PAPER *Special Section on Special Section on Parallel and Distributed Computing and Networking*

Offline Permutation Algorithms on the Discrete Memory Machine with Performance Evaluation on the GPU

Akihiko KASAGI[†], ***Nonmember*, Koji NAKANO**[†], ***and* Yasuaki ITO**[†], ***Members***

SUMMARY The Discrete Memory Machine (DMM) is a theoretical parallel computing model that captures the essence of the shared memory access of GPUs. The bank conflicts should be avoided for maximizing the bandwidth of the shared memory access. Offline permutation of an array is a task to copy all elements in array *a* into array *b* along a permutation given in advance. The main contribution of this paper is to implement a conflict-free permutation algorithm on the DMM in a GPU. We have also implemented straightforward permutation algorithms on the GPU. The experimental results for 1024 double (64-bit) numbers on NVIDIA GeForce GTX-680 show that the straightforward permutation algorithm takes 247.8 ns for the random permutation and 1684ns for the worst permutation that involves the maximum bank conflicts. Our conflict-free 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-free permutation is 1.48 times faster for the random permutation and 10.0 times faster for the worst permutation.

key words: *memory machine models, data movement, bank conflict, shared memory, GPU, CUDA*

1. Introduction

The GPU (Graphics Processing Unit), is a specialized circuit designed to accelerate computation for building and manipulating images [1]–[3]. 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 [1], [4]. NVIDIA provides a parallel computing architecture called *CUDA* (Compute Unified 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 [6], since they have hundreds of processor cores and very high memory bandwidth.

CUDA uses two types of memories in the NVIDIA GPUs: *the shared memory* and *the global memory* [5]. The shared memory is an extremely fast on-chip memory with lower capacity, say, 16-64 Kbytes. The global memory is implemented as an off-chip DRAM, and has large capacity, say, 1.5-6 Gbytes, but it has high access latency. 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 [2], [6], [7]. The address space of the shared memory is mapped into several physical memory banks. If two or more threads access to the same memory banks at the same time, the access requests are processed sequentially. Hence, to maximize the memory access performance, threads of CUDA should access to 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 [8], we have introduced two models, *the Discrete Memory Machine (DMM)* and *the Unified Memory Machine (UMM)*, which reflect the essential features of the shared memory and the global memory of NVIDIA GPUs. The outline of the architectures of the DMM and the UMM are illustrated in Figure 1. In both architectures, *a sea of threads (Ts)* are connected to *the memory banks (MBs)* through *the memory management unit (MMU)*. Each thread is a Random Access Machine (RAM) [9], which can execute fundamental operations in a time unit. We do not discuss the architectures of the sea of threads and the MMU in this paper. We can imagine that the sea of threads consists of a set of multi-core processors which can execute many threads in parallel. Also, the MMU should include a multistage network to route memory access requests to the MBs. Threads are executed in SIMD [10] 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 $(i \bmod w)$-th bank, where w is the number of MBs. **The main difference of the architectures of the DMM and the UMM 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. In our previous papers [8], [11], we have presented efficient algorithms

Manuscript received January 1, 2010.
Manuscript revised January 1, 2010.
[†]The authors are with Department of Information Engineering, Hiroshima University, Kagamiyama 1-4-1, Higashi-Hiroshima, 739-8527, JAPAN
DOI: 10.1587/trans.E0.??.1

including matrix transpose and computing the sum and the prefix-sums on the DMM and the UMM.

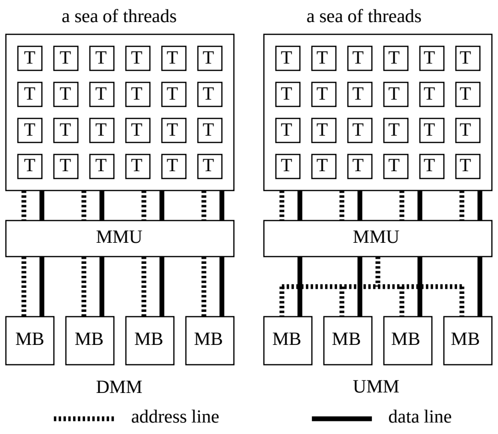

Fig. 1 The architectures of the DMM and the UMM

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 [12]. Sorting network such as bitonic sorting [13], [14] 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 [14] be implemented using the perfect shuffle permutation [15] 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. As illustrated in Figure 2, the implementation for 16 data includes 4 stages of perfect shuffle permutation and 4 stages of 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 contribution of this paper is to present conflict-free 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 $(0, 1, \ldots, n-1)$. In other words, $P(0), P(1), \ldots, P(n-1)$ take distinct integer values in the range $[0, n-1]$. Offline permutation along P is a task

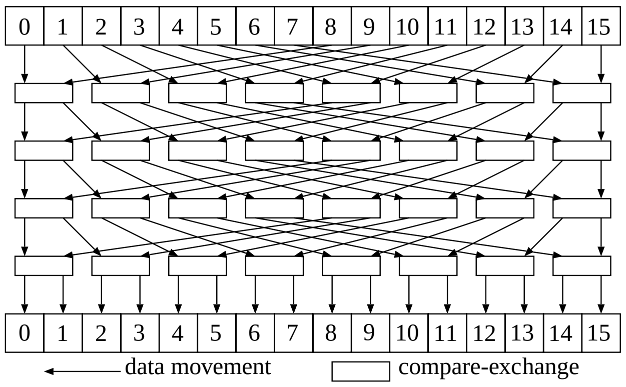

Fig. 2 An implementation of bitonic merge using shuffle exchange

to copy $a[i]$ to $b[P(i)]$ for all i $(0 \leq i \leq n-1)$. The destination-designated (D-designated) algorithm just performs $b[P(i)] \leftarrow a[i]$ 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-free permutation algorithm performs $b[D(i)] \leftarrow a[S(i)]$ 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 our previous paper [8]. Our main contribution is to actually implement permutation algorithms including the D-designated and our conflict-free 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 247.8 ns for the random permutation and 1684ns for the worst permutation that involves the maximum bank conflicts. Our conflict-free 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-free permutation is 1.48 times faster for the random permutation and 10.0 times faster for the worst permutation. Further, we show a conflict-free 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-free in-place permutation method has very small overhead for permutation. We also present the experimental results for 1024 float (32-bit) numbers.

This paper is organized as follows. First, we define the DMM formally in Section 2. In Section 3, we define off-line permutation and show straightforward algorithms. Section 4 shows our conflict-free permutation algorithm and Section 5 describes the details of the implementation. In Section 6, we define several important permutations used for our experiment, and present an in-place permutation method. In

Section 7, experimental results using GeForce GTX-680 are shown. Section 8 concludes our work.

2. Discrete Memory Machine (DMM)

The main purpose of this section is to define the Discrete Memory Machine (DMM) introduced in our previous paper [8]. The reader should refer [8] for the details of the DMM.

We will define *the Discrete Memory Machine (DMM)* of width (or the number of memory banks) w and memory access 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. Figure 3 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. Also, we assume that l time units are necessary to complete an access request and continuous requests are processed in a pipeline fashion through the MMU. Thus, it takes $k + l - 1$ time units to complete k access requests to a particular bank.

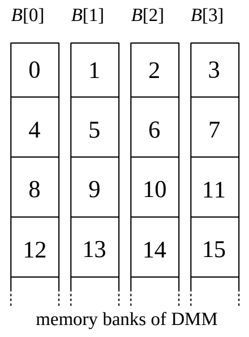

Fig. 3 Memory banks for $w = 4$

Let $T(0), T(1), \ldots, T(p-1)$ be p threads. We assume that p threads are partitioned into $\frac{p}{w}$ groups of w threads called *warps*. More specifically, p threads are partitioned into $\frac{p}{w}$ warps $W(0), W(1), \ldots, W(\frac{p}{w}-1)$ such that $W(i) = \{T(i \cdot w), T(i \cdot w+1), \ldots, T((i+1) \cdot w-1)\}$ ($0 \leq i \leq \frac{p}{w}-1$). Warps are 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 thread in $W(i)$ sends memory access requests, one request per thread, to the memory. We say that *the bank conflict* occurs if two or more threads in a warp access the same bank. *The cost of the memory access* by a warp is the maximum number of memory requests destined for a single bank. For example, if w threads in a warp access the distinct memory banks, the cost is 1. If w threads access to the same bank, the cost is w. We also assume that a thread cannot send memory requests continuously. If a thread sends a memory request, it is transferred to the (imaginary) memory access queue with l-stage pipeline registers. We assume that the memory access is completed when it reaches the last stage. Figure 4 shows an example of memory access on the DMM with w (= 4) memory banks and memory access latency of l (= 5). 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 $W(1)$ occupies 1 stage, because all requests are destined for distinct banks, one request for each bank. Thus, the memory requests occupy three stages, and it takes $3 + 5 - 1 = 7$ time units to complete the memory access.

3. Offline Permutation and Conventional Algorithms

The main purpose of this section is to define 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 $(0, 1, \ldots, n-1)$. In other words, $P(0), P(1), \ldots, P(n-1)$ take distinct integer values in the range $[0, n-1]$. Offline permutation along P is a task to copy $a[i]$ to $b[P(i)]$ for all i ($0 \leq i \leq n-1$).

Suppose that we have n threads for the task of offline permutation. 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 ($0 \leq i \leq n-1$). Let $T(i)$ ($0 \leq i \leq n-1$) denote a thread. The following algorithm, D-designated permutation algorithm, performs the offline permutation along P.

[Destination-designated permutation algorithm]

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

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

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 first 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 3). Hence, writing operations by $W(0)$ have bank conflict.

We can avoid writing bank conflict if we use the S-designated permutation. Let P^{-1} be the inverse of P, that is, $P^{-1}(P(i)) = i$ for all i ($0 \leq i \leq n-1$). We assume that $P^{-1}(0), P^{-1}(1), \ldots, P^{-1}(n-1)$ 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 $i \leftarrow 0$ to $n-1$ do
  $T(i)$ performs $b[i] \leftarrow a[q[i]]$
```

Let us show that this algorithm performs the offline permu-

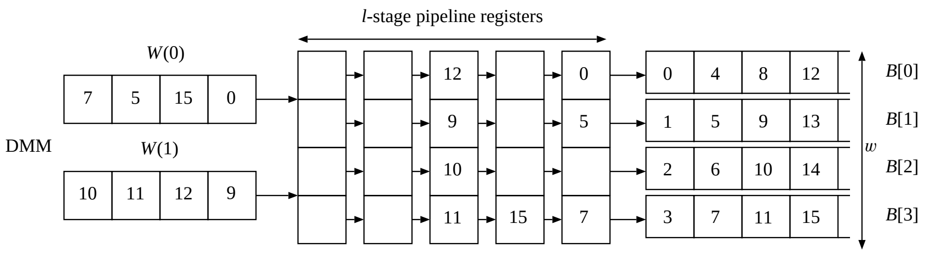

Fig. 4 The memory access on the DMM

tation along P correctly. Clearly, when the S-designated permutation algorithm terminates, $b[i] = a[P^{-1}(i)]$ holds for all i $(0 \le i \le n-1)$. It follows that $b[P(i)] = a[P^{-1}(P(i))] = a[i]$ holds for all i. Thus, the S-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 defined above, we have $P = P^{-1}$. Hence, reading operations have always bank conflicts.

We will show that, bank conflict-free 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 $D(S^{-1}(i)) = P(i)$ for all i $(0 \le i \le n-1)$, 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-free permutation algorithm]

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

Let us see the correctness of the algorithm. When the algorithm terminates, $b[D(i)]$ is storing $a[S(i)]$ for all i $(0 \le i \le n-1)$. Hence, $b[D(S^{-1}(i))]$ is storing $a[S(S^{-1}(i))]$ for all i. Thus, $b[P(i)] = a[i]$ is satisfied 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 define S and D appropriately, access to arrays s and d can be conflict-free. Let P be a permutation defined above. We define 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, $D \cdot S^{-1} = (0, 4, 8, 12, 1, 5, 9, 13, 2, 6, 10, 14, 3, 7, 11, 15) = P$. Thus, after our conflict-free 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-free 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 D-designated and S-designated permutation algorithms take $O(n + l)$ time units if memory access by a warp is performed to the same bank.

4. Graph coloring based conflict-free permutation

This section is devoted to show how S and D are determined from P to guarantee that the conflict-free permutation using S and D involves no bank conflict. The same idea is used in our previous paper [8].

We use an important graph theoretic result [16], [17] as follows:

Theorem 1 (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. Note that the graph may have multiple edges. 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 [16], [17] for the proof of Theorem 1.

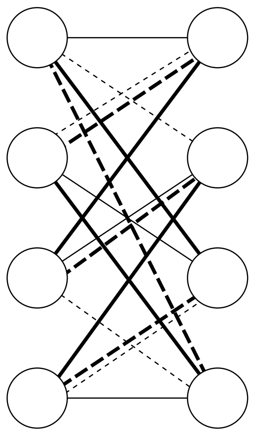

Fig. 5 A regular bipartite graph with degree 4

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

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

Clearly, an edge $(B[u], B[v])$ $(0 \leq u, v \leq w-1)$ corresponds to a word of data to be copied from bank $B[u]$ of a to $B[v]$ of b. Also, $G = (U, V, E)$ is a regular bipartite graph with degree $\frac{n}{w}$. Hence, G is $\frac{n}{w}$-colorable from Theorem 1. Suppose that all of the n edges in E are painted by $\frac{n}{w}$ colors 0, 1, ..., $\frac{n}{w}-1$. We determine value $c_{i,j}$ $(0 \leq i \leq \frac{n}{w}-1, 0 \leq j \leq w-1, 0 \leq c_{i,j} \leq n-1)$ 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 confirm that, for each i,

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

Thus, we have the following important lemma:

Lemma 2: Let $c_{i,j}$ $(0 \leq i \leq \frac{n}{w}-1, 0 \leq j \leq w-1, 0 \leq c_{i,j} \leq n-1)$ denote a source defined above. For each i, we have, (1) $a[c_{i,0}], a[c_{i,1}], \ldots, a[c_{i,w-1}]$ are in different banks, and (2) $b[P(c_{i,0})], b[P(c_{i,1})], \ldots, b[P(c_{i,w-1})]$ are in different banks.

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

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

Suppose that the conflict-free 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 $W(i)$ $(0 \leq i \leq \frac{n}{w}-1)$ performs copy operation from $a[c_{i,0}], a[c_{i,1}], \ldots, a[c_{i,w-1}]$ to $b[P(c_{i,0})], b[P(c_{i,1})], \ldots, b[P(c_{i,w-1})]$. From Lemma 2, reading from a and writing in b by warp $W(i)$ are conflict-free.

5. Implementation of conflict-free permutation algorithm

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

Suppose that a permutation P of $(0, 1, \ldots, n-1)$ is given. We first draw a bipartite graph $G = (U, V, E)$ of P shown in the previous section and find 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 [18]. We perform a bipartite graph coloring in offline. So, it is not necessary to find 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 defined as arrays of n 32-bit float (or 64-bit double) numbers in the shared memory of the GPU and arrays p, q, s, and d are defined arrays of n int numbers in the shared memory as follows:

```
__shared__ float 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(float *a, float *b, int *p){
  b[p[threadIdx.x]]=a[threadIdx.x];
}
```

[Source-designated permutation algorithm]
```
__device__ s-designated(float *a, float *b, int *q){
  b[threadIdx.x]=a[q[threadIdx.x]];
}
```

[Conflict-free permutation algorithm]
```
__device__ conflict-free(float *a, float *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(float *a, float *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 uses one or two arrays of p, q, s, and d, we call it *the array-use* methods. Table 1 summarizes memory access operations performed by each of the permutation algorithms. For example, the D-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-free permutation algorithm performs $4n$ memory access operations. Thus, if each memory access operation have the same access time, the conflict-free permutation algorithm is $\frac{4n}{3n} = \frac{4}{3}$ times slower than the D-designated and S-designated permutation algorithms. However, as we are going to show later, our conflict-free permutation algorithm can be much faster than the D-designated and S-designated

Table 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-free	r	w			r	r	$3n$	n

permutation algorithms.

6. Important permutations and in-place permutation method

This section first 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.
Random: One of all possible n! permutations is selected uniformly at random.
Transpose: Suppose that a and b are matrices 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 ($0 \leq i \leq \sqrt{n} - 1, 0 \leq j \leq \sqrt{n} - 1$).
Shuffle: Let $i_m i_{m-1} \cdots i_1$ be the binary representation of i. The shuffle permutation is defined as $P(i_m i_{m-1} \cdots i_1) = i_{m-1} \cdots i_1 i_m$. Shuffle permutation is used for shuffle exchanging in sorting networks [13], [14].
Bit-reversal: The bit-reversal permutation is defined as $P(i_m i_{m-1} \cdots i_1) = i_1 \cdots i_{m-1} i_m$. Bit-reversal is used for data reordering in the FFT algorithms [12].

If a permutation P is simple and regular, it may be possible to compute the value of $P(i)$ for every i ($0 \leq i \leq n-1$) 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);
```

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 S-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 S-designated permutation method can be done in the same manner as the D-designated permutation.

We can apply the same technique to the conflict-free 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-free 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-free 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 [19], 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.

7. 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 of three permutation algorithms, the D-designated permutation algorithm, the S-designated permutation algorithm, and our conflict-free permutation algorithm for both of the two modes. The computing time for five 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 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-free algorithm runs in the same time units for any permutation. Also, if the in-place computation is used, our conflict-free 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-free permutation algorithm.

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

Table 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 D-designated permutation algorithm for the transpose permutation is 1/32, because reading of a has no bank conflict 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 D-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 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 accessed at the same time without bank conflict. The reader should refer to Figure 6 for illustrating the word alignment of the 64-bit and the 32-bit mode. Please see Section 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 3. For example, in the D-designated permutation algorithm of the shuffle permutation, the first warp writes in $b[0], b[2], \ldots, b[62]$. Since each of 16 pairs $(b[0], b[32]), (b[2], b[34]), \ldots, (b[30], b[62])$ are aligned in a 64-bit word, the writing operations by a warp are conflict-free. From Tables 3 and 4, we can see that if more requests are destined for the same bank, a permutation takes more time.

By comparing Tables 2, 3, and 4, we can see that the execution time is almost proportional to the total cost. More

Table 2 The execution time (ns) of the three algorithms for an 64-bit (double) array of size 1024.

	Algorithms (array-use)			Algorithms (in-place)			
Permutations	D-designated	S-designated	Conflict-free	D-designated	S-designated	Conflict-free	Copy
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	

Table 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-free	D-designated	S-designated	Conflict-free
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 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-free	D-designated	S-designated	Conflict-free	Copy
Identical	135.5	123.6	164.9	-	-	-	102.8
Random	245.9	265.8	164.9	-	-	-	
Transpose	876.3	891.0	164.7	839.3	847.5	105.5	
Shuffle	135.3	183.2	164.9	104.0	161.3	105.0	
Bit-reversal	876.3	891.2	164.8	862.0	870.5	108.9	

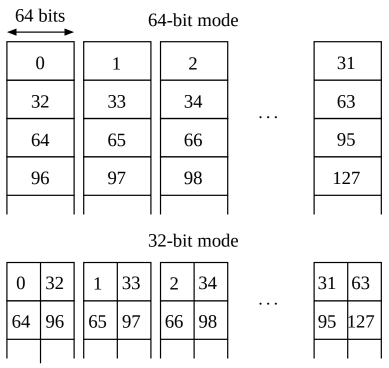

Fig. 6 The word alignments of the 64-bit and 32-bit modes

specifically, the total cost multiplying by 50ns is a moderately good estimation of the execution time. For example, the total cost of the D-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 D-designated or the S-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-free 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-free 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 find such simple program, we should use graph-coloring based conflict-free permutation algorithm using two additional arrays s and d.

8. Conclusion

The main contribution of this paper is to implement several permutation algorithms including our conflict-free 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-free permutation algorithm runs in 167ns for any permutation including the random permutation and the worst permutation, although it performs more memory accesses.

References

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

[2] D. Man, K. Uda, Y. Ito, and K. Nakano, "A GPU implementation of computing euclidean distance map with efficient memory access," Proc. of International Conference on Networking and Computing, pp.68–76, Dec. 2011.

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

Dec. 2011.
[4] K. Nishida, Y. Ito, and K. Nakano, "Accelerating the dynamic programming for the matrix chain product on the GPU," Proc. of International Conference on Networking and Computing, pp.320–326, Dec. 2011.
[5] NVIDIA Corporation, "NVIDIA CUDA C programming guide version 4.2," 2012.
[6] 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.
[7] NVIDIA Corporation, "NVIDIA CUDA C best practice guide version 3.1," 2010.
[8] K. Nakano, "Simple memory machine models for GPUs," Proc. of International Parallel and Distributed Processing Symposium Workshops, pp.788–797, May 2012.
[9] A.V. Aho, J.D. Ullman, and J.E. Hopcroft, Data Structures and Algorithms, Addison Wesley, 1983.
[10] M.J. Flynn, "Some computer organizations and their effectiveness," IEEE Transactions on Computers, vol.C-21, pp.948–960, 1972.
[11] K. Nakano, "An optimal parallel prefix-sums algorithm on the memory machine models for GPUs," Proc. of International Conference on Algorithms and Architectures for Parallel Processing (ICA3PP, LNCS 7439), pp.99–113, Springer, Sept. 2012.
[12] J. D. Scott Parker, "Notes on shuffle/exchange-type switching networks," IEEE Trans. on Computers, vol.C-29, no.3, pp.213 – 222, March 1980.
[13] A. Gibbons and W. Rytter, Efficient Parallel Algorithms, Cambridge University Press, 1988.
[14] K.E. Batcher, "Sorting networks and their applications," Proc. AFIPS Spring Joint Comput. Conf., pp.307–314, 1968.
[15] H.S. Stone, "Parallel processing with the perfect shuffle," IEEE Trans. on Computers, vol.C-20, no.2, pp.153–161, Feb. 1971.
[16] K. Nakano, "Optimal sorting algorithms on bus-connected processor arrays," IEICE Trans. Fundamentals, vol.E76-A, no.11, pp.2008–2015, Nov. 1993.
[17] R.J. Wilson, Introduction to Graph Theory, 3rd edition, Longman, 1985.
[18] J.E. Hopcroft and R.M. Karp, "An $n^{5/2}$ algorithm for maximum matchings in bipartite graphs," SIAM Journal on Computing, vol.2, no.4, pp.225–231, 1973.
[19] M. Li and P.M. Vitányi, An Introduction to Kolmogorov Complexity and Its Applications, 3rd Edition, Springer, 2008.

Akihiko Kasagi received the BE from the Department of Information Engineering, Hiroshima University in 2012. Currently, he is a master 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. From 2004 to 2007 he was a Research Associate at Hiroshima University. Since 2007, Dr. Ito has been with the School of Engineering, at Hiroshima University, where he is working as an Assistant Professor. His research interests include reconfigurable architectures, parallel computing, computational complexity and image processing.