Contemporary Engineering Sciences, Vol. 7, 2014, no. 24, 1415 - 1423
HIKARI Ltd, www.m-hikari.com
http://dx.doi.org/10.12988/ces.2014.49174

Improved Integral Histogram Algorithm for Big Sized Images in CUDA Environment

Chang Won Lee and Tae-Young Choe

Department of Computer Engineering, Kumoh national Institute of Technology, 61 Daehak-ro, Gumi, Gyeongbuk 730-701, Korea

Abstract

Although integral histogram enables histogram computation of a sub-area within constant time, construction of the integral histogram requires $O(nm)$ steps for $n \times m$ sized image. Such construction time can be reduced using parallel prefix sum algorithm. Mark Harris proposed an efficient parallel prefix sum and implemented it using CUDA GPGPU. Mark Harris' algorithm has two problems: leakage of shared memory and inefficiency against big sized images. In this paper, we propose a parallel prefix sum algorithm that prevents the leakage and deals big sized images efficiently. Our proposed algorithm corrects the memory leakage using exact indexing against bank conflicts and eliminates inefficient global memory access by allocating multiple pixels to each thread. As the result, average execution time of the proposed algorithm ranges 95.6% ~ 101.9% compared to that of Harris' algorithm.

Keywords: Integral histogram, Parallel prefix sum, CUDA shared memory

1 Introduction

Histogram is widely used in computer vision or image processing areas. Histogram is a graphical representation of data such as frequencies of pixel values. In a gray image where pixel values range from zero to 255, one pixel is represented as a byte. The histogram of the image is contained in an array with size

256, where each element contains the number of pixels that have the value or the gray color. Some applications need to retrieve histograms of rectangular sub-regions. Object recognition and object tracking [1] are the examples of the applications. They select multiple rectangular sub-regions in an image and compute histograms of the sub-regions. Traditional histogram computation takes $O(nm)$ steps for $n \times m$ sized rectangular region. Such time complexity is heavy burden to the applications that compute histograms of many sub-regions.

Porikli proposed new concept called *integral histogram* in order to reduce the burden [2]. The integral histogram of a pixel (x, y) is the histogram of rectangular sub-region that has two vertices: leftmost top pixel (1, 1) and (x, y). In order to compute histogram of a sub-region, integral histograms of all pixels should be computed in the initialization time. After the initialization, histogram computation of a sub-region takes constant steps. Although the integral histogram reduces the time complexity of histogram computation, construction of all integral histograms requires $O(n'm')$ steps for $n' \times m'$ sized rectangular region. Parallel construction of the integral histograms is an efficient method that reduces the initialization time. Before designing and implementing a parallel version of the integral histogram construction, we need to select one among various parallel frameworks like Cloud computing system, cluster system, parallel computer system, or GPGPU system. In the paper, GPGPU system is selected because many image-processing applications are interactive with fast response time provided by GPGPU.

Mark Harris proposed a parallel prefix sum algorithm used for integral histogram construction in CUDA GPGPU [3]. Given an array *a* [1..*n*], prefix sum *ps* [1..*n*] is constructed as *ps* [*k*] = $\sum_{i=1}^{k} a[i]$. A parallel construction of integral histogram uses the parallel prefix sum algorithm as follows: the constructor allocates a line into a block that allocates a pixel into a thread, applies the parallel partial sum algorithm on each block, transposes lines and columns, allocates and applies the partial sum algorithm again, and finishes the whole operation by transposing the image again. Such allocation uses shared memories in order to make partial sum on each line. A block is a set of threads and each thread takes charge of two pixels. Thus, there should be an efficient communication method between threads. Harris' algorithm uses *shared memory* for the communication in CUDA. Each block takes a line into a shared memory accessible by threads of the block in parallel. Although Harris' algorithm maximizes degree of parallelism and manipulates shared memory efficiently, it has two problems in the big sized images management and memory indexing. Firstly, since Harris' algorithm maps two pixels to a thread, basic algorithm handles 2*n* sized images at maximum if the maximum number of threads in a block is *n*. In order to deal with images larger than 2*n*, the algorithm allocates multiple blocks to each line. Such allocation leads to sequential global memory access in order to communication between the blocks

in a line. Secondary, Harris' algorithm changes pixel indices in shared memory in order to solve bank conflict. Unfortunately, the changed index method leaks shared memory.

In this paper, we propose an improved parallel construction algorithm of integral histogram that eliminates the shared memory leakage and reduces the number of the global memory accesses. The shared memory leakage is remedied by an improved indexing method and the global memory access is reduced by mapping multiple pixels into a thread. The remainder of this paper is organized as follows: Section 2 presents review of Harris' algorithm. The proposed algorithm is explained in Section 3. Section 4 gives experimental results. Finally, Section 5 concludes.

2 Previous Works and Review of Harris' Algorithm

In the simple parallel prefix sum, thread t_i reads two bins $a[i-2^{s-1}]$ and $a[i]$, adds them, and stores the result to $a[i]$ at the s^{th} step for $1 \le s \le \lceil \log m \rceil$ where *m* is the length of a line [4], [5]. An array of the bins is stored in the shared memory of a block and any shared memory is composed of 16 or 32 banks in CUDA. When two threads concurrently access memory locations in the same bank, the situation is called *bank conflict*. For example, assume that a shared memory has 16 banks. If thread t_i accesses a shared memory location *i* in bank *k* and thread t_{i+16} tries to access another memory location *i*+16 in the same bank *k*, they cause bank conflict. Bank conflict forces sequential access and degrades concurrency as the result.

Sengupta et al. implemented segmented scan in CUDA [6]. The algorithm uses head flag in order to divide entire array to multiple different sized blocks. It is possible for each block to apply different operations. Mark Harris proposed a parallel prefix sum algorithm that reduces bank conflict [3]. In order to reduce the bank conflict, above algorithms consists of two phases: an up-sweep phase and a down-sweep phase. The up-sweep phase is similar to bottom up tree traverse such as the value of a node is added to its right sibling node recursively until root node. The down-sweep phase exchanges and adds values of elements in order to complete prefix sum.

Harris' algorithm reduces bank conflict more by adding a padding on index. The algorithm adds a padding to local memory index given an index *i* as follows:

$$pad(i) = \lfloor i/b_n \rfloor, \tag{1}$$

where b_n is the number of banks in the shared memory. Thread t_i accesses

shared memory location $i + pad(i)$. For example, when thread t_{10} accesses memory location 10 in bank 10, thread t_{26} accesses memory location 27 (= 26 + $\lfloor 26/16 \rfloor$) in bank 11. Thus, the padding reduces bank conflict.

Another problem is that the simple parallel prefix sum is not scalable to big sized images. In NVIDIA graphic cards a block can contains maximum 512 or 1024 threads according to compute capability. Since a thread covers two pixels in Harris' algorithm, a block cannot deal with images that exceed 2048×2048. In order to solve the problem, Harris' algorithm divides a large line into multiple sub-lines. Each sub-line is stored in a shared memory of a block and threads in the block compute prefix sum. Next, the largest values of blocks are gathered in global memory and prefix sum is computed on them. The prefix sums are returned to their blocks and are added to elements in the block. Another algorithm proposed by Sengupta et al. divides an array into multiple blocks with warp size in order to maximize warp occupancy [7].

The disadvantage of Harris' algorithm is the overhead occurred by big images. Gathering the largest values, computing prefix sum in global memory, and redistribution access global memory. Since such overhead occurs on every lines, they should wait to access global memory sequentially.

3 Proposed Algorithms

3.1 Parallel Prefix Sum on a Large Image

It is noticed that the amount of shared memory for a block can accommodate a line of big image like 4K video frame. For example, NVDIA GTX670 allocates 48Kbytes to a block [8]. Thus, the size of shared memory does not restrict prefix sum computation within a block. Only the maximum number of threads in a block limits size of image.

We solve the problem by allocating four or more pixels to a thread instead of using multiple blocks to cover a line. Let l be the length of a line and N_t be the maximum number of available threads in a block. If $l \le 2N_t$, the number of threads to be used is $l/2$ and the algorithm works as same as Harris' algorithm does. Otherwise, the number of threads to be used is N_t. Let integer Q satisfy range $2^Q N_t < l \le 2^{Q+1} N_t$. Thre $t_i (0 \le i < N_t)$ covers pixel index $2^{Q+1} i \sim 2^{Q+1}(i+1) - 1$. The number of step increases up to Q.

3.2 Fully Utilized Shared Memory Access

As previously stated, Harris' algorithm reduces bank conflict by index padding. Padding is computed and is used as follows: pixel $a[i]$ in global memory is mapped to shared memory $as[\,]$ using Equation (1) as follows:

$$as[i + \lfloor i/b_n \rfloor] = a[i]. \tag{2}$$

Fig 1 shows an example that a line with 64 elements in the global memory is allocated to a shared memory with padding where the number of banks is 16. Because of the padding method, there are unused elements notated as NULL, which leak memory or cause buffer overflow. The amount of the leaked memory is $\lfloor n/b_n \rfloor - 1$ where *n* is the length of a line.

5th	61	62	63													
4th	46	47	NULL	48	49	50	51	52	53	54	55	56	57	58	59	60
3rd	31	NULL	32	33	34	35	36	37	38	39	40	41	42	43	44	45
2nd	NULL	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30
1st	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Fig. 1. Shared memory structure not fully utilized (NULL is empty element)

In order to utilize shared memory fully and to minimize memory leak, we propose memory mapping of global index *i* to local index i_l as following equation:

$$i_l = i + \lfloor i/b_n \rfloor - b_n\left(\left\lfloor \frac{i+\lfloor i/b_n \rfloor}{b_n} \right\rfloor - \lfloor i/b_n \rfloor\right) \tag{3}$$

The second term in the right side is the padding proposed by Harris. The third term decides whether the index moves down one line. If an index is wrapped around and moves to the upper line, expression $\left\lfloor \frac{i+\lfloor i/b_n \rfloor}{b_n} \right\rfloor - \lfloor i/b_n \rfloor$ is 1. Otherwise, the expression is 0. For example, indices 31, 46, or 47 wrap around as in Fig 1, and they move down one line by the third term. Such movements fill NULL elements. The mapping equation (3) is expressed as pseudo codes as follows:

```
1: s_idx = g_idx
2: check = s_idx / b_n
3: s_idx = s_idx + check
4: s_idx = s_idx – b_n * (s_idx / b_n – check)
5: shared[s_idx] = global[g_idx]
```

Fig 2 shows mapping of global indices to local memory. Indices upon the NULL elements in Fig 1 moved down, and they fill all NULL elements.

4th	61	62	63	48	49	50	51	52	53	54	55	56	57	58	59	60
3rd	46	47	32	33	34	35	36	37	38	39	40	41	42	43	44	45
2nd	31	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30
1st	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Fig. 2. Shared memory structure using proposed algorithm

4 Performance Analysis

4.1 Experimental Environment

Harris' algorithm and our proposed algorithm are implemented in C programming language using CUDA. Two algorithms are compiled in Visual Studio 2010 with CUDA version 5.5. Experimental environment is Windows 7 operating system, Intel core i5 750 (2.67GHz), 8Gbytes main memory, and NVDIA GeForce GTX 670 graphics card.

4.2 Performance Analysis

Fig 3 shows performances of Harris' algorithm and its improvement by allocating multiple pixels to each thread and one line to each block for large images as shown in Section 3.1. In the Figure, 'Multiple' label means the improved version. Four types of images are used for comparisons: FHD (1920×1080), QHD (2560×1440), UHD (3840×2160), and DCI-4K (4096×2160). Two algorithms run 10 times. Since GTX 670 graphic card allows a block to contain 1024 threads, FHD image does not require multiple block manipulation of Harris' algorithm. As the result, the performances of the two algorithms are almost the same in the case of FHD image. The performance of the proposed algorithm passes ahead that of the Harris' algorithm from QHD image because Harris' algorithm should process two blocks and access global memory. The relative execution time of the proposed algorithm against Harris' algorithm is 94.7% when DCI-4K image is processed.

Fig 4 shows performances of Harris' algorithm and its improvement by resolving memory leak as explained in Section 3.2. In the Figure, 'Shared' label means the improved version. Since the improved version just added 5 operations when threads access to shared memory, execution time is almost same or little

degraded. The highest relative execution time is 100.8% when FHD image is processed. The lowest relative execution time is 99.6% when QHD image is processed.

Fig 5 shows performance comparison Harris' algorithm and proposed algorithm that improves two points: deleting global memory access and preventing memory leak. The proposed algorithm shows that the relative execution time is 95.5% compared with Harris' algorithm when DCI-4K image is processed.

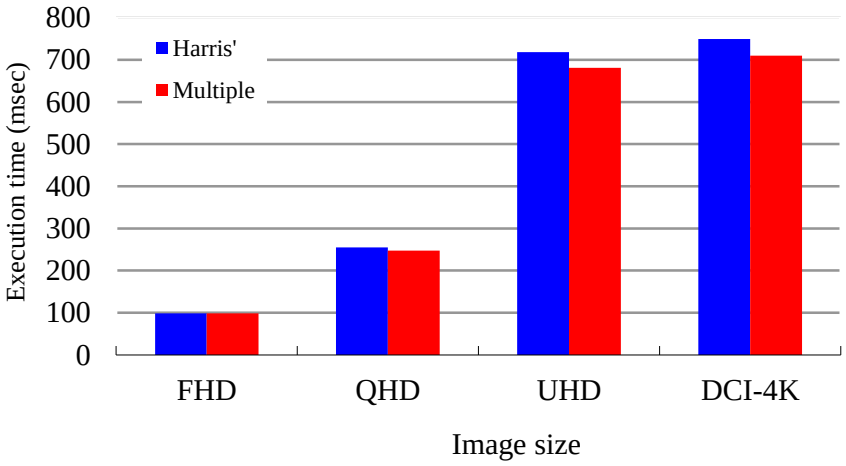

Fig. 3. Performance comparison Harris' algorithm and the improved version that allocates multiple pixels to each thread

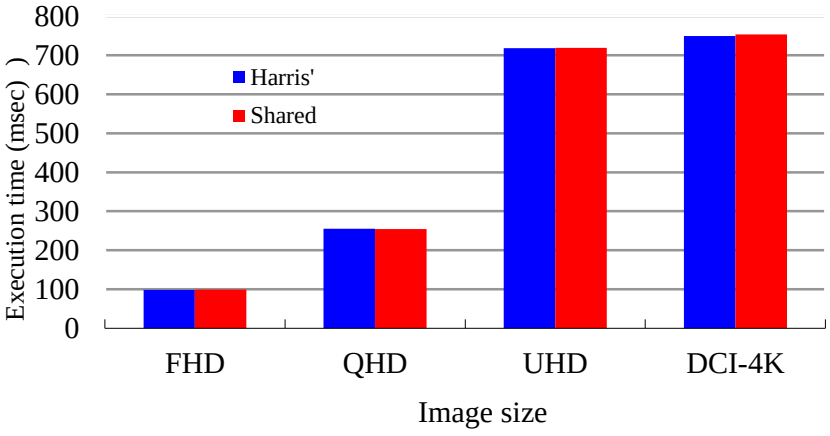

Fig. 4. Performance comparison Harris' and the improved version that resolves memory leak

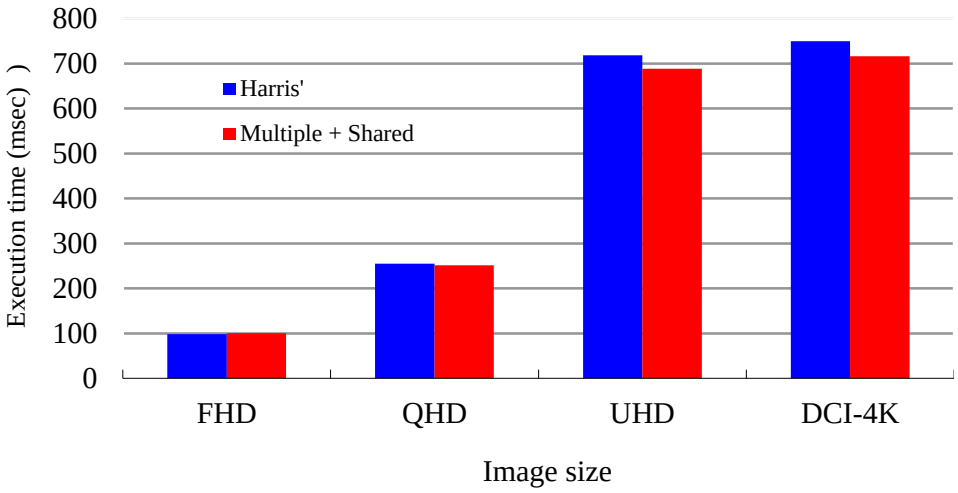

Fig. 5. Performance comparison Harris' algorithm and proposed algorithm

5 Conclusion

In this paper, we propose an improved construction algorithm for integral histogram. The proposed algorithm efficiently deals with big sized image and prevents shared memory leak. The proposed algorithm eliminates global memory access and maximizes the parallelism within threads in a block. Furthermore, it eliminates shared memory leak by adjusting padding when threads access to shared memory. The proposed algorithm shows 95.6% ~ 101.9% execution time compared to Harris' algorithm.

References

[1] A. Adam, E. Rivlin, and I. Shimshoni, Robust fragments-based tracking using the integral histogram, *IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, **1** (2006), 798 – 805.

[2] F. Porikli, Integral Histogram: A FastWay to Extract Histograms in Cartesian Spaces, *Proceedings of the 2005 IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, **1** (2005), 829 – 836.

[3] M. Harris, S. Sengupta, and J.D. Owens, Parallel Prefix Sum (Scan) with CUDA, *GPU gems*, **3(39)** (2007), 851 – 876.

[4] W.D. Hills and G.L. Steele JR., Data Parallel Algorithms, *Communications of the ACM*, **29(12)** (1986), 1170 – 1183, ACM Press.

[5] D. Horn, Stream reduction operations for GPGPU applications, *GPU Gems*, **2(36)** (2005), 573 – 589, Addison Wesley.

[6] S. Sengupta, M. Harris, Y. Zhang, and J.D. Owens, Scan Primitives for GPU Computing, *Graphics Hardware* (2007), 1 – 11, ACM.

[7] Y. Dotsenko, N.K. Govindaraju, P.P. Sloan, C. Boyd, and J. Manferdelli, Fast scan algorithms on graphics processors, *In Proceedings of the 22nd annual international conference on Supercomputing* (2008), 205 – 213, ACM.

[8] M. Silberstein, A. Schuster, D. Geiger, A. Patney, and J.D. Owens, Efficient computation of sum-products on GPUs through software-managed cache, *In Proceedings of the 22nd annual international conference on Supercomputing* (2008, June), 309 – 318, ACM.

Received: August 31, 2014