Cluster Comput (2017) 20:1149–1154
DOI 10.1007/s10586-017-0788-5

Massively parallel acceleration methods for image handling operations

Nakhoon Baek[1,2] · **Kwan-Hee Yoo**[3]

Received: 9 July 2016 / Revised: 27 December 2016 / Accepted: 10 February 2017 / Published online: 24 February 2017

Abstract In the image handling and image processing areas, most operations can be executed in a pixel-by-pixel or cluster-by-cluster manner. These parallel and simultaneous executions have many benefits, and many researchers showed remarkable improvements. In this paper, we started from a specific and practical image handling and feature extraction sequences. We focused on the detailed design and robust implementation on the modern massively parallel architecture of CUDA. We present the enhanced features of our implementation and their design details. Our final result shows 13 times faster execution speed, in comparison with its previous CPU-based implementation. These methods can be applied to the variety of image manipulation processes.

Keywords Massively parallel acceleration · Image handling · Accelerated implementation

1 Introduction

In modern computer architectures, we have several kinds of computational processors, including the traditional *CPU* (central processing unit), the graphics-based *GPU* (graphics processing unit), and others. Traditionally, CPUs are widely used for general computational purposes. In contrast, GPUs are originated from the graphics cards, and now, commonly used for massively-parallel processing and simultaneous computations [1,2].

In the case of CPUs, the modern architecture shows the parallel processing with multiple cores. Those multi-core CPUs usually focused on the execution of dozens of parallel threads. However, GPUs can execute more than million threads simultaneously, with more than thousands of processing cores. Nowadays, these kinds of massively-parallel processing is ubiquitous [3].

We also have a set of parallel computation models, even in the commercial markets. For CPU-based parallel processing, *MPI* (message passing interface) [4,5] and *OpenMP* (open multi-processing) [6] are widely used. For GPU-based massively-parallel processing, *CUDA* (compute unified device architecture) [7,8] and *OpenCL* (open computing language) [9] are the dominant ones.

In this paper, we select CUDA as the massively parallel execution framework. CUDA is a general purpose programming model, and its users kick off batches of many threads on the GPUs. In this model, GPUs are dedicated super-threaded, massively data parallel co-processors. We will show the details of massively parallel accelerations with CUDA for image manipulation methods.

In this paper, we will focus on a practical feature extraction process. The input images, as shown in Fig. 1a, are captured from grayscale digital cameras in real-time. Our process performs various interpretation and verification steps. Finally, the verified and extracted features are presented on the output images, as shown in Fig. 1b. These steps have been implemented with C++ programming language, as a typical serialized program [10,11].

Our goal is to accelerate these kinds of programs with the modern massively parallel architectures, more precisely, the

✉ Kwan-Hee Yoo
khyoo@cbnu.ac.kr

Nakhoon Baek
oceancru@gmail.com

1 School of Computer Science and Engineering, Kyungpook National University, Daegu 41566, Republic of Korea

2 Software Technology Research Center, Kyungpook National University, Daegu 41566, Republic of Korea

3 Department of Software, Chungbuk National University, Cheongju, Chungbuk 28644, Republic of Korea

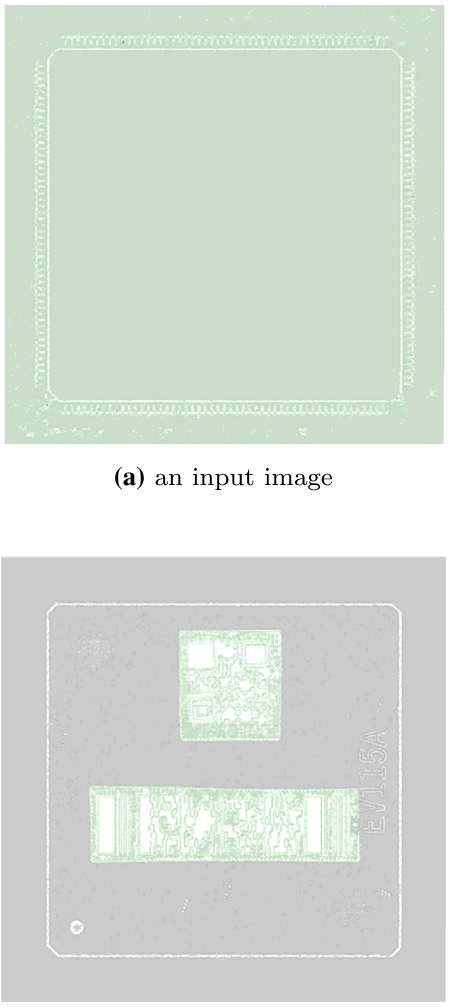


Fig. 1 Example images from our target process

CUDA architecture. To achieve these goals, we dissolved all steps in the original programs, and converted them into their corresponding massively parallel versions. Through fully optimizing these massively parallel versions, we got much impressive speedups on the target process. It is a practical and also an easy-to-extend solution to the image handling processes.

All the details are shown in the following sections. Implementation results are followed. Finally, Sect. 4 shows our conclusions and future work.

2 Design and implementation

As explained in the previous section, we focused on the parallel execution on the CUDA architecture, for some image handling operations. Among many cases of CUDA-based optimizations, we show some remarkable speed up-cases, including statistical value calculations, median filtering, and connected-component labeling. Each will be explained in the following subsections.

2.1 Data fetch operations

For a given image, we immediately need a set of fundamental statistical values, including the total sum, the average value, the standard deviation value, and others. To calculate any statistical values for an image, we start from the fetching of pixel values in the image. Typically, we have a set of grayscale images and also a set of true-color images. Since the true color images can be interpreted as three (red, green and blue) or four (red, green, blue and alpha) independent channels of grayscale images, we will focus on the handling of grayscale images, from now on.

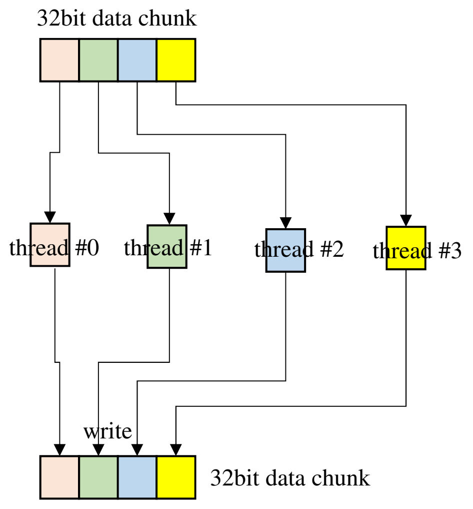
To get parallel implementations of these statistical operations, we first focused on the optimization of data fetch operations. For grayscale images, a pixel corresponds to a single byte data. Since modern PC architectures need 32-bit word data handling, we need to read 32-bit data chunks to get the 8-bit pixel data, as shown in Fig. 2a. In the case of typical CUDA parallel implementation, each thread will internally read a 32-bit data chunk and extract 8-bit pixel data. After its own processing, the thread will write back the 8-bit data, and the memory architecture will update a single 32-bit data chunk four times, by the 4 independent threads. It can cause serious buffer access delays in the read/write operations.

As a more optimized way, we let a single thread treat four adjacent pixels, as shown in Fig. 2b. A single 32-bit data chunk will be read, and then the chunk is divided into 4 bytes. The thread will perform its own processing four times repeatedly. All the results are combined into a single 32-bit data chunk, and then write back the data once. Since this optimized thread can read and write exactly once for processing totally four adjacent pixels, it is much better in its execution speed. All our operations are intuitively processed with these 32-bit chunk-based processing, to finally achieve remarkable speedups.

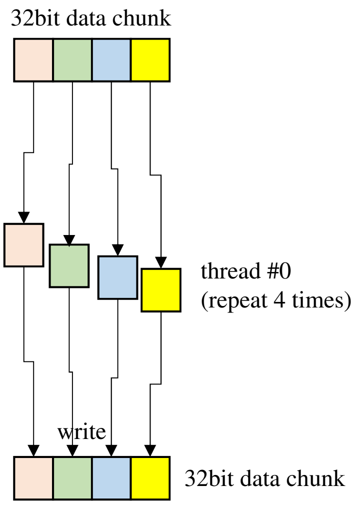
2.2 Reduction problems

We need to perform a set of reduction processing including total sum, average values, and others. *Reduction processing* means extracting a single numerical value (or a small number of numerical values) from a large set of input values. In our case, extracting total sum of pixel values in an image can be regarded as a typical reduction processing. The most important operation in the reduction processing is how to globally propagate the processed values [12].

To extract the total sum of all pixel values, a typical reduction processing can be started with invoking a set of threads for each pixel, and each thread will simultaneously update

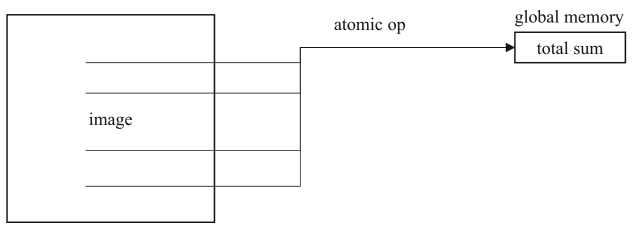


(a) typical data flow

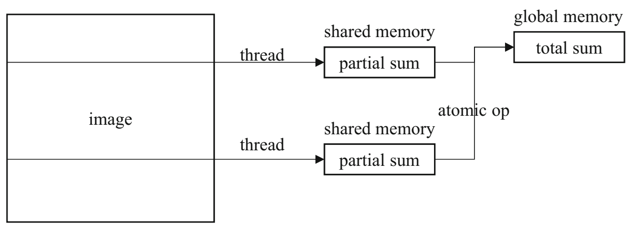


(b) our new approach

Fig. 2 Data read and write operations in our CUDA architecture



(a) typical implementation



(b) our new approach

Fig. 3 Propagation of reduction values in our operations

the global sum value through atomic operations. This brute-force approach results in the bottle-neck phenomena due to the heavy atomic operations, as shown in Fig. 3a.

In contrast, we use a hierarchical approach to minimize the number of atomic operations. First, we reduce the number of threads, and a thread will handle an entire row in the image rather than a single pixel, as shown in Fig. 3b. Then, the thread block will invoke a set of threads, and each thread will update the partial sum variable in the shared memory, through atomic operations. Though these updates requires atomic operations, they are performed on the shared memory, rather than on the global memory. Thus, we can achieve much speedups. Finally, the thread blocks will get the partial sum, and updated the total sum on the global memory, with the global memory access atomic operations. This hierarchical approach does the same work, with more efficient shared memory uses and casual use of memory access patterns, to finally get much speedups.

2.3 Standard deviation calculation

Through efficient use of shared memory and reducing the number of required atomic operations, we can achieve impressive speedups with CUDA-based reduction processing implementations. One more remarkable point on the statistics value calculation is the standard deviation calculation.

In the case of standard deviations, typical serialized implementations use the corrected sample standard deviation s of the following forms [13]:

$$s = \sqrt{\frac{1}{N-1}\sum_{i=1}^{N}(x_i - \bar{x})^2}, \qquad (1)$$

where N is the number of sampling points, x_i is the pixel value and $\bar{x}$ is the mean of the whole pixels.

For the deviation calculation, alternatively, we can use the standard deviation σ of the following formula:

$$\sigma = \sqrt{\frac{1}{N}\sum_{i=1}^{N}(x_i - \mu)^2}, \qquad (2)$$

where the mean μ is calculated as:

$$\mu = \frac{1}{N}\sum_{i=1}^{N} x_i .$$

In this case, we can calculate the standard deviation somewhat differently, as follows:

$$\begin{aligned}\sigma &= \sqrt{E\left[(X-\mu)^2\right]} \\ &= \sqrt{E\left[X^2\right]+E\left[-2\mu X\right]+E\left[\mu^2\right]} \\ &= \sqrt{E\left[X^2\right]-2\mu E\left[X\right]+\mu^2} \\ &= \sqrt{E\left[X^2\right]-2\mu^2+\mu^2} \\ &= \sqrt{E\left[X^2\right]-\mu^2} \\ &= \sqrt{E\left[X^2\right]-(E\left[X\right])^2}\end{aligned}$$

where $E(X)$ is the expected value of X.

In CUDA-based implementations, the standard deviation σ can be calculated simultaneously for each pixel, and more efficiently for the parallel executions. Actually, we implemented two independent versions of standard deviation calculation: the corrected sample standard deviation, using Eq. 1, and the standard deviation, using Eq. 2. Users can freely select one of them, where the standard deviation calculation of Eq. 2 can enjoy the parallel calculation on each thread, to get final speedups.

2.4 Median filtering

Median filters are basically selections of the median values among its neighboring pixels [14,15]. As an example, in the 3×3 pixel case, we pick up a pixel and its neighboring 8 pixels, and get the median as the final output pixel, as shown in Fig. 4. During its calculations, the core action will be to pick up the median of 9 pixel values. In CPU-based calculations and even straight-forward CUDA-based implementations, they usually use sorting operations to get the median value.

In contrast, CUDA-based implementations show one of the most inefficient operations, when we use any kind of sorting operations. Actually, to sort 9 elements with typical $O(n^2)$ sorting algorithms, we need at most 56 swap operations for each pixel [16]. However, median selection does not requires the whole sorting of the 9 elements. More optimized median selection algorithm can be used, and it requires only 19 swapping operations. Similarly, we can cleverly performs minimized number of swapping operations, for median selection. Practically, they need 104 swaps rather than 625 swaps, for 5×5 median finding [17,18].

However, for general case median finding, it is not easy to implement a median finding operation efficiently. Theoretically, median finding can be easily achieved through sorting the whole elements, and then selecting the middle position value. Generally, the sorting operations can be achieved in $O(n \log n)$ or $O(n^2)$ time, and it is too inefficient, in comparison with other median finding techniques. We focused that our input domain is only grayscale pixel values, ranged between 0 and 255. In this case, we can use the counting sort, with $O(n)$ worst case time complexity. Through applying the counting sort technique, we achieved a remarkably efficient median filtering operations even for general filter sizes [19].

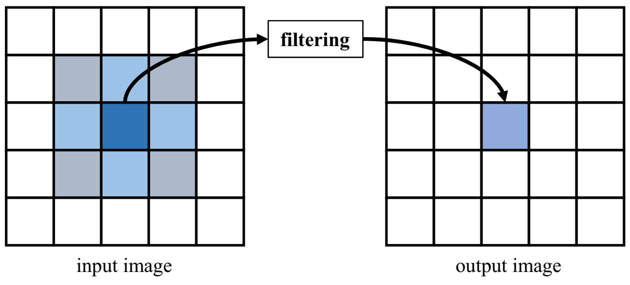


Fig. 4 Calculation of median filters, for 9 pixels case

2.5 Connected-component labeling operations

Connected-component labeling (shortly, CCL) is an algorithmic application of graph theory, where subsets of connected components are uniquely labeled, based on a given heuristic. It is also known as connected-component analysis, blob extraction, region labeling, blob discovery, or region extraction. Connected-component labeling is used in the area of image handling to detect connected regions in binary or grayscale digital images.

In image handling applications, CCL algorithms start with classifying the pixels with the same properties. For a pixel, we can apply 4-connectivity or 8-connectivity, as shown in Fig. 5, to connect the pixels with the same properties. After globally connecting all the pixels with the same property, we can extract the connected components as shown in Fig. 6.

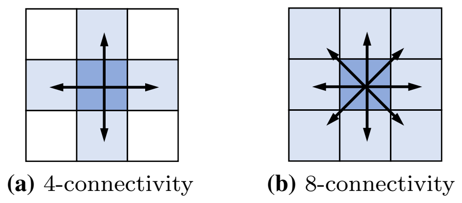


Fig. 5 4-connectivity and 8-connectivity for CCL operations

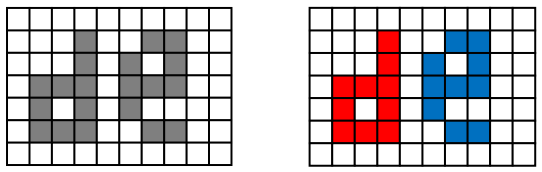


Fig. 6 Input grayscale image and two connected components

The CCL operation itself is widely known and many efficient implementations are already available [20–22]. We also have CUDA-based implementations [23,24]. Among them, we chose the most stable implementation [25], and modified it for integer pixel values in the grayscale images.

Based on the CUDA-based CCL implementation, we extend it to find our interest regions. For example, after the first path of CCL operations, we remove all the connected components with smaller region sizes, regarding them as noise areas. The remaining connected components are now regarded as meaningful image segments, and assign its own identification number for further processing.

2.6 Flood filling

Flood filling is an algorithm that determines the area connected to a given start pixel in the image. When applied on an image to fill a particular bounded area with color, it is also known as boundary filling. In general cases, they use the scan-line filling algorithms for fast flood filling. Instead of testing each potential pixels, this scan-line filling algorithm processes each scan-line in the image as a chunk, and get the global result more efficiently.

In our case, we need faster implementation base on CUDA parallel architecture. After carefully considering several candidates, we finally use the previous CUDA-based CCL implementation as the underlying operations to achieve flood filling. Our idea is simply applying CUDA-based CCL operations on the input image. If a target pixel is selected, its corresponding connected component and all the indirectly connected components for the pixel are combined to get the final flood filling result. Through these combinations, we achieved much faster flood filling operation.

3 Implementation results

As shown in the previous section, we designed and implemented all the CUDA-based image handling operations. As an example application, we use a feature extraction application, as shown in Fig. 1. From the input image of a camera-captured grayscale image, it get some intermediate results, and finally produce the final feature extracted image of Fig. 1b. The example application process internally performs totally 4 times of CCL operations, one flood filling operation, several statistical operations including average and standard deviation calculations.

For the test purpose, our previous CPU-based implementation works on a dual Xeon 3.10GHz CPU workstation, with 64GB RAM. On this machine, the overall process works in 779 millisecond, excluding any input/output processing.

Our CUDA-based implementation works on another workstation with Intel i5 3.40GHz CPU and 8GB RAM. In contrast, this machine has equipped with a single NVIDIA GTX 980 graphics card. With combining CUDA 5.2 computation library, it provides 16 parallel processors, which totally support 2,048 simultaneous thread executions. Our CUDA-based implementation shows the final execution speed of 60 millisecond.

Actually, we need to transfer the original input data on the RAM area to CUDA graphics memory area. Those transfer operations are performed in asynchronous copy operations, and achieved in 98 millisecond. Finally, it shows that the pure processing time was reduced from 779 millisecond to 60 millisecond, which is 12.98 times speedup, for the overall processing, even including some CPU-specific operations.

4 Conclusions and future work

We applied the speed up techniques to the CUDA-based implementations. The over-all process accepts the input image shown in Fig. 1a, and finally, it produces the output image shown in Fig. 1b. We checked the output image with respect to the original CPU-based implementations, and we have confirmed it is exactly the same to the previous results.

Our current implementation is based on CUDA architecture, and makes 12.98 times faster, in comparison to the previous CPU-based implementation. We can achieve much more speedups with more and more optimizations. The presented methods can be applied to the variety of image manipulation processes.

Acknowledgements 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 (Grant 2016R1D1A3B03935488). This study was also supported by the 2015 Technology Innovation Development program (project name: "Development of 3D GIS Platform for the LiDAR Data Utilization") funded by the Small and Medium Business Administration, Korea (Project Number: S2306348).

References

1. Malizia, A.: Mobile 3D Graphics. Springer-Verlag New York Inc, Secaucus (2006)
2. Pulli, K., Vaarala, J., Miettinen, V., Aarnio, T., Roimela, K.: Mobile 3D Graphics: with OpenGL ES and M3G. Morgan Kaufmann Publishers Inc., San Francisco (2007)
3. Sutter, H., Larus, J.: Software and the concurrency revolution. ACM Queue **3**(7), 54–62 (2005)
4. Aoyama, Y., Nakano, J.: RS/6000 SP: Practical MPI Programming. ITSO (1999)
5. Pacheco, P.S.: Parallel Programming with MPI. Morgan Kaufmann (1997)
6. ARB, O.: OpenMP Application Programming Interface, Version 4.5. OpenMP.org (2015)
7. NVIDIA CUDA Home page http://www.nvidia.com/object/cuda_home_new.html

8. NVIDIA: CUDA C Programming Guide. NVIDIA (2016)
9. Munshi, A.: The OpenCL Specificaiton, Version 1.0. Khronos OpenCL Working Group (2012)
10. Josuttis, N.: The C++ Standard Library, 2nd edn. Addison-Wesley Professional, Boston (2012)
11. Stroustrup, B.: The C++ Programming Language, 4th edn. Addison-Wesley Professional, Boston (2013)
12. Harris, M.: Optimizing Parallel Reduction in CDUA. NVIDIA (2016)
13. Reid, H.: Introduction to Statistics. SAGE Publications, New York (2013)
14. Burger, W., Burge, M.: Digital Image Processing. Springer, London (2016)
15. Gonzalez, R.: Digital Image Processing, 3rd edn. Pearson Education Inc., New Delhi (2014)
16. Devillard, N.: Fast median search: an ansi c implementation. http://ndevilla.free.fr/median/median/
17. Smith, J.: Implementing median filters in xc4000e fpgas. XCell **23**, 16 (1996)
18. Paeth, A.W.: Median finding on a 3-by-3 grid. In: Graphics Gems, pp. 171–175 (1993)
19. Huang, T.S., Yang, G.J., Tang, G.Y.: A fast two-dimensional median filtering algorithm. IEEE Trans. Acoust. Speech Signal Process. ASSP **27**(1), 13–18 (1979)
20. Suzuki, K., Horiba, I., Sugie, N.: Linear-time connected-component labeling based on sequential local operations. Comput. Vis. Image Underst. **89**, 1–23 (2003)
21. He, L., Chao, Y., Suzuki, K., Wu, K.: Fast connected-componentlabeling. Pattern Recognit. **42**, 1977–1987 (2009)
22. Wenke, H., Kolodzey, S., Vornberger, O.: A work-optimal parallel connected-component labeling algorithm for 2d-image-data using pre-contouring. In: Scientific Cooperations International Workshops on Electrical and Computer Engineering Subfields, pp. 154–161 (2014)
23. Paravecino, F., Kaeli, D.: Accelerated Connected Component Labeling Using Cuda Framework. Lecture Notes in Computer Science, vol. 8671, pp. 502–509 (2014)
24. Kalentev, O., Rai, A., Kemnitz, S., Schneider, R.: Connected component labeling on a 2d grid using cuda. J. Parallel Distrub. Comput. **71**, 615–620 (2011)
25. Stava, O., Benes, B.: Connected Component Labeling in CUDA. GPU Computing Gems (2010)

Nakhoon Baek is currently a professor in the School of Computer Science and Engineering at Kyungpook National University, Korea. He received his B.A., M.S., and Ph.D. degrees in Computer Science from Korea Advanced Institute of Science and Technology (KAIST) in 1990, 1992, and 1997, respectively. His research interests include graphics standards, graphics algorithms and real-time rendering. He is now also the Chief Engineer of Dassomey.com Inc., Korea.

Kwan-Hee Yoo is a professor working for the Department of Computer Science at Chungbuk National University, Korea. He received his BS in Computer Science from Chonbuk National University, Korea, in 1985, and his MS and Ph.D. degrees in Computer Science from Korea Advanced Institute of Science and Technology, Korea, in 1988 and 1995, respectively. His research interests include computer graphics, integral imaging system, dental/medical systems, and smart learning.