# The Super Warp Architecture with Random Address Shift

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

Susumu Matsumae
*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 by a streaming multiprocessor on CUDA-enabled GPUs. The DMM has $w$ memory banks that constitute a shared memory, and each warp of $w$ threads access the shared memory 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. However, it is not easy to minimize the memory access congestion for some problems. The main contribution of this paper is to present novel and practical parallel computing models in which the congestion is small for any memory access requests. We first present the Super Discrete Memory Machine (SDMM), an extended version of the DMM, which supports a super warp with multiple warps. Memory access requests by multiple warps in a super warp are packed through pipeline registers to reduce the memory access congestion. We then go on to apply the random address shift technique to the SDMM. The resulting machine, the Random Super Discrete Memory Machine (RSDMM) can equalize memory access requests by a super warp. Quite surprisingly, for any memory access requests by a super warp on the RSDMM, the overhead of the memory access congestion is within a constant factor of perfectly scheduled memory access. Thus, unlike the DMM, developers of parallel algorithms do not have to consider the memory access congestion on the RSDMM. The congestion on the RSDMM is evaluated by theoretical analysis as well as by experiments.**

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

## I. Introduction

### A. Background

*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], [5], [6]. NVIDIA provides a parallel computing architecture called *CUDA* (Compute Unified Device Architecture) [7], 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 has 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* [7]. 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 with large capacity, say, 1.5-6 Gbytes, but its access latency is very large. 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 accesses and *coalescing* of the global memory accesses [8]. 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 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) [9], [10], 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. Although GPUs have the shared memory and the global memory accessed by multiple threads, parallel algorithms developed for the PRAM may not achieve good performance on GPUs. We should consider the memory access characteristics such as bank conflicts and coalescing when we develop efficient parallel algorithms for GPUs.

### B. Memory Machine Models for GPUs

*The Discrete Memory Machine (DMM)* [11], [12], *the*

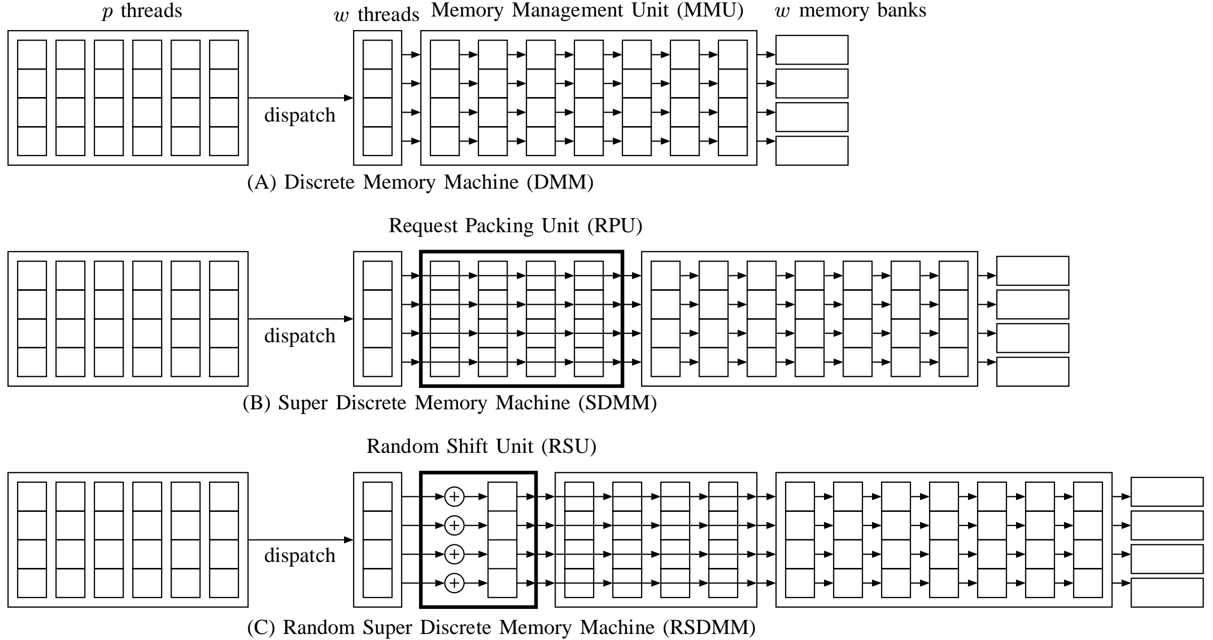


Figure 1. The DMM, the SDMM, and the RSDMM for $w = 4$

*Unified Memory Machine (UMM)* [13], [14], [15], and *the Hierarchical Memory Machine* [16], [17], [18] are theoretical parallel computing models of CUDA-enabled GPUs. The DMM reflects parallel computing by multiple cores in a single streaming multiprocessor with the shared memory. The UMM corresponds to parallel computing using the global memory of CUDA-enabled GPUs. The HMM captures the hierarchical architecture of CUDA-enabled GPUs using multiple streaming processors with multiple cores. In this paper, we focus on the DMM.

Figure 1 (A) illustrates the basic structure of the Discrete Memory Machine (DMM). The DMM has $p$ threads, each of which is a Random Access Machine (RAM) [19], which can execute fundamental operations in a time unit. It also has $w$ memory banks that constitute a single address space. We assume that address $i$ is mapped to bank $i \bmod w$. Threads are executed in SIMD [20] 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 *a warp*. For simplicity, we assume that $p$ is a multiple of $w$. On the DMM, $\frac{p}{w}$ warps are dispatched for memory access in turn, and $w$ threads in a dispatched warp send memory access requests. The $w$ memory access requests sent by a dispatched warp are sent to the $w$ memory banks through the Memory Management Unit (MMU). We do not discuss the architecture of the MMU, but we assume that it moves memory access requests to destination memory banks in a pipeline fashion. The memory access requests destined for the same memory bank are processed sequentially. Also, we assume that memory access takes latency $l$. Intuitively, it takes $l$ time units for a memory access request to move from a thread to the memory banks through the MMU. Note that the DMM has three parameters: width $w$ (i.e. the number of memory banks and the number of threads in a warp), latency $l$, and the number $p$ of threads.

It is very important for developing efficient algorithms on the DMM to reduce *the memory access congestion*, the maximum number of memory access requests destined for the same bank. However, it is not easy and sometimes impossible to minimize the memory access congestion for some problems. In our previous paper [11], we have developed a graph coloring technique to minimize the memory access congestion for off-line permutation. Later, we have implemented this offline permutation algorithm on GeForce GTX-680 GPU [12]. 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. This fact implies that, the DMM is a good theoretical model for GPU computing using a streaming multiprocessor, and it is very important to minimize the memory access congestion when we implement parallel algorithms on the GPU.

### C. Our Contribution

The main contribution of this paper is to present novel and practical parallel computing models, in which the memory access congestion can be reduced.

We first present the Super Discrete Memory Machine (SDMM), an extended version of the DMM, which supports a super warp with multiple warps. Memory access requests

by a super warp are packed through the Request Packing Unit (RPU) to reduce the memory access congestion. Figure 1 (B) illustrates the basic structure of the SDMM. Let $s$ be the number of warps in each super warp. Memory access requests sent by $sw$ threads in a super warp are packed such that they are sent to the memory banks continuously. Since memory access requests by multiple warps are packed on the SDMM, we can expect to average memory access requests for memory banks and to reduce the memory access congestion.

We then go on to apply the random address shift technique presented in [21]. Usually, a single address space is mapped to the $w$ memory banks such that each address $j \cdot w + k$ ($j \geq 0, 0 \leq k \leq w-1$) is arranged in the $j$-th memory cell of memory bank $k$. The idea of the random address shift is to arrange each address $j \cdot w + k$ to the $j$-th memory cell of memory bank $(k + r_j) \bmod w$, where $r_0, r_1, \ldots$ are independent random numbers in $[0, w-1]$. In other words, each address $j \cdot w + k$ is arranged to address $j \cdot w + (k + r_j) \bmod w$. We apply this technique to the SDMM, and obtain the Random Super Discrete Memory Machine (RSDMM). Figure 1 (C) illustrates the basic structure of the RSDMM. The Random Shift Unit (RSU) is used to route address $j \cdot w + k$ to address $j \cdot w + (k + r_j) \bmod w$. On the SDMM, all $sw$ memory requests by a super warp are destined for the same memory bank in the worst case, while $s$ memory requests are destined for the same bank if they are equally sent to $w$ banks. On the other hand, on the RSDMM, we can guarantee that expected $O(\frac{\log s \log w}{\log \log w})$ memory requests are destined for the same bank for all cases if $s \leq \log w$.

Quite surprisingly, if a super warp on the RSDMM has $s = \log w$ warps, expected $O(\log w)$ memory access requests are destined for the same bank. Since the perfectly scheduled memory access on the SDMM must have $\log w$ memory access requests to the same bank, the overhead of the memory access congestion of the RSDMM for any memory access is within a constant factor of that of the SDMM for perfectly scheduled memory access. In this paper, the congestion on the RSDMM is evaluated by theoretical analysis as well as by experiments. We can say that, unlike the DMM or the SDMM, developers do not have to consider the memory access congestion on the RSDMM for optimizing parallel algorithms. They can assume a uniformly accessible shared memory when they implement parallel algorithm in the RSDMM. Our results on the RSDMM suggest a new architecture of streaming multiprocessor for next generation GPUs.

To clarify the difference of the DMM, the SDMM, and the RSDMM in terms of computational power, we evaluate the time for transposing a matrix of size $\sqrt{n} \times \sqrt{n}$, where $\sqrt{n}$ is a multiple of $w$. We use two algorithms: the Naive Transpose Algorithm and the Diagonal Transpose Algorithm. The Naive Transpose Algorithm follows the definition of the transpose as it is. Every thread is assigned to the matrix in row major order, and copies elements along the definition of the transpose. Figure 2 illustrates how the matrix is transposed by the Naive Transpose Algorithm. The congestion of this algorithm is very large on the DMM. The memory write requests by a warp are always destined for the same bank, and the transposing takes a lot of time. Actually, on the DMM with width $w$ and latency $l$, the Naive Transpose Algorithm runs in $O(n + \frac{nl}{p})$ time units using $p$ threads. Since the running time has a factor $O(n)$, the Naive Transpose Algorithm on the DMM has no contribution of parallel computation for transposing a matrix.

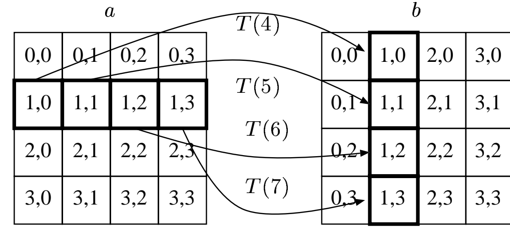


Figure 2. Illustrating the Naive Transpose Algorithm

The Diagonal Transpose Algorithm is designed to minimize the memory access congestion on the DMM. Every thread is assigned to the matrix in a diagonal order. Figure 3 illustrates how the matrix is transposed by the Diagonal Transpose Algorithm. The memory write requests by a warp are always destined for distinct banks. Thus, the transposing can be done much faster. On the DMM with width $w$ and latency $l$, the Diagonal Transpose Algorithm runs in $O(\frac{n}{w} + \frac{nl}{p})$ time units using $p$ threads. This computing time matches the sum of two lower bounds [15]: the bandwidth lower bound $\Omega(\frac{n}{w})$ and the latency lower bound $\Omega(\frac{nl}{p})$. Hence, the Diagonal Transpose Algorithm on the DMM is optimal.

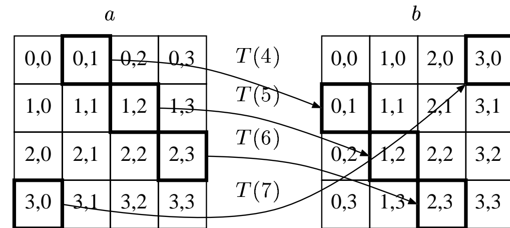


Figure 3. Illustrating the Diagonal Transpose Algorithm

It may be difficult and a heavy task for users to develop ingenious and optimal algorithms such as the Diagonal Transpose Algorithm. Hence, it is challenging to extend the DMM such that even the Naive Transpose Algorithm runs very efficiently. The SDMM can accelerate the Naive Transpose Algorithm if the size of matrix is small. We will

show that, on the SDMM with width $w$, latency $l$, and super warps of $\log w$ warps, the Naive Transpose Algorithm runs in $O(\frac{n}{w} + \min(n, \frac{n\sqrt{n}}{w \log w}) + \frac{nl}{p})$ time units using $p$ threads. The SDMM runs faster than the DMM when $w \log w > \sqrt{n}$, but it is not time optimal. On the other hand, the Naive Transpose Algorithm runs in expected $O(\frac{n}{w} + \frac{nl}{p})$ time units on the RSDMM. On the RSDMM, both the Naive Transpose Algorithm and the Diagonal Transpose Algorithm run in optimal running time. Table I summarizes the running time of two transpose algorithms on the three models.

This paper is organized as follows. In Section II, we first describe the Discrete Memory Machine (DMM). Section III shows the Naive Transpose Algorithm and the Diagonal Transpose Algorithm, and evaluates their performance on the DMM. In Section IV, we present the Super Discrete Memory Machine (SDMM) and evaluate the performance of two transpose algorithms on the SDMM. In Section V, we show the random address shift technique and the Random Super Discrete Memory Machine (RSDMM). We also prove that, for any memory access by a super warp with $s$ warps with $w$ threads each, the memory access congestion is expected $O(\frac{\log s \log w}{\log \log w})$. Section VI shows the experimental results of the memory access congestion on the RSDMM. Section VII 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 [11]. The reader should refer to [11] for the details of the DMM.

Recall that the DMM has three parameters: the width $w$, the latency $l$, and the number $p$ of threads. 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, each memory cell $m[i]$ ($i \geq 0$) is in bank $B[i \bmod w]$. 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 $T(0), T(1), \ldots, T(p-1)$ denote $p$ threads. We assume that $p$ threads are partitioned into $\frac{p}{w}$ groups of $w$ threads, each of which called *a warp*. 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 banks 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. When $W(i)$ is dispatched, $w$ threads in $W(i)$ send memory access requests, one request per thread, to the memory banks. Threads are executed in SIMD [20] fashion, and all threads 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, after a thread send a memory access request, it must wait $l$ time units to send a new one.

Figure 4 shows an example of memory access on the DMM with $w$ $(= 4)$ memory banks and memory access latency of $l$ $(= 7)$. We assume that each memory access request is completed when it reaches the last pipeline stage. Three warps $W(0), W(1)$, and $W(2)$ of 4 threads each access the 4 memory banks. In the DMM, memory access requests by $W(0)$ are separated into two pipeline stages, because $m[4]$ and $m[16]$ are in the same bank $B[0]$. Those by $W(1)$ occupy three pipeline stages, because three memory access requests destined for $B[3]$. Also, those by $W(0)$ occupy two pipeline stages, and thus it takes $(2+3+2)+l-1 = 13$ time units to complete the memory access by the three warps.

Let us define *the memory access congestion of a warp* with $w$ threads. Suppose that each of $w$ threads in a warp sends one access memory access request to a memory bank. The memory access congestion of a warp is the maximum number of requests destined for the same bank. More specifically, the congestion $c$ is $\max\{n_i \mid 0 \leq i \leq w-1\}$, where each $n_i$ ($0 \leq i \leq w-1$) is the number of memory requests destined for each $B[i]$. For example, the congestion of memory access by $W(0)$ in Figure 4 is 2, because two requests are destined for $m[4]$ and $m[16]$ in bank $B[0]$. Clearly, the congestion $c$ is 1 if all $w$ threads in a warp access distinct banks. On the other hand, if they access the same bank, the congestion is $w$. Hence, the congestion by a warp takes value between 1 and $w$.

We also 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 to the same address are removed. Thus, the DMM works as the Concurrent Read Concurrent Write (CRCW) mode with arbitrary resolution of simultaneous writing [9], [10], [22].

We can evaluate the performance of algorithms on the DMM by the number of rounds of memory access and the congestion. *A round of memory access* is an operation such that all of the $p$ threads perform a single memory access to the memory. Clearly, a round of memory access by $p$ threads is partitioned into memory access by $\frac{p}{w}$ warps. Suppose that the DMM performs $\mathcal{T}$ rounds of memory access and let $c_{i,j}$ ($0 \leq i \leq \frac{p}{w} - 1, 0 \leq j \leq \mathcal{T} - 1$) denote the congestion of warp $W(i)$ in the $j$-th round. Also, let

Table I
THE RUNNING TIME OF TRANSPOSE ALGORITHMS FOR A $\sqrt{n} \times \sqrt{n}$ MATRIX ON THE DMM, THE SDMM AND THE RSDMM

| | DMM | SDMM | RSDMM |
|---|---|---|---|
| Naive Transpose | $O(n + \frac{nl}{p})$ | $O(\frac{n}{w} + \min(n, \frac{n\sqrt{n}}{w \log w}) + \frac{nl}{p})$ | expected $O(\frac{n}{w} + \frac{nl}{p})$ |
| Diagonal Transpose | $O(\frac{n}{w} + \frac{nl}{p})$ | $O(\frac{n}{w} + \frac{nl}{p})$ | expected $O(\frac{n}{w} + \frac{nl}{p})$ |
| Lower bound | $\Omega(\frac{n}{w} + \frac{nl}{p})$ | | |

$w$: width, $l$: latency, $p$: # of threads, $\log w$:# of warps in a super warp

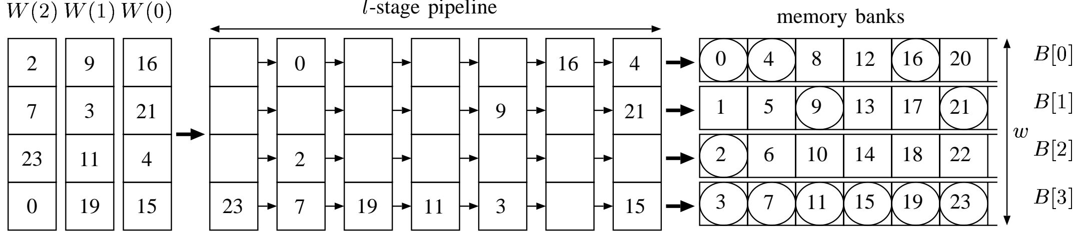


Figure 4. An example of memory access by the Discrete Memory Machine (DMM)

$\mathcal{C} = \sum_{j=0}^{\mathcal{T}-1} \sum_{i=0}^{\frac{p}{w}-1} c_{i,j}$ denote *the total congestion* of the algorithm. The $j$-th round of memory access can be done in

$$\sum_{i=0}^{\frac{p}{w}-1} c_{i,j} + l - 1$$

time units. Hence, the $\mathcal{T}$ rounds of memory access can be completed in

$$\begin{aligned}\sum_{j=0}^{\mathcal{T}-1} \left( \sum_{i=0}^{\frac{p}{w}-1} c_{i,j} + l - 1 \right) &= \sum_{j=0}^{\mathcal{T}-1} \sum_{i=0}^{\frac{p}{w}-1} c_{i,j} + (l-1)\mathcal{T} \\ &= \mathcal{C} + (l-1)\mathcal{T}\end{aligned}$$

time units. Thus, we have,

*Lemma 1:* An algorithm running in $\mathcal{T}$ rounds of memory access and $\mathcal{C}$ total congestion on the DMM with latency $l$ runs in $\mathcal{C} + (l-1)\mathcal{T}$ time units.

## III. THE NAIVE TRANSPOSE ALGORITHM AND THE DIAGONAL TRANSPOSE ALGORITHM

The main purpose of this section is to evaluate the computing time of two algorithms, the Naive Transpose Algorithm and the Diagonal Transpose Algorithm that transpose a matrix on the DMM. We also prove that the Diagonal Transpose Algorithm is time optimal.

We first evaluate the computing time of two memory access operations, *row-major access* and *column-major access* on the DMM. Suppose that we have a matrix $a$ of size $\sqrt{n} \times \sqrt{n}$. We assume that $\sqrt{n}$ is a multiple of $w$. Note that $(i, j)$-element of the matrix is arranged in offset $i \cdot \sqrt{n} + j$ and thus, it is arranged in bank $B[j \bmod w]$.

Suppose that $p$ threads of the DMM access all of the $n$ elements in $a$ such that each thread accesses $\frac{n}{p}$ elements. We can consider two memory access operations, *row-major access* and *column-major access* as follows:

**[Row-major access]**
for $i \leftarrow 0$ to $p-1$ do in parallel
  for $t \leftarrow 0$ to $\frac{n}{p} - 1$ do
    $j \leftarrow (t \cdot p + i)/\sqrt{n}$
    $k \leftarrow (t \cdot p + i) \bmod \sqrt{n}$
    thread $T(i)$ accesses $a[j][k]$

**[Column-major access]**
for $i \leftarrow 0$ to $p-1$ do in parallel
  for $t \leftarrow 0$ to $\frac{n}{p} - 1$ do
    $j \leftarrow (t \cdot p + i)/\sqrt{n}$
    $k \leftarrow (t \cdot p + i) \bmod \sqrt{n}$
    thread $T(i)$ accesses $a[k][j]$

The reader should refer to Figure 5 illustrating the row-major access and the column-major access for $\sqrt{n} = p = 4$.

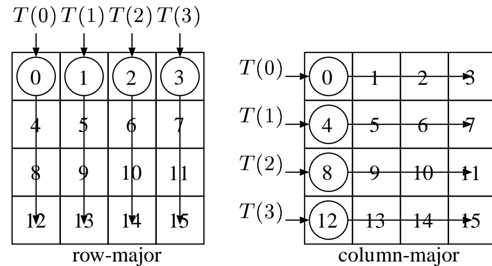


Figure 5. Row-major access and column-major access

Using Lemma 1, we can evaluate the running time of these memory access operations on the DMM. In the row-major access, $w$ threads in a warp access contiguous address. Thus,

they access different banks and the congestion is 1, and the total congestion is $\frac{n}{w}$. Also, it performs $\frac{n}{p}$ rounds memory access. Hence, from Lemma 1, the row-major access takes $\frac{n}{w} + \frac{n(l-1)}{p}$ time units. On the other hand, in the column-major access, $w$ threads access the same bank, and the congestion is $w$. Hence, the total congestion is $n$ and the column-major access takes $n + \frac{n(l-1)}{p}$ time units.

Suppose that we have two matrices $a$ and $b$ of size $\sqrt{n} \times \sqrt{n}$. Again, we assume that $\sqrt{n}$ is a multiple of $w$. The following algorithm stores the transpose of $a$ in $b$ using $p$ threads:

**[Naive Transpose Algorithm]**
for $i \leftarrow 0$ to $p-1$ do in parallel
  for $t \leftarrow 0$ to $\frac{n}{p} - 1$ do
    $j \leftarrow (t \cdot p + i)/\sqrt{n}$
    $k \leftarrow (t \cdot p + i) \bmod \sqrt{n}$
    thread $T(i)$ performs $b[k][j] \leftarrow a[j][k]$

Let us evaluate the running time. The reader should refer to Figure 2 illustrating the Naive Transpose Algorithm. Clearly, $p$ threads read all elements in $a$ in row-major order. However, they write in $b$ in column-major order. Thus, the Naive Transpose Algorithm runs in $O(n + \frac{nl}{p})$ time units.

Using the diagonal memory access technique [11], the transpose can be done with memory access congestion 1.

**[Diagonal Transpose Algorithm]**
for $i \leftarrow 0$ to $p-1$ do in parallel
  for $t \leftarrow 0$ to $\frac{n}{p} - 1$ do
    $j \leftarrow (t \cdot p + i)/\sqrt{n}$
    $k \leftarrow (t \cdot p + i) \bmod \sqrt{n}$
    thread $T(i)$ performs
      $b[k][(j+k) \bmod \sqrt{n}] \leftarrow a[(j+k) \bmod \sqrt{n}][k]$

Figure 3 illustrates the Diagonal Transpose Algorithm. It should be clear that this algorithm performs the transpose correctly. Let us evaluate the running time. Clearly, the memory access to $a$ and $b$ by a warp destined for distinct memory banks, and the congestion is 1. Thus, the transpose algorithm with diagonal access runs in $O(\frac{n}{w} + \frac{nl}{p})$ time units using $p$ threads. Thus, we have,

*Theorem 2:* The Naive Transpose Algorithm and the Diagonal Transpose Algorithm for a matrix of size $\sqrt{n} \times \sqrt{n}$ run in $O(n + \frac{nl}{p})$ time units and in $O(\frac{n}{w} + \frac{nl}{p})$ time units using $p$ threads on the DMM with width $w$ and latency $l$, respectively.

We prove that the Diagonal Transpose Algorithm is time optimal. Clearly, each of the $n$ elements in $a$ must be accessed at least once, and at most $w$ elements in $a$ can be read in a time unit. Thus it takes at least $\Omega(\frac{n}{w})$ time for transposing $a$. Also, each of the $p$ thread can send at most one memory read request in $l$ time units. Hence, $p$ threads can send $\frac{pt}{l}$ memory read requests in $t$ time units. Since $\frac{pt}{l} \geq n$ must be satisfied, it takes at least $t = \Omega(\frac{nl}{p})$ time units to read $n$ elements in $a$. Therefore, any algorithm takes $\Omega(\frac{n}{w} + \frac{nl}{p})$ time units to transpose a matrix of size $\sqrt{n} \times \sqrt{n}$ and the Diagonal Transpose Algorithm is time optimal.

## IV. The Super Discrete Memory Machine (SDMM)

The main purpose of this section is to define the Super Discrete Memory Machine (SDMM), the DMM supporting super warps.

*A super warp* is a set of warps. Let $s$ denote the number of warps in a super warp. Since each warp has $w$ threads, a super warp has $sw$ threads totally. Suppose that each of the $sw$ threads in a super warp sends a memory access request. In the SDMM, all memory requests destined for the same bank are packed such that they are sent to the memory bank continuously. Figure 6 illustrates an example of memory access by a super warp with three warps $W(0)$, $W(1)$, and $W(2)$ of 4 threads each. The reader should compare Figure 6 with Figure 4, in which three warps access the same addresses. As illustrated in the figure, memory access requests in pipeline registers are packed so that they occupy fewer pipeline stages. In this paper, we do not discuss the details of the hardware implementation of the SDMM, but it is not difficult to implement the RPU for the SDMM (Figure 1 (B)) by few additional circuits.

We define *the memory access congestion by a super warp* to be the maximum number of requests destined for the same bank. More specifically, the congestion is $c = \max\{n_i \mid 0 \leq i \leq w-1\}$ where each $n_i$ $(0 \leq i \leq w-1)$ is the number of memory access requests out of $sw$ requests destined for $B[i]$. For example, the memory access congestion by the super warp of three warps in Figure 6 is 6, because 6 memory access requests are destined for bank $B[3]$. If $sw$ memory requests are destined for the $w$ memory banks equally, each bank receives $s$ memory access requests. We also define *the memory access congestion ratio* to be the memory access congestion per warp, that is, $\frac{c}{s}$. For example, in Figure 6, the memory access congestion ratio is $\frac{6}{3} = 2$. Clearly, if all $sw$ threads in a super warp access distinct address, the memory access congestion is at least $s$ and the memory access congestion ratio is at least 1. Note that, if multiple threads access the same address, these values can be smaller. For example, all $sw$ threads in a super warp read from the same address, the memory access congestion and the memory access congestion ratio are 1 and $\frac{1}{s}$ respectively.

Let $S(0), S(1), \ldots, S(\frac{p}{sw} - 1)$ denote super warps such that each super warp $S(i)$ $(0 \leq i \leq \frac{p}{sw} - 1)$ consists of $s$ warps $W(i \cdot sw), W(i \cdot sw + 1), \ldots, W((i+1) \cdot sw - 1)$. For simplicity, we assume that $p$ is a multiple of $sw$. Suppose that the SDMM performs $\mathcal{T}$ rounds of memory access and let $c_{i,j}$ $(0 \leq i \leq \frac{p}{sw} - 1, 0 \leq j \leq \mathcal{T} - 1)$ denote the memory access congestion of super warp $S(i)$ in the $j$-th round. Also, let $\mathcal{C} = \sum_{j=0}^{\mathcal{T}-1} \sum_{i=0}^{\frac{p}{sw}-1} c_{i,j}$ denote *the total congestion* of the algorithm. The reader should have no difficulty to confirm that Lemma 1 also holds for the SDMM.

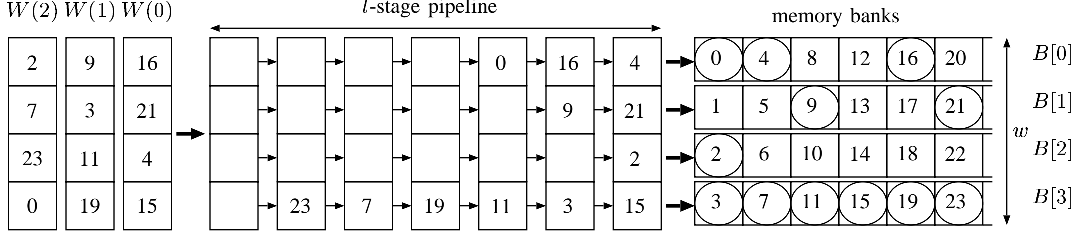


Figure 6. An example of memory access by the Super Discrete Memory Machine (SDMM)

Let us evaluate the time necessary to complete the Naive Transpose Algorithm on the SDMM using Lemma 1 for the SDMM. For this purpose, we evaluate the column-major access to a matrix $a$ of size $\sqrt{n} \times \sqrt{n}$ on the SDMM.

**Case 1:** $sw \leq \sqrt{n}$
Recall that each column of the matrix is in the same bank. All $sw$ memory requests by a super warp are destined for the same bank. Thus, the total congestion is $n$. Hence, the column-major access takes $n + \frac{n(l-1)}{p}$ time units.

**Case 2:** $sw > \sqrt{n}$
Let $q = \frac{sw}{\sqrt{n}}$. If this is the case, a super warp accesses $q$ columns of $a$. Thus, $sw$ threads in a super warp access $q$ memory banks such that $\sqrt{n}$ threads access the same bank. Thus, the memory access congestion of a round of memory access by each super warp is $\sqrt{n}$. Since the column-major access performs $\frac{n}{p}$ rounds of memory access, the total congestion is $\sqrt{n} \cdot \frac{p}{sw} \cdot \frac{n}{p} = \frac{n\sqrt{n}}{sw}$. Hence, the column-major access takes $\frac{n\sqrt{n}}{sw} + \frac{n(l-1)}{p}$ time units.

Combining Cases 1 and 2, we get that the column-major access is $O(\min(n, \frac{n\sqrt{n}}{sw}) + \frac{nl}{p})$ time units. Also, in the row-major access on the SDMM, the total congestion is $\frac{n}{w}$. Similarly, in the the Diagonal Transpose Algorithm on the SDMM, the total congestion is $\frac{n}{w}$. Thus, we have,

*Theorem 3:* The Naive Transpose Algorithm and the Diagonal Transpose Algorithm run in $O(\frac{n}{w} + \min(n, \frac{n\sqrt{n}}{sw}) + \frac{nl}{p})$ time units and $O(\frac{n}{w} + \frac{nl}{p})$ time units using $p$ threads on the SDMM with width $w$, latency $l$, and super warp having $s$ warps.

## V. The Random Super Discrete Memory Machine (RSDMM)

The main purpose of this section is to present a novel technique that we call *the random address shift*.

Recall that on the DMM and the SDMM, each $m[i]$ is arranged in bank $B[i \bmod w]$. The idea of the random address shift is to randomly shift the address mapping to average the memory access requests destined for memory banks. We can consider that $m$ is a 2-dimensional array of width $w$ such that $m[j][k]$ $(j \geq 0, 0 \leq k \leq w-1)$ corresponds to $m[j \cdot w + k]$ in the 1-dimensional context. Clearly, $m[j][k]$ is in bank $B[k]$ on the DMM and the SDMM.

Let $r_0, r_1, \ldots$ denote independent random integers uniformly selected from $[0, w-1]$. Intuitively, the random address shift technique rotates each $j$-th row of $m$ by $r_j$. More specifically, each $m[j][k]$ $(j \geq 0, 0 \leq k \leq w-1)$ is arranged in bank $B[(k + r_j) \bmod w]$. The reader should refer to Figure 7 illustrating how each address is mapped in memory banks.

Intuitively, the RSDMM is the SDMM with the Random Shift Unit (RSU) as illustrated in Figure 1 (C). The RSU is used to convert address $j \cdot w + k$ into address $j \cdot w + (k + r_j) \bmod w$. In other words, a memory access request destined for bank $k$ is routed to bank $(k + r_j) \bmod w$. Usually, parallel hashing to average the memory access needs complicated computation [23], [24]. However, the Random Shift Unit is so simple that it just reads the value of $r_j$ and performs one addition operation for each memory access request. Figure 7 illustrates how memory access requests stored in the $l$-stage pipeline. Since the super warp of three warps has 4 memory requests destined for $B[1]$, it takes $4 + l - 1 = 10$ time units to complete these memory access. The reader should compare Figure 7 with Figure 4 and Figure 6, in which three warps access the same addresses. As illustrated in the figure, memory access requests in pipeline registers of the RSDMM occupy fewer pipeline stages than the others.

We will prove that the expected value of the congestion ratio is at most $O(1)$ for any memory access by a super warp of $\log w$ warps on the RSDMM with width $w$. More generally, we show that the congestion ratio is $O(\frac{\log s \log w}{s \log \log w})$ for any memory access for all $w$ $(\geq 4)$ and $s$ $(2 \leq s \leq \log w)$ where $s$ is the number of warps in a super warp. 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 4 (Chernoff Bound [25]):* Let $X_0$, $X_1$, $\ldots$, $X_{n-1}$ be independent Poisson 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

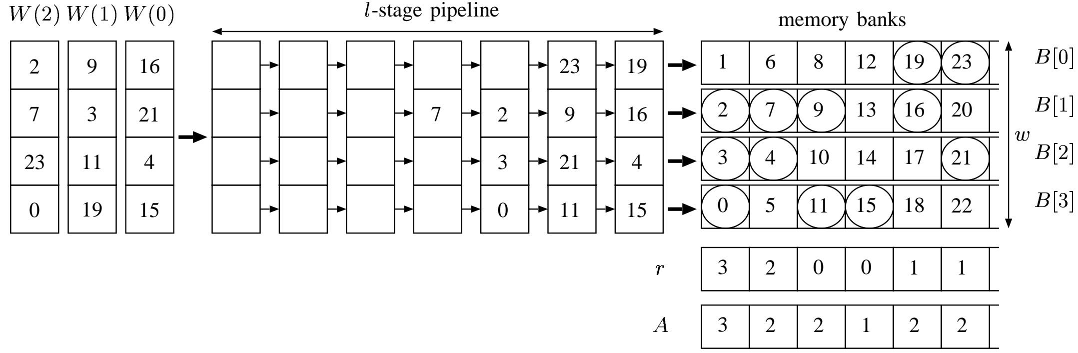


Figure 7. An example of memory access by the Random Super Discrete Memory Machine (RSDMM)

$\mu = E[X] = \sum_{i=0}^{n-1} p_i$. 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 for $s$ ($2 \leq s \leq \log w$). Note that the base of $\log$ is 2 and $e$ is the base of the natural logarithm. We assume that $sw$ threads in a super warp access to an array $m$ of size $n$. Let $j_0, j_1, \ldots, j_{sw-1}$ and $k_0, k_1, \ldots, k_{sw-1}$ be the indexes of $m$ such that each thread $T(i)$ ($0 \leq i \leq sw-1$) in a super warp 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 $m[j][*]$. Figure 7 also shows the values of $A$. Clearly, $\sum_{j=0}^{\frac{n}{w}-1} A(j) = sw$.

We fix a particular bank $B[v]$ ($0 \leq v \leq w-1$) and evaluate the number of memory access requests destined for $B[v]$ 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 random binary variables such that $X_j = 1$ iff $m[j][v]$ ($0 \leq j \leq \frac{n}{w} - 1$) is accessed by at least one of the $sw$ threads. Clearly, $X_j = 1$ with probability $\frac{A(j)}{w}$, because $A(j)$ elements in $m[j][*]$ 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 4 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 $B[v]$.

Let $f_w(s)$ be a function such that

$$f_w(s) \quad = \quad \frac{2e(\log s + 1)\log w}{\log\log w + 1}.$$

We have the following lemma:

*Lemma 5:* For random variable $X$ defined above, we have,

$$\Pr[X > f_w(s)] \quad < \quad w^{-2e}$$

for any $w$ and $s$ such that $w \geq 4$ and $2 \leq s \leq \log w$.

*Proof:* Throughout the proof, we assume that $w \geq 4$ and $2 \leq s \leq \log w$. Since $\mu = E[X] = \sum_{j=0}^{n/w-1} \frac{A(j)}{w} = s$, it should be clear that

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

from Theorem 4. Let $(1+\delta)s = f_w(s)$ and we have,

$$\begin{aligned}
&\log \Pr[X > f_w(s)] \\
&\quad < \quad (1+\delta)s \log \frac{e}{1+\delta} = f_w(s) \log \frac{es}{f_w(s)} \\
&\quad = \quad \frac{2e(\log s + 1)\log w}{\log\log w + 1} \log \frac{(\log\log w + 1)s}{2(\log s + 1)\log w}. \qquad (1)
\end{aligned}$$

Let $g_w(s)$ be a function of $s$ such that

$$g_w(s) \quad = \quad \frac{\log s + 1}{\log\log w + 1} \log \frac{(\log\log w + 1)s}{2(\log s + 1)\log w}. \qquad (2)$$

From (1) and (2) it is sufficient to prove that $g_w(s) \leq -1$, because $\log \Pr[X > f_w(s)] < 2e \log w \cdot g_w(s) \leq -2e \log w$ if this is the case. We first show $g_w(s) \leq -1$ for the boundary cases as follows:

$$\begin{aligned}
g_w(2) \quad &= \quad \frac{2}{\log\log w + 1} \log \frac{(\log\log w + 1)}{2\log w} \\
&= \quad \frac{2(\log(\log\log w + 1) - (\log\log w + 1))}{\log\log w + 1} \\
&\leq \quad \frac{2(-\frac{1}{2}(\log\log w + 1))}{\log\log w + 1} = -1 \\
&\qquad \text{(from } \log x - x \leq -\tfrac{x}{2} \text{ for all } x \geq 2\text{)} \\
g_w(\log w) \quad &= \quad \frac{\log\log w + 1}{\log\log w + 1} \log \frac{(\log\log w + 1)\log w}{2(\log\log w + 1)\log w} \\
&= \quad \log \frac{1}{2} = -1.
\end{aligned}$$

We next show $g_w(s) \leq -1$ for all $s$. Let $u = \log s + 1$. Also, let $h_w(u)$ be the function such that

$$h_w(u) \quad = \quad g_w(s) = \frac{u}{\log\log w + 1}(u - \log u + C),$$

where

$$C = \log(\log\log w + 1) - \log\log w - 2.$$

From the boundary cases above, it is sufficient to show that $h_w(u)$ $(2 \le u \le \log\log w + 1)$ is a convex function for the purpose of proving $g_w(s) \le -1$ for all $s$. We differentiate $h_w$ twice as follows:

$$\begin{aligned} h'_w(u) &= \frac{u - \log u + C + u(1 - \frac{\log e}{u})}{\log\log w + 1} \\ &= \frac{2u - \log u + C - \log e}{\log\log w + 1} \\ h''_w(u) &= \frac{2 - \frac{\log e}{u}}{\log\log w + 1} \end{aligned}$$

Hence, $h''_w(u) > 0$ for all $u \ge 2$. It follows that $h_w$ is convex and thus, $h_w(u) \le -1$ for all $u$ $(2 \le u \le \log\log w + 1)$. This completes the proof. ■

Let $Y$ denote a random variable denoting the maximum number of memory access requests over all banks $B(v)$ $(0 \le v \le w-1)$. From Lemma 5, we have

$$\Pr[Y > f_w(s)] \le \Pr[X > f_w(s)] \cdot w < w^{-2e+1}.$$

Thus, we have,

$$\begin{aligned} \Pr[0 \le Y \le f_w(s)] &< 1, \text{ and} \\ \Pr[f_w(s) < Y \le sw] &< w^{-2e+1}. \end{aligned}$$

Hence, the expected value of $Y$ is at most:

$$\begin{aligned} E[Y] &\le \Pr[0 \le Y \le f_w(s)] \cdot f_w(s) \\ &\quad + \Pr[f_w(s) < Y \le sw] \cdot sw \\ &< 1 \cdot f_w(s) + w^{-2e+1} \cdot sw = O(f_w(s)), \end{aligned}$$

from $s \le \log w$. Since the congestion ratio is $\frac{E[Y]}{s}$, we have,

*Theorem 6:* For any memory access by a super warp of $s$ warps on the RSDMM with width $w$, the congestion ratio is expected $O(\frac{f_w(s)}{s})$ for any $w \ge 4$ and $s$ $(2 \le s \le \log w)$.

Clearly, $\frac{f_w(s)}{s} = \frac{\log s \log w}{s \log\log w} = 1$ if $s = \log w$, and thus, we have,

*Corollary 7:* For any memory access by a super warp of $\log w$ warps on the RSDMM with width $w$, the congestion ratio is expected $O(1)$ for any $w \ge 4$.

Let us evaluate the running time of the Naive Transpose Algorithm and the Diagonal Transpose Algorithm on the RSDMM with super warps having $\log w$ warps. Since the congestion ratio is $O(1)$, the memory access by a super warp of $w \log w$ threads takes $O(\log w)$ time units to send memory access requests. Thus, the transpose algorithm runs in $\frac{n}{w \log w} \cdot O(\log w) + \frac{n(l-1)}{p} = O(\frac{n}{w} + \frac{nl}{p})$ time units. Thus, we have,

*Theorem 8:* Both The Naive Transpose Algorithm and the Diagonal Transpose Algorithm run in expected $O(\frac{n}{w} + \frac{nl}{p})$ time units using $p$ threads on the RSDMM with width $w$, latency $l$, and super warp having $\log w$ warps.

## VI. Experimental results

We will show that the actual value of the congestion ratio $\frac{E[Y]}{s}$ is enough small by simulation. Table II shows these values obtained by 1,000,000 rounds of simulation, where a round is a round of memory access by a super warp with $sw$ threads. In the simulation, every thread selects an address of the memory independently at random. Since multiple threads in a warp may select the same address, the congestion ratio can be less than 1. Since the number $w$ of memory banks of current CUDA-enabled GPUs are 16 or 32 [7], we evaluate the congestion ratio $\frac{E[Y]}{s}$ for $w = 16, 32, 64, 128$ and 256 for considering future extension of GPUs. Also, we perform the experiment for various number $s$ of warps in a super warp to see the relation between $s$ and the congestion $\frac{E[Y]}{s}$. We use the size $n$ of array $m$ is 1024 ($= 2^{10}$) and 1048576 ($= 2^{20}$).

In the table, the congestion is underlined when $s = \log w$. For example, $\frac{E[Y]}{s} = 1.819$ when $w = 32$, $s = 5$, and $n = 1024$. When $n = 1024$, the congestion may be smaller than 1 for large $w$ and $s$, because many threads access the same address. We can see that the congestion $\frac{E[Y]}{s}$ is less than 2 when $s = \log w$ and $n = 1024$. Also, the congestion is almost 2 when $s = \log w$ and $n = 1048576$. Recall that, for any memory access by a super warp, the congestion ratio on the RSDMM is $O(\frac{f_w(s)}{s})$ from Theorem 6. The table also shows the values of $\frac{f_w(s)}{es} = \frac{2(\log s+1)\log w}{s(\log\log w+1)}$. We can see that the values of $\frac{f_w(s)}{es}$ approximates the congestion ratio for $n = 1048576$.

## VII. Conclusion

We have presented the Random Super Discrete Memory Machine (RSDMM), which supports super warps and random address shift technique to reduce the memory access congestion to memory banks. On the RSDMM, it is guaranteed that the overhead of the memory access congestion by any memory access is at most $O(1)$ factor over the perfectly scheduled memory access. Hence, users do not have to consider to minimize the memory access congestion when they develop algorithm on the RSDMM. Thus, the RSDMM is a promising candidate of a new streaming multiprocessor architecture for next generation GPUs.

## 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.

Table II
THE CONGESTION RATIO $E[Y]/s$ ON THE RSDMM

| $n$ | 1024 $(= 2^{10})$ | | | | | 1048576 $(= 2^{20})$ | | | | | $\frac{f_w(s)}{es} = \frac{2(\log s+1)\log w}{(\log\log w+1)}$ | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $w$ | 16 | 32 | 64 | 128 | 256 | 16 | 32 | 64 | 128 | 256 | 16 | 32 | 64 | 128 | 256 |
| $s$ = 1 | 3.038 | 3.433 | 3.714 | 3.808 | 3.458 | 3.080 | 3.533 | 3.959 | 4.379 | 4.766 | 2.667 | 3.010 | 3.347 | 3.677 | 4.000 |
| 2 | 2.358 | 2.574 | 2.678 | 2.572 | 1.999 | 2.416 | 2.708 | 2.982 | 3.246 | 3.494 | 2.667 | 3.010 | 3.347 | 3.677 | 4.000 |
| 3 | 2.064 | 2.201 | 2.217 | 2.016 | 1.333 | 2.134 | 2.363 | 2.576 | 2.778 | 2.971 | 2.298 | 2.594 | 2.884 | 3.168 | 3.447 |
| 4 | 1.888 | 1.975 | 1.936 | 1.674 | 1.000 | 1.970 | 2.163 | 2.342 | 2.511 | 2.671 | 2.000 | 2.258 | 2.510 | 2.758 | 3.000 |
| 5 | 1.766 | 1.819 | 1.738 | 1.438 | 0.800 | 1.861 | 2.029 | 2.186 | 2.333 | 2.472 | 1.772 | 2.000 | 2.224 | 2.443 | 2.658 |
| 6 | 1.675 | 1.700 | 1.587 | 1.256 | 0.667 | 1.781 | 1.932 | 2.072 | 2.205 | 2.329 | 1.593 | 1.799 | 2.000 | 2.197 | 2.390 |
| 7 | 1.603 | 1.604 | 1.466 | 1.118 | 0.571 | 1.719 | 1.857 | 1.986 | 2.106 | 2.218 | 1.450 | 1.637 | 1.821 | 2.000 | 2.176 |
| 8 | 1.542 | 1.526 | 1.365 | 0.996 | 0.500 | 1.670 | 1.798 | 1.917 | 2.027 | 2.131 | 1.333 | 1.505 | 1.674 | 1.839 | 2.000 |
| 9 | 1.492 | 1.458 | 1.279 | 0.889 | 0.444 | 1.629 | 1.749 | 1.859 | 1.963 | 2.059 | 1.236 | 1.395 | 1.551 | 1.704 | 1.853 |
| 10 | 1.448 | 1.399 | 1.204 | 0.800 | 0.400 | 1.595 | 1.708 | 1.812 | 1.909 | 1.999 | 1.153 | 1.301 | 1.447 | 1.589 | 1.729 |


[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.

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

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

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

[10] J. JáJá, *An Introduction to Parallel Algorithms*. Addison-Wesley, 1992.

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

[12] 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.

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

[14] ——, "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.

[15] ——, "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.

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

[17] 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.

[18] 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.

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

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

[21] 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.

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

[23] 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.

[24] 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.

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