# An Optimal Parallel Algorithm for Computing the Summed Area Table on the GPU

Yutaro Emoto, Shunji Funasaka, Hiroki Tokura, Takumi Honda, Koji Nakano, and Yasuaki Ito
*Department of Information Engineering*
*Hiroshima University*
*Kagamiyama 1-4-1, Higashi Hiroshima, 739-8527 Japan*

***Abstract*—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 present a very efficient parallel algorithm for computing the SAT of a matrix stored in the global memory of the GPU. Our new parallel algorithm uses two techniques, single kernel soft synchronization and look back techniques to compute the SAT efficiently. It performs approximately one read and one write operations per element to the global memory. Since all elements in the matrix must be read once, and those in the resulting SAT must be written, any SAT computation cannot be faster than duplication of the matrix in the global memory. Thus, our algorithm is theoretically optimal in terms of global memory access. We have implemented our parallel algorithm and previously published algorithms for computing the SAT to run on NVIDIA TITAN V GPU. Our parallel SAT algorithm runs faster than all previous algorithms for matrices of sizes from $256 \times 256$ to $32K \times 32K$. Also, the overhead ratio over matrix duplication can be only 5.7%, so it is also practically optimal.**

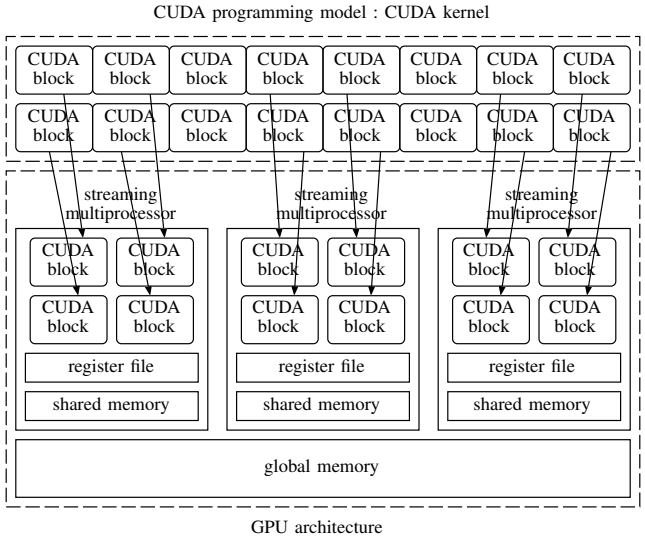


Figure 1. CUDA programming model and GPU architecture

## I. Introduction

### A. Background

*The GPU* (Graphics Processing Unit), is a specialized circuit designed to accelerate computation for building and manipulating images [1], [2]. 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) [3], 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], [4], [5], since they have hundreds of processor cores and very high memory bandwidth.

A CUDA-enabled GPU has multiple steaming multiprocessors, each of which has processor cores, integer and floating point operation units, shared memory, register file, and L1 cache. A CUDA program running on a host PC invokes one or more *CUDA kernels* executed on a GPU one by one. Each CUDA kernel consists of one or more *CUDA blocks*, each of which is a set of threads running on a streaming multiprocessor. Further, threads in a CUDA block is partitioned into groups of $w = 32$ threads called *warps*. Warps are dispatched to processor cores by schedulers, and all $w$ threads in the same warp execute the same instruction. When a CUDA kernel is executed, CUDA blocks are dispatched to streaming multiprocessors in turn. If the number of CUDA blocks in a CUDA kernel running on a GPU exceeds the total number of CUDA blocks as illustrated in Figure 1, CUDA blocks that are not allocated in a streaming multiprocessor wait for termination of a running CUDA block. Since there is no explicit rule of CUDA block assignment to streaming multiprocessors, we need to design CUDA kernel programs so that they work correctly for any CUDA block assignment to streaming multiprocessors by a dispatcher. Hence, there is no direct way to communicate between CUDA blocks in the same CUDA kernel.

CUDA programs can use two types of memories in the GPU: *the shared memory* and *the global memory* [3]. Each streaming multiprocessor has the shared memory, an extremely fast on-chip memory with lower capacity, say, 16-96 Kbytes, and low latency. Every streaming multiprocessor shares the global memory implemented as an off-chip DDR/GDDR DRAM or on-chip HBM2 DRAM with large

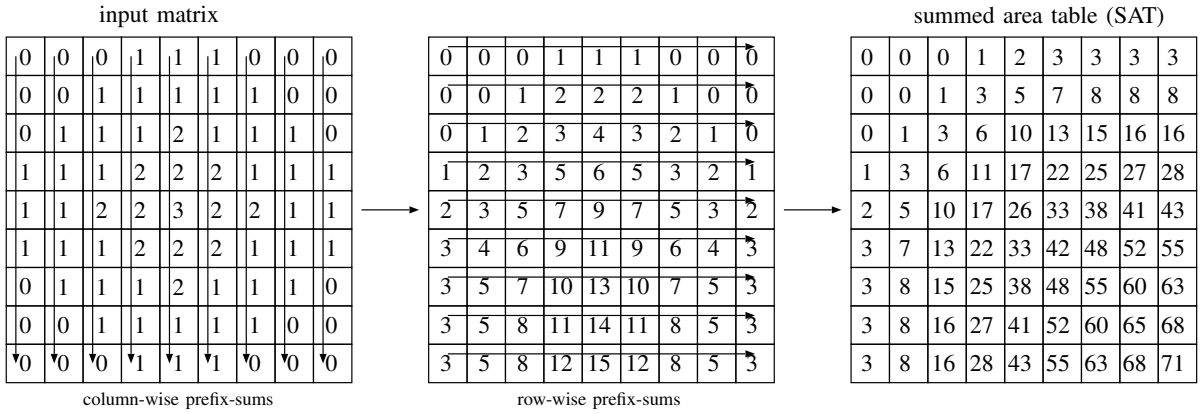


Figure 2. The summed area table (SAT) of a $9 \times 9$ matrix

capacity, say, 1.5-12 Gbytes, but its access latency is quite large. When CUDA block is dispatched to the streaming multiprocessor, space of the shared memory is allocated to it. Each CUDA block can access the allocated space, but it cannot access the space allocated to the other CUDA blocks. 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 [4], [3]. The address space of shared memory is mapped into several physical memory banks. If two or more threads in a warp access the same memory bank 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 bank conflicts. Also, to maximize the bandwidth of the global memory, threads in the same warp should perform coalesced access.

The computation of prefix-sums is an important task in the area of parallel computing [6]. Suppose that an array $p$ of size $n$ is given. *The prefix-sums* of an array $p$ is a sequence of length $n$ such that each $i$-th $(0 \leq i \leq n-1)$ element is $p[0] + p[1] + \cdots + p[i]$. Clearly, by executing $p[i] \leftarrow p[i-1]+p[i]$ for all $i$ $(1 \leq i \leq n-1)$ in turn, an array $p$ stores the prefix-sums. *The summed area table (SAT)* [7], [8], [9] is the prefix-sums extended to a 2-dimensional array defined as follows. Let $a$ be a matrix $a$ of size $\sqrt{n} \times \sqrt{n}$ is given. The SAT is a matrix $b$ of the same size such that

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

It should have no difficulty to confirm that the SAT can be obtained by computing the column-wise prefix-sums and then the row-wise prefix-sums as illustrated in Figure 2. Thus, the SAT of an $n \times n$ matrix can be computed in $O(n^2)$ sequential time. Once we have the SAT, the sum of any rectangular area of $a$ can be computed by the following formula

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

Thus, the sum of any rectangular area in matrix $a$ can be computed using four elements of the SAT $b$. Since the sum of any rectangular area can be computed in $O(1)$ time, the SAT has a lot of applications in the area of image processing and computer vision [8].

### *B. The summed area table computation and related work*

The SAT can be obtained by computing the column-wise prefix-sums and then the row-wise prefix-sums of an input matrix as illustrated in Figure 2. We can assign each of $n$ threads to a column and compute the prefix-sums of each column sequentially by one thread. Clearly, each element is read once and write once, $n^2$ read and $n^2$ write operations to the global memory are performed. The row-wise prefix-sums can be computed in the same way. This algorithm calls two kernels with $n$ threads for the column-wise and the row-wise prefix-sum computation, and performs $2n^2$ read and $2n^2$ write operations. Since each element is read and written two times each, we call this algorithm *2R2W algorithm*. Note that a matrix is arranged in row-major order of the memory space, that is, each element $a[i][j]$ $(0 \leq i, j \leq n-1)$ at row $i$ and column $j$ is arranged in offset $i \cdot n + j$ of the memory space for $a$. Thus, the column-wise prefix-sum computation illustrated in Figure 2 performs coalesced memory access, because $n$ threads access to the same raw at the same time. On the other hand, $n$ threads in the row-wise prefix-sum computation access to the same column at the same time and so stride memory access is performed. Hence, the running time of 2R2W algorithm is much larger than that of matrix duplication. Also, the number of used threads is small, it does not fully utilize memory access bandwidth. For coalesced global memory access, we should use one or more CUDA blocks for each row to compute

the prefix-sums, because elements in a row are consecutive. By executing the 1-dimensional prefix-sum algorithm by Merrill and Garland [10], [11] for every row, the row-wise prefix-sums can be computed very efficiently. Further, we can use a column-wise prefix-sum algorithm presented by Tokura *et al.* [12] which also uses more threads. We call the SAT algorithm using the prefix-sum algorithm by Merrill and Garland [10], [11] and the column-wise prefix-sum algorithm by Tokura *et al.* [12] *2R2W-optimal algorithm*.

To reduce the global memory access, the shared memory should be used as a cache. Nehab *et al.* [13] have presented a tile-based SAT algorithm, that we call *2R1W algorithm*. The input matrix is partitioned into $\frac{N^2}{W^2}$ tiles of size $W \times W$ each. Usually, $W$ is $w = 32$ or a small multiple of $w$, such that all elements in a tile can be stored in the shared memory. It executes three kernel calls, in which $2n^2 + O(\frac{n^2}{W})$ read and $n^2 + O(\frac{n^2}{W})$ write operations are performed totally. Kasagi *et al.* [14] have presented a global memory access optimal SAT algorithm that we call *1R1W algorithm*. Since it performs only $n^2 + O(\frac{n^2}{W})$ read and $n^2 + O(\frac{n^2}{W})$ write operations, it is optimal in terms of the global memory access. However, it executes $2\frac{n}{W} - 1$ kernel calls using fewer threads and the performance is degraded due to overhead of many kernel calls and low parallelism. To improve the performance, they also presented $(1+r)$R1W algorithm, which is a hybrid of 2R1W algorithm and 1R1W algorithm. Quite recently, Funasaka *et al.* [15] have presented a global memory access optimal SAT algorithm with single kernel soft synchronization (SKSS) technique, that we call *1R1W-SKSS algorithm*. It executes only one kernel call and uses more threads than *1R1W algorithm*. However, tiles are processed from the top to the bottom one by one, so parallelism is not high enough. Our main contribution of this paper is to present a SAT algorithm that we call *1R1W-SKSS-LB algorithm*. It executes only one kernel call with SKSS and look back (LB) technique. Since our 1R1W-SKSS-LB algorithm assigns CUDA blocks to all tiles, it uses much more threads than 1R1W-SKSS algorithm. Thus, it runs faster than previously published SAT algorithm including 1R1W-SKSS algorithm.

### C. Our contribution

In this paper, we will present an almost optimal parallel algorithm for computing the SAT on the GPU. We assume that an input matrix $a$ is stored in the global memory of the GPU and the resulting SAT $b$ must be written in the global memory. The performance of a parallel algorithm is evaluated by the running time necessary to write the SAT $b$ in the global memory from an input matrix $a$. Since all elements in an input matrix $a$ must be read and those in the resulting matrix $b$ must be written, any SAT algorithm must issue $n^2$ read and $n^2$ write requests to the global memory and the running time cannot be smaller than matrix duplication of the same size. Thus, the time for matrix duplication is a lower bound of that for the SAT computation. Since our parallel SAT algorithm that we call *1R1W-SKSS-LB algorithm* runs can be only 5.7% slower than matrix duplication by cudaMemcpy() on NVIDIA TITAN V GPU, it is practically optimal.

Table I summarizes parallel SAT algorithms for an $n \times n$ input matrix. It shows the total number of kernel calls (kernel calls, smaller is better), the maximum number of used threads over all kernel calls (threads, larger is better), the total number of read operations to the global memory (global memory reads, smaller is better), and that of write operations (global memory writes, smaller is better) for each parallel algorithm. In the table, tile-based parallel algorithms partitions an $n \times n$ input matrix into $\frac{n^2}{W^2}$ tiles of size $W \times W$ each and $m$ is a parameter such that a CUDA block with $\frac{W^2}{m}$ threads ($1 \leq m \leq W$) are assigned to each tile for SAT computation. We can select the values of $W$ and $m$ that maximize the performance by experiment. Further, we can classify SAT algorithm in Table I in terms of the maximum number of used threads as follows: *low parallelism*-$n$, *medium parallelism*-$\frac{nW}{m}$, and *high parallelism*-$\frac{n^2}{m}$. Note that $n \leq \frac{nW}{m} \leq \frac{n^2}{m}$ always holds from $W \leq n$ and $1 \leq m \leq W$. To hide memory access latency to the shared/global memories and to fully utilize hardware resources such as processor cores and memory access bandwidth, memory access requests must be sent by a lot of threads and parallelism must be high. Also, since kernel calls have overheads to invoke CUDA blocks/threads and following kernel call can start only after all threads terminate, the number of kernel calls executed by a CUDA program must be as small as possible. From Table I, we can see that our 1R1W-SKSS-LB algorithm has more potential than previously published algorithms from the theoretical point of view.

This paper is organized as follows. Section II shows fundamental GPU techniques and algorithms to understand SAT algorithms. In Section III, we go on to review previously published tile-based SAT algorithms. Section IV presents our 1R1W-SKSS-LB algorithm. Finally, in Section V, we show experimental results using NVIDIA TITAN V. Section VI concludes our work.

## II. Fundamental GPU techniques and algorithms

Let $w = 32$ denote the number of threads in a warp of the GPU. We introduce *the diagonal arrangement* [16], [17] to store a $w \times w$ matrix in the shared memory. Usually, each $a[i][j]$ ($0 \leq i, j \leq w - 1$) is arranged in offset $i \cdot w + j$ in the shared memory. Since offset $k$ is arranged in bank $k \bmod w$, $a[i][j]$ is stored in bank $j$. Thus, row-wise access to $w$ elements $a[i][0], a[i][1], \ldots, a[i][w-1]$ for each $i$ is conflict-free, because no two elements are arranged in the same bank. On the other hand, column-wise access to $w$ elements $a[0][j], a[1][j], \ldots, a[j][w-1]$ for each $j$ is destined for the same bank $j$. For conflict-free access, we

Table I
PARALLEL ALGORITHMS FOR COMPUTING THE SUMMED AREA TABLE

| Parallel algorithms | kernel calls | threads | parallelism | global memory reads | global memory writes |
|---|---|---|---|---|---|
| 2R2W algorithm | 2 | $n$ | low | $2n^2$ | $2n^2$ |
| 2R2W-optimal algorithm [10], [12] | 2 | $\frac{n^2}{m}$ | high | $2n^2 + O(n^2)$ | $2n^2 + O(n^2)$ |
| 2R1W algorithm [13] | 3 | $\frac{n^2}{m}$ | high | $2n^2 + O(\frac{n^2}{W})$ | $n^2 + O(\frac{n^2}{W})$ |
| 1R1W algorithm [14] | $2\frac{n}{W} - 1$ | $\frac{nW}{m}$ | medium | $n^2 + O(\frac{n^2}{W})$ | $n^2 + O(\frac{n^2}{W})$ |
| $(1+r)$R1W algorithm [14] | $2(1-\sqrt{r})\frac{n}{W} + 5$ | $\max(\frac{rn^2}{2m}, \frac{nW}{m})$ | medium | $(1+r)n^2 + O(\frac{n^2}{W})$ | $n^2 + O(\frac{n^2}{W})$ |
| 1R1W-SKSS algorithm [15] | 1 | $\frac{nW}{m}$ | medium | $n^2 + O(\frac{n^2}{W})$ | $n^2 + O(\frac{n^2}{W})$ |
| Our 1R1W-SKSS-LB algorithm | 1 | $\frac{n^2}{m}$ | high | $n^2 + O(\frac{n^2}{W})$ | $n^2 + O(\frac{n^2}{W})$ |

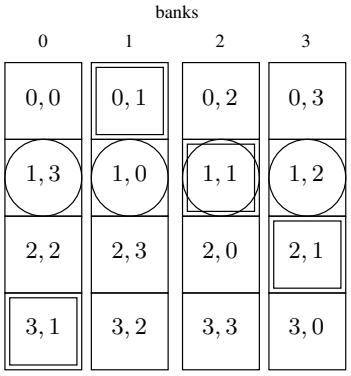


Figure 3. The diagonal arrangement when $w = 4$

can use the diagonal arrangement, which arranges $a[i][j]$ in offset $i \cdot w + (i+j) \bmod w$. Row-wise access to $w$ elements $a[i][0], a[i][1], \ldots, a[0][w-1]$ is still conflict-free, because they are arranged in banks $(i+0) \bmod w$, $(i+1) \bmod w$, $\ldots$, $(i+w-1) \bmod w$. In addition, column-wise access to $w$ elements $a[0][j], a[1][j], \ldots, a[w-1][j]$ is conflict-free, because they are arranged in banks $(0+j) \bmod w$, $(1+j) \bmod w$, $\ldots$, $(w-1+j) \bmod w$. Figure 3 shows the diagonal arrangement when $w = 4$. We can see that $w$ elements $a[1][0], a[1][1], a[1][2]$, and $a[1][3]$ in the same row are arranged in distinct banks, and $w$ elements $a[0][1], a[1][1], a[2][1]$, and $a[3][1]$ in the same column are also stored in distinct banks. Thus, memory access by $w$ threads in a warp to the same row/column of a $w \times w$ matrix in the shared memory is conflict-free if it is arranged in the diagonal arrangement. Also, the diagonal arrangement can be easily generalized to a $W \times W$ matrix, where $W$ is a small multiple of $w$, such that any $w$ consecutive elements in the same row or column are arranged in distinct banks.

We show fundamental techniques to compute the SAT, the column-wise/row-wise sums, and the sum of a matrix $a$ of size $W \times W$ stored in the global memory by a CUDA block efficiently. We assume that $W$ is so small that $a$ can be stored in the shared memory. The SAT of $a$ can be computed using a CUDA block with $\frac{W^2}{m}$ threads, where $m$ is a parameter $(1 \leq m \leq W)$. The SAT can be computed as follows:

**[Shared memory SAT algorithm]**

**Step 1** We copy matrix $a$ in the global memory to the shared memory in the diagonal arrangement using $\frac{W^2}{m}$ threads. More specifically, $\frac{W^2}{m}$ threads are partitioned into $\frac{W}{m}$ groups of $W$ threads each, and each group works for copying $m$ consecutive rows such that each thread copies $m$ elements in the same column.

**Step 2** We use $W$ threads and each thread $i$ $(0 \leq i \leq W-1)$ computes the prefix-sums of row $i$ of the matrix sequentially.

**Step 3** We use $W$ threads and compute the column-wise prefix-sums of the matrix in the same way.

**Step 4** The resulting SAT is copied to the global memory using $\frac{W^2}{m}$ threads.

Note that, barrier synchronization call __syncthreads() must be executed after each of Steps 1, 2, and 3. Since the global memory latency is large, we should use as many threads as possible for Steps 1 and 2. On the other hand, the shared memory latency is not large, the performance cannot be improved even if we use more threads in Steps 2 and 3. Actually, we have implemented a variety of algorithms for the same task and we found that no algorithm can not be better than this shared memory SAT algorithm.

Next, we will show an efficient GPU algorithm for computing the column-wise and row-wise sums of $a$. Again, we use a CUDA block with $\frac{W^2}{m}$ threads $(1 \leq m \leq \frac{W}{2})$ and compute the column-wise and row-wise sums as follows:

**[Shared memory column-wise/row-wise sum algorithm]**

**Step 1** We copy matrix $a$ in the global memory to the shared memory in the diagonal arrangement using $\frac{W^2}{m}$ threads. Again, $\frac{W^2}{m}$ threads are partitioned into $\frac{W}{m}$ groups of $W$ threads each, and each group work for copying $m$ consecutive rows. During the copy operation, each group computes the column-wise sums of assigned $\frac{W}{m}$ rows at the same time. The column-wise sums thus obtained can be stored in an $\frac{W}{m} \times W$ array in the shared memory.

**Step 2** We use $2W$ threads such that column-wise sums and row-wise sums of $a$ are computed using $W$ threads each. The column-wise sums are obtained by computing the column-wise sums of the $\frac{W}{m} \times W$ array using $W$ threads, and writes them in the shared memory. At the same time, the row-wise

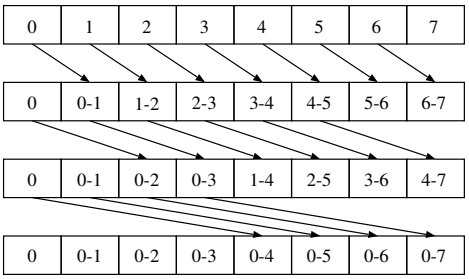


Figure 4. Warp prefix-sum algorithm when $w = 8$

sums of $a$ in the shared memory can be computed using $W$ threads.

Note that, __syncthreads() must be executed after Step 1.

The prefix-sums of an array $a$ of size $w = 32$ can be computed in $\log_2 w = 5$ iterations [18]. We assume that $a$ is stored in the registers of $w$ threads such that each thread $i$ has $a[i]$. The prefix-sums can be computed by the following algorithm:

**[Warp prefix-sum algorithm]**
for $j \leftarrow 0$ to $\log_2 w - 1$ do
  for $i \leftarrow 0$ to $w - 1$ do in parallel
    if $i \geq 2^j$ then thread $i$
      performs $a[i] \leftarrow a[i] + a[i - 2^j]$

Each thread $i$ must read $a[i + 2^j]$ stored in a register of thread $i + 2^j$. This can be done by a warp shuffle function call __shfl(), by which register values in threads in the same warp can be exchanged directly [3]. Thus, this algorithm runs very efficiently on the GPU. Figure 4 shows computation performed by warp prefix-sum algorithm when $w = 8$. From the figure, we can see that the prefix-sums can be computed correctly. Since the last element $a[w - 1]$ stores the sum, this algorithm can also be used to compute the sum.

## III. Previously published tile-based SAT algorithms

This section reviews previously presented SAT algorithms.

Let $S[U : D][L : R]$ denote the sum of a rectangular region of an $n \times n$ matrix $a$ such that

$$S[U : D][L : R] = \sum_{i'=U}^{D} \sum_{j'=L}^{R} a[i'][j'].$$

Clearly, the SAT of $a$ is a matrix $s$ of the same size such that $s[i][j] = S[0 : i][0 : j]$ for all $i$ and $j$ $(0 \leq i, j \leq n-1)$. We partition matrix $a$ into $\frac{n^2}{W^2}$ tiles $T(I, J)$ $(0 \leq I, J \leq \frac{n}{W} - 1)$ of size $W \times W$ such that $T(I, J)$ has $W^2$ elements $a[WI + i][WJ + j]$ $(0 \leq i, j \leq W-1)$. For later reference, we define the sums of various regions for each tile $T(I, J)$ as shown in Table II. Further, for reader's benefits, Figure 5 illustrates some of various sums in Table II. For example, $\mathrm{GSAT}(I, J)$ is a $W \times W$ matrix such that each $(i, j)$ $(0 \leq i, j \leq W - 1)$ element is

$$S[0 : WI + i][0 : WJ + j] = \sum_{i'=WI}^{WI+i} \sum_{j'=WJ}^{WJ+j} a[i'][j'].$$

Hence, computation of the SAT of $a$ is completed if we have $\mathrm{GSAT}(I, J)$ for all tiles $T(I, J)$ $(0 \leq I, J \leq \frac{n}{W} - 1)$.

### A. 2R1W algorithm [13]

Following three kernel calls are performed by 2R1W algorithm:

**Kernel 1**: Compute the LRS, LCS and LS of all tiles.

**Kernel 2**: Compute the GRS, GCS, and GS of all tiles, from the LRS, LCS and LS.

**Kernel 3**: Compute the GSAT of all tiles from the GRS, GCS, and GS.

Figure 6 illustrates an example of computation performed by 2R1W algorithm. Kernel 1 performs shared memory column-wise/row-wise sum algorithm to compute the LRS and LCS. After that, it executes warp prefix-sum algorithm for the LCS to compute the $\mathrm{GS}(I, J)$, and writes it in the global memory. We can assume that, when Kernel 1 terminates, all values of both LRS and LCS can be stored in an array of size $\frac{n}{W} \times n$ each so that $W$ elements in the LRS/LCS in the same tile are consecutive positions in the same row of the array. Also, all values of LS can be stored in an $\frac{n}{W} \times \frac{n}{W}$ array. In Kernel 2, GRS and GCS can be obtained by computing the column-wise prefix-sums of the LRS and LCS stored in $\frac{n}{W} \times n$ arrays using $n$ threads each. More specifically, each of $n$ threads is assigned to a column of the array and computes the prefix-sums sequentially. Further, all values of GS can be obtained by recursive computation for the SAT of the $\frac{n}{W} \times \frac{n}{W}$ array storing values of LS. Alternatively, we can simply use 2R2W algorithm for computing the GS. In Kernel 3, each CUDA block executes shared memory SAT algorithm for $T(I, J)$ as follows. After Step 1 of the algorithm, $\mathrm{GRS}(I, J - 1)$, $\mathrm{GCS}(I-1, J)$ and $\mathrm{GS}(I-1, J-1)$ are added to the leftmost column, the topmost row and the top-left element of $T(I, J)$ in the shared memory, respectively. Steps 2, 3, and 4 are executed as they are. Since the SAT obtained in Kernel 3 is $\mathrm{GSAT}(I, J)$, 2R1W algorithm computes the SAT correctly.

### B. 1R1W algorithm [14]

The SAT can be computed in $2\frac{n}{W} - 1$ kernel calls as follows. In each Kernel $K$ $(0 \leq K \leq 2\frac{n}{W} - 2)$, 1R1W algorithm computes $\mathrm{GSAT}(I, J)$ with $I + J = K$. Figure 7 shows computation performed by Kernel 3 of 1R1W algorithm, which computes $\mathrm{GSAT}(1, 2)$ and $\mathrm{GSAT}(2, 1)$. A CUDA block assigned to $T(I, J)$ basically performs shared memory SAT algorithm as follows. After Step 1 is completed, $\mathrm{GRS}(I, J-1)$, $\mathrm{GCS}(I-1, J)$ and $\mathrm{GS}(I-1, J-1)$ are added to the leftmost column, the topmost row, and the

Table II
VARIOUS SUMS FOR TILE $T(I, J)$

| | size | notations | values |
|---|---|---|---|
| Local Column-wise Sum (LCS) | $W$ | $\mathrm{LCS}(I, J)[j]$ | $S[WI : WI + W - 1][WJ + j : WJ + j]$ |
| Local Row-wise Sum (LRS) | $W$ | $\mathrm{LRS}(I, J)[i]$ | $S[WI + i : WI + i][WJ : WJ + W - 1]$ |
| Local Sum (LS) | 1 | $\mathrm{LS}(I, J)$ | $S[WI : WI + W - 1][WJ : WJ + W - 1]$ |
| Global Column-wise Sum (GCS) | $W$ | $\mathrm{GCS}(I, J)[j]$ | $S[0 : WI + W - 1][WJ + j : WJ + j]$ |
| Global Row-wise Sum (GRS) | $W$ | $\mathrm{GRS}(I, J)[i]$ | $S[WI + i : WI + i][0 : WJ + W - 1]$ |
| Global L-shape Sum (GLS) | 1 | $\mathrm{GLS}(I, J)$ | $\mathrm{GS}(I, J) - \mathrm{GS}(I - 1, J - 1)$ |
| Global Sum (GS) | 1 | $\mathrm{GS}(I, J)$ | $S[0 : WI + W - 1][0 : WJ + W - 1] = \mathrm{GSAT}(I, J)[W - 1][W - 1]$ |
| Global Column-wise Prefix-sums (GCP) | $W$ | $\mathrm{GCP}(I, J)[j]$ | $S[0 : WI + W - 1][0 : WJ + j]$ |
| Global Summed Area Table (GSAT) | $W \times W$ | $\mathrm{GSAT}(I, J)[i][j]$ | $S[0 : WI + i][0 : WJ + j]$ |

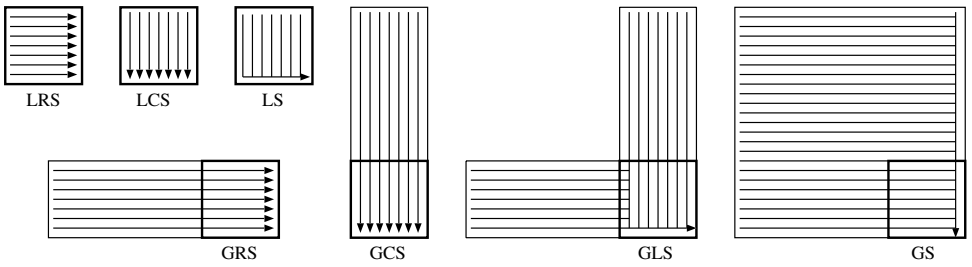


Figure 5. Illustrating various sums for tile $T(I, J)$

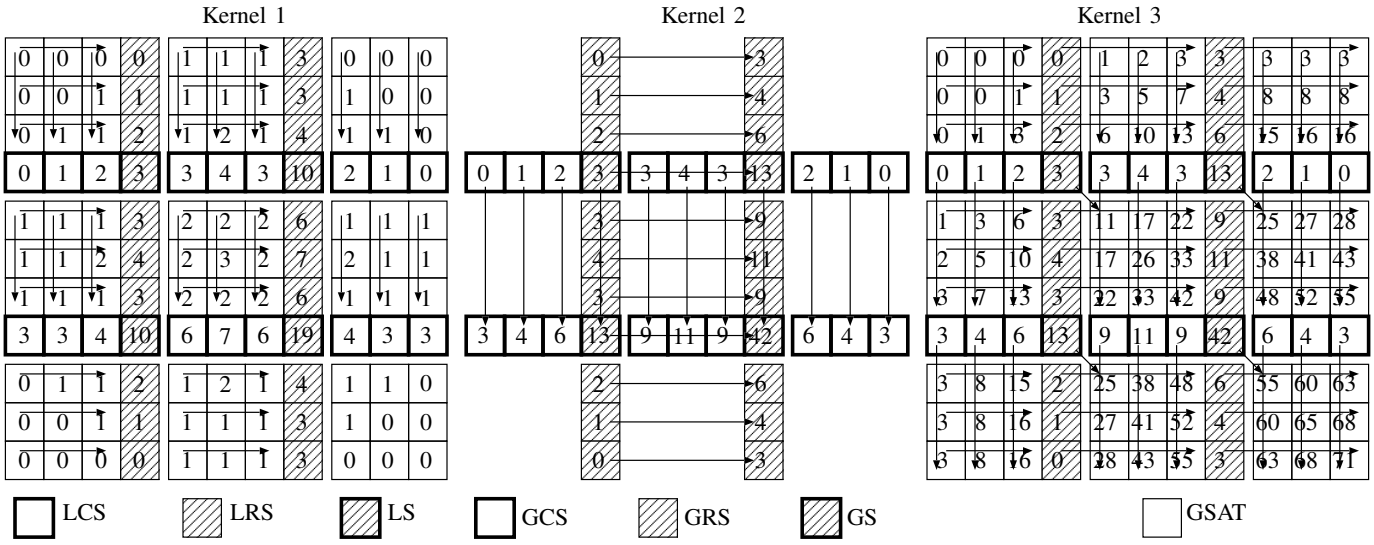


Figure 6. Tile-based 2R1W SAT algorithm for $n = 9$ and $W = 3$

top-left corner of $T(I, J)$ in the shared memory, respectively. By executing the following steps, $\mathrm{GSAT}(I, J)$ is computed and copied to the global memory. Note that, we need to write $\mathrm{GRS}(I, J)$, $\mathrm{GCS}(I, J)$, and $\mathrm{GS}(I, J)$ for Kernel $K+1$. Clearly, $\mathrm{GS}(I, J)$ is the bottom-right corner of $\mathrm{GSAT}(I, J)$. $\mathrm{GRS}(I, J)$ and $\mathrm{GCS}(I, J)$ can be obtained by subtracting adjacent pairs in the rightmost column and the bottom row of $\mathrm{GSAT}(I, J)$, respectively. Similarly to 2R1W algorithm, we write $\mathrm{GRS}(I, J)$, $\mathrm{GCS}(I, J)$, and $\mathrm{GS}(I, J)$ in the global memory.

We can accelerate 1R1W algorithm by a hybrid algorithm with 2R1W algorithm. In 1R1W algorithm, Kernel $K$ with small or large $K$ uses fewer CUDA blocks and so parallelism is quite low. Hence, we use 2R1W SAT algorithm for tiles processed by Kernel $K$ with small or large $K$. We call it $(1+r)$R1W SAT algorithm, where $r$ $(0 < r < 1)$ is a parameter. As illustrated in Figure 8, we partition tiles into three groups such that (A) $T(I, J)$s with $I + J < \sqrt{r}\frac{n}{W}$, (B) $T(I, J)$s with $\sqrt{r}\frac{n}{W} \leq I + J < (2 - \sqrt{r})\frac{n}{W}$, and (C) $T(I, J)$s with $(2 - \sqrt{r})\frac{n}{W} - 1 < I + J$. In Figure 8, a parameter $r = 0.25$

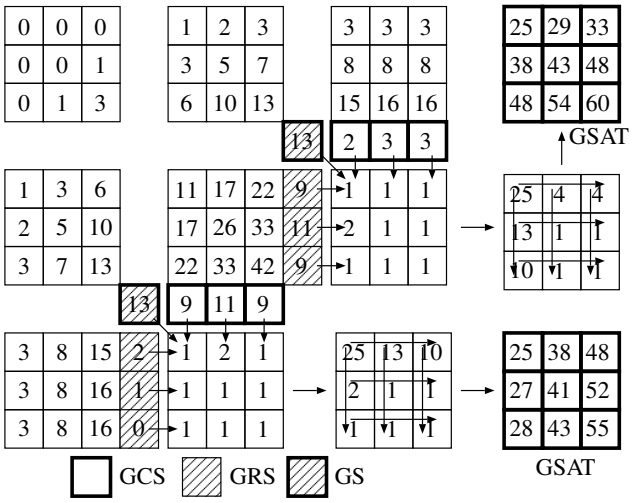


Figure 7. Computation performed by Kernel 3 of Tile-based 1R1W SAT algorithm.

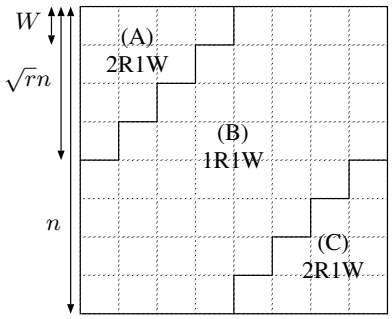


Figure 8. $(1+r)$R1W algorithm for $r = 0.25$

is applied. The hybrid algorithm uses 2R1W algorithm for (A) and (C), and 1R1W algorithm is executed for (B). Since 2R1W algorithm is executed $\sqrt{r}\frac{n}{W}(\sqrt{r}+1)\frac{n}{W} \approx r\frac{n^2}{W^2}$ tiles out of $\frac{n^2}{W^2}$ tiles, it totally performs $(1+r)n^2 + O(\frac{n^2}{W})$ read and $n^2 + O(\frac{n^2}{W})$ write operations. We can choose the best value of $r$ that minimizes the running time.

### C. 1R1W-SKSS algorithm [15]

Let column $J$ ($0 \leq J \leq \frac{n}{W} - 1$) denote a column of $\frac{n}{W}$ tiles $T(0, J)$, $T(1, J)$, ..., $T(\frac{n}{W} - 1, J)$. We use $\frac{n}{W}$ CUDA blocks, each of which is assigned to a column. This assignment is done by a global counter $c$ in the global memory initialized by zero as follows. A CUDA kernel invokes $\frac{n}{W}$ CUDA blocks and the first thread of every CUDA block performs $J \leftarrow$atomicAdd(&$c$,1), which exclusively increments $c$ and returns the value of $c$ before addition. Thus, atomicAdd(&c,1) returns 0, 1, 2, ... in turn and no two threads receive the same return value. A CUDA block with the first thread receiving return value $J$ is assigned to column $J$ if $J < \frac{n}{W}$. It terminates if $J \geq \frac{n}{W}$. After the GSAT of every tile in column $J$ is completed, it executes $J \leftarrow$atomicAdd(&$c$,1) again and is assigned to column $J$ provided that $J < \frac{n}{W}$ is satisfied. Otherwise, it terminates. The same procedure is repeated until return value $J$ satisfies $J \geq \frac{n}{W}$.

The algorithm uses an array $R$ of flags initialized by zero such that, after GRS$(I, J)$ is written in the global memory 1 is written in $R[I][J]$. Each CUDA block assigned to column $J$ computes the GSAT of every tile in column $J$ from top to bottom. A CUDA block uses $T(I, J)$, GRS$(I, J-1)$ and GCP$(I-1, J)$ to compute GSAT$(I, J)$. First, it copies $T(I, J)$ in the global memory to the shared memory. After that, it repeats reading $R[I][J-1]$ until it stores 1. After $R[I][J-1]$ stores 1, GRS$(I, J-1)$ is read and added to the leftmost column of $T(I, J)$. The row-wise prefix-sums are computed and the rightmost column, which is equal to GRS$(I, J)$, is written in the global memory and 1 is written in $R[I][J]$. Since GCP$(I-1, J)$ is the bottom row of GSAT$(I-1, J)$, which the CUDA block has computed, no global memory access is necessary to get GCP$(I-1, J)$. GCP$(I-1, J)$ is added to the topmost row of the resulting matrix of row-wise prefix-sums and the column-wise prefix-sums are computed. The resulting matrix is GSAT$(I, J)$ and it is written in the global memory.

## IV. Our 1R1W-SKSS-LB algorithm

We show our 1R1W-SKSS-LB algorithm, which uses single kernel soft synchronization (SKSS) technique and look back (LB) technique. Similarly to 1R1W and 1R1W-SKSS algorithm, a CUDA block is assigned to $T(I, J)$ and computes GSAT$(I, J)$.

We assign serial numbers to tiles in diagonal-major order as illustrated in Figure 9. More specifically, a serial number $\frac{(I+J)(I+J+1)}{2} + I$ is assigned to each tile $T(I, J)$ ($0 \leq I, J \leq \frac{n^2}{W^2} - 1$). Figure 9 illustrates serial numbers assigned to tiles when $\frac{n}{W} = 5$. All CUDA blocks are assigned to tiles in the order of serial numbers. Similarly to 1R1W-SKSS algorithm, we use atomicAdd function with a global counter $c$ initialized by zero in the global memory. In 1R1W-SKSS-LB, each CUDA block is assigned to a tile by the return value if it is less than $\frac{n^2}{W^2}$.

We will show the outline of GSAT$(I, J)$ computation by a CUDA block with multiple threads.

**Step 1**: $T(I, J)$ is copied from the global memory to the shared memory in the diagonal arrangement.

**Step 2.A.1**: LRS$(I, J)$ is computed and written in the global memory.

**Step 2.A.2**: GRS$(I, J-1)$ is computed.

**Step 2.A.3**: GRS$(I, J)$ is computed and written in the global memory.

**Step 2.B.1**: LCS$(I, J)$ is computed and written in the global memory.

**Step 2.B.2**: GCS$(I-1, J)$ is computed.

**Step 2.B.3**: GCS$(I, J)$ is computed and written in the global

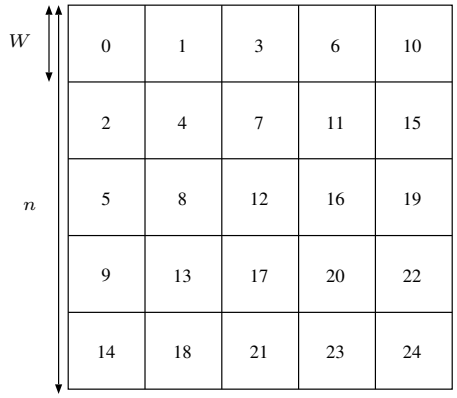


Figure 9. Serial numbers assigned to tiles

memory.

**Step 3.1**: $\mathrm{GLS}(I, J)$ is computed and written in the global memory.

**Step 3.2**: $\mathrm{GS}(I-1, J-1)$ is computed.

**Step 3.3**: $\mathrm{GS}(I, J)$ is computed and written in the global memory.

**Step 4**: $\mathrm{GSAT}(I, J)$ is computed and written in the global memory.

Note that, $\mathrm{LRS}(I, J)$, $\mathrm{GRS}(I, J)$, $\mathrm{LCS}(I, J)$, $\mathrm{GCS}(I, J)$, $\mathrm{GLS}(I, J)$, and $\mathrm{GS}(I, J)$ are written in the global memory, because these values will be used by the other CUDA blocks. Thus, we use two 8-bit integers $R$, and $C$ initialized by zero in the global memory to store the status of computation of these steps for $T(I, J)$ as follows:

- $R$ Integers 1, 2, 3, and 4 are written in $R$ after $\mathrm{LRS}(I, J)$, $\mathrm{GRS}(I, J)$, $\mathrm{GLS}(I, J)$ and $\mathrm{GS}(I, J)$ are written in the global memory, respectively.
- $C$ Integers 1 and 2 are written in $C$ after $\mathrm{LCS}(I, J)$ and $\mathrm{GCS}(I, J)$ are written in the global memory, respectively.

Since we have $\frac{n^2}{W^2}$ tiles, $2\frac{n^2}{W^2}$ 8-bit integers are used in total. For later reference, let $R[I][J]$ and $C[I][J]$ denote $R$ and $C$ for tile $T(I, J)$, respectively. By reading these two 8-bit integers, the other CUDA blocks can learn the status of computation for $\mathrm{GSAT}(I, J)$.

Next, we will show how each step is performed by a CUDA block. Steps 1, 2.A.1, and 2.B.1 can be done in the same way as shared memory column-wise/row-wise sum algorithm. Steps 2.A.2 and 2.B.2 use the look back technique. In Step 2.A.2, the values of $R$ are read leftwards and $\mathrm{GRS}(I, J-1)$ is obtained by computing the pairwise sums of $\mathrm{GRS}(I, J')$, $\mathrm{LRS}(I, J'+1)$, $\mathrm{LRS}(I, J'+2)$, ..., $\mathrm{LRS}(I, J-1)$ for some $J'$ $(0 \leq J' \leq J-1)$ as illustrated in Figure 10. For this purpose, $R[I][J-1]$ is repeatedly read until it becomes 1 or larger. If $R[I][J-1] \geq 2$, then Step 2.A.2 is completed by simply reading $\mathrm{GRS}(I, J-1)$. If $R[I][J-1] = 1$, then $\mathrm{LRS}(I, J-1)$ is read and Step 2.A.2 is continued leftwards. Similarly, $R[I][J-2]$ is repeatedly read until it becomes 1 or larger. If $R[I][J-2] \geq 2$, then $\mathrm{GRS}(I, J-2)$ is read and the pairwise sums of $\mathrm{GRS}(I, J-2)$ and $\mathrm{LRS}(I, J-1)$ are computed and written in the global memory. If $R[I][J-2] = 1$, then the values of $\mathrm{LRS}(I, J-2)$ are read and the pairwise sums of $\mathrm{LRS}(I, J-2)$ and $\mathrm{LRS}(I, J-1)$ are computed. After that, $R[I][J-3]$ is read similarly to obtain $\mathrm{LRS}(I, J-3)$ or $\mathrm{GRS}(I, J-3)$. The same procedure is repeated leftwards until $R[I][J'] \geq 2$ and $\mathrm{GRS}(I, J')$ is obtained for some $J'$ $(0 \leq J' \leq J-1)$. When $\mathrm{GRS}(I, J')$ is obtained, the pairwise sums of $\mathrm{GRS}(I, J')$ and $\mathrm{LRS}(I, J'+1) + \mathrm{LRS}(I, J'+2) + \ldots + \mathrm{LRS}(I, J-1)$ are computed. We can see that the resulting pairwise sums are equal to $\mathrm{GRS}(I, J-1)$ from Figure 10. This completes Step 2.A.2. After the values of $\mathrm{GRS}(I, J-1)$ are obtained, the pairwise sums $\mathrm{GRS}(I, J-1) + \mathrm{LRS}(I, J)$, which are equal to $\mathrm{GRS}(I, J)$, are computed and written in the global memory. This completes Step 2.A.3. Similarly, Steps 2.B.2, and 2.B.3 can be done at the same time.

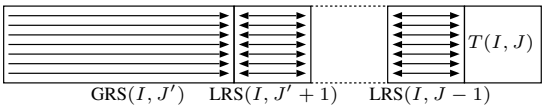


Figure 10. Illustrating the computation of $\mathrm{GRS}(I, J-1)$ by look back technique in Step 2.A.2

In Step 3.1, the pairwise sums $\mathrm{GRS}(I, J-1) + \mathrm{GCS}(I-1, J) + \mathrm{LRS}(I, J)$ are computed by $W$ threads in an obvious way, and the sum of the pairwise sums is computed by warp prefix-sum algorithm and is written in the global memory. From Figure 11, we can see that the sum of the pairwise sums is equal to $\mathrm{GLS}(I, J)$. Step 3.2 can be done by the look back technique similarly to Step 2.A.2. The idea is to compute the sums of $\mathrm{GS}(I-k, J-k)$, $\mathrm{GLS}(I-k+1, J-k+1)$, $\mathrm{GLS}(I-k+2, J-k+2)$, ..., $\mathrm{GLS}(I-1, J-1)$ for some $k$ $(1 \leq k \leq \min(I, J))$, which is equal to $\mathrm{GS}(I-1, J-1)$. By reading the status $R(I-k, J-k)$ for $k = 1, 2, \ldots, \min(I, J)$, we can obtain these values in the same way as Step 2.A.2 by the look back technique. The look back is repeated until $\mathrm{GLS}(I-k, J-k)$ is obtained for some $k$ $(1 \leq k \leq \min(I, J))$. When $\mathrm{GS}(I-k, J-k)$ is read, the values of $\mathrm{GLS}(I-k+1, J-k+1) + \mathrm{GLS}(I-k+2, J-k+2) + \ldots + \mathrm{GLS}(I-1, J-1)$ have been already computed. Thus, they are added and the sum written in the global memory. From Figure 11, we can see that the written sum is $GS(I-1, J-1)$. In Step 3.3, the sum of $\mathrm{GS}(I-1, J-1)$ and $\mathrm{GLS}(I, J)$, which is equal to $\mathrm{GS}(I, J)$, is computed and written in the global memory.

In Step 4, $\mathrm{GSAT}(I, J)$ is computed using $\mathrm{GRS}(I, J-1)$, $\mathrm{GCS}(I-1, J)$, and $\mathrm{GS}(I-1, J-1)$ and written in the global memory in the same way as 1R1W algorithm. Note that a CUDA block must execute barrier synchronization __syncthreads() when Steps 1, 2, and 3 are completed. Thus,

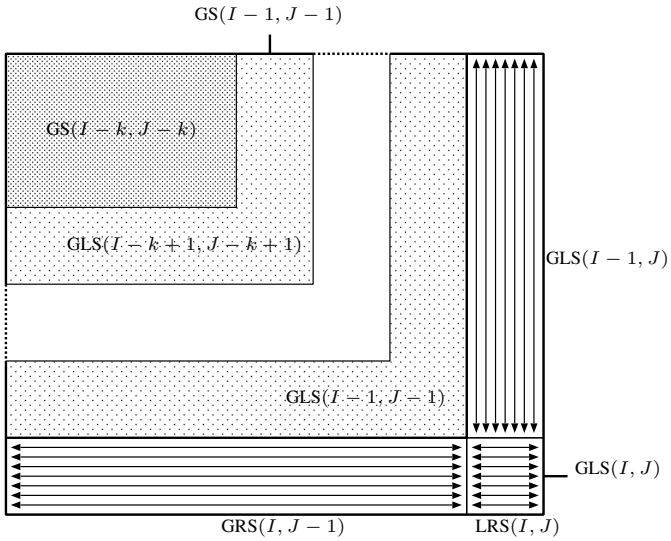


Figure 11. Illustrating the computation of GLS$(I, J)$ in Step 2.1 and GS$(I-1, J-1)$ in Step 2.2

only three barrier synchronization operations are performed.

## V. Experimental results

We have implemented our 1R1W-SKSS-LB algorithm using CUDA and evaluated the performance using NVIDIA TITAN V GPU, which has 80 streaming multiprocessors with 64 cores each. We have also implemented previously published SAT algorithms, 2R2W, 2R2W-optimal, 2R1W, 1R1W, $(1+r)$R1W, and 1R1W-SKSS algorithms, and evaluated the performance for comparison. Since the capacity of global memory is 12Gbytes, we evaluated the performance for 4-byte float matrices of up to $32K \times 32K$, which occupies 4Gbytes. Since each CUDA block running on NVIDIA TITAN V can have up to 96Kbytes shared memory, we use parameter $W = 32$, 64, and 128. When $W = 128$, 4-byte float matrices of size $128 \times 128$ needs 64Kbytes. For tile-based SAT algorithms, we use CUDA blocks with 1024 threads each to maximize parallelism.

Table III shows the running time in milliseconds of these SAT algorithms for matrices of sizes from $256 \times 256$ to $32K \times 32K$. It also shows the time for matrix duplication using cudaMemcpy() function call, which copies the specified bytes between two memory area in the global memory. Thus, the running time of any SAT algorithm cannot be faster than that of cudaMemcpy(). In the table, the best running time obtained by thee parameters $W = 32$, 64, and 128 for each algorithm and matrix size is highlighted. Also, the overhead in percent is computed for the best running time with respect to the matrix duplication. In other words, the overhead of a tile-based SAT algorithm is

$$(\min(T_{32}, T_{64}, T_{128}) - D)/D \times 100$$

where $T_W$ is the running time for parameter $W$ and $D$ is the matrix duplication time. From the table, we can see that our 1R1W-SKSS-LB algorithm is the fastest and the lowest overhead than previously published SAT algorithms for all sizes of matrices. Our 1R1W-SKSS-LB algorithm is also faster than 1R1W-SKSS algorithm, since look back technique is performed efficiently.

Since the row-wise prefix-sum computation in 2R2W algorithm performs stride access to the global memory the running time of 2R2W algorithm is quite large. On the other hand, 2R2W-optimal algorithm performs no stride memory access and uses much more threads, it runs much faster than 2R2W algorithm. However, it performs at least 2 read and 2 write operations to the global memory, the overhead cannot be smaller than 100%. Actually, from the table, we can see that the overhead is larger than 100%, but it is very close to 100% for large matrices. Thus, we can say that 2R2W-optimal algorithm is optimal under the condition that the SAT must be computed by the column-wise and row-wise prefix-sums computation.

To attain the overhead below 100%, tile-based SAT algorithms that use the shared memory as a cache to store tiles of an input matrix are necessary. The overhead of 2R1W algorithm is more than 50% because it performs at least 3 memory access operations per element, while matrix duplication performs 2 memory access operations per element.

No tile-based algorithm achieves overhead less than 100% for matrices no larger than $512 \times 512$ due to low parallelism. For example, if a tile-based SAT algorithm is executed for a $256 \times 256$ input matrix with a $128 \times 128$ tiles, only 4 tiles are processed by 4 CUDA blocks with 4096 threads totally. Since NVIDIA TITAN V has 80 streaming multiprocessors, at least 80 CUDA blocks should be invoked by kernels to fully utilize hardware resources. Hence, the overhead is large when the input matrix is small.

## VI. Conclusion

We have presented a very efficient parallel algorithm that computes the summed area table (SAT) of a matrix running on a GPU. We have also implemented previously published SAT algorithms and evaluated the performance on NVIDIA TITAN V GPU. Our experimental results show that our SAT algorithm runs faster than known SAT algorithms for all matrices of sizes from $256 \times 256$ to $32K \times 32K$. Further, the overhead of our parallel SAT algorithm over matrix duplication can be only 5.7%, so it is practically optimal.

## References

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

[2] H. Kouge, T. Honda, T. Fujita, Y. Ito, K. Nakano, and J. L. Bordim, "Accelerating digital halftoning using the local exhaustive search on the GPU," *Concurrency and Computation: Practice and Experience*, vol. 29, no. 2, 2017.

Table III
THE RUNNING TIME IN MILLISECONDS AND THE OVERHEAD IN PERCENT OF PARALLEL SAT ALGORITHMS FOR MATRICES OF SIZES FROM $256 \times 256$ TO $32K \times 32K$

| Parallel algorithms | $W^2$ | $256^2$ | $512^2$ | $1K^2$ | $2K^2$ | $4K^2$ | $8K^2$ | $16K^2$ | $32K^2$ |
|---|---|---|---|---|---|---|---|---|---|
| matrix duplication by cudaMemcpy | - | 0.00512 | 0.00614 | 0.0165 | 0.0645 | 0.237 | 0.927 | 3.69 | 14.7 |
| 2R2W algorithm | - | 0.0901 | 0.167 | 0.338 | 1.01 | 2.57 | 8.47 | 24.4 | 87.1 |
| | overhead | 1659.8% | 2619.9% | 1948.5% | 1465.9% | 984.4% | 813.7% | 561.2% | 492.5% |
| 2R2W-optimal algorithm | - | 0.0224 | 0.0224 | 0.0467 | 0.136 | 0.478 | 1.86 | 7.52 | 30.0 |
| | overhead | 337.5% | 264.8% | 183.0% | 110.9% | 101.7% | 100.6% | 103.8% | 104.1% |
| 2R1W algorithm [13] | $32^2$ | 0.0191 | 0.0272 | 0.0669 | 0.182 | 0.577 | 2.04 | 7.88 | 30.9 |
| | $64^2$ | **0.0161** | **0.0191** | **0.0489** | **0.141** | **0.434** | **1.53** | **5.81** | **22.8** |
| | $128^2$ | 0.0271 | 0.0284 | **0.0489** | 0.155 | 0.459 | 1.65 | 6.35 | 25.1 |
| | overhead | 214.5% | 211.1% | 196.4% | 118.6% | 83.1% | 65.0% | 57.5% | 55.1% |
| 1R1W algorithm [14] | $32^2$ | 0.059 | 0.108 | 0.249 | 0.524 | 1.13 | 2.97 | 8.47 | 27.9 |
| | $64^2$ | 0.0363 | 0.0829 | **0.194** | **0.402** | **0.866** | 2.03 | 6.32 | 21.7 |
| | $128^2$ | **0.0301** | **0.0653** | 0.195 | 0.417 | 0.890 | **2.02** | **6.23** | **21.0** |
| | overhead | 487.9% | 963.5% | 1075.8% | 523.%3 | 265.4% | 117.%9 | 68.8% | 42.9% |
| $(1+r)$R1W algorithm [14] | $32^2$ | **0.0453** | **0.0555** | 0.118 | 0.302 | 0.862 | 2.45 | 7.47 | 25.4 |
| | $64^2$ | 0.0464 | 0.0582 | **0.0809** | 0.197 | 0.539 | 1.67 | 5.95 | 21.2 |
| | $128^2$ | 0.0638 | 0.0709 | 0.0871 | **0.188** | **0.517** | **1.60** | **5.81** | **20.6** |
| | overhead | 784.8% | 803.9% | 390.3% | 191.5% | 118.1% | 72.6% | 57.5% | 40.1% |
| 1R1W-SKSS algorithm [15] | $32^2$ | **0.0298** | 0.0476 | 0.0692 | 0.128 | 0.387 | 1.20 | 4.55 | 17.5 |
| | $64^2$ | **0.0298** | **0.0356** | **0.0606** | 0.136 | 0.330 | 1.15 | 4.26 | 16.4 |
| | $128^2$ | 0.0409 | 0.0398 | 0.0753 | **0.124** | **0.319** | **1.14** | **4.18** | **16.2** |
| | overhead | 482.0% | 479.8% | 267.3% | 92.2% | 34.6% | 23.0% | 13.3% | 10.2% |
| Our 1R1W-SKSS-LB algorithm | $32^2$ | 0.0146 | 0.0209 | 0.0444 | 0.147 | 0.542 | 2.16 | 8.64 | 37.5 |
| | $64^2$ | **0.0126** | 0.0156 | 0.0266 | 0.0790 | 0.266 | 1.06 | 4.28 | 17.4 |
| | $128^2$ | 0.0132 | **0.0136** | **0.0208** | **0.0753** | **0.258** | **0.980** | **3.92** | **15.8** |
| | overhead | **146.1%** | **121.5%** | **26.%1** | **16.7%** | **8.9%** | **5.7%** | **6.2%** | **7.5%** |


[3] N. Corporation, "NVIDIA CUDA PC Programing Guide Version 9.0," Jan. 2018.

[4] 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.

[5] T. Fujita, K. Nakano, and Y. Ito, "Bitwise parallel bulk computation on the GPU, with application to the CKY parsing for context-free grammars," in *Proc. of International Parallel and Distributed Processing Symposium Workshops*, May 2016, pp. 589–598.

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

[7] 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.

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

[9] 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.

[10] D. Merrill and M. Garland, "Single-pass parallel prefix scan with decoupled look-back," NVIDIA, Tech. Rep. NVR-2016-002, March 2016.

[11] D. Merrill, "CUB : A library of warp-wide, block-wide, and device-wide gpu parallel primitives," https://nvlabs.github.io/cub/, 2017.

[12] H. Tokura, T. Fujita, K. Nakano, Y. Ito, and J. L. Bordim, "Almost optimal column-wise prefix-sum computation on the GPU," in *The Journal of Supercomputing*, 2018.

[13] 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.

[14] A. Kasagi, K. Nakano, and Y. Ito, "Parallel algorithms for the summed area table on the asynchronous hierarchical memory machine, with GPU implementations," in *Proc. of International Conference on Parallel Processing (ICPP)*, Sept. 2014, pp. 251–250.

[15] S. Funasaka, K. Nakano, and Y. Ito, "Single kernel soft synchronization technique for task arrays on CUDA-enabled GPUs, with applications," in *Proc. International Symposium on Networking and Computing*, Nov. 2017, pp. pp.11–20.

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

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

[18] M. Harris, S. Sengupta, and J. D. Owens, "Chapter 39. parallel prefix sum (scan) with CUDA," in *GPU Gems 3*. Addison-Wesley, 2007.