u_{m/2}u_{m/2-1}\cdots u_1u_mu_{m-1}\cdots u_{m/2+1}$.

For later reference, we define *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 define the distribution $D_w(P)$ of a permutation P with respect to width w as follows:

$$D_w(P)=\sum_{j=0}^{\frac{n}{w}-1}|\{\lfloor\frac{P(j\cdot w)}{w}\rfloor,\lfloor\frac{P(j\cdot w+1)}{w}\rfloor,\ldots,\lfloor\frac{P((j+1)\cdot w-1)}{w}\rfloor\}|,$$

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 $D_w(P)+L-1$ time units. Similarly, the S-designated permutation algorithm for P takes $D_w(P^{-1})+L-1$ time units for reading from a. Thus, we have,

Lemma 4: The D-designated permutation algorithm and

the S-designated permutation algorithm for a permutation P take time units shown in Table 1.

Clearly, D_w(identical) = $\frac{n}{w}$ and D_w(shuffle) = D_w(shuffle^{-1}) = $2\frac{n}{w}$. Further, the values of D_w(bit-reversal), D_w(bit-reversal^{-1}), D_w(transpose), and D_w(transpose^{-1}) are n. Since the random permutation is not a fixed permutation, D_w(random) is not a constant value.

5. In-Place Permutation Algorithms for Transpose, Shuffle, and Bit-Reversal Permutations

The main purpose of this section is to show *in-place permutation algorithms* for transpose, shuffle, and bit-reversal permutations. Unlike the conventional algorithm, these algorithms does not use an array p. Hence, we call them in-place permutation algorithms. For simplicity, we show permutation algorithms for $n = 256$ numbers and width $w = 4$. The reader should have no difficulty to design these algorithms for any n and w.

We assume that $\frac{n}{w^2} = 16$ DMMs are used for the permutations. The outline of algorithms for the three permutations are very simple.

- Each DMM with $w^2 = 16$ threads is responsible to perform the permutation of $w^2 = 16$ numbers out of $n = 256$ numbers.
- The $w^2 = 16$ threads perform coalesced access to read $w^2 = 16$ numbers in the global memory and performs conflict-free access to write them in the shared memory (Step 1).
- The $w^2 = 16$ threads perform conflict-free access to read $w^2 = 16$ numbers in the shared memory and performs coalesced access to write them in the global memory (Step 2).

For the purpose of avoiding bank conflicts, we use a matrix α of size $w \times w$ in the shared memory of each DMM. We write each element in α such that $\alpha[i, j]$ $(0 \leq i, j \leq w-1)$, which is allocated in address $i \cdot w + (i+j) \bmod w$. We call such arrangement *the diagonal arrangement*. Clearly, $\alpha[i, j]$ is in bank $B[(i + j) \bmod w]$. Let α_i $(0 \leq i \leq 15)$ be such matrix on the shared memory of the i-th DMM. Also, let $T_i(j)$ be the i-th thread of the j-th DMM.

The transpose can be done by the following algorithm:

[Transpose permutation]
for $i_4i_3i_2i_1 \leftarrow 0$ to 15 do in parallel
 for $j_4j_3j_2j_1 \leftarrow 0$ to 15 do in parallel
  Step 1: $T_{i_4i_3i_2i_1}(j_4j_3j_2j_1)$ performs
   $a[i_4i_3j_4j_3i_2i_1j_2i_1] \rightarrow \alpha_{i_4i_3i_2i_1}[j_4j_3, j_2j_1]$
  Step 2: $T_{i_4i_3i_2i_1}(j_4j_3j_2j_1)$ performs
   $\alpha_{i_4i_3i_2i_1}[j_2j_1, j_4j_3] \rightarrow b[i_2i_1j_4j_3i_4i_3j_2j_1]$

Let us verify that this algorithm performs the transpose permutation correctly. For this purpose, we will trace the value stored in each $a[u_8u_7u_6u_5u_4u_3u_2u_1]$. In Step 1, $a[u_8u_7u_6u_5u_4u_3u_2u_1] \rightarrow \alpha_{u_8u_7u_4u_3}[u_6u_5, u_2u_1]$ is performed. In Step 2, $\alpha_{u_8u_7u_4u_3}[u_6u_5, u_2u_1] \rightarrow b[u_4u_3u_2u_1u_8u_7u_6u_5]$ is performed. Thus, the transpose permutation algorithm correctly stores the transpose of a in b. Next, let us confirm that the memory access to a and b is coalesced, and that to α is conflict-free. The least significant two bits of indexes of memory access to a and b are j_2j_1. Since width is $w = 4$, these memory access operations are coalesced. In Step 1, each $T_{i_4i_3i_2i_1}(j_4j_3j_2j_1)$ access $\alpha_{i_4i_3i_2i_1}[j_4j_3, j_2j_1]$ is in bank $B[(j_4j_3 + j_2j_1) \bmod 4]$. Thus, memory access by each warp is conflict-free. Similarly, memory access in Step 2 is conflict-free, because $\alpha_{i_4i_3i_2i_1}[j_2j_1, j_4j_3]$ is in bank $B[(j_2j_1 + j_4j_3) \bmod 4]$.

The shuffle permutation can be done by the following algorithm:

[Shuffle permutation]
for $i_4i_3i_2i_1 \leftarrow 0$ to 15 do in parallel
 for $j_4j_3j_2j_1 \leftarrow 0$ to 15 do in parallel
  Step 1: $T_{i_4i_3i_2i_1}(j_4j_3j_2j_1)$ performs
   $a[j_4i_4i_3i_2i_1j_3j_2j_1] \rightarrow \alpha_{i_4i_3i_2i_1}[j_4j_3, j_2j_1]$
  Step 2: $T_{i_4i_3i_2i_1}(j_4j_3j_2j_1)$ performs
   $\alpha_{i_4i_3i_2i_1}[j_2j_4, j_3j_1] \rightarrow b[i_4i_3i_2i_1j_4j_3j_1j_2]$

Again, we will trace the value of each $a[u_8u_7u_6u_5u_4u_3u_2u_1]$. In Step 1, $a[u_8u_7u_6u_5u_4u_3u_2u_1] \rightarrow \alpha_{u_7u_6u_5u_4}[u_8u_3, u_2u_1]$ is performed. In Step 2, $\alpha_{u_7u_6u_5u_4}[u_8u_3, u_2u_1] \rightarrow b[u_7u_6u_5u_4u_3u_2u_1u_8]$ is performed. Thus, the shuffle permutation algorithm is correct. Since the least significant two bits of indexes of memory access to a and b are j_2j_1 and j_1j_2, respectively, these memory access operations are coalesced. Similarly to the transpose permutation, memory access to $\alpha_{i_4i_3i_2i_1}[i_4i_3, i_2i_1]$ in Step 1 is conflict-free. Memory access in Step 2 is also conflict-free, because $\alpha_{i_4i_3i_2i_1}[j_2j_4, j_3j_1]$ is in bank $B[(j_2j_4 + j_3j_1) \bmod 4]$.

The bit-reversal permutation can be done by the following algorithm:

[Bit-reversal permutation]
for $i_4i_3i_2i_1 \leftarrow 0$ to 15 do in parallel
 for $j_4j_3j_2j_1 \leftarrow 0$ to 15 do in parallel
  Step 1: $T_{i_4i_3i_2i_1}(j_4j_3j_2j_1)$ performs
   $a[j_4j_3i_4i_3i_2i_1j_2j_1] \rightarrow \alpha_{i_4i_3i_2i_1}[j_4j_3, j_2j_1]$
  Step 2: $T_{i_4i_3i_2i_1}(j_4j_3j_2j_1)$ performs
   $\alpha_{i_4i_3i_2i_1}[j_2j_1, j_4j_3] \rightarrow b[j_3j_4i_1i_2i_3i_4j_1j_2]$

Similarly, we will trace the value of each $a[u_8u_7u_6u_5u_4u_3u_2u_1]$. In Step 1, $a[u_8u_7u_6u_5u_4u_3u_2u_1] \rightarrow \alpha_{u_6u_5u_4u_3}[u_8u_7, u_2u_1]$ is performed. In Step 2, $\alpha_{u_6u_5u_4u_3}[u_8u_7, u_2u_1] \rightarrow b[u_1u_2u_3u_4u_5u_6u_7u_8]$ is performed. Thus, the bit-reversal permutation algorithm is correct. Since the least significant two bits of indexes of memory access to a and b are i_2i_1 and i_1i_2, respectively, these memory access operations are coalesced. Since memory access to each matrix α is the same as the transpose permutation, it is conflict-free.

Clearly, the in-place permutation involves coalesced read, conflict-free write, conflict-free read, and coalesced write rounds. The coalesced read and the coalesced write rounds take $\frac{n}{w} + L - 1$ time units each. The conflict-free read and the conflict-free write to the shared memory is performed in k DMMs in parallel. Hence, these rounds takes

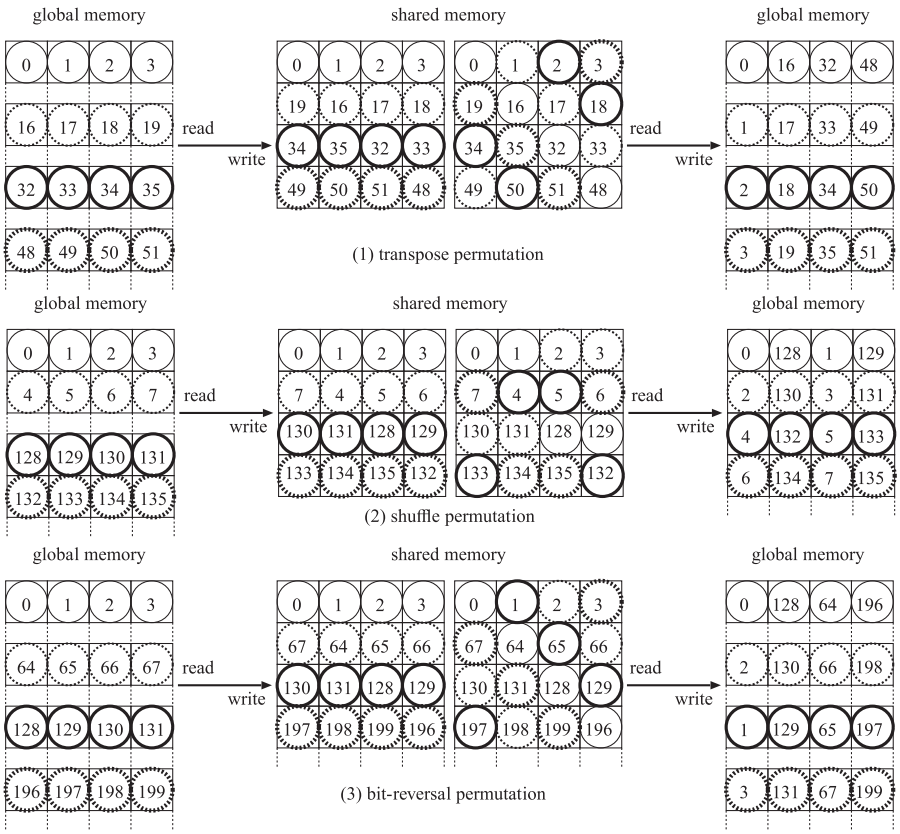

Fig. 4 Data movement of permutation algorithms for shuffle, bit-reversal, and transpose permutations.

$\frac{n}{kw}$ time units each. Thus, we have,

Lemma 5: In-place permutation algorithms for transpose, shuffle, and bit-reversal permutations by memory access rounds and running time in Table 1.

For the reader's benefits, Fig. 4 illustrates the data movement of permutation algorithms performed by the first DMM with 16 threads. We can confirm that the permutations performed correctly, and memory access to the global memory is coalesced and that to the shared memory is conflict-free.

6. Row-Wise and Column-Wise Permutations

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. The idea of row-wise permutation is to execute the offline permutation on the DMM presented in our paper [11] for each row independently in parallel.

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

Let D_i and S_i $(0 \le i \le \sqrt{n}-1)$ be permutations such that $P_i(S_i(j)) = D_i(j)$ is satisfied for all i and j $(0 \le i, j \le \sqrt{n}-1)$. 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 $\sqrt{n}$ blocks of $\sqrt{n}$ threads each. Let $B_0, B_1, \ldots, B_{\sqrt{n}-1}$ denote the $\sqrt{n}$ blocks. Also, let $T_i(j)$ $(0 \le i, j \le \sqrt{n})$ denote the j-th thread of block B_i. Each B_i $(0 \le i \le \sqrt{n}-1)$ is assigned to a row $a[i]$ of a and works for the permutation of $a[i]$. We assume that each block B_i $(0 \le i \le \sqrt{n}-1)$ has two arrays α_i and β_i of size $\sqrt{n}$ each in the shared memory of the DMM. Further, each $T_i(j)$ $(0 \le i, j \le \sqrt{n})$ has two local (register) variables $\mathcal{S}_{i,j}$ and $\mathcal{D}_{i,j}$. The details of the row-wise permutation are spelled out as follows:

[Row-wise permutation]
for $i \leftarrow 0$ to $\sqrt{n}-1$ do in parallel
  for $j \leftarrow 0$ to $\sqrt{n}-1$ do in parallel
    Step 1: $T_i(j)$ performs $a[i][j] \rightarrow \alpha_i[j]$
    Step 2: $T_i(j)$ performs $s[i][j] \rightarrow \mathcal{S}_{i,j}$ and $d[i][j] \rightarrow \mathcal{D}_{i,j}$
    Step 3: $T_i(j)$ performs $\alpha_i[\mathcal{S}_{i,j}] \rightarrow \beta_i[\mathcal{D}_{i,j}]$
    Step 4: $T_i(j)$ performs $\beta_i[j] \rightarrow b[i][j]$

It should be clear that $b[i][D_i(j)]$ stores $a[i][S_i(j)]$. 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.

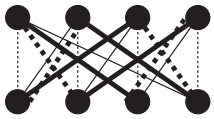

Fig. 5 A regular bipartite graph with degree 4 painted by 4 colors.

We use the following graph theoretic result [22], [23]:

Theorem 6 (König): A regular bipartite graph with degree ρ is ρ-edge-colorable.

Figure 5 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 [22], [23] for the proof of Theorem 6.

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:

- $U = \{B[0], B[1], \ldots, B[w-1]\}$ is a set of nodes each of which corresponds to a bank of α_i.
- $V = \{B[0], B[1], \ldots, B[w-1]\}$ is a set of nodes each of which corresponds to a bank of β_i.
- For each pair source $\alpha_i[j]$ and destination $\beta_i[P(j)]$, E has a corresponding edge connecting $B[j \bmod w](\in U)$ and $B[P_i(j) \bmod w](\in V)$.

Clearly, an edge $(B[u], B[v])$ $(0 \le u, v \le w-1)$ 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 $\frac{\sqrt{n}}{w}$. Hence, G is $\frac{\sqrt{n}}{w}$-colorable from Theorem 6. Suppose that all of the $\sqrt{n}$ edges in E are painted by $\frac{\sqrt{n}}{w}$ colors 0, 1, ..., $\frac{\sqrt{n}}{w} - 1$. We can determine integer values $f_i(j,k)$ $(0 \le j \le \frac{\sqrt{n}}{w} - 1, 0 \le k \le w-1, 0 \le f_i(j,k) \le \sqrt{n} - 1)$ 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 confirm that, for each j,

- w banks $B[f_i(j,0) \bmod w]$, $B[f_i(j,1) \bmod w]$, ..., $B[f_i(j,w-1) \bmod w]$ are distinct, and
- w banks $B[P(f_i(j,0)) \bmod w]$, $B[P(f_i(j,1)) \bmod w]$, ..., $B[P(f_i(j,w-1)) \bmod w]$ are distinct.

It follows that,

- $\alpha_i[f_i(j,0)], \alpha_i[f_i(j,1)], \ldots, \alpha_i[f_i(j,w-1)]$ are in different banks, and
- $\beta_i[P(f_i(j,0))], \beta_i[P(f_i(j,1))], \ldots, \beta_i[P(f_i(j,w-1)]$ are in different banks.

Hence, we define S_i and D_i from $f_i(j,k)$ such that $S_i(j \cdot w + k) = f_i(j,k)$ and $D_i(j \cdot w + k) = P(f_i(j,k))$ for all j and k $(0 \le j \le \frac{\sqrt{n}}{w}, 0 \le k \le w-1)$. 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 read from a and one round of coalesced (conflict-free) write in α. Step 2 performs one round of coalesced read from s and d each. Step 3 involves one round of conflict-free read from α and one round of conflict-free write in β. Finally, Step 4 performs one round of coalesced (conflict-free) read from β and one round of coalesced write 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 7: The row-wise permutation can be done by memory access rounds and running time in Table 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 and 7 we have,

Lemma 8: The column-wise permutation can be done by memory access rounds and running time in Table 1.

7. 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 $(0, 1, \ldots, n-1)$. For convenience, we can think that both a and b are matrices of size $\sqrt{n} \times \sqrt{n}$. For simplicity, we assume that $\sqrt{n}$ is a multiple of w. The goal of permutation is to move a number stored in $a[i][j]$ to $b[\lfloor P(i \cdot w + j)/\sqrt{n} \rfloor][P(i \cdot w + j) \bmod \sqrt{n}]$ for every i and j $(0 \le i, j \le w-1)$. Note that, the permutation is defined 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:

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

Clearly, G is a regular bipartite graph with degree $\sqrt{n}$. From Theorem 6, the bipartite graph G thus obtained can be painted using $\sqrt{n}$ colors such that $\sqrt{n}$ 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 Fig. 6 for illustrating how input numbers are painted.

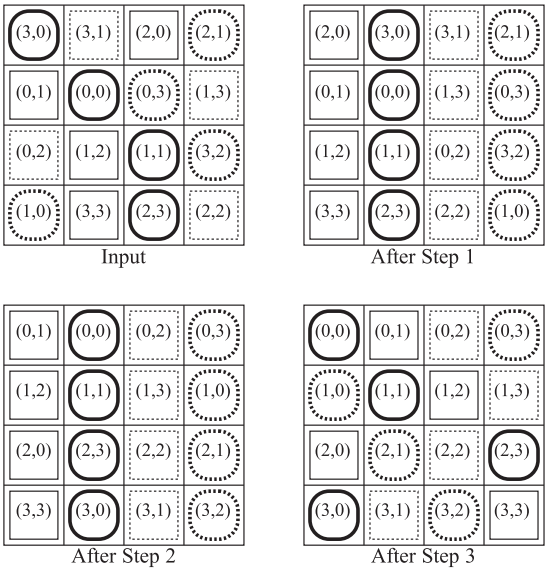

Fig. 6 Illustrating how numbers are routed by the permutation algorithm.

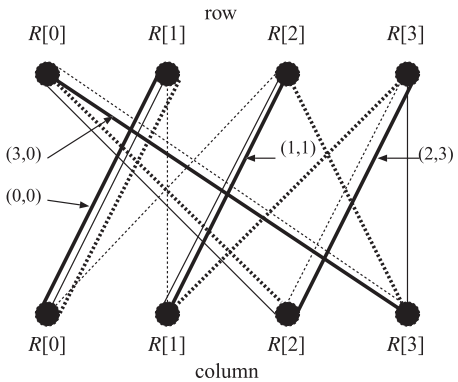

Fig. 7 The bipartite graph for offline permutation in Fig. 6.

In Step 1, row-wise permutation is performed such that a number with color i ($0 \leq i \leq \sqrt{n}-1$) in each row is transferred to the i-th column. From (1) above, $\sqrt{n}$ numbers in each row are painted by $\sqrt{n}$ colors and thus, Step 1 is possible. Step 2 uses column-wise permutation to move numbers to the final row destinations. From (2) above, $\sqrt{n}$ numbers in each column has different $\sqrt{n}$ row destinations and Step 2 is possible. Finally, in Step 3, row-wise permutation is performed to move numbers to the final column destinations. The readers should refer to Fig. 6 for illustrating how numbers are routed by the permutation algorithm for $\sqrt{n} = 4$. In this figure, $(\lfloor P(i \cdot w + j)/\sqrt{n} \rfloor, P(i \cdot w + j) \bmod \sqrt{n})$ is stored in $a[i][j]$ initially, and after the permutation algorithm terminates, (i, j) is stored in $b[i][j]$. Also, Fig. 7 illustrates the bipartite graph corresponding to the permutation in Fig. 6. Each edge corresponds to an element of a matrix such that it connects the source row and the destination row. For example, elements $(3, 0)$, $(0, 0)$, $(1, 1)$, and $(2, 3)$ are painted by the same color in the bipartite graph. Hence, they are in distinct rows in the input matrix. In Step 1, they are moved to the same column by row-wise permutation. Step 2 performs column-wise permutation such that they are in the destination rows. By row-wise permutation in Step 3, they can be moved to the final destination.

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

Theorem 9: Our scheduled permutation algorithm on the HMM can be done by memory access rounds and running time in Table 1.

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

8. Experimental Results

The main purpose of this section is to show experimental results on GeForce GTX 680. We have implemented D-designated, S-designated, our scheduled algorithm and in-place algorithm and evaluate the performance for various size of array a both of float (32-bit) numbers and of double (64-bit) numbers. Also, five permutations, identical, random, transpose, shuffle, and bit-reversal permutations are used to evaluate the performance.

We have invoked $\frac{n}{1024}$ CUDA blocks [8] 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 our 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 five steps, three for row-wise permutation and two for transpose. The implementation of our scheduled algorithms performs five sequential kernel calls for these five steps. For the row-wise permutation, $\sqrt{n}$ CUDA blocks are invoked. However, since each CUDA block can have up to 1024 threads [8], each block is assigned 1024 threads when $\sqrt{n} \geq 1024$. If this is the case, each thread works for $\frac{\sqrt{n}}{1024}$ 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 2 shows the running time of the three permutation algorithms for five permutations. Since the shared memory of GeForce GTX680 has up to 48Kbytes, it is not possible to implement our scheduled algorithm for $4096 \times$

Table 2 The running time (milliseconds) of D-designated, S-designated, our scheduled, and in-place permutation algorithms for GeForce GTX 680.

(a) Permutation for float (32-bit) numbers

	D-designated				S-designated				Our scheduled				In-Place			
$\sqrt{n}$	512	1024	2048	4096	512	1024	2048	4096	512	1024	2048	4096	512	1024	2048	4096
identical	2.48	9.06	33.2	130	2.49	9.13	33.1	129	11.7	46.9	173	780	-	-	-	-
random	15.1	93.9	425	1756	15.7	89.8	398	1644	11.7	47.0	173	780	-	-	-	-
transpose	21.2	127	636	2850	17.8	87.0	370	2037	11.7	46.9	173	780	2.13	8.70	36.5	149
shuffle	3.05	11.5	44.7	186	2.47	9.09	33.6	133	11.7	46.9	174	780	2.44	9.28	38.9	146
bit-reversal	15.6	95.3	459	2328	20.8	96.6	414	1870	11.7	47.0	173	780	2.66	10.1	41.4	157

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

	D-designated				S-designated				Our scheduled				In-Place			
$\sqrt{n}$	256	512	1024	2048	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	-	-	-	-
random	2.98	21.6	104	452	3.40	21.3	100	424	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	1.00	3.04	12.6	56.9
shuffle	1.44	5.14	19.7	82.2	1.08	3.57	13.6	54.6	5.09	17.0	66.7	275	0.92	3.29	12.5	51.7
bit-reversal	3.00	22.0	108	559	3.36	25.0	104	498	5.09	17.0	66.6	275	1.10	3.88	14.1	70.3

4096 double (64-bit) numbers. Thus, we evaluate the performance up to 2048×2048 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. Note that, the running time of our scheduled permutation algorithm does not include the time for preliminary computation such as computation of α_i and β_i in row-wise permutations. This makes sense because a permutation is offline and fixed, and the same permutation is repeatedly performed in practical applications such as FFT and bitonic sorting. From Table 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 $\sqrt{n}$. 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 random, bit-reversal, and transpose permutations have large distribution, our scheduled permutation algorithm runs faster.

Let us compare the theoretical analysis in Table 1, and the experimental results in Table 2. The global memory access latency of CUDA-enabled GPU has several hundreds [8], we can ignore the latency overhead in Table 2 when n is so large that $\sqrt{n} \geq 256$. Also, it makes sense to use parameters $w = 32$ and $k = 8$ for Geforce GTX680, because each warp has 32 threads and it has 8 streaming multiprocessors. If this is the case, the running time of our scheduled permutation algorithm for any permutation is $16\frac{n}{w} + 16\frac{n}{kw} = \frac{18}{32}n$. Further, the running time of D-designated and S-designated algorithms is $\frac{n}{w} + 2\frac{n}{w} = \frac{3}{32}n$ for identical permutation, $2\frac{n}{w} + 2\frac{n}{w} = \frac{4}{32}n$ for shuffle permutation, and $n + 2\frac{n}{w} = \frac{34}{32}n$ for transpose and bit-reversal permutations. In other words, our scheduled algorithm, D-designated algorithm for shuffle permutation, and D-designated algorithm for transpose and bit-reversal permutations are 6 times, $\frac{4}{3}$ times, and $\frac{34}{3}$ times slower than D-designated algorithm for identical permutation. From Table 2, we can see that our scheduled algorithm is approximately 5-6 times slower than D-designated algorithm for identical. Hence, the theoretical analysis on the HMM gives good approximation of the experimental results on Geforce GTX680. On the other hand, D-designated algorithm for transpose and bit-reversal permutations, is 5-20 times slower than D-designated algorithm for identical. Hence, we can say that the theoretical analysis and the experimental results have 50%-200% gaps. This is because memory access requests to the global memory on the GPU are routed through complicated Network-on-Chip to the off-chip DRAM. Although the theoretical analysis and the experimental results have no small gap, we can see that permutation with larger distribution takes more running time. Thus, it makes sense to optimize parallel algorithms on the UMM for the purpose of obtaining efficient implementation on CUDA-enabled GPUs.

Table 2 also shows the running time of our in-place permutation algorithms for transpose, shuffle, and bit-reversal permutations. Clearly, they are faster than all the other algorithms. The in-place permutation algorithm for bit-reversal permutation is slower than those for transpose and shuffle, because it involves computation of bit-reverse. More specifically, it need to compute $i_1 i_2 \cdots i_m$ from $i_m \cdots i_2 i_1$, which takes $O(\log m)$ arithmetic/logic operations. Note that in-place permutation algorithm can be designed only if the permutation is regular. Even if a permutation is regular, it may be very hard to design in-place permutation algorithm for it. In-place permutation can be designed only for very limited permutations. Actually, it is not possible to design in-place permutation algorithm for random permutations.

9. Conclusion

In this paper, 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 schedule offline permutation algo-

rithm is faster than the conventional permutation algorithm for permutations with large distributions.

References

[1] J.D. Scott Parker, "Notes on shuffle/exchange-type switching networks," IEEE Trans. Comput., vol.C-29, no.3, pp.213–222, March 1980.
[2] A. Gibbons and W. Rytter, Efficient Parallel Algorithms, Cambridge University Press, 1988.
[3] K.E. Batcher, "Sorting networks and their applications," Proc. AFIPS Spring Joint Comput. Conf., pp.307–314, 1968.
[4] H.S. Stone, "Parallel processing with the perfect shuffle," IEEE Trans. Comput., vol.C-20, no.2, pp.153–161, Feb. 1971.
[5] W.W. Hwu, GPU Computing Gems Emerald Edition, Morgan Kaufmann, 2011.
[6] D. Man, K. Uda, Y. Ito, and K. Nakano, "A GPU implementation of computing Euclidean distance map with efficient memory access," Proc. International Conference on Networking and Computing, pp.68–76, Dec. 2011.
[7] A. Uchida, Y. Ito, and K. Nakano, "Fast and accurate template matching using pixel rearrangement on the GPU," Proc. International Conference on Networking and Computing, pp.153–159, Dec. 2011.
[8] NVIDIA Corporation, "NVIDIA CUDA C programming guide version 5.0," 2012.
[9] D. Man, K. Uda, H. Ueyama, Y. Ito, and K. Nakano, "Implementations of a parallel algorithm for computing euclidean distance map in multicore processors and GPUs," International Journal of Networking and Computing, vol.1, no.2, pp.260–276, July 2011.
[10] K. Nakano, "Simple memory machine models for GPUs," Proc. International Parallel and Distributed Processing Symposium Workshops, pp.788–797, May 2012.
[11] A. Kasagi, K. Nakano, and Y. Ito, "Offline permutation algorithms on the discrete memory machine with performance evaluation on the GPU," IEICE Trans. Inf. & Syst., vol.E96-D, no.12, pp.2617–2625, Dec. 2013.
[12] NVIDIA Corporation, "NVIDIA CUDA C best practice guide version 5.0," 2012.
[13] K. Nishida, Y. Ito, and K. Nakano, "Accelerating the dynamic programming for the optial poygon triangulation on the GPU," Proc. International Conference on Algorithms and Architectures for Parallel Processing (ICA3PP, LNCS 7439), pp.1–15, Sept. 2012.
[14] A. Kasagi, K. Nakano, and Y. Ito, "An implementation of conflict-free off-line permutation on the GPU," Proc. International Conference on Networking and Computing, pp.226–232, 2012.
[15] K. Nakano, "Asynchronous memory machine models with barrier syncronization," Proc. International Conference on Networking and Computing, pp.58–67, Dec. 2012.
[16] K. Nakano, "Efficient implementations of the approximate string matching on the memory machine models," Proc. International Conference on Networking and Computing, pp.233–239, Dec. 2012.
[17] K. Nakano, "An optimal parallel prefix-sums algorithm on the memory machine models for GPUs," Proc. International Conference on Algorithms and Architectures for Parallel Processing (ICA3PP, LNCS 7439), pp.99–113, Springer, Sept. 2012.
[18] S.H. Hsiao and C.Y.R. Chen, "Performance evaluation of circuit switched multistage interconnection networks using a hold strategy," IEEE Trans. Parallel Distrib. Syst., vol.3, pp.632–640, Sept. 1992.
[19] K. Nakano, "The hierarchical memory machine model for GPUs," Proc. International Parallel and Distributed Processing Symposium Workshops, pp.591–600, May 2013.
[20] D. Man, K. Nakano, and Y. Ito, "The approximate string matching on the hierarchical memory machine, with performance evaluation," Proc. International Symposium on Embedded Multicore/Many-core System-on-Chip, pp.79–84, Sept. 2013.
[21] M. Rubio, R. Gónez, and K. Drouiche, "A new superfast bit reversal algorithm," Int. J. Adaptive Control and Signal Processing, vol.16, no.10, pp.703–707, 2002.
[22] K. Nakano, "Optimal sorting algorithms on bus-connected processor arrays," IEICE Trans. Fundamentals, vol.E76-A, no.11, pp.2008–2015, Nov. 1993.
[23] R.J. Wilson, Introduction to Graph Theory, 3rd ed., Longman, 1985.

Akihiko Kasagi received the BE and ME from the Department of Information Engineering, Hiroshima University in 2012 and 2013. Currently, he is a Ph.D student at the Department of Information Engineering, Hiroshima University.

Koji Nakano received the BE, ME and Ph.D. degrees from Department of Computer Science, Osaka University, Japan in 1987, 1989, and 1992 respectively. In 1992-1995, he was a Research Scientist at Advanced Research Laboratory. Hitachi Ltd. In 1995, he joined Department of Electrical and Computer Engineering, Nagoya Institute of Technology. In 2001, he moved to School of Information Science, Japan Advanced Institute of Science and Technology, where he was an associate professor. He has been a full professor at School of Engineering, Hiroshima University from 2003. He has published extensively in journals, conference proceedings, and book chapters. He served on the editorial board of journals including IEEE Transactions on Parallel and Distributed Systems, IEICE Transactions on Information and Systems, and International Journal of Foundations on Computer Science. He has also guest-edited several special issues including IEEE TPDS Special issue on Wireless Networks and Mobile Computing, IJFCS special issue on Graph Algorithms and Applications, and IEICE Transactions special issue on Foundations of Computer Science. He has organized conferences and workshops including International Conference on Networking and Computing, International Conference on Parallel and Distributed Computing, Applications and Technologies, IPDPS Workshop on Advances in Parallel and Distributed Computational Models, and ICPP Workshop on Wireless Networks and Mobile Computing. His research interests includes image processing, hardware algorithms, GPU-based computing, FPGA-based reconfigurable computing, parallel computing, algorithms and architectures.

Yasuaki Ito received B.E. degree from Nagoya Institute of Technology (Japan), M.S. degree from Japan Advanced Institute of Science and Technology in 2003, and D.E. degree from Hiroshima University (Japan), in 2010. Dr. Ito is currently with the School of Engineering, at Hiroshima University, where he is currently an Associate Professor at Hiroshima University. His research interests include reconfigurable architectures, parallel computing, computational complexity and image processing.