J Sign Process Syst
DOI 10.1007/s11265-012-0715-1

Highly Parallelable Bidimensional Median Filter for Modern Parallel Programming Models

Ricardo M. Sánchez · Paul A. Rodríguez

Received: 22 June 2012 / Revised: 29 October 2012 / Accepted: 31 October 2012

Abstract The median filter is a non-linear filter used for removal of Salt & Pepper noise from images, where each pixel of the image is replaced by the median of its surrounding elements, which is calculated by sorting the data. The complexity of the sorting algorithms used for the median filters are $O(n^2)$ or $O(n)$, depending on the kernel size. These algorithms were formulated for scalar single processor computers, with few of them successfully adapted and implemented for computers with a parallel architecture. In this paper we greatly improve the results of our earlier work, in which by means of a novel sorting algorithm, based on the Complementary Cumulative Distribution function, with $O(n)$ computational complexity and a highly parallelable structure, we presented a 2D median filter that achieved $O(1)$ or $O(n)$ computational complexity, depending memory constraints. The improvements are twofold: we propose a trade-off between $O(1)$ complexity and $O(n)$ complexity in order to improve the overall throughput; additionally we make use of the Salt & Pepper noise model to improve the image reconstruction quality with a small performance impact. The proposed algorithm have been implemented in three parallel programming models: SIMD Intel, Multicore Intel with SIMD, and SIMT (CUDA), achieving a peak throughput of 27.0, 100.1 and 91.6 megapixels per second respectively.

Keywords Nonlinear filters · Parallel algorithms · Image processing

R. M. Sánchez (✉) · P. A. Rodríguez
Department of Electrical Engineering,
Pontificia Universidad Católica del Perú, Lima, Perú
e-mail: ricardo.sanchez@pucp.pe

P. A. Rodríguez
e-mail: prodrig@pucp.pe

1 Introduction

The median filter is a basic operation for digital image processing, as it removes Salt & Pepper noise while preserving the edges of the image. This type of noise is usually generated by lost packets on digital transmission, noisy digital channels, or errors on the acquisition system (dust on the lens, defective pixels on the sensor) [2]. However, the usage of the 2D median filter (originally proposed in 1974 [23]) is restricted by its high computational cost and its non-linear nature.

The algorithms for median filtering differ one from each other by the sorting algorithm that each one use. For large kernels it is typical to use sorting algorithms with $O(n)$ computational complexity, where n is the kernel size. Those algorithms require additional memory for data structures or containers. Also, they can reduce memory access by keeping the previous results and use them to calculate the median of the next pixel [3, 10, 11]. Furthermore, $O(1)$ computational complexity can be obtained by storing partial results of the containers needed to get the median value [6, 17]. For small kernels, $O(n^2)$ sorting algorithms are faster than the $O(n)$ ones, and some of them make use of vector processors to achieve a very high performance [9, 14, 20]. This improvement is due to the data parallelism of the underlying sorting algorithm, as the Sorting Network [20], that allows to sort consecutive datasets with common elements. Parallel sorting algorithms, for multiprocessors systems, have been

Published online: 21 December 2012

proposed, but they have high communication cost [15] or unbalanced computational load [4].

In this paper we greatly improve the results of our earlier work [22] where we proposed a novel sorting algorithm with $O(n)$ computational complexity that can be implemented efficiently on modern parallel programming models. This algorithm was based on the Complementary Cumulative Distribution Function ($ccdf$) [19] and it had similar properties to the ones based on probability mass function (pmf or histograms). With this sorting algorithm we developed a new median filter algorithm well suited to modern parallel programming models, such as SIMD (single instructions multiple data) [7], Multicore processors, or SIMT (single instructions multiple threads, also known as CUDA) [8, 13] models, and is benefited by the optimizations originally proposed for the histogram (pmf) based algorithms (i.e.: Constant-Time Median Filter [17]), achieving $O(1)$ or $O(n)$ computational complexity, depending memory constraints. In this paper we propose two key improvements: first we explore a trade-off between $O(1)$ complexity and $O(n)$ complexity in order to improve the overall throughput; additionally we make use of the Salt & Pepper noise model to improve the image reconstruction quality with a small performance impact. The implementation of the new median filter (and its variants) has been developed for CUDA-enabled graphics card and for Intel processors, as each one offer different parallel architectures.

This paper is organized as follows: in Section 2 we give a succinctly description of the three parallel programming models (SIMD, Multicore Shared Memory and SIMT) that we target in this paper; then in Section 3 the novel sorting algorithm and the median filter algorithms are described, along with the proposed variants that make use of the Salt & Pepper noise model. In Section 4 we present the complexity analysis and evaluate the parallel capabilities of our proposed algorithm in comparison with histogram based algorithms. We describe the median implementations used and show the computational results on Section 5. On Section 6 we discuss the results and give our concluding remarks.

2 Modern Parallel Programming Models

Innovations in hardware architecture have made possible a large (and cheap) amount of parallel computing options. The focus of this section is to give some details on three particular modern parallel programming models: the SIMD (single instructions multiple data), the shared memory (multicore) model and the SIMT (single instructions multiple threads) or CUDA models. For a broader introduction to the topic of parallel programming models we recommend [18] and the many references therein.

2.1 SIMD Model

In general purpose microprocessors, the SIMD unit is independent from the standard arithmetic logic unit (ALU), and the floating-point unit (FPU). The main difference among the SIMD, the ALU, and the FPU units is that while the former two operate over a scalar value at a time, the SIMD unit operates over a vector of scalars. For the current, general-purpose microprocessors, SIMD vectors can consist of integer scalars, or four/eight single-precision floating point, or two/four double-precision floating point numbers. For integer scalars, SIMD vectors can be: (i) 16/32 8-bit integers (or 16/32 chars), (ii) 8/16 16-bit integers (or 8/16 shorts), (iii) 4/8 32-bit integers (or 4/8 integers), or (iv) 2/4 64-bit integers (or 2/4 long long integers). SIMD units have an inherent parallelism at a fine-grain level since they perform the same operation on all the elements of an array.

Any SIMD capable processor has a set of special registers, whose characteristics (length and number) are architecture dependent. Particularly of the Intel architecture we have: (i) 8 simd-registers of 128-bit long for Intel's IA32 SSE/SSE2/SEE3, (ii) 16 simd-registers of 128-bit long for Intel's IA_64 SSE4.x and (iii) 16 simd-registers of 256-bit long for Intel's IA_64 AVX (advanced vector extensions). We must point out that, for the AVX, only the floating point operations were scalated to use the 256-bit long registers, integer operations still use 128-bit long registers.

The SIMD execution model operates over packed data elements (or scalar data) which could be located in memory or in a SIMD register. A packed data element is a vector with S contiguous elements. Let $S = 4$ and $X = [x_1, x_2, x_3, x_4]$, $Y = [y_1, y_2, y_3, y_4]$ be two packed data elements. Also let **op** be a SIMD math operation. Then a SIMD operation is given by $Z = X \mathbf{op} Y = [x_1 \mathbf{op} y_1, x_2 \mathbf{op} y_2 x_3 \mathbf{op} y_3, x_4 \mathbf{op} y_4]$.

Finally we stress that memory data access has a great impact on the performance of any SIMD application (load a simd-register with memory data and vice versa). Addressed memory should be 16-bit aligned (32-bit aligned for the AVX case) and data elements also should be contiguous in memory.

2.2 Shared Memory Model

Nowadays ubiquitous multicore (desktop and laptops) computers are an example of computers with a shared memory organization where several processors (usually between 4 and 8) physically shared a global (RAM) memory. In this

subsection, we succinctly summarize the key highlights of the shared memory model (for a more detail description, we recommend [18, Ch. 6]).

A natural programming model for this type of architecture (multicore computers) is the use of a thread model in which all the threads have access to the shared variables, which are used to interchange information and data among the running threads. For a given problem, ideally the shared memory model assumes that the problem can be divided into sub-problems and the work require for each (subproblem) will be completed mostly independently by each thread. In order to avoid race conditions in concurrent accesses, the model defines several synchronization mechanisms (where particular implementations depend on the chosen programming environment).

There are several programming environment options, such as Posix threads (Pthreads), Java threads, OpenMP, etc., to fully exploit the advantages of the multicore architecture; in this work we have chosen the Pthreads standard as our programming environment.

2.3 SIMT Model/CUDA Architecture

There are several good tutorials, articles and/or books that deal with the SIMT Model (or CUDA architecture). Due to space considerations we only list [8, 13], and will proceed to give a brief overview of the SIMT model/CUDA architecture.

In Fig. 1 we show a CUDA device memory model where different types of memory (Global, "Pitch", "Array") have been depicted as independent; here we stress that this is not necessarily represents an actual hardware. Also from a more general fashion, Fig. 1 shows the standard configuration of a general purpose computation system based on GPU (graphics processor unit) which consist of a host (CPU) that can communicate to a device (a GPU, such cards with the CUDA technology) via a given interface (such the PCI interface).

For a given problem, the SIMT model assumes that the problem is divided into *blocks* (to be understood as a bidimensional partition of the problem) and in turn the work required to be completed by each *block* (sub-problem) will be carried out by a group of *threads* (to be understood as a bidimensional or tridimensional partition of the sub-problem). Each *thread* within the same block has access to local registers and can communicate with other *threads* (in the same *block*) via the shared memory; if *threads* from different *blocks* need to communicate among them, they must use the Global memory (or the "Pitch" memory, see Fig. 1). The SIMT model executes the same instruction for all the

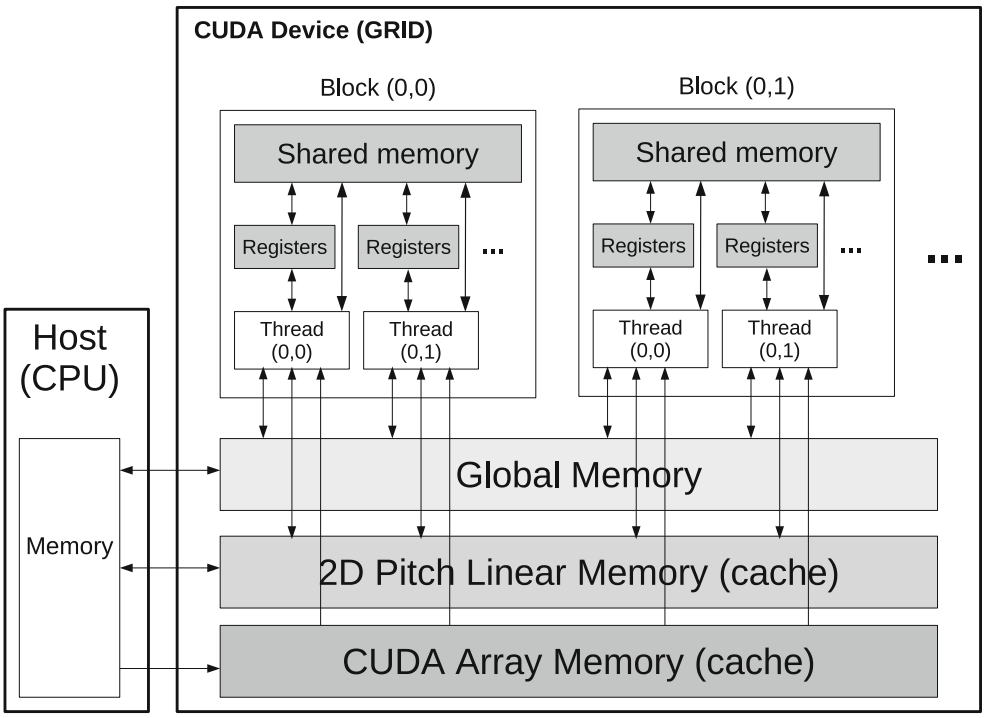


Figure 1 We depict a CUDA device memory model that summarizes our description. We stress that this model does not represent an actual hardware. This figure is a free adaption of [13, Fig. 3.7].

threads in a particular group (called *warp* in the CUDA lexicon, typically consisting of 32 simultaneous threads); this is most efficient when all the *threads* follow the same control flow path or have a balanced load.

As shown in Fig. 1 the CUDA (SIMT) device memory model supports several types of memories that have different scopes. Registers which is the fastest memory (roughly $8e3$ GB/s theoretical bandwidth [8]) are only visible to individual *threads*; the shared memory is visible to all threads within the same *block* and is considered to be fast (roughly $1.6e3$ GB/s theoretical bandwidth [8]) Global memory, as well as "Pitch" and "Array" memory are the only way data can be transfer between the Host and the Device, although there are several key points to have in mind: the Global memory is R/W (read/write) and can be accessed by all *threads*, nevertheless is comparatively slow (roughly 177 GB/s theoretical bandwidth [8]) with respect to all other type of memories. "Pitch" (R/W by all *threads*) and "Array" (R by all *threads*) memories are both examples of Texture memory and are cache optimized for spatial locality; they can be substantially faster than the Global memory if memory accesses are well organized. Whenever possible it is desirable that a program written for the SIMT programming model will use/map input data to an "Array" memory and write the output (and intermediate) data to a "Pitch" memory.

3 Algorithms Description

3.1 CCDF-sorting

The (Complementary Cumulative Distribution Function) CCDF-sorting algorithm was originally introduced in [22]; on what follows we succinctly describe its key features (for details see [21]). This new algorithm for sorting data needs to generate a vector with the complementary cumulative distribution function from the dataset to be sorted; using this (auxiliary) vector we can obtain the kth biggest number in the data set.

Given the vector $\mathbf{x} = [x_0, x_1, \ldots, x_{n-1}]$ with n elements, where $x_i \in \mathbb{N}$ and $a \leq x_i \leq b$, the complementary cumulative distribution function, or reliability function [19], is defined by:

$$\bar{F}_{\mathbf{x}}(j) = 1 - F_{\mathbf{x}}(j) = 1 - Pr(\mathbf{x} \leq j) = Pr(\mathbf{x} > j)$$

where $F_{\mathbf{x}}(j)$ is the cumulative distribution function of the vector $\mathbf{x}$ and $Pr(\mathbf{x} > j)$ is the probability of $x_i > j$. For the sorting algorithm we replace the probability function $Pr(\mathbf{x} > j)$ with a counting function $C_j(\mathbf{x})$:

$$C_j(\mathbf{x}) = \sum_{i=0}^{n-1} I_{[x_i > j]} \tag{1}$$

where $I_{[x_i > j]}$ is the indicator function. It is straightforward to show that $\bar{F}_{\mathbf{x}}(j)$ monotonically decreases to zero. Finally, in order to obtain the sorted vector $\mathbf{y}$, we need to set the auxiliary vector $\boldsymbol{\tau} = \{\tau_j\} = ccdf(\mathbf{x}), j \in [a, b], \tau_j = C_j(\mathbf{x}) \in [0, n]$. Then $\mathbf{y} = \{y_k\} = ccdf(\boldsymbol{\tau}), k \in [0, n-1]$, $y_k = C_k(\boldsymbol{\tau}) \in [a, b]$ is the sorted vector of $\mathbf{x}$ [22].

In order to illustrate the CCDF-sorting algorithm we consider the following example: given the set $\mathbf{x} = [4, 6, 2, 9, 8]$, with $n = 5$, $a = 0$ and $b = 10$, we first compute $\boldsymbol{\tau} = ccdf(\mathbf{x})$ which can be understood the summation of the vectors resulting from applying the Counting function to each element of $\mathbf{x}$: $v_k = [I_{[x_k>0]}, I_{[x_k>1]}, \ldots I_{[x_k>9]}]$ for $k \in [0, 4]$ in this example; thus we obtain $\boldsymbol{\tau} = [5, 5, 4, 4, 3, 3, 2, 2, 1, 0, 0] = [1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0] + [1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0] + [1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0] + [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0] + [1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0]$. Finally if we apply the Counting function to $\boldsymbol{\tau}$ the obtain the sorted vector $\mathbf{y} = [9, 8, 6, 4, 2]$. This example also give us a hint regarding how to parallelize the CCDF-sorting algorithm: the computation of the Counting function for each element in a vector can be fully parallelize as well as the summation that follows. In Section 4 we formalize the complexity analysis and parallel properties of the CCDF-sorting algorithm.

3.2 Median Filter with CCDF-sorting

To obtain the kth biggest element of vector $\mathbf{x}$ (with n elements, where n is assume to be odd) from the auxiliary vector $\boldsymbol{\tau}$ we only need to calculate $C(\boldsymbol{\tau} > k)$. Following this, if we require the minimum, maximum and median value we need to set $k = n - 1$, $k = 0$ and $k = (n-1)/2$ respectively.

An important property of the $ccdf$ is the separability. Given the set $\mathbf{x}$, with $n_{\mathbf{x}}$ elements and the subsets $\mathbf{a}, \mathbf{b}$ with $n_{\mathbf{a}}, n_{\mathbf{b}}$ elements respectively, such that $n_{\mathbf{x}} = n_{\mathbf{a}} + n_{\mathbf{b}}$, $\mathbf{x} = \mathbf{a} \cup \mathbf{b}$, and an additional dataset $\mathbf{y}$, with $n_{\mathbf{y}}$ elements and the subset $\mathbf{c}$ with $n_{\mathbf{c}}$ elements, such that $n_{\mathbf{y}} = n_{\mathbf{b}} + n_{\mathbf{c}}$, $\mathbf{y} = \mathbf{b} \cup \mathbf{c}$, it can be shown that (see [21] for details)

$$\boldsymbol{\tau}^{(\mathbf{x})} = \boldsymbol{\tau}^{(\mathbf{a})} + \boldsymbol{\tau}^{(\mathbf{b})}, \tag{2}$$

$$\boldsymbol{\tau}^{(\mathbf{y})} = \boldsymbol{\tau}^{(\mathbf{x})} + \boldsymbol{\tau}^{(\mathbf{c})} - \boldsymbol{\tau}^{(\mathbf{a})}. \tag{3}$$

This property allows us to reduce memory access by keeping a $\boldsymbol{\tau}_{acc}$ and updating it for the next pixel. This approach was originally proposed for the histogram-based median filter [11] and it can be applied for the $ccdf$-based median filter as well.

Algorithm 1: Parallel Ccdf-based Median Filter (PCMF)

Input: I: $N \times M$ matrix to be filtered.
Kernel size: $k \times k$, k odd.
Output: O: $N \times M$ filtered matrix
begin
  foreach *i-th Column in* I **do**
    $x_{i,l} \leftarrow \{I_{i,n} : |n - l| \leq k\}$
    $\boldsymbol{\tau}_i \leftarrow$ `CCDF(`$\mathbf{x}_i$`)`
  foreach *j-th Row in* I **do**
    $\boldsymbol{\tau}_{acc}^{(i,j)} \leftarrow \sum \boldsymbol{\tau}_n, \quad n = -k : k$ `// Init.` τ_{acc} `for pixel` (i,j) `(Fig. 2(a))`
    foreach *i-th Pixel in j-th Row* **do**
      $O_{i,j} \leftarrow$ `KthFromCCDF(`$\boldsymbol{\tau}_{acc}^{(i,j)}, \frac{k^2-1}{2}$`)`
      `UpdateCCDFfromTempVectors(`$\boldsymbol{\tau}_{acc}^{(i,j)}$`)` `// According to Fig. 2(b)`
    foreach *i-th Column in* I **do**
      `UpdateCCDFfromMatrix(`$\boldsymbol{\tau}_i$`)` `// According to Fig. 2(c)`

Another approach for median filtering is to generate partial histograms for a whole row of the image and getting the kernel's histogram using those partial histograms only. For the next row the partial histograms are updated (Fig. 2) [17]. This procedure can be also applied for the ccdf-based median filter. The computational complexity is lowered and the memory access are reduced, but additional memory is required. Algorithm 1 describes the proposed algorithm for median filtering. The function `KthFromCCDF(`$\boldsymbol{\tau}, k$`)` extract the kth biggest element from vector $\boldsymbol{\tau}$.

3.3 Salt & Pepper Noise Model and Median-type Noise Detector

In the previous subsection we described the median filter algorithm. In this subsection we describe the Salt & Pepper noise model and derive a procedure to improve the denoising capabilities of the median filter.

An observed grayscale image I corrupted with Salt & Pepper noise is characterized by

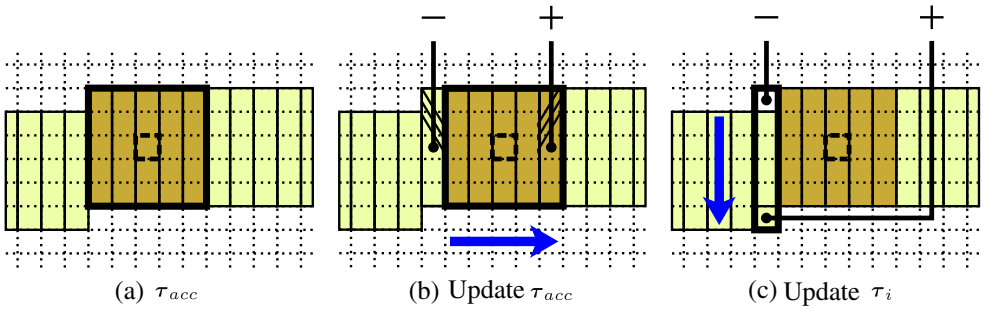
$$I_{i,j} = \begin{cases} \mathrm{v}_{\min}, & \text{with probability } p_1 \\ \mathrm{v}_{\max}, & \text{with probability } p_2 \\ I^*_{i,j}, & \text{with probability } 1 - p_1 - p_2 \end{cases} \tag{4}$$

where $I^*_{i,j}$ describes a pixel in the noise-free image I^*, $p = p_1 + p_2$ represents the noise level and $\mathrm{v}_{\min}$ and $\mathrm{v}_{\max}$ are assumed to be 0 and 255 respectively.

Ideally, for the Salt & Pepper noise model, noise-free pixels should preserve their values in the reconstructed image; this is not the case for the standard 2D median filter, where all the pixels are replaced by the median value (even if they are known to be noise-free). There are several approaches to solve this problem (preserve noise-free pixels in the Salt & Pepper noise scenario), and while this is not the main topic of this paper, particularly we mention that in [12] a median-type noise detector was developed and used to propose an adaptive median filter which we succinctly described next. Let the set of noise-corrupted pixel of the observed image I be defined by

$$\mathcal{N} : \left\{ (k,l) \in \Omega : \left(\hat{I}_{k,l}^{w^{(k,l)}} \neq I_{k,l} \right) \wedge \left(I_{k,l} \in \{\mathrm{v}_{\min}, \mathrm{v}_{\max}\} \right) \right\}, \tag{5}$$

where $\hat{I}_{k,l}^{w^{(k,l)}}$ is the median of all the elements that are no further away than $w^{(k,l)}$ pixels from location (k,l). Initially $w^{(k,l)} = 1 \ \forall (k,l)$; $w^{(k,l)}$ is increased (plus 1) if $\hat{I}_{k,l}^{w^{(k,l)}}$ is equal to the minimum or maximum value in the analyzed set, and the procedure is repeated until $w^{(k,l)} = w_{\max}$ or the pixel is declared noise-free.



(a) τ_{acc} (b) Update τ_{acc} (c) Update τ_i

Figure 2 $O(1)$ complexity approach (originally proposed by [17]).

In the Section 5.2 we will use the idea of the adaptive median filter (originally described in [12]) taking only one iteration, e.g. we will assume that $w_{\max}$ is equal to the initial window size $w^{(k,l)}$ (which is not necessarily equal to 1). In other words, if the central pixel (of the analyzed neighborhood or set) is not the minimum or maximum value in the analyzed set, then it is preserved otherwise it is replaced by its median value; this approach greatly improve the reconstruction performance with a small impact on the computational performance of our proposed implementation.

If we consider $v_{\min} = 0$ and $v_{\max} = 255$ we have a simple (naive) noise detector that can be implemented easily by modifying slightly the PCMF algorithm. Algorithm 2 (called loPCMF) shows the modification of Algorithm 1 in order to consider the noise detector. Experimental results (see Section 5.2) shows that there is no performance penalty when this noise detector is used.

Another approach is to consider the first step of the adaptive median filter. This uses the minimum and the maximum value of the window from which we computed the median value to detect if the pixel is a noisy one. Algorithm 3 (called spPCMF) describes this procedure; furthermore, in order to give computational support to the worthiness of this approach we have performed the following simulation: we corrupted several images (due to space constrains we

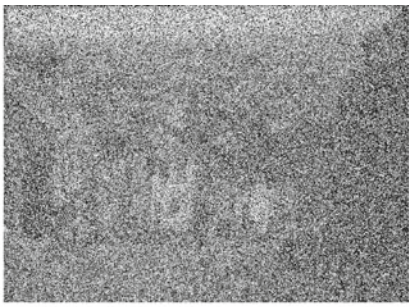
Algorithm 2: local PCMF (loPCMF)

Input: I: $N \times M$ matrix to be filtered.
Kernel size: $k \times k$, k odd.
Output: O: $N \times M$ filtered matrix
begin
  foreach *i-th Column in I* **do**
    $x_{i,l} \leftarrow \{I_{i,n} : |n - l| \leq k\}$
    $\boldsymbol{\tau}_i \leftarrow$ `CCDF(`$\mathbf{x}_i$`)`
  foreach *j-th Row in I* **do**
    $\boldsymbol{\tau}_{acc}^{(i,j)} \leftarrow \sum \boldsymbol{\tau}_n, \quad n = -k : k$ `// According to Fig. 2(a)`
    foreach *i-th Pixel in j-th Row* **do**
      $m_{\boldsymbol{\tau}} \leftarrow$ `KthFromCCDF(`$\boldsymbol{\tau}_{acc}^{(i,j)}, \frac{k^2-1}{2}$`)`
      $O_{i,j} \leftarrow I_{i,j}$
      if $m_{\boldsymbol{\tau}} \neq I_{i,j} \wedge (I_{i,j} = 0 \vee I_{i,j} = 255)$ **then**
        $O_{i,j} \leftarrow m_{\boldsymbol{\tau}}$
      `UpdateCCDFfromTempVectors(`$\boldsymbol{\tau}_{acc}^{(i,j)}$`)` `// According to Fig. 2(b)`
    foreach *i-th Column in I* **do**
      `UpdateCCDFfromMatrix(`$\boldsymbol{\tau}_i$`)` `// According to Fig. 2(c)`

have only considered three typical test images: Goldhill, Lena and Peppers) with Salt & Pepper noise with noise levels of 10, 40 and 70 %; then we proceeded to filter them using the standard median filter with window sizes of 3×3, 5×5 and 7×7 and with the proposed "first step" adaptive median filter (Algorithm 3) with $w = 2$ and $w_{\max} = 2$ (giving an adaptive window of 5×5). In Fig. 3 we show the results obtained for the Goldhill image, where it is clear that the "first step" adaptive median filter (Algorithm 3) gave superior reconstruction quality (measured with three different metrics: signal-to-noise ratio, $\text{SNR} = 10\log_{10}\frac{\text{VAR}\{I\}}{\|I-I^*\|_2^2}$, peak signal-to-noise ratio, $\text{PSNR} = 10\log_{10}\frac{\text{N}(\max\{I^*\})^2}{\|I-I^*\|_2^2}$, and SSIM [24]). Additionally, in Table 1 we list the reconstruction quality metrics for all the considered cases for the test images (Goldhill, Lena and Peppers) from where it also clear that Algorithm 3 gave superior reconstruction quality for all the considered cases.

(a) Image corrupted with Salt & Pepper noise 10%. SNR=2.91, PSNR=14.86 and SSIM=0.197

(b) Image corrupted with Salt & Pepper noise 40%. SNR=−3.11, PSNR=8.84 and SSIM=0.035



(c) Image corrupted with Salt & Pepper noise 70%. SNR=−5.57, PSNR=6.38 and SSIM=0.012

(d) Median filter 3×3. SNR=20.09, PSNR=32.04 and SSIM=0.887

(e) Median filter 5×5. SNR=14.64, PSNR=26.59 and SSIM=0.737

(f) Median filter 7×7. SNR=6.44, PSNR=18.39 and SSIM=0.470

(g) Adative Median filter with $w = 2$ and $w_{max} = 2$. SNR=21.79, PSNR=33.74 and SSIM=0.944

(h) Adative Median filter with $w = 2$ and $w_{max} = 2$. SNR=18.27, PSNR=30.22 and SSIM=0.884

(i) Adative Median filter with $w = 2$ and $w_{max} = 2$. SNR=12.92, PSNR=24.88 and SSIM=0.714

Figure 3 We present the Goldhill image corrupted with different levels (10, 40 and 70 %) of Salt & Pepper noise in sub figures **a**, **b** and **c**. Each corrupted image was filtered with a median filter with window size of 3×3, 5×5, 7×7 and with the "first step" adaptive median filter with $w = 2$ and $w_{\max} = 2$; in sub figures **d**, **e** and **f** we present the best result for the (standard) median filter and in sub figures **g**, **i** and **j** we present the result of the 'first step" adaptive median filter. The latter group has far better reconstruction quality.

Table 1 We report the SNR, PSNR and SSIM [24] reconstruction quality metrics after filtering the corrupted (with Salt & Pepper noise levels of 10, 40 and 70 %) test images with the standard median filter with window sizes of 3×3, 5×5 and 7×7 and with the proposed "first step" adaptive median filter (Algorithm 3, called spPCMF) with $w = 2$ and $w_{\max} = 2$.

Image	Noise	SNR (dB)				PSNR (dB)				SSIM Index [24]			
	(%)	3×3	5×5	7×7	spPCMF	3×3	5×5	7×7	spPCMF	3×3	5×5	7×7	spPCMF
Goldhill	10	20.09	17.43	15.74	21.79	32.04	29.38	27.69	33.74	0.887	0.786	0.709	0.944
	40	6.71	14.64	14.08	18.27	18.66	26.59	26.03	30.22	0.485	0.737	0.686	0.884
	70	−2.52	1.76	6.44	12.92	9.43	13.71	18.39	24.88	0.052	0.235	0.470	0.714
Lena	10	18.67	15.82	13.96	22.51	33.21	30.35	28.50	37.04	0.911	0.858	0.814	0.953
	40	4.37	10.89	9.97	16.47	18.91	25.42	24.51	31.01	0.455	0.813	0.792	0.907
	70	−4.50	−0.45	3.48	10.07	10.03	14.08	18.01	24.61	0.053	0.228	0.527	0.762
Peppers	10	19.23	17.78	16.07	20.46	32.72	31.27	29.56	33.94	0.878	0.847	0.823	0.934
	40	5.35	12.29	11.52	16.15	18.84	25.78	25.00	29.64	0.448	0.802	0.799	0.892
	70	−3.63	−0.40	4.33	10.34	9.86	13.89	17.82	23.83	0.050	0.230	0.533	0.754

For all cases, Algorithm 3 gave superior results.

Algorithm 3: Salt-and-Pepper PCMF (spPCMF)

Input: I: $N \times M$ matrix to be filtered.
Kernel size: $k \times k$, k odd.
Output: O: $N \times M$ filtered matrix
begin
  foreach *i-th Column in* I **do**
    $x_{i,l} \leftarrow \{I_{i,n} : |n - l| \leq k\}$
    $\boldsymbol{\tau}_i \leftarrow$ `CCDF(`$\mathbf{x}_i$`)`
  foreach *j-th Row in* I **do**
    $\boldsymbol{\tau}_{acc}^{(i,j)} \leftarrow \sum \boldsymbol{\tau}_n, \quad n = -k : k$ `// According to Fig. 2(a)`
    foreach *i-th Pixel in j-th Row* **do**
      $med_{\boldsymbol{\tau}} \leftarrow$ `KthFromCCDF(`$\boldsymbol{\tau}_{acc}^{(i,j)}, \frac{k^2-1}{2}$`)`
      $min_{\boldsymbol{\tau}} \leftarrow$ `KthFromCCDF(`$\boldsymbol{\tau}_{acc}^{(i,j)}, k^2 - 1$`)`
      $max_{\boldsymbol{\tau}} \leftarrow$ `KthFromCCDF(`$\boldsymbol{\tau}_{acc}^{(i,j)}, 0$`)`
      $O_{i,j} \leftarrow I_{i,j}$
      if $med_{\boldsymbol{\tau}} \neq I_{i,j} \wedge (I_{i,j} = min_{\boldsymbol{\tau}} \vee I_{i,j} = max_{\boldsymbol{\tau}})$ **then**
        $O_{i,j} \leftarrow med_{\boldsymbol{\tau}}$
      `UpdateCCDFfromTempVectors(`$\boldsymbol{\tau}_{acc}^{(i,j)}$`)` `// According to Fig. 2(b)`
    foreach *i-th Column in* I **do**
      `UpdateCCDFfromMatrix(`$\boldsymbol{\tau}_i$`)` `// According to Fig. 2(c)`

Finally, we point out that the improved reconstruction quality comes with a cost: in Section 5.2 our computational performance results shows that the throughput of Algorithm 3 is reduced between 20 and 30 % (depending on the kernel's size) when compared to Algorithm 1 (or Algorithm 2). Although the authors believe that this trade-off is always worthy, we are also aware that its real value is problem dependent.

4 Algorithm Analysis

4.1 Computational Complexity

In what follows we analyze the computational complexity of the functions `CCDF(x)` and `KthFromCCDF(`$\boldsymbol{\tau}_{\mathbf{x}}$`)` (see Algorithm 1), since our proposed algorithm is based on this two basic operations.

- `CCDF(x)`: From Eq. 1, to compute the $ccdf$ of a vector $\mathbf{x}$ of n elements, with $a \leq x_i \leq b$, the number of operations needed is:

$$\begin{aligned}\boldsymbol{\tau}_{\mathbf{x}} &= (b-a)((n-1)\text{ additions} + n\text{ comparisons})\\ &= O(n)\end{aligned}$$

- `KthFromCCDF(`$\boldsymbol{\tau}_{\mathbf{x}}$`)`: Given $\boldsymbol{\tau}_{\mathbf{x}}$, the number of operations needed to extract the kth biggest value is

$$\begin{aligned}\texttt{KthFromCCDF}(\boldsymbol{\tau}_{\mathbf{x}}) &= (b-a-1)\text{ additions}\\ &\quad + (b-a)\text{ comparisons}\\ &= O(1).\end{aligned}$$

- Median value ($k = \frac{n^2-1}{2}$) of a vector $\mathbf{x}$: overall its complexity is given by

$$median(\mathbf{x}) = O(n) + O(1) = O(n).$$

With these results, given I, an $N \times M$ image, the computational complexity of the PCMF (Algorithm 1) for the whole image is:

$$\underbrace{N \cdot O(n^2)}_{\text{Initialize all } \boldsymbol{\tau}} + \underbrace{M \cdot O(n)}_{\text{Initialize } \boldsymbol{\tau}_{acc} \text{ per Row}} + \underbrace{N \cdot O(1)}_{\text{Median Values}} + \underbrace{N \cdot M \cdot O(1)}_{\text{Update } \boldsymbol{\tau}} = O(n^2)$$

As the initialization stage takes place only one time per row, a better representation for the asymptotical behavior of the algorithm is to find its complexity per pixel. This value is:

$$\begin{aligned}\frac{\text{Complexity}}{\#\text{ of Pixels}} &= \frac{N \cdot O(n^2) + M \cdot O(n) + 2 \cdot N \cdot M \cdot O(1)}{N \cdot M}\\ &= 2 \cdot O(1) + \frac{O(n^2)}{M} + \frac{O(n)}{N}\end{aligned}$$

Then, for a big image (large N and M), the factors $\frac{O(n^2)}{M}$ and $\frac{O(n)}{N}$ tends to 0. This consideration allows us to say that the complexity of the PCMF is $O(1)$. From these results, if we want to decrease the $O(n^2)$ part of the algorithm we must increase the iterations in M. In a similar way, if we want to decrease the $O(n)$ part, we must increase the iterations in N. These observations have a direct relation in the performance of the algorithm in massive parallel architectures, like CUDA, because we split the image in sub image which are processed by a group of threads. By modifying the size of the sub images we can get $O(1)$, $O(n)$ or $O(n^2)$ computational complexity.

Regarding memory usage, our proposed algorithm, the PCMF, needs $(b-a+n-1)N = O(n)$ additional memory. This is costly for parallel architectures with limited memory, such CUDA-enabled graphics cards, and it can make prevent the conditions $\frac{O(n^2)}{M} \to 0$ or $\frac{O(n)}{N} \to 0$ to be held.

4.2 Parallel Capabilities and Memory Access of *ccdf* and *pmf*

In this subsection we present a theoretical performance comparison between the $ccdf$ based median filter and the histogram based median filters. We will focus on the parallel capabilities and memory access patterns of both methods. We will consider a parallel computing model with many concurrent threads available, concurrent read (for aligned data or *SIMD* access pattern) and penalized exclusive read and writes access (for random memory access).

In that context, calculating and updating $\boldsymbol{\tau}_{\mathbf{x}}$ needs the same number of steps as calculating and updating the histogram (see Eq. 1), although the key difference is the memory access (read) pattern. For the histogram based case, a random access is used, while for the $ccdf$ based case an ordered access pattern is used. Moreover, for the $ccdf$ based case we can use $(b-a)$ threads per $\boldsymbol{\tau}_{\mathbf{x}}$, and we can read the data in a concurrent fashion, whereas for the histogram based case we are restricted to only one thread per histogram and all memory access is penalized.

While the previous point give us a hint about the superior parallel capabilities of the $ccdf$, the main difference is the actual computation of the median value. For the histogram based case, we need to iterate over the histogram

vector, accumulating its contents until the median condition is met; the median value is the index of that element. This procedure is executed only by one thread, and the number of steps required for the search depends on the dataset, giving us unbalanced loads on the threads and data dependency. For massive multithread architectures the execution of only one thread implies a performance penalty, and this penalty is shown in Section 5.2.

For the $ccdf$, we compare each element of the $\boldsymbol{\tau}_{\mathbf{x}}$ vector with the median index. This operation can be done in parallel, thus we need only one step to compare all the data. Finally, to extract the median value we calculate a summation of the result of the previous comparisons. In the many concurrent threads context, this summation can be done in $\log_2(b-a)$ steps (assuming $(b-a)$ is a power of two). The whole procedure to extract the median value from the $\boldsymbol{\tau}_{\mathbf{x}}$ is independent of the data set. From this analysis we can state that the $ccdf$ based median filter is better suited for parallel computing systems than the histogram based ones.

As a simple example, to stress the previous statements, consider the vector $\mathbf{x} = [6, 6, 1, 2, 7]$, with $0 \leq x_i \leq 7$, its histogram $\mathbf{h}$, with eight elements, can be computed by one thread in five steps. Each element of the vector $\boldsymbol{\tau} = \tau_j = ccdf(\mathbf{x})$, $j \in [0, 7]$ can be computed by one thread (8 threads in total) in 5 steps. To extract the median value from $\mathbf{h}$ only one thread can be used, and it requires six steps (each of them include conditional branches and additions). For the $ccdf$ we can use eight threads to apply the indicator function to $\boldsymbol{\tau}$ in one step, and then we can use a binary reduction of three levels to solve the summation. The total steps required to get the median value of $\mathbf{x}$ is eleven for the histogram based algorithm, whilst we need only nine steps with the $ccdf$ based one.

5 Performance

The following tests were executed on two different Intel processors: a first-generation Core i7-930 CPU (2.8GHz, 8MB Cache memory, SEE4.2 Instruction set) and a second-generation Core i7-2600K CPU (3.4GHz, 8MB Cache memory, AVX Instruction set). The first-generations Core processor features 128-bit SIMD register, and the second-generation has a 256-bit SIMD registers. Even though the newer Core processor have larger SIMD register, twice as large, the performance's improvement was less than two (see Fig. 5c and d). This is because the integer operations on the Intel SIMD units are still mapped to 128-bit SIMD registers. In both cases the operative system was Kubuntu 10.4, with Linux kernel 2.6.28 x86_64.

The CUDA implementations were tested on a nVidia Tesla C2075 and on a nVidia GeForce GTX480 graphics card. Both cards have 2.0 computational capability and 48KB shared memory, but the Tesla C2075 has 5GB of global memory, 448 CUDA cores, 1.15GHz GPU clock rate and 1.6 GHz Memory clock rate, while the GeForce GTX480 has 1.5GB of global memory, 480 CUDA cores, 1.4GHz GPU clock rate and 1.8GHz Memory clock rate. Both cards were connected to the host system through a PCI Express 2.0 bus.

Each condition tested (kernel size, input image, algorithm used) is repeated 1000 times and the result shown is the median value. The time measures for the CUDA implementations include memory transfer operations. It is important to note that, for a kernel size k, the elements to be sorted are $n = k^2$. For the tests we use $k = \{3, 5, 7, 9, 11, 13, 15\}$.

5.1 Description of Median Filter Implementations

We test nine implementations of median filters, three for Intel processor and six for CUDA graphics cards. For the Intel platform we have two reference algorithms and for CUDA we have three, the others are PCMF implementations.

For the Intel processor, the first reference implementation is the Constant Time Median Filter (CTMF) algorithm [17]. It is a histogram based median filter with $O(1)$ computational complexity. The second reference implementation is provided by the MatLab function `medfilt2` [16]. We use MatLab version 7.9.0.529 (R2009b) 64-bit (glnxa64). The algorithm is based on histograms and has $O(n)$ computational complexity.

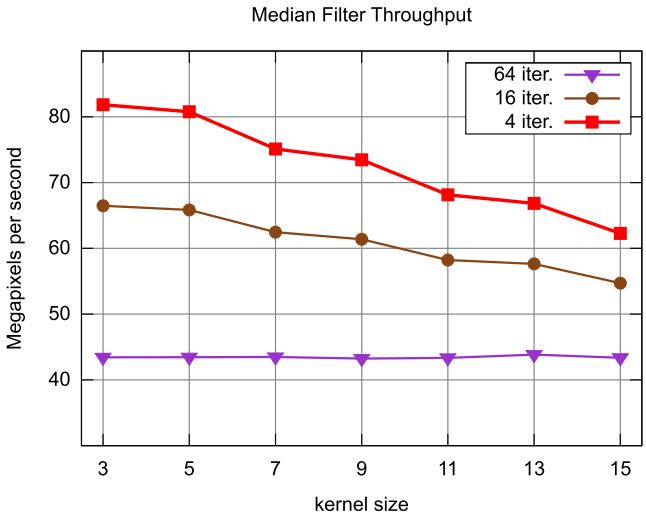


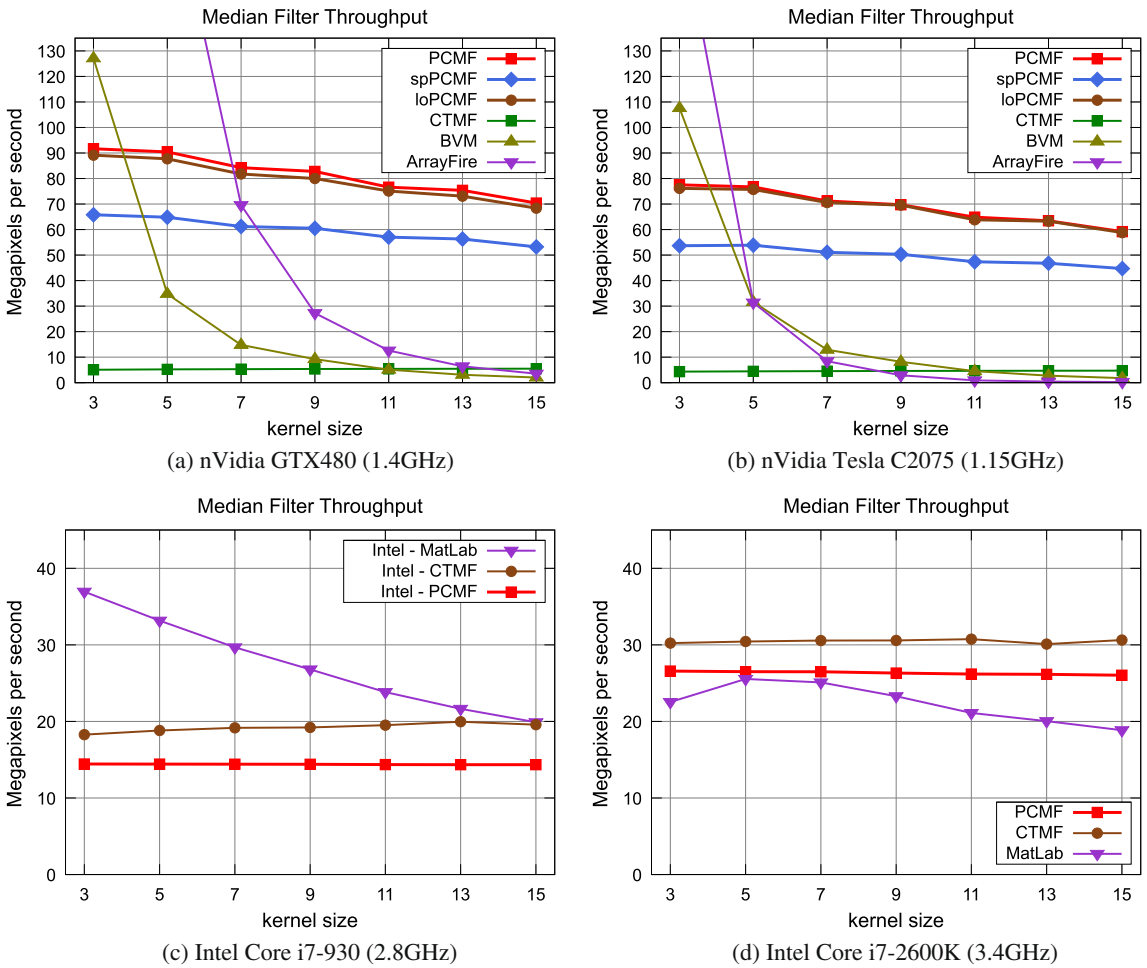
Figure 4 CUDA-PCMF implementations iterating over different number of pixels on the nVidia Tesla C2075.

For CUDA, the first reference implementation is the Branchless Vector Median (BVM) algorithm [5] and the second reference implementation is a closed-source commercial library for CUDA, named libJacket and its new version, named ArrayFire (libaf) [1]. Both algorithms have $O(n^2)$ computational complexity. The third one is our implementation of the CTMF algorithm.

In the Intel implementation of the PCMF, the SIMD unit is used for almost all the operations, which includes the generation of all $\boldsymbol{\tau}$ vectors and calculating the median value. This implementation is expected to have a $O(1)$ complexity, as the condition $\frac{O(n^2)}{M} \rightarrow 0$ is held. The Pthreads framework was used in order to obtain a multithread implementation for the Intel Multicore architecture. For the CUDA implementations, the image is split in sub images and each sub images is processed by one CUDA block with the maximum threads per block allowed by the graphics card. The structure of the algorithm allow us to keep all threads working most of the time, and help us to avoid the bank conflicts.

5.2 Computational Results

In the first test we explore the relation computational complexity and computational performance. To achieve $O(1)$ complexity we need to iterate over each row in order to decrease the effect of the initialization of $\boldsymbol{\tau}_{acc}$ in the performance. For the SIMD single core and Multicore implementations the constant complexity can be achieved easily, but for the massive parallel architecture implemented on the CUDA-enabled graphics cards there is a trade-off between performance and computational complexity. In CUDA, iterations should be avoided whenever is possible and replaced by parallel operations. To test this behavior we implement three versions of the PCMF for CUDA. All the



(a) nVidia GTX480 (1.4GHz)

(b) nVidia Tesla C2075 (1.15GHz)

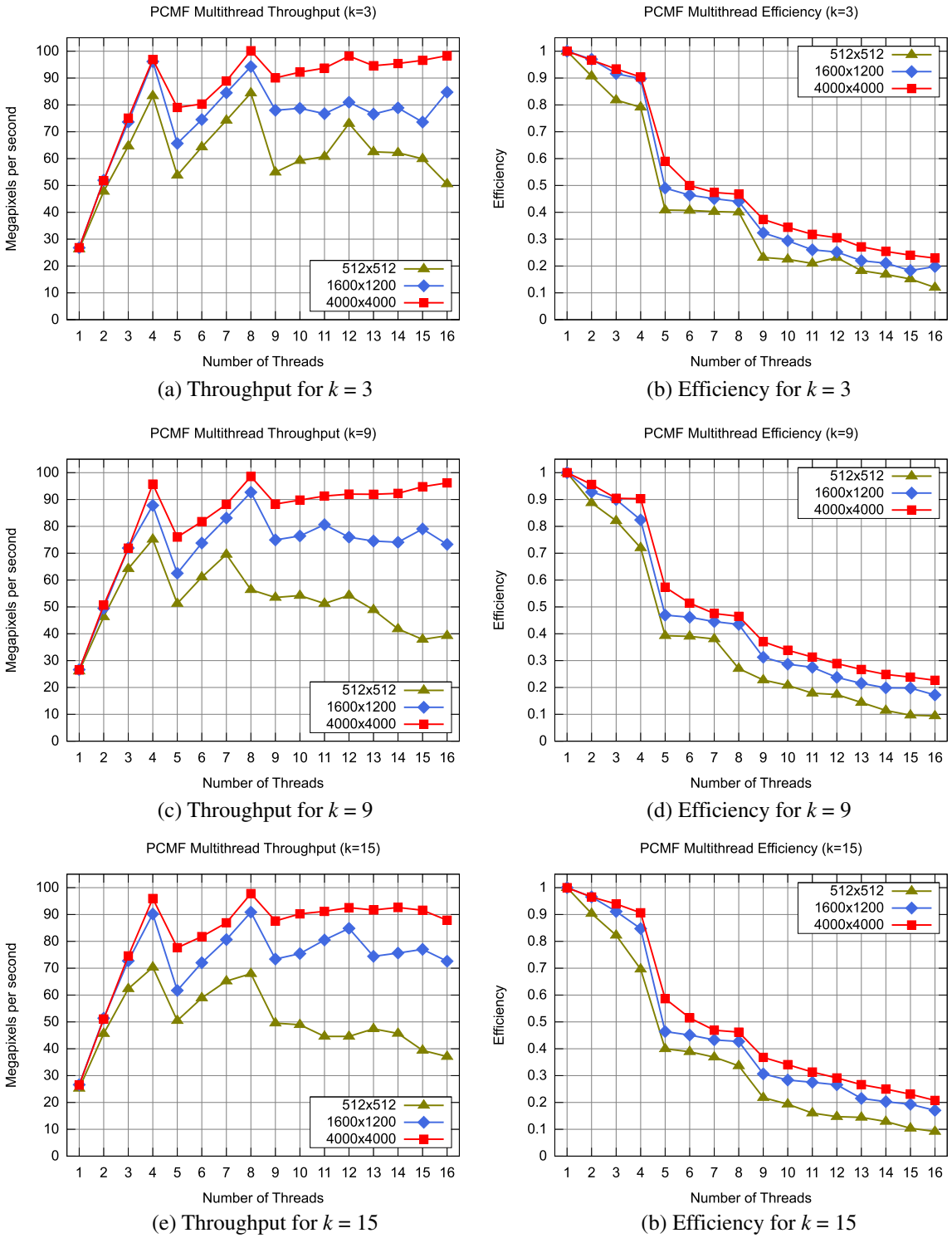
(c) Intel Core i7-930 (2.8GHz)

(d) Intel Core i7-2600K (3.4GHz)

Figure 5 Computational results in megapixels per second.

implementations were tested on the nVidia Tesla C7025 and each block process a sub image of 64 × 16 pixels. The first implementation iterates over the 64 pixels of the sub image and 64 threads were used. The second implementation uses 128 threads and the iterations is over 16 pixels only. The third implementation iterates over only 4 pixels and it uses 1024 threads. The test image used was a 512×512 grayscale image.

From the results (Fig. 4) we can see that we can achieve the constant time computational complexity by iterate over



(a) Throughput for $k = 3$

(b) Efficiency for $k = 3$

(c) Throughput for $k = 9$

(d) Efficiency for $k = 9$

(e) Throughput for $k = 15$

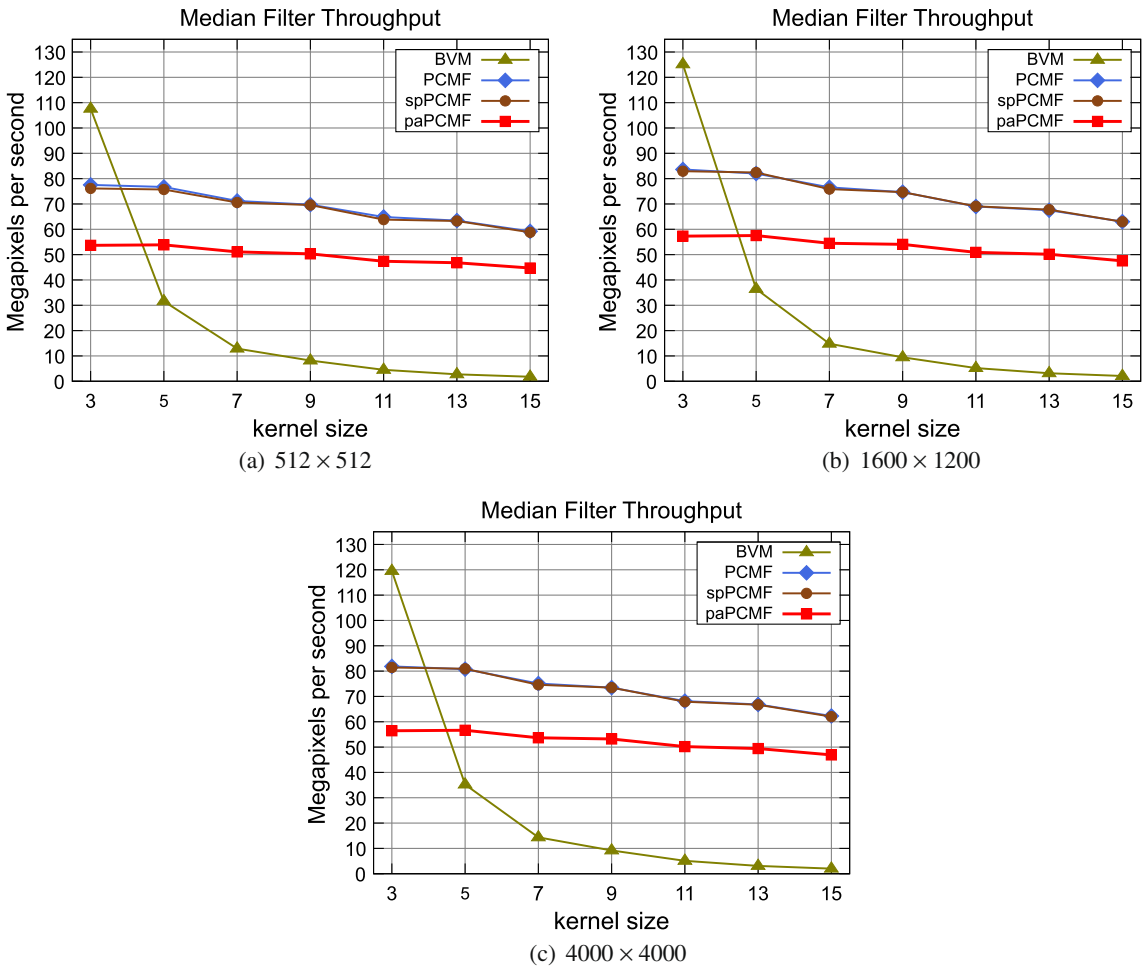
(b) Efficiency for $k = 15$

Figure 6 Performance of the Multicore-SIMD PCMF implementation for different number of threads used on a Intel Core i7 processor.

all the 64 pixels of the sub image. But, if we iterate over 4 pixels only, we can achieve a better performance with $O(n)$ computational complexity. For the following tests we use the linear complexity CUDA implementation.

In the second test we measure the throughput of the median filter implementations. The size of the test image used is 512 × 512 pixels. For the nVidia GTX480 we use all the algorithms described, but for the nVidia Tesla C2075 we only use the BVM algorithms and our implementations of the PCMF. Additionally to the PCMF, we test the performance of the PCMF algorithm in which the Salt & Pepper noise model (see Section 3.3) is considered in a naive fashion: if the current pixel (of the analyzed neighborhood or set) is either 0 or 255 then it is replaced by the median value; we label this test as "loPCMF" (Algorithm 2). We also test the performance of the PCMF algorithm, where we only considered one iteration of the adaptive median filter [12] as described in Section 3.3; we label this test as "spPCMF" (Algorithm 3).

In Fig. 5 we compare the performance of the implementations for the two Intel Processors and the two nVidia cards. We can see that the performance is better in the second-generation Core processor than in the first-generation. The improvement is mainly because of the new form to operate the registers, available in the AVX instruction set, rather than in the increase in the length of the SIMD registers, as only the floating point operations were escalated to use this longer registers. It is also important to highlight the results presented in Fig. 5a and b where all three versions of the PCMF algorithm are compared: there is almost no computational performance difference between the implementations of Algorithms 1 ("PCMF") and 2 ("loCTMF"), nevertheless for Algorithm 3 ("spCTMF") there is a trade-off between computational performance and reconstruction quality: the computational performance of spPCMF is reduced (between 20 and 30 % depending of the kernel's size) when compared with "PCMF" and/or "loPCMF", although (see Section 3.3) the "spCTMF" implementation gives far superior reconstruction results (than "PCMF" and/or "loPCMF"), being of more than 10 dB (SNR metric) for moderate to high levels (40–70 %) of Salt & Pepper noise. We must also note that our CUDA implementation of the CTMF algorithm [17] has



(a) 512 × 512 (b) 1600 × 1200

(c) 4000 × 4000

Figure 7 Comparison of CUDA implementations for various image sizes on the nVidia Tesla C2075.

a throughput of almost 6.3 megapixels per second, and is almost 7 times slower than the $O(1)$ PCMF (see Fig. 4).

For the third test we use the Intel's SIMD Multicore implementation of the PCMF algorithm. The processor used was a Second Generation Intel Core i7 with 4 physical cores and 8 logical cores. We used three images with different sizes and, and we tested all the kernel sizes mentioned before, but we show only three cases (Fig. 6).

In Fig. 7 we can see a peak of performance of 100.1 megapixels per second, and the Efficiency of the implementation is greater than 0.9 when we use up to 4 threads. The Efficiency drops as we increase the number of threads, and this is because we only have 4 physical cores, where we have the first peak of performance. The second peak of performance is obtained when 8 threads is used, and this correspond to the number of logical cores of the processor.

For the final test we use three images of different size to see the scalability of the implementation on CUDA graphics cards. For this tests we use the BVM implementation and the three versions of the PCMF.

A typical behavior in CUDA implementations is that the performance improves as the amount of data to process increases. This is because there is available a large amount of processors to be used to process the data, and this behavior can be seen in Fig. 7. We can see an increase in the performance when we process the 1600×1200 image in comparison to the 512×512 image. The amount of processors in a nVidia graphics card is large but limited, so the improvement of the performance by increasing the input data is limited too, as we can see when we process a 4000×4000 image.

6 Conclusions

In this paper we expand our previous work on the Parallel Ccdf-based Median Filter (PCMF) [22]. The structure of our algorithm allows us to get an efficient and simple implementation for modern parallel programming models. The computational results show that the CUDA implementation of the proposed median filter algorithm is efficient and can outperform other generic median filters for CUDA.

The behavior of the implementation Intel PCMF is clearly $O(1)$. The reference algorithm, the CTMF, also has $O(1)$ computational complexity and has a better performance than our implementation. The MatLab median filter has linear complexity and it has, for larger n, worse performance than the other median filter tested. The multithread implementation of the PCMF algorithm achieved a peak performance of 100.1 megapixels per second when using 8 threads, with 4 threads the performance is 96.9 megapixels per second. The efficiency of the multithread implementation of the PCMF for the Intel CPU is greater than 0.9 for 4 threads, which correspond to the number of physical cores of the processor.

The CUDA implementation of the PCMF has the best overall performance of the tested algorithms, and has a lower computational complexity than the reference CUDA implementations. $O(n^2)$ algorithms are usually faster than $O(n)$ and $O(1)$ for small n, and this is the case for $n = 3$ and $n = 5$. However, overall, our implementations outperform the other reference algorithms in the current literature. Additionally, we implemented the CTMF for CUDA and its performance is lower than the PCMF implementation and lower than its Intel implementation, which means that this algorithm, the CTMF, is not well suited for parallel computer architectures.

Finally, we use Salt & Pepper noise models to improve the reconstruction quality of the filter (see Section 3.3). The two proposed algorithms, the "loPCMF" and the "spPCMF" improve the image quality with a trade-off in performance. In the case of the "loPCMF" there is almost no penalty in using the simple noise model, but in the case of the "spPCMF" we have a greater penalty (between 20 and 30 %) but the noise model used allows for a better reconstruction.

Acknowledgement The authors would like to thank NVIDIA for hardware support through its CUDA Teaching Center Program.

References

1. AccelerEyes (2012). ArrayFire Library. http://www.accelereyes.com/products/arrayfire. Accessed 27 Sept 2012.
2. Bovik, A. (2000). *Handbook of image and video processing*. New York: Academic Press.
3. Chaudhuri, B. (1990). An efficient algorithm for running Window pel gray level ranking 2-D images. *Pattern Recognition Letters*, *11*(2), 77–80.
4. Chen, S., Qin, J., Xie, Y., Zhao, J., Heng, P. (2009). A fast an flexible sorting algorithm with CUDA. In *9th international conference on algorithms and architectures for parallel processing* (pp. 281–290).
5. Chen, W., Beister, M., Kyriakou, Y., Kachelries, M. (2009). High performance median filtering using commodity graphics hardware. In *IEEE nuclear science symposium conference record (NSS/MIC)* (pp. 4142–4147).
6. Cline, D., White, K.B., Egbert, P.K. (2007). Fast 8-bit median filtering based con separability. In *International conference on image processing (ICIP)* (pp. 281–284).
7. Cockshott, P., & Renfrew, K. (2010). *SIMD programming manual for Linux and Windows*. New York: Springer.
8. Farber, R. (2011). *CUDA application design and development*. San Mateo: Morgan Kaufmann.

9. Furtak, T., Amaral, J.N., Niewiadomsk, R. (2007). Using SIMD register and instructions to enable instruction-level parallelism in sorting algorithms. In *SPAA '07: Proceedings of the nineteenth annual ACM symposium on parallel algorithms and architectures* (pp. 348–357).
10. Gil, J. (1993). Computing 2-D min, median and max filters. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, *15*(5), 504–507.
11. Huang, T., Yang, G., Tang, G. (1979). A fast two-dimensional median filter algorithm. *IEEE Transactions on Acoustics, Speech and Signal Processing*, *27*(2), 13–18.
12. Hwang, H., & Haddad, R.A. (1995). Adaptive median filters: new algorithms and results. *IEEE Transactions on Image Processing*, *4*(4), 499–502.
13. Kirk, D.B., & Hwu, W.m.W. (2010). *Programming massively parallel processors: A hands-on approach*. San Mateo: Morgan Kaufmann.
14. Kolte, P., Smith, R., Su, W. (1999). A fast median filter using AltiVec. In *International conference on computer design (ICCD'99)* (pp. 384–391).
15. Li, Y., Peng, S., Chu, W. (2009). An efficient parallel sorting algorithm on Metacube multiprocessors. In *9th international conference on algorithms and architectures for parallel processing* (pp. 372–383).
16. MatLab (2011). medfilt2. http://www.mathworks.com/help/toolbox/images/ref/medfilt2.html. Accessed 14 Oct 2011.
17. Perreault, S., & Hébert, P. (2007). Median filter in constant time. *IEEE Transactions on Image Processing*, *16*(9), 2389–2394.
18. Rauber, T., & Rünger, G. (2010). *Parallel programming: For multicore and cluster systems* (1st ed.). New York: Springer.
19. Ryan, T. (2007). *Modern engineering statistics* (Chapter 14, p. 468). New York: Wiley-Interscience.
20. Sánchez, R. (2011). *Diseño e implementación del filtro mediano de dos dimensiones para arquitecturas* SIMD. Bacherlor's degree thesis, Pontifical Catholic University of Peru (PUCP), Lima, Peru.
21. Sánchez, R. (2012). *Constant complexity bidimensional median filter for parallel computing architectures*. Master's thesis, Pontifical Catholic University of Peru (PUCP), Lima, Peru.
22. Sánchez, R., & Rodriguez, P. (2012). Bidimensional median filter for parallel computing architectures. In *37th international conference on acoustics, speech, and signal processing (ICASSP)* (pp. 1549–1552).
23. Tukey, J. (1974). Nonlinear (nonsuperimposable) methods for smoothing data. In *IEEE Electronics and Aerospace Conference (EASCON), conference records* (p. 673).
24. Wang, Z., Bovik, A., Sheikh, H., Simoncelli, E. (2004). Perceptual image quality assessment: from error visibility to structural similarity. *IEEE Transactions on Image Processing*, *13*(4), 600–612.

Ricardo M. Sánchez received the B.Sc. degree in electrical engineering from the Pontificia Universidad Católica del Perú, Lima, Peru, in 2010, and the M.Sc. degree in electrical engineering from the same university in 2012.

He is currently a Professor in the Department of Electrical Engineering at the Pontificia Universidad Católica del Perú. His research interests include SIMD and SIMT algorithms, inverse problems in signal and image processing, immersive and virtual reality systems.

Paul A. Rodríguez received the B.Sc. degree in electrical engineering from the Pontificia Universidad Católica del Perú, Lima, Peru, in 1997, and the M.Sc. and Ph.D. degrees in electrical engineering from the University of New Mexico in 2003 and 2005, respectively.

He spent two years as a postdoctoral researcher at Los Alamos National Laboratory, Los Alamos, NM, and is currently an Associate Professor with the Department of Electrical Engineering at the Pontificia Universidad Católica del Perú. His research interests include AM-FM models, SIMD and SIMT algorithms, adaptive signal decompositions, and inverse problems in signal and image processing.