An Optimal Offline Permutation Algorithm on the Hierarchical Memory Machine, with the GPU implementation

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

***Abstract*—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. This conventional algorithm simply performs three rounds of memory access for reading from a, reading from p, and writing in b. The main contribution of this paper is to present an optimal offline permutation algorithm running in $O(\frac{n}{w} + L)$ time units using n threads on the HMM with width w and latency L. We also implement our optimal offline permutation algorithm on GeForce GTX-680 GPU and evaluate the performance. Quite surprisingly, our optimal offline permutation algorithm achieves better performance than the conventional algorithm in most permutations, although it performs 32 rounds of memory access. For example, the bit-reversal permutation for 4M float (32-bit) numbers can be completed in 780ms by our optimal permutation algorithm, while the conventional algorithm takes 2328ms. We can say that the experimental results of this paper 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.**

***Keywords*-Memory machine models, offline permutation, GPU, CUDA**

I. Introduction

The GPU (Graphics Processing Unit), is a specialized circuit designed to accelerate computation for building and manipulating images [1], [2], [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]. NVIDIA provides a parallel computing architecture called *CUDA* (Compute Unified Device Architecture) [4], 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 [5], since they have hundreds of processor cores and very high memory bandwidth.

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* [4]. 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 [2], [5], [6], [7]. 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 [8], 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 [9], [10], [11], [12], [10]. 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 [13] 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 Figure 1 have 4 threads in each warp and 4 MBs.

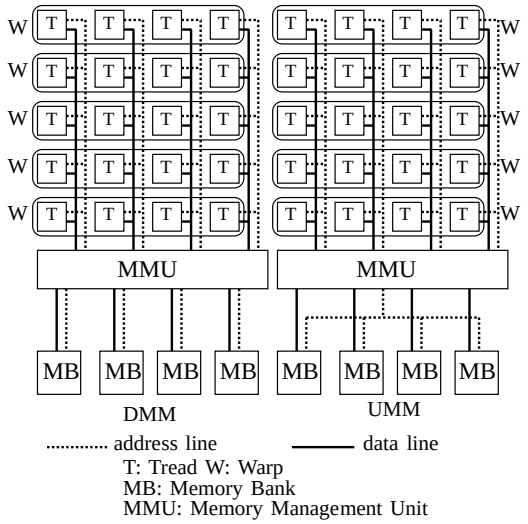

Figure 1. The architectures of the DMM and the UMM with width $w = 4$

Quite Recently, we have introduced the Hierarchical Memory Machine (HMM) [14], [15], 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 2 illustrates the architecture of the HMM. The HMM consists of d DMMs and a single UMM. Each DMM has w memory banks and the UMM 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 NVIDIA 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 shared memory and the global memory of NVIDIA GPUs have low latency and several hundred clock cycles, respectively. 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 [15].

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. The offline permutation has many applications in the area of parallel computing. For example, matrix transpose, which is one of the important permutations, is frequently used in matrix computation. It is known that the computation of the FFT can be done by a multistage network in which each stage involves permutation [16]. Sorting networks such as bitonic sorting [17], [18] also involve permutation in each stage. Further, communication on processor networks such as hypercubes, meshes, and so on can be emulated by permutation. Further, random permutation is very helpful for randomized algorithms [19].

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

In our previous paper [8], 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 DMM with width w and latency l. Later, we have implemented the conventional offline permutation algorithm and this conflict-free permutation algorithm on a single SM of GeForce GTX-680 GPU and evaluated the performance [9]. 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. It is also shown in [8] an offline permutation algorithm running in $O(3^{\log \frac{\log n}{\log w}}(\frac{n}{w} + \frac{nl}{p} + l))$ time units using p threads on the UMM with width w and latency l. This algorithm is time optimal only for small n such that $n \leq w^{O(1)}$. This permutation algorithm has large overhead for large n.

The main contribution of this paper is to present an optimal permutation algorithm for larger arrays on the global memory of the HMM. Our scheduled offline permutation algorithm performs three step permutations, row-wise permutation, column-wise permutation, and row-wise permutation, each of which is performed in DMMs of the HMM in parallel. Our scheduled offline permutation runs in $32\frac{n}{w} + 16L - 16$ 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 $3\frac{n}{w} + 3L - 3$

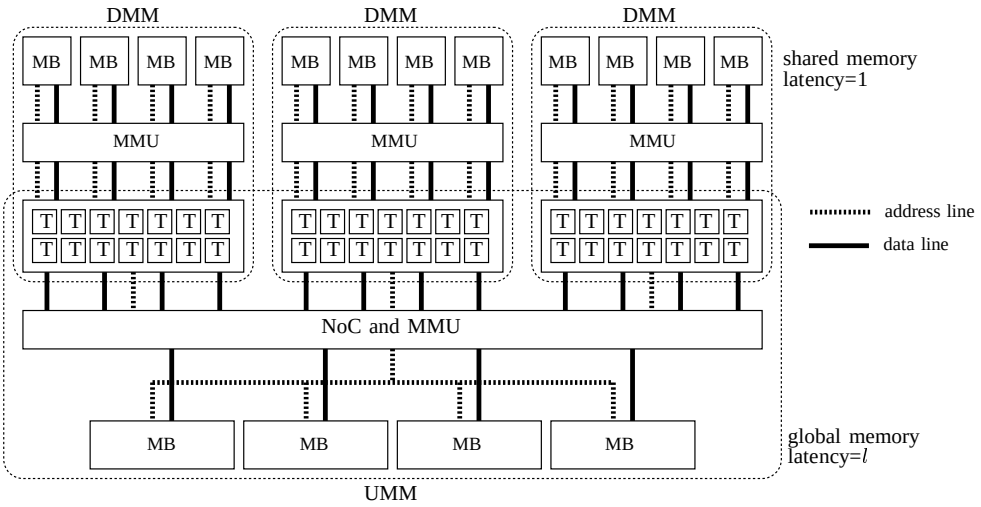

Figure 2. The architecture of the HMM with $d = 3$ DMMs and width $w = 4$

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 factors 32 and 16 in the running time seem too large to achieve better performance than the conventional algorithm with small constant factors in the computing time. However, contrary to this instinct, our scheduled permutation algorithm can run faster than the conventional algorithm. To show this fact, we have implemented our scheduled 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 $n \geq$ 256K ($= 2^{18}$). 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, for almost all of the permutations over all possible $n!$ permutations, our scheduled permutation algorithm is faster than the conventional algorithm. To show this fact, we pick 1000 permutations from all possible $n!$ permutations at random for n =4M($= 2^{22}$). The conventional algorithm takes 424.87-426.39ms, while our scheduled permutation algorithm takes 173.50-173.92ms. Thus, our scheduled permutation algorithm is 2.45 time faster than the conventional algorithm for almost all permutations over all possible $n!$ permutations.

This paper is organized as follows. First, we define three memory machines, DMM, UMM, and HMM in Section II. In Section III, we define three memory access operations, casual memory access, coalesced memory access, and conflict-free memory access and evaluate the running time. Section IV defines the offline permutation and show two conventional permutation algorithms, destination-designated permutation algorithm and source-designated permutation algorithm. Section V presents an algorithm for transposing a matrix, and Section VI shows algorithms for row-wise permutation and column-wise permutation of a matrix. In Section VII, we present our scheduled permutation algorithm and show the optimality. Finally, Section VIII shows experimental results for comparing the conventional permutation algorithms and our scheduled permutation algorithm. Section IX concludes our work.

II. Memory Machine Models: DMM, UMM, and HMM

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

We first 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 $(i \bmod w)$-th memory bank. We assume that memory cells in different banks can be accessed in a time unit, but no two memory cells in the same bank can be accessed in a time unit. Also, we assume that l time units are necessary

to complete an access request and continuous requests are processed in a pipeline fashion through the MMU.

We assume 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 $W(0), W(1), \ldots, W(\frac{p}{w}-1)$ 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 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 3 shows examples of memory access on the DMM and the UMM. We assume that each memory access request is completed when it reaches the last pipeline stage. Two warps $W(0)$ and $W(1)$ access to $\langle m[7], m[5], m[15], m[0] \rangle$ 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, 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.

Finally, we define *the Hierarchical Memory Machine (HMM)*. The HMM consists of d DMMs and a single UMM as illustrated in Figure 2. Each DMM has w memory banks and the UMM also has w memory banks. We call the memory banks of each DMM *the shared memory* and those of the UMM *the global memory*. Each DMM works independently. Threads are partitioned into warps of w threads, and each warp is dispatched for the memory access for the shared memory in turn. Further, each warp of w threads in all DMMs can send memory access requests to the global memory. Figure 2 illustrates the architecture of the HMM with $d = 3$ DMMs. Each DMM and the UMM has $w = 4$ memory banks. The shared memory of each DMM and the global memory of the UMM correspond to "the shared memory" of each streaming multiprocessor and "the global memory" of GPUs. Since the latency of "the shared memory" in existing GPUs is very low [4], we assume that the memory access latency of the shared memory of the DMM is 1 for simplicity. Also, since the latency of "the global memory" in the GPUs is several hundred clock cycles [4], it makes sense to use parameter L for the global memory access of the HMM.

III. 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

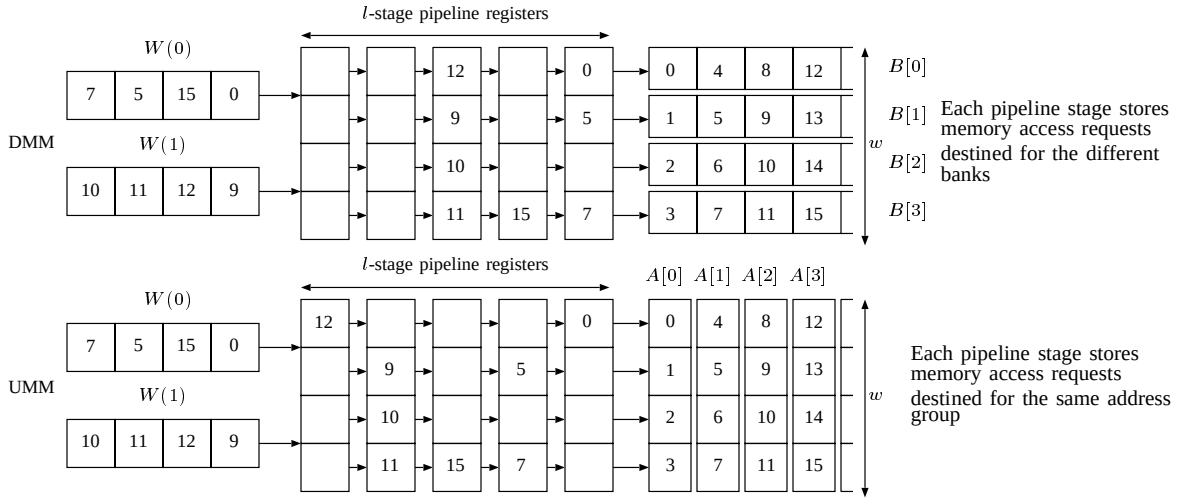

Figure 3. Examples of memory access on the DMM and the UMM

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 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.

IV. 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 $(0\leq i\leq n-1)$. 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 $(0\leq i\leq n-1)$. 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 I.

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 $(0\leq i\leq n-1)$. 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 $(0\leq i\leq n-1)$. 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 I.

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

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

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 [17], [18].

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

Bit-reversal: The bit-reversal permutation is defined as

Table I
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	-	-	1	1	1	1	$4\frac{n}{w} + 2L - 2$
Row-wise permutation	-	-	3	1	2	2	$8\frac{n}{w} + 4L - 4$
Column-wise permutation	-	-	5	3	4	4	$16\frac{n}{w} + 8L - 8$
Our scheduled permutation	-	-	11	5	8	8	$32\frac{n}{w} + 16L - 16$

$P(i_m i_{m-1} \cdots i_1) = i_1 \cdots i_{m-1} i_m$. Bit-reversal is used for data reordering in the FFT algorithms [16]

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 $(0 \leq i, j \leq \sqrt{n} - 1)$.

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 I.

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

V. TRANSPOSE OF A MATRIX ON THE HMM

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

We first show that a matrix a of size $w \times w$ on the global memory can be transposed using one DMM with w^2 threads. We use an array α of size $w \times w$ on the shared memory. 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 allocation *the diagonal arrangement*. Figure 4 illustrates the diagonal arrangement of a 4×4 matrix. The advantage of the diagonal arrangement is:

- all elements $\alpha[i, 0], \alpha[i, 1], \ldots, \alpha[i, w-1]$ in the same row are arranged in different memory banks, and
- all elements $\alpha[0, j], \alpha[1, j], \ldots, \alpha[w-1, j]$ in the same column are arranged in different memory banks.

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

[**Transpose of a matrix of size** $w \times w$]
for $i \leftarrow 0$ to $w - 1$ do in parallel
  for $j \leftarrow 0$ to $w - 1$ do in parallel
    Step 1: $T(i \cdot w + j)$ performs $\alpha[i, j] \leftarrow a[i][j]$
    Step 2: $T(i \cdot w + j)$ performs $a[i][j] \leftarrow \alpha[j, i]$

Since each $a[i][j]$ is copied to $a[j][i]$ through $\alpha[i, j]$, the transposing can be done correctly. Every element of a in the global memory is read once and written once. Also, every element of α in the shared memory is read once and written once. Clearly, memory access to a is coalesced, and that to α is conflict-free.

Next, we will show that the transpose of a matrix a of size $\sqrt{n} \times \sqrt{n}$ can be done using that of size $w \times w$. We assume that $\sqrt{n}$ is a multiple of w. We partition a into $\frac{\sqrt{n}}{w} \times \frac{\sqrt{n}}{w}$ submatrices of size $w \times w$. Let $A(i, j)$ $(0 \leq i, j \leq \frac{\sqrt{n}}{w} - 1)$ denote a submatrix of elements $a[i'][j']$ $(i \cdot w \leq i' \leq (i + 1) \cdot w - 1, j \cdot w \leq j' \leq (j + 1) \cdot w - 1)$. The transpose can be done by storing the transpose of each $A(i, j)$ in $A(j, i)$

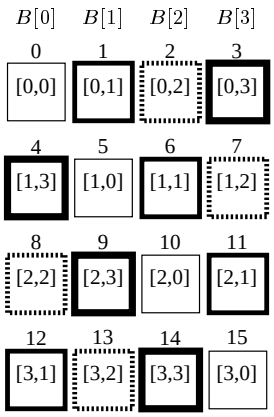

Figure 4. Diagonal arrangement of a 4×4 matrix

for all i and j ($0 \leq i, j \leq \frac{\sqrt{n}}{w} - 1$). This can be done by the transposing algorithm for a $w \times w$ matrix. Thus, we have,

Lemma 5: The transpose of a matrix of size $\sqrt{n} \times \sqrt{n}$ can be done by memory access rounds and running time in Table I.

VI. 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.

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 $a[i][j]$ ($0 \leq i, j \leq \sqrt{n}$) to $b[i][P_i(j)]$.

Let D_i and S_i ($0 \leq i \leq \sqrt{n} - 1$) be permutations such that $P_i(S_i(j)) = D_i(j)$ is satisfied for all i and j ($0 \leq i, j \leq \sqrt{n} - 1$). 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 $T_i(j)$ ($0 \leq i, j \leq \sqrt{n}$) denote the j-th thread of block B_i. Each B_i ($0 \leq i \leq \sqrt{n} - 1$) is assigned to a row $a[i]$ of a and works for the permutation of $a[i]$. Since we have d DMMs, each DMM has $\frac{\sqrt{n}}{d}$ blocks. We assume that each block B_i ($0 \leq i \leq \sqrt{n}-1$) has two arrays α_i and β_i of size $\sqrt{n}$ each in the shared memory of the DMM. Further, each $T_i(j)$ ($0 \leq i, j \leq \sqrt{n}$) 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 $\alpha_i[j] \leftarrow a[i][j]$
    Step 2: $T_i(j)$ performs $\mathcal{S}_{i,j} \leftarrow s[i][j]$ and $\mathcal{D}_{i,j} \leftarrow d[i][j]$
    Step 3: $T_i(j)$ performs $\beta_i[\mathcal{D}_{i,j}] \leftarrow \alpha_i[\mathcal{S}_{i,j}]$
    Step 4: $T_i(j)$ performs $b[i][j] \leftarrow \beta_i[j]$

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.

We use the following graph theoretic result [20], [21]:

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 [20], [21] for the proof of Theorem 6.

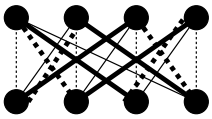

Figure 5. A regular bipartite graph with degree 4 painted by 4 colors

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 $(B[u], B[v])$ ($0 \leq u, v \leq w-1$) 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 $f_i(j, k)$ ($0 \leq j \leq \frac{\sqrt{n}}{w} - 1, 0 \leq k \leq w - 1, 0 \leq f_i(j, k) \leq \sqrt{n} - 1$) 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, (1) 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 (2) w banks $B[P(f_i(j, 0)) \bmod w]$, $B[P(f_i(j, 1)) \bmod w]$, ..., $B[P(f_i(j, w-1)) \bmod w]$ are distinct. It follows

that, (1) $\alpha_i[f_i(j,0)]$, $\alpha_i[f_i(j,1)]$, ..., $\alpha_i[f_i(j,w-1)]$ are in different banks, and (2) $\beta_i[P(f_i(j,0))]$, $\beta_i[P(f_i(j,1))]$, ..., $\beta_i[P(f_i(j,w-1)]$ 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 ($0 \leq j \leq \frac{\sqrt{n}}{w}, 0 \leq k \leq w-1$). 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 reading from a and one round of coalesced (conflict-free) writing in α. Step 2 performs one round of coalesced reading from s and d each. Step 3 involves one round of conflict-free reading from α and one round of conflict-free writing in β. Finally, Step 4 performs one round of coalesced (conflict-free) reading from β and one round of coalesced writing in b. Note that a, b, s, and d are in the global memory, and α and β are in the shared memory. Thus, we have,

Lemma 7: The row-wise permutation can be done by memory access rounds and running time in Table I.

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 I.

VII. 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 ($0 \leq i, j \leq w-1$). 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 Figure 6 for illustrating how input numbers are painted.

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 Figure 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 $b[i][j]$.

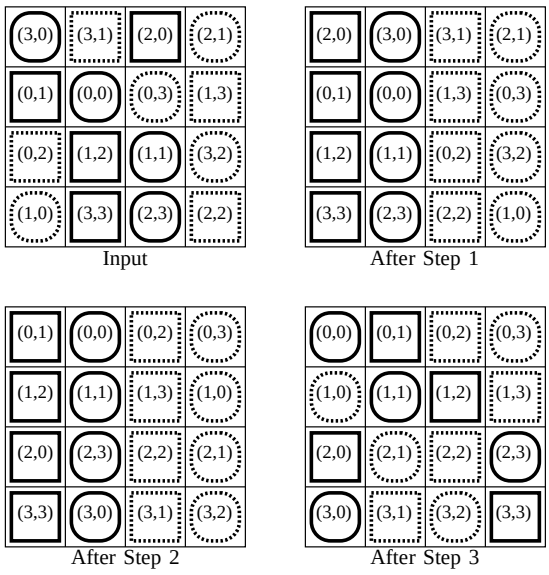

Figure 6. Illustrating how numbers are routed by the permutation algorithm

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

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

We can prove $\Omega(\frac{n}{w} + L)$-time lower bound for the permutation on the HMM. Since all of the n elements in a must be read at least once and w elements can be read in a

time unit, $\frac{n}{w}$ time units are necessary. Also, reading of one element takes L time units. Thus, $\Omega(\frac{n}{w}+L)$ time units are necessary for permutation of n elements and our scheduled permutation algorithm is optimal from the theoretical point of view.

VIII. Experimental Results

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

We have invoked $\frac{n}{1024}$ CUDA blocks [4] of 1024 threads each for D-designated and S-designated permutation algorithms. In the D-designated algorithm, each block is assigned to a row of a and works for the copy of the assigned row. Similarly, in the S-designated algorithm, each block is assigned to a row of b. Also, arrays p and q used in D-designated and S-designated are arrays of int (32-bit) numbers, since at most $\log 4096^2 = 24$ bits are necessary.

Recall that scheduled permutation algorithm involves three steps, row-wise permutation, column-wise permutation, and row-wise permutation. Also, column-wise permutation has three substeps, transpose, row-wise permutation, and transpose. Thus, it has essentially 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 [4], 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 II 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 4096×4096 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.

From Table II, 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 the random, the bit-reversal, and the transpose permutations have large distribution, our scheduled permutation algorithm runs faster when $\sqrt{n} \geq 512$. However, our scheduled permutation algorithm is slower when $\sqrt{n} = 256$. We can presume that the L2 cache of size 512Kbytes [22] on GeForce GTX-680 decreases the overhead of the casual memory access performed by the D-designated and S-designated permutation algorithms efficiently for small n. Also, in most cases, the S-designated permutation algorithm is more efficient that the D-designated. This is because the casual writing takes more running time than the casual reading due to the overhead of cache coherency in writing.

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

	D-designated	S-Designated	Scheduled	$\frac{D_w(P)}{n}$
Minimum	424.87	397.89	173.50	0.99987
Average	425.52	398.27	173.66	0.99989
Maximum	426.39	398.77	173.92	0.99990

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

IX. 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 optimal offline permutation algorithm is faster than the conventional permutation algorithm for most cases.

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*. IEEE CS Press, Dec. 2011, pp. 68–76.

Table II
THE RUNNING TIME (MILLISECONDS) OF D-DESIGNATED, S-DESIGNATED AND OUR SCHEDULED ALGORITHM

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

	D-designated					S-designated					Our scheduled				
$\sqrt{n}$	256	512	1024	2048	4096	256	512	1024	2048	4096	256	512	1024	2048	4096
identical	0.86	2.48	9.06	33.2	130	0.86	2.49	9.13	33.1	129	3.87	11.7	46.9	173	780
shuffle	0.94	3.05	11.5	44.7	186	0.84	2.47	9.09	33.6	133	3.87	11.7	46.9	174	780
random	1.55	15.1	93.9	425	1756	3.30	15.7	89.8	398	1644	3.87	11.7	47.0	173	780
bit-reversal	1.60	15.6	95.3	459	2328	3.12	20.8	96.6	414	1870	3.87	11.7	47.0	173	780
transpose	1.44	21.2	127	636	2850	2.72	17.8	87.0	370	2037	3.87	11.7	46.9	173	780

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

	D-designated				S-designated				Our scheduled			
$\sqrt{n}$	256	512	1024	2048	256	512	1024	2048	256	512	1024	2048
identical	1.07	3.57	13.5	54.6	1.07	3.60	13.8	54.6	5.07	16.9	66.6	275
shuffle	1.44	5.14	19.7	82.2	1.08	3.57	13.6	54.6	5.09	17.0	66.7	275
random	2.98	21.6	104	452	3.40	21.3	100	424	5.09	17.0	66.6	275
bit-reversal	3.00	22.0	108	559	3.36	25.0	104	498	5.09	17.0	66.6	275
transpose	2.07	22.2	134	638	2.99	15.4	80.3	358	5.12	17.0	66.6	275

[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*. IEEE CS Press, Dec. 2011, pp. 153–159.

[4] NVIDIA Corporation, "NVIDIA CUDA C programming guide version 5.0," 2012.

[5] 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.

[6] NVIDIA Corporation, "NVIDIA CUDA C best practice guide version 5.0," 2012.

[7] K. Nishida, Y. Ito, and K. Nakano, "Accelerating the dynamic programming for the optial poygon triangulation on the GPU," in *Proc. of International Conference on Algorithms and Architectures for Parallel Processing (ICA3PP, LNCS 7439)*. IEEE CS Press, Sept. 2012, pp. 1–15.

[8] K. Nakano, "Simple memory machine models for GPUs," in *Proc. of International Parallel and Distributed Processing Symposium Workshops*. IEEE CS Press, May 2012, pp. 788–797.

[9] A. Kasagi, K. Nakano, and Y. Ito, "An implementation of conflict-free off-line permutation on the GPU," in *Proc. of International Conference on Networking and Computing*, 2012, pp. 226–232.

[10] K. Nakano, "Asynchronous memory machine models with barrier syncronization," in *Proc. of International Conference on Networking and Computing*, Dec. 2012, pp. 58–67.

[11] ——, "Efficient implementations of the approximate string matching on the memory machine models," in *Proc. of International Conference on Networking and Computing*, Dec. 2012, pp. 233–239.

[12] ——, "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)*. Springer, Sept. 2012, pp. 99–113.

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

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

[15] D. Man, K. Nakano, and Y. Ito, "The approximate string matching on the hierarchical memory machine, with performance evaluation," in *to appear in Proc. of International Symposium on Embedded Multicore/Many-core System-on-Chip*, Sept. 2013, p. to appear.

[16] 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.

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

[18] K. E. Batcher, "Sorting networks and their applications," in *Proc. AFIPS Spring Joint Comput. Conf.*, vol. 32, 1968, pp. 307–314.

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

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

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

[22] NVIDIA Corporation, "NVIDIA GeForce GTX680 GPU whitepaper," 2012.