Image Noise Removal on Heterogeneous CPU-GPU Configurations

María G. Sánchez¹, Vicente Vidal², Josep Arnal³, and Anna Vidal⁴

¹ Dpto. de Sistemas y Computación, Instituto Tecnológico de Cd. Guzmán, Cd. Guzmán, México
msanchez@dsic.upv.es

² Dpto. de Sistemas Informáticos y Computación, Universidad Politécnica de Valencia, Valencia, Spain
vvidal@dsic.upv.es

³ Dpto. de Ciencia de la Computación e Inteligencia Artificial, Universidad de Alicante, Alicante, Spain
arnal@ua.es

⁴ Dpto. de Matemática Aplicada, Universidad Politécnica de Valencia, Valencia, Spain
avidal@mat.upv.es

Abstract
A parallel algorithm to remove impulsive noise in digital images using heterogeneous CPU/GPU computing is proposed. The parallel denoising algorithm is based on the peer group concept and uses an Euclidean metric. In order to identify the amount of pixels to be allocated in multi-core and GPUs, a performance analysis using large images is presented. A comparison of the parallel implementation in multi-core, GPUs and a combination of both is performed. Performance has been evaluated in terms of execution time and Megapixels/second. We present several optimization strategies especially effective for the multi-core environment, and demonstrate significant performance improvements. The main advantage of the proposed noise removal methodology is its computational speed, which enables efficient filtering of color images in real-time applications.

Keywords: parallel computing, noise removal in images, GPU, CUDA, multi-core, OpenMP

1 Introduction

Noise removal is an important problem in the field of image processing which has many applications in different fields. Very often noise corrupting the image is of impulsive nature. Impulsive noise is commonly caused by the malfunction of sensors and other hardware in the process of image formation, storage or transmission [2, 18]. This type of noise affects some individual pixels, changing their original values. The most usual model of impulsive noise is the Salt and Pepper noise or fixed value noise, which considers that the new, wrong, pixel value is an
extreme value within the signal range. This is the noise type we consider in this paper. Many algorithms have been proposed for correcting impulsive noise, for instance those mentioned in [3–5, 7–9, 14–18, 20]. In the context of color image filtering, one of the most used techniques is based on a vector ordering scheme defined through the ordering of aggregated distances [12]. Filters based on the ordering principle are efficient reducing the impulses, but they do not preserve fine image structures, which are treated as noise. In order to avoid the problems caused by the blurring properties of filters based on the ordering principle, a filtering method based on the concept of peer group was introduced in [6] and widely used in filtering design [3,4,8,10,18,19]. The peer group associated with the central pixel is a set of neighboring pixels from the filtering window $W$ being similar to $x_i$ according to an appropriate metric value, this is, the nearest neighbors. This type of filters have recently shown good results in quality but they do not seem appropriate for real-time processing [3–6, 8, 9, 18, 19]. In this context, parallel computing emerges as a solution to decrease computational time. Nowadays parallel algorithms are widely present in noise removal literature [1,13,21]. In this paper, we introduce a parallel version of peer group based filters in order to retain their good quality results while trying to improve its performance, making them usable in real-time processing. We have tested these parallel algorithms developing programs for GPUs and multi-cores and we did an analysis of the best distribution of pixels in these two devices to take advantage of the hardware.

This paper is organized as follows: Section 2 explains the parallel noise removal method and how the parallel algorithm was implemented on GPUs and multi-core. Experimental results are shown in Section 3, and lastly, the conclusions are presented in Section 4.

## 2 Parallel image noise removal algorithm

Let the color image $A$ be defined as a mapping $\mathbb{Z}^2 \rightarrow \mathbb{Z}^3$. That is, the color image is defined as a two-dimensional matrix $A$ of size $M \times N$ consisting of pixels $x_i = (x_i(1), x_i(2), x_i(3))$, indexed by $i$, which gives the pixel position on the image domain $\Omega$. Components $x_i(l)$, for $i = 1, 2, ..., M \times N$ and $l = 1, 2, 3$, represent the color channel values in RGB quantified into the integer domain. Let $W = \{x_k \in \mathbb{Z}^2, k = 1, 2, ..., n\}$ represents a square filtering window (Figure 3) consisting of $n$ color pixels centered at pixel $x_1$ (in the present study, $n = 3$ was considered). The parallel denoising algorithm introduced in this study uses the peer group of a central pixel $x_i$ in a window $W$ according to [18] and uses an Euclidean metric. In order to describe the parallel algorithm, and how the pixels were assigned to each computing element, we consider a domain decomposition of the image domain $\Omega$ in $P$ subdomains $\{\Omega_{j}\}_{j=1}^{P}$, where $P$ is the number of computing elements. Figure 1 shows an example of the image domain decomposition used in the experiments. Then, to detect and reduce the impulse noise the fuzzy peer group concept is used. For impulse noise reducing, we use the arithmetic mean filter AMF (e.g. [3]). Algorithm 1 shows the parallel filtering algorithm. The following lines detail the two steps of the filter. For a central pixel $x_i$ in a $n \times n$ filtering window $W$ and fixed the distance threshold $d \in [0, 1]$, we denote by $P(x_i, d)$ the set

$$P(x_i, d) = \{x_j \in W : \|x_i - x_j\| \leq d\}. \quad (1)$$

Using the terminology employed in [3, 19], given a non-negative integer $m$, it will be called peer group of $m$ peers associated to $x_i$ a subset $P(x_i, m, d)$ of $P(x_i, d)$ formed by $x_i$ and other $m$ pixels, where $m \leq n^2 - 1$. Note that if $P(x_i, d)$ contains $c + 1$ pixels then $P(x_i, d)$ is a peer group of $c$ peers. The algorithm performs two main steps. In the first step (detection) the pixels are labeled as either corrupted or uncorrupted. In the second step (noise reduction) corrupted pixels are corrected. Then, the detection and filtering steps are described for a single pixel $x_i$:
Algorithm 1 Parallel filtering algorithm.

Require: Image $A$, a domain decomposition $\{A_{\Omega_k}\}_{k=1}^P$, $m$, $d$

Ensure: Filtered image.

1: for $k = 1, \ldots, P$, in parallel do
2: for $x_i$ pixel in $A_{\Omega_k}$ do
3: Impulse noise detection:
4: Calculate $P(x_i, d)$:
5: distance = $\|x_i - x_j\|$
6: if distance $\leq d$ then
7: pixel $x_j \in P(x_i, d)$
8: endif
9: if ($\#P(x_i, d) \geq m + 1$) then
10: pixel $x_i$ is free of impulse noise
11: else
12: Impulse noise reduction:
13: $x_i$ is an impulse and it is replaced with $AMF_{\text{out}}$
14: end if
15: end for
16: end for

Figure 1: Distributed image: (3/4) on 4 GPUs and (1/4) on 8 cores.

- Detection: $x_i$ is declared as corrupted if $\#P(x_i, d) < (m + 1)$, where $m$ is the voting threshold and $\#P$ the cardinality of set $P$.

- Noise reduction: Given a pixel $x_i$ previously marked as corrupted, it is replaced by the arithmetic mean of uncorrupted pixels in its window $W$. This is, the new value for $x_i(l)$ is $\frac{\sum_{x_j \in W'} x_j(l)}{\#W'}$, where $W'$ is the set of uncorrupted pixels of $W$.

2.1 Comments on the GPU and multi-core implementation

We have developed three implementations. The first on multi-core using OpenMP, the second with CUDA on GPUs and the third is a combination of multi-core and GPUs. Figure 1 shows an example of the pixel distribution used in the experiments. The flowchart, Figure 2, shows the elimination of noise with these three implementations. In the detection process described in Algorithm 1, on the GPUs, the kernel was configured so that each thread processed one
The thread corresponding to the pixel \( x_i \) analyzes the \( n \times n \) pixels of \( W \), calculates the peer group and if this satisfies the cardinality \( m + 1 \), the central pixel is diagnosed as uncorrupted; if not, it is diagnosed as corrupted. Given that AMF considers only uncorrupted pixels for mean computation, the noise reduction step cannot start until the detection phase is completed. In consequence, to ensure this synchronisation requirement, in the parallel implementation on GPU we have developed two kernels, so that the noise reduction kernel is not launched until the detecting kernel has finished. In multi-core two separate functions have been implemented. The arithmetic mean was used instead of the median for two main reasons: first, because in the calculation of the median, comparison operations are required and these operations are not recommended on the GPU. Second, because the computational cost of the arithmetic mean is lower than the median. In the GPU implementation, we reserve space in memory using 4 bytes per pixel and we access data through the texture memory. After several tests, this proved to be the best option.

### 2.2 Optimized detection step

To calculate the peer group and the Euclidean metric in the detection step, each pixel \((i, j)\) in the domain \( \Omega \) of the image forms a filtering window \( W \) with \( 3 \times 3 \) pixels (see Figure 3). Euclidean metric considers the distance between the central pixel \((i, j)\) and its 8 neighbors in the window \( W \). Sequential algorithm performs \( M \times N \) cycles, corresponding to the \( M \times N \) image pixels. Figure 4 shows the execution of two cycles. Pixel \((i, j)\) calculates the distance to its 8 neighbors. The value of the pixel \((i, j)\) is used to calculate the distance from its 8 neighbors to it. In Figure 4 can be seen that when analyzing pixel \((3, 4)\), pixels \((2, 3), (2, 4), (2, 5), (3, 3)\) are accessed to calculate the distance between those pixels (Figure 2.2). These distances have been calculated previously, i.e., distance form pixel \((3, 4)\) to \((2, 3)\) is calculated when analyzing pixel \((2, 3)\). Then, in this process some distances are computed more than once. Concretely, distance
computation and data access in the peer group are repeated 4 times per pixel. In order to reduce the computation, 8 distances per pixel, we propose an optimized detection step, in which for each pixel \((i,j)\) only distances to 4 neighbors, \((i-1,j-1)\), \((i-1,j)\), \((i-1,j+1)\), \((i,j-1)\), are computed. Then to compute the cardinality of the peer group associated with the pixel \((i,j)\), we use information computed when analyzing the neighboring pixels. This process can be seen in Figure 5. Then, pixel \((i-1,j-1)\) is classified as corrupt or not after pixel \((i,j)\) is analyzed (see Figure 6).

3 Experimental results and discussion

We carried out specific experiments and developments using a Mac OS X Intel Xeon Quad-Core processor at 2 x 2.26 GHz with 8GB memory and with four NVIDIA GPUs (GeForce GT 120 with 512MB of memory (see [11])). This GPU supports CUDA Compute Capability 1.1. The CUDA toolkit 4.0.50 was used. Our implementation used C language and single-precision calculations. In order to adjust the filter parameters \(d\) and \(m\) the filter performance has been analyzed in terms of PSNR as a function of \(d\) and \(m\) contaminating images with different densities of impulse noise. The best results were obtained when \(d = 0.95\) and \(m = 2\).

The results presented in this paper were obtained using the Lenna image (Figure 1) with RGB format and square dimensions 512 \(\times\) 512, 1024 \(\times\) 1024, 2048 \(\times\) 2048 and 4096 \(\times\) 4096 pixels. These images have been corrupted with 10% impulse noise.

To determine the number of threads per block that best fits the application, a heuristic study concluded that 64 \(\times\) 64 threads per block gave lowest computational costs. Optimized detection step proposed in Section 2.2 has been implemented in parallel on multi-core, but not on GPUs. This is due to the fact that GPUs are particularly useful performing calculations, but they are penalized when memory access is needed. Moreover, optimized detection step algorithm generates many memory accesses conflicts among threads.
(a) Cycle $i$

(b) Cycle $i + 1$, contribution of pixel $(i, j + 1)$ to the cardinality of pixel $(3, 3)$

(c) Contribution of pixel $(i + 1, j - 1)$ to the cardinality of pixel $(3, 3)$

(d) Contribution of pixel $(i + 1, j)$ to the cardinality of pixel $(3, 3)$

(e) Contribution of pixel $(i + 1, j + 1)$ to the cardinality of pixel completed for pixel $(3, 3)$

(f) Distances and cardinality completed for pixel $(3, 3)$

Figure 5: Optimized detection step.

Figure 6: Pixel $(i - 1, j - 1)$ is classified as corrupt or not after pixel $(i, j)$ is analyzed.

Table 1 shows the results obtained on the multi-core dividing the image only in cores. As it can be seen, the less time is presented when the image is divided among the 8 available cores, even for small size images ($512 \times 512$ pixels). Next test consists of dividing image among the available GPUs. As shown in Table 2, the best results parallelizing an image smaller than $2048 \times 2048$ were obtained using 2 GPUs. When the image is larger than $2048 \times 2048$, the best time is obtained by using 4 GPUs. If times obtained on GPUs and CPUs are compared for
Each size, the best results were obtained using 8 cores for images smaller than $4096 \times 4096$, and for images of size $4096 \times 4096$ a better performance was obtained using 4 GPUs.

Table 3 presents the results obtained for image size $4096 \times 4096$ for different combinations of CPUs and GPUs. Table 3 also shows the portion of the image assigned to the GPUs. The rest was processed by the CPUs. Similar experiments were performed for the image size $2048 \times 2048$. For this size, Table 4 shows the best distribution using available hardware.

Figure 7 shows the results obtained when the image is divided into 4 GPUs and all the cores, for different image sizes. Figure 7 shows that the best results for the image $2048 \times 2048$ were obtained assigning $1/2$ of the image in 8 cores and $1/2$ in 4 GPUs. As can be seen in Figure 7, if the image size increases, then the best results were obtained assigning more processing on the GPU. As can be seen in the results, the parallelization performed using a combination of cores and GPUs gives better results than the parallelization performed only in 4 GPUs. For the image of $4096 \times 4096$ pixels, the reduction rate is 9.5% when the CPUs option is compared with the GPUs option, and 24.8% when CPUs model is compared with hybrid CPUs/GPUs model. Figure 8(a) presents time obtained using the optimized detection proposed in Section 2. Detection step is compared with the optimized detection step when the process is executed.
Table 4: Hybrid CPU-GPU. Image size 2048 × 2048.

<table>
<thead>
<tr>
<th>Number of GPUs</th>
<th>Optimal size on GPUs</th>
<th>Optimal number of CPUs</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1/8</td>
<td>8</td>
<td>0.051960</td>
</tr>
<tr>
<td>2</td>
<td>1/2</td>
<td>8</td>
<td>0.049696</td>
</tr>
<tr>
<td>4</td>
<td>3/4</td>
<td>8</td>
<td>0.053220</td>
</tr>
</tbody>
</table>

Figure 7: Hybrid CPU-GPU using 4 GPUs sequentially in one CPU. It can be seen from Figure 8(a) that the improvement is significant. If the image size increases, improvement increases. Figure 8(b) compares Mpixels processed per second. For image of size 4096 × 4096 optimized version of detection step processes 6 Mpixels per second more than non optimized version. Figure 9 presents this comparison in parallel on multi-core. Figure 10 analyzes GFlops performed in the optimized detection step. In the sequential version, for the 4096 × 4096 image, the GFlops decrease from 1.501 to 1.08876 GFlops (27% reduction) when comparing detection step with optimized detection step. For the same image, in the parallel version using 8 cores, the GFlops decrease 41% in the optimized version.

Figure 8: Optimized detection step. Sequential version.
4 Conclusions

A parallel algorithm to remove impulsive noise of a digital image using heterogeneous CPU/GPU computing has been proposed. The denoising parallel algorithm is based on the peer group concept and uses an Euclidan metric. We have implemented it to be run on GPUs using the CUDA library and on multi-cores using OpenMP. This processing was divided into two steps: noise detection and noise elimination. For detection, the Euclidean metric and the concept of peer group were used. In the correction stage, corrupted pixel values were replaced by calculating the mean of those neighbors not labeled as corrupted in the detection step. Three implementations have been developed to be executed either on a multi-core, on several GPUs, or using a combination of CPUs and GPUs. Results showed that hybrid implementation CPU/GPU obtains the best performance. Several optimization strategies especially effective for the multi-core environment have been presented, demonstrating significant performance improvements. Numerical experiments show the efficient computational speed of the proposed noise removal methodology, which enables efficient filtering of color images in real-time applications.
5 Acknowledgements

This work was supported by the Spanish Ministry of Science and Innovation [grant number TIN2011-26254].

References


