Parallel Algorithms for the Summed Area Table on the Asynchronous Hierarchical Memory Machine, with GPU implementations

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

***Abstract*—The Hierarchical Memory Machine (HMM) is a theoretical parallel computing model that captures the essence of computing on CUDA-enabled GPUs. The summed area table (SAT) of a matrix is a data structure frequently used in the area of computer vision which can be obtained by computing the column-wise prefix-sums and then the row-wise prefix-sums. The main contribution of this paper is to introduce the asynchronous Hierarchical Memory Machine (asynchronous HMM), which supports asynchronous execution of CUDA blocks, and show a global-memory-access-optimal parallel algorithm for computing the SAT on the asynchronous HMM. A straightforward algorithm (2R2W SAT algorithm) on the asynchronous HMM, which computes the prefix-sums in every column using one thread each and then computes the prefix-sums in every row, performs 2 read operations and 2 write operations per element of a matrix. The previously published best algorithm (2R1W SAT algorithm) performs 2 read operations and 1 write operation per element. We present a more efficient algorithm (1R1W SAT algorithm) which performs 1 read operation and 1 write operation per element. Clearly, since every element in a matrix must be read at least once, and all resulting values must be written, our 1R1W SAT algorithm is optimal in terms of the global memory access. We also show a combined algorithm ($(1+r)$R1W SAT algorithm) of 2R1W and 1R1W SAT algorithms that may have better performance. We have implemented several algorithms including 2R2W, 2R1W, 1R1W, $(1+r)$R1W SAT algorithms on GeForce GTX 780 Ti. The experimental results show that our $(1+r)$R1W SAT algorithm runs faster than any other SAT algorithms for large input matrices. Also, it runs more than 100 times faster than the best SAT algorithm using a single CPU.**

***Keywords*-memory machine models, prefix-sums, summed area table, image processing, GPU, CUDA**

I. Introduction

The GPU (Graphics Processing Unit), is a specialized circuit designed to accelerate computation for building and manipulating images [1], [2], [3]. Latest GPUs are designed for general purpose computing and can perform computation in applications traditionally handled by the CPU. Hence, GPUs have recently attracted the attention of many application developers [1]. NVIDIA provides a parallel computing architecture called *CUDA* (Compute Unified Device Architecture) [4], the computing engine for NVIDIA GPUs. CUDA gives developers access to the virtual instruction set and memory of the parallel computational elements in NVIDIA GPUs.

CUDA-enabled GPUs have streaming multiprocessors (SMs) each of which executes multiple threads in parallel. CUDA can use two types of memories in the NVIDIA GPUs: *the shared memory* and *the global memory* [4]. 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 shared memory access and *coalescing* of global memory access [5], [6], [7]. The address space of shared memory is mapped into several physical memory banks. If two or more threads access the same memory banks at the same time, the access requests are processed in turn. Hence, to maximize the memory access performance, CUDA threads should access distinct memory banks to avoid the bank conflicts of the memory accesses. To maximize the bandwidth of the global memory, CUDA threads should perform coalesced access.

In our previous paper [8], we have introduced two models, *the Discrete Memory Machine (DMM)* and *the Unified Memory Machine (UMM)*, which reflect the essential features of the shared memory and the global memory of NVIDIA GPUs. The outline of the architectures of the DMM and the UMM is illustrated in Figure 1. In both architectures, *a sea of threads (Ts)* are connected to *the memory banks (MBs)* through *the memory management unit (MMU)*. Each thread is a Random Access Machine (RAM) [9], which can execute fundamental operations in a time unit. MBs constitute a single address space of the memory. A single address space of the memory is mapped to the MBs in an interleaved way such that the word of data of address i is stored in the $(i \bmod w)$-th bank, where w is the number of MBs. The main difference of the two architectures is the connection of the address line between the MMU and the MBs, which can transfer an address value. In the DMM, the address lines connect the MBs and the MMU separately, while a single address line from the MMU is connected to the MBs in the UMM. Hence, in the UMM, the same address value is broadcast to every MB, and the same address of the MBs can be accessed in each time unit. On the other hand, different addresses of the MBs can be accessed in the DMM.

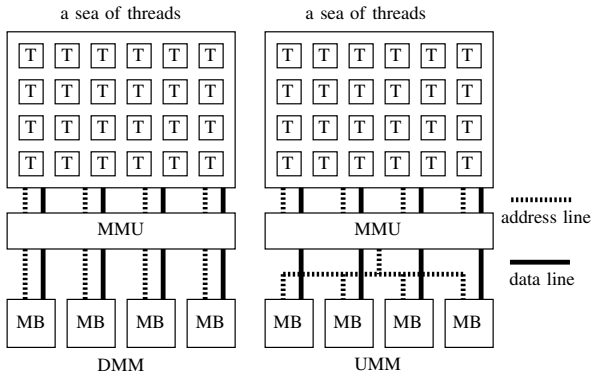

Figure 1. The architectures of the DMM and the UMM with $w = 4$

Quite recently, we have introduced the Hierarchical Memory Machine (HMM) [10], which is a hybrid of the DMM and the UMM. The HMM is a more practical parallel computing model that reflects the hierarchical architecture of CUDA-enabled GPUs. Figure 2 illustrates the architecture of the HMM. The HMM consists of d DMMs and a single UMM. Each DMM has w memory banks and the UMM has w memory banks. We call the memory banks of each DMM *the shared memory* and those of the UMM *the global memory* after CUDA-enabled GPUs. Each DMM can work independently and can perform the computation using its shared memory. Also, all threads of DMMs work as a single UMM and can access to the global memory. While the memory access latency of the shared memory of GPUs is very low, that of the global memory is several hundred clock cycles [4]. Hence, we assume that the latency of the shared memory is 1, and we use parameter l to denote the latency of the global memory in the HMM.

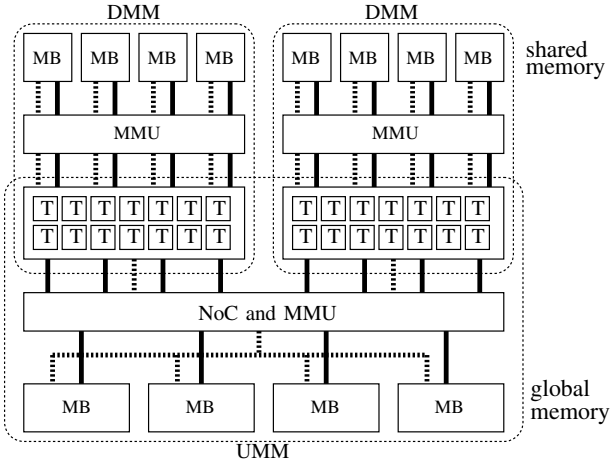

Figure 2. The architecture of the HMM with $d = 2$ DMMs and width $w = 4$

Suppose that a matrix a of size $\sqrt{n} \times \sqrt{n}$ is given. The summed area table (SAT) [11] is a matrix b of the same size such that

$$b[i][j] = \sum_{0 \le i' \le i, 0 \le j' \le j} a[i'][j'].$$

It should have no difficulty to confirm that the SAT can be obtained by computing the column-wise prefix-sums and the row-wise prefix-sums as illustrated in Figure 3. Once we have the summed area table, the sum of any rectangular area of a can be computed by evaluating

$$\sum_{u < i \le d, l < j \le r} a[i][j] = b[d][r] - b[u][r] - b[d][l] + b[u][l].$$

Since the sum of any rectangular area can be computed in $O(1)$ time the summed area table has a lot of applications in the area of image processing and computer vision [12]. In our previous paper [13], we have presented a parallel algorithm that computes the SAT in $O(\frac{n}{w} + \frac{nl}{p} + l \log n)$ time units using p threads on the UMM with width w and latency l. This algorithm is optimal in the sense that any SAT algorithm takes at least $\Omega(\frac{n}{w} + \frac{nl}{p} + l \log n)$ time units. However, this algorithm repeats pairwise addition and has a large constant factor in the computing time and it is not practically efficient.

The main contribution of this paper is to introduce the asynchronous Hierarchical Memory Machine (asynchronous HMM), which supports asynchronous execution of CUDA blocks, and show a global-memory-access-optimal parallel algorithm for computing the summed area table stored in the global memory of the asynchronous HMM. A straightforward algorithm (2R2W SAT algorithm) on the asynchronous HMM, which computes the column-wise prefix-sums and then the row-wise prefix-sums, performs 2 read operations and 2 write operations per element of a matrix. The best known algorithm (2R1W SAT algorithm) so far performs 2 read operations and 1 write operation per element [14]. We present a more efficient algorithm (1R1W SAT algorithm) on the asynchronous HMM, which performs only 1 read operation and 1 write operation per element. Clearly, since every element in a matrix must be read at least once and all resulting values must be written, our 1R1W SAT algorithm is optimal in terms of the global memory access. We also show a combined algorithm ($(1 + r)$R1W SAT algorithm) of 2R1W and 1R1W SAT algorithms that can run faster than any other algorithms for large matrices. Table I shows the total number of memory access operations to the global memory and the shared memory, the number of barrier synchronization steps, and the global memory access cost. The global memory access cost, which is computed from the number of global memory access operations and the number of barrier synchronization steps, approximates the computing time on the HMM. For simplicity, in the table, we omit small terms to focus on dominant terms. For example, 2R2W SAT algorithm performs $n - \sqrt{n}$ coalesced write operations, but we simply write n in the corresponding entry.

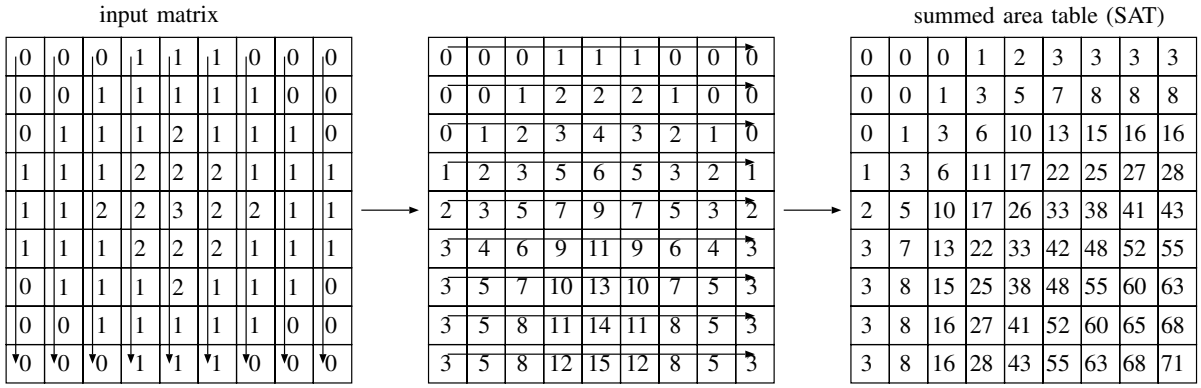

Figure 3. The summed area table (SAT) of a 9×9 matrix and 2R2W SAT algorithm

Table I
THE PERFORMANCE OF SAT ALGORITHMS ON THE HMM

SAT algorithms	global memory access		shared memory access	barrier synchronization steps	global memory access cost
	Coalesced Read/Write	Stride Read/Write	Read/Write		
2R2W	n / n	n / n	-	1	$2n + 2\frac{n}{w} + 2l$
4R4W	$4n$ / $4n$	-	n / n	3	$8\frac{n}{w} + 4l$
4R1W	-	$4n$ / n	-	$2\sqrt{n}$	$5n + 2\sqrt{n}l$
2R1W	$2n$ / n	-	$4n$ / $4n$	$2k + 2$	$3\frac{n}{w} + (2k + 3)l$
1R1W	n / n	-	$2n$ / $2n$	$2\frac{\sqrt{n}}{w}$	$2\frac{n}{w} + 2\frac{\sqrt{n}}{w}l$
1.25R1W	$1.25n$ / n	-	$2.5n$ / $2.5n$	$\frac{\sqrt{n}}{w} + 4k + 4$	$1.25\frac{n}{2} + (\frac{\sqrt{n}}{w} + 4k + 5)l$
$(1 + r)$R1W	$(1 + r)n$ / n	-	$(2 + \sqrt{r})n$ / $(2 + \sqrt{r})n$	$2\frac{(1-\sqrt{r})\sqrt{n}}{w} + 4k + 4$	$(1 + r)\frac{n}{w} + (2\frac{(1-\sqrt{r})\sqrt{n}}{w} + 4k + 5)l$

k is the depth of recursion, which takes value no more than 1 from the practical point of view.
r can take any value in the range $[0, 1]$

We have also implemented several algorithms including 2R2W, 2R1W, 1R1W, $(1 + r)$R1W SAT algorithms on GeForce GTX 780 Ti. The experimental results show that our $(1 + r)$R1W SAT algorithm runs faster than any other SAT algorithms for large input matrices. Further, it runs more than 100 times faster than the best SAT algorithm using a single CPU.

II. THE DMM, THE UMM, THE HMM AND THE ASYNCHRONOUS HMM

We first define *the Discrete Memory Machine (DMM)* of width w and latency l. Let $B[j] = \{j, j + w, j + 2w, j + 3w, \ldots\}$ $(0 \leq j \leq w - 1)$ be a set of address of *the j-th memory bank* of the memory. In other words, address i is in the $(i \bmod w)$-th memory bank. We assume that addresses in different banks can be accessed in a time unit, but no two addresses 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. Thus, it takes $k + l - 1$ time units to complete memory access requests to k addresses in a particular bank.

We assume that p threads are partitioned into $\frac{p}{w}$ groups of w threads called *warps*. More specifically, p threads $T(0)$, $T(1)$, $\ldots$, $T(p - 1)$ are partitioned into $\frac{p}{w}$ warps $W(0), W(1), \ldots, W(\frac{p}{w}-1)$ such that $W(i) = \{T(i \cdot w), T(i \cdot w + 1), \ldots, T((i + 1) \cdot w - 1)\}$ $(0 \leq i \leq \frac{p}{w} - 1)$. Warps are dispatched for memory access in turn, and w threads in a warp try to access the memory at the same time. In other words, $W(0), W(1), \ldots, W(\frac{p}{w} - 1)$ are dispatched in a round-robin manner if at least one thread in a warp requests memory access. If no thread in a warp needs memory access, such warp is not dispatched for memory access. When $W(i)$ is dispatched, w threads in $W(i)$ send memory access requests, at most one request per thread, to the memory. We also assume that a thread cannot send a new memory access request until the previous memory access request is completed. Hence, if a thread sends a memory access request, it must wait at least l time units to send a new memory access request.

We next define *the Unified Memory Machine (UMM)* of width w and latency l as follows. Let $A[j] = \{j \cdot w, j \cdot w + 1, \ldots, (j + 1) \cdot w - 1\}$ denote a set of addresses in *the j-th address group*. We assume that addresses in the same address group are processed at the same time. However, if they are in the different groups, one time unit is necessary for each of the groups. Also, similarly to the DMM, p threads are partitioned into warps and each warp accesses the memory

in turn.

Figure 4 shows examples of memory access on the DMM and the UMM. We assume that each memory access request is completed when it reaches the last pipeline stage. Two warps $W(0)$ and $W(1)$ access to $\langle 7, 5, 15, 0\rangle$ and $\langle 10, 11, 12, 9\rangle$, respectively. In the DMM, memory access requests by $W(0)$ are separated into two pipeline stages, because addresses 7 and 15 are in the same bank $B(3)$. Those by $W(1)$ occupies 1 stage, because all requests are in distinct banks. Thus, the memory requests occupy three stages, it takes $3 + 5 - 1 = 7$ time units to complete the memory access. In the UMM, memory access requests by $W(0)$ are destined for three address groups. Hence the memory requests occupy three stages. Similarly those by $W(1)$ occupy two stages. Hence, it takes $5+5-1 = 9$ time units to complete the memory access.

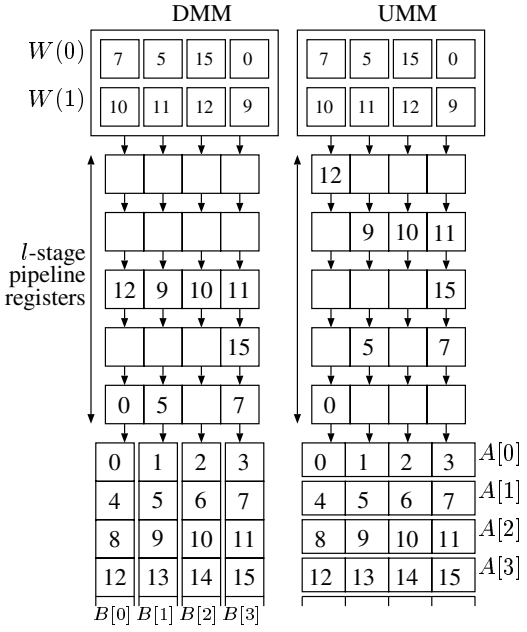

Figure 4. Examples of memory access on the DMM and the UMM

Next, we define *the Hierarchical Memory Machine (HMM)* [10]. The HMM consists of d DMMs and a single UMM as illustrated in Figure 2. Each DMM has w memory banks and the UMM also has w memory banks. We call the memory banks of each DMM *the shared memory* and those of the UMM *the global memory*. Each DMM works independently. Threads are partitioned into warps of w threads, and each warp are dispatched for the memory access for the shared memory in turn. Further, each warp of w threads in all DMMs can send memory access requests to the global memory. Figure 2 illustrates the architecture of the HMM with $d = 2$ DMMs. Each DMM and the UMM has $w = 4$ memory banks. The shared memory of each DMM and the global memory of the UMM correspond to "the shared memory" of each streaming multiprocessor and "the global memory" of GPUs. We also assume that the shared memory in each DMM of the HMM can store up to $O(w^2)$ numbers. The capacity of the shared memory of latest CUDA-enabled GPUs is up to 48KBytes and the number w of the banks is 32 [4]. Since an array of 32^2 double (64-bit) numbers occupy 8KBytes, each shared memory can store at most 6 such matrices. Thus, it is reasonable to assume that DMM can store $O(w^2)$ numbers in the shared memory.

For more realistic model for GPUs, we introduce *the asynchronous Hierarchical Memory Machine* (asynchronous HMM). In the asynchronous HMM, DMMs work asynchronously in the sense that some DMMs may work slightly slower or faster than the others. Instead, all threads in all DMMs can execute a barrier synchronization instruction. If a thread in a DMM executes the barrier synchronization instruction, it must wait until all the other threads in all DMMs execute it. Also, after all threads execute the barrier synchronization instruction, all DMMs are *reset*, that is, the shared memory of all DMMs are initialized and all data stored in it are lost. The reader may think that this reset assumption of all DMMs is not reasonable. However, this assumption is mandatory for program scalability of DMMs in the HMM. More specifically, suppose that a programmer write a program of the HMM with d DMMs. It may be possible to execute this program in the HMM with d' DMMs such that $d' < d$. If this is the case, the program of d' DMMs are executed until a barrier synchronization instruction is executed. After that, the d' DMMs are reset and the program of next d' DMMs are executed. The same procedure is repeated until all threads execute the barrier synchronization instruction. Hence, it makes sense to assume that all DMMs are reset after each barrier synchronization step. The previous DMMs is responsible for copying the data stored in the shared memory to the global memory before barrier synchronization if they are used after the synchronization. Actually, we need to terminate a CUDA kernel call for a GPU when barrier synchronization of all threads is necessary [4]. When a CUDA kernel call is terminated for barrier synchronization, the data stored in the shared memory by a CUDA block are lost. This is because CUDA blocks are executed in streaming multiprocessors with small shared memory one by one in turn.

III. The global memory access cost on the HMM and the diagonal arrangement on the DMM

Let C, S, and B be the total number of coalesced global memory access operations, the total number of stride global memory access operations, and the number of barrier synchronization steps performed on the HMM. *The global memory access cost* is defined to be $\frac{C}{w} + S + (B+1)(l-1)$. We will show that the global memory access cost approximates the computing time on the HMM if the computation performed in each DMM is negligible.

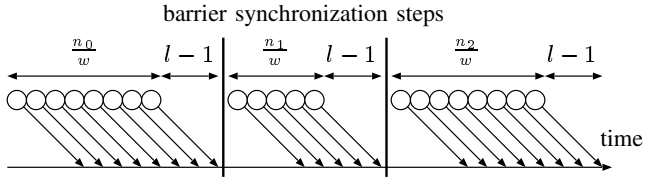

Figure 5. Timing chart of coalesced memory access to the global memory with two barrier synchronization steps

Suppose that an algorithm performs n coalesced memory access operations and two barrier synchronization steps. Clearly, by two barrier synchronization steps, the memory access is partitioned into three stages as illustrated in Figure 5. Let n_0, n_1, and n_2 such that $n = n_0 + n_1 + n_2$ be the numbers of memory access operations performed in the three stages. Since w coalesced memory access operations by a warp of w threads occupy one pipeline stage, the three stages takes $\frac{n_0}{w} + l - 1$, $\frac{n_1}{w} + l - 1$, and $\frac{n_2}{w} + l - 1$ time units respectively. Hence, the algorithm runs $\frac{n}{w} + 3(l-1)$ time units. Also, if n memory access operations are stride, each memory access operation occupy one pipeline stage, the algorithm runs in $n + 3(l-1)$ time units. In general, if an algorithm performs C coalesced memory access operations, S stride memory access operations, and B barrier synchronization steps, it runs $\frac{C}{w} + S + (B+1)(l-1)$, which is equal to the global memory access cost.

Suppose that we have a matrix of size $w \times w$ in the shared memory of a DMM in the HMM. Since a column of the matrix is in the same bank, column-wise access by w threads has bank conflicts, while row-wise access is conflict-free. In our previous paper [15], we have presented *a diagonal arrangement* of a matrix such that each (i, j) element is arranged in $a[i][(i + j) \bmod w]$. Figure 6 illustrates the diagonal arrangement of a 4×4 matrix. We can confirm that both a row-wise access to $(1,0), (1,1), (1,2), (1,3)$ and a column-wise access to $(0,1), (1,1), (2,1), (3,1)$ are conflict-free. Thus, we have,

Lemma 1: In the diagonal arrangement of a $w \times w$ matrix, both a row-wise access and a column-wise access are conflict-free.

The diagonal arrangement is used to compute the SAT of a $w \times w$ matrix in a shared memory and transpose of a matrix in the global memory of the HMM.

IV. 2R2W AND 4R4W SAT ALGORITHMS

Let s_i denote a local register of thread $T(i)$ $(0 \leq i \leq n-1)$. As illustrated in Figure 3, the summed area table (SAT) of a $\sqrt{n} \times \sqrt{n}$ matrix a can be computed by the column-wise prefix-sums and the row-wise prefix-sums:

[2R2W SAT algorithm]

for $i \leftarrow 0$ to $\sqrt{n}$ do in parallel // column-wise prefix-sums
  $T(i)$ performs $s_i \leftarrow a[0][i]$
  for $j \leftarrow 1$ to $\sqrt{n} - 1$ do
    $T(i)$ performs $s_i \leftarrow s_i + a[j][i]$
    $T(i)$ performs $a[j][i] \leftarrow s_i$
barrier_synchronization
for $i \leftarrow 0$ to $\sqrt{n}$ do in parallel // row-wise prefix-sums
  $T(i)$ performs $s_i \leftarrow a[i][0]$
  for $j \leftarrow 1$ to $\sqrt{n} - 1$ do
    $T(i)$ performs $s_i \leftarrow s_i + a[i][j]$
    $T(i)$ performs $a[i][j] \leftarrow s_i$

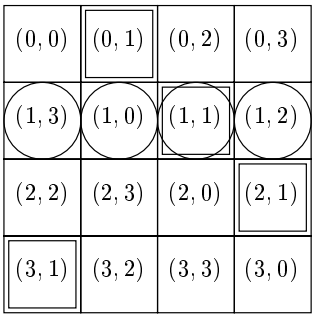

Figure 6. Diagonal arrangement of a 4×4 matrix

In the computation of the column-wise prefix-sums, $a[0][0], a[0][1], \ldots, a[0][\sqrt{n}-1]$ are read. After that, for each j $(1 \leq j \leq \sqrt{n} - 1)$, $a[j][0], a[j][1], \ldots, a[j][\sqrt{n} - 1]$ are read and written. Clearly, memory access to these elements are coalesced. In the computation of the column-wise prefix-sums, $a[0][0], a[1][0], \ldots, a[\sqrt{n} - 1][0]$ are read. After that, for each j $(1 \leq j \leq \sqrt{n}-1)$, $a[0][j], a[1][j], \ldots, a[\sqrt{n}-1][j]$ are read and written. Memory access to these elements are coalesced. Hence, 2R2W SAT algorithm performs $2n - \sqrt{n}$ coalesced memory access operations and $2n - \sqrt{n}$ stride memory access operations. Since 2R2W SAT algorithm has one barrier synchronization step, we have,

Lemma 2: The global memory access cost of 2R2W SAT algorithm is at most $2n + 2\frac{n}{w} + 2(l-1)$.

We can avoid stride memory access when we compute the row-wise prefix-sums by transposing a matrix. More specifically, the row-wise prefix-sums can be obtained by transpose, column-wise prefix-sums, and transpose. It has been shown in [16] that transpose of a matrix of size $\sqrt{n} \times \sqrt{n}$ in the global memory of the HMM can be done in $2n$ coalesced memory access operations with no barrier synchronization step. The idea of the transpose is to partition the matrix into $\frac{\sqrt{n}}{w} \times \frac{\sqrt{n}}{w}$ blocks with $w \times w$ elements each. We can transpose a block via a $w \times w$ matrix with diagonal arrangement in a shared memory of a DMM efficiently. First, a block in the global memory is read in row-wise and it is written in a $w \times w$ matrix with diagonal arrangement in row-wise. After that, the $w \times w$ matrix is read in column-wise and it is written in a a block in the global memory in row-wise. The reader should refer to Figure 7 illustrating transpose of a block using a 4×4 matrix with diagonal

arrangement. By executing this block transpose for all blocks in parallel so that a pair of corresponding two blocks are swapped appropriately, the transpose of a $\sqrt{n} \times \sqrt{n}$ can be done. The reader should refer to [16] for the details of the transpose.

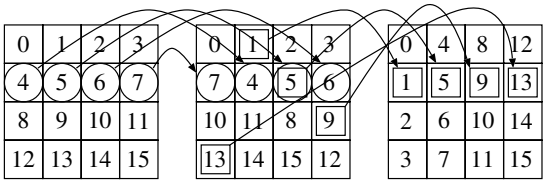

Figure 7. Transpose of a block using a 4×4 matrix with diagonal arrangement

By executing matrix transpose twice, we can design 4R4W SAT algorithm as follows:

[4R4W SAT algorithm]

Step 1: Compute the column-wise prefix-sums

Step 2: Transpose

Step 3: Compute the column-wise prefix-sums

Step 4: Transpose

After Steps 1, 2, and 3, barrier synchronization is necessary. Also, each step needs no more than $2n$ coalesced global memory access. Thus, we have,

Lemma 3: The global memory access cost of 4R4W SAT algorithm is at most $8\frac{n}{w} + 4(l-1)$.

V. 2R1W SAT ALGORITHM

The main purpose of this section is to review a SAT algorithm for GPU shown in [14]. Since this SAT algorithm performs $2n$ read and n write operations to the global memory, we call it 2R1W SAT algorithm. 2R1W SAT algorithm that we will explain is slightly different from that in [14] for easy understanding of the algorithm.

Suppose that a $\sqrt{n} \times \sqrt{n}$ matrix a is partitioned into $\frac{\sqrt{n}}{w} \times \frac{\sqrt{n}}{w}$ blocks of $w \times w$ elements each. 2R1W SAT algorithm has three steps as follows:

[2R1W SAT algorithm]

Step 1: Each DMM reads a block in the global memory and write it in the shared memory. The column-wise sums, the row-wise sums, and the sum of the block are computed. More specifically, for a block a' of size $w \times w$,

- column-wise sums: $C[i] = \sum_{j=0}^{w-1} a'[j][i]$ for all i $(0 \leq i \leq w-1)$,
- row-wise sums: $R[i] = \sum_{j=0}^{w-1} a'[i][j]$ for all i $(0 \leq i \leq w-1)$, and
- sum: $S = \sum_{i=0}^{w-1} \sum_{j=0}^{w-1} a'[i][j]$,

are computed. The column-wise sums of all blocks excluding the bottom blocks are written in the global memory such that they constitute a matrix of size $(\frac{\sqrt{n}}{w} - 1) \times \sqrt{n}$ in the global memory. Similarly, the row-wise sums of all blocks constitute a matrix of size $\sqrt{n} \times (\frac{\sqrt{n}}{w} - 1)$, and the sums constitute a matrix of size $(\frac{\sqrt{n}}{w} - 1) \times (\frac{\sqrt{n}}{w} - 1)$. The reader should refer to Figure 8 for illustrating the resulting values for a matrix in Figure 3 with $w = 3$. Let $\mathcal{C}$, $\mathcal{R}$, and $\mathcal{S}$ denote matrices of the resulting values for the column-wise sums, the row-wise sums, and the sums, respectively. In Figure 8, the sizes of $\mathcal{C}$, $\mathcal{R}$, and $\mathcal{S}$ are 2×9, 9×2, and 2×2, respectively.

Step 2: The column-wise prefix-sums of $\mathcal{C}$ and the row-wise prefix-sums of $\mathcal{R}$ are computed in the same way as 2R2W SAT algorithm. If $\mathcal{S}$ is no larger than $w \times w$ then we compute the SAT of $\mathcal{S}$ using a single DMM. Otherwise, we execute 2R1W SAT algorithm recursively for $\mathcal{S}$. The reader should refer to Figure 8 for the resulting values of $\mathcal{C}$, $\mathcal{R}$, and $\mathcal{S}$.

Step 3-1: Each DMM reads a block from the global memory and write it in the shared memory. Let a' denote a block read by a particular DMM. It reads w elements in $\mathcal{C}$, and adds them to the top row of a' so that each of the resulting sums is the sum of all elements above it, inclusive, in the same column. Similarly, it reads w elements in $\mathcal{R}$, and adds them to the leftmost column of a' so that each of the resulting sums is the sum of all elements to the left-side of it, inclusive, in the same row. Further, an element in $\mathcal{S}$ is added to the top left corner of a' so that the resulting sum is the the sum of all blocks above and to the left of it, inclusive. The reader should refer to Figure 9 illustrating these operations for a block. Also, Figure 8 illustrates the resulting values of all blocks.

Step 3-2: Each DMM computes the SAT of a block obtained in Step 3-1 and the resulting values are written in the global memory. Figure 9 illustrates the values of a block before and after this step. The reader should have no difficulty to confirm that the block thus obtained stores the SAT of the input matrix correctly.

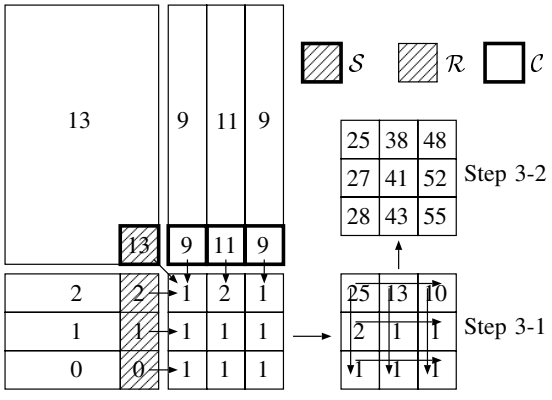

Figure 9. Step 3 of 2R1W SAT algorithm

Let us evaluate the global memory access cost. In Step 1, all elements in a are read, and $\mathcal{C}$, $\mathcal{R}$, and $\mathcal{S}$ are written. Thus, n elements are read from the global memory and less than $2\frac{n}{w} + \frac{n}{w^2}$ elements are written in the global memory.

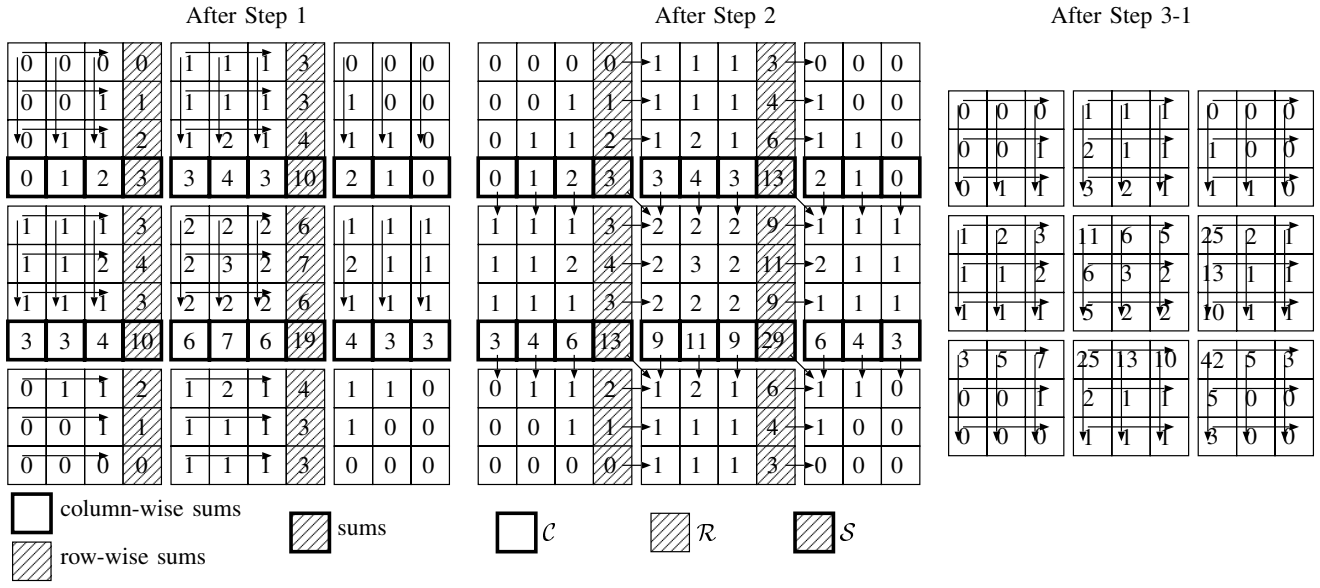

Figure 8. 2R1W SAT algorithm executed for a matrix in Figure 3 with $w = 3$

Note that we should write $\mathcal{R}^T$, that is, transposed $\mathcal{R}$ in the global memory for the purpose of coalesced memory access for the row-wise prefix-sums computation in Step 2. If this is the case, the row-wise prefix-sums of $\mathcal{R}$ corresponds to the column-wise prefix-sums of $\mathcal{R}^T$, which can be computed by coalesced memory access to the global memory. Also, in Step 1, the column-wise sums, the row-wise sums, and the SAT of a block in a shared memory are computed. This computation can be done without bank conflicts by diagonal arrangement of a block. Each of the d DMMs performs the computation of the column-wise sums for $\frac{n}{w^2 d}$ blocks of size $w \times w$. Since the memory access is conflict-free, this takes only $\frac{n}{wd}$ time units, which is so small that it can be hidden by latency overhead.

Step 2 performs the computation of the the column-wise prefix-sums and the row-wise prefix-sums for matrices of $\frac{n}{w}$ elements. It also computes the SAT of $\frac{n}{w^2}$ elements, recursively, which performs global memory access to $O(\frac{n}{w^2})$ elements. In Step 3-1, all elements in a and the resulting values of $\mathcal{C}$, $\mathcal{R}$, and $\mathcal{S}$ are read from the global memory. In Step 3-2 the resulting SAT is written in the global memory. Thus, 2R1W SAT algorithm performs at most $3n+8\frac{n}{w}+O(\frac{n}{w^2})$ coalesced memory access operations. This includes the memory access by recursive computation of $\mathcal{S}$.

Let us evaluate the number of barrier synchronization steps. Barrier synchronization step is necessary after Steps 1 and 2 if the SAT of $\mathcal{S}$ is computed without recursion. If the SAT of $\mathcal{S}$ is computed recursively, additional two barrier synchronization steps is necessary for each recursion. Hence, if 2R1W SAT algorithm involves depth k recursions, it performs $2k + 2$ barrier synchronization steps. Thus, we have,

Lemma 4: The global memory access cost of 2R1W SAT algorithm with recursion depth k is $3\frac{n}{w} + 8\frac{n}{w^2} + O(\frac{n}{w^3}) + (2k + 3)(l - 1)$.

Since $w = 32$ in current GPUs, $d = 0$ if $n \leq 2^{20}$ and $d = 1$ if $n \leq 2^{30}$. Thus, d is no more than 1 from the practical point of view.

VI. Our 1R1W SAT algorithm

The main purpose of this section is to show our novel SAT algorithm called 1R1W SAT algorithm. This algorithm performs only $n + O(\frac{n}{w})$ read operations and $n + O(\frac{n}{w})$ write operations to the global memory. Before showing 1R1W SAT algorithm, we present 4R1W SAT algorithm. By combining techniques used in 4R1W SAT and 2R1W SAT algorithms, we can obtain 1R1W SAT algorithm.

Let b be the SAT of an input $\sqrt{n} \times \sqrt{n}$ matrix a. Suppose that the values of $b[i-1][j-1]$, $b[i][j-1]$, and $b[i-1][j]$ are already computed. We can obtain the value of $b[i][j]$ by evaluating the following formula:

$$b[i][j] = a[i][j] - b[i][j-1] - b[i-1][j] + b[i-1][j-1] \quad (1)$$

From this formula, 4R1W SAT algorithm computes the SAT in a diagonal scan order from the top left to the bottom right. More specifically, 4R1W algorithm has $2\sqrt{n} - 1$ stages and each Stage k ($0 \leq k \leq 2\sqrt{n} - 2$) computes Formula (1) for all i and j such that $i + j = k$. Figure 10 illustrates the computation performed in Stage 7.

Let us evaluate the performance of 4R1W SAT algorithm. To compute each $b[i][j]$, 3 elements in b and 1 element in a are read. Also, the resulting value is written in b. Thus, $4n$ reading operations and n writing operations are performed. Unfortunately, all memory access are stride. Further, barrier

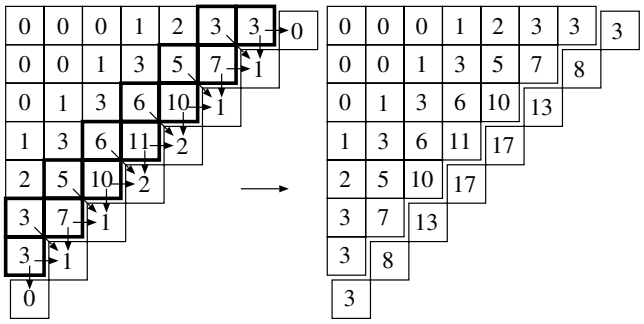

Figure 10. Stage 7 of 4R1W SAT algorithm

synchronization step is necessary after every stage from 0 to $2\sqrt{n}-3$. Thus, we have,

Lemma 5: The global memory access cost of 4R1W SAT algorithm is $5n + (2\sqrt{n}-1)l$.

We are now in a position to show our new 1R1W SAT algorithm. The idea is to extend 4R1W SAT algorithm to perform SAT computation in block-wise. In each block-wise computation, a similar computation to 2R1W SAT algorithm is performed. Again, an input matrix a of size $\sqrt{n} \times \sqrt{n}$ is partitioned into $\frac{\sqrt{n}}{w} \times \frac{\sqrt{n}}{w}$ blocks of size $w \times w$ each. Let $A(i,j)$ $(0 \leq i, j \leq \frac{\sqrt{n}}{w} - 1)$ denote a block in the i-th row and in the j-th column. 1R1W SAT algorithm has $2\frac{\sqrt{n}}{w} - 1$ stages. Each Stage k $(0 \leq k \leq 2\frac{\sqrt{n}}{w} - 2)$ computes the SAT of block $A(i,j)$ with $i + j = k$. The reader should refer to Figure 11 for illustrating the computation performed in Stage 3. In Stage 3, $A(2,1)$ and $A(1,2)$ are computed using the resulting values in $A(2,0)$, $A(1,1)$, and $A(0,2)$. From these resulting values, we can obtain $\mathcal{C}$, $\mathcal{R}$, and $\mathcal{S}$ for each of $A(2,1)$ and $A(1,2)$. For example, in Figure 11, $\mathcal{C}$ for $A(2,1)$ is $(9,11,9)$. This can be obtained by the resulting value 13 of the bottom top corner in $A(1,0)$ and the resulting values $(22,33,42)$ in the bottom row of $A(1,1)$. More specifically, we can obtain $\mathcal{C}$ by computing pairwise subtraction $(22,33,42) - (13,22,33) = (9,11,9)$. After the values of $\mathcal{C}$, $\mathcal{R}$, and $\mathcal{S}$, these values are added to $A(2,1)$ in the same way as Step 3-1 of 2R1W SAT algorithm. Finally, we compute the SAT of $A(2,1)$ in the same way as Step 3-2 of 2R1W SAT algorithm.

Let us evaluate the performance of 1R1W SAT algorithm. In each stage, the values of a block in the global memory are read and the resulting values are written to the global memory. Also, the values necessary to compute $\mathcal{C}$, $\mathcal{R}$, and $\mathcal{S}$ are read and written to the global memory. For each block, $2w+1$ elements are read from the global memory for this task. Since we have $\frac{n}{w^2}$ blocks, $n + (2w+1) \cdot \frac{n}{w^2}$ elements are read and $n + (2w+1) \cdot \frac{n}{w^2}$ elements are written in all stages. Barrier synchronization is necessary after each of Stages from 0 to $2\frac{\sqrt{n}}{w} - 3$. Thus, we have,

Theorem 6: The global memory access cost of 4R1W SAT algorithm is $2\frac{n}{w} + O(\frac{n}{w^2}) + (2\frac{\sqrt{n}}{w} - 2)l$.

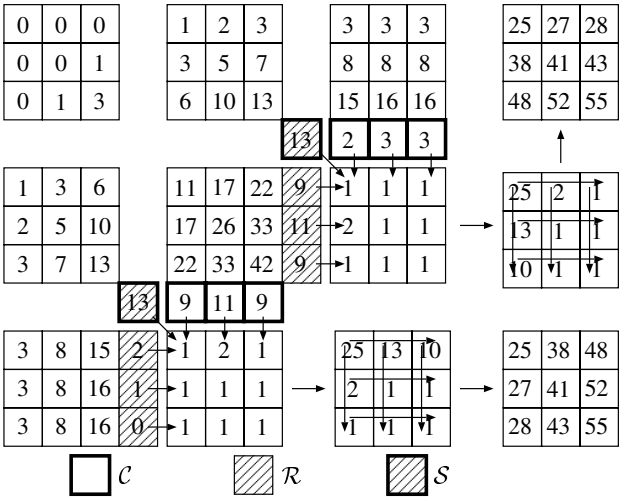

Figure 11. Stage 3 of 1R1W SAT algorithm for a matrix in Figure 3

VII. $(1+r)$R1W SAT ALGORITHM

The main purpose of this section is to accelerate the SAT computation further by combining 1R1W and 2R1W SAT algorithms. The idea of further acceleration is to use 2R1W SAT algorithm in early and late stages of 1R1W SAT algorithms to reduce the latency overhead.

Again, suppose that a $\sqrt{n} \times \sqrt{n}$ matrix a is partitioned into $\frac{\sqrt{n}}{w} \times \frac{\sqrt{n}}{w}$ blocks of size $w \times w$ each. As illustrated in Figure 12, for any fixed parameter r $(0 < r < 1)$, we partition blocks into (A) top left triangle, (B) bottom right triangle, and (C) remaining blocks. Clearly, (A) and (B) have $\frac{\sqrt{r}\sqrt{n}}{w} + (\frac{\sqrt{r}\sqrt{n}}{w} - 1) + \cdots + 1 = \frac{\sqrt{r}\sqrt{n}}{w} \cdot (\frac{\sqrt{r}\sqrt{n}}{w} + 1)/2 \approx \frac{rn}{2w^2}$ blocks each. We first use 2R1W SAT algorithm to compute the SAT of (A). After that, we use 1R1W SAT algorithm for (C). Finally, 2R1W SAT algorithm is used for the computation of the SAT in (B).

Let us evaluate the performance. Since (A) and (B) have approximately $\frac{rn}{2}$ elements in $\frac{rn}{2w^2}$ blocks each, 2R1W SAT algorithm for (A) and (B) performs $rn + O(\frac{rn}{w})$ read operations and $\frac{rn}{2} + O(\frac{rn}{w})$ write operations each. Also, since (C) has $(1-r)n$ elements, 1R1W SAT algorithm for (C) performs $(1-r)n + O(\frac{(1-r)n}{w})$ read operations and $(1-r)n + O(\frac{(1-r)n}{w})$ write operations. Hence, this SAT algorithm performs $(1+r)n + O(\frac{n}{w})$ read operations and $n + O(\frac{n}{w})$ write operations. Thus, we call this SAT algorithm $(1+r)$R1W SAT algorithm. Further, 2R1W SAT algorithm for (A) and (B) needs $2 + 2k$ barrier synchronization steps each, where k is the depth of the recursion of 2R1W SAT algorithm. Since 1R1W SAT algorithm for (C) has $2\frac{(1-\sqrt{r})\sqrt{n}}{w} - 1$ stages, it needs $2\frac{(1-\sqrt{r})\sqrt{n}}{w} - 2$ barrier synchronization steps. Also, after the computation of the SAT for (A) and (B), 1 barrier synchronization step each

is necessary. Totally, $(1+r)$R1W SAT algorithm executes $2\frac{(1-\sqrt{r})\sqrt{n}}{w}+4+4k$ barrier synchronization steps. Thus, we have,

Theorem 7: The global memory access cost of $(1+r)$R1W SAT algorithm is $(2+r)\frac{n}{w}+O(\frac{n}{w^2})+(2\frac{(1-\sqrt{r})\sqrt{n}}{w}+5+4k)l$.

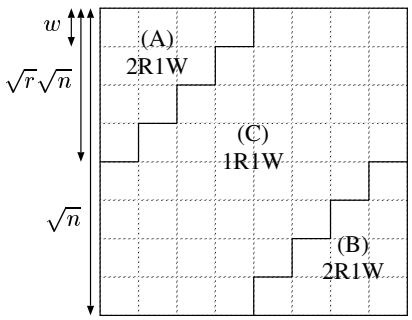

Figure 12. Partition of a matrix for $(1+r)$R1W SAT algorithm

When $r = 0.25$, the global memory access cost of 1.25R1W SAT algorithm is $2.5\frac{n}{w}+(\frac{\sqrt{n}}{w}+5+4d)l$ time units. Since 2R1W and 1R1W SAT algorithms run approximately $3\frac{n}{w}+(3+2d)l$ and $2\frac{n}{w}+2\frac{\sqrt{n}}{w}l$ time units, respectively, 1.25R1W SAT algorithm may run faster than these algorithms. Further, we can select the best value r that minimize the running time of $(1+r)$R1W SAT algorithm.

VIII. Experimental results

We have implemented all SAT algorithms presented so far in this paper on GeForce GTX 780 Ti. Since the number of memory banks and the number of threads in a warp is 32 [4], we have implemented SAT algorithms with $w = 32$. Barrier synchronization of all threads is implemented by invoking separate CUDA kernel calls. For example, one CUDA kernel call is invoked for each of $2\frac{\sqrt{n}}{w}-1$ stages of 4R1W SAT algorithm. We have tested several configuration in terms of the number of threads in a CUDA block, and selected the best configuration. For example, in 2R2W, 4R4W, and 4R1W SAT algorithms, CUDA blocks with 64 threads each are invoked. Since each Stage i and each Stage $2\frac{\sqrt{n}}{w}-1-i$ $(0 \leq i \leq \frac{\sqrt{n}}{w}-1)$ of 4R1W SAT algorithm computes $i+1$ values of the SAT, it uses $i+1$ threads in $\frac{i+1}{64}$ CUDA blocks. In 2R1W and 1R1W SAT algorithm, a 32×32 block in a matrix is copied to the shared memory in a streaming multiprocessor and the column-wise sums, the row-wise sums, and/or the SAT of it is computed. After that, the resulting values are copied to the global memory. For this operation, we use one CUDA block with 128 threads for each 32×32 block. All 128 threads in a CUDA block are used to copy a 32×32 block in the shared memory and 32 threads out of 128 threads are used to compute the column-wise sums, the row-wise sums, and/or the SAT.

Table II shows the running time of SAT algorithms for a double (64-bit) matrix of size from 1K×1K ($= 1024 \times 1024$) to 18K×18K ($= 18432 \times 18432$). Since a 18K×18K 64-bit matrix uses 2.53GBytes, it is hard to store a matrix larger than it in the global memory of GeForce GTX 780 Ti of size 3GBytes. The running time of the best SAT algorithm for each value of $\sqrt{n}$ is highlighted in boldface. Since 4R1W SAT algorithm performs a lot of kernel calls and stride memory access, and has large memory access latency overhead, it needs much more computing time than the other algorithms. Recall that 4R4W SAT algorithm corresponds to 2R2W SAT algorithm with transpose and 4R4W SAT algorithm performs much more memory access operations than 2R2W SAT algorithm. Since 2R2W SAT algorithm performs stride memory access, it is much slower than 4R4W SAT algorithm. These experimental results imply that stride memory access imposes a large penalty on the computing time.

Recall that 2R1W and 1R1W SAT algorithms are block-based algorithms, that perform $3n+O(\frac{n}{w})$ and $2n+O(\frac{n}{w})$ global memory access operations, respectively. Hence, they are faster than 4R4W SAT algorithm, which performs approximately $8n$ global memory access operations. Although 1R1W SAT algorithm performs fewer global memory access operations than 2R1W SAT algorithm, it runs slower when $\sqrt{n} \leq$6K. The reason is that 1R1W SAT algorithm has a larger latency overhead than 2R1W SAT algorithm and the latency overhead dominates the bandwidth overhead when the size of input is small. 1.25R1W SAT algorithm runs faster than both 2R1W and 1R1W SAT algorithms whenever $\sqrt{n} \geq$5K. We have evaluated the computing time for all possible values $r = \frac{w^2}{n}, \frac{(2w)^2}{n}, \ldots, \frac{(\sqrt{n}-w)^2}{n}$ to find the best value r that minimize the running time of $(1+r)$R1W. Table II also shows the values of r $(0 < r < 1)$ that minimize the running time of $(1+r)$R1W SAT algorithm. From the table, we can see that $(1+r)$R1W SAT algorithm attain the best performance when $\sqrt{n} \geq$5K. Also, the value of r that gives the best performance decreases as the size of a matrix increases. This is because the memory bandwidth overhead of 1R1W SAT algorithm dominates the latency overhead for larger matrices and 1R1W SAT algorithm has better performance than 2R1W algorithm. We can conjecture that 1R1W SAT algorithm could be the best if an input matrix was much larger than 18K×18K.

To see a speed-up factor of SAT algorithms running on the GPU over a conventional CPU, we have evaluated the performance of several sequential SAT algorithms on Intel Xeon X7460 (2.66GHz). Table II shows the running time of top two sequential algorithms as follows:

2R2W(CPU): The column-wise prefix-sums are computed in a raster scan order from the top row to the bottom row. More specifically, $a[i+1][j] \leftarrow a[i+1][j]+a[i][j]$ is executed in a raster scan order of (i,j). The row-wise

Table II
THE RUNNING TIME OF SAT ALGORITHM (IN MILLISECONDS) AND THE VALUE OF r THAT MINIMIZE THE RUNNING TIME OF $(1+r)$R1W SAT ALGORITHM FOR MATRICES OF SIZES FROM 1K$\times$1K TO 18K$\times$18K

SAT Algorithms	1K	2K	3K	4K	5K	6K	7K	8K	10K	12K	14K	16K	18K
2R2W	1.47	3.28	5.71	9.53	13.6	23.9	27.1	47.8	90.8	163	160	234	401
4R4W	1.07	2.52	4.48	6.77	9.67	13.7	17.2	22.2	33.9	50.4	64.2	83.1	117
4R1W	11.5	22.9	36.4	50.1	113	104	173	252	315	597	437	742	1600
2R1W	**0.332**	**0.850**	**1.83**	**3.09**	4.79	6.78	9.25	12.3	18.9	27.2	36.8	48.7	61
1R1W	0.902	1.46	2.43	3.65	5.05	6.81	8.71	10.9	16.2	22.6	29.7	38	53.8
1.25R1W	0.453	1.05	1.96	3.25	4.71	6.41	8.47	10.8	16.5	23	31.2	40.7	57.6
fastest $(1+r)$R1W	0.365	0.958	1.94	3.16	**4.58**	**6.32**	**8.25**	**10.5**	**15.7**	**22.0**	**29.1**	**37.5**	**53.1**
r $(0<r<1)$	0.168	0.174	0.172	0.159	0.136	0.123	0.0876	0.103	0.0963	0.0710	0.0835	0.0694	0.0725
2R2W(CPU)	25.9	107	241	427	670	966	1310	1690	2670	3850	5250	6760	8670
4R1W(CPU)	18.0	73.2	165	293	459	660	904	1160	1830	2660	3600	4590	5950

prefix-sums are also computed in a raster scan order, that is, $a[i][j+1] \leftarrow a[i][j+1]+a[i][j]$ is executed in a raster scan order of (i, j).

4R1W(CPU): Formula (1) is evaluated in a raster scan order of (i, j).

From the table, we can see that 4R1W(CPU) SAT algorithm runs faster than 2R2W(CPU) SAT algorithm, because of the memory access locality. Also, $(1+r)$R1W SAT algorithm runs more than 100 times faster than 4R1W(CPU) SAT algorithm when $\sqrt{n} \geq$5K.

IX. CONCLUSION

The main contribution of this paper is to introduce the asynchronous Hierarchical Memory Machine, which capture the essence of CUDA-enabled GPUs. We have also presented a global-memory-access-optimal parallel algorithm for computing the summed area table on the asynchronous HMM. The experimental results on GeForce GTX 780 Ti showed that our best algorithm, $(1+r)$R1W SAT algorithm, runs faster than any other algorithms for an input matrix of size 5K$\times$5K or larger. It also runs at least 100 times faster than the best sequential algorithm running on a single CPU.

REFERENCES

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

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

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

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

[5] D. Man, K. Uda, H. Ueyama, Y. Ito, and K. Nakano, "Implementations of a parallel algorithm for computing euclidean distance map in multicore processors and GPUs," *International Journal of Networking and Computing*, vol. 1, no. 2, pp. 260–276, July 2011.

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

[7] Y. Ito and K. Nakano, "A GPU implementation of dynamic programming for the optimal polygon triangulation," *IEICE Transactions on Information and Systems*, vol. E96-D, no. 12, pp. 2596–2603, Dec. 2013.

[8] K. Nakano, "Simple memory machine models for GPUs," *International Journal of Parallel, Emergent and Distributed Systems*, vol. 29, no. 1, pp. 17–37, 2014.

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

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

[11] F. Crow, "Summed-area tables for texture mapping," in *Proc. of the 11th annual conference on Computer graphics and interactive techniques*, 1984, pp. 207–212.

[12] A. Lauritzen, "Chapter 8: Summed-area variance shadow maps," in *GPU Gems 3*. Addison-Wesley, 2007.

[13] K. Nakano, "Optimal parallel algorithms for computing the sum, the prefix-sums, and the summed area table on the memory machine models," *IEICE Trans. on Information and Systems*, vol. E96-D, no. 12, pp. 2626–2634, 2013.

[14] D. Nehab, A. Maximo, R. S. Lima, and H. Hoppe, "GPU-efficient recursive filtering and summed-area tables," *ACM Trans. Graph.*, vol. 30, no. 6, p. 176, 2011.

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

[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*. IEEE CS Press, Oct. 2013, pp. pp. 1–10.