Journal of Korea Game Society 2012 Oct; 12(5): 67-78
http://dx.doi.org/10.7583/JKGS.2012.12.5.67

컴퓨터 그래픽스

CUDA를 이용한 효율적인 합산 영역 테이블의 생성 방법

하상원[*], 최문희[**], 전태준[*], 김진우[*], 변혜란[*], 한탁돈[*]
연세대학교 컴퓨터 과학과[*], 삼성전자(주)[**]
swha815@gmail.com, mhchoi.0310@samsung.com, taejoon89@me.com,
jwkim@msl.yonsei.ac.kr, hrbyun@yonsei.ac.kr, hantack55@gmail.com

Bandwidth Efficient Summed Area Table Generation for CUDA

Sang-Won Ha[*], Moon-Hee Choi[**], Tae-Joon Jun[*], Jinwoo Kim[*],
Hyeran Byun[*], Tack-Don Han[*]
Dept. of Computer Science, Yonsei Univ.[*], Samsung Electronics Corp.[**]

요 약

합산 영역 테이블은 모든 픽셀에 대해 임의의 크기 사각영역의 이미지 필터링 처리를 일정 시간 안에 가능케 한다. 이러한 특성은 각각의 픽셀에 대해서 주변 픽셀의 밝기의 합 혹은 평균을 필요로 하는 이미지 처리 적용 분야에 유용하게 쓰일 수 있다. 합산 영역 테이블의 생성은 단지 행 혹은 열 단위의 합만을 구하는 메모리 바운드 작업임에도 불구하고 기존 연구들은 이미 존재하는 데이터 병렬성만을 활용하기 위하여 대기 시간이 긴 전역 메모리에 과도한 접근을 하여야만 했다. 본 논문에서는 입력 데이터를 정방의 서브 이미지로 분할하고 매개 데이터를 이들 간에 파급시킴으로써 GPGPU 환경 적합한 알고리즘을 제안하고자 한다. 이를 통하여 기존 방법 대비 전역 메모리 접근 량을 거의 반으로 줄임으로써 주어진 메모리 대역폭을 효율적으로 사용한다. 결과에서도 성능이 대폭 향상되었다.

ABSTRACT

Summed area table allows filtering of arbitrary-width box regions for every pixel in constant time per pixel. This characteristic makes it beneficial in image processing applications where the sum or average of the surrounding pixel intensity is required. Although calculating the summed area table of an image data is primarily a memory bound job consisting of row or column-wise summation, previous works had to endure excessive access to the high latency global memory in order to exploit data parallelism. In this paper, we propose an efficient algorithm for generating the summed area table in the GPGPU environment where the input is decomposed into square sub-images with intermediate data that are propagated between them. By doing so, the global memory access is almost halved compared to the previous methods making an efficient use of the available memory bandwidth. The results show a substantial increase in performance.

Keywords : Summed area table(합산 영역 테이블), Integral Map(적분 맵), GPGPU(지피지피유), Parallel Prefix Scan(병렬 프리픽스 스캔), Parallel Algorithm(병렬 알고리즘)

Received: Agu. 31, 2012 Accepted: Sept. 27, 2012
Corresponding Author: Tack-Don Han(Yonsei University)
E-mail: hantack55@gmail.com

ISSN: 1598-4540

Ⓒ The Korea Game Society. All rights reserved. This is an open-access article distributed under the terms of the Creative Commons Attribution Non-Commercial License http://creativecommons.otg/licenses/by-nc/3.0), which permits unrestricted non-commercial use, distribution, and reproduction in any medium, provided the original work is properly cited.

1. Introduction

Originally devised as an alternative to mip mapping, the usage of summed area table has been extended to various types of blur filtering to increase the realism in 3D graphics.

With the minuscule additional overhead of generating the summed area table, the sum of an arbitrary rectangular region for all the pixels of an image can be simply calculated by a few addition operations per pixel during run-time. This can be used to filter regions of different size for every pixel in constant time per pixel, invariant of the region size.

There are numerous applications for summed area table including anti-aliasing[1], image filtering[2] environmental mapping[3], and depth of field[4]. Moreover, feature tracking applications[5,6] in augmented reality use summed area table as a way to filter the image to make the edges more prominent and suitable for feature candidates.

The rest of the paper is organized as the following. Section 2 briefly describes previous related work. Then in section 3, fundamental idea behind summed area table and issues regarding GPU implementation is given. Section 4 proposes our memory bandwidth efficient algorithm. Section 5 demonstrates the experimental results of the proposed algorithm and makes comparison against previous implementations. Finally, we draw to a conclusion in section 6 and express future related work.

2. Related Work

Crow[1] invented the concept of summed area table to find a more accurate value for anti-aliasing mapped texture. Because the main concern was on the quality of the output image, he had inconclusive results on the computation time.

Hensley et. al.[3] extended the application of the summed area table to creating glossy and transparent surfaces at interactive rates. However, they employed a work-inefficient method of recursive doubling to calculate the prefix scan.

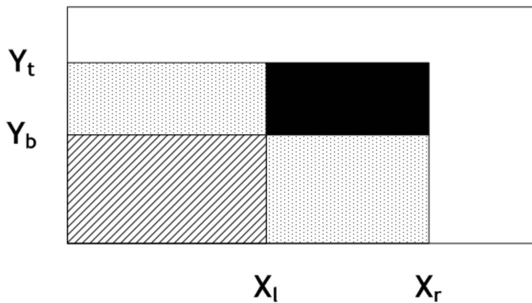
Harris et. al.[7] described summed area table as an example of their work-efficient parallel prefix scan. In order to maintain global memory coalescing when fetching data in vertical columns, they transposed the image so that the columns can be processed horizontally.

In this paper, we present a novel method of generating the summed area table. All accesses to the global memory are coalesced and the total amount is reduced by nearly half compared to previous methods. In addition, the performance can be optimized for various graphics hardware by modulating the data processed per thread-block, a tradeoff between communication time and memory latency hiding capability.

3. Summed Area Table and GPU

3.1 Summed Area Table

The summed area table contains the sum of the pixel values from the lower left corner of an image to the location of the current pixel for all pixels. The sum of the pixel values for an axis-aligned rectangular region, as shown



[Fig. 1] Calculation of summed area from table

by the blackened area in [Fig. 1], can be found by simply taking the value at (X_r, Y_t) and subtracting the values at (X_l, Y_t) and (X_r, Y_b) followed by adding the value at (X_l, Y_b). This is true because the integral of the (X_l, Y_b, X_r, Y_t) region equals the integral of (0, 0, X_r, Y_t) minus the integrals of regions (0, 0, X_l, Y_t) and (0, 0, X_r, Y_b) and then adding the integral for (0, 0, X_l, Y_b) since the hatched area has been subtracted twice. Similarly, the average of the target region can be found by calculating the sum and then dividing by the area of the region.

3.2 GPGPU

The GPU is a commodity hardware which provides performance rivaling that of a supercomputer at a fraction of the cost. It is composed of a highly parallel architecture which leverages between the cost-effective single-instruction-multiple-data(SIMD) design and the programmable flexibility of the multiple-instruction-multiple-data(MIMD) design. In respect to that matter, it resembles more of a single-program-multiple-data (SPMD) model. The SIMD characteristic of the GPU is realized by a method called single-instruction-multiple-thread(SIMT) where a batch of cooperatively synchronized computation units called scalar processors(SP) execute identical instruction at any given time. This batch is named streaming multiprocessor (SM) and is composed of a fixed number of SPs. On the other hand, each of the SMs are completely exclusive of each other in terms of instruction dispatch and execution. Therefore, SPMD property is achieved.

As there may be thousands of SPs present on a GPU, supplying the appropriate data to the computation core is a critical issue. In the current GPU, memory is organized in a hierarchical structure to alleviate the bandwidth problem[8].

The global memory which may be used as a communication channel between all the cores, provides huge storage space reaching up to 3GB. However, the latency can be up to hundreds of cycles per memory fetch. Therefore, it is more widely used as a medium for initial transfer of data from the host memory to the SPs and returning the final results back to the host. In order to increase the performance of the global memory subsystem, the GPU employs a method called coalescing. Here, memory fetch is made in units of 128-byte aligned contiguous memory block. If all the SPs within a SM can be serviced by a block, access is compacted into a single transaction. However, if data required by a SM spans across multiple blocks, the access are serialized, thus fracturing coalescing and degrading performance. As global memory is the slowest component in a GPU by hundreds of times than any other units, loss of coalesced access has the greatest impact on the overall performance and even discourages

GPU parallelization all together.

The second type of memory in the GPU is the shared memory. It is a low latency memory built into every SM and is utilized as a means to transfer data between SPs within a SM. Most applications implemented on the GPU depend heavily on it for fast storage. However, the size of the shared memory is restrictive to at most 48KB per SM. In addition, it is one of the limiting factors on how many logical kernels, called warps, can be resident on a SM at a time. The number of resident warps equates to the chances of hiding memory latencies and branch overhead. Therefore, careful negotiation between the amount of shared memory usage and making efficient use of the available memory resources is critical in obtaining optimal performance.

Lastly, the private memory is exclusive to a particular SP and usually used as a per-thread cache substitute or storing temporary data. There are other types of memory such as constant or local memory but are seldomly used.

Except for operations which involve usage of the special function unit, the throughput of computation units is one or two cycles per instruction. On the other hand, strict criteria, such as avoidance of memory bank conflict or instruction-level parallelism, must be met to achieve comparable performance from the on-chip memory subsystem such as shared memory or registers[9]. Thus, implementations on the GPU must be memory bound and try to make the memory access pattern and bandwidth as optimal as possible.

3.3 Image Transpose Method

The generation of the summed area table in GPGPU environment typically incorporates a two-pass algorithm. On the first pass, each row of the image is prefix scanned individually and stored in 2D array format. On the second pass, the stored values are prefix scanned by columns. However, a straightforward implementation as mentioned would hamper performance, because a typical GPU performs transaction on the global memory in chunks of fixed-size bytes usually between 32 to 128 bytes. Therefore, executing prefix scan on the columns would require long strides through the memory where each transaction would result in most of the fetched data to be discarded from non-coalesced reads.

Harris et. al.[7] solved this problem by transposing the 2D array after the row-wise scan. This way, the second pass can be made on rows of data instead of the columns and still have the same effect.

The image transpose method proposed by Harris et. al initiates by de-interlacing the input 8-bit per channel RGBA to four 32-bit floating point values. This is due to the fact that the GPU is optimized for processing 32-bit floating point values streaming through the hardware's pixel pipeline[8]. As such, they tried to use the most proficient data type. Afterwards, prefix scan is performed on the row-wise fashion. In another words, each row is scanned independent of other rows. The method incorporated in the parallel prefix scan algorithm is known as recursive doubling[10] devised by Kogge and Stone. At the added cost of O(nlogn) computations, this method reduces the original O(n) processing time to O(logn) if enough parallel processors are

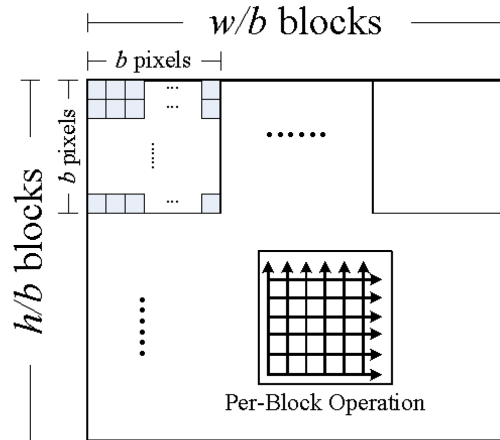
available.

Thereafter, the image is transposed. The input image data is partitioned into sub-images of 16x16 pixels to be processed by each thread-block consisting of 256 threads which equates to one thread per pixel. Appropriate sub-image for each thread-block is loaded from the global memory and transposed within the shared memory. Then, the processed sub-image stored with the x and y axis swapped in the global memory space.

Lastly, the first phase of row-wise parallel prefix scan is repeated on the transpoed image data. The resulting summed area table is transposed compared to the desired orientation. Therefore, when accessing the generated summed area table, the x and y coordinates must be swapped to obtain the correct value.

4. Proposed Method

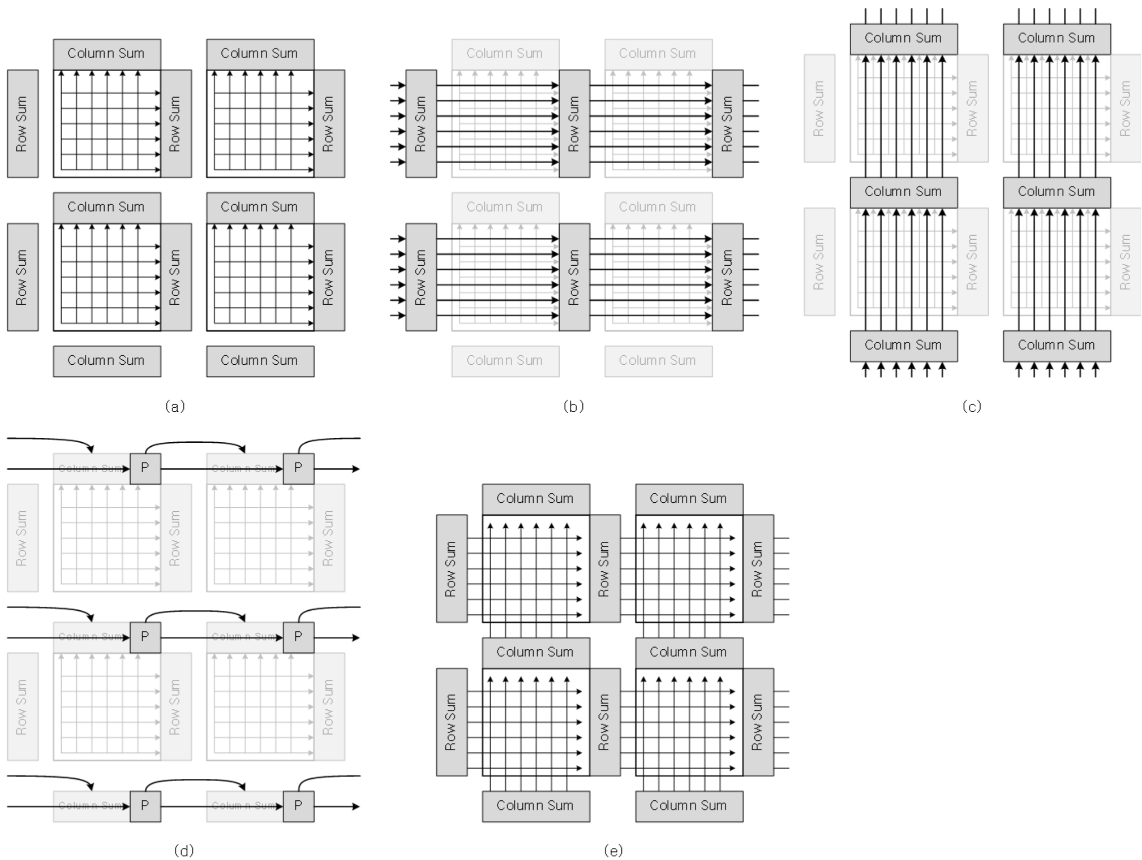
Performance of the applications implemented on the GPU are usually bound by the amount of memory access and the efficiency of the access pattern. This is because the device memory is the slowest component in the GPU in addition to the fact that the memory subsystem is optimized for bandwidth rather than latency unlike the host memory. Assuming one byte per pixel, the previous method of transposing the image to maintain global memory coalescing requires $2*w*h$ accesses each for row-wise scan, transpose, and column-wise scan for a total of $6*w*h$ where w and h are width and height in pixels of the image respectively. This amounts to the image being accessed recursively six times.



[Fig. 2] The proposed divide and conquer method. The input data is evenly divided between multiple thread blocks which perform row and column-wise summation.

In order to minimize the duplicative memory access to the high latency global memory, we propose a four-pass algorithm suitable for GPU architectures where the image is uniformly divided into square sub-images of b2 pixels per thread-block. A small value for b means relatively less pixels are processed by a thread-block causing higher number of overall partial sums to be created which translates to increased communication overhead. A high value for b lowers the overhead but since fewer thread-blocks are created, the GPU has less resident thread-blocks to rotate through to hide memory latencies.

During the first pass, each thread-block sums each column and row of the assigned 2D input data towards the upper and right edge, respectively. The summed row and column data of each thread-block is collected into separate arrays. The second pass performs



[Fig. 3] The overall algorithmic view of the proposed three-pass summed area table. Each thread-block performs row and column-wise prefix sum on the sub-image in parallel in (a). The second pass is composed of prefix scanning the row-wise (b) and column-wise partial sums (c). Also, collecting, scanning, and propagating pivot values (d) is processed during the second pass. In the third pass, each thread-block references the row and column-wise partial sums from the adjacent thread-blocks to the left and lower thread-blocks to complete the whole computation.

prefix sum on the combined row and column data. In addition, pivot values which rest at the corners of where four adjacent thread-blocks exist are collected and prefix scanned. The resulting scanned data contain the row and column-wise sums up to every thread-block. The third pass scans the 2D input data which was originally assigned to a particular thread-block. Lastly, the result of the third pass is propagated with the results from the second pass to produce the final summed area table. The schematics are illustrated in [Fig. 2].

The first pass is comprised of a grid of 2D thread-blocks in $(w/b)\times(h/b)$ assignment operating in parallel. Each thread-block is allocated a $b\times b$ sub-image. The cooperative threads execute parallel prefix scan to compute the partial sum of the pixel values for individual rows and columns. The resulting

$2*b$ elements are stored in two separate 1D arrays (row- and column-wise partial sums arrays) for all thread-blocks. The array is logically partitioned into b elements per segment where the segments are ordered in the same consecutive order as the thread-block index number. For example, a thread-block at (x, y) engages writing to row-wise array starting at index $w*y+b*x$. Note that the partial sums array for the column-wise sums is also a 1D array so the global memory access can be coalesced.

On the second pass, a single thread-block is utilized since concurrency is only guaranteed within a single thread-block through barrier synchronization. First, the threads load the array for the row-wise partial sums created in the previous pass. Exclusive scan is performed on the array and stored back to the global memory. Afterwards, the column-wise partials sums array is processed similarly. The vital portion of the second pass is generating the pivot values which serve as the storage for the sum up to the lower left thread-block relative to the current thread-block position on the input image. The pivot values are acquired from the last values of the column-wise partial sums from each thread-block. These pivot values are then scanned and propagated to the column-wise sums which belong to the adjacent thread-block to the right.

The third pass facilitates multiple thread-blocks again. The pixel values for the allocated sub-image is loaded along with the $2b$ elements pertaining to the relative position of the thread-block in the row-wise and column-wise partial sums arrays. This time inclusive scan in both directions is performed on allocated sub-image and the partial sums are added accordingly. The overall algorithm is demonstrated in Figure 3 and the pseudo code is given in [Fig. 4].

The proposed method requires $w*h+2*w*h/b$ accesses for the first pass (reduction), $4*w*h/b$ accesses for the second pass (partial sum scan), and $2*w*h+2*w*h/b$ accesses for the

```
1.  procedure reduce
2.    for bid := 0 to blk_cnt in parallel do
3.      for tid := 0 to 256 in parallel do
4.        copy_data_from_global_memory();
5.        row_wise_prefix_scan();
6.        column_wise_prefix_scan();

7.        if warp_id == (warp_cnt - 1) then
8.          store_column_sum_to_global_memory();

9.        if warp_id == 0 then
10.         store_row_sum_to_global_memory();
11. end

12. procedure row_sum_scan
13.   for bid := 0 to (img_height / 256) in parallel do
14.     for tid := 0 to max_thd_cnt in parallel do
15.       copy_row_sum_from_global_memory();
16.       row_wise_prefix_scan_on_row_sum();
17.       store_scanned_row_sum_to_global_memory();
18. end

19. procedure col_sum_scan
20.   for tid := 0 to 256 in parallel do
21.     copy_col_sum_from_global_memory();
22.     col_wise_prefix_scan_on_col_sum();

23.     if tid % 32 == 31 then
24.       collect_pivot_value();

25.     if tid < (img_height / 32 - 1) then
26.       scan_pivot_values();

27.     if warp_id > 0 then
28.       propagate_scanned_pivot_value_to_col_sum();

29.     store_scanned_col_sum_to_global_memory();
30. end

31. procedure intra_scan
32.   for bid := 0 to blk_cnt in parallel do
33.     for tid := 0 to 256 in parallel do
34.       copy_data_from_global_memory();
35.       copy_row_col_sum_from_global_memory();
36.       row_wise_prefix_scan();
37.       column_wise_prefix_scan();
38.       store_data_to_global_memory();
39. end
```

[Fig. 4] Pseudo code for generating the summed area table for the proposed method. The method is composed of four phases or kernel calls, each executed in the given order.

third pass (scan). The total accumulates to $3*w*h+8*w*h/b$. Therefore, as long as $b>2$, the amount of global memory access is reduce compared to Harris' method.

The implementation of the proposed implementation algorithm is restricted by the amount of usable shared memory available on a SM. In order to support arbitrary sized sub-image, we fixed the concurrent internal processing task to 32x32 sub-image in order to minimize the memory footprint. Then, the assigned sub-image is processed in a patch 32x32 in size at a time recursively to reduce or scan the whole input sub-image.

The internal processing size of 32x32 was selected based on the shared memory available on the commodity GPU which ranges from 16KB to 48KB. More importantly, the prefix scan employed is work inefficient although the depth is optimal. Therefore, increasing the input data size when even more memory is available will hinder the execution performance with $O(nlogn)$ work complexity. On the other hand, decreasing the input data size below the SIMD data width of 32 specified as the warp size in the GPU will only cause parts of the data path to be idle and have no effect performance-wise.

The shared memory is divided into banks of equal size to concurrently supply data to the multiple cores with access demands without delay. However, memory bank conflict will cause the access to be serialized causing performance to degrade substantially. Therefore, we mapped the data into the shared memory so that every thread in any warp will be assigned to a different memory bank. This is accomplished by padding the actual image data with empty padded region to avoid shared memory bank conflict during the column-wise scans. Otherwise, every SM must endure maximum amount of serialization due to bank conflicts. Since the warp size is 32 threads, each shared memory access will be subject to 32 times the original latency of the memory.

As for the cooperative threads working on prefix scanning the 2D sub-image allotted to each thread-block, we employ Kogge-Stone style graph of which the depth is equal to the theoretical lower bound. However, Kogge-Stone style graphs require 2^d less computations as each depth of the graph is traversed. Thus, branching instructions must be employed to mask out illegal memory accesses. However, these instructions can be avoided by appending padding zones for useless threads to simply spill onto them. In addition, data is partitioned into segments equal to the SIMD width, therefore, barrier synchronization can also be removed.

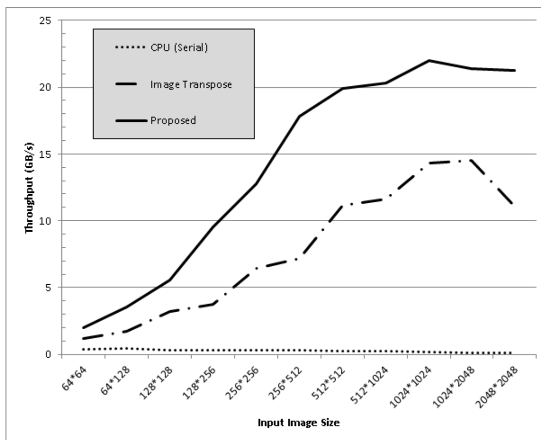
5. Experimental Results

The benchmark system was composed of AMD FX6100 CPU running at 3.3GHz with 8GB of memory. It was also equipped with a dedicated NVIDIA GeForce GTX580 with 3GB of device memory for undivided parallel processing. This hardware is equipped with 16 SMs each of which facilitates 32 SPs for a total of 512 cores. The memory subsystem is capable of transferring 192.4GB of data to those cores per second.

The measurement of the execution time was achieved with the help of the Visual Profiler

provided with the NVIDIA CUDA Toolkit. The tool supports measurements in microsecond accuracy and also produces useful statistics such as peak memory bandwidth, shared memory conflict percentage, and hardware occupancy.

The input image was generated with random pixel values to simulate arbitrary images and the color space was converted to gray-scale, because most applications of the summed area table uses this color space to either reduce memory/computation amount or application specific characteristics such as making edges more prominent in image processing applications.



[Fig. 5] Performance comparison between the serial CPU execution, image transpose method by Harris and the proposed method.

Since the image transpose method devised by Harris et el. is publically available on the web as part of the CUDPP library[11], we utilized it with minimal modifications for our benchmark. The input data, as stated above, has been reduced to a single channel and performance was measured using the NVIDIA Visual Profiler.

The experiment started out at an image size of 64x64 and was increased to 2048x2048. The image transpose method showed a noticeable performance drop past 1024x2048. This is due to the extra burden of splitting the input data into manageable size and combining the partial results to achieve the global summed area table. As expected, the experiments past 2048x2048 only showed results that had considerable gap between the throughput of the previous and the proposed method.

[Fig. 5] depicts the comparison results of the summed area table generated by serially executing on the CPU, the previous method of image transposing devised by Harris, and the proposed parallel thread-block based divide and conquer method. The serial CPU execution showed a steady throughput while the other two parallel implementations (previous and proposed) exhibited log based performance saturation as is known as the theoretical speedup of parallel machines.

The graph shows that the generation of the summed area table is an application of the GPU which is completely memory bound. Although the proposed method requires noticeably more computations during memory address calculations and thread assignment, the reduction in global memory access hides most of the computation overhead. In our tests, the proposed algorithm was up to 2.5 times faster in the small problem set (under 256x256 image size) and up to twice as fast compared to the previous method in the large problem set (over 256x256 image size).

6. Conclusion

We have presented a technique of generating summed-area table efficiently on GPGPU environment. By incorporating divide and conquer method, the access to the global memory was reduced by almost half compared to the previous methods. This is achieved by incorporating intermediate values which are transmitted between thread-blocks working on square sub-images. In addition, pivot values are also transmitted between the intermediate values so that each thread-block has the necessary information falling between the current sub-image and the starting point of the input image.

The benchmark tests executed on the high performance commodity GPU showed that efficient access pattern of the proposed system results in increased performance in generating the summed area table.

ACKNOWLEDGMENT

This research was supported by Basic Science Research Program through the National Research Foundation of Korea (NRF) funded by the Ministry of Education, Science, and Technology. (2011-No.2011-0027450)

REFERENCES

[1] Crow, F. C. “Summed-area tables for texture mapping,” In SIGGRAPH '84: Proceedings of the 11th annual conference on Computer graphics and interactive techniques, NY, NY, USA, pp 207-212, 1984.

[2] Heckbert, P. S., “Filtering by Repeated Integration,” ACM SIGGRAPH Computer Graphics, Vol. 20, No. 4, pp 315-321, 1986.

[3] Hensley, J., Scheuermann, T., Coombe, G., Singh, M., and Lastra, A. “Fast summed-area table generation and its applications,” Computer Graphics Forum, Vol. 24, No. 3, pp 547-555, Sept. 2005.

[4] Demers, J., “Depth of Field: A Survey of Techniques,” GPU Gems, Addison Wesley, pp 375-390, 2004.

[5] Grabner, M., Grabner, H., and Bischof, H., “Fast approximated SIFT,” ACCV 2006, LNCS, Vol. 3851, pp 918 - 927, 2006.

[6] Bay, H., Tuytelaars, T., and Gool, L. V., “SURF: Speeded Up Robust Features,” ECCV 2006, LNCS, Vol. 3951, pp 404-417, 2006.

[7] Harris, M., Sengupta, S., and Owens, J. D. “Parallel prefix sum (scan) with CUDA,” In Nguyen, H., ed., GPU Gems 3. Addison Wesley, 2007.

[8] NVIDIA CUDA C Programming Guide, Ver. 4.0, 2011.

[9] Harris, M., Sengupta, S., and Owens, J.D., “Parallel Prefix Sum (Scan) with CUDA,” GPU Gems 3, H. Nguyen, Addison-Wesley, Ch. 31, Aug. 2007.

[10] Kogge, P. M. and Stone, S. S., “A Parallel Algorithm for the Efficient Solution of a General Class of Recurrence Equations,” IEEE Trans. on Computers, Vol. C-22, No. 8, pp 786-793, 1973.

[11] CUDA Data Parallel Primitives Library, http://code.google.com/p/cudpp

하 상 원 (Ha, Sang-Won)

2003 연세대학교 기계전자공학부 정보산업전공 학사
2006 연세대학교 컴퓨터과학과 석사
2006-현재 연세대학교 컴퓨터과학과 박사과정

관심분야 : 컴퓨터 그래픽스 SW/HW, GPGPU, 병렬처리 알고리즘

김 진 우 (Kim, Jinwoo)

2006 상명대학교 소프트웨어전공 학사
2007-현재 연세대학교 컴퓨터과학과 석박통합과정

관심분야 : 컴퓨터 그래픽스 SW/HW, GPGPU, 렌더링 알고리즘

최 문 희 (Choi, Moon-Hee)

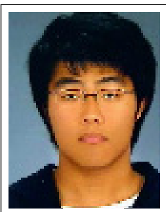
1999 덕성여자대학교 전산학과 학사
2001 연세대학교 컴퓨터과학과 석사
2007 연세대학교 컴퓨터과학과 박사
2007-현재 삼성전자 무선사업부 책임연구원

관심분야 : 모바일 그래픽스 SW/HW, 모바일 GPU

변 혜 란 (Byun, Hyeran)

1980 연세대학교 수학과 학사
1983 연세대학교 수학과 석사
1993 Purdue Univ. 컴퓨터과학과 박사
1995-현재 연세대학교 컴퓨터과학과 교수

관심분야 : 패턴 인식, 영상 처리, 영상 인식



전 태 준 (Jun, Tae-Joon)

2009-현재 연세대학교 컴퓨터과학과 재학

관심분야 : GPGPU, 데이터 마이닝

한 탁 돈 (Han, Tack-Don)

1978 연세대학교 전자공학과 학사
1983 Wayne State Univ. 석사
1987 Univ. of Massachusetts at Amherst 컴퓨터공학과 박사
1989-현재 연세대학교 컴퓨터과학과 교수

관심분야 : 고성능 컴퓨터 구조, 미디어 시스템 구조, HCI(Human Computer Interface), 유비쿼터스 컴퓨팅