Random Address Permute-Shift Technique for the Shared Memory on GPUs

Koji Nakano*, Susumu Matsumae†, and Yasuaki Ito*

*Department of Information Engineering
Hiroshima University
Kagamiyama 1-4-1, Higashi Hiroshima, 739-8527 Japan

†Department of Information Science
Saga University
Honjo 1, Saga, 840-8502 Japan

***Abstract*—The Discrete Memory Machine (DMM) is a theoretical parallel computing model that captures the essence of memory access to the shared memory of a streaming multiprocessor on CUDA-enabled GPUs. The DMM has w memory banks that constitute a shared memory, and w threads in a warp try to access them at the same time. However, memory access requests destined for the same memory bank are processed sequentially. Hence, it is very important for developing efficient algorithms to reduce the memory access congestion, the maximum number of memory access requests destined for the same bank. The main contribution of this paper is to present a novel algorithmic technique called the random address permute-shift (RAP) technique that reduces the memory access congestion. We show that the RAP reduces the memory access congestion to $\tilde{O}(\frac{\log w}{\log \log w})$ for any memory access requests including malicious ones by a warp of w threads. Also, we can guarantee that the congestion is 1 both for contiguous access and for stride access. The simulation results for $w = 32$ show that the expected congestion for any memory access is only 3.53. Since the malicious memory access requests destined for the same bank take congestion 32, our RAP technique substantially reduces the memory access congestion. We have also applied the RAP technique to matrix transpose algorithms. The experimental results on GeForce GTX TITAN show that the RAP technique is practical and can accelerate a direct matrix transpose algorithm by a factor of 10.**

***Keywords*-GPU, CUDA, memory bank conflicts, memory access congestion, randomized technique**

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. 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.

NVIDIA GPUs have streaming multiprocessors (SMs) each of which executes multiple threads in parallel. CUDA uses two types of memories in 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 very long. 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 *bank conflicts* of the shared memory access and *coalescing* of the global memory access [5]. The address space of the shared memory is mapped into several physical memory banks. If two or more threads access the same memory bank at the same time, the access requests are processed in turn. Hence, to maximize the memory access performance, threads in a warp should access distinct memory banks to avoid the bank conflicts of the shared 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 [6], 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 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 [7], [8] and the UMM [9]. 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 $w = 4$. Each thread works as a Random Access Machine (RAM) [10], which can execute fundamental operations in a time unit. Threads are executed in SIMD [11] fashion, and they run on the same program

and work on different data. 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 [12] 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.

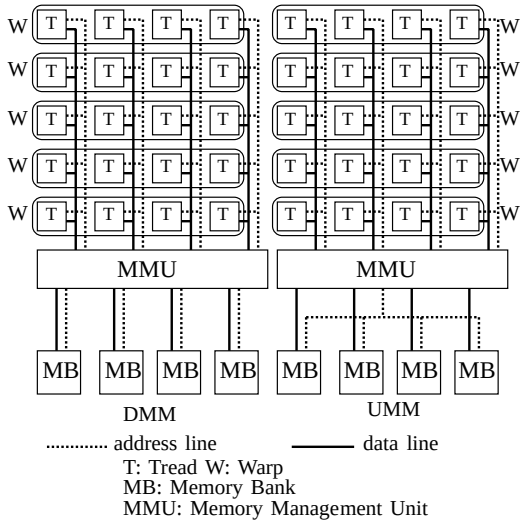

Figure 1. The architectures of the DMM and the UMM with width $w = 4$

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 set of address lines 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. Since the memory access of the UMM is more restricted than that of the DMM, the UMM is less powerful than the DMM. Also, we assume that MBs are accessed in a pipeline fashion with latency l. In other words, if a thread sends a memory access request, it takes at least l time units to complete it. A thread can send a new memory access request only after the completion of the previous memory access request and thus, it can send at most one memory access request in l time units.

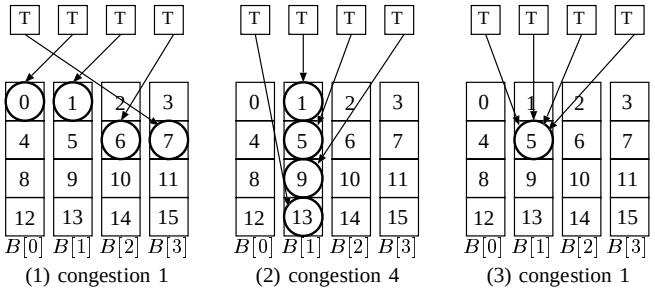

Figure 2. Examples of memory access and the congestion for $w = 4$

It is very important for developing efficient algorithms on the DMM to reduce *the memory access congestion*, the maximum number of unique memory access requests by a warp destined for the same bank. The memory access congestion takes value between 1 and w. The reader should refer to Figure 2 showing examples of the memory access and the congestion. If w threads send memory access requests to distinct banks, the congestion is 1 and the memory access is conflict-free. If all memory access requests are destined to the same bank, the congestion is w. It is not easy and sometimes impossible to minimize the memory access congestion for some problems. For example, a straightforward matrix transpose algorithm that reads a matrix in row major order and writes in column major order involves memory access with congestion w. On the other hand, by an ingenious memory access technique, we can transpose a matrix with congestion 1 [6]. Further, in our previous paper [6], we have developed a complicated graph coloring technique to eliminate bank conflicts in off-line permutation. We have implemented this offline permutation algorithm on GeForce GTX-680 GPU [13]. The experimental results showed that the offline permutation algorithm developed for the DMM runs on the GPU much faster than the conventional offline permutation algorithm [13]. Although it is very important to minimize the memory access congestion, it may be a very hard task.

In latest CUDA-enabled GPUs such as GeForce GTX TITAN, the number w of memory banks and threads in a warp is 32, and the size of a shared memory is no more than 48Kbytes [4]. Hence, a matrix with $w \times w$ double (64-bit) numbers in such CUDA-enabled GPUs occupies 8Kbytes and it is not possible to store more than 6 matrices of size $w \times w$ in a shared memory. Thus, many algorithms designed for CUDA-enabled GPUs use one or several matrices of size $w \times w$ in the shared memory [1], [4], [14]. For example, paper [14] has presented an optimal offline permutation algorithm for the global memory. This optimal algorithm repeats offline permutation for 32×32 matrices in the shared

memory of each streaming multiprocessor in a GPU. Also, an efficient matrix multiplication for a large matrix in the global memory repeats multiplication of 32×32 submatrices in the shared memory [4]. Hence, it makes sense to focus on a matrix of size $w \times w$. Usually, each (i, j) element $(0 \leq i, j \leq w-1)$ of a matrix of size $w \times w$ is mapped to address $i \cdot w + j$ in a conventional implementation. We call such a straightforward implementation, *RAW (RAW access to memory) implementation*. In the RAW implementation, the congestion of stride access is w, while that of contiguous access is 1. Hence, CUDA developers should implement algorithms in GPUs so that it never performs stride access to the shared memory.

The main contribution of this paper is to present sophisticated algorithmic technique called *the random address permute-shift (RAP)*, which reduces the memory access congestion for any memory access to a matrix of size $w \times w$ by a warp of w threads. Let r be a random permutation of $(0, 1, \ldots, w-1)$ uniformly selected at random from all possible $n!$ permutations. In other words, w integers $r_0, r_1, \ldots, r_{w-1}$ are distinct in the range $[0, w-1]$. By the RAP technique, each (i, j) element $(0 \leq i, j \leq w-1)$ of a matrix is mapped to address $i \cdot w + ((j + r_i) \bmod w)$ and thus, it is in memory bank $(j + r_i) \bmod w$. Our first contribution is to show that, by the RAP technique, it is guaranteed that:

- any contiguous access and any stride access has no bank conflict, and
- the congestion is at most $\tilde{O}(\frac{\log w}{\log \log w})$ for any memory access including malicious ones by a warp of w threads, where $\tilde{O}(f)$ denotes expected $O(f)$.

Quite recently, we have presented an algorithmic technique called *the random address shift (RAS)* to reduce the memory access congestion on the DMM [7], [15]. Basically, the random address shift technique is inspired by parallel hashing that averages the access to memory modules [16], [17]. The idea is to arrange address $i \cdot w + j$ in bank $(j + r_i) \bmod w$ for independent random numbers $r_0, r_1, \ldots$ computed beforehand. However, the RAS implementation involves bank conflicts for stride memory access. On the other hand, our new RAP implementation has no bank conflict for stride memory access and the congestion is 1. Table I summarizes the memory access congestion by the RAW, the RAS, and the RAP implementations.

Table I
THE MEMORY ACCESS CONGESTION OF THE RAW, THE RAS, AND THE RAP

	RAW	RAS	RAP
Any	$[1, w]$	$\tilde{O}(\frac{\log w}{\log \log w})$	$\tilde{O}(\frac{\log w}{\log \log w})$
Contiguous	1	1	1
Stride	32	$\tilde{O}(\frac{\log w}{\log \log w})$	1

The second contribution is to show simulation results of memory access by the RAW, the RAS and the RAP. Our simulation results show that the congestions of the RAW, the RAS and the RAP are the same for random memory access. By the RAP, contiguous and stride memory access operations have no bank conflict. Also, when $w = 32$, the congestion of the RAP for a stride memory access is always 1, while the congestions of RAW and the RAS are 32 and 3.53, respectively. Hence, the RAP is much more efficient for the stride memory access.

The third contribution is to implement the RAP technique in a streaming multiprocessor on GeForce GTX TITAN [18] which supports CUDA Compute Capability 3.5 [4]. In particular, we have implemented three matrix transpose algorithms, Contiguous Read Stride Write (CRSW), Stride Read Contiguous Write (SRCW), and Diagonal Read Diagonal Write (DRDW). The CRSW and the SRCW follow the definition of a matrix transpose. More specifically, in the CRSW, a matrix is read in row major order and is written in column major order to transpose it. The SRCW reads a matrix in column major order and writes in row major order. Since memory access requests in column major order are destined for the same bank, these algorithms take a lot of time. The DRDW is optimized for the RAW implementation and performs reading and writing in diagonal order to reduce the memory access congestion to 1. Thus, the DRDW runs much faster than the others in the RAW implementation. However, it may not be easy for CUDA developers to find an efficient algorithm such as the DRDW for complicated problems. The implementation results of CRSW and SRCW algorithms for a 32×32 matrix in the shared memory show that the RAP implementation is much faster than the others. More specifically, the RAP runs only 154.5ns, while the RAW and the RAS run 1595ns and 303.6ns for CRSW algorithm, respectively.

We also present several methods to extend the RAP for arrays larger than w^2. The RAP for larger arrays has fewer bank conflicts using fewer random numbers than the RAS.

From the theoretical analysis, the simulation results, and the implementation results shown in this paper, we can say that the RAP is a potent method to reduce memory access congestion and bank conflicts that spoil high computing power of the GPUs. It is not necessary for CUDA developers to avoid bank conflicts if they use the RAP. The memory access congestion can be automatically reduced by the RAP even if it involves a lot of bank conflicts. Further, it will be a nice idea to implement the RAP technique as embedded hardware in future GPUs. More specifically, a circuit that evaluates $j \cdot w + (k + r_j) \bmod w$ for address conversion by the RAP can be embedded. Using such hardware support, the overhead of address conversion by the RAP can be negligible.

This paper is organized as follows. In Section II, we first define the DMM. Section III introduces fundamental memory access operations and matrix transpose algorithms which

are used to evaluate the performance. In Section IV, we present the random address permute-shift (RAP) technique, and evaluate the memory access congestion by theoretical analysis. Section V shows simulation results to evaluate the actual values of the congestion by the RAW, the RAS, and the RAP. In Section VI, we show experimental results on GeForce GTX TITAN. Section VII introduces several ideas to extend the RAP for larger arrays. Section VIII concludes our work.

II. Discrete Memory Machine (DMM)

The main purpose of this section is to define the Discrete Memory Machine (DMM) introduced in our previous paper [6]. The reader should refer to [6] 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. 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.

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 threads in $W(i)$ send memory access requests, one request per thread, to the memory. Threads are executed in SIMD [11] fashion, and all thread must execute the same instruction. Hence, if one of them sends a memory read request, none of the others can send memory write request. 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 one.

Figure 3 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 $W(1)$ occupy 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.

Let us define *the congestion* of memory access by a warp of w threads. Suppose that w threads in a warp access the memory banks. The memory access congestion is the maximum number of unique memory access requests destined for the same bank. We assume that, if two or more threads access the same address, the memory access requests are merged and processed as a single request. Thus, if all w threads in a warp access the same address, the congestion is 1. We also assume that if multiple memory writing requests are sent to the same address, one of them is arbitrary selected and its writing operation is performed. The other writing requests are ignored. Thus, the DMM works as the Concurrent Read Concurrent Write (CRCW) mode with arbitrary resolution of simultaneous writing [19]. For example, the congestion of memory access in Figure 2 (1) is 1, because all requests are destined for distinct banks. In Figure 2 (2), the congestion is 4, because all requests are destined for bank $B[1]$. In Figure 2 (3), all threads access $m[5]$. Thus, these memory requests are merged into one and the congestion is 1.

III. Fundamental memory access operations and matrix transpose algorithms

The main purpose of this section is to show three fundamental memory access operations for a matrix, *contiguous access*, *stride access* and *diagonal access* [6]. We also show three transposing algorithms of a matrix using these three memory access operations.

Suppose that we have a matrix a of size $w \times w$ in the memory of the DMM. We assume that $a[i][j]$ ($0 \leq i, j \leq w-1$) is arranged in address $i \cdot w + j$. Since $(i \cdot w + j) \bmod w = j$, each $a[i][j]$ is in bank $B[j]$. In these memory access operations, each element in a matrix is accessed by a thread. In the contiguous access, threads are assigned to the matrix in row-major order. Threads are assigned to the matrix in column-major order in the stride access. In the diagonal access, threads are assigned in diagonal order. The readers should refer to Figure 4 for illustrating these three memory access operations for $w = 4$.

More formally, these three memory access operations can be written as follows:

[Contiguous Access]
for $i \leftarrow 0$ to $w-1$ do in parallel
  for $j \leftarrow 0$ to $w-1$ do in parallel
    thread $T(i \cdot w + j)$ accesses $a[i][j]$

[Stride Access]
for $i \leftarrow 0$ to $w-1$ do in parallel

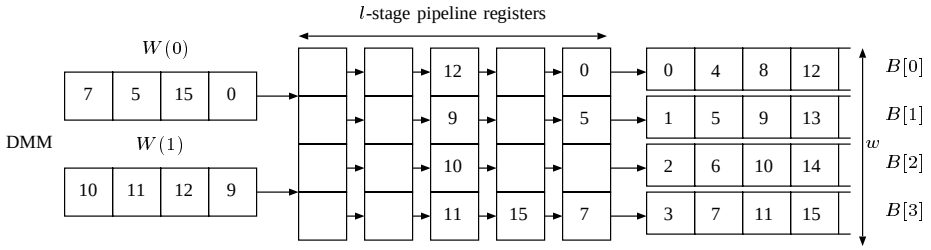

Figure 3. The Discrete Memory Machine (DMM)

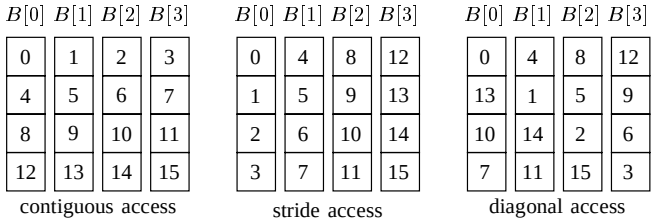

Figure 4. The contiguous access, the stride access, and the diagonal access for $w = 4$

for $j \leftarrow 0$ to $w-1$ do in parallel
  thread $T(i \cdot w + j)$ accesses $a[j][i]$

[Diagonal Access]
for $i \leftarrow 0$ to $w-1$ do in parallel
 for $j \leftarrow 0$ to $w-1$ do in parallel
  thread $T(i \cdot w + j)$ accesses $a[j][(i+j) \bmod w]$
   (or $a[(i+j) \bmod w][j]$)

It should be clear that the congestion of the contiguous access and the diagonal access is 1. On the other hand, in the stride access, w threads in a warp access distinct addresses in the same bank, and the congestion is w. In the contiguous access, w warps send memory access requests in w time units. Thus, it takes $w + l - 1$ time units to complete the contiguous access. In the stride access, w memory access requests sent by a warp occupy w pipeline stages. Hence, it takes $w^2 + l - 1$ time units to complete the stride access. Since the congestion of the diagonal access is 1, the diagonal access takes $w + l - 1$ time units similarly to the contiguous access.

We can design three matrix transpose algorithms, Contiguous Read Stride Write (CRSW), Stride Read Contiguous Write (SRCW), and Diagonal Read Diagonal Write (DRDW), using these three memory access operations. In the CRSW, a matrix is read in row major order and is written in column major order. In other words, the CRSW performs the contiguous read and the stride write for matrix transpose. Similarly, the SRCW performs the stride read and the contiguous write. In the DRDW, a matrix is read and written in diagonal order. The reader should refer to Figure 5 illustrating the three matrix transpose algorithms. The details of the three matrix transpose algorithms are spelled out as follows:

[Contiguous Read Stride Write (CRSW)]
for $i \leftarrow 0$ to $w-1$ do in parallel
 for $j \leftarrow 0$ to $w-1$ do in parallel
  thread $T(i \cdot w + j)$ performs $a[j][i] \leftarrow a[i][j]$

[Stride Read Contiguous Write (SRCW)]
for $i \leftarrow 0$ to $w-1$ do in parallel
 for $j \leftarrow 0$ to $w-1$ do in parallel
  thread $T(i \cdot w + j)$ performs $a[i][j] \leftarrow a[j][i]$

[Diagonal Read Diagonal Write (DRDW)]
for $i \leftarrow 0$ to $w-1$ do in parallel
 for $j \leftarrow 0$ to $w-1$ do in parallel
  thread $T(i \cdot w + j)$ performs
   $a[j][(i+j) \bmod w] \leftarrow a[(i+j) \bmod w][j]$

Let us evaluate the computing time of three transpose algorithms on the DMM. The CRSW transpose and the SRCW transpose involve the stride memory access. Thus, they take $O(w^2 + l)$ time units. The DRDW transpose performs diagonal read/write, it takes $O(w + l)$ time units. Hence, we have,

Lemma 1: The CRSW, the SRCW, and the DRDW transpose algorithms for a matrix of size $w \times w$ take $O(w^2 + l)$ time units, $O(w^2 + l)$ time units, and $O(w + l)$ time units, respectively, using w^2 threads on the DMM with width w and latency l.

We can implement these algorithms in a streaming multiprocessor of a GPU without any modification. We call such implementations *RAW (RAW access to memory) implementations*.

For example, the RAW implementation of the CRSW transpose algorithm for a matrix of size 32×32 is described as follows:

[The RAW implementation of the CRSW]

```
__shared__ double a[32][32],b[32][32];
int i = threadIdx.x/32;
int j = threadIdx.x%32;
double c;
b[j][i] = a[i][j];
```

We assume that matrices a and b allocated in the shared

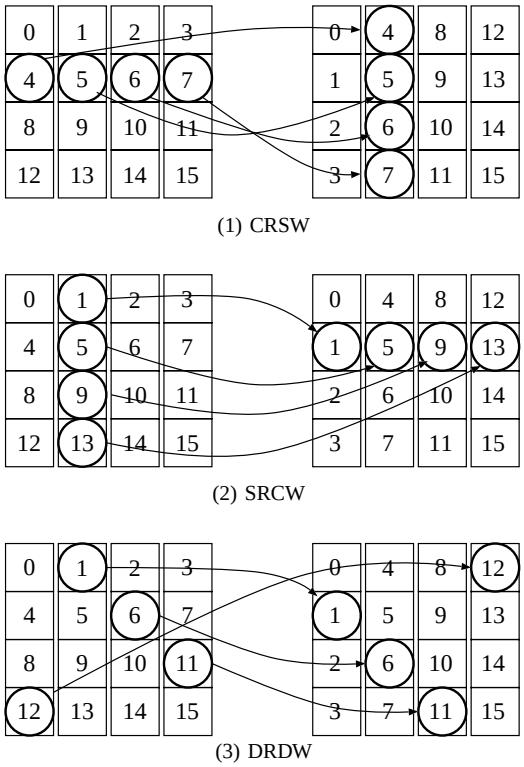

Figure 5. Illustrating the three matrix transpose algorithms for $w = 4$

memory stores the values of a matrix. In the RAW implementation, a CUDA block with 1024 threads are invoked. The value of "threadIdx.x" is a thread ID and takes value from 0 to 1023. The value of $a[i][j]$ is copied $b[j][i]$ by a thread with thread ID $i \cdot 32 + j$.

IV. The random address permute-shift (RAP) technique

The main purpose of this section is to present a novel technique, *the random address permute-shift (RAP)*, in which the memory access congestion for stride access is reduced to 1. Further, the memory access congestion by the RAP is still $\tilde{O}(\frac{\log w}{\log \log w})$ for any memory access by a warp of w threads.

Let a be a matrix of size $w \times w$ on the DMM. Note that each $a[i][j]$ is in bank $B[j]$ of the DMM. The key idea of the RAP is to use a random permutation of $(0, 1, \ldots w - 1)$. Suppose that each of w threads in a warp accesses an element of a at the same time. If all w elements are in distinct banks, the congestion is 1. On the other hand, the congestion is w if they are in the same bank. We will show that, using the RAP, the expected value of the congestion is at most $O(\frac{\log w}{\log \log w})$ for any memory access by w threads including malicious ones.

Let r be a permutation of $(0, 1, \ldots, w-1)$ selected from all possible $w!$ permutations uniformly at random. Hence, $r_0, r_1, \ldots, r_{w-1}$ take distinct integers in the range $[0, w-1]$. Intuitively, the random address permute-shift technique rotates each i-th row ($0 \leq i \leq w-1$) of matrix a by r_i. In other words, each $a[i][j]$ ($0 \leq i, j \leq w$) is mapped to $a[i][(j + r_i) \bmod w]$. If a thread try to access $a[i][j]$, it accesses $a[i][(j + r_i) \bmod w]$ instead. Hence, $a[i][j]$ is arranged in bank $B[(j + r_i) \bmod w]$ of the DMM. Figure 6 illustrates an example of the RAP for $w = 4$, where we select $r = (2, 0, 3, 1)$. For example, $a[10](= a[2][2])$ is mapped to $a[9](= a[2][(2 + 3) \bmod 4])$ in $B[1]$.

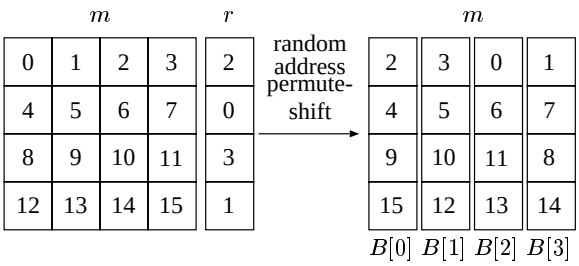

Figure 6. An example of the random address permute-shift

Recall that a memory access by a warp is *contiguous* if all w threads in a warp access the same row, and it is *stride* if all threads in a warp access the same column. Clearly, the congestion of the contiguous access is always 1, because $(0+r_i) \bmod w$, $(1+r_i) \bmod w$, ..., $(w-1+r_i) \bmod w$ are distinct. Also, that of the stride is 1, because $(j+r_0) \bmod w$, $(j + r_1) \bmod w$, ..., $(j + r_{w-1}) \bmod w$ are distinct. In our previous paper [7], we have presented the random address shift (RAS) technique, which uses independent random numbers $r_0, r_1, \ldots$ instead of a random permutation used by the RAP. Clearly, the stride access by the RAS involves bank conflicts with high probability, while that of the RAP is always 1.

We will show that, by the RAP, the congestion of the row-wise access and the column-wise access is 1. Further, the congestion of any memory access is $\tilde{O}(\frac{\log w}{\log \log w})$. More specifically, we prove the following important theorem:

Theorem 2: By the RAP, the congestion is $\tilde{O}(\frac{\log w}{\log \log w})$ for any memory access by a warp. In particular, the congestion of the contiguous access and the stride access is 1.

We will prove that the congestion of any memory access is at most $\tilde{O}(\frac{\log w}{\log \log w})$. For the purpose of the proof, we use an important probability theorem called the Chernoff bound that estimates the tail probability of the Poisson trials as follows:

Theorem 3 (Chernoff Bound [20]): Let $X_0, X_1, \ldots, X_{n-1}$ be independent Poison trials such that $X_i = 1$ with probability p_i ($0 \leq i \leq n-1$). Let $X = \sum_{i=0}^{n-1} X_i$ and $\mu = E[X] = \sum_{i=0}^{n-1} p_i$. We have the following inequality for any $\delta > 0$:

$$\Pr[X > (1+\delta)\mu] \quad < \quad \left(\frac{e^{\delta}}{(1+\delta)^{(1+\delta)}}\right)^{\mu}$$

Please see [20] for the details of the Chernoff bound. In paper [7] we use Theorem 3 to prove Theorem 2 for the RAS. This is possible because random numbers r_0, r_1, ... used by the RAS are independent. However, these random numbers by the RAP are not independent. Hence, it is not possible to use Theorem 3 as it is for the proof of Theorem 2. We use several new proof techniques to prove Theorem 2 by Theorem 3.

For simplicity, we assume that no two threads access the same address. Clearly, this assumption makes sense for the proof of Theorem 2, because it does not decrease the probability of bank conflicts and the memory access congestion. We partition w threads in a warp into two half warps such that each half warp has $\frac{w}{2}$ threads. We will show that the memory access congestion by $\frac{w}{2}$ threads in a half warp is $\tilde{O}(\frac{\log w}{\log\log w})$. This implies that the congestion by w threads in a warp is at most $2 \cdot \tilde{O}(\frac{\log w}{\log\log w}) = \tilde{O}(\frac{\log w}{\log\log w})$. Let $i_0, i_1, \ldots, i_{\frac{w}{2}-1}$ and $j_0, j_1, \ldots, j_{\frac{w}{2}-1}$ be the indexes of a such that each thread $T(k)$ ($0 \leq k \leq \frac{w}{2}-1$) by a half warp accesses $a[i_k][j_k]$. Using the RAP technique, each $T(k)$ accesses $a[i_k][(j_k + r_{i_k}) \bmod w]$ instead. Let $A(i)$ ($0 \leq i \leq w-1$) be the number of memory access requests destined for the i-th row of a. Since no two threads access the same address, we have $\sum_{i=0}^{w-1} A(i) = \frac{w}{2}$.

For a fixed bank $B[u]$ ($0 \leq u \leq w-1$), we will show that more than $\frac{e^2 \ln w}{\ln\ln w}$ memory requests is destined for $B[u]$ with probability at most $\frac{1}{w^2}$. Let i_0, i_1, ..., $i_{w'-1}$ ($0 \leq i_0 < i_1 < \cdots < i_{w'-1} \leq w-1$) denote rows accessed by $\frac{w}{2}$ threads in a half warp. In other words, $A(i_k) \geq 1$ for all k ($0 \leq k \leq w'-1$) and $w' \leq \frac{w}{2}$. Imagine that r_{i_0}, r_{i_1}, ..., $r_{i_{w'-1}}$ are determined one by one for the purpose of evaluating the congestion. In other words, each r_{i_k} is selected from integers in $[0, w-1] - \{r_{i_0}, r_{i_1}, \ldots, r_{i_{k-1}}\}$ at random. First, let us evaluate the probability that a half warp accesses $B[u]$ by memory access requests in the i_0-th row. Since $A(i_0)$ memory cells in the i_0-th row are accessed, the probability is $\frac{A(i_0)}{w}$. Next, we evaluate the probability that a half warp accesses $B[u]$ in the i_1-th row. Since r_{i_1} is selected from $[0, w-1] - \{r_{i_0}\}$ at random, the probability is 0 at most $\frac{A(i_1)}{w-1}$. Similarly, the probability that a half warp accesses $B[u]$ in the i_2-th row is at most $\frac{A(i_2)}{w-2}$, because r_{i_2} is selected from $[0, w-1] - \{r_{i_0}, r_{i_1}\}$ at random, In general, the probability that a half warp accesses $B[u]$ in the i_k-th row is at most $\frac{A(i_k)}{w-k}$ for each k ($0 \leq k \leq w'-1$). From $w' \leq \frac{w}{2}$, we have $\frac{A(i_k)}{w-k} \leq \frac{2A(i_k)}{w}$. To evaluate the number of memory cells in $B[u]$ accessed by a half warp, let $X_0, X_1, \ldots, X_{w'-1}$ be independent random binary variables such that $X_k = 1$ with probability $\frac{2A(i_k)}{w}$. Further, let $X = X_0 + X_1 + \cdots + X_{w'-1}$. Clearly, X is the random variable that provides the upper bound of the number of memory access destined for bank $B[u]$ by a half warp. Since random variables $X_0, X_1, \ldots, X_{w'-1}$ are independent, we can apply Theorem 3 to evaluate the tail probability of X and we have the following lemma:

Lemma 4: For random variable X defined above, we have,

$$\Pr[X > \frac{e^2 \ln w}{\ln\ln w}] \quad < \quad \frac{1}{w^2}.$$

Proof: Clearly, the expected value of X is

$$\mu \quad = \quad E[X] = \sum_{k=0}^{w'-1} \frac{2A(i_k)}{w} = 1.$$

Hence, from Theorem 3 with $\mu = 1$, we have

$$\Pr[X > (1+\delta)] \quad < \quad \frac{e^\delta}{(1+\delta)^{(1+\delta)}}$$

for any $\delta > 0$. Let $1 + \delta = \frac{e^2 \ln w}{\ln\ln w}$. We will prove that $\frac{e^\delta}{(1+\delta)^{(1+\delta)}} < \frac{1}{w^2}$, that is, $\ln \frac{e^\delta}{(1+\delta)^{(1+\delta)}} < -2\ln w$ as follows:

$$\begin{aligned}
&\ln \frac{e^\delta}{(1+\delta)^{(1+\delta)}} \\
&= \quad \delta - (1+\delta)\ln(1+\delta) \\
&= \quad \frac{e^2 \ln w}{\ln\ln w} - 1 - \frac{e^2 \ln w}{\ln\ln w} \ln \frac{e^2 \ln w}{\ln\ln w} \\
&< \quad -\frac{e^2 \ln w}{\ln\ln w}(-1 + \ln e^2 + \ln\ln w - \ln\ln\ln w) \\
&< \quad -\frac{e^2 \ln w}{\ln\ln w} \cdot \frac{\ln\ln w}{2} < -2\ln w
\end{aligned}$$

This completes the proof. ■

Let Y be a random variable denoting the memory access congestion by $\frac{w}{2}$ threads in a half warp. In other words, Y is the maximum number of memory access requests over all banks $B[u]$ ($0 \leq u \leq w-1$). From Lemma 4, we have

$$\Pr[Y > \frac{e^2 \ln w}{\ln\ln w}] \leq \Pr[X > \frac{e^2 \ln w}{\ln\ln w}] \cdot \frac{w}{2} < \frac{1}{2w}.$$

Thus, we have,

$$\Pr[0 \leq Y \leq \frac{e^2 \ln w}{\ln\ln w}] < 1 \text{ and } \Pr[\frac{e^2 \ln w}{\ln\ln w} < Y \leq w] < \frac{1}{2w}.$$

Hence, the expected value of Y is at most:

$$\begin{aligned}
E[Y] \quad &\leq \quad \Pr[0 \leq Y \leq \frac{e^2 \ln w}{\ln\ln w}] \cdot \frac{e^2 \ln w}{\ln\ln w} \\
&\quad + \Pr[\frac{e^2 \ln w}{\ln\ln w} < Y \leq w] \cdot w \\
&< \quad 1 \cdot \frac{e^2 \ln w}{\ln\ln w} + \frac{1}{2w} \cdot w = O(\frac{\log w}{\log\log w}).
\end{aligned}$$

We have proved that the congestion of any memory access by a half warp is $\tilde{O}(\frac{\log w}{\log\log w})$ by the RAP. Since the congestion of a warp is not more than the sum of those of the first warp and the second warp, we have Theorem 2.

V. Simulation results

The main purpose of this section is to show simulation results to evaluate the congestions. We have shown in Theorem 2, the memory access congestion by the RAP is $\tilde{O}(\frac{\log w}{\log \log w})$ for any memory access. Sometimes, the big-O notation has a large constant factor, and the actual value is too large to use it in practical applications. We will show that the actual value here is enough small for the purpose of practical GPU implementations.

Table II shows the congestion of memory access obtained by simulation. The number w of memory banks and threads in a warp of latest CUDA-enabled GPUs is 32. The value of w may be increased in future GPUs. Thus, we have performed simulation for various values of w up to 256.

From the table, we can confirm that the contiguous memory access has no bank conflict for all implementations. The RAW and RAS implementations for the stride access involve bank conflicts while the RAP implementation has no bank conflict. Since the diagonal access is optimized for the RAW implementation, the congestion of the RAW implementation is 1. On the other hand, the RAS and the RAP implementations have bank conflicts, but the congestion is moderately small. Also, the congestion by the RAP is slightly larger than that by the RAS. For example, when $w = 32$, the congestion by the RAP is 3.61 while that by the RAS is 3.53. This is because the probability of bank conflicts by the RAP is slightly larger than the RAS. For example, two memory access requests to distant addresses are destined for the same bank with probability $\frac{1}{w}$ if the RAS is used. On the other hand, the probability is $\frac{1}{w-1}$ if the RAP is used. For the random memory access, all three implementations have the same congestion for each w.

Consequently, we can say that the congestion of the RAP implementation is smaller than or equal to that of the RAW and the RAS implementations for most memory accesses excluding the diagonal memory access. The RAP has the largest congestion for the diagonal memory access, but the overhead is small. Hence, we should use the RAP implementation if

- addresses accessed by threads are not known beforehand, or
- a CUDA developer has difficulty to minimize bank conflicts.

A CUDA developer can reduce the congestion automatically in most cases by using the RAP.

VI. Experimental results using GTX GeForce TITAN

The main purpose of this section is to show how we have implemented the RAP in a CUDA-enabled GPU. We also show the experimental results for matrix transpose by the RAW, the RAS, and the RAP implementations.

We can implement the three matrix transpose algorithms using the RAP technique. For example, the CRSW transpose algorithm by the RAP for a matrix of size 32×32 is described as follows:

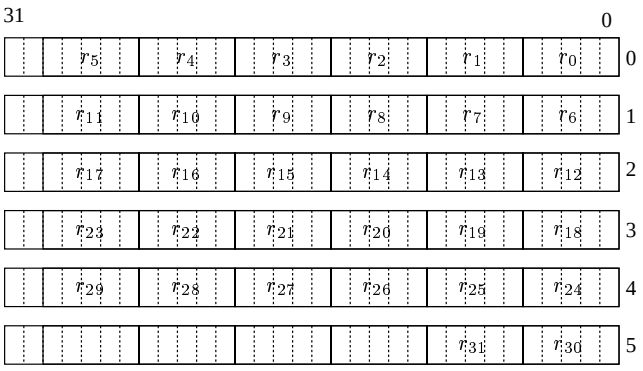

Figure 7. Arrangement of random numbers $r_i (0 \leq i \leq 31)$ in local registers r[*]

[The RAP implementation of the CRSW transpose]

```
__shared__ double a[32][32], b[32][32];
int r[6];
int i = threadIdx.x/32;
int j = threadIdx.x%32;
b[(j+(r[i/6]>>(5*(i%6))))&0x1f][i]
  = a[i][(j+(r[i/6]>>(5*(i%6))))&0x1f];
```

The transpose is performed for matrix a of size 32×32 and the resulting values are written in matrix b. We assume that $a[i][(j + r_i) \bmod 32]$ $(0 \leq i, j \leq w-1)$ stores the (i, j) elements of a matrix. Also, an array r of six local registers stores random numbers $r_0, r_1, \ldots, r_{31}$ in the range $[0, 31]$ such that each $r[i]$ $(0 \leq i \leq 5)$ stores 6 random numbers $r_{i\cdot 6}, r_{i\cdot 6+1}, \ldots, r_{i\cdot 6+5}$. Since each $r[i]$ has 32 bits and each r_j has 5 bits, this is possible. The reader should refer to Figure 7 illustrating how random numbers r are stored in local registers $r[*]$. Since the value of $a[i][(j + r_i) \bmod 32]$ is copied to $b[(j + r_i) \bmod 32][i]$, the transpose is completed correctly.

We have implemented three matrix transpose algorithms, Contiguous Read Stride Write (CRSW), Stride Read Contiguous Write (SRCW), and Diagonal Read Diagonal Write (DRDW) using the RAW, the RAS, and the RAP. For the CRSW and the SRCW transpose, the RAP implementation runs about 10 times faster than the RAW implementation and 2 times faster than the RAS implementation. The DRDW transpose performs the worst memory access for the RAP implementation. Nonetheless, it runs only 2.5 times slower than the RAW implementation. It follows that, the RAP is practical and works efficiently in current GPUs. When a CUDA developer implements some algorithm in the GPUs, it is not necessary to analyze and reduce the memory access congestion. It is sufficient to apply the RAP and the resulting implementation has small memory access congestion.

Table II
THE CONGESTION OF MEMORY ACCESS TO A MATRIX OF SIZE $w \times w$

	RAW Implementation					RAS Implementation					RAP Implementation				
w	16	32	64	128	256	16	32	64	128	256	16	32	64	128	256
Contiguous	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
Stride	16	32	64	128	256	3.08	3.53	3.96	4.38	4.77	1	1	1	1	1
Diagonal	1	1	1	1	1	3.08	3.53	3.96	4.38	4.77	3.20	3.61	4.00	4.41	4.78
Random	2.92	3.44	3.90	4.34	4.75	2.92	3.44	3.90	4.34	4.75	2.92	3.44	3.90	4.34	4.75

Table III
THE CONGESTION ON THE DMM AND THE COMPUTING TIME ON THE GPU FOR CRSW, SRCW, AND DRDW ALGORITHMS BY THE RAW, THE RAS, AND THE RAP IMPLEMENTATIONS

	RAW Implementation			RAS Implementation			RAP Implementation		
	congestion		time (in ns)	congestion		time (in ns)	congestion		time (in ns)
	read	write	on the GPU	read	write	on the GPU	read	write	on the GPU
CRSW Transpose	1	32	1595	1	3.53	303.6	1	1	154.5
SRCW Transpose	32	1	1596	3.53	1	297.1	1	1	159.1
DRDW Transpose	1	1	158.4	3.53	3.53	427.4	3.61	3.61	433.3

VII. THE RANDOM ADDRESS PERMUTE-SHIFT FOR HIGHER DIMENSION

So far, we have presented the random address permute-shift (RAP) for a matrix of size $w \times w$. The main purpose of this section is to discuss several ideas to extend the RAP for larger arrays than $w \times w$. For simplicity, we focus on a 4-dimensional array a of size $w \times w \times w \times w$. Note that, each element $a[i][j][k][l]$ ($0 \leq i, j, k, l \leq w-1$) is allocated in address $i \cdot w^3 + j \cdot w^2 + k \cdot w + l$ and it is in bank $B[l]$.

Basically, the RAP technique maps each $a[i][j][k][l]$ to $a[i][j][k][(l + f(i, j, k)) \bmod w]$ for some linear function f that determines the length of shift from i, j, and k. We use several memory accesses by a warp of w threads to evaluate the congestion as follows:

Contiguous: $a[i][j][k][0 : w-1]$, that is, $a[i][j][k][0]$, $a[i][j][k][1]$, ..., $a[i][j][k][w-1]$ are accessed.
Stride1: $a[i][j][0 : w-1][l]$ are accessed.
Stride2: $a[i][0 : w-1][k][l]$ are accessed.
Stride3: $a[0 : w-1][j][k][l]$ are accessed.
Random: w elements in a are selected at random are accessed.
Malicious: Malicious memory accesses that maximize the memory access congestion.

We introduce several possible RAP techniques as follows:

One permutation (1P): For a random permutation r of $(0, 1, \ldots, w-1)$, $f(i, j, k) = r_k$.
Repeated one permutation (R1P): For a random permutation r of $(0, 1, \ldots, w-1)$, $f(i, j, k) = r_i + r_j + r_k$.
Three random permutations (3P): For three independent random permutations r, s, and t of $(0, 1, \ldots, w-1)$, $f(i, j, k) = r_i + s_j + t_k$.
w^2 random permutations (w^2P): For w^2 random permutations $r^0, r^1, \ldots r^{w^2-1}$ of $(0, 1, \ldots, w-1)$, $f(i, j, k) = r_k^{i \cdot w + j}$.
one random permutation and w^2 random numbers (1Pw^2R): For a random permutation r of $(0, 1, \ldots, w-1)$ and w^2 independent random integers $s_0, s_1, \ldots, s_{w^2-1}$ in $[0, w-1]$, $f(i, j, k) = s_{i \cdot w + j} + r_k$.

Table IV summarizes the congestion and used random numbers by each RAP for an array of size w^4. Due to the stringent page limitation, we omit the details of explanation, but the reader should have no difficulty to confirm that the congestion and the random numbers are correct. From the table, we can see that R1P and 3P have good performance in terms of the congestion and the used random numbers. Note that R1P have malicious inputs with high congestion. For example, 6 memory access requests to $a[0][1][2][l]$, $a[0][2][1][l]$, $a[1][0][2][l]$, $a[1][2][0][l]$, $a[2][0][1][l]$, and $a[2][1][0][l]$ are destined to bank $B[(l + r_0 + r_1 + r_2) \bmod w]$. Since we can have $\frac{w}{6}$ groups of such 6 memory access requests each, the congestion can be as large as $6 \cdot \tilde{O}(\frac{\log \frac{w}{6}}{\log \log \frac{w}{6}})$, which is much larger than $\tilde{O}(\frac{\log \frac{w}{6}}{\log \log \frac{w}{6}})$ from the practical point of view. Thus, we believe that 3P is the best method to extend the RAP for larger arrays.

VIII. CONCLUSION

We have presented a novel algorithmic technique called the random address permute-shift (RAP) that achieves $\tilde{O}(\frac{\log w}{\log \log w})$ memory access congestion for any memory access requests by a warp of w threads. In particular, the congestion of any contiguous and any stride memory access is always 1. We have also applied the RAP to matrix transpose on the shared memory of a streaming multiprocessor on the GeForce GTX TITAN. The experimental results show that, even the direct transpose algorithms run very fast by the RAP technique. From the experimental results, we can say that our new RAP technique is practical and a potent

Table IV
THE CONGESTION AND THE USED RANDOM NUMBERS BY THE RAW, THE RAS, AND THE RAPS FOR AN ARRAY OF SIZE w^4

	RAW	RAS	RAP				
			1P	R1P	3P	w^2P	1Pw^2R
Contiguous	1	1	1	1	1	1	1
Stride1	w	$\tilde{O}(\frac{\log w}{\log\log w})$	1	1	1	1	1
Stride2	w	$\tilde{O}(\frac{\log w}{\log\log w})$	w	1	1	$\tilde{O}(\frac{\log w}{\log\log w})$	$\tilde{O}(\frac{\log w}{\log\log w})$
Stride3	w	$\tilde{O}(\frac{\log w}{\log\log w})$	w	1	1	$\tilde{O}(\frac{\log w}{\log\log w})$	$\tilde{O}(\frac{\log w}{\log\log w})$
Random	$\tilde{O}(\frac{\log w}{\log\log w})$	$\tilde{O}(\frac{\log w}{\log\log w})$	$\tilde{O}(\frac{\log w}{\log\log w})$	$\tilde{O}(\frac{\log w}{\log\log w})$	$\tilde{O}(\frac{\log w}{\log\log w})$	$\tilde{O}(\frac{\log w}{\log\log w})$	$\tilde{O}(\frac{\log w}{\log\log w})$
Malicious	w	$\tilde{O}(\frac{\log w}{\log\log w})$	w	$6\cdot\tilde{O}(\frac{\log\frac{w}{6}}{\log\log\frac{w}{6}})$	$\tilde{O}(\frac{\log w}{\log\log w})$	$\tilde{O}(\frac{\log w}{\log\log w})$	$\tilde{O}(\frac{\log w}{\log\log w})$
Random numbers	0	w^3	w	w	$3w$	w^3	w^2+w

method to reduce the memory access congestion for the shared memory automatically.

REFERENCES

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

[2] K. Ogawa, Y. Ito, and K. Nakano, "Efficient Canny edge detection using a GPU," in *Proc. of International Conference on Networking and Computing*. IEEE CS Press, Nov. 2010, pp. 279–280.

[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] ——, "NVIDIA CUDA C best practice guide version 3.1," 2010.

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

[7] K. Nakano, S. Matsumae, and Y. Ito, "The random address shift to reduce the memory access congestion on the discrete memory machine," in *Proc. of International Symposium on Computing and Networking*, Dec. 2013, pp. 95–103.

[8] A. Kasagi, K. Nakano, and Y. Ito, "Offline permutation algorithms on the discrete memory machine with performance evaluation on the GPU," *IEICE Transactions on Information and Systems*, vol. Vol. E96-D, no. 12, pp. 2617–2625, Dec. 2013.

[9] K. Nakano, "Sequential memory access on the unified memory machine with application to the dynamic programming," in *Proc. of International Symposium on Computing and Networking*, Dec. 2013, pp. 85–94.

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

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

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

[13] 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.

[14] ——, "An optimal offline permutation algorithm on the hierarchical memory machine, with the GPU implementation," in *Proc. of International Conference on Parallel Processing*. IEEE CS Press, Oct. 2013, pp. pp. 1–10.

[15] K. Nakano and S. Matsumae, "The super warp architecture with random address shift," in *Proc. of High Performance Computing*, Dec. 2013.

[16] K. Mehlhorn and U. Vishkin, "Randomized and deterministic simulations of PRAMs by parallel machines with restricted granularity of parallel memories," *Acta Informatica*, vol. 21, no. 4, pp. 339 – 374, Nov. 1984.

[17] M. Dietzfelbinger and F. M. auf der Heide, "Simple, efficient shared memory simulations," in *Proc.of ACM Symposium on Parallel Algorithms and Architectures*, June 1993, pp. 110 – 119.

[18] NVIDIA Corporation. (2013) NVIDIA GeForce GTX TITAN. [Online]. Available: http://www.geforce.com/hardware/desktop-gpus/geforce-gtx-titan/

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

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