Proceedings of the Federated Conference on Computer Science and Information Systems pp. 653–656
DOI: 10.15439/2017F265
ISSN 2300-5963 ACSIS, Vol. 11

GPU Accelerated 2D and 3D Image Processing

Anca Morar, Florica Moldoveanu, Alin Moldoveanu, Oana Balan,Victor Asavei
University POLITEHNICA of Bucharest
Email: anca.morar@cs.pub.ro, florica.moldoveanu@cs.pub.ro, alin.moldoveanu@cs.pub.ro, oana.balan@cs.pub.ro, victor.asavei@cs.pub.ro

***Abstract*— The current advances in hardware led to the development of the GPGPU (General-purpose computing on graphics processing units) paradigm. Thus, nowadays, the GPU (Graphics Processing Unit) is used not only for graphics programming, but also for general purpose algorithms. This paper discusses several methods regarding the use of CUDA (Compute Unified Device Architecture) for 2D and 3D image processing techniques. Some general rules for writing parallel algorithms in computer vision are pointed out. A theoretic comparison between the complexity for CPU (Central Processing Unit) and GPU implementations of image processing algorithms is given. Also, real computing times are provided for several algorithms in order to point out the actual performance gain of using the GPU over CPU. The factors that contribute to the difference between theoretic and real performance gain are also discussed.**

I. Introduction

UNTIL recently, the GPU was used only for graphics programming. The transition from a fixed to a programmable rendering pipeline allowed programmers to write high level code for graphics applications through shaders. Shaders are defined for an element belonging to one of the types that are processed in the graphics pipeline, for example vertex or fragment, and are executed for all the elements of that type in a parallel manner. According to Soller [1], early approaches to using the GPU for general computation date back to the year 2000. However, for this purpose, all tasks had to be mapped to the computer graphics domain. The development of the GPGPU paradigm led to a revolution in terms of computing times for many algorithms. This paper describes some general rules when implementing computer vision algorithms with CUDA, as well as theoretical and real performance gains of GPGPU implementations as compared to sequential ones. The second section discusses the state of the art in GPU based image processing algorithms. The third section presents theoretic comparisons between GPU and CPU implementations of several 2D image processing algorithms. The fourth section discusses some issues when processing very big volume data. Several comparisons between theoretical results and tests conducted on real hardware are presented in the fifth section. The conclusions are drawn in the final section.

This work has been funded by University Politehnica of Bucharest, through the "Excellence Research Grants" Program, UPB – GEX. Identifier: UPB-EXCELENTA-2016 "3Diafano – Reconstructia si vizualizarea tesuturilor pe baza transiluminarii in NIR si a camerelor video 3D", contract number 01/26.09.2016, code 514. This work has also received funding from the European Union's Horizon 2020 research and innovation program under grant agreement 643636 (www.soundofvision.net).

II. State of the Art in GPU-Based Image Processing

Some of the GPGPU image processing methods are briefly discussed.

A. Acceleration of 2D Image Processing Algorithms

Takamura and Shimizu [2] describe a denoising filter with genetic programming schemes for dynamic procedure generation. Abdellah [3] presents an easy-to-use CUDA library that implements Fast Fourier Transform-shift operations. Agrawal et al. [4] perform a real-time GPU-based generation of the saliency map for a given image. Lee et al. [5] improve the computing times of the Viola-Jones algorithm for face detection by employing different strategies for CPU-GPU task-level parallelism. Ma et al. [6] propose a CUDA-based acceleration of the Fisher Vector extraction method for various video monitoring applications. Hwang et al. [7] present a CUDA implementation of foreground detection based on background modeling. Yao et al. [8] describe a CUDA-based image inpainting algorithm for virtual viewpoint synthesis.

B. Acceleration of 3D Image Processing Algorithms

Shewale et al. [9] analyze the performance of different CPU/GPU parallel implementations of the Gaussian filter, k-means clustering based segmentation and Fourier based coefficient registration of medical images such as CTs and MRIs. Valero [10] proposes a GPU-based implementation for accelerating the DARTEL algorithm for diffeomorphic registration of brain biomedical images. Langdon et al. [11] use genetic programming to improve the performance of an existing CUDA implementation for 3D medical image registration.

C. GPGPU Frameworks

Lee et al. [12] propose optimization strategies for compute- and memory-bound algorithms using the CUDA architecture. They test their optimization strategies on a 3D unbiased nonlinear image registration technique and on a non-local means surface denoising algorithm. Ravishankar et

al. [13] present a domain-specific language for image processing, namely Forma, which provides syntax for stencil computation, sampling and other 2D or 3D algorithms.

III. Image Processing Algorithms with CUDA

From the parallel implementation point of view, most of the image processing algorithms belong to one of four categories: pixel-to-pixel, neighborhood, global and multi-steps (Fig. 1). Each of these classes is discussed below.

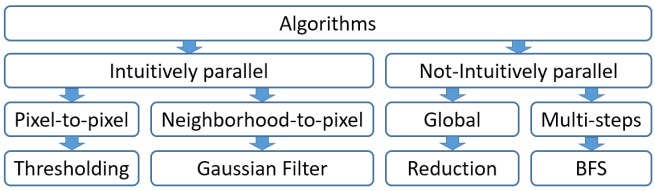

Fig. 1 Discussed 2D image processing algorithms

A. Pixel-to-pixel (P2P) Algorithms

Pixel-to-pixel algorithms assume that each pixel in an image is processed based solely on its characteristics. One of the most common *pixel-to-pixel algorithms* encountered in image processing is pixel value remapping, based only on the value of the current pixel. Value remapping can be used for enhancement of structures characterized by certain intensity values. A particular case of value remapping is segmentation based on thresholds. This type of algorithm is naively parallel due to the fact that each pixel is handled independently by a thread. An easy implementation in CUDA of this type of algorithm is to copy the image or volume into texture memory, so that the access to the pixel values is very fast. The output of the algorithm is an image/volume with the same size as the input.

Let C_{pixel} be the complexity of the operations applied to one pixel from an image in *a pixel-to-pixel algorithm*. Then the complexity of the entire algorithm for a 2D image in a sequential approach, on the CPU, is:

$$C_{P2P\_CPU} = C_{pixel} \cdot w \cdot h, \qquad (1)$$

where w is the width of the image and h is its height. As previously mentioned, in a parallel approach on the GPU, each thread handles only one pixel and accesses only the memory related to that pixel. Therefore, the theoretic complexity of the same algorithm in a GPGPU approach is C_{pixel}. The theoretic performance gain obtained is $w \cdot h$. However, the transfer between host and device memory introduces a latency that decreases the performance gain in real applications. Also, the actual computing time for the operations applied per pixel is not the same for the CPU and the GPU and depends very much on the actual hardware. The memory transfer latency can be reduced through page locked memory and the zero-copy feature, but not significantly.

B. Neighborhood-to-pixel (N2P) Algorithms

Spatial filters are applied locally, at the level of each image pixel, by replacing the value of the current pixel depending on the values of the neighboring pixels. Among the *neighborhood algorithms* we can mention the Gaussian filter, for noise removal, or the Sobel filter, for edge extraction. The difference between *neighborhood algorithms* and *pixel-to-pixel algorithms* is the access to memory. In *neighborhood algorithms*, the thread corresponding to one pixel has to access information not only about the current pixel, but also about its neighboring pixels. These algorithms can be implemented in the same manner as the *pixel-to-pixel* algorithms, by copying the image/volume into texture memory. Another possibility is the use of shared memory, as described in [14]. Each thread block can copy parts of the image by loading data from texture to shared memory. The barrier synchronization forces each thread to wait until all the other threads have finished loading the corresponding data from texture to shared memory. Even if shared memory is faster, the transfer between texture and shared memory introduces a lag that determines an insignificant difference between the two implementations. The theoretic complexities for sequential and parallel implementations are similar to those of the *pixel-to-pixel algorithms*. However, the differences between theoretic and actual performance gains are bigger in this case.

C. Global (G) Algorithms

Global algorithms refer to computations that access information about all the pixels in an image, not just a neighborhood. Examples of global algorithms are the computation of the average intensity or the maximum/minimum intensity in an image. The computation of a global parameter in an image is not intuitively parallel, because it depends on all the pixels in the image.

Let C_{pixel} be the complexity of the operations applied to one pixel in a global algorithm. For example, when computing the maximum intensity in one image, C_{pixel} is the complexity of comparing the intensity of the current pixel with the current maximum value and modifying the current maximum value, if necessary. The complexity of a global algorithm in a sequential implementation is $C_{pixel} \cdot w \cdot h$. A parallel approach to implementing global algorithms is the *reduction method*. CUDA threads are organized into blocks and grids. The blocks can be structured into a one-dimensional grid of size h and the threads can be structured into one-dimensional blocks of size w. Thus, each block handles one row in an image. The threads in a block cooperate in order to determine a partial global parameter which depends only on the current row. For example, when computing the maximum intensity in an image, this partial global parameter is the maximum intensity for the pixels located on the current row. Each block loads the data into shared memory, into an array of size w. The computation of

the partial global parameter for the current block is done in $\log_2(w)$ iterations. In each iteration, the number of active threads is divided by 2. The computation of the final global parameter is also accomplished with the reduction method, but in $\log_2(h)$ iterations. In each iteration, the active threads run in parallel, but before going to the next iteration, they need to synchronize. The theoretical complexity for the parallel implementation of a global algorithm is:

$$C_{G\_GPU} = C_{pixel} \cdot \left(\log_2(w) + \log_2(h)\right). \quad (2)$$

The theoretical performance gain obtained when using the reduction method for global algorithms is $w \cdot h / \log_2(w)$. Besides the lag introduced by the memory transfers and access, the latency caused by the barrier synchronization in the reduction method influences the real performance gain.

In a multi-GPGPU approach, the image can be divided based on the number of available GPUs. After each GPU computes a partial global parameter, the final global parameter is computed on the CPU in *N* iterations. For *N* GPUs, the complexity of the global algorithm is:

$$C_{G\_MultiGPU} = C_{pixel} \cdot \left(\frac{\log_2(w) + \log_2(h)}{N} + N\right) \quad (3)$$

D.Multi-steps (MS) Algorithms

These algorithms are executed in more iterations, the processing of the k^{th} iteration depending on the result of the processing from the $(k-1)^{th}$ iteration. An example of *multi-steps algorithm* is breadth first search (BFS) for images, which starts with a seed pixel and discovers similar pixels connected with this one. The similarity measure can be defined based on the intensity values, the gradient, etc.

An image can be interpreted as a graph where each node is a pixel. The graph edges can be defined based on the similarity of the pixels. A common practice in BFS is to define the edges that connect pixels in a 4-neighborhood. In the worst-case scenario, the seed pixel is the one located in the middle of the image and all the pixels are similar. If the complexity for one pixel is C_{pixel}, the sequential complexity for the worst-case scenario is:

$$C_{MS\_CPU} = C_{pixel} \cdot \left(1 + 4 + 4^2 + \ldots + 4^{\max(w/2,h/2)}\right). \quad (4)$$

The CUDA implementation of the BFS in image processing is described in [15]. The complexity for the CUDA implementation of the BFS for the worst case scenario is:

$$C_{MS\_GPU} = C_{pixel} \cdot \max(w/2, h/2) \cdot 4. \quad (5)$$

The theoretic performance gain obtained when approaching the BFS in a parallel manner is:

$$Gain_{MS} = \frac{1 + 4 + 4^2 + \ldots + 4^{\max(w/2,h/2)}}{\max(w/2, h/2) \cdot 4}. \quad (6)$$

A recursive algorithm is not suitable for multi-GPGPU approaches. Multiple GPUs can be used only if there are more than one seed pixel in the BFS, or more than one initial image in the recursive splitting.

IV. Volume Data (3D) Processing with CUDA

The computation of the theoretic complexities can be easily extended to the 3D image processing. A 3D volume can be seen as a stack of s 2D images or slices, each of size $w \cdot h$. For the *pixel-to-pixel* and the *neighborhood-to-pixel algorithms*, the sequential complexity is:

$$C_{3D\_P2P\_CPU} = C_{pixel} \cdot w \cdot h \cdot s. \quad (9)$$

The theoretic parallel complexity remains C_{pixel}, as in the 2D case.

The sequential complexity of the global algorithms is $C_{pixel} \cdot w \cdot h \cdot s$. The parallel implementation of a global algorithm assumes the computation of partial global parameters, one for each slice, and the computation of the final global parameter, for the whole volume, with the reduction method, in $\log_2(s)$ iterations. Thus, the theoretic parallel complexity of the global algorithms is:

$$C_{3D\_G\_GPU} = C_{pixel} \cdot \left(\log_2(w) + \log_2(h) + \log_2(s)\right). \quad (10)$$

The BFS extension to 3D assumes the inspection of two more neighbors for each voxel, one on the upper adjacent slice and the other one on the lower adjacent slice. Thus, the sequential complexity for the worst-case scenario becomes:

$$C_{3D\_MS\_CPU} = C_{pixel} \cdot \left(1 + 6 + 6^2 + \ldots + 6^{\max(w/2,h/2,s/2)}\right). \quad (11)$$

The parallel implementation will have the following complexity:

$$C_{3D\_MS\_GPU} = C_{pixel} \cdot \max(w/2, h/2, s/2) \cdot 6. \quad (12)$$

Many volume data come from CT or MRI scans. The main problem of 3D image processing is the large size of the data acquired from the scanning devices. An example of neighborhood algorithm that processes volumes is marching cubes. The increased complexity of the marching cubes algorithm, caused by the huge number of intersections that are processed, implies slow computing times and high memory usage. The classic CUDA implementation of this algorithm [15] leads to real time surface reconstruction, but can handle only small datasets. GPUs can lead to significant performance gains as compared to sequential implementations, but the GPU memory is limited. We proposed an approach that divides the initial volume into sub-volumes, which can be computed serially on the GPU without exceeding the memory pool [16].

V. Results

This section presents results derived from tests conducted on real hardware. The tests were made on an i7-2600K 3.40

GHz processor with 8 GB RAM and an Nvidia GeForce GTX 590 card with 1.5 GB RAM.

Table I presents the computing times of running a value remapping on the CPU and on the GPU.

Table II presents the computing times of applying a Gaussian filter on the CPU and on the GPU. We tested the implementation using only global memory, using shared memory and a multi-GPGPU implementation.

Table III presents a comparison between serial and parallel implementations of determining the maximum intensity in an image.

TABLE I.
COMPUTING TIMES FOR VALUE REMAPPING

Image size (pixels)	CPU implem (ms)	GPGPU implem (ms)	multi-GPGPU (ms)
256^2	0.2	0.19	0.39
512^2	1	0.34	0.42
1024^2	3.4	0.85	0.7

TABLE II.
COMPUTING TIMES FOR GAUSSIAN FILTER

Image size (pixels)	CPU implem (ms)	GPGPU implem (ms)	GPGPU shred implem (ms)	multi-GPGPU (ms)
256^2	1.1	0.2	0.21	0.39
512^2	3.7	0.38	0.37	0.39
1024^2	14.6	1.04	0.99	0.71

TABLE III.
TIMES FOR COMPUTING THE MAXIMUM INTENSITY IN AN IMAGE

Image size (pixels)	CPU implem (ms)	GPGPU implem (ms)
256^2	2.1	0.7
512^2	2.2	0.88
1024^2	2.4	1.68

VI. CONCLUSIONS

This paper focuses on theoretic comparisons between sequential and parallel implementations of 2D/3D image processing algorithms. It also provides comparisons of real computing times for several CPU and GPU algorithms.

Four main classes of image processing algorithms are discussed. The main issues of CUDA programming in relation to these algorithms are presented. The paper also gives general rules for implementing image processing algorithms in CUDA, such as the type of GPU memory which should be used based on the particularities of the algorithms and the manner of translating non-intuitively parallel algorithms to parallel ones or best practices for multiple GPUs.

In theory, parallel implementations introduce a very high performance gain as compared to sequential implementations. In practice, memory transfer lags, memory access and real hardware characteristics lead to a smaller performance gain. Still, GPGPU image processing algorithms are undeniably faster than CPU ones.

REFERENCES

[1] Stephan Soller, "GPGPU Origins and GPU and GPU Hardware Architecture", Practical Term Report, High Performance Computing Center Stuttgart, Stuttgart Media University, 2011.

[2] S. Takamura, A. Shimizu, “GPGPU-assisted denoising filter generation for video coding”, GECCO Comp '14 Proceedings of the Companion Publication of the 2014 Annual Conference on Genetic and Evolutionary Computation, 2014, pp. 151-152.

[3] M. Abdellah, “CufftShift: High Performance CUDA-accelerated FFTshift Library”, Proceedings of the High Performance Computing Symposium, ser. HPC ’14. San Diego, CA, USA: Society for Computer Simulation International, 2014.

[4] R. Agrawal, S. Gupta, J. Mukherjee, R.K. Layek, “A GPU based real-time CUDA implementation for obtaining visual saliency”, Proceedings of the 2014 Indian Conference on Computer Vision Graphics and Image Processing, ACM, 2014

[5] 5. Y. Lee, C. Jang, H. Kim, “Accelerating a computer vision algorithm on a mobile SoC using CPU-GPU co-processing: a case study on face detection”, Proceeding MOBILESoft '16 Proceedings of the International Conference on Mobile Software Engineering and Systems, 2016.

[6] W. Ma, L. Cao, L. Yu, G. Long, Y. Li, “GPU-FV: Realtime Fisher Vector and Its Applications in Video Monitoring”, ICMR '16 - Proceedings of the 2016 ACM on International Conference on Multimedia Retrieval, pp. 39-46.

[7] S. Hwang, Y. Uh, M.Ki, K. Lim, D. Park, H. Byun, “Real-time background subtraction based on GPGPU for high-resolution video surveillance”, IMCOM '17 Proceedings of the 11th International Conference on Ubiquitous Information Management and Communication, 2014.

[8] L. Yao, Y. Han, X. Li, “Virtual Viewpoint Synthesis using CUDA Acceleration”, 22nd ACM Conference on Virtual Reality Software and Technology, pp/ 367-368, 2016.

[9] A. Shewale, N. Waghmare, A. Sonawane, U. Teke, “High Performance Computation Analysis for Medical Images using High Computational Methods”, ICTCS '16 Proceedings of the Second International Conference on Information and Communication Technology for Competitive Strategies, 2016

[10] P. Valero-Lara, “A GPU approach for accelerating 3d deformable registration (Dartel) on brain biomedical images”, in Proceedings of the 20th European MPI Users’ Group Meeting, EuroMPI ’13, New York, NY, USA, 2013, ACM, pp. 187–192.

[11] W.B. Langdon, M. Modat, J. Petke, M. Harman, “Improving 3D Medical Image Registration CUDA Software with Genetic Programming”, Annual Conference on Genetic and Evolutionary, pp. 951-958, 2014.

[12] D. Lee, I. Dinov, B. Dong, B. Gutman, I. Yanovsky, A. W. Toga, "CUDA Optimization Strategies for Compute- and Memory-Bound Neuroimaging Algorithms", Jounral on Computer Methods and Programs in Biomedicine", vol 106(3), pp. 175-187, 2012.

[13] M. Ravishankar, J. Holewinski, V. Grover, “Forma: A DSL for image processing applications to target GPUs and multi-core CPUs”, GPGPU, 2015, pp. 109–120

[14] A. Morar, F. Moldoveanu, V. Asavei, A. Egner, "Multi-GPGPU Based Medical Image Processing in Hip Replacement", Journal of Control Engineering and Applied Informatics, vol. 14(3), pp. 25-34, 2012.

[15] A. Morar, "Analysis and Visualization of Data from Medical Images", PhD Thesis, University POLITEHNICA of Bucharest, 2012.

[16] L. Petrescu, A. Morar, F. Moldoveanu, V. Asavei, "Real Time Reconstruction of Volumes from Very Large Datasets using CUDA", Proceedings of the 15th International Conference on System Theory, Control and Computing, pp. 462-466, 2011.