An Implementation of Conflict-Free Offline Permutation on the GPU

Akihiko Kasagi, Koji Nakano, and Yasuaki Ito
Department of Information Engineering
Hiroshima University
Kagamiyama 1-4-1, Higashi Hiroshima, 739-8527 Japan

***Abstract*—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 goal 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 float numbers on NVIDIA GeForce GTX-680 show that a straightforward permutation algorithm takes 246ns and 877ns for random permutation and bit-reversal permutation, respectively. Quite surprisingly, our conflict-free permutation algorithm runs in 165ns both for random permutation and for bit-reversal permutation although it performs more memory access operations. It follows that our conflict-free permutation is 1.5 times faster for random permutation and 5.3 times faster for bit-reversal permutation.**

***Keywords*-memory machine models, data movement, bank conflict, shared memory, GPU, CUDA**

I. Introduction

The GPU (Graphical 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 architecture of the sea of threads in this paper, but we can imagine that it consists of a set of multi-core processors which can execute many threads in parallel. Threads are executed in SIMD [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 two architectures is the connection of the address line between the MMU and the MBs, which can transfer an address value. In the DMM, the address lines connect the MBs and the MMU separately, while a single address line from the MMU is connected to the MBs in the UMM. Hence, in the UMM, the same address value is broadcast to every MB, and the same address of the MBs can be accessed in each time unit. On the other hand, different addresses of the MBs can be accessed in the DMM. The DMM and the UMM capture the essence of the shared memory access and the global memory access of current GPUs. In our previous papers [8], [11], we have presented efficient algorithms including matrix transpose and computing the sum and the prefix-sums on the DMM and the UMM.

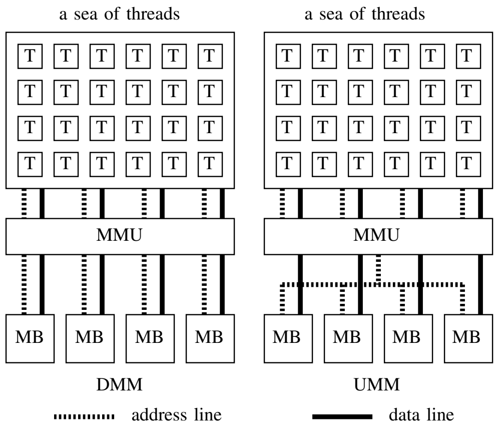

Figure 1. The architectures of the DMM and the UMM

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 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 simulated by permutation on the shared memory. Thus, parallel algorithms on processor networks can be simulated on the shared memory machine by data permutations.

The main contribution of this 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 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 to 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 b and a 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 destination-designated and our conflict-free permutation algorithms on the shared memory of the latest GPU, NVIDIA GeForce GTX-680.

The experimental results for 1024 float numbers on NVIDIA GeForce GTX-680 show that a straightforward permutation algorithm takes 246ns and 877ns for random permutation and bit-reversal permutation, respectively. Quite surprisingly, our conflict-free permutation algorithm runs in 165ns for random permutation and bit-reversal permutation each although it performs more memory access operations. It follows that our conflict-free permutation is 1.5 times faster for random permutation and 5.3 times faster for bit-reversal permutation.

This paper is organized as follows. First, we define the DMM formally in Section II. In Section III, we define off-line permutation and show straightforward algorithms. Section IV shows our conflict-free permutation algorithm and Section V describes the details of this implementation. In Section VI, experimental results using GeForce GTX-680 are shown. Section VII offers conclusions.

II. 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.

Let $m[i]$ $(i \geq 0)$ denote a memory cell of address i in the memory. Let $B[j] = \{m[j], m[j+w], m[j+2w], m[j+3w], \ldots\}$ $(0 \leq j \leq w-1)$ denote *the j-th bank* of the memory. Clearly, a memory cell $m[i]$ is in the $(i \bmod w)$-th memory bank. Figure 2 illustrates memory banks of DMM for $w = 4$. We assume that memory cells in different banks can be accessed in a time unit, but no two memory cells in the same bank can be accessed in a time unit. 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.

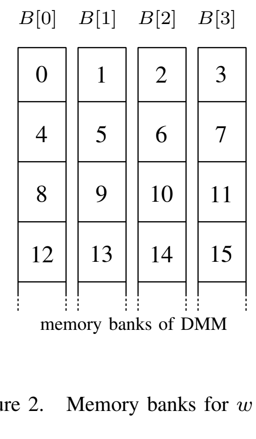

Figure 2. Memory banks for $w = 4$

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 and is skipped. 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. We also assume that a thread cannot send a new memory access request until the previous memory access request is completed. Hence, if a thread send a memory access request, it must wait l time units to send a new memory access request.

III. Offline Permutation and Conventional Algorithms

The main purpose of this section is to define offline permutation and show conventional algorithm 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, destination 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 $b[12]$ and they are in the same bank $B[0]$ (Figure 2). Hence, writing operations by $W(0)$ have bank conflict.

We can avoid writing bank conflict if we use the source-designated permutation Q. Let P^{-1} be the inverse of P, that is, $P^{-1}(P(i)) = i$ for all i $(0 \leq i \leq n-1)$. 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 permutation along P correctly. The goal of the permutation along P is to satisfy $b[P(i)] = a[i]$ for all i $(0 \leq i \leq n-1)$. Hence, it is sufficient to satisfy $b[P^{-1}(P(i))] = a[Q(i)]$ for all i $(0 \leq i \leq n-1)$. From $P^{-1}(P(i)) = i$, it is also sufficient to satisfy $b[i] = a[P^{-1}(i)]$. Thus, the source-designated permutation algorithm performs the offline permutation along P correctly.

It should be clear that writing operations in b and reading operations from q have no bank conflict. However, reading operations from a may have bank conflict. For example, for P defined above, we have $P = P^{-1}$. Hence, reading operations has 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 $S^{-1}(D(i)) = P(i)$ for all i $(0 \leq i \leq n-1)$, 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 $(0 \leq i \leq n-1)$. In other words, $b[S^{-1}(D(i))]$ is storing $a[S^{-1}(S(i))]$ for all i. Thus, $b[P(i)] = a[i]$ is 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, $S^{-1} \cdot D = (0, 4, 8, 12, 1, 5, 9, 13, 2, 6, 10, 14, 3, 7, 11, 15) = P$. Thus, 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 destination-designated and source-designated permutation algorithms take $O(n+l)$ time units if memory access by a warp is performed to the same bank.

IV. 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 [15], [16] as follows:

Theorem 1 (König): A regular bipartite graph with degree ρ is ρ-edge-colorable.

Figure 3 illustrates an example of a regular bipartite graph with degree 4 painted by 4 colors. Each edge is painted by 4 colors such that no node is connected to edges with the same color. In other words, no two edges with the same color share a node. The readers should refer to [15], [16] for the proof of Theorem 1.

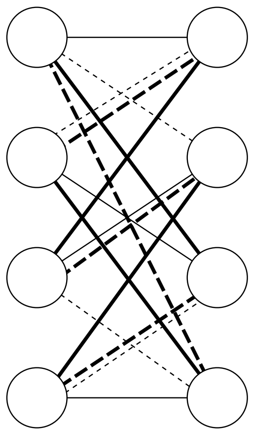

Figure 3. A regular bipartite graph with degree 4

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 $(B[u],B[v])$ $(0\le u,v\le w-1)$ 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 $0,1,\ldots,\frac{n}{w}-1$. We determine value $c_{i,j}$ $(0\le i\le \frac{n}{w}-1, 0\le j\le w-1, 0\le c_{i,j}\le n-1)$ 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 $B[c_{i,0} \bmod w]$, $B[c_{i,1} \bmod w]$, $\ldots$, $B[c_{i,w-1} \bmod w]$ are distinct, and
- w banks $B[P(c_{i,0}) \bmod w]$, $B[P(c_{i,1}) \bmod w]$, $\ldots$, $B[P(c_{i,w-1}) \bmod w]$ are distinct.

Thus, we have the following important lemma:

Lemma 2: Let $c_{i,j}$ $(0\le i\le \frac{n}{w}-1, 0\le j\le w-1, 0\le c_{i,j}\le n-1)$ 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:

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

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 $W(i)$ $(0\le i\le \frac{n}{w}-1)$ 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.

V. 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.

A permutation P of $(0,1,\ldots,n-1)$ is given as an input. We first draw a bipartite graph $G=(U,V,E)$ of P shown in 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.

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

- two terminals of A are not connected to M, and
- edges of M and $\overline{M}(=E-M)$ appear alternatively in A.

Figure 4 shows examples of augmenting paths.

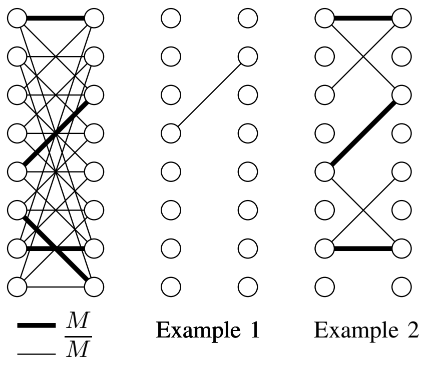

Figure 4. Examples of augmenting paths

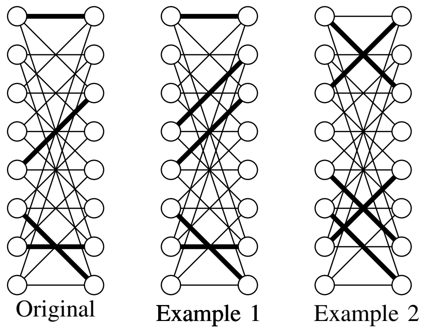

Figure 5. The resulting bipartite matching after flipping operation

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

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

- $M \leftarrow M - (A \cap M)$, that is, remove edges in $A \cap M$ from M.
- $M \leftarrow M \cup (A \cap \overline{M})$, that is, add edges in $A \cap \overline{M}$ to A.

The reader should refer to Figure 5 for illustrating the resulting bipartite matching after the flipping operation. Clearly, the resulting M is a matching and the number of edges in M increases by one.

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

From these observation, we can find a maximum matching of a bipartite graph G as follows. Initially, let $M = \emptyset$. Find an augmenting path with respect to G and M and performs flipping operation. This task is repeated until we can find no augmenting path with respect to G and M. The resulting matching M is a maximum matching. If the graph is a regular bipartite graph, M is also a maximum matching. For graph coloring, we repeat finding the maximum matching. First, find the maximum matching M, paint edges in M with color 0, and remove edges in M from G. In this way, we can find a bipartite graph coloring in polynomial time.

Note that, 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 float 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 reveal 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.

Table I summarizes memory access operations performed by the algorithms. For example, the destination-designated permutation algorithm performs read operations for arrays a and p, and write operations for array b. Hence, it performs $2n + n = 3n$ memory access operations. Our conflict-

Table I
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

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 destination-designated and source-designated permutation algorithms. However, as we are going to show in the next section, our conflict-free permutation algorithm can be much faster than the destination-designated and source-designated permutation algorithms.

VI. EXPERIMENTAL RESULTS

This section is devoted to show the experimental results using GeForce GTX-680. To evaluate the performance of permutation algorithms We use several widely-used important permutations as follows:
Identical: Permutation such that $P(i) = i$ for every i.
Random: One of all the possible $n!$ permutations is selected uniformly at random.
Transpose: Suppose that a and b are matrix with dimension $\frac{n}{w} \times w$. Transpose corresponds to the data movement such that a is read in row-major order and b is written in column-major order as illustrated in Figure 6. That is, $P(i \cdot w + j) = j \cdot \frac{n}{w} + i$ for every i and j $(0 \leq i \leq \frac{n}{w} - 1, 0 \leq j \leq w - 1)$.
Shuffle: Let $i_m i_{m-1} \cdots i_1$ be the binary representation of i. Shuffle permutation is defined by $P(i_m i_{m-1} \cdots i_1) = i_{m-1} \cdots i_1 i_m$. Shuffle permutation is widely used for shuffle exchanging in sorting networks [13], [14].
Bit-reversal: Shuffle permutation is defined by $P(i_m i_{m-1} \cdots i_1) = i_1 \cdots i_{m-1} i_m$. Bit-reversal is used for data reordering in the FFT algorithms [12].

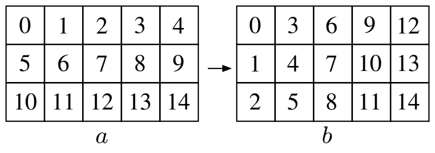

Figure 6. Transpose permutation

We have evaluated the performance three permutation algorithms, the destination-designated permutation algorithm (D-designated), the source-designated permutation algorithm (S-designated), and our conflict-free permutation algorithm (conflict-free). Also, to estimate the overhead of these three permutation algorithms, we have evaluated the performance of the copy algorithm. Since any permutation algorithm cannot be faster than the copy algorithm, its computing time is the lower bound of that for any permutation algorithm.

The performance has been evaluated for $n = 1024$ using NVIDIA GeForce GTX-680. A CUDA kernel with a single block of 1024 threads is invoked from the host. The 1024 threads executes one of the four device functions, D-designated, S-designated, conflict-free, and copy. Note that the number w of memory banks is 32. For Transpose and Bit-reversal permutations, wiring operation by the D-designated and S-designated algorithms involves many bank conflicts in the sense that most of threads in a warp writing in the same bank. Table II shows the executing time for an array of size $n = 1024$. Since the executing time of each algorithm is too short to measure, each algorithm has been executed for each permutation 1 million times and took its average. Table II also shows the ratio of the execution time with respect to that of the simple copy. Note that, any permutation algorithm cannot be faster than the simple copy. Thus, the ratio in the table clarifies the overhead of each permutation algorithm.

According to the table, for Identical and Shuffle permutations that rarely involve bank conflicts, the D-designated and S-designated algorithms run faster than our conflict-free algorithm because extra memory access operations are necessary for the conflict-free algorithm as shown in Table I. The executing time of the S-designated algorithm for Shuffle permutation is longer than that of the D-designated algorithm since the number of bank conflict increases. On the other hand, for Random, Transpose, and Bit-reversal permutations, whose number of bank conflicts is not small, our conflict-free permutation algorithm runs faster than the others though extra memory access is necessary since all the memory access can avoid bank conflict. For Transpose and Bit-reversal permutations, our conflict-free permutation algorithm attains a speedup factor of more than 5 over the others. That is, our conflict-free permutation algorithm is efficient for permutations that frequently involve bank conflict. Also, the execution time of our conflict-free is almost constant for all permutations.

In applications using the GPU, the permutation algorithm is often executed for multiple arrays in parallel. Therefore, we have also evaluated the performance of the permutation algorithms if they executed for multiple arrays of size 1024. More specifically, a kernel call of CUDA generates multiple blocks, each of which executes a permutation algorithm for an array of size 1024 in parallel. Figure 7 shows the executing time for multiple arrays of size 1024. From the figure, when the number of arrays is less than or equal to 8, the executing time is almost the same. In other words, at most 8 CUDA blocks executing a permutation algorithm run at the same time. Further, if a kernel call generates $8k$ $(k \geq 1)$ CUDA blocks, the execution time is almost

Table II
THE EXECUTING TIME AND THE RATIO WITH RESPECT TO THE COPY FOR AN ARRAY OF SIZE $n = 1024$.

Permutations	Algorithms			Copy
	D-designated	S-designated	Conflict-free	
Identical	135.366ns/1.315	123.637ns/1.201	165.180ns/1.605	102.937/1.000
Random	246.918ns/2.390	265.786ns/2.582	164.544ns/1.598	
Transpose	876.329ns/8.513	891.006ns/8.656	164.851ns/1.601	
Shuffle	136.073ns/1.322	183.192ns/1.780	164.773ns/1.601	
Bit-reversal	876.891ns/8.519	891.390ns/8.660	164.764ns/1.601	

proportional to k. These facts make sense because GTX-680 has a GPU with 8 multicore processors work in parallel.

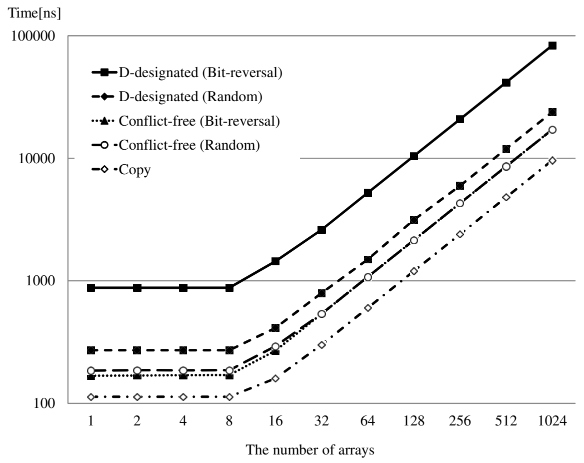

Figure 7. The executing time to permute arrays of 1024 elements.

VII. 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 float numbers on NVIDIA GeForce GTX-680 show that the destination-designated permutation algorithm takes 246ns for the random permutation and 877ns for the worst permutation that involves many bank conflicts. Our conflict-free permutation algorithm runs in approximately 165ns 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," in *Proc. of International Conference on Networking and Computing*, Dec. 2011, pp. 68–76.

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

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

[5] NVIDIA Corporation, "NVIDIA CUDA C programming guide version 4.0," 2011.

[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, 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," in *Proc. of International Parallel and Distributed Processing Symposium Workshops*, May 2012, pp. 788–797.

[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," in *Proc. of International Conference on Algorithms and Architectures for Parallel Processing (ICA3PP, LNCS 7439)*, Sept. 2012, pp. 99–113.

[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," in *Proc. AFIPS Spring Joint Comput. Conf.*, vol. 32, 1968, pp. 307–314.

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

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