Efficient Algorithms for the Summed Area Tables Primitive on GPUs

Peng Chen[1,3], Mohamed Wahib[2], Shinichiro Takizawa[3], Ryousei Takano[2], Satoshi Matsuoka[1,4]
[1] *Tokyo Institute of Technology, Dept. of Mathematical and Computing Science, Tokyo, Japan*
[2] *National Institute of Advanced Industrial Science and Technology, Tokyo, Japan*
[3] *AIST-Tokyo Tech Real World Big-Data Computation Open Innovation Laboratory, National Institute of Advanced Industrial Science and Technology*
[4] *RIKEN Center for Computational Science, Hyogo, Japan*
chen.p.aa@m.titech.ac.jp, {mohamed.attia, shinichiro.takizawa, takano-ryousei}@aist.go.jp, matsu@acm.org

***Abstract*—Two-dimensional Summed Area Tables (SAT) is a fundamental primitive used in image processing and machine learning applications. We present a collection of optimization methods for computing SAT on CUDA-enabled GPUs. Conventional approaches rely on computing the prefix sum in one dimension in parallel, transposing the matrix, then computing the prefix sum for the other dimension in parallel. Additionally, conventional methods use the scratchpad memory as cache. We propose a collection of algorithms that are scalable with respect to problem size. We use the register cache technique instead of the scratchpad memory and also employ a naive serial scan on the thread level for computing the prefix sum for one of the dimensions. Using a novel transpose-in-registers method we increase the inter-thread parallelism and outperform conventional SAT implementations. In addition, we significantly reduce both the communication between threads and the number of arithmetic instructions. On an Nvidia Pascal P100 GPU and Volta V100, our evaluations demonstrate that our implementations outperform state of the art libraries and yield up to 2.3x and 3.2x speedup over OpenCV and Nvidia NPP libraries, respectively.**

***Index Terms*—GPU, CUDA, Summed Area Tables**

I. Introduction

Summed Area Tables (SAT)[1], is an intermediate lookup table, which stores the sum of all elements in an input matrix between the left-top corner and the current location, and allows fast computation of rectangular summation at a constant time regardless of the area of subregions. A wide range of different applications proposed and utilized algorithms for efficient SAT. For instance, it is crucial to design efficient SAT algorithms in order to accelerate the computation pipeline in a vast range of real-time computer vision [1]–[11] and Deep Learning workloads [12]–[14]. The importance of SAT makes it imperative to design SAT algorithms that would scale vertically (i.e. on one node), as well as horizontally (i.e. on the entire system). In this paper, we focus on the acceleration of SAT at the node level using a method that is scalable, with respect to problem size.

In this paragraph, we summarize some of the high-performance computer vision and machine learning applications that are based on the SAT in order to illustrate the importance of accelerating the computation of SAT. Crow et al. [1] innovated the idea of summed area table to implement an algorithm for general texture filtering in the computer graphics field. Hensley et al. [6] applied SAT to dynamically compute image-based lighting from high-dynamic-range maps. Lewis et al. [15] successfully implemented a fast template matching algorithm based on integral imaging. Viola and Jones [2] demonstrated a real-time face detection cascade network by speeding-up the haar-liked feature computation. For visual tracking, Han [3] et al. applied an integral imaging technology to speed-up the local feature computation. Bay et al. [5] implemented an efficient and accurate object recognition algorithm based on SAT. The authors in [16] used integral imaging to speed-up a HOG-LBP human detector and achieved outstanding performance. Nehab et al. [9] presented an efficient recursive filtering implementation using summed area tables.

In recent years, GPUs have played an important role in high-performance computing by providing superior computing capability over conventional CPUs. For some compute-intensive algorithms, GPUs can offer an order of magnitude improvement in performance compared to CPUs. However, to achieve high performance with GPUs, it is essential to optimize algorithms that meet the requirements and limitations of the complex GPU architecture. Typically there are two classes of SAT algorithms on GPUs. One is the scan-scan algorithm (and its variations). The other is the scan-transpose-scan algorithm, which adopts three or four steps to compute the SAT. The algorithms from the second class require a regular coalesced access pattern of the device global memory in one of the steps, and an expensive matrix transpose operation is applied in another step (e.g. [10], [17]). *Register cache* is a promising, yet complex technique, to improve the performance of memory-bound GPU applications [18]–[20]. In this work, we implement a collection of efficient algorithms that compute the SAT by register cache. More specifically, we manually cache data by register files. Also, we propose a novel Block-Register-Local-Transpose methodology (BRLT) and an efficient parallelized serial scan algorithm, which is more efficient than conventional approaches, due to the reduction in inter-thread communication and arithmetic instructions. The use of register cache to compute partial prefix sums, an implicit in-register transpose and a serialized computation of the prefix sum in one dimension differentiate the algorithms proposed in

[1]SAT is a 2D scan operation also called Integral Image in image processing field.

this paper from the prevalent SAT algorithms on GPUs. On the contrary, the existing methods typically rely on scratchpad memory and transposition of the matrix in-between the prefix sum in each dimension.

The bottleneck of The SAT computation is data movement. Efficiently accessing global memory in a coalesced pattern is critical to the algorithm's performance. Additionally, it is essential to design algorithms that reduce the global memory access. To reduce the data movement, we rely on the register cache method since registers in GPUs provide more advantages than scratchpad memory: higher throughput, higher capacity, and lower latency. However, it is a challenge to effectively use the registers while avoiding contention. Moreover, the technology for communicating registers between threads, known as *shuffle* instruction, works only within a single warp (a set of threads executed together in CUDA). Hence, we adapt our algorithm to do the scan at the warp level while avoiding intra-warp communication. In this work, we propose three SAT computation algorithms. Using Nvidia Pascal P100 and Volta V100 GPUs, the evaluation result shows that our fastest algorithm outperforms the state-of-the-art libraries, OpenCV and Nvidia NPP, in addition to being generic to all kinds of data types.

The contributions in this paper are as follows:

- We propose a novel Block-Register-Local-Transpose (BRLT) algorithm. To our knowledge, this is the first algorithm to apply the register cache method to compute SAT.
- We propose another efficient SAT computation algorithm that relies on an optimized intra-thread serial scan method to reduce the inter-thread communication.
- We achieved outstanding performance on computing SAT on Tesla P100 and V100, with up to 2.3× and 3.2× speedup over the production-level OpenCV and Nvidia NPP libraries.

The rest of this paper is organized as follows. In Section II we introduce related work and our motivation. Section III presents the related algorithms on scan and SAT. In Section IV we illustrate the proposed SAT algorithms. In Section V we discuss a performance model to reason about the efficiency of the proposed algorithms. Section VI describes the evaluation results. Finally, Section VII concludes.

The source code of all algorithms in this paper is publicly available at https://github.com/pengdada/SAT

II. Related Work

SAT computation has been widely researched in many fields, e.g. image processing, computer graphics, and machine learning. To achieve an efficient object detector, previous works employed Field Programmable Gate Array (FPGA) to implement SAT computation [21]–[27]. On the parallel SIMD (Single Instruction Multiple Data) architectures, the authors in [28]–[30] optimized the SAT algorithm by parallel instructions. GPUs are enjoying significant popularity in HPC [31]–[33] and are widely adopted to accelerate the SAT computation, such as in [17], [34]–[38]. Several state-of-the-art

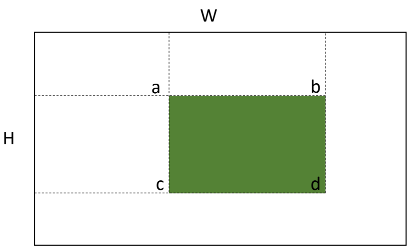


Fig. 1: Summed Area Table illustration. The SAT size is $H \times W$. The values a, b, c, and d denote the four values at the corners of shade rectangle.

libraries also provide SAT primitive programming interfaces, such as Intel IPP [39], Nvidia NPP, Matlab [40], OpenCV [41], OpenCLIPP [42] ...etc. Typically the GPU implementations employ the device scratchpad memory as fast cache to improve data reuse and use a parallel warp-scan algorithm (Sec. III-C2) to compute the prefix sum efficiently [17], [34]–[38]. To the best of our knowledge, scratchpad memory is not as efficient as register files and requires careful software designation to avoid bank conflicts [18]–[20]. Finally, existing research proved that on Nvidia GPUs, shuffle-based one-dimensional scan operator performs better than scratchpad memory solutions [43]–[46].

III. Background

SAT computation performs the scan[2] algorithm twice on an input matrix. First, we perform rows (namely horizontal) scan on the input matrix, then we perform a scan on the columns (namely vertical) of the result matrix of the first step. Because different rows (and columns) are independent of each other, the scan operation of different rows or columns can be computed in parallel.

A. Computing SAT

Summed Area Tables is an effective and quick algorithm used to compute the summation of values over an arbitrary rectangular region. Formally, the inclusive SAT is defined as

$$J(x,y) = \sum_{j=0}^{y} \sum_{i=0}^{x} I(i,j) \tag{1}$$

while the exclusive SAT is defined as

$$J(x,y) = \begin{cases} 0 & \text{x=0 or y=0,} \\ \sum_{j=0}^{y-1} \sum_{i=0}^{x-1} I(i,j) & \text{others} \end{cases} \tag{2}$$

where I is an input 2D matrix of size $H \times W$. J is the output SAT matrix with the same size as I. A point in the 2D space is indicated as (x, y), where $0 \leq x < W$ and $0 \leq y < H$. The inclusive and exclusive SAT can be transformed easily to one another. In this paper, we only consider the inclusive SAT computation; the more general case of SAT. Alg. 1 shows the naive serial inclusive SAT computation. For the input matrix of size $H \times W$, $2 * H * W$ addition operations are needed.

[2]Scan is also known as all-prefix-sum: the sums of prefixes (running totals) of an input sequence

Algorithm 1 Naive Serial Inclusive SAT Algorithm

```
Input:  2D matrix $I$ of size $H \times W$
Output: 2D matrix $J$ of size $H \times W$
 1: $J[0][0] \leftarrow I[0][0]$
 2: for $i = 1; i < W; i++$ do
 3:     $J[0][i] \leftarrow I[0][i] + J[0][i-1]$
 4: end for
 5: for $j = 1; j < H; j++$ do
 6:     $sum \leftarrow 0$
 7:     for $i = 0; i < W; i++$ do
 8:         $sum \leftarrow sum + I[j][i]$
 9:         $J[j][i] \leftarrow J[j-1][i] + sum$
10:     end for
11: end for
```

As Fig. 1 shows, in the original matrix, the summation of the data in the shaded rectangle can be calculated efficiently as $a+d-b-c$, which needs only four accesses of the SAT lookup table and three arithmetic operations to obtain the summation of a specified rectangular area in the original 2D matrix.

B. Parallel Computation With CUDA

We briefly introduce the fundamental concepts of Nvidia's CUDA architecture and programming model in this section, more details can be found at [47].

1) CUDA Architecture: CUDA (Compute Unified Device Architecture) is built on a large array of SMs (multi-threaded streaming processors). Massive thread-level parallelism is abstracted into a hierarchy of threads running in a SIMT (single instruction multi-thread) fashion. The threads in CUDA are managed as warps, blocks, and grids. In comparison to latency-optimized CPUs, GPUs are throughput-optimized processors.

In all generations of CUDA architecture, multi-threads execute in groups of threads, called warp. We call the thread count in a single warp $WarpSize$[3]. Each thread in a warp is assigned an ID, called $laneId$, which ranges from 0 to 31.

2) CUDA Memory Hierarchy: GPUs supports many memory types: global, shared and registers, etc. It is essential to consider the memory hierarchy when designing an algorithm. Global memory is the largest but also the slowest on-chip memory, and only in a coalesced access pattern (i.e. unit-strided contiguous memory access), can the global memory achieve throughput that approaches the peak memory bandwidth. Shared memory is faster than global memory but limited in size. Shared memory is a fast on-chip memory sharing the same space with the L1 cache. Its scope is restricted to a single CUDA block. Like banks in Dynamic Random Access Memory (DRAM) modules, shared memory is similarly divided into banks (32 banks in the latest GPUs). Each bank may process only one Read/Write request at a time. Bank conflicts result in lower throughput for CUDA kernels. Registers are the fastest but most difficult to be used on-chip memory since their scope is limited to a single thread. Data in registers could be exchanged within a warp by the *shuffle* instruction.

[3]$WarpSize = 32$, through all generations of Nvidia GPUs

TABLE I: Comparison between shared memory and register files.

Tesla GPU	M40	P100	V100
Shared Memory/SM (KB)	16/32/48	64	≤96
Registers/SM (KB)	256	256	256
SMs	24	56	80

3) Register Cache & Scratchpad Memory: Register cache is an efficient data access mechanism built as a virtual layer on the top of register files. More specifically, the large amount of register memory available in the latest GPUs can be used to store some data arrays. However, the register cache method is complex to use, mainly due to the scope considerations and the restrictions of the SIMT execution model of the threads in a warp.

Table I shows the register files capacity to be more than 2.7 times ($\frac{256KB}{96KB}$) larger than shared memory. Using register files as a cache can greatly improve applications' performance. Especially, since the number of SMs in modern GPU generations has increased, the gap between the capacity of shared memory and registers files is becoming larger. It is important to mention that careful design of algorithms is required to avoid bank conflicts when using shared memory. Otherwise, the performance may drop dramatically.

C. Efficient Scan Algorithms

Scan (or prefix sum), a fundamental primitive, is defined as an associative and binary operator $\bigoplus$, where given a sequence of N elements $V = \{v_1, v_2, v_3......v_N\}$, an inclusive scan on vector V computes a new array $U = \{u_1, u_2, u_3......u_N\}$, where $u_i = \bigoplus_{j=1}^{i} u_j$. On the other hand, exclusive scan computes $u_i = \bigoplus_{j=1}^{i-1} u_j$. In this work, we only focus the discussion on the inclusive SAT computation since it the general case.

1) Serial Scan Algorithm: Given a vector V with N elements, as Fig. 2a shows, we require $N-1$ stages of computation and $N-1$ addition operations to compute the output array. A naive serial scan is performed by a single thread, hence the computing efficiency is very low.

2) Parallel Scan Algorithm: Unlike the serial scan, parallel scan computes prefix sum in parallel. The authors in [44] implemented several kinds of CUDA-optimized scan algorithms, such as Brent-Kung pattern [48] [49], Kogge-Stone pattern [50], Han-Carlson pattern [51], and Ladner-Fischer Pattern (namely LF-scan) [52]. In theory, the LF-scan achieves the highest computing efficiency with log_2N stages and $\frac{Nlog_2N}{2}$ addition operations as Fig. 2c shows, where N indicates the size of the input array. Kogge-Stone scan, shown in Alg. 3, is a widely adopted algorithm. In comparison, it needs log_2N stages to compute and each stage has $S_k = N - 2^k$ addition operations, where k indicates the k^{th} step of computation.

It is worth mentioning that the work proposed in this paper is the first to apply the LF-scan algorithm to SAT computation workload. The serial scan is not as efficient as the parallel scan

Algorithm 2 Serial scan to a vector

```
Input:  1D vector $V$ of size N.
Output: 1D vector $U$ of size N.
1: $U[0] \leftarrow V[0]$
2: for $i = 1$; $i < N$; $i++$ do
3:     $U[i] \leftarrow V[i] + U[i-1]$
4: end for
```

Algorithm 3 Kogge-Stone scan by a CUDA warp

```
Input:  $data$, $laneId$                         ▷ data is a register
Output: prefix sum of $data$.
1: #pragma unroll                                ▷ unroll loop
2: for $i = 1$; $i <= N$; $i* = 2$ do
3:     $val \leftarrow shfl\_up(data, i, WarpSize)$
4:     if $laneId > i$ then
5:         $data \leftarrow data + val$              ▷ in parallel
6:     end if
7: end for
```

Algorithm 4 LF-scan by a CUDA warp

```
Input:  $data$, $laneId$                         ▷ data is a register
Output: prefix sum of $data$.
1: #pragma unroll                                ▷ unroll loop
2: for $i = 1$; $i <= N$; $i* = 2$ do
3:     $val \leftarrow shfl(data,\ i-1,\ i* = 2)$
4:     if $(laneId\ \&\ ((i*2)-1))) \geq i$ then
5:         $data \leftarrow data + val$              ▷ in parallel
6:     end if
7: end for
```

algorithms to a specified one dimension array, yet the serial scan can be utilized perfectly to accelerate two-dimensional SAT computation on GPUs (as will be shown in Section IV-B).

D. Terminology in this work

Throughout this paper, we define T to represent a data type. Regarding data types, 8u is an unsigned char, 32u is an unsigned int32, 32s is an int32, 32f is a float32, 64f is double. T_AT_B means the input data type is T_A and the output data type is T_B. For example, 8u32u means that input data type is 8u, after computation the output data type is 32u. For the size of matrices, W denotes width, and H is height. we use the name *warp-scan* to describe a single warp performing a scan operation.

IV. Proposed SAT Computation by CUDA

Using CUDA, we implement three algorithms to compute SAT efficiently. Our algorithms compute the prefix sum (scan) twice, namely, row scan then column scan. More specifically, given an input 2D matrix I, we first compute the rows prefix sums of I to a temporary matrix L, then compute the columns prefix sums of L to output 2D matrix Y. There are three key technologies to the proposed algorithms.

*1) **Caching Data Using Register Files**:* Global memory access is critical to the performance of CUDA applications. We seek to minimize the data transfer from global memory as much as possible. In CUDA, all threads in a single warp execute simultaneously in a SIMT fashion. Since SAT computation is bound by memory-bandwidth, we use the registers such that each CUDA thread caches multiple words to the registers to increase data reuse. As Alg. 5 line 1 shows, each thread uses 32 registers to cache global memory data that is declared as $T\ data[32]$. Note that registers cannot be used as a dynamic array, i.e. the array size residing in registers must be determined at compile time. In a single warp, a 32×32 register matrix is built up by the threads in the warp. During computation, the data can be fetched from registers without accessing global memory. By carefully designing the register cache, the global memory and caches accesses can be greatly reduced. Since the register's scope is limited to a single thread, we exchange data between threads in a warp using the CUDA shuffle instruction.

*2) **Block-Register-Local-Transpose Method**:* We propose a novel Block-Register-Local-Transpose (BRLT) parallel algorithm to perform transposing a matrix residing in registers. We build a symmetric register matrix where each CUDA thread reserves 32 registers to cache data. Since the $WarpSize$ is fixed as 32, each warp holds a register matrix of size 32×32. For each register matrix, we use an independent shared memory matrix of size 32×33 as a temporary buffer to transpose the register matrix. As shown in Alg. 5, we reserve S blocks in shared memory depending on the number of batches in a single transposition and with consideration to the available capacity of shared memory. For instance, the size of shared memory on Tesla P100 GPUs is 64KB and the maximum block size ($BlockSize = 1024$) is used to achieve the highest occupancy, if the data type T is 32f or 32s. To avoid register pressure we use a block size ($BlockSize = 512$) instead, when T is double. The S can then be computed as $S = 32/sizeof(T)$. This indicates that S varies between 4 and 8 according to the data type of T. It is worth mentioning that we avoid shared memory bank conflicts by setting the $sMem$ stride to the size 33 rather than 32 (shown in line 2 in Alg. 5).

*3) **Partial Sum Scan in a CUDA Block**:* In Fig. 3c, we illustrate how to compute the partial prefix sum of each warp using the shared memory. Basically, there are three steps. Step #1 we store the partial sum (last row of the register matrix) from registers to shared memory according to their $warpId$: in total, we populate a $WarpCount \times WarpSize$ matrix on shared memory. Step #2 we compute the prefix sum of data which is stored in shared memory. Finally, step #3, in each block, each warp fetches the specified values from shared memory and accumulates them to the corresponding registers.

A. Register-based ScanRow-BRLT Method

Register-based ScanRow-BRLT algorithm is the improvement of the scan-transpose-scan SAT algorithm in [17]. The original scan-transpose-scan SAT algorithm saves the row scan result to global memory and executes a transposing operation on global memory explicitly. In contrary, in our Register-based ScanRow-BRLT algorithm we use register cache to perform the parallel warp-scan and apply BRLT to the prefix sum result. In other words, before the row scan data is stored to global memory, the BRLT is performed. Finally, the stored

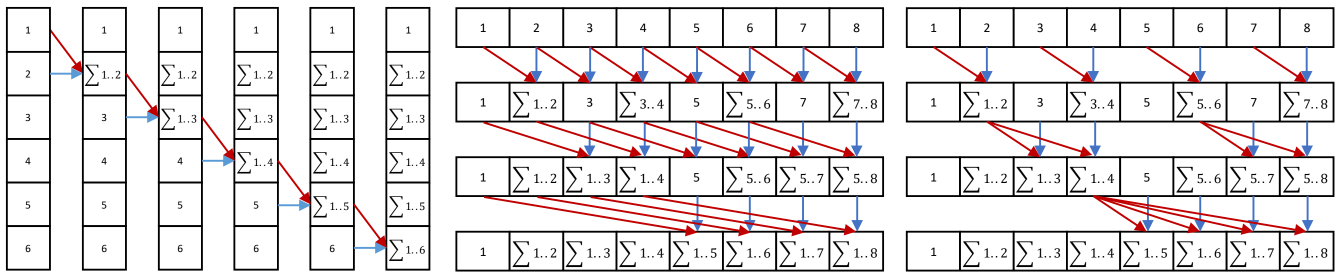

(a) Serial scan for 6 elements (b) Parallel Kogge-Stone scan for 8 elements [50] (c) Parallel LF-scan for 8 elements [52]

Fig. 2: Illustration of scan algorithms. Algorithms in (b) and (c) are widely used.

Algorithm 5 Block-Register-Local-Transpose

```
1: $T\ data[32]$;    ▷ an array of 32 registers
2: $\_\_shared\_\_\ T\ sMem[S][32][33]$    ▷ S is tuned to the shared memory size
3: for $i = 0; i < WarpCount; i+ = S$ do
4:     if $i \leqslant warpId < i + S$ then
5:         $k \leftarrow warpId - i$
6:         #pragma unroll
7:         for $j = 0; j < 32; j + +$ do
8:             $sMem[k][j][laneId] = data[j]$    ▷ in parallel
9:         end for
10:        #pragma unroll
11:        for $j = 0; j < 32; j + +$ do
12:            $data[j] = sMem[k][laneId][j]$    ▷ in parallel
13:        end for
14:    end if
15:    if $i \leqslant warpId$ then    ▷ select working warps
16:        __syncthreads();
17:    end if
18: end for
```

result in the global memory is the row prefix sum result that is already transposed.

B. *Register-based BRLT-ScanRow Method*

As Fig. 3 shows, the BRLT-ScanRow method computes the prefix sum of all rows and transposes the input matrix in the same step, without using the global memory as a temporary buffer (as in [17] [10]). The input 2D matrix is divided along the column as Fig 3a shows. For instance, given sizeof(T) is 4, we would use $blockDim.y = blockDim.z = 1$, $blockDim.x = 1024$, $gridDim.x = gridDim.z = 1$, $gridDim.y = \lceil H/32 \rceil$.

As Fig 3b shows, first we build a register matrix of size $32 \times WarpSize$ for each warp (each thread holds 32 elements of data using registers). Next, we apply the BRLT method to transpose the register matrix in each CUDA block. Then, we apply a serial scan in each CUDA thread to compute the prefix sum. After that, as Fig 3c shows, the partial prefix sum of each warp is stored to the shared memory of size $WarpCount \times WarpSize$ and compute prefix sum in the shared memory. Finally, the data stored in the shared memory is parallelly aggregated to all the values held in the register cache in their respective $warpId$.

We repeat the BRLT-ScanRow process twice, where the first output matrix become the input of the second computation, and the final result is the required SAT result.

C. *Register-based ScanRowColumn Method*

1) Register-based ScanRow Method: Register-based ScanRow method is used to compute the prefix sum of rows (horizontal direction) for the input matrix as Fig 3a shows. The input matrix is divided into multiple successive blocks along the column. Each CUDA block computes a subset of input matrix of size $WarpCount \times W$. To avoid intra-block communication, each CUDA warp caches and processes successive data in a row of the input matrix. Because each thread caches C elements ($C = 32$ is used), each warp can cache and process data of size $WarpSize * C = 1024$ elements at each step. If the input matrix width is bigger than 1024, the warps repeat the same process to the data after 1024. As Fig. 4 shows, a parallel warp-scan algorithm is applied to the first 32 elements as step $\#1$, the last partial sum is added to the first element of the next 32 elements (the shuffle instruction is used for intra-warp communication), then we repeat the same parallel warp-scan to the elements as shown in step $\#2$. After repeating this process C times, a row of prefix sum results of size $C * WarpSize$ can be obtained. We use the following values when launching our CUDA kernel: the $BlockDim.y = BlockDim.z = 1$, $BlockDim.x = 1024 * sizeof(float)/sizeof(T) = 4096/sizeof(T)$. $GridDim.x = GridDim.z = 1$, and $GridDim.y = \lceil H/WarpCount \rceil$.

2) Register-based ScanColumn Method: Similar to Sec. IV-C1, the input matrix is divided into multiple blocks along the row (horizontal direction). The register-based ScanColumn method is used to compute the prefix sum along the columns by serial warp-scan method rather than parallel warp-scan. Because serial warp-scan performs much better in the column direction, the detailed discussion is reserved to a later section (Sec. V-B3). Each thread caches C elements of data, then each warp builds a register matrix of size $C \times WarpSize$ (namely 32×32) at each step. For each CUDA block, at each step, the prefix sum of size $BlockSize \times C$ can be computed. When the H (namely height) of the input matrix is larger than $BlockSize$, this process will be repeated for all the rows. We use the following values when launching our CUDA kernel: $BlockDim.x = BlockDim.y = 32$, $BlockDim.z = 1$.

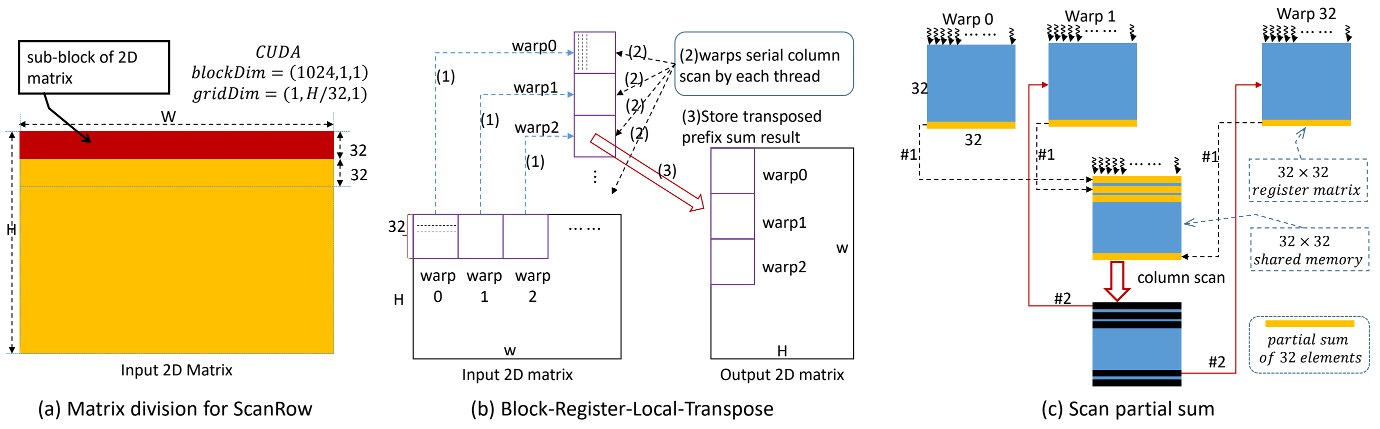


Fig. 3: BRLT-ScanRow algorithm: (a) illustration of how to map CUDA grids and blocks to the input matrix, (b) shows that first we apply the BRLT algorithm (Alg. 5) to transpose the 32×32 register matrix which is cached by each warp, then we in parallel do a scan on the transposed register matrix for the columns using a serial-scan as Alg. 2, finally we save the transposed prefix sum to the output matrix. (c) illustration of using shared memory in the intra-block communication and accumulating the partial-sum of each warp.

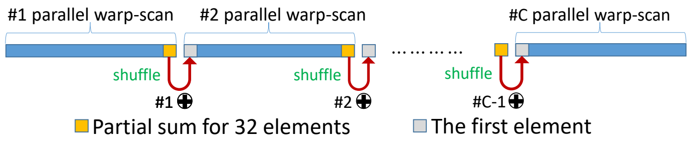


Fig. 4: Using ScanRow to compute the prefix sum of a row in the input 2D matrix by a single warp. ScanRow is the first step of ScanRowColumn algorithm, each block computes the prefix sum for a row of input matrix.

$$GridDim.x = \lceil W/C \rceil, \text{ and } GridDim.y = GridDim.z = 1.$$

V. Performance Model

We analyze the benefit of using a serial scan on each CUDA thread: a main overhead in the BRLT algorithm. We prove that this overhead is negligible by analyzing its effects. Additionally, based on the architecture details of GPU, we give a deep insight into the performance-improving details of our algorithms.

A. Micro-benchmarking

We rely on micro-benchmarking to better understand the characteristics of GPUs. Previous work investigated some details about memory latencies and throughput of GPUs [53], [54]. We further extended and improved the open-source micro-benchmarks cudabmk [53] to measure the shuffle instruction and shared memory access latencies.

Our micro-benchmark measured the latency of accessing shared memory to be 36 clocks on Tesla P100 GPU, 27 clocks on Tesla V100 GPU. The latency of shuffle instruction is 33 clocks/warp on P100, and 39 on V100. The latency of the arithmetic addition and Boolean AND operations are 6 clocks on P100, and 4 clocks on V100. As reported by Nvidia's official programming manual [47], the throughput of shuffle, addition and Boolean *AND* operations are 32, 64 and 64 operations/clock, respectively.

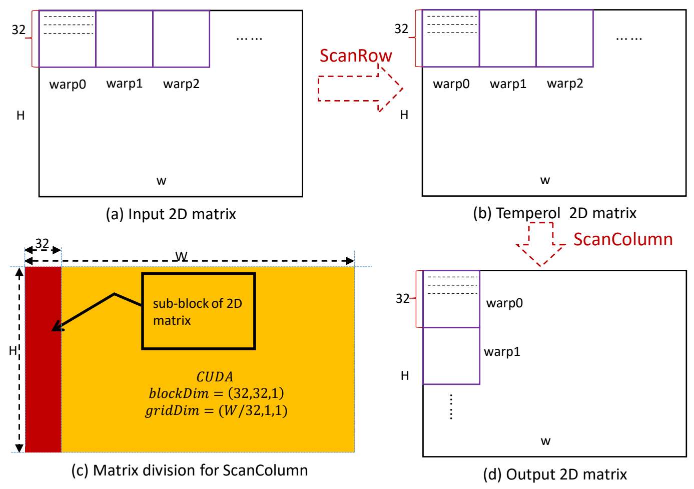


Fig. 5: Illustration of the ScanRowColumn algorithm. First we use the ScanRow algorithm to process the input matrix where the CUDA block mapped as in Fig. 3a. Next we use ScanColumn to process the output from the previous step where the CUDA block is mapped as the figure in the bottom left.

B. Single Warp and Single 32×32 Register Matrix

Our application's optimization strategy is to use a single warp to process a register matrix of size 32×32 (namely $C \times WarpSize$, $C = 32$) at each step. We apply three kinds of methods to compute the prefix sum in the column or row directions. Since the latency of registers is as small as 1 clock, we ignore its effect on performance and focus on discussing the shared memory access and arithmetic operations.

1) Transpose Operation: To locally transpose the register matrix of size 32×32 efficiently, we use the shared memory of size 32×33 as a temporary buffer (Alg. 5) by moving the data from the registers to the shared memory row by row before loading the data from shared memory column to column again. The total number of accesses shared memory (both load and store) is

$$N_{trans\_store\_smem} = N_{trans\_load\_smem} = 32 * 32 = 1024$$

the number of required operation stages is

$$N_{scan\_row\_stage} = C + C = 64$$

Based on the shared memory latency we have measured, the latency of transposing a register matrix of size 32×32 can be estimated as

$$L_{transpose} = N_{scan\_row\_stage} * 36 = 2304\ clocks \tag{3}$$

2) Scan Row Operation: Using parallel warp-scan to compute the prefix sum of 32×32 registers in-place, at least the number of required stages (e.g. Kogge-Stone scan and LF-scan) is

$$N_{scan\_row\_stage} = log_2(WarpSize) * C = 160$$

The number of addition operations in Kogge-Sone scan is

$$N_{Kogge-Stone\_add} = (31 + 30 + 28 + 24 + 16) * C = 4128$$

The number of the shuffle instructions in both parallel warp-scan (Alg. 3 and Alg. 4) in a warp is

$$N_{scan\_row\_sfl} = N_{scan\_row\_stage} = 160$$

the number of addition operations in LF-scan is

$$N_{LF\_add} = (16 + 16 + 16 + 16 + 16) * 32 = 2560$$

In a scan row algorithm the latency that comes from shuffle instructions and addition operations is

$$L_{scan\_row} = N_{scan\_row\_stage} * (33 + 6) = 6240\ clocks \tag{4}$$

However, additional boolean AND operations are needed as shown in Alg. 4 line 4

$$N_{LF\_and} = (WarpSize * N_{scan\_row\_stage}) * C = 5120$$

3) Scan Column Operation: In each warp, we compute the prefix sum in the column direction using the serial warp-scan method (Alg. 2). The number of needed stages is

$$N_{scan\_col\_stage} = C - 1 = 31$$

Since each thread in a warp executes the serial warp-scan without both any intra-warp communication and thread divergence, the total number of concurrent addition operations is

$$N_{scan\_col\_add} = WarpSize * N_{scan\_col\_stage} = 992$$

The latency of scan column can be estimated as

$$L_{scan\_col} = N_{scan\_col\_stage} * 6 = 186\ clocks \tag{5}$$

C. Analyzing Latency and Throughput

On one hand, based on the Equ. 3∼5, we can find out that

$$L_{transpose} + L_{scan\_col} \ll L_{scan\_row} \tag{6}$$

It means the latency of transposing the register matrix and serial scan for column is very low. On the other hand, we consider the aggregate throughput for warps in a CUDA kernel. The number of active warps can be computed as

$$N_{wpb} = \lfloor \frac{BlockSize}{WarpSize} \rfloor \tag{7}$$

$$N_{active\_warps} = N_{sm} * min(\lfloor \frac{Reg\_sm}{Reg\_thread * WarpSize} \rfloor, \lfloor \frac{Smem\_sm}{Smem\_block} \rfloor * N_{wpb}, N_{wpb} * N_{max\_blk\_sm}); \tag{8}$$

where N_{wpb} denotes warp count per SMs, N_{active_warps} is the total warp count of our CUDA kernel, N_{sm} is the total count of SMs, $Smem\_sm$ is the available shared memory size per SMs, $Smem\_block$ is the used shared memory in our CUDA kernel, and $N_{max\_blk\_sm}$ is the maximum available block count per SM. The shared memory bandwidth for each active warp is

$$BW_{warp} = BW_{Smem}/N_{active\_warps} \tag{9}$$

where BW_{Smem} is the shared memory bandwidth. In our CUDA kernel, the time required to transpose a register matrix of size 32×32 is

$$T_{trans} = \frac{(N_{trans\_store\_smem} + N_{trans\_load\_smem}) * sizeof(T)}{BW_{Smem}} \tag{10}$$

Using serial warp-scan, the time of arithmetic operations is

$$T_{scan\_col\_add} = N_{scan\_col\_add}/BW_{add} \tag{11}$$

Using parallel warp-scan, the required time for shuffle instructions is

$$T_{shuffle} = \frac{N_{scan\_row\_sfl}}{BW_{shuffle}} \tag{12}$$

where $BW_{shuffle}$ is the shuffle instruction throughput. The required time for the arithmetic pipeline varies from method to another. Take the Kogge-Stone scan for example

$$T_{Kogge-Stone\_add} = \frac{N_{Kogge-Stone\_add}}{BW_{add}} \tag{13}$$

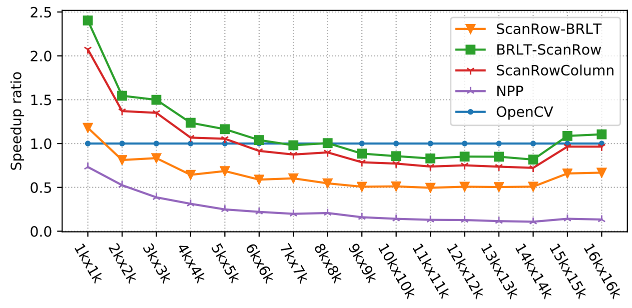
where BW_{add} indicates addition operations throughput. We can easily derive that

$$T_{Kogge-Stone\_add} + T_{shuffle} \gg T_{trans} + T_{scan\_col\_add} \tag{14}$$

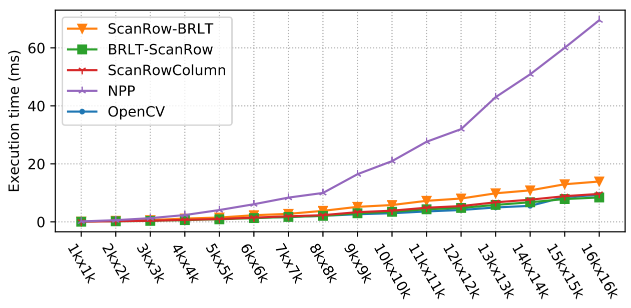
Similarly, we can obtain the function also as

$$T_{LF\_add} + T_{LF\_and} + T_{shuffle} \gg T_{trans} + T_{scan\_col\_add} \tag{15}$$

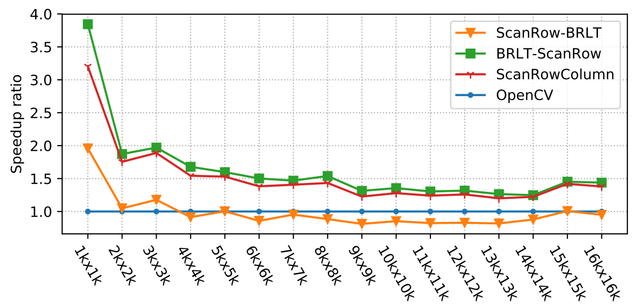
Note that to Equ. 14 and Equ. 15, we suppose the shared memory access achieves the peak of memory bandwidth without bank conflicts. Otherwise, the shared memory throughput will dramatically degrade, and our model would have to be adjusted accordingly. Finally, the bandwidth of shared memory we use in our model is 9519GB/s on P100 and 13800GB/s on V100 as reported by [55], respectively.



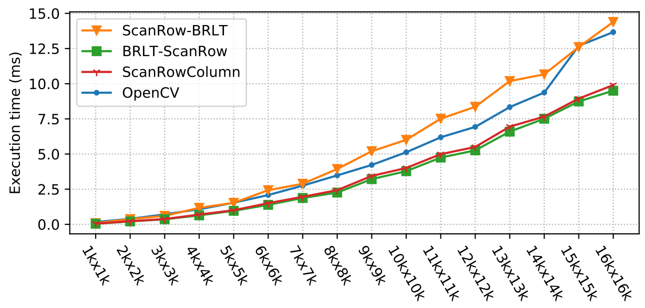
(a) Speedup of 8u32s on Tesla P100 GPU.



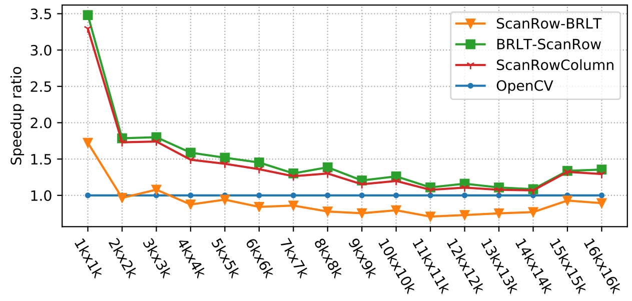
(b) Execution time of 8u32s on Tesla P100 GPU.



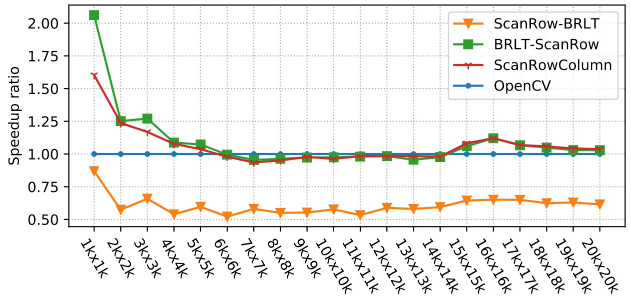
(c) Speedup of 32f32f on Tesla P100 GPU.



(d) Execution time of 32f32f on Tesla P100 GPU.



(e) Speedup of 32u32u on Tesla P100 GPU.



(f) Speedup of 64f64f on Tesla P100 GPU.

Fig. 6: Speedup (we use OpenCV's implementation as the baseline) and execution time of SAT on a single Tesla P100 GPU.

VI. Evaluation

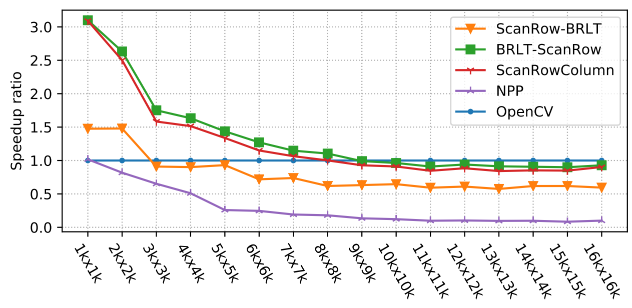
We report the performance of the proposed algorithms and discuss comparisons with NPP and OpenCV libraries.

A. Experimental Setup

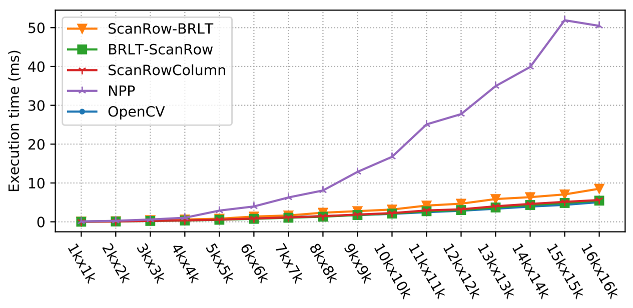
Our algorithms are implemented by Nvidia CUDA Toolkit 9.0. To obtain the highest accuracy of CUDA kernel execution time (often tens of us), the $nvprof$ with print-gpu-trace option is used to observe the CUDA kernel execution. SAT functions that are implemented by the OpenCV 3.4.1 [41] and Nvidia NPP 9.0 [47] libraries are adopted for performance comparison. Nvidia Tesla P100 and V100 GPUs are used for performance evaluation and the ECC option is ON. Different sizes ($1k \times 1k \sim 16k \times 16k$) and data types are evaluated, e.g. 8u, 32u, 32f and 64f. It is worth mentioning that precision loss and overflow can happen in scan operators. It depends on the problem size, data type, and the input values. It is outside the scope of this work to address such problem on a case-by-case basis (as in [4], [56]).

B. Performance Overview

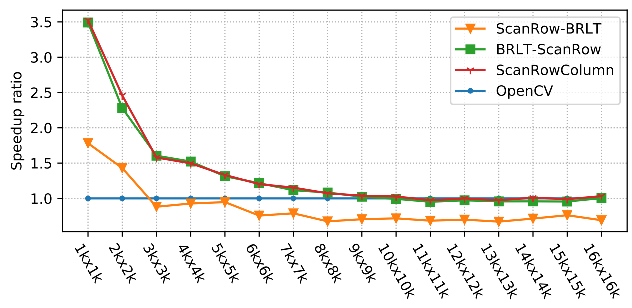
Both our algorithms and OpenCV implement SAT by C++ template functions, so we can compare performance with OpenCV for all kinds of data types. In our experiments, we evaluate both Kogge-Stone scan and LF-scan. However, they achieve nearly the same computing efficiency in our implementation. To simplify, in Fig. 6 and Fig. 7, Kogge-Stone method is applied for parallel warp-scan to compute all of the results. For some data types, the performance graphic nearly the same, so we report one of them: e.g. 8u32s, 8u32u, and 8u32f. When comparing the execution time with other



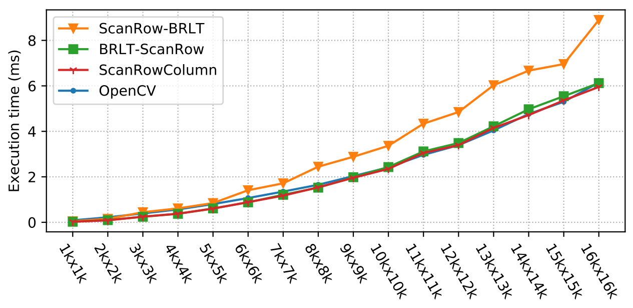
(a) Speedup of 8u32s on Tesla V100 GPU.



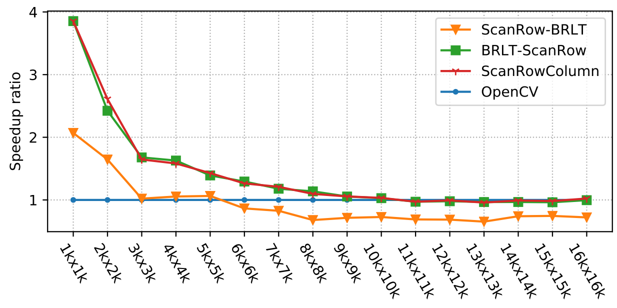
(b) Execution time of 8u32s on Tesla V100 GPU.



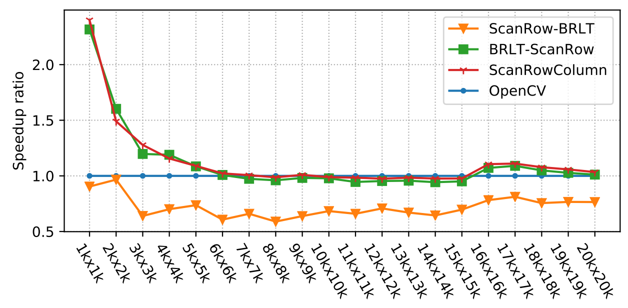
(c) Speedup of 32f32f on Tesla V100 GPU.



(d) Execution time of 32f32f on Tesla V100 GPU.



(e) Speedup of 32u32u on Tesla V100 GPU.



(f) Speedup of 64f64f on Tesla V100 GPU.

Fig. 7: Speedup (we use OpenCV's implementation as the baseline) and execution time of SAT on a single Tesla V100 GPU.

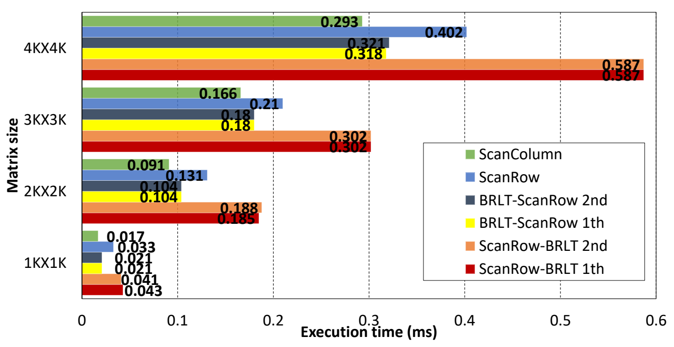
libraries, we report the execution time for different data types, such as in Fig. 6b, Fig. 6d, Fig. 7b and Fig. 7d.

1) Nvidia NPP: The NVIDIA Performance Primitives (NPP) is an optimized library for performing CUDA accelerated computations. The SAT function provided by NPP is $nppIntegral$. However, only two kinds of input and output data type are provided by NPP: 8u32s and 8u32f. Considering that NPP is closed-source, we investigated its binary by the $nvprof$ and $cuobjdump$ tools, which are provided by Nvidia. The investigated results are listed in Table II. *Regs* is the number of registers used per CUDA thread. *SSMem* and *DSMem* are the static and dynamic shared memory allocated per CUDA block, respectively.

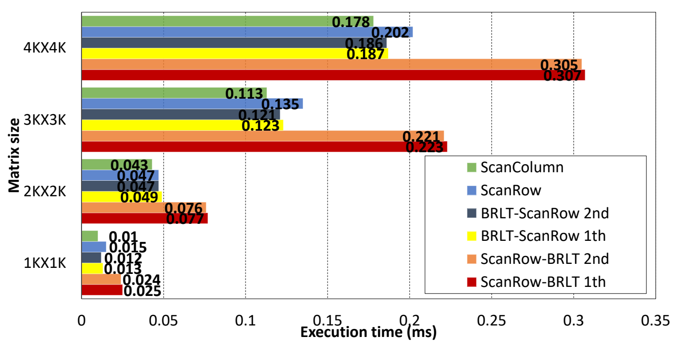
2) OpenCV Library: OpenCV (Open Source Computer Vision Library) is a highly optimized library designed for efficient computation and real-time applications. We use the latest version v3.4.1 for our evaluation. The SAT function $integral()$ is implemented by OpenCV using the scan-scan algorithm, their functions are $vertical\_pass()$ and $horisontal\_pass()$. Additionally, using $shuffle$ instruction, OpenCV specifically optimizes $8u$ (namely $unsigned\ char$) input data for the horizontal scan, the function name is $horisontal\_pass\_8u\_shfl$. It is worth mentioning that $horisontal\_pass\_8u\_shfl$ employs two optimization strategies, one of them is using the register-based parallel Kogge-Stone scan algorithm, the other is that each thread loads 16 $bytes$ data to registers using $uint4$ data type, then applies a serial-scan to these 16 ele-

TABLE II: NPP CUDA kernel details

kernel	blockSize	gridSize	Regs	SSMem	DSMem
$scanRow$	$(256, 1, 1)$	$(1, H, 1)$	20	2.25KB	0
$scanCol$	$(1, 256, 1)$	$(W + 1, 1, 1)$	18	2.25KB	0



(a) 32f32f evaluation on Tesla P100 GPU



(b) 32f32f evaluation on Tesla V100 GPU

Fig. 8: Performance breakdown of computing 32f32f ranging from $1k \times 1k$ to $4k \times 4k$ input matrices. For the specified input matrix, the 1^{st} and 2^{nd} computing time are displayed: namely 1^{st} scan and 2^{nd} scan.

ments on each threads. However, $horisontal\_pass\_8u\_shfl$ is customized and works only for $8u$ input data.

C. Speedup

Fig. 6 and Fig. 7 show several comparison results. The execution times are measured by $nvprof$. We disregard the data copy time between host and device since it is the same for all libraries. We manually accumulate the two compute kernels (namely row and column prefix sum) and average 10 executions. Register pressure results in the speedup disappear when matrices go to larger. Even though we evaluated matrix sizes up to $20k \times 20k$, in real applications, the matrix sizes are typically smaller than $8k \times 8k$. On a single Tesla P100 GPU and V100 GPU, our best algorithm performs up to 2.3× faster than OpenCV, and 3.2× than NPP.

1) Kogge-stone Scan and LF-scan: In our experiments, both Kogge-stone scan and LF-scan are evaluated. Since the SAT computation is memory-bound, we can not gain speedup from LF-scan as in [44].

2) Generality & Simplicity: Our fastest algorithm, namely BRLT-ScanRow, is implemented as a single CUDA kernel and repeatedly called twice to compute a SAT. It is simple to be implemented and general to all of the data types. However, NPP, OpenCV and other implementations [17], [36] use multiple CUDA kernels, often customized to different data types.

3) 32f32f for Deep Learning: As discussed earlier, OpenCV's implementation varies from data type to another. For image processing computation, the data type 8u is widely used to represent the image data. However, in Deep Learning computation, 32f is the prevalent data type. Our 32f32f outperform OpenCV as shown in Fig. 6c and Fig. 7c.

4) 64f64f: Fig. 6f and Fig. 7f shows that our implementation performs better than OpenCV for up to image matrix size $6k \times 6k$.

D. Model Verification on BRLT Overhead

In Fig. 8, we display the detailed execution time of the two steps of computation. Since the BRLT algorithm is used, the BRLT-ScanRow and ScanRow-BRLT function are called twice. ScanRow and ScanColumn are used once to compute the prefix sum of rows and columns, respectively. In this section, T denotes the execution time. We can verify our performance model as

1) $T_{ScanColumn} < T_{BRLT-ScanRow}$. After BRLT finishes, ScanColumn and BRLT-ScanRow compute the column prefix sum similarly, so we can easily derive that BRLT is the overhead operation.
2) $2 * T_{BRLT-ScanRow} < T_{ScanRow} + T_{ScanColumn}$ indicates some benefits from BRLT operation in computing SAT.
3) $T_{BRLT-ScanRow} > T_{ScanRow-BRLT}$ means that our serial warp-scan(Alg. 2) method is more efficient than shuffle-based parallel warp-scan(Alg. 3, Alg.4).

VII. Conclusion and Future Work

SAT is a widely used primitive in many applications. In the past two decades, SAT successfully accelerated many image filtering and local feature extracting algorithms, and recently boosted many deep learning workloads also. In terms of hardware architecture and computing efficiency, it is challenging to optimize the SAT implementation on GPUs. Based on the register cache technology, we proposed three kinds of algorithms for fast SAT computation on GPUs. With our novel BRLT method, we can transform the scan problem from horizontal operations to vertical, which results in reducing both the arithmetic computation and communication between threads, while achieving coalesced access to global memory. Furthermore, the BRLT overhead is shown to be negligible. We provide a deep insight to register cache and shared memory cache for programming CUDA-enabled GPUs. The BRLT method is general and can be applied to optimize many other algorithms, such as Fast Fourier Transform (FFT), Wavelet Transform, Discrete Cosine Transform (DCT), etc.

Acknowledgment

This work was partially supported by JST-CREST under Grant Number JPMJCR1303 and JPMJCR1687.

References

[1] F. C. Crow, "Summed-area tables for texture mapping," in *ACM SIGGRAPH computer graphics*, vol. 18, no. 3. ACM, 1984, pp. 207–212.

[2] P. Viola and M. Jones, "Rapid object detection using a boosted cascade of simple features," in *Computer Vision and Pattern Recognition, 2001. CVPR 2001. Proceedings of the 2001 IEEE Computer Society Conference on*, vol. 1. IEEE, 2001, pp. I–I.

[3] B. Han, C. Yang, R. Duraiswami, and L. Davis, "Bayesian filtering and integral image for visual tracking," in *Workshop on Image Analysis for Multimedia Interactive Services*. Citeseer, 2005.

[4] J. Hensley, T. Scheuermann, G. Coombe, M. Singh, and A. Lastra, "Fast summed-area table generation and its applications," in *Computer Graphics Forum*, vol. 24, no. 3. Wiley Online Library, 2005, pp. 547–555.

[5] H. Bay, T. Tuytelaars, and L. Van Gool, "Surf: Speeded up robust features," in *European conference on computer vision*. Springer, 2006, pp. 404–417.

[6] J. Hensley, T. Scheuermann, M. Singh, and A. Lastra, "Fast hdr image-based lighting using summed-area tables," in *Symposium on Interactive 3D Graphics and Games (Poster)*. Citeseer, 2007.

[7] D. Bradley and G. Roth, "Adaptive thresholding using the integral image," *Journal of graphics tools*, vol. 12, no. 2, pp. 13–21, 2007.

[8] C. Messom and A. Barczak, "Stream processing of geometric and central moments using high precision summed area tables," in *International Conference on Neural Information Processing*. Springer, 2008, pp. 1095–1102.

[9] D. Nehab, A. Maximo, R. S. Lima, and H. Hoppe, "Gpu-efficient recursive filtering and summed-area tables," in *ACM Transactions on Graphics (TOG)*, vol. 30, no. 6. ACM, 2011, p. 176.

[10] D. Oro, C. Fernández, J. R. Saeta, X. Martorell, and J. Hernando, "Real-time gpu-based face detection in hd video sequences," in *Computer Vision Workshops (ICCV Workshops), 2011 IEEE International Conference on*. IEEE, 2011, pp. 530–537.

[11] M. P. B. Slomp, "Real-time photographic local tone reproduction using summed-area tables," 2008.

[12] M. Xi, L. Chen, D. Polajnar, and W. Tong, "Local binary pattern network: a deep learning approach for face recognition," in *Image processing (ICIP), 2016 IEEE international conference on*. IEEE, 2016, pp. 3224–3228.

[13] K. Fernandes and J. S. Cardoso, "Deep local binary patterns," *arXiv preprint arXiv:1711.06597*, 2017.

[14] A. Kasagi, T. Tabaru, and H. Tamura, "Fast algorithm using summed area tables with unified layer performing convolution and average pooling," in *Machine Learning for Signal Processing (MLSP), 2017 IEEE 27th International Workshop on*. IEEE, 2017, pp. 1–6.

[15] J. P. Lewis, "Fast template matching," in *Vision interface*, vol. 95, no. 120123, 1995, pp. 15–19.

[16] X. Wang, T. X. Han, and S. Yan, "An hog-lbp human detector with partial occlusion handling," in *Computer Vision, 2009 IEEE 12th International Conference on*. IEEE, 2009, pp. 32–39.

[17] B. Bilgic, B. K. Horn, and I. Masaki, "Efficient integral image computation on the gpu," in *Intelligent vehicles symposium (IV), 2010 IEEE*. IEEE, 2010, pp. 528–533.

[18] A. Davidson and J. D. Owens, "Register packing for cyclic reduction: A case study," in *Proceedings of the fourth workshop on general purpose processing on graphics processing units*. ACM, 2011, p. 4.

[19] J. Lai and A. Seznec, "Performance upper bound analysis and optimization of sgemm on fermi and kepler gpus," in *Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE Computer Society, 2013, pp. 1–10.

[20] P. Enfedaque, F. Auli-Llinas, and J. C. Moure, "Implementation of the dwt in a gpu through a register-based strategy," *IEEE Transactions on Parallel and Distributed Systems*, vol. 26, no. 12, pp. 3394–3406, 2015.

[21] Y. Wei, X. Bing, and C. Chareonsak, "Fpga implementation of adaboost algorithm for detection of face biometrics," in *Biomedical Circuits and Systems, 2004 IEEE International Workshop on*. IEEE, 2004, pp. S1–6.

[22] V. Nair, P.-O. Laprise, and J. J. Clark, "An fpga-based people detection system," *EURASIP journal on applied signal processing*, vol. 2005, pp. 1047–1061, 2005.

[23] T. Theocharides, N. Vijaykrishnan, and M. J. Irwin, "A parallel architecture for hardware face detection," in *Emerging VLSI Technologies and Architectures, 2006. IEEE Computer Society Annual Symposium on*. IEEE, 2006, pp. 2–pp.

[24] J. Svab, T. Krajnik, J. Faigl, and L. Preucil, "Fpga based speeded up robust features," in *Technologies for Practical Robot Applications, 2009. TePRA 2009. IEEE International Conference on*. IEEE, 2009, pp. 35–41.

[25] C. He, A. Papakonstantinou, and D. Chen, "A novel soc architecture on fpga for ultra fast face detection," in *Computer Design, 2009. ICCD 2009. IEEE International Conference on*. IEEE, 2009, pp. 412–418.

[26] L. Zhang, K. Zhang, T. S. Chang, G. Lafruit, G. K. Kuzmanov, and D. Verkest, "Real-time high-definition stereo matching on fpga," in *Proceedings of the 19th ACM/SIGDA international symposium on Field programmable gate arrays*. ACM, 2011, pp. 55–64.

[27] S.-S. Lee, S.-J. Jang, J. Kim, Y. Hwang, and B. Choi, "Memory-efficient surf architecture for asic implementation," *Electronics Letters*, vol. 50, no. 15, pp. 1058–1059, 2014.

[28] J.-P. Derutin, F. Dias, L. Damez, and N. Allezard, "Simd, smp and mimd-dm parallel approaches for real-time 2d image stabilization," in *Computer Architecture for Machine Perception, 2005. CAMP 2005. Proceedings. Seventh International Workshop on*. IEEE, 2005, pp. 73–80.

[29] O. Bilaniuk, E. Fazl-Ersi, R. Laganiere, C. Xu, D. Laroche, and C. Moulder, "Fast lbp face detection on low-power simd architectures," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition Workshops*, 2014, pp. 616–622.

[30] T. Wu and A. Toet, "Speed-up template matching through integral image based weak classifiers," *Journal of Pattern Recognition Research*, vol. 1, pp. 1–12, 2014.

[31] S. Matsuoka, "The road to tsubame and beyond," in *High Performance Computing on Vector Systems 2007*. Springer, 2008, pp. 265–267.

[32] J. Wells, B. Bland, J. Nichols, J. Hack, F. Foertter, G. Hagen, T. Maier, M. Ashfaq, B. Messer, and S. Parete-Koon, "Announcing supercomputer summit," ORNL (Oak Ridge National Laboratory (ORNL), Oak Ridge, TN (United States)), Tech. Rep., 2016.

[33] S. Matsuoka, "Being bytes-oriented in hpc leads to an open big data/ai ecosystem and further advances into the post-moore era," in *Big Data (Big Data), 2017 IEEE International Conference on*. IEEE, 2017, pp. 5–5.

[34] M. Poostchi, K. Palaniappan, F. Bunyak, M. Becchi, and G. Seetharaman, "Efficient gpu implementation of the integral histogram," in *Asian Conference on Computer Vision*. Springer, 2012, pp. 266–278.

[35] A. Kasagi, K. Nakano, and Y. Ito, "Parallel algorithms for the summed area table on the asynchronous hierarchical memory machine, with gpu implementations," in *Parallel Processing (ICPP), 2014 43rd International Conference on*. IEEE, 2014, pp. 251–260.

[36] Q. Dang, S. Yan, and R. Wu, "A fast integral image generation algorithm on gpus," in *Parallel and Distributed Systems (ICPADS), 2014 20th IEEE International Conference on*. IEEE, 2014, pp. 624–631.

[37] S. Ehsan, A. F. Clark, K. D. McDonald-Maier *et al.*, "Integral images: efficient algorithms for their computation and storage in resource-constrained embedded vision systems," *Sensors*, vol. 15, no. 7, pp. 16 804–16 830, 2015.

[38] M. Poostchi, K. Palaniappan, D. Li, M. Becchi, F. Bunyak, and G. Seetharaman, "Fast integral histogram computations on gpu for real-time video analytics," *arXiv preprint arXiv:1711.01919*, 2017.

[39] S. Taylor, *Intel integrated performance primitives*. Intel Press,, 2004.

[40] U. G. Matlab, "The mathworks," *Inc., Natick, MA*, vol. 1992, 1760.

[41] G. Bradski and A. Kaehler, "Opencv," *Dr. Dobbs journal of software tools*, vol. 3, 2000.

[42] M. Akhloufi and A. Campagna, "Openclipp: Opencl integrated performance primitives library for computer vision applications," in *Proc. SPIE Electronic Imaging*, 2014, pp. 25–31.

[43] J. Demouth, "Shuffle: Tips and tricks.(2013)," 2013.

[44] A. P. Diéguez, M. Amor, and R. Doallo, "Efficient scan operator methods on a gpu," in *Computer Architecture and High Performance Computing (SBAC-PAD), 2014 IEEE 26th International Symposium on*. IEEE, 2014, pp. 190–197.

[45] Y. Liu and S. Aluru, "Lightscan: Faster scan primitive on cuda compatible manycore processors," *arXiv preprint arXiv:1604.04815*, 2016.

[46] S. Maleki, A. Yang, and M. Burtscher, *Higher-order and tuple-based massively-parallel prefix sums*. ACM, 2016, vol. 51, no. 6.

[47] C. Nvidia, "Cuda c programming guide," 2018.

[48] R. P. Brent and H.-T. Kung, "A regular layout for parallel adders," *IEEE transactions on Computers*, no. 3, pp. 260–264, 1982.

[49] G. E. Blelloch, "Prefix sums and their applications," 1990.

[50] P. M. Kogge and H. S. Stone, "A parallel algorithm for the efficient solution of a general class of recurrence equations," *IEEE transactions on computers*, vol. 100, no. 8, pp. 786–793, 1973.

[51] S.-W. Ha and T.-D. Han, "A scalable work-efficient and depth-optimal parallel scan for the gpgpu environment," *IEEE Transactions on Parallel and Distributed Systems*, vol. 24, no. 12, pp. 2324–2333, 2013.

[52] R. E. Ladner and M. J. Fischer, "Parallel prefix computation," *Journal of the ACM (JACM)*, vol. 27, no. 4, pp. 831–838, 1980.

[53] H. Wong, M.-M. Papadopoulou, M. Sadooghi-Alvandi, and A. Moshovos, "Demystifying gpu microarchitecture through microbenchmarking," in *Performance Analysis of Systems & Software (ISPASS), 2010 IEEE International Symposium on*. IEEE, 2010, pp. 235–246.

[54] X. Mei, K. Zhao, C. Liu, and X. Chu, "Benchmarking the memory hierarchy of modern gpus," in *IFIP International Conference on Network and Parallel Computing*. Springer, 2014, pp. 144–156.

[55] Z. Jia, M. Maggioni, B. Staiger, and D. P. Scarpazza, "Dissecting the NVIDIA Volta GPU Architecture via Microbenchmarking," *ArXiv e-prints*, Apr. 2018.

[56] M. Rouhifar, M. Hosseinzadeh, S. Bahanfar, and M. Teshnehlab, "Fast overflow detection in moduli set {2n–1, 2n, 2n+ 1}," *International Journal of computer science issues*, vol. 8, no. 3, pp. 407–414, 2011.