The Random Address Shift to Reduce the Memory Access Congestion on the Discrete Memory Machine

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 of the 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 memory access congestion takes value between 1 and w. The main contribution of this paper is to present a novel algorithmic technique called the random address shift that reduces the memory access congestion. We show that the memory access congestion is expected $O(\frac{\log w}{\log \log w})$ for any memory access requests including malicious ones by a warp of w threads. The simulation results show that the expected congestion for $w = 32$ threads is only 3.436. Since the malicious memory access requests destined for the same bank take congestion 32, our random address shift technique substantially reduces the memory access congestion. We have applied the random address shift technique to matrix transpose algorithms. The experimental results on GeForce GTX Titan show that the random address shift technique is practical and can accelerate the straightforward matrix transpose algorithms by a factor of 5.**

***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], [4]. 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], [3], [5], [6], [7]. NVIDIA provides a parallel computing architecture called *CUDA* (Compute Unified Device Architecture) [8], the computing engine for NVIDIA GPUs. CUDA gives developers access to the virtual instruction set and memory of the parallel computational elements in NVIDIA GPUs. In many cases, GPUs are more efficient than multicore processors [2], 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 in the NVIDIA GPUs: *the shared memory* and *the global memory* [8]. Each SM has the shared memory, an extremely fast on-chip memory with lower capacity, say, 16-48 Kbytes, and low latency. Every SM shares the global memory implemented as an off-chip DRAM, and has large capacity, say, 1.5-6 Gbytes, but its access latency is 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 *the bank conflict* of the shared memory access and *the coalescing* of the global memory access [9]. 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.

The most well-studied parallel computing model is the Parallel Random Access Machine (PRAM) [10], [11], [12], which consists of processors and a shared memory. Each processor on the PRAM can access any address of the shared memory in a time unit. The PRAM is a good parallel computing model in the sense that parallelism of each problem can be revealed by the performance of parallel algorithms on the PRAM. GPUs have the shared memory and the global memory accessed by multiple threads. However, parallel

algorithms developed for the PRAM may not achieve good performance on GPUs. We should consider the memory access characteristics such as the bank conflicts and the coalescing when we develop efficient parallel algorithms for GPUs.

In our previous paper [13], 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 and the UMM [14], [15], [16], [17], [18], [19], [20]. 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 is a Random Access Machine (RAM) [21], which can execute fundamental operations in a time unit. Threads are executed in SIMD [22] fashion, and run on the same program and work on the 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 [23] 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.

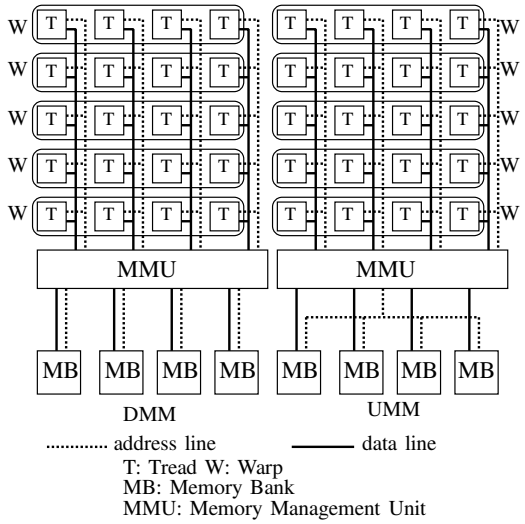

Figure 1. The architectures of the DMM and the UMM with width $w = 4$

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.

It is very important for developing efficient algorithms on the DMM to reduce *the memory access congestion*, the maximum number of 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 [13]. Further, in our previous paper [13], we have developed a complicated graph coloring technique to minimize the memory access congestion for off-line permutation. We have implemented this offline permutation algorithm on GeForce GTX-680 GPU [14]. 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 [14]. Although it is very important to minimize the memory access congestion, it is not easy.

The main contribution of this paper is to present a novel algorithmic technique called *the random address shift* to reduce the memory access congestion on the DMM. Basically, the random address shift technique is inspired by parallel hashing that averages the access to memory modules [24], [25]. Quite surprisingly, for any memory access requests by a warp of w threads, the memory access

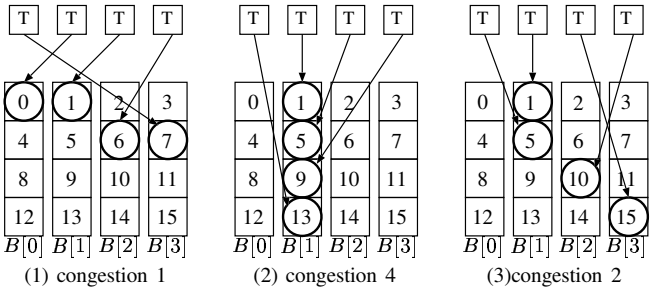

Figure 2. Examples of memory access and the congestion for $w = 4$

congestion is expected $O(\frac{\log w}{\log \log w})$ using the random address shift technique. In particular, if threads perform contiguous memory access, the congestion is still 1. Using the random address shift technique, parallel algorithm developers for the GPU do not have to take care of the memory access congestion. We also show that the expected value of the memory access congestion by simulation. The simulation results show that the expected value is less than 3.5 for current GPUs with $w = 32$. Even if the number of memory banks is increased to $w = 256$ in the distant future, it is less than 5.

Further, we have implemented the random address shift technique in a streaming multiprocessor on GeForce GTX TITAN [26] which supports CUDA Compute Capability 3.5 [8]. 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 a matrix. The SRCW reads a matrix in column major order and writes in row major order. Both the CRSW and the SRCW involve memory access with congestion w, and these algorithms take a lot of time. The DRDW performs reading and writing in diagonal order to reduce the memory access congestion to 1. Thus, the DRDW runs much faster than the others. However, it may not be easy for CUDA developers to find an efficient algorithm such as the DRDW for complicated problems. We have applied the random address shift technique for these three algorithm. The resulting algorithms runs almost the same running time. Also, their running time are much faster than the CRSW and the SRCW implemented as they are. It follows that, the random address shift is practical and works efficiently in current GPUs. When a user implements some algorithm in the GPUs, it is not necessary to analyze and reduce the memory access congestion. It is sufficient to apply the random address shift technique, and the resulting implementation has small memory access congestion.

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 of the random address shift technique. In Section IV, we present the random address shift technique and evaluate the memory access congestion by theoretical analysis as well as by simulation. Section V, we show experimental results on GeForce GTX TITAN. Section VI 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 [13]. The reader should refer to [13] 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 [22] 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 $W(0)$

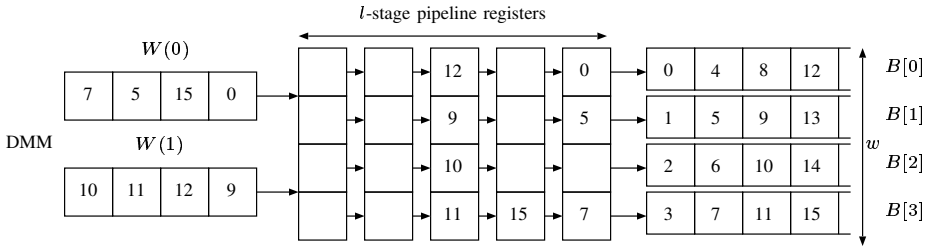

Figure 3. The Discrete Memory Machine (DMM)

are separated into two pipeline stages, because $m[7]$ and $m[15]$ are in the same bank $B[3]$. Those by $W(1)$ occupies 1 stage, because all requests are destined for distinct banks, one request for each bank. Thus, the memory requests occupy three stages, and it takes $3 + 5 - 1 = 7$ time units to complete the memory access.

Let us define *the congestion* of memory access by a warp of w threads. Suppose that a warp of w threads access the memory banks. The memory access congestion is the maximum number of requests destined for the same bank. More specifically, if x_i memory requests are destined for each $B[i]$ $(0 \leq i \leq w-1)$, then the congestion is $\max\{x_i \mid 0 \leq i \leq w-1\}$. For example, the congestion of memory access by $W(0)$ in Figure 3 is 2, because two requests are destined for $m[7]$ and $m[15]$ in bank $B[3]$. That by $W(1)$ is 1 because all requests are destined for distinct banks. 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 [10].

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, *the contiguous access*, *the stride access* and *the diagonal access* [13]. 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 $w = 4$.

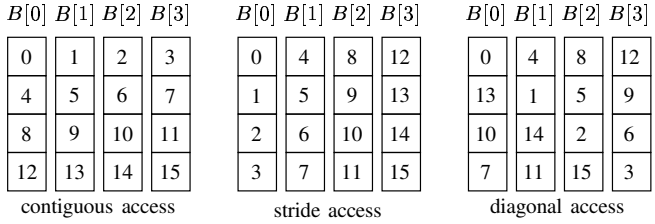

Figure 4. The contiguous access, the stride access, and the diagonal access for $w = 4$

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
  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, 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 contiguous 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 the stride read and the stride 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 $w \times w$ takes $O(w^2 + l)$ 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 the streaming multiprocessor of the GPU as they are. 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];
int i = threadIdx.x/32;
int j = threadIdx.x%32;
double c;
c = a[i][j];
__syncthreads();
a[j][i] = c;
__syncthreads();
```

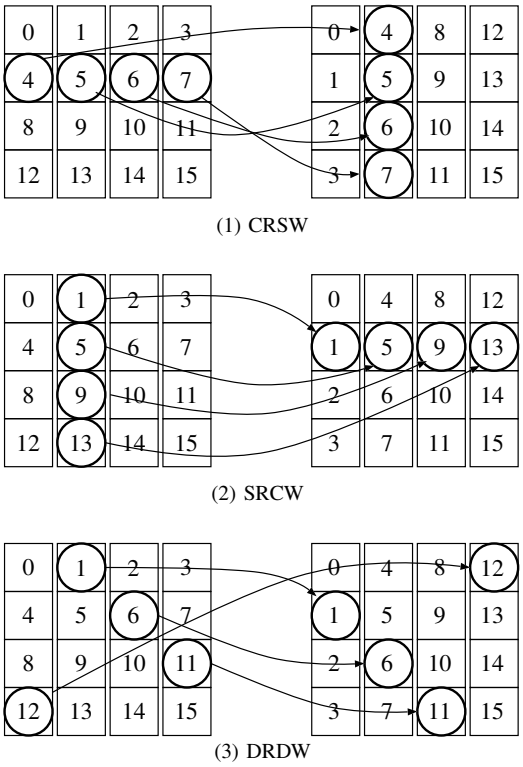

Figure 5. Illustrating the three matrix transpose algorithms for $w = 4$

We assume that a 2-dimensional array a allocated in the shared 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 to the local register c of thread ID with $i \cdot 32 + j$. After that, a barrier synchronization operation __syncthreads() is executed to make sure that all elements in a are copied to local registers in all threads. Finally, the value of the local register c is copied to $a[j][i]$. In Section V, we show experimental results of the RAW implementation on the GPU.

IV. The random address shift technique

The main purpose of this section is to present a novel technique that we call *the random address shift*. The memory access congestion is guaranteed to be expected $O(\frac{\log w}{\log \log w})$ for any memory access by a warp of w threads. In particular, the memory access congestion of the contiguous memory access is still 1 even if the random address shift is used.

Let m denote an array of size n on the DMM. We can consider that a 1-dimensional array m of size n is a 2-dimensional one of size $\frac{n}{w} \times w$. In other words, each $m[j][k]$ $(0 \leq j \leq \frac{n}{w} - 1, 0 \leq k \leq w-1)$ in the 2-dimensional context corresponds to $m[j \cdot w + k]$ in the 1-dimensional context. Note that each $m[j][k]$ is in bank $B[k]$ of the DMM. The key

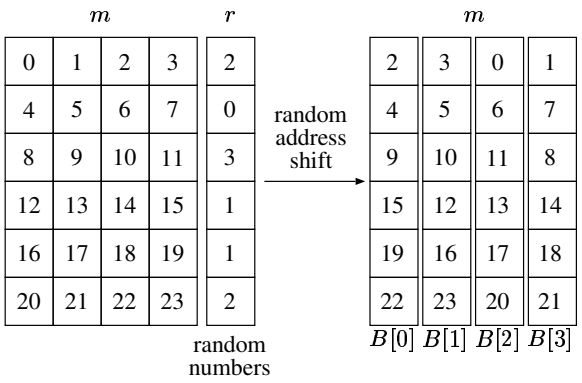

Figure 6. An example of the random address shift

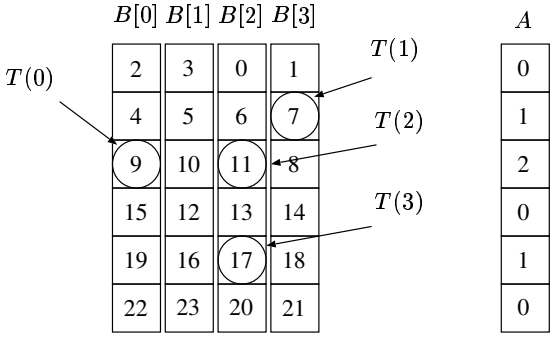

Figure 7. An example of memory access and the values of A

idea is to randomly rotate the address mapping to equalize the memory access requests destined for memory banks.

Suppose that each of w threads in a warp accesses an element of m 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 random address shift technique, 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 $r_0, r_1, \ldots, r_{\frac{n}{w}-1}$ denote independent random integers uniformly selected from $[0, w-1]$. Intuitively, the random address shift technique rotates each j-th row ($0 \leq j \leq \frac{n}{w} - 1$) of the 2-dimensional array m by r_j. More specifically, each $m[j][k]$ ($0 \leq j \leq \frac{n}{w} - 1, 0 \leq k \leq w$) is mapped to $m[j][(k + r_j) \bmod w]$. In other words, if a thread accesses $m[j][k]$, it accesses $m[j][(k + r_j) \bmod w]$ instead. Hence, $m[j][k]$ is arranged in bank $B[(k+r_j) \bmod w]$ of the DMM. Figure 6 illustrates an example of the random address shift for $n = 24$ and $w = 4$, where randomly selected integers r are 2, 0, 3, 1, 1, and 2. For example, $m[10](= m[2][2])$ is mapped to $m[9](= m[2][1])$ in $B[1]$.

We will prove that the expected value of the congestion is at most $O(\frac{\log w}{\log\log w})$. For simplicity, we assume that w threads always access distinct address. Clearly, this assumption does not decrease the congestion, because memory access requests to the same address by multiple threads are merged into one. For the proof, we use an important probability theory called the Chernoff bound that estimates the tail probability of the Poisson trials as follows:

Theorem 2 (Chernoff Bound [27]): 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 $\delta > 0$:

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

We are now in a position of evaluating the expected value of the congestion using the Chernoff bound. Let $j_0, j_1, \ldots, j_{w-1}$ and $k_0, k_1, \ldots, k_{w-1}$ be the indexes of m such that each thread $T(i)$ ($0 \leq i \leq w-1$) accesses $m[j_i][k_i]$. Using the random address shift technique, each $T(i)$ accesses $m[j_i][(k_i + r_{j_i}) \bmod w]$ instead. Let $A(j)$ ($0 \leq j \leq \frac{n}{w} - 1$) be the number of memory access requests destined for the j-th row of m. Figure 7 shows an example of memory access by $w = 4$ threads and the values of A. Clearly, $\sum_{j=0}^{\frac{n}{w}-1} A(j) = w$.

We fix a particular bank $B[u]$ ($0 \leq u \leq w-1$) and evaluate the number of memory access requests destined for $B[u]$ for random selection of $r_0, r_1, \ldots, r_{\frac{n}{w}-1}$. Let $X_0, X_1, \ldots, X_{\frac{n}{w}-1}$ be a random binary variable such that $X_j = 1$ iff $m[j][u]$ ($0 \leq j \leq \frac{n}{w} - 1$) is accessed by at least one of the w threads. Clearly, $X_j = 1$ with probability $\frac{A(j)}{w}$, because $A(j)$ elements in the j-th row of m are accessed. Since $r_0, r_1, \ldots, r_{\frac{n}{w}-1}$ are independent, random variables $X_0, X_1, \ldots, X_{\frac{n}{w}-1}$ are also independent. Thus, Theorem 2 can be used to evaluate the value of $X = \sum_{i=0}^{\frac{n}{w}-1} X_i$, which is equal to the number of memory access requests destined for $B[u]$.

We prove the following lemma that evaluates the tail probability of X.

Lemma 3: 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, $\mu = E[X] = \sum_{i=0}^{\frac{n}{w}-1} \frac{A(j)}{w} = 1$ holds. Hence, from Theorem 2 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:

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

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

This completes the proof. ■

Let Y be a random variable denoting the memory access congestion, which is the maximum number of memory access requests over all banks $B[u]$ $(0 \leq u \leq w-1)$. From Lemma 3, we have

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

Thus, we have,

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

Hence, the expected value of Y is at most:

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

Consequently, we have the following important theorem:

Theorem 4: By the random address shift, the congestion of any memory access by a warp of w threads is expected $O(\frac{\log w}{\log \log w})$.

Recall that the congestion of the contiguous memory access of the RAW implementation is 1. We will show that, even if we use the random address shift technique, the congestion of the contiguous memory access is still 1. In the contiguous memory access, each warp $W(i)$ $(0 \leq i \leq w-1)$ accesses w elements of the i-th row of matrix a. Hence, by the random address shift, it still accesses the i-th row. Thus, the w threads in $W(i)$ access different banks and we have,

Theorem 5: By the random address shift, the congestion of the contiguous memory access by a warp of w threads is still 1.

As shown in Theorem 4 the congestion of any memory access by a warp is $O(\frac{\log w}{\log \log w})$ by the random address shift. In other words, memory access by a warp occupies $O(\frac{\log w}{\log \log w})$ pipeline registers. Hence, the contiguous access, the stride access, and the diagonal access of a matrix of size $w \times w$ take $O(\frac{w \log w}{\log \log w} + l)$ time units. Thus, we have,

Lemma 6: The CRSW, the SRCW, and the DRDW transpose algorithms for a matrix of size $w \times w$ run in $O(\frac{w \log w}{\log \log w} + l)$ time units by the random address shift using w^2 threads on the DMM with width w and latency l.

We next show that the actual value of $E[Y]$ is not large and Y has narrow distribution by simulation experiments. Table I shows these values obtained by 10,000,000 rounds of simulation, where a round is a single memory access by w threads. Since the number w of memory banks of current CUDA-enabled GPUs are 16 or 32 [8], we evaluate the value of $E[Y]$ for $w = 16, 32, 64, 128$ and 256 for considering future extension of GPUs. We use the size n of array m is 1024 $(= 2^{10})$ and 1048576 $(= 2^{20})$. For example, $E[Y] = 3.436$ when $w = 32$ and $n = 1024$. Also, $Y = 3$ with probability 55.988%. We can see that the values of $E[Y]$ is less than 5 even if $w = 256$ and that $Y \leq 5$ with very high probability for all cases.

V. Experimental results of the random address shift technique in a CUDA-enabled GPU

The main purpose of this section is to show how we have implemented the random address shift technique in a CUDA-enabled GPU. We also show the experimental results of our implementation.

We can implement the three matrix transpose algorithms using the random address shift technique. We call such implementation *RAS (Random Address Shift access to memory) implementation*. For example, the RAS implementation of the CRSW transpose algorithm for a matrix of size 32×32 is described as follows:

[The RAS implementation of the CRSW transpose]

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

We assume that a 2-dimensional array b stores the value of a matrix a such that each $b[i][(i+r_j) \bmod 32]$ $(0 \leq i, j \leq w-1)$ stores the value of $a[i][j]$. Also, a 1-dimensional 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 8 illustrating how random numbers r are stored in local registers r. In the RAS implementation, a CUDA block of 1024 threads are invoked similarly to the RAW implementation. The value of $b[i][(j+r_i) \bmod 32]$ is copied in the local register c of thread ID with $i \cdot 32 + j$. After that, a barrier synchronization operation __syncthreads() is executed to make sure that all elements in b are copied to local registers of all threads. Finally, the value of the local register c is copied to $b[j][(i+r_j) \bmod 32]$.

We have evaluated the performance of the RAW and the RAS implementations for the CRSW transpose, the SRCW transpose, and the DRDW transpose. We have used a square matrix of $n = 1024$ double float (64-bit) numbers in the shared memory of a streaming multiprocessor for experiments. Each warp of GeForce GTX Titan has 32

Table I
THE VALUES OF $E[Y]$ AND THE DISTRIBUTION OF Y IN PERCENT

n		1024 ($= 2^{10}$)					1048576 ($= 2^{20}$)				
w		16	32	64	128	256	16	32	64	128	256
$E[Y]$		3.038	3.436	3.713	3.808	3.458	3.078	3.533	3.958	4.378	4.766
Y	1	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000
	2	21.208	4.207	0.284	0.009	0.003	19.466	2.923	0.065	0.000	0.000
	3	57.378	55.988	40.620	30.505	54.203	57.415	51.635	27.335	6.914	0.444
	4	18.197	32.784	47.940	58.854	45.794	19.415	36.176	53.019	56.226	38.451
	5	2.862	6.158	9.912	9.936	0.000	3.263	7.924	16.404	30.052	47.452
	6	0.325	0.782	1.143	0.672	0.000	0.401	1.181	2.764	5.867	11.670
	7	0.028	0.075	0.095	0.024	0.000	0.037	0.145	0.368	0.830	1.741
	8	0.002	0.006	0.007	0.000	0.000	0.003	0.015	0.041	0.099	0.217
	9	0.000	0.000	0.000	0.000	0.000	0.000	0.001	0.004	0.011	0.023
	10	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.001	0.002

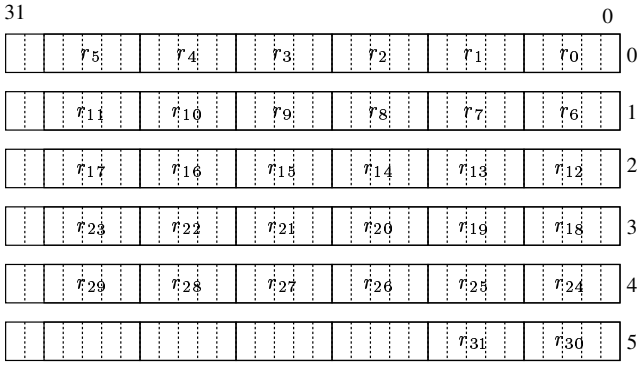

Figure 8. Arrangement of random numbers $r_i (0 \leq i \leq 31)$ in local registers $r[*]$

threads, and the number of banks in the shared memory is 32. Table II shows the experimental results including the congestion and the running time. Note that the expected value of the memory access congestion by the random address shift is 3.50. This value is a bit larger than 3.436 shown in Table I, because no two memory access request is destined for the same address in the experiments shown in Table II.

Clearly, if an algorithm has larger congestion, the running time is longer. Roughly speaking, we can see that (the total read/write congestions) $\times$ 40 + 100 gives good approximations of the running time in nanoseconds. For example, the RAS implementation of the DRDW transpose takes total congestion 2, and thus, the approximation value is 180, which is very close to actual running time 171.8ns.

From the experimental results, the RAS implementation for the CRSW transpose and the SRCW transpose is five times faster than the RAW implementation. However, for the DRDW transpose, the RAS implementation is slower. Hence, we can say that an algorithm involves large congestion can be accelerated by the RAS implementation. On the other hand, if an algorithm is carefully optimized to minimize the congestion, the RAS implementation may run slower. Hence, we should use the RAS implementation, if it is difficult or impossible to minimize the congestion.

VI. CONCLUSION

We have presented a novel algorithmic technique called the random address shift that achieves expected $O(\frac{\log w}{\log \log w})$ memory access congestion for any memory access requests by a warp of w threads. In particular, the congestion of the contiguous memory access is still 1. We have also applied the random address shift to the matrix transpose on the shared memory of a streaming multiprocessor on the GeForce GTX TITAN. The experimental results show that the theoretical analysis on the DMM provides a good approximation of the performance on the actual GPU. From the experimental results, we can say that the random address shift technique is practical and a potent method to reduce the memory access congestion for the shared memory.

REFERENCES

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

[2] 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.

[3] K. Ogawa, Y. Ito, and K. Nakano, "Efficient canny edge detection using a gpu," in *Proc. of International Conference on Networking and Computing*, Nov. 2010, pp. 279–280.

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

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

Table II
THE CONGESTION AND THE COMPUTING TIME ON THE DMM, AND THE COMPUTING TIME ON THE GPU OF TRANSPOSE BY RAW (RAW ACCESS TO MEMORY) AND RAS(RANDOM ADDRESS SHIFT ACCESS TO MEMORY)

	RAW Implementation				RAS Implementation			
	congestion		time	time (in ns)	congestion		time	time (in ns)
	read	write	on the DMM	on the GPU	read	write	on the DMM	on the GPU
CRSW Transpose	1	32	$O(w^2+l)$	1398	1	3.50	$O(\frac{w \log w}{\log \log w}+l)$	279.1
SRCW Transpose	32	1	$O(w^2+l)$	1399	3.50	1	$O(\frac{w \log w}{\log \log w}+l)$	278.7
DRDW Transpose	1	1	$O(w+l)$	171.8	3.50	3.50	$O(\frac{w \log w}{\log \log w}+l)$	389.0

[6] ——, "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)*, Sept. 2012, pp. 1–15.

[7] A. Uchida, Y. Ito, and K. Nakano, "An efficient GPU implementation of ant colony optimization for the traveling salesman problem," in *Proc. of International Conference on Networking and Computing*, Dec. 2012, pp. 94–102.

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

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

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

[11] A. Grama, G. Karypis, V. Kumar, and A. Gupta, *Introduction to Parallel Computing*. Addison Wesley, 2003.

[12] M. J. Quinn, *Parallel Computing: Theory and Practice*. McGraw-Hill, 1994.

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

[14] 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.

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

[16] A. Kasagi, K. Nakano, and Y. Ito, "An optimal offline permutation algorithm on the hierarchical memory machine, with the GPU implementation," in *Proc. of International Conference on Parallel Processing*, Oct. 2013.

[17] K. Nakano, "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.

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

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

[20] K. Nakano, "An optimal parallel prefix-sums algorithm on the memory machine models for GPUs," in *Proc. of International Conference on Algorithms and Architectures for Parallel Processing (ICA3PP, LNCS 7439)*. Springer, Sept. 2012, pp. 99–113.

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

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

[23] 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.

[24] 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.

[25] 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.

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

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