Peer Group and Fuzzy Metric to Remove Noise in Images Using Heterogeneous Computing

Ma. Guadalupe Sánchez[1], Vicente Vidal[2], and Jordi Bataller[2]

[1] Departamento de Sistemas y Computación, Instituto Tecnológico de Cd. Guzmán, 49100, Cd. Guzmán, Jalisco, Mexico
msanchez@dsic.upv.es
[2] Departamento de Sistemas Informáticos y Computación E.P.S. Gandia, Universidad Politécnica de Valencia, 46730, Grao de Gandia, Valencia, Spain
{vvidal,bataller}@dsic.upv.es

Abstract. In this paper, we report a study on the parallelization of an algorithm for removing impulsive noise in images. The algorithm is based on the concept of peer group and fuzzy metric. We have developed implementations using Open Multi-Processing (OpenMP) and Compute Unified Device Architecture (CUDA) for Graphics Processing Unit (GPU). Many sequential algorithms have been proposed to remove noise, but their computational cost is excessive for real-time processing of large images. We developed implementations for a multi-core CPU, for a multi-GPU (several GPUs) and for a combination of both. These implementations were compared also with different sizes of the image in order to find out the settings with the best performance. A study is made using the shared memory and texture memory to minimize access time to data in GPU global memory. The result shows that when the image is distributed in multi-core and multi-GPU a greater number of Mpixels/second are processed.

Keywords: remove impulsive noise, peer group, fuzzy metric, parallel algorithm, CUDA, OpenMP, multi-core, multi-GPU.

1 Introduction

Image denoising is still an open problem in the field of image processing, because damaged images may affect the performance and accuracy of some processes. Also, images under consideration may be very large and may require a real-time processing and this requires optimal implementations. For instance, noise removal is of paramount importance in emerging applications related to biomedical science, earth science, cultural heritage preservation, video communications, image post-processing, robotic inspection and surveillance. Impulsive noise is commonly found in images caused by the malfunction of sensors during the process of image formation, aging of the storage material or transmission errors due to natural or man-made processes [1]. This type of noise affects individual pixels, changing its original value. The widespread model of impulse noise is the "Salt and Pepper" model, or fixed-value noise. It considers that when a pixel is wrong, its value is an extreme value within the signal range. This is the model that we assume in the present paper.

M. Alexander et al. (Eds.): Euro-Par 2011 Workshops, Part I, LNCS 7155, pp. 502–510, 2012.

Many algorithms (filters) to reduce impulsive noise in images have been introduced: [1–13]. These cited works are based on the concept of "peer group". Given a pixel x_i, a peer group is the set of its neighbors that are similar to it, according to a chosen metric, [1] [10]. These filters have recently shown good results in quality but they do not seem to be appropriate for real-time processing.

Resources for real-time processing are easily available. For example, the Graphics Processing Units (GPUs) are currently a very popular platform for developing parallel applications, considering price and speed. It is obvious the convenience to develop parallel filter implementations for them.

In this paper, we introduce a parallel version of filters based on peer group and fuzzy metric in order to keep their best quality results while trying to improve its performance, making them usable for real-time processing. We have tested these algorithms using several GPUs (multi-GPU) in parallel, using also a multi-core CPU, and using the GPUs and the CPU in combination. We have investigated the most convenient distribution of the pixels on the memories and caches for these devices (for instance: shared memory versus texture memory) to take the most advantage of the hardware.

The paper is organized as follows: Section 2 explains the two steps of the denoising algorithm to be parallelized. Section 3 discusses how the algorithm has been implemented in a multi-GPU and in a multi-core CPU. The results of the experimental study are shown in Section 4. Finally, section 5 presents the conclusions reached using heterogeneous computing to eliminate noise in the images.

2 Denoising Algorithm

Our algorithm uses the peer group of a central pixel x_i in a window W according to [1], but with a fuzzy metric instead. The fuzzy distance [12] between pixels x_i and x_j in the color image is given by the following function:

$$M(x_i, x_j) = \prod_{l=1}^{3} \frac{min\{x_i(l), x_j(l)\}+k}{max\{x_i(l), x_j(l)\}+k}, \quad (1)$$

where $(x_i(1), x_i(2), x_i(3))$ is the color vector for the pixel x_i in RGB and x_j is a neighbor of x_i. In [6], it was shown that $k = 1024$ is an appropriate setting to maintain the image quality, and this is therefore the value that we use in our study.

The peer group of a pixel x_i is comprised by the pixels of a window centered in x_i whose distance M from x_i exceeds d:

$$P(x_i, d) = \{x_j \in W : M(x_i, x_j) \geq d\} \quad (2)$$

where $0 \leq d \leq 1$ is the distance threshold.

The sketch of the denoising algorithm has two main steps. In the first step, *detection*, the pixels are labeled either as *corrupted* or as *uncorrupted*. In the second step, *filtering*, the corrupted pixels are corrected. This is the description of the processing for each pixel x_i.

- Detection: x_i is declared as corrupted if $\#\, P\,(x_i, d) \leq (q + 1)$, where $q$is the value defined to decide if the pixel is labeled as corrupted or uncorrupted, and $\#\, P$ the cardinality of the set P.
- Filtering: if x_i was previously marked as corrupted, it is replaced using the well known arithmetic mean filter (AMF) [2], [3]. This is the new value for x_{ik} (color component $k$of x_i) is $(\sum x_{ij})/(\#W\text{-}1)$ for all $x_j \in W$ with $j \neq i$.

The algorithm must be divided into two steps because the AMF considers only uncorrupted pixels for the mean computation and, therefore, the second step cannot start until the detection is completely done.

3 Multi-GPU and Multi-core CPU Implementations

We have developed three implementations of the algorithm described in the previous section, for two parallel architectures. The first implementation is based on OpenMP and targets a multi-core CPU. The second implementation is developed in CUDA for a multi-GPU. The third version uses the combination of CPU and GPU.

The first decision to make is which part of the image is to be processed by each computing unit. For instance, figure 1 shows the case of the CPU, in which rows above the line $s$are assigned to one core, and the rest to a second core, leaving the remaining cores and the GPUs idle. Figure 2 shows an example of dividing an image into two parts, each one to be processed by a different GPU. Finally, figure 3 shows a partition of the image into eight horizontal blocks, to be processed by a combination of GPUs and cores CPU.

The implementations on GPUs have some differential traits. Data to be processed must be transferred from main memory (RAM) to the GPU memory and the results must be copied back to main memory. On the other hand, tasks to be executed by a GPU are coded into functions called kernels. In order to ensure the synchronization of the two steps of the algorithm, we have developed two kernels, one for detection and another for filtering, so that the latter won't start until the first ends. Therefore, before the actual processing starts in a GPU, the CPU control program must select which GPU to be used, to copy data to them, to launch the kernels (with the appropriate scheduling) and to recover the results. It is obvious that the transfers will affect the processing time.

An additional issue is that the GPUs have a stack of memories [16], [14] with different features (size, speed and access). This requires considering where to place and how to access data once they are into the GPU. For instance, figure 4 shows two choices: using shared memory (a) or using texture memory (b).

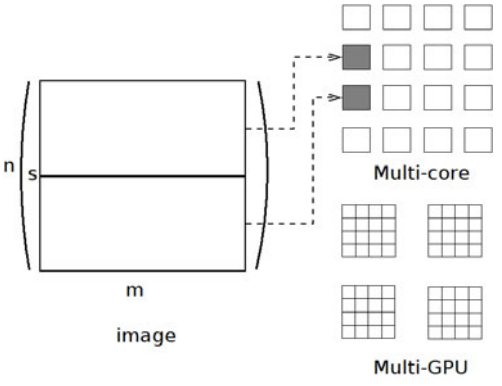

Fig. 1. Image divided in 2 cores, none GPUs used

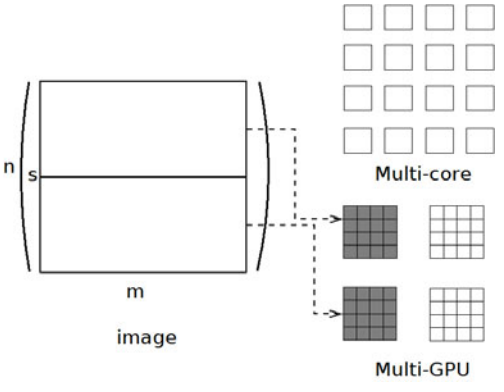

Fig. 2. Image divided in 2 GPUs, none cores used

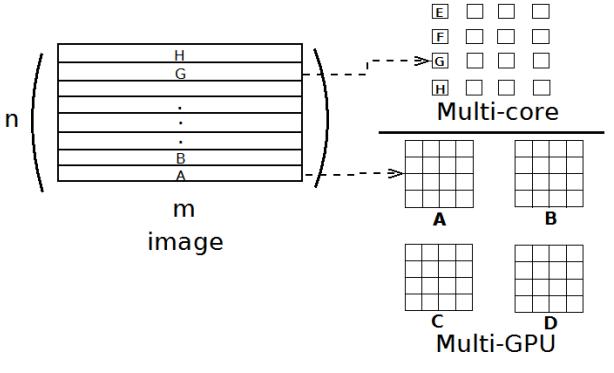

Fig. 3. Image divided in 4 GPUs and 4 cores

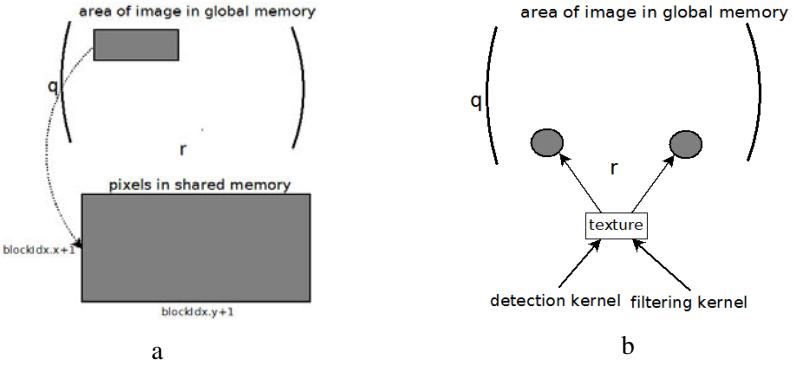

Fig. 4. GPU global memory access with a) shared memory b) texture memory

Fig. 5. Original image of a building

Using shared memory implies that data must be further copied from the GPU global memory to a shared memory block, available only to the set of threads being executed by the same multi-processor in a GPU. Texture memory is used in read-only mode and accessed by the detection and filtering kernels.

4 Experimental Study

We conducted the experimentation on a Mac OS X (Intel Quad-Core Xeon 2 x 2.26 GHz, 8GB of RAM) with 4 NVIDIA GPUs. Each card is a GeForce GT120, 512MB of memory, and 4 multi-processors. The image for the tests is shown in figure 5. This image was taken from the Kodak database ([15]) and it was resized in several sizes.

The first tests performed were to compare the use of shared memory versus texture memory, using 1, 2 and 4 GPUs. Table 1 summarizes the results. This results shows improvements if texture memory is used. Therefore, in the following tests, texture memory is always used.

The following test was the use of the CPU with 16 cores for the noise removal on different sizes of the image. Figure 6 shows the results. For sizes larger than 384x256, we got the best results when all the 16 cores are used. On the contrary, for smaller sizes, it is better using only 8 cores.

Table 1. Results in Mpixels/sec when the image is processed and stored in texture memory and shared memory

Image size	Texture memory	Shared memory	GPU
6144x4096	20.6599	17.3378	1
6144x4096	30.5744	29.4889	2
6144x4096	47.1182	41.3707	4

Now, our study focuses on the effect of using several GPUs. Figure 7 collects the results in function of the number of GPUs used, but only for the execution inside the GPUs, this is, without data transfer RAM-GPU. It is clear that the better outcomes occur when all the GPUs are used (4 in our case). On the other hand, figure 8 shows results for the overall execution, now including the data transfer RAM-GPU. If the image is smaller than 1536x1024 pixels, it is better to use a single GPU because the time used in transfers is not compensated by the use of more GPUs. On the contrary, for larger sizes than 1536x1024, using more GPUs improves the performance despite the memory transfers. Therefore, we conclude the following:

- for image sizes less than 768x512, it's better to use one GPU.
- for image sizes between 768x512 and 3072x2048, it's better to use two GPUs.
- for image sizes greater than 6144x4096, it's better to use four GPUs.

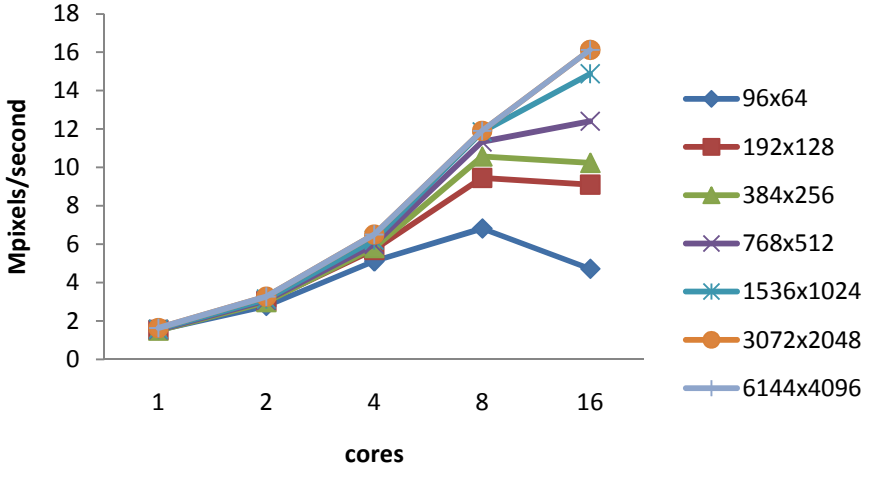

Fig. 6. Parallelizing image with multi-core

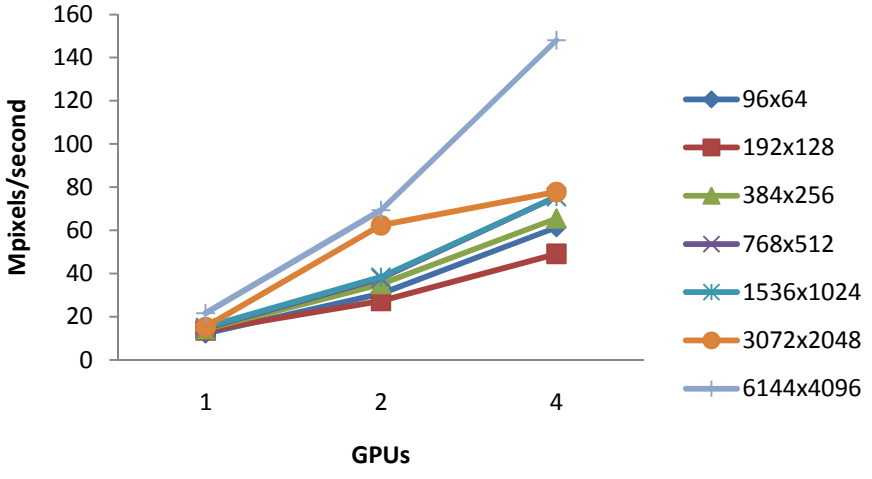

Fig. 7. Mpixels/sec executed by both kernels (no memory transfers accounted)

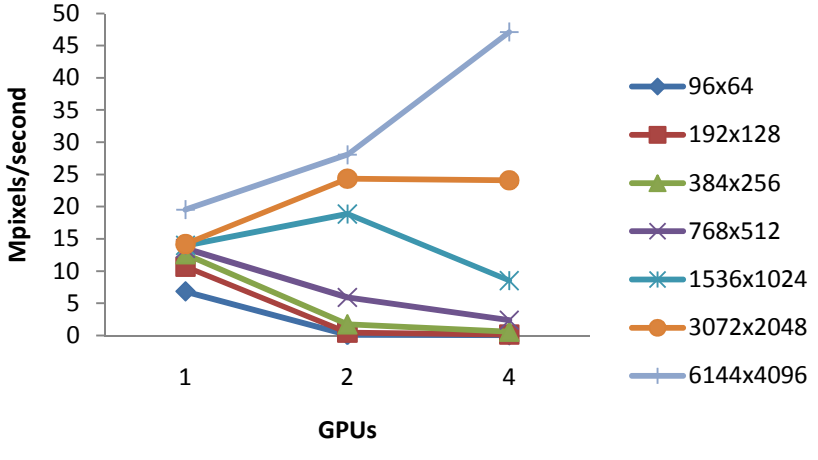

Fig. 8. Mpixels Mpixels/sec executed by both kernels (total processing)

In the last tests, the cores CPU and the GPUs were simultaneously used. Table 2 compares the best results if only the CPU, only the GPU or both are used. For any size of the image, it is better to use the combination of both.

Table 3 shows the combination GPU and CPU, and how many cores and GPUs was used for each size of the image, also the table shows the size of the image assigned to the GPU (the rest is processed by the CPU, obviously). If the image size increases, more processing is done on the GPU and more GPUs are used.

Table 2. Megapixels/sec results for processing using only Multi-core, only Multi-GPU, and both

Image size	Multicore	MultiGPU	Multicore and MultiGPU
96x64	6.8267	6.83	8.3706
192x128	9.4523	10.69	15.5446
384x256	10.5703	12.60	20.0130
768x512	12.4043	13.51	37.1379
1536x1024	14.8805	18.86	42.3953
3072x2048	15.9601	24.34	43.7819
3072x2048	15.9601	24.10	69.9105
6144x4096	16.1206	47.12	68.4600

Table 3. The size of the image assigned to the GPU for processing, the cores and the GPU used

Image size	GPUs	Cores	Size on GPU
96x64	1	16	1/4
192x128	1	11	3/8
384x256	1	7	1/2
768x512	1	9	3/4
1536x1024	2	9	3/4
3072x2048	2	9	3/4
3072x2048	4	11	7/8
6144x4096	4	7	7/8

5 Concluding Remarks

The availability of inexpensive parallel processing hardware provided by graphics processing units has boosted the development of many applications for demanding problems. Image denoising is a problem that fits very well in this scenario because images may be large, the processing is costly, and image pixels, to an extent, can be handled simultaneously.

In this paper we have adapted a denoising algorithm, based on the peer group concept and using a fuzzy metric, to run concurrently. Our implementation was developed to be executed either on a multi-core CPU, on several GPUs, or using the CPU along with the GPUs. The results showed that this latter option (CPU+GPUs) gives the best performance. On the way, we have shown the best settings when using GPUs, because they need a very fine tuning to get the best yield.

The final conclusion is that implementing image denoising algorithms to be run on multi-core CPUs and GPUs are very advisable. This opens the door to use such algorithms for real-time processing. In future works, we plan to test our programs on the last generation GPUs cards and to try other common problems on images, such as edge detection.

Acknowledgments. This work was funded by the Spanish Ministry of Science and Innovation (Project TIN2008-06570-C04-04) and M. Guadalupe would also like to acknowledge DGEST ITCG for the scholarship awarded through the PROMEP program (Mexico).

References

1. Smolka, B.: Peer group switching filter for impulse noise reduction in color images. Pattern Recognition Letters 31, 484–495 (2010)
2. Camarena, J.G., Gregori, V., Morillas, S., Sapena, A.: Fast detection and removal of impulsive noise using peer group and fuzzy metrics. Journal of Visual Communication and Image Representation 19, 20–29 (2008)
3. Toprak, A., Guller, I.: Impulse noise reduction in medical images with the use of switch mode fuzzy adaptive median filter. Digital Signal Processing 17(4), 711–723 (2007)
4. Schulte, S., Nachtegael, M., De Witte, V., Van der Weken, D., Kerre, E.E.: A Fuzzy Impulse Noise Detection and Reduction Method. IEEE Transaction on Image Processing 15, 5 (2006)
5. Shulte, S., Morillas, S., Gregori, V., Kerre, E.E.: A New Fuzzy Color Correlated Impulse Noise Reduction Method. IEEE Transaction on Image Processing 15, 10 (2007)
6. Shulte, S., De Witte, V., Nachtegael, M., Van der Weken, D., Kerre, E.E.: Fuzzy Two Step Filter for Impulse Noise Reduction From Color Images. IEEE Transaction on Image Processing 15, 11 (2006)
7. Shulte, S., De Witte, V., Nachtegael, M., Van der Weken, D., Kerre, E.E.: Fuzzy random impulse noise reduction method. Journal Fuzzy Sets and Systems 158(3) (2007)
8. Mélange, T., Nachtegael, M., Kerre, E.E.: Fuzzy Random Impulse Noise Removal From Colour Image Sequences: IEEE (2010)
9. Morillas, S., Gregori, V., Hervas, A.: Fuzzy Peer Groups for Reducing Mixed Gaussian-Impulse Noise From Color Images. IEEE Transaction on Image Processing 18, 7 (2009)
10. Camarena, J.G., Gregori, V., Morillas, S., Sapena, A.: Some improvements for image filtering using peer group techniques. Image Vis. Comput. 28(1), 188–201 (2010)
11. Morillas, S., Gregori, V., Peris-Fajarnés, G.: Isolating impulsive noise pixels in color images by peer group techniques. Comput. Vis. Image Underst. 110(1), 102–116 (2008)
12. Camarena, J.G., Gregori, V., Morillas, S., Sapena, A.: Two-step fuzzy logic based method for impulse noise detection in colour images. Pattern Recognition Letters 31, 1842–1849 (2010)
13. Smolka, B.: Fast detection and impulsive noise remolval in color images. Real-Time Imaging 11, 389–402 (2005)
14. Sánchez, M.G., Vidal, V., Bataller, J., Arnal, J.: Implementing a GPU fuzzy filter for Impulsive Image Noise Correction. In: CMSSE (2010)
15. Kodak, `http://r0k.us/graphics/kodak/index.html`
16. Nvidia, `http://www.nvidia.es/page/home.html`