http://ansinet.com/itj

ITJ

ISSN 1812-5638

INFORMATION TECHNOLOGY JOURNAL

ANSInet
Asian Network for Scientific Information
308 Lasani Town, Sargodha Road, Faisalabad - Pakistan

Information Technology Journal 12 (20): 5440-5446, 2013
ISSN 1812-5638 / DOI: 10.3923/itj.2013.5440.5446

GPGPU-accelerated Parallelization Practice and Analysis for Image Segmentation Methods

[1]Guo He, [2]Wang Yu-Xin, [1]Feng Zhen, [1]Yu Yu-Long, [1]Jia Qi, [1]Wang Yuan-Yuan,
[1]Liu Yao, [1]Zhang Li-Jie and [1]Hou Yi-Ting
[1]School of Software Technology, Dalian University of Technology, 116620, Dalian, China
[2]School of Computer Science and Technology, Dalian University of Technology, 116624, Dalian, China

Abstract: Image segmentation is an important issue in the field of computer vision, it serves as a bridge linking the basic image processing methods to the high-level semantic recognition methods. With the increasing applications of the image segmentation methods in the modern industries, such as the defect detection in the production lines, the real-time requirements are greatly raised. Recently, with the advent of the General-purpose Graphics Processing Unit (GPGPU) platform, the parallelized implementations on this new platform open a new way to accelerate the image segmentation methods to meet the real-time requirements. In this work, various methods are analyzed and parallelized on the GPGPU platform, the horizontal comparisons are made to evaluate the potentials of parallelization for different segmentation methods. The parallelization strategies are performed on two levels: on the algorithm development level and on the program development level. It is expected that this investigation may provide guidance to the future parallelization tasks for the more advanced image segmentation methods and other computer vision applications.

Key words: Computer vision, image segmentation, parallel computing, software refactoring, GPGPU, CUDA

INTRODUCTION

From the basic image processing scope to the advanced semantic recognition scope, the image segmentation methods can be classified into three levels (Gonzalez and Woods, 2002). The first (basic) level includes some fundamental operations such as the filter-based edge detection method and the Hough transform-based line detection method. The second (middle) level includes more sophisticated methods such as the threshold-based region segmentation method and the region growing-based method. The third (advanced) level contains more semantic information, such as the tight-fitting oriented bounding box method. The wide applications of the image segmentation methods in industries raise the real-time requirements. In the recent years, with the development of the General-purpose Graphics Processing Unit (GPGPU), a new software refactoring strategy to parallelize the image segmentation methods on this new platform becomes a hot topic. In the published literatures, several segmentation methods have been implemented on the GPGPU platform (Van den Braak *et al*., 2011; Harris, 2007; Jorgensen *et al*., 2010; Kolawole and Tavakkoli, 2011; Luo and Duraiswami, 2008; Podlozhnyuk, 2007). The analysis, investigation and horizontal comparison for the parallelization practice of different segmentation methods will provide beneficial reference and guidance for the parallelization implementations of the current and future algorithms. However, as far as we know, there is no published literature to do this job. In this work, we make an effort to fill this gap. Two contributions are made in this work. First, varieties of image segmentation methods were parallelized and a horizontal comparison was made according to their interior parallelism and the achieved speedups in the experiments. Second, the parallelization strategies are analyzed and summarized on the algorithm development level and on the program development level. It is expected that this work may provide guidance to the future parallelized refactoring tasks for the more advanced image segmentation methods.

PARALLELIZATION PRACTICE FOR SELECTED METHODS

In the broad sense, various methods are classified as the image segmentation methods, from the basic edge detection to the region division to the advanced bounding box. In this section, we will describe how to parallelize them on the GPGPU platform. In our experiments, we implemented both the sequential and parallel methods on a Ubuntu system with the Intel Core

Corresponding Author: Guo He, School of Software Technology, Dalian University of Technology, 116620, Dalian, China

Table 1: Runtimes for the edge detection methods

Platform	256×256	512×512	1024×1024	2048×2048
CPU	2.140 ms	8.63 ms	35.04 ms	229.98 ms
GPGPU	0.030 ms	0.10 ms	0.380 ms	1.4700 ms
Speedup	71.33	86.3	92.21	156.45

i7 2600 CPU, 12 GB DDR3 memory and an NVIDIA Tesla C2075 GPGPU card. The sequential methods run on the CPU platform; for the parallel versions, the parallelized parts of the code run on the GPGPU platform and the CUDA technique was used for the software refactoring from the sequential code to the parallel code.

Filter-based edge detection: The filter-based edge detection method aims at identifying pixel points at which the image brightness changes sharply, then these points are organized into edges. This method can be implemented by firstly filtering an image with a 3×3 horizontal kernel; then, the image is filtered again using a 3×3 vertical kernel; thirdly, the pixel values of each filtered image are squared, the two results are added and their square root is computed; finally the square root image is threshold to get the edge image.

The filtering computation and the threshold computation take most of the running time. And also, they are quite suitable for parallelization. For the filtering computation, each thread is assigned to process one pixel point. The filtered result of one pixel point needs to fetch the eight neighboring pixel values. The adjacent threads of the same warp access overlapping pixel values from the global memory, therefore, this is beneficial to fetch the neighboring pixel values into the shared memory to achieve high memory bandwidth for concurrent accesses and reduce the data access from the global memory. To further optimize the parallelization, it can be adjusted that one thread processes multiple pixel points of the same columns, the addressing computations of the threads could be simplified and the access to the global memory is still coalesced.

Figure 1 shows the edge detection results of a test image. The CPU program and the GPGPU program got the same edge image, as shown in the right column of Fig. 1. The test image was resized from 256×256, 512×512, 1024×1024 to 2048×2048. The runtimes for the different sizes were recorded for both the CPU program and the GPGPU program, as shown in Table 1.

Hough transform-based line detection: Ideally, the edge detection methods discussed above should yield pixels lying on the edges. However, for some images, due to the noises or non-uniform illustration circumstances, some noise points are mistaken as edge pixels and the real edges are broken into several segmentations because of

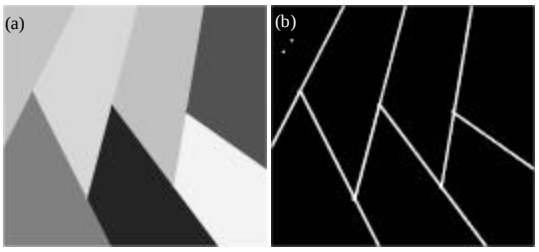

Fig. 1(a-b): Edge detection results. Left column: Original image. Right column: Edge image

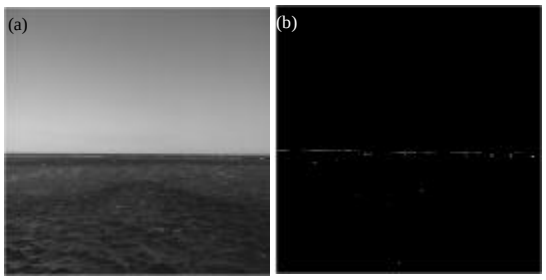

Fig. 2(a-b): Broken edges after the filter-based edge detection methods. Left column: the test sea image. Right column: the edge image with broken edges in the skyline and noises of the sea waves

the intensity discontinuities. An broken example is shown in Fig. 2, given a sea image, the edge detector got the broken skylines and some noise points of the sea waves. One way to eliminate the noises and link the broken edges is the Hough transform-based line detection.

The Hough transform theory transforms the point (x_i, y_i) into the parameter space (θ,ρ). For points (x_i, y_i) and (x_j, y_j) on the same line, they share the same (θ, ρ) values as:

$$x_i\cos\theta + y_i\sin\theta = \rho$$

$$y_j\cos\theta + y_j\sin\theta = \rho$$

This observation gives the inclination that a complete edge can be detected by the so-called accumulator matrix. In the accumulator matrix, the (θ, ρ) parameter space is divided into grids. Given a candidate edge point (x_k, y_k), for any θ value the corresponding ρ can be computed as: $\rho = x_k\cos\theta + y_k\sin\theta$. Therefore, the elements with the largest values in the accumulator matrix indicate a line with the parameters (θ, ρ).

The most time-consuming part of the Hough transform-based method is the computation of the accumulator matrix. Each thread is assigned to process

Table 2: Runtimes for the Hough transform-based line detection methods

Platform	256×256	512×512	1024×1024	2048×2048
CPU	3.20 ms	19.71 ms	27.07 ms	79.34 ms
GPGPU	3.15 ms	6.080 ms	10.43 ms	21.00 ms
Speedup	1.02	3.240	2.600	3.780

Table 3: Runtimes for the threshold-based segmentation methods

Platform	256×256	512×512	1024×1024	2048×2048
CPU	0.44 ms	1.53 ms	6.57 ms	83.43 ms
GPGPU	0.39 ms	0.45 ms	0.87 ms	2.840 ms
Speedup	1.13	3.40	7.55	29.38

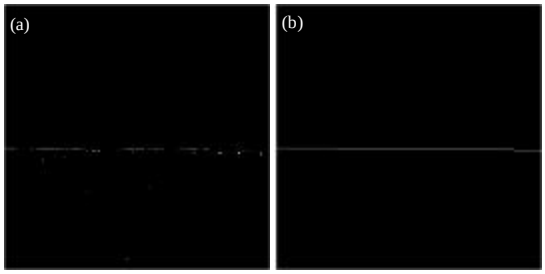

Fig. 3: Detected line using the Hough transform-based method. Left column: Edges detected by the filter-based method. Right column: A complete line is detected and the sea wave noises are erased.

one candidate edge point, the θ value increases from 0 to π, the corresponding ρ was computed and rounded off. Then the value in the θ-th column, ρ-th row increases by one. Multiple threads may access the same position of the accumulator matrix, therefore the atomic function atomicAdd is used. The usage of atomic functions will reduce the execution efficiencies, however, in some cases, they must be used to obtain the right answers. The next time-consuming part is to find the local peak values in the accumulator matrix. By keeping the local peak values and suppressing the neighboring values, the overlapping lines are eliminated to get a single line. To find the local peek values, each thread needs to access the neighboring nine pixels. The adjacent threads access the overlapping pixels, therefore, the shared memory should be used to reduce redundant loads from the global memory.

Figure 3 shows the detected line using the Hough transform-based method. The overlapping and broken skylines are constrained into one line and the noises of the sea waves are eliminated. Table 2 compares the runtimes of the sequential method on the CPU platform and the parallel method on the GPGPU platform.

Threshold-based region segmentation: The threshold-based method divides an image based on whether the pixel values are larger or smaller than an given threshold value. This method is quite suitable for images with strong discontinuities between adjacent regions and all the regions have similar sizes. The key for this method is to find an appropriate threshold value to correctly divide the image into regions. One intuitional way to find the threshold value is by generating the histogram of the image to observe the distributions of gray values. For each thread block, a shared memory with the size of 256 is allocated to record the sub-histogram computed by the thread block. The addition operation on the buckets of the sub-histogram on the shared memory is atomic, therefore, the atomic function has to be used. The function atomicAdd is quite time-consuming, in this work, a strategy is used to minimize the invoking of the atomic function: a thread is assigned to process 16 pixels of the same column, if the 16 pixel values are the same, then the atomicAdd function can only be invoked 1 times instead of 16 times; if the 16 pixel values are different, the invocation numbers can also be reduced by postponing the function invocation to the time the first different pixel value appears. This strategy is illustrated in the following pseudo code.

```
// The postponing strategy to reduce the invocations of the atomic functions
int inindex = threadIdx.y * blockDim.x + threadIdx.x; // Get the index of the thread
int threadnumber = 0; // the number processed by the thread, be limited to be 16
int counter = 0; // counter to record the number of the same pixel values
do {
int cur_gray = img[r][c]; // get the current gray value
// The postponing strategy to reduce the invocations of the atomic functions
int inindex = threadIdx.y * blockDim.x + threadIdx.x; // Get the index of the thread
int threadnumber = 0; // the number processed by the thread, be limited to be 16
int counter = 0; // counter to record the number of the same pixel values
do {
int cur_gray = img[r][c]; // get the current gray value
int nxt_gray = img[r+1][c]; // get the next gray value, from the next row
if(cur_gray == nxt_gray)
counter ++; // the invoking of atomic function is postponed
else {
atomicAdd(andhistogram[cur_gray], counter); // invoke the atomic function
counter = 0;
cur_gray = nxt_gray;
}
threadnumber ++;
} while(threadnumber < 16)
```

Figure 4 shows the histogram of the target image, the threshold value is determined as 100 and the segmentation image is obtained herein. Table 3 compares the runtimes for the threshold-based methods on different platforms.

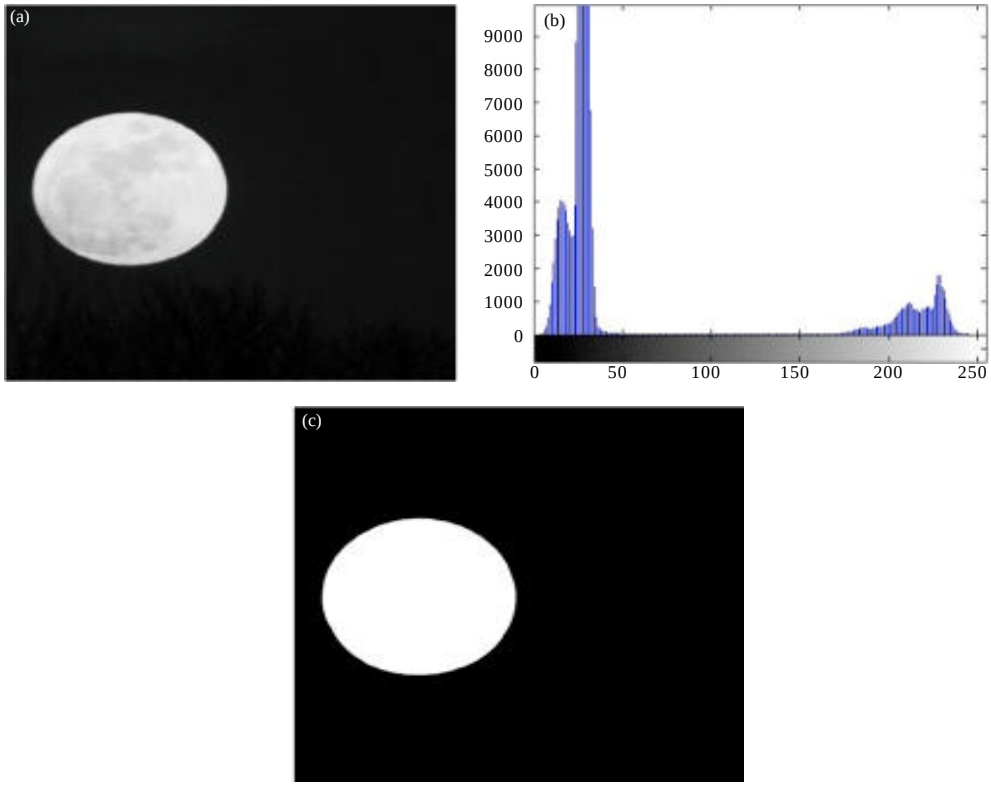

Fig. 4(a-c): Threshold-base segmentation method. Left column: the target image. Middle column: Generated histogram. Right column: Resulted segmentation image

Region growing-based method: The region growing-based method begins with a set of seed points, the seed points search around their neighboring pixels to see if they meet the predefined requirements, if yes the neighboring pixels will be set as the new seeds. The growing process ends when all the pixels satisfying the requirements are merged into regions. This method is suitable for the images in which the target regions are relatively small compared with the whole images, such as the lighting region in the raining night sky image.

The sequential method is implemented by the stack data structure. First, the seed points are pushed into the stack. Second, the seed point in the top of the stack is popped, the neighboring pixels of this seed point is accessed to see if it meets the requirements, if yes, the neighboring seed points are pushed into the stack. The second step repeats until the stack is empty. A flag is set to ensure seeds popped from the stack will not be pushed into the stack again.

The parallel method is implemented by iterative merging in image blocks. First, the image is divided into blocks, each block is processed by a thread block. Second, the thread block allocates a shared memory with the same size of the image block, the data is loaded from the image block to the shared memory. Third, one thread is assigned to process one seed point in the shared memory, the neighboring points satisfying the requirements are set as the new seed points. The third step repeats until all the left neighboring points of all the seeds cannot meet the requirements. A demonstration for the iteration is shown in Fig. 5. Lastly, the data is written back from the shared memory to the global memory, since the different pixels in the shared memory corresponds to the different pixels in the global memory, the atomic functions are not needed in this written-back process. In this parallelized strategy, shared memory is used to perform the region growing process, pixels are repeatedly read and written in the shared memory instead of the global memory, in this way, the data access time is greatly saved.

Figure 6 shows the segmentation results of a lighting image, the pixels with value 255 are set as the seed points and the pixels with the values larger than 225 are set as the candidate points. Table 4 shows the observed runtimes for the sequential and parallel programs with different problem sizes.

Table 4: Runtimes for the region growing methods

Platform	256×256	512×512	1024×1024	2048×2048
CPU	0.59 ms	2.51 ms	15.82 ms	65.72 ms
GPGPU	0.35 ms	0.76 ms	1.890 ms	5.100 ms
Speedup	1.71	3.30	8.370	12.89

Table 5: Runtimes for the tight-fitting oriented bounding box methods

Platform	256×256	512×512	1024×1024	2048×2048
CPU	0.68 ms	2.670 ms	10.58 ms	42.06 ms
GPGPU	0.24 ms	0.250 ms	0.250 ms	0.510 ms
Speedup	2.83	10.68	43.32	82.47

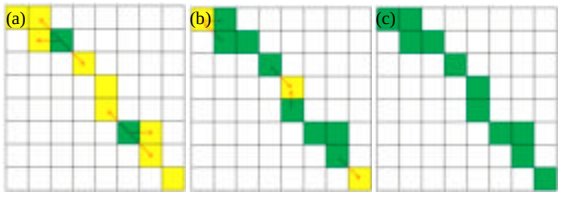

Fig. 5: Iteration processes for region growing. The green points are set as the seeds and the yellow points are set as the candidate points. The red arrows show the merging directions in the iterations, (a) Iteration 1, (b) Iteration 2 and (c) Iteration 3

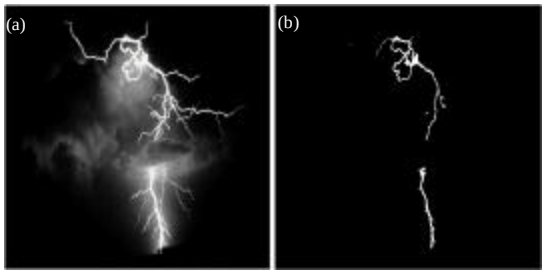

Fig. 6(a-b): Results for the region growing methods. Left column: A lighting in the night sky. Right column: Obtained brightest parts of the lighting

Tight-fitting oriented bounding box method: In this method, the input image is a binary map in which the points with the value 0 indicate the background area and the points with the value 1 indicate the target object with the irregular shape. The output is a bounding box to tightly surround the irregularly shaped object. Usually, the semantic information of the whole images concentrates on the bounding box region and also with the bounding box, more sophisticated tasks can be coped with, such as the collision detection problems (Zhang and Kim, 2007).

In the literature (Gottschalk *et al*., 1996), a method was proposed to obtain the tight-fitting oriented bounding box by computing the covariance matrix of the foreground pixels in the binary map.. From the covariance

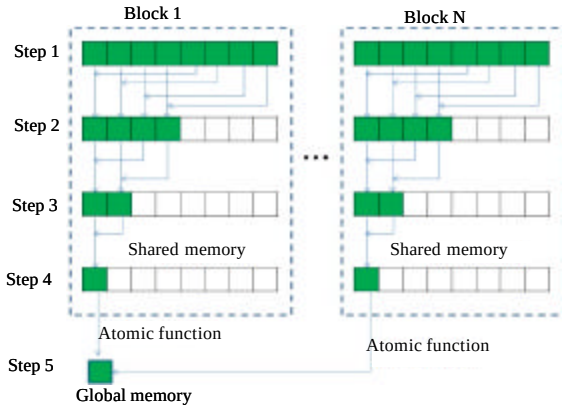

Fig. 7): Reduction process in the parallelized implementation

matrix, the largest eigenvector can be generated, which reveals the principal orientation of the bounding box. With the knowledge of the principal orientation, the four corner points of the tight-fitting oriented bounding box can be computed.

The most time-consuming parts are the computations of the center point and the covariance matrix. These two computations can be classified as accumulation computations. In this work, the accumulation is implemented by the reduction computation. Several strategies have been used to parallelize the reduction computation. First, one thread is assigned to access 32 pixels of the same columns. Second, the shared memory with the size of the block size is allocated, each entry in the shared memory stores the information of 32 pixels acquired by the same threads. Third, the information in the thread blocks are summarized by reducing the sizes of available shared memory into half in each step, the sequential addressing should be guaranteed in the process. Finally, the information of the pixels in one block is summarized into the first entries of the blocks and then written into the global memory with the atomic functions. The reduction process can be illustrated in Fig. 7.

Figure 8 shows the obtained tight-fitting oriented bounding box of a banana image. The box's orientation is consistent with the banana's orientation. Tab.5 shows the runtimes of the proposed method on the CPU platform and the GPGPU platform.

HORIZONTAL COMPARISONS BETWEEN DIFFERENT METHODS

The filter-based edge detection method is the most suitable for parallelization. In this method, the data

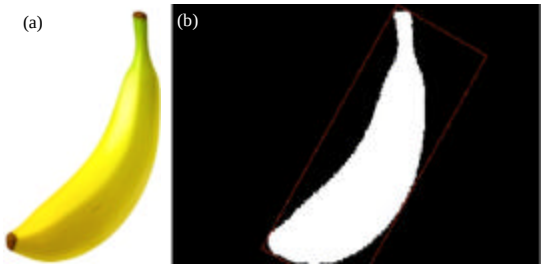

Fig. 8(a-b): Tight-fitting oriented bounding box method. Left: Banana image. Right column: Resulted bounding box colored in red

correlations between different threads are minimized. The tight-fitting oriented bounding box method got the second highest speedups. In this method, the computations to get the center point and the covariance matrix can be implemented by the reduction computation. By iteratively using multiple threads to accumulate data in the shared memory, the reduction computation is effectively parallelized. The most critical computation in the threshold-based method is the histogram computation, by deploying the postponing strategy to reduce the usage of atomic functions, the threads are capable of writing the buckets of the histogram in parallel. The region growing iteration achieved relatively low speedups for two reasons. First, the degree of parallelization depends on the number of seeds, if there is not enough seeds the number of the active threads will be reduced. Second, the growing process is an iterative process in the shared memory that cannot be parallelized. The Hough transform-based method achieved the lowest speedup due to the loop computation in the accumulator matrix generation: each thread is assigned to process one pixel, the θ value increases from 0 to π to compute the corresponding ρ. In our experiments, attempts to use multiple threads to parallelize the loop failed due to the thread number limitations in one block.

In addition, in this study, we didn't compare the speedups with other literatures for it would be unfair because of the performance gaps between different hardware platforms.

MULTI-LEVEL PARALLELIZATION STRATEGIES

Parallelization strategies on the algorithm development level: There are three ways to develop parallelization algorithms: first, to directly parallelize the sequential algorithms; second, to develop completely new algorithms and third, to deploy the existing parallelization algorithm into the new problems. The parallelization practice in this work provided vivid examples for these three parallelization algorithm development strategies.

Probably the easiest way is to directly parallelize the sequential algorithm. In the filter-based edge detection method, the sequential filtering and the threshold computations are parallelized by using threads to concurrently process multiple pixels. In the Hough transform-based line detection method, each candidate edge is managed by one thread to compute the accumulator matrix and each entry of the accumulator matrix is managed by one thread to find the local peek values. In the threshold-based method, multiple threads are concurrently accessed from the global memory to compute the buckets of the histograms. In the above three tasks, the mathematical formulations do not need to be changed during the parallelization.

The second strategy to algorithm parallelization is to develop new algorithms. As in the region growing method, the sequential algorithm uses the stack data structure to store the seed points. In the parallel version, image blocks are loaded into the shared memory and iterations are performed in the shared memory to expend the regions.

The third strategy is to modify a suitable existing parallelization algorithm to parallelize the new problem. In the tight-fitting oriented bounding box method, the computations of center point and covariance matrix are accumulation computations. The accumulation computation can be implemented by the reduction computation which has been finely parallelized on the GPGPU platform on some published documents (Harris, 2007).

Parallelization strategies on the program development level: Several strategies have been actively used in this work to optimize the programs. The first or perhaps the most important strategy is to find the time-consuming parts. If these parts do not occupy most of the runtimes, any effort for parallelization is useless.

The second strategy is the access of various memories. The global memory access should be coalesced whenever possible. Therefore, when the pixels are threshold in the filter-based edge detection method and the threshold-based region segmentation, global memory are sequentially loaded and stored by thread warps. In the cases of histogram computations and reductions, we use one thread to process multiple pixels of the same column, the adjacent pixels on the same row are still processed by the adjacent threads to keep the coalesce access. The shared memory is an important issue in the modern

GPGPU programming, in all the above methods, the shared memory has been widely used. For example, the filtering operation and the finding local peak values operation need to acquire the neighboring pixels, by loading overlapping pixels between threads from the global memory to the shared memory, redundant loads from the global memory are reduced. In the center point computation and the covariance matrix computation, the shared memory is used in the reduction process because the access to the shared memory is much faster than the access to the global memory. For the same reason, in the region growing iteration process, the repeated read-write operations are performed in the shared memory instead of the global memory.

Third, in the computations of the histograms and the reductions, the atomic functions are used. The atomic operation is performed on a global or shared memory by one thread, without interference from other threads, consequently, the parallelization degree will be reduced. However, in some cases, to ensure the correctness of results, the atomic functions have to be used. Though the atomic functions cannot be avoided, the function invoking frequencies can be reduced, as the strategy shown in the pseudo code of the histogram parallelization.

CONCLUSIONS

In this study, a complete set of image segmentation methods was parallelized for horizontal comparisons. The most time-consuming parts of each method are pointed out to evaluate their parallel potentials. On the algorithm development level, three methods were introduced to generate the parallelization algorithms. On the program development level, three issues are emphasized to optimize the paralleled programs. It is expected that the conclusions of this report may contribute to the parallelization work of the more advanced image segmentation methods and other computer vision applications.

REFERENCES

Gonzalez, R.C. and R.E. Woods, 2002. Digital Image Processing. 2nd Edn., Pearson Education Asia Limited, Hong Kong, China, ISBN-13: 978-0201180756, Pages: 793.

Gottschalk, S., M. Lin and D. Manocha, 1996. OBB-tree: A hierarchical structure for rapid interference detection. Proceedings of the 23rd Annual Conference on Computer Graphics and Interactive Techniques, August 4-9, 1996, New Orleans, LA., USA., pp: 171-180.

Harris, M., 2007. Optimizing parallel reduction in CUDA. NVIDIA Corporation, Santa Clara, CA., USA. http://developer.download.nvidia.com/assets/cuda/files/reduction.pdf

Jorgensen, J.A., A.R. Fugl and H.G. Petersen, 2010. Accelerated hierarchical collision detection for simulation using CUDA. Proceedings of the 7th Workshop on Virtual Reality Interactions and Physical Simulations, November 11-12, 2010, Copenhagen, Denmark, pp: 97-104.

Kolawole, A. and A. Tavakkoli, 2011. Robust foreground detection in videos using adaptive color histogram thresholding and shadow removal. Proceedings of the 7th International Symposium on Advances in Visual Computing, September 26-28, 2011, Las Vegas, NV., USA., pp: 496-505.

Luo, Y. and R. Duraiswami, 2008. Canny edge detection on NVIDIA CUDA. Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition Workshops, June 23-28, 2008, Anchorage, AK., USA., pp: 1-8.

Podlozhnyuk, V., 2007. Histogram calculation in CUDA. NVIDIA Corporation, November 2007, Santa Clara, CA., USA. http://www.naic.edu/~phil/hardware/nvidia/doc/src/histogram/doc/histogram.pdf

Van den Braak, G.J., C. Nugteren, B. Mesman and H. Corporaal, 2011. Fast hough transform on GPUs: Exploration of algorithm trade-offs. Proceedings of the 13th International Conference on Advanced Concepts for Intelligent Vision Systems, August 22-25, 2011, Ghent, Belgium, pp: 611-622.

Zhang, X.Y. and Y.J. Kim, 2007. Interactive collision detection for deformable models using streaming AABBs. IEEE Trans. Vis. Comput. Graph., 13: 318-329.