Journal of Supercomputing manuscript No.
(will be inserted by the editor)

How separable median filters can get better results than full 2D versions.

Theoretical approach, experimental study and GPU-optimized implementation.

Gilles Perrot* · **Stéphane Domas** ·
Raphaël Couturier

Received: date / Revised: date

Keywords median, filter, separable, GPU

Abstract The well-known method of median filtering is used both in a wide range of application frameworks and as a standalone filter. Small-window median filters can highly reduce the power of salt and pepper or additive Gaussian noise and minimize edge blurring. Large-window filters are also used, for example to estimate the background of images. Currently, the window size should not be an issue as a constant time algorithm and several implementations, including GPU[1]-based codes, have been proposed in recent years. Unfortunately, none of these constant time implementations manage to fully exploit the capabilities of modern GPUs and thus the throughputs of large-window median filters remain far below the peak throughputs allowed by recent GPU models.

This paper aims at showing that a separable approximation of a 2D median filtering is often stronger than its full 2D implementation. Statistical and theoretical analysis are conducted to show and explain this fact which had so far remained unobserved. It is confirmed by experimentations on a dataset composed of 10,000 images, corrupted by different levels of salt and pepper noise. Separable and full 2D median filter algorithms are compared with several metrics, notably PSNR and MSSIM. In addition, a GPU implementation of 2D separable-median filters is also proposed. This implementation is able to output up to 125 billion pixels per second on a recent Volta V100, which significantly outperforms existing implementations like Nvidia's NPP[2] library or Green's code, resulting in the fastest median filtering solution to date.

FEMTO-ST institute, Univ. Bourgogne Franche-Comté, CNRS, 19, av. Maréchal Juin, 90000 Belfort, France.
E-mail: gilles.perrot@univ-fcomte.fr

[1] GPU: Graphical Processing Unit
[2] NPP: Nvidia Performance Primitives

1 Introduction

Median filtering is a mathematically simple technique, first introduced by Tukey in [7]. Applying a median filter to an input image consists in replacing each pixel value by the median value of its neighbors. Most of the time, the neighbors belong to a $n = k \times k$ window around the working pixel. Median filtering has been widely studied since its introduction and several median selection methods have been proposed. These methods can be classified into two main classes: histogram-based and sorting-based algorithms.
Histogram-based solutions work by building the histogram of the gray levels of all the pixels inside the filtering window. Then, starting from either the top or the bottom class, the number of pixels in each gray level class are summed up until half the total pixel count of the windows is reached. The current class holds the median value.
Sorting-based solutions simply sort the gray level values of the filtering window and select the one located at the middle index in the sorted vector.

Originally, the computational cost, the data-dependent execution time and most of all the computational complexity were seen as drawbacks. Researchers have since addressed these issues and designed many implementations, among which efficient histogram-based median filters featuring predictable runtimes [4, 8] and even a constant-time algorithm proposed in [9], which exploits histogram properties to reduce the computational complexity to a single suppress and add operation for each consecutive window position. More recently, authors have managed to take advantage of the newly opened perspectives offered by modern GPUs to develop CUDA[3]-based filters such as the Branchless Vectorized Median filter (BVM) [3, 5] featuring a significant runtime improvement, along with the histogram-based PCMF[4] median filter [6]. Lately, in a previous work, the sorting-based PRMF[5] implementation [10] has been proposed and remains the fastest method to date for 3×3 and 5×5 windows. Since then, other solutions have been proposed, that allow higher speed for larger kernels, as it is the case for recent works proposed in [21] or [22], adapted from the original [9]. PRMF uses simplified sorting networks to iteratively eliminate min and max values. Recently in [2], the author worked on the separability of sorting networks and obtained significant speedups, reaching almost the highest throughput of PRMF for 3×3 and 5×5 filters with better scalability for larger sizes.

Median filtering is mainly efficient in reducing noises whose Probability Density Functions (PDFs) show high standard deviation values, which amounts to saying that it is a robust estimator of the average value. It is this very property of robustness that makes median filtering so interesting in signal processing, as it can ignore extreme or aberrant values present in the signal and select the most representative value inside the filtering window. Two-

[3] Compute Unified Device Architecture, Nvidia's programming model.
[4] Parallel Ccdf-based Median Filter (Ccdf: Cumulative Complementary Distribution Function).
[5] Parallel Register-only Median Filter.

dimensional median filters are derived from a non-linear and non-separable mathematical operator but, as shown by Tukey in [12], the *median of medians* [6] obtained by two computationally simpler stages of median filtering remains as robust an estimator of the mean value as the exact median and can replace it advantageously, mostly for runtime reasons.

Some authors have already studied the smoothing performances of both full and separable median filters, as in [13] where coarse approximations are given for the output variances under both uniform and Gaussian noise PDFs assumptions.

In the field of noise reduction in digital images, Tukey's observations can be adapted with small datasets built from the pixels of the sliding window. The first computational stage would then provide one median value for each of the k rows of the window; the second stage would issue the median value of those k median values. However, the median of median has not been widely adopted as a noise reduction technique despite the increase in image sizes and the need for larger filtering windows. The reason is probably its higher output variance value. Instead, researchers have mainly kept trying to compute the full 2D median filter at high speed rates on larger filtering windows.

On GPU architecture, some histogram-based algorithms are quite efficiently implemented and show almost constant speed with respect to the filtering window size, but the memory size required by the histogram forbids to achieve near to peak throughput values. Sorting-based algorithms proved highly efficient for small window sizes, but their complexity does not allow an extended scalability. The fastest implementations of both categories [21] and [10] define a threshold around 13×13, where the histogram-based implementation becomes faster than the sorting-based one.

The contribution of this paper is summarized in this paragraph. First, it can be observed that a separable median filter is most of the time stronger than a full median filter. Second, the output variance difference between both versions of the median filter decreases very abruptly when the window size increases. Third, a study is proposed to show that the noise types which are best reduced by median filters have spiky looking PDFs, such as salt and pepper noise. However, this does not lessen the interest of median filtering for Gaussian noise reduction. Finally, a very efficient implementation on GPU is described and throughputs are given on a Volta V100.

In the following, the median of medians two-stage process is referred to as the "separable" median filter for simplicity and clarity's sake, as opposed to the "full" median filter. In addition, it will be shown that the separable filter is almost always a better choice. For this purpose, section 2 first details the comparison between full and separable median filters. Then, section 3 presents a reminder of the median value selection method used in the PRMF

[6] In that chapter, Tukey adresses the issue of statistical analysis on large datasets and discusses a method in which data are sliced into nine-value sets (or 81 values for larger datasets) and where the mean estimation is taken as the median value of the median values of all those sets, *i.e.* the ninther. The process can be recursive. In our implementation, data slices are mapped to image rows.

and how is has been efficiently implemented on GPU within the proposed 2D separable median filter. Section 4 then presents the two experimental setups used to evaluate the performances of the proposed implementation and to conduct the statistical comparison of both median filter versions. Finally, the execution time measurements and numerical properties of the compared filters are exposed in section 5 before the conclusion drawn in section 6.

2 Theoretical comparison of separable and full median filter under salt and pepper noise

This section proposes a theoretical study that demonstrates the conditions under which the separable filter is stronger than the full version on bi-chromatic images. It exposes a threshold for window size above which a separable filter corrupts less pixels than the separable version, while the number of denoised pixels tends to be similar.

Salt and pepper noise is caused by bad pixels in image sensors or by transmission errors. Its name refers to the visual aspect of the resulting corruption: black and white pixels scattered over the image.

The power of salt and pepper noise is represented by the probability P of each pixel to be corrupted. For bpp 8 gray level images, the PDF can be characterized by the expressions below where $p(v \mid u)$ is the probability to observe the gray-level value v knowing the ground-truth value u. These expressions are derived from the classical equations of the model, whose description can be found, for example in [23].

$$p(v \mid u) = \begin{cases} \frac{P}{2} + (1-P) & \text{if } v = 0 \text{ and } u = 0 \\ \frac{P}{2} + (1-P) & \text{if } v = 255 \text{ and } u = 255 \\ \frac{P}{2} & \text{if } v = 0 \text{ and } u \neq 0 \\ \frac{P}{2} & \text{if } v = 255 \text{ and } u \neq 255 \\ (1-P) & \text{if } v = u \text{ and } u \notin \{0, 255\} \\ 0 & \textit{elsewhere } \forall (v,u) \in [0;255]^2 \end{cases} \tag{1}$$

To establish which filter provides the strongest denoising, let us assume that the original image is *noiseless* (none of its pixels is white or black) and that it is corrupted by a salt and pepper noise of probability P, which means that all black or white pixels are noise. Such a case is easily simulated by scaling the gray levels and does not cause too much distortion, especially when dealing with bpp 12 or higher-depth images.

2.1 Noisy pixels removed by full 2D median filter

If *glmax* is the highest gray-level value allowed by the pixel format, the probability for one pixel x of the input image to be corrupted is $P(x = 0 \textit{ or } x =$

$glmax) = \lambda$. The full median filtering window is $k \times k$ pixels wide and contains a total amount of $n = k^2$ pixels. At each position of the filtering window, if one of the extrema values (black or white) occurs at least $\lceil k^2/2+1 \rceil$ times, the output pixel y will be of that same extremum gray level, and will still remain a noisy pixel. The probability of the output pixel y being left noisy, denoted $P_f(yn)$ is then given by the following equation:

$$P_f(yn) = 2 \sum_{q=\lceil k^2/2+1 \rceil}^{n} C_n^q \left(\frac{\lambda}{2}\right)^q \left(1 - \frac{\lambda}{2}\right)^{n-q} \quad (2)$$

2.2 Noisy pixels removed by separable median filtering

As for the separable filter, similar considerations still apply, but two 1D filters, $1 \times k$, then $k \times 1$ are processing each image consecutively.

Here, the threshold is $\lceil \frac{k}{2} + 1 \rceil$ and the probability $P_{sh}(x_h n)$ of x_h being a noisy pixel after the first pass (horizontal by assumption) and before the second pass of 1D median filtering is given by the following expression, analogous to equation 2

$$P_{sh}(x_h n) = 2 \sum_{q=\lceil \frac{k}{2} \rceil}^{k} C_k^q \left(\frac{\lambda}{2}\right)^q \left(1 - \frac{\lambda}{2}\right)^{k-q}$$

and the probability of y being noisy is then

$$P_{sv}(yn) = 2 \sum_{q=\lceil \frac{k}{2} \rceil}^{k} C_k^q \left(\frac{P_{sh}(x_h n)}{2}\right)^q \left(1 - \frac{P_{sh}(x_h n)}{2}\right)^{k-q} \quad (3)$$

Fig. 1 displays the difference $P_{sv} - P_f$ when the noise level ranges from 7% to 50% and filtering window sizes from 3×3 to 11×11. For larger filtering windows, the difference becomes insignificant compared to the other parameters that will be discussed later on in this paper. For example at a 50% level, the probability of both 11×11 median filters to output a noisy pixel at a given position differs by less than 2.6×10^{-4} point. It becomes obvious that, the number of noisy pixels left by both filters rapidly tends towards extremely close values.

However, median filters do not just alter noisy pixels, but are likely to alter other pixels in the input image, thus generating unwanted corruption (noise).

Actually, computing the number of pixels which would be corrupted through this mechanism, inside a natural image, is far beyond standard computing capabilities. Nevertheless, in order to begin to understand the distortion brought by each of the full and separable median filters, let us make the strong hypothesis of a bi-chromatic image composed of a background color c_b and some objects or shapes of color c_a. Three illustrative examples of such images are provided at Fig. 2.

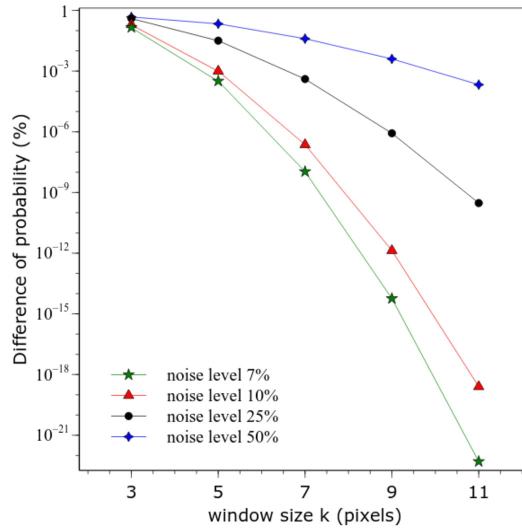


Fig. 1 Difference between the probabilities of a pixel being left noisy by full and separable median filters for filtering windows ranging from 3×3 to 11×11 in presence of salt and pepper noise of power 7% to 50%.

2.3 Pixels corrupted by full median filters

One pixel is altered by $k \times k$ full median filtering if there are at least $\lceil \frac{k^2}{2} \rceil$ pixels of the other color inside the filtering window (and at most $n-1$).

Consequently, a central pixel of color c_a is altered if at least $\lceil \frac{k^2}{2} \rceil$ pixels are of color c_b inside the $k \times k$ filtering window. This color balance is provided by a number s_a of possible combinations, given by

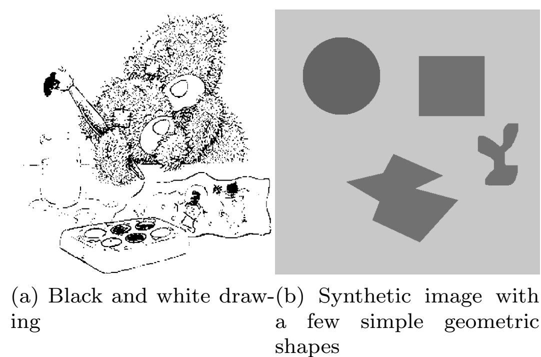
$$s_a = \sum_{q=\lceil \frac{k^2}{2} \rceil}^{n-1} C_{n-1}^{q}$$

The combination count s_b that leads to altering a central pixel of color c_b follows the same expression and the sum $s_{full} = s_a + s_b$ represents the total number of color combinations where the central pixel will be altered by median filtering. The following equation describes s_{full}.

$$s_{full} = 2^{n-1} - C_{n-1}^{r-2} \tag{4}$$

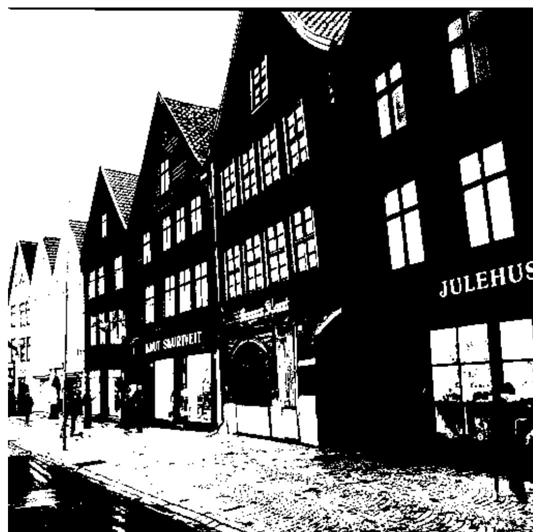
2.4 Pixels corrupted by separable median filtering

A similar path leads to the count of combinations s_{sep} in the separable case. Assuming that the first 1D filtering is achieved through a $1 \times k$ horizontal window, and that the color of the central pixel P is c_a, this pixel will be altered if at least $\lceil \frac{k}{2} \rceil$ pixels are c_b-colored inside the $k \times 1$ vertical window of the second



(a) Black and white drawing

(b) Synthetic image with a few simple geometric shapes



(c) Natural image converted to B&W

Fig. 2 Examples of 512×512 bi-chromatic images.

filtering step. The pixel's colors inside this vertical window are determined by the first filtering step and thus, each c_b-colored pixel is the output of one $1 \times k$ median applied on one row where at least t pixels are c_b-colored. Note that combination counts regarding the middle row have to be adjusted as it contains the central pixel, whose color is known by assumption(c_a).

The same applies to c_b-colored central pixels and the total combination count s_{sep} is obtained by adding up both subtotals. That leads to the expression of equation 5, which obviously applies whatever the direction of the first filtering step.

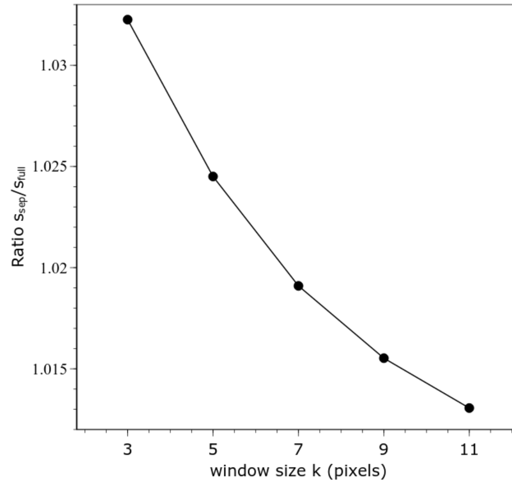


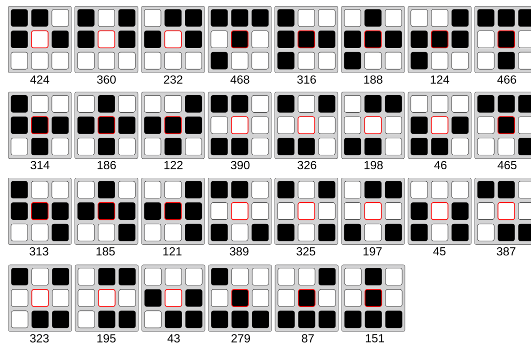
Fig. 3 Ratio between the combination counts of pixels altered by separable and full median filters, for windows ranging from 3×3 to 11×11

$$
\begin{aligned}
s_{sep} &= \sum_{l=\lceil k/2 \rceil}^{k} Q_k^{l-1} C_k^l \frac{(2^k - Q_k)^{k-l-1}}{k} \\
&\quad \times \left(R_k.l(2^k - Q_k) + Q_k(k-l)(2^{k-1} - R_k) \right) \\
\textit{where} & \\
Q_k &= \sum_{q=\lceil k/2 \rceil}^{k} C_k^q \\
R_k &= \sum_{q=\lceil k/2 \rceil}^{k-1} C_k^q
\end{aligned}
\tag{5}
$$

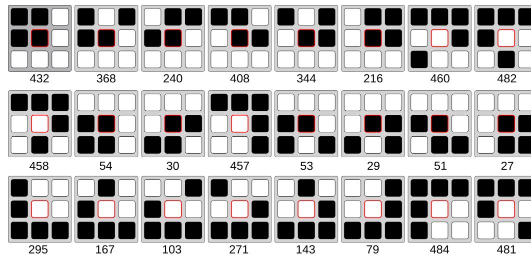
Fig. 3 displays the ratio s_{sep}/s_{full} for filtering windows under 13×13 and shows that both counts are very close and are asymptotically tending to identical values when k exceeds 11.

Judging from the above comparisons, it seems clear that in presence of salt and pepper noise, full median filtering removes a higher number of noisy pixels and also corrupts pixels in fewer situations than separable filtering.

However, a more in-depth analysis reveals that, inside natural images, not all color combinations resulting in the corruption of a given pixel have the same probability to occur. Unfortunately, these probabilities cannot be formally expressed as they depend on each image content. The combinatorial complexity also forbids a numerical evaluation. For example, the smallest filtering window sizes (3×3 and 5×5) would lead to analyse respectively more than 4×10^{21} and 1.6×10^{60} combinations.



(a) Separable filter alters the output pixel and full filter does not.



(b) Full filter alters output pixel and separable does not

Fig. 4 Combinations, inside a bi-chromatic image, where the central pixel is altered by one and only one version of the 3×3 median filter. Due to the symmetry, colors c_a and c_b can be associated to either the white or black pixels. The number under each combination is the decimal value of the binary representation obtained by concatenating the rows from top to bottom (most significant bits first) and considering that a black pixel is associated to the digit 1.

2.5 When the separable median outputs better images than its full version.

For $k = 3$, the filtering window includes $k^2 = 9$ pixels, each of them colored with one of the two possible gray levels c_a and c_b. Thus, it allows a binary representation of the entire filtering window, with the first row holding the k most significant bits and the last row holding the k least significant ones. Among the $2^{k^2} = 512$ possible binary combinations offered by this representation, separable median will add noise by altering the center pixel (*ie.* changing its color) for 192 of these combinations and only 186 for the full filter. Moreover, there are 30 combinations where the separable filter alters the central pixel and the full filter does not, while the opposite is observed in only 24 situations. Fig. 4 details all these combinations where only one of the filters alters the central pixel.

Nevertheless, the above figures do not take into account the probability of each above combinations to appear in natural images. At a first glance, one can notice that most of the situations of Fig. 4b seem likely to be found

in natural images, as they can represent pieces of simple continuous contours or segments. This is the case for 22 out of the 24 combinations detailed in 4b, only numbers 460 and 481 are not continuous. The situations displayed in Fig. 4a describe more scattered pixels less likely to be found in natural images. Indeed only 6 out of the 30 combinations of Fig. 4a could represent a piece of simple continuous contour or segment (numbers 424, 232, 124, 46, 313 and 43). These observations can lead to the assumption that the full filter alters more than the separable one when processing natural images. This is confirmed by parsing the three images of Fig. 2 and counting the occurrences of each combination. The results are shown in Table 1.

combination counts→ image↓	Only altered by full	Only altered by separable
Teddy bear	3352	93
Synthetic	10	0
Houses	1620	497

Table 1 Total count of pixels in the bi-chromatic image that are altered by only one of the 3×3 median filters (full or separable).

The full filter removes a higher number of noisy pixels than the separable one but the gap decreases very rapidly when k increases. Meanwhile, the full filter seems to corrupt images more than the separable version. These two remarks tend to imply that for any given power of noise, there is a threshold value for k above which separable median filtering is better than full median filtering. This is confirmed by the measurements conducted on grayscale images detailed in sections 4 and 5.

3 Implementing a fast 2D median filter in a separable way

In an earlier work, the 2D full-median filter implementation called PRMF [10] was proposed. Though extremely fast for small filtering windows, it is outperformed for large window sizes by several other implementations like [3], [6], [9] and [21]. However, section 2 demonstrates that the separable median filter is as robust as the full version, and could output less noisy images in a wide range of situations, notably for large window sizes. This is because the computational complexity of the full median is higher than that of separable median, so that the expected runtimes of the latter are significantly faster.

The main issue to achieve good performances is the choice and implementation of the selection method used to identify the median value among a collection of gray levels. A second key issue is how to rule redundancy between consecutive positions of the sliding window, since two neighboring pixels share some of the values to be sorted, as shown in Fig. 5. The separable implementation proposed in this paper uses the same optimizations for the selection process as in PRMF [10]. It consists in a forgetful selection algorithm using

Fig. 5 Overlapping windows (framed in red and blue) of two 5×1 median filters applied on adjacent pixels colored in red and blue. Filtering windows share 4 pixels.

thread registers and exploiting the Instruction Level Parallelism (ILP) capability of the GPU. These principles are exposed in sections 3.1 and 3.2. Compared to the full version, the way of ruling the redundancy has been extended and is discussed in section 3.3. It is also the case for the strategies to manage the global memory used to store input, output and intermediate images. They are presented in section 3.4.

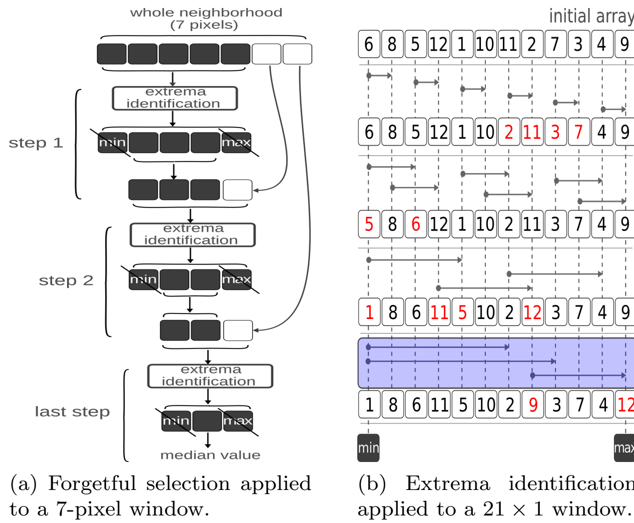
3.1 Using registers

As register access is at least 20 times faster than all the other memory types available on the GPU, it is natural to try to use thread registers as a means to store temporary data inside our kernels, keeping in mind that from Kepler to Volta architectures [15]–[18], each individual thread can use a maximum of 255 registers within the limit of 32K or 64K per thread block, depending on the GPU family. However, a high register usage, even if below the above-mentioned limitations, may result in a loss of performance due to a lower parallelism level inside each block, *i.e.* less threads actually run in parallel. Consequently, registers need to be used as sparingly as possible, in order to preserve high pixel throughput values. To do so, the forgetful selection algorithm already described in [10] has been implemented. Its principle is to build an initial list of R_k pixels values taken among the k pixels inside the 1D window (horizontal or vertical), then to identify and eliminate (forget) both elements showing the maximum and the minimum values in the list. Finally, one of the values left apart of the original list is included in the current list. This process is repeated until no more values can be included in the list. The remaining element in the list is the global median value. It is important to notice that this algorithm has a fixed and known number of steps, equal to $\left(k - \lceil\frac{k}{2}\rceil\right)$, which implies that all threads have almost the same workload, despite the data dependency of the extrema identification step.

The number R_k of elements (and thus registers) in the initial list is chosen as the minimum element count which allows to identify the global median value through the above process. It is obtained by considering the constraint of keeping the global median in the list at each elimination step. This leads to:

$$R_k = \lceil\frac{k}{2}\rceil + 1$$

It is also noticeable that each elimination step uses one register less than the previous one as two elements are eliminated and one element added at each step.



(a) Forgetful selection applied to a 7-pixel window.

(b) Extrema identification applied to a 21 × 1 window.

Fig. 6 (a) Determination of the Median value by the forgetful selection process.(b) First extrema identification step of the forgetful selection. It begins at the top row with unsorted elements and ends with the minimum value at the first position (left) and the maximum at the last position (right). The last pack of instructions, in light purple, may contains interdependent instructions, while the others contain only independent ones. The state of the array is displayed at the end of each pack.

Fig. 6a illustrates the forgetful selection process applied to a 7-pixel 1D median filter. The process begins with $R_7 = 5$ elements and ends after 3 iterations, when there is no more candidate element to add to the current list. This also corresponds to the state where there is only one element left in the list: the median value. The selection of both *extrema* is similar to the first iteration of a bubble sort. It uses a basic 2-element swapping function to push the minimum (resp. the maximum) value to the beginning (resp. the end) of the list. The implementation of this selection process ensures that the GPU kernel code is free of divergent branches liable to severely impact performances.

3.2 Hiding Latencies

Optimizing a GPU kernel also means hiding latencies potentially generated by memory accesses and data dependent instruction calls. Indeed, modern GPUs are able to pipeline instruction processes so as to reduce the average latency of an instruction sequence. This capability is called ILP (Instruction Level Parallelism). As for global memory accesses, when two or more consecutive arithmetic operators manipulate (read or write) independent variables, only the first access generates latency. The massive thread parallelism of CUDA-enabled devices helps in hiding some of those latencies transparently. In our parallel code, an analysis of the actual computation performed by each thread, shows that ILP has to be optimized:

First, the Instruction Level Parallelism can be increased within the forgetful selection method by re-arranging the instruction sequence as it would be done in a sorting network [1], so as to reduce the data dependency of consecutive instructions and thus to prevent frequent empty pipelines. Fig. 6b shows the scheduling of the first extrema identification step of a 21×1 median filter, carried out with $R_{21} = 12$ elements. Each arrow represents one call to the 2-element swapping function: after the call, the starting point symbolizes the lowest value element and the ending arrow points out the highest one. In addition, horizontal dashed lines separate packs of instructions, where the first packs contain instructions with independent variables and the last pack contains the swapping instructions with the largest strides. This last pack may contain some interdependent instructions, depending on the size of the array. After the execution of each pack, the current state of the array is also displayed to help in following the individual moves of each value. In the example of Fig. 6b, 3 packs of independent instructions are defined, and the fourth contains interdependent ones. Considering the whole sequence, the schedule of each instruction is adjusted to maximize the distance (in terms of line of code) between interdependent instructions. It allows to maximize the ILP. In this example, the last pack cannot be optimized, but it is negligible compared to a naive implementation that would not have achieved any ILP at all.

3.3 Exploiting the window overlapping

The goal here is to increase computations inside the GPU threads, as the performances of the median filter are limited by the cost of memory accesses. Increasing the computation ratio must be done without impairing the possible parallelism level, i.e. by avoiding to process too many redundant operations at thread level, thus multiplying the registers use. The following design intends to reduce the registers count per thread while increasing the threads computation load. It also allows to reduce the effect of global memory access latencies.

Since each thread processes m input pixels, it has to carry out m selections. Thus, the window overlapping cannot be exploited in the same way as it is achieved within histogram-based solutions. Nevertheless, the window overlapping implies that these m selections are not independent. Thus, an efficient implementation must avoid redundant extrema identifications. This is achieved by choosing the R_k elements of the first selection step among those shared by all windows. It only makes sense if the consecutive windows actually share at least R_k elements. For example, if $m = 2$, k must be greater or equal to 5. The first selection steps can then be considered common to all windows. The following selection steps apply on the remaining pixels, which are shared by fewer windows.

This principle was already exploited in the 2D PRMF design, but was limited to $m = 2$. Indeed, when two 2D consecutive windows overlap, a total of $2 \times k$ pixels are not shared and have to be processed separately, using two

Algorithm 1: Code generator for the forgetful selection. The generator actually writes the required number of `#define minMax_i` macros inside the CUDA code, in order to achieve the forgetful selection process. Each minMax_i function works on a set of i single variables a_m stored in registers. During execution of each minMax_i function, a swapping function $s(a_m, a_n)$ is called multiple time to rearrange a_m and a_n in ascending order whenever necessary.

input : r, the radius of the $(2r+1) \times (2r+1)$ pixels filtering window
input : $(a_0 ... a_P)$ the set of pixel values stored in registers
output: The set of `#define minMax_i` definitions

```
1  kmin ← r + 2;
2  for i ← kmin to 3 do
       output: "#define minMax_i(a_0, ..., a_(i−1)) "
       // Init sorting network loop
3      step ← 1;
4      ia ← 0;
5      N ← i;
       // first stage (common to min and max selection)
6      minList ← empty list of integers;
7      maxList ← empty list of integers;
8      while (ia + step < N) do
9          ib ← ia + step;
           output: "s(a_ia, a_ib);"
10         ia ← ia + 2 × step;
11     end
12     if (N%2 = 1) then
13         minList.add(N-1);
14         maxList.add(N-1);
15     end
16     step = step × 2;
17     while (step < N) do
           // minima
18         taille ← sizeOf(minList) − 1;
19         minList.clear();
20         idmin ← 0;
21         while idmin + 1 < taille do
               output: "s(a_minList(idmin), a_minList(idmin+1));"
22             idmin ← idmin + 2;
23         end
24         if (idmin < taille) then
25             minList.add(idmin);
26         end
           // maxima
27         taille ← sizeOf(maxList) − 1;
28         maxList.clear();
29         idmax ← 0;
30         while idmax + 1 < taille do
               output: "s(a_maxList(idmax), a_maxList(idmax+1));"
31             idmax ← idmax + 2;
32         end
33         if (idmax < taille) then
34             maxList.add(idmax);
35         end
36         step ← step × 2;
37     end
       output: "\n"
38 end
```

different groups of registers. For example, if $k = 5$, there are 20 pixels shared by the two 2D windows. The selection starts using $R_{5\times5} = 14$ pixels, and thus registers. After seven selection steps, all shared pixels have been processed and there are only 6 registers that contain relevant values for the next steps. Their values are copied to a new group of 7 registers that is used to process the selection for the second window, while the original group of registers is used for the first window. In total, the process uses $14 + 7 = 21$ registers, instead of $14 + 14 = 28$ if overlapping is not exploited.

Nevertheless, when processing two 1D windows, using the same k as in 2D, the computation load of a thread is very low. For example, if $k = 5$, a thread processes only 6 pixels, compared to 30 in 2D. Since $R_5 = 4$, our selection process uses 7 registers instead of 8 if both windows are processed independently. But even if there is a gain, experiments have demonstrated that it is not sufficient to compensate the weak load of each thread. This is why the method needed to be extended to larger pixel packets for 1D filters. Fig. 7 details the extended method in the case of a 9×1 horizontal median processing four pixels per thread ($m = 4$). A solution that does not take the overlapping into account would use $m \times R_9 = 4 \times 6 = 24$ registers. Our method drastically reduces this number. Each of the m center pixels is displayed in a specific color and is associated to a group of registers printed in the same color. These groups are numbered from 0 (the blue one) to 3 (the yellow one), and registers of group i are named x_{i*}. The numbers in the square pixels are sample gray-level values. The first stage consists in devoting group 0 (blue) for each of the $R_9 = 6$ pixels shared by all filtering windows. The first $minMax[R_k]$ (here minMax6) selection stage is then achieved, which frees two registers (the min and the max). The four remaining relevant registers of group 0 are copied to group 1 (red), dedicated to the second center pixel. Both groups are next augmented with one pixel value chosen among the remaining pixels, starting from the innermost ones. Now two $R_k - 1$ (here minMax5) selection stages are achieved. This duplication-augmentation-minMax process is then repeated. At each iteration i, group i is created and filled by copy from the relevant registers of group $i - 1$, then for each group, one of the remaining pixels is integrated, before executing the minMax. It goes until four minMax3 selection steps output the four median values. In this example, groups have respectively a size of 6, 5, 4 and 3, leading to a total of 18 registers, instead of 24 for a naive implementation. For $m = 4$, the number of registers used by each thread is then given by $(R_k + 12)$, while it is $(R_k + 42)$ for $m = 8$ (provided k is large enough). It is worth noting that since each thread uses more registers than with $m = 1$, it may lead to overpass the limit of the register count per thread block. Nevertheless, it is easy to stay under this limit by dividing the block size by m, while preserving the grid size.

The GPU specific access patterns required to ensure coalescence to and from global memory lead to choose packet sizes m as powers of two. The maximum packet size m_{max} is then the greater power of two that remains less than $k/2$. Above $k = 65$, all packet sizes can be chosen between 2 and 32.

The case $k = 3$ cannot be treated in the same way, as no overlapping can be exploited at this size. However, devoting one thread to processing only one pixel does not provide a satisfying level of performance, because of unbalanced computations and communications (memory accesses). Instead, experiments show that the highest throughput is obtained by processing $m = 4$ pixels per thread, without taking the overlapping into account. It appears that it is also the case for 5×5 and 7×7, even though sufficient overlapping exists at these particular sizes. That confirms that computation load has a higher impact than the parallelism level, which is not a limiting factor in our design, for small sizes.

3.4 Optimizing the sequence of both 1D filtering stages

As already stated, the choice of running the horizontal 1D step first is arbitrary and does not impact the filtering performances. Separable implementations (not only median operators) frequently adopt a symmetrical architecture that allows the same function to execute twice, for both vertical and horizontal stages, separated by an image transposition. The proposed implementation does not follow this scheme and uses one specific GPU kernel for each 1D stage. This is imposed by the memory allocations and alignment constraints that must be fulfilled in order to achieve optimal accesses to and from the GPU global memory.

Three global memory areas are allocated onto the GPU before calling the filtering kernels: d_in, d_outh and d_outv. Assuming an input image of M rows and N columns, stored row after row :

- d_in contains the input image. It has the same height as M, but is larger. In order to avoid special processing, the input image must be horizontally zero-padded by adding $\lfloor \frac{k}{2} \rfloor$ columns on both the right and the left sides. To limit non coalescent accesses to global memory, the first column of d_in must be 128-Byte aligned. Under these conditions, the width of d_in is the closest multiple of 128 above $N + k - 1$. It is filled with zeroes and the input image is stored starting at row 0 and column $\lfloor \frac{k}{2} \rfloor$.
- d_outh receives the intermediate image output by the horizontal kernel that will then become the input of the vertical kernel. For the same reasons as for d_in, the output needs to be vertically zero-padded by adding $\lfloor \frac{k}{2} \rfloor$ rows on both bottom and top sides and the first column of d_outh must be 128-Byte aligned. Thus, the height of d_outh is $M + k - 1$ and its width is the closest multiple of 128 above N. It is filled with zeroes and the intermediate image is stored starting at row $\lfloor \frac{k}{2} \rfloor$ and column 0.
- d_outv receives the output image. It only requires a 128-Byte alignment. The first pixel column of the image is written in column 0.

To summarize, the optimization techniques described above allow each thread to use fewer registers while processing more pixels, thus increasing

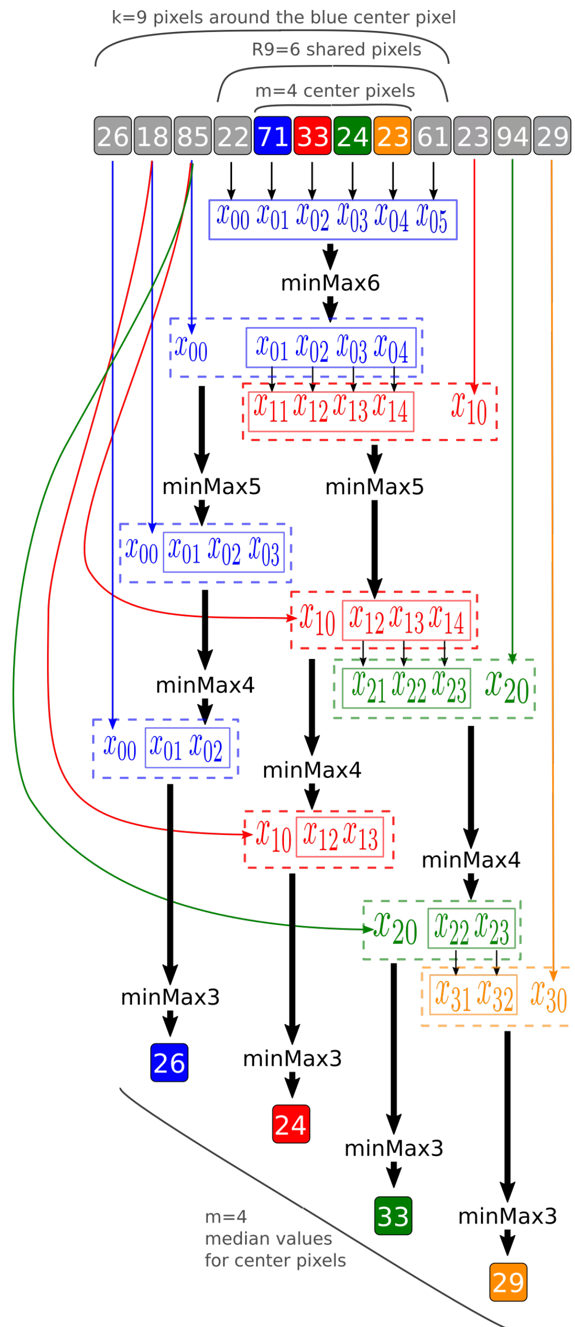


Fig. 7 Reducing register count in a 9×1 register-only median kernel by processing 4 input pixels in parallel (in blue, red, green and orange). The 4 filtering windows are numbered from 1 to 4 and contain gray-level values printed in white. A set of 4 groups of registers x_{ij} is defined by its color and by the number of the associated filtering window as i index. The first forgetful selection step is processed once for the 4 filtering windows, as $R_9 = 6$ is the exact count of pixels that are shared by all of the 4 windows. During each following step, the leftmost branch of register groups follows the standard forgetful selection process until the median value is output. At the same time, the rightmost branch of register groups is divided after each selection step in order to devote, at the end, one group to each filtering window. In this example, only 18 GPU registers are used by each thread to process four 9×1 median values.

the level of parallelism and allowing larger filtering windows without suffering a drastic drop in performances, which occurs when the register file limit is reached. In addition, global memory accesses are coalescent and cache misses are minimized. All of these properties contribute to providing a high level of performance that is demonstrated in the next sections.

4 Experimental setup

This section describes the software and hardware context to evaluate the applicability of the results of Section 2 to grayscale images, and the performances of our GPU implementation presented in Section 3.

4.1 Context of the statistical comparison of full and separable median filters

The approach has been to conduct a statistical comparison over a large number of natural images and odd window filtering sizes. The chosen images are a collection of 10,000 gray-level 8-bit depth Portable Gray Map natural pictures from the BOSSRank database [14], whose original purpose was to benchmark steganalysers. These images, called cover images are meant to represent original images, *i.e.* images with no significant and/or visible noise. During the simulations, cover images have been corrupted by salt and pepper noise with different strengths: 7%, 10%, 25% and 50%, which cover a wide range from light to strong power of noise.

Measurements have been conducted to compare the output of both full and separable median filters for filtering sizes ranging from 3×3 to 131×131. For each pair of filtered images, the difference between output images has then been measured through PSNR[7] and MSSIM[8] [24] indicators. Absolute values have not been displayed as they have been detailed in the literature during the past decades and would not provide any additional information.

For any given image $I_{in}(i, nl)$ of size $H \times W$ pixels, where $i \in [1 : 10,000]$ and the noise level $nl \in [0; 1]$, the output images I_{full} and I_{sep} are obtained respectively by filtering $I_{in}(i, nl)$ with a $k \times k$ median filter in its full or separable version. For each given set of parameters, the two main indicators Δ_{PSNR} and Δ_{MSSIM} are obtained by evaluating the following expressions:

$$\begin{cases} \Delta_{PSNR} = PSNR(I_{full}, I_{in}) - PSNR(I_{sep}, I_{in}) \\ \Delta_{MSSIM} = MSSIM(I_{full}, I_{in}) - MSSIM(I_{sep}, I_{in}) \end{cases} \quad (6)$$

In addition to the average Δ_{PSNR} and Δ_{MSSIM} values, each pair of output images has been analysed to count how many images are best denoised respectively by the full and the separable median filter. This led to produce a supplementary indicator that helps in deciding which filter version to use,

[7] Peak Signal to Noise Ratio.
[8] Mean Structural SIMilarity.

regarding the window size as well as the type and power of the noise. This indicator is simply displayed in the various figures as a label with either of the following texts: “full likely better” or “separable likely better”.

Eventually, another batch of measurements has been conducted in order to confirm what the study of the bi-chromatic case has revealed: there are more color combinations where only the separable filter would alter the central pixel, but they are less likely, inside the filtering window, than those where the full would be the only one to alter the central pixel. To avoid mixing noisy and non-noisy pixels, both median filters have been applied on every cover images of the database and the two interesting situations described above have been counted.

To avoid any doubt regarding the simulation results, all measurements have been performed under Matlab R2017b, using standard functions like medfilt2(), psnr() and the original mssim implementation proposed in [24].

4.2 Evaluation context

Results have been obtained by averaging runtimes of a variable number of executions. The actual number of runtimes was dynamically set to ensure a total execution time under 0.5 second. This setup avoids any overhead caused by the RCU scheduler on Linux 3.16.0-4-amd64 host platform powered by one Xeon E5620 @2.40 GHz processor and CUDA v9.0. Each kernel has been run by Titan-X[9] and V100[10] GPUs on 8 bit images of sizes 1,024×1,024 and 8,192×8,192. Cover images of the BOSS database, whose original size is 512×512, have then been resized and corrupted by a medium power salt and pepper noise of 25%.

The pixel throughput value of the GPU kernels was used as a main performance indicator. It does not include transfer times to and from the GPU, as median filter kernels are likely to be part of a complex process fully run on the GPU.

To evaluate the absolute performance of the proposed implementation, the maximum effective pixel throughput that our own GPU/host couple is able to achieve has been measured. Such a peak value helps in deciding on further investigation. We performed this measurement by running dummy kernels that fetch the gray-level of each pixel from GPU's global memory and output it into another area into the GPU's global memory, exclusive of any other instruction. Each thread may process several pixels in order to balance the compute and read/write loads.

An experimental tuning stage shows that processing four pixels by thread achieves the peak pixel throughput of respectively 80 (Titan-X) and 200 (V100) billion pixels per second on an 8192×8192 pixel image. The most significant

[9] Titan-X GPU: Maxwell family, compute capability 5.2, 3,072 cuda cores at 1.24 GHz, 12 GByte of RAM.

[10] V100 SXM2 GPU: Volta family, compute capability 7.0, 5,120 cuda cores at 1.45 GHz, 32 GByte of RAM.

factor in the speedup brought by V100 is its memory bandwidth of 900 Go/s, compared to 336 Go/s for Titan-X. This speedup of ×2.5 achieved by the Volta V100 against the Titan-X is observed whatever the median kernels executed on both GPU models. That led to display only the results of the most recent V100 GPU model. If needed, readers can easily deduce Titan-X throughputs values by dividing those of V100 by 2.5.

5 Results

The first part of this section presents the denoising quality obtained for the considered image set, using PSNR and MSSIM metrics. It is followed by a comparison on the pixel corruption generated by both versions. Finally, an analysis of the computational complexity and performance is given.

5.1 Comparison of full and separable median filters under salt and pepper noise

Fig. 8 displays the evolution of the average Δ_{PSNR} and Δ_{MSSIM} values when k ranges from 3 to 131 on the whole set of images. It should be noted that on average, whatever the noise level, the separable median filter achieves better PSNR and MSSIM values for every window size above or equal to 11×11, and the average differences tend very rapidly towards the pair of values below :

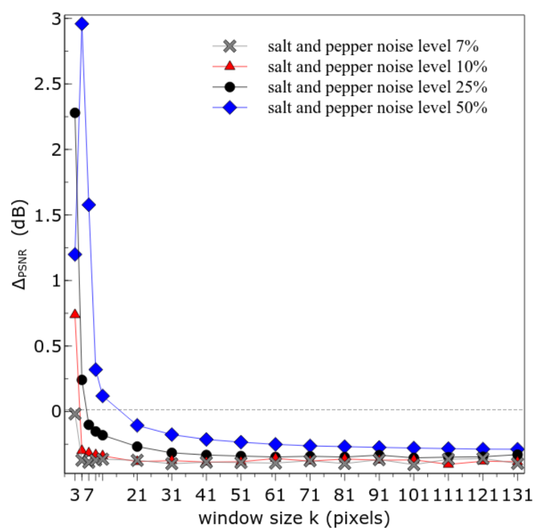
$$\begin{cases} \lim\limits_{k \to min(H,W)} \Delta_{PSNR} = -0.4\ dB \\ \lim\limits_{k \to min(H,W)} \Delta_{MSSIM} = -0.005 \end{cases}$$

Another interesting fact is detailed by Fig. 9 which displays the percentage of images, among the whole database, that are best denoised by full median filtering. It reveals that, whatever the power of noise, the separable version gives better results above 21×21, and that below 25% of salt and pepper noise, this threshold is downsized to 7×7.

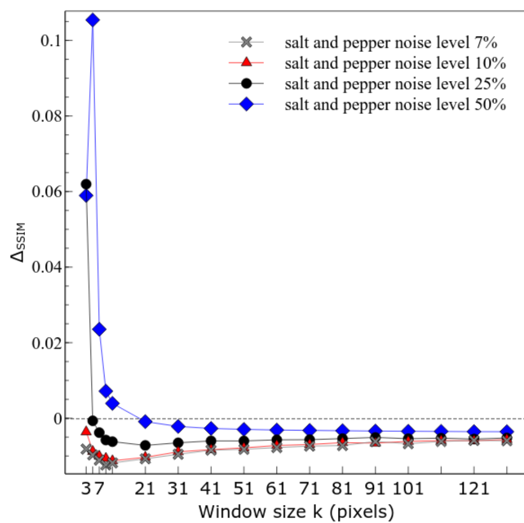
Given the filtering size and the power of noise, Fig. 9 can then provide a rule for selecting the best type of median filter to use, from a statistical point of view: below 50%, on average, the separable filter is probably the best choice.

It is also important to note that, up to a 7% of salt and pepper noise, separable filtering is always better than full filtering.

On the basis of the above statistical study, and with all the precautions that must be taken regarding the data-dependency of median filtering as well as its intrinsically limited scope (though quite large) provided by the 10,000 natural scenes of the database, the conclusion is that in common situations, separable median filtering is likely to achieve higher salt and pepper noise reduction than full median filtering. As for larger filtering windows often used for other reasons than noise reduction, both filters have a very close behavior which is in favor of the much faster separable version.

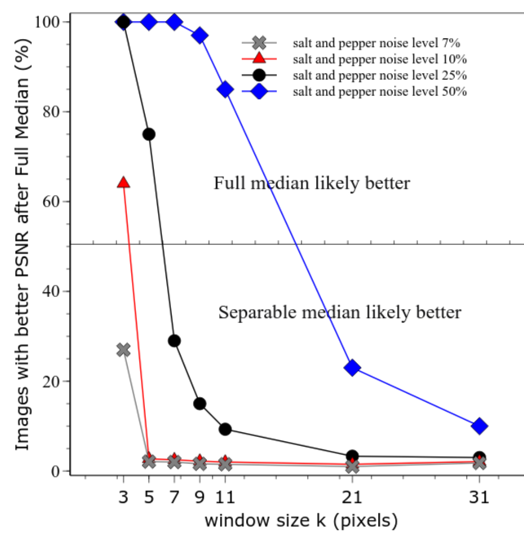


(a) $\Delta_{PSNR}(dB)$

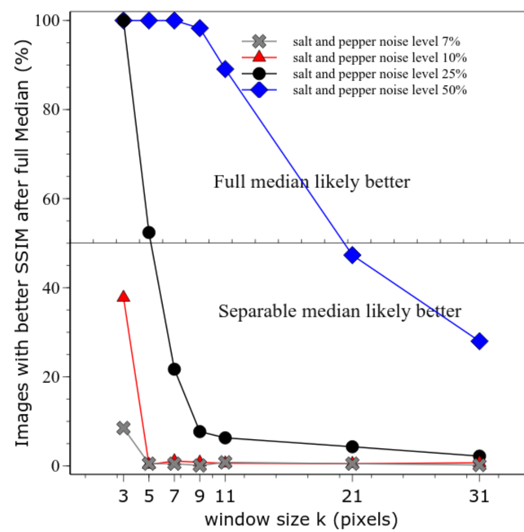


(b) Δ_{SSIM}

Fig. 8 Relative performance of the full median filter compared to its separable version, on average, over 10,000 natural images for filtering window sizes ranging from 3×3 to 131×131. A positive value indicates that full filtering is better than the separable one.

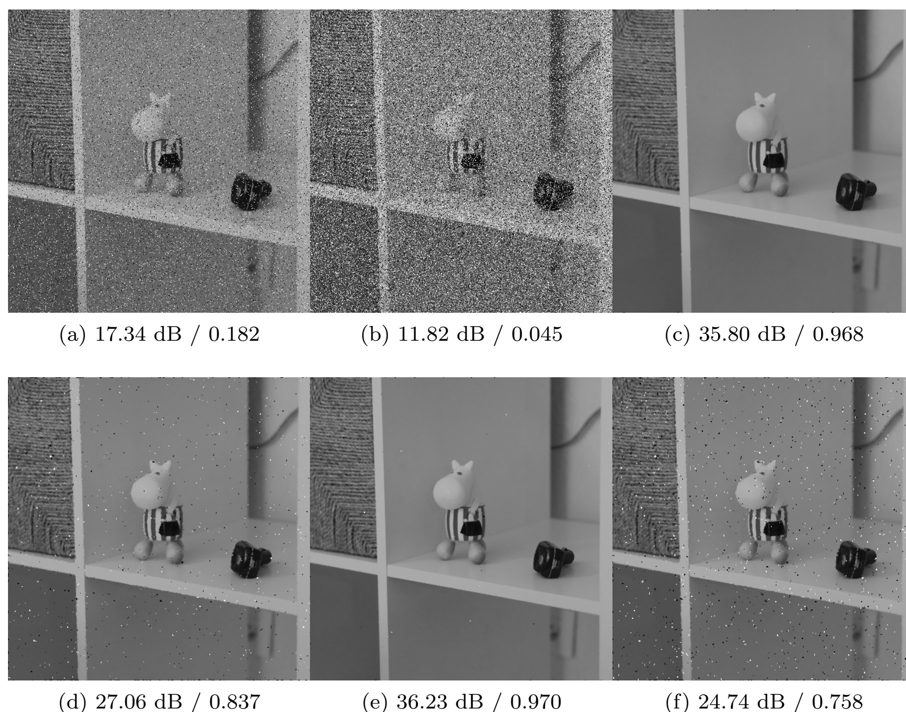


(a) Ranking on PSNR value



(b) Ranking on SSIM value

Fig. 9 Percentage of images that are best denoised by Full Median filter, over 10,000 natural images for filtering window sizes ranging from 3×3 to 31×31 under salt and pepper noise corruption. Above 50%, the full median will more likely provide a better output image than the separable version.



(a) 17.34 dB / 0.182 (b) 11.82 dB / 0.045 (c) 35.80 dB / 0.968

(d) 27.06 dB / 0.837 (e) 36.23 dB / 0.970 (f) 24.74 dB / 0.758

Fig. 10 Example of a 512×512 image, corrupted by 7% (a) and 25% (b) salt and pepper noise and then processed by both full and separable 3×3 filters. c) and d) are filtered using the full version while e) and f) are filtered by the separable one. For each image, results are given through PSNR(dB)/MSSIM.

An example is presented in Fig. 10, where image #9850 of the database, randomly chosen, is displayed in two configurations: after corruption by 7% and 25% salt and pepper noise (images a and b). Images c and d show the results after a full 3×3 median filtering stage, while images e and f display the results of the separable 3×3 filtering stage. The visual comparison confirms the statistical results of Fig. 9 and Fig. 8 with very close output images in the 7%-noise case, while the full filter outperforms the separable one in the 25%-noise case.

To illustrate the discussion about the filtering size, Figure 11 shows the results of the same processes applied on a 8192×8192 image corrupted by a 25% salt and pepper noise. For a better visibility, a small square region of interest is displayed, making it easier to compare both output images. In this configuration, a 15×15 filter is well adapted to noise reduction and both filters visually produce very close results.

(a) 10.61 dB / 0.015

(b) 10.61 dB / 0.015

(c) 40.97 dB/ 0.994

(d) 47.74 dB / 0.994

Fig. 11 Example of a 8192×8192 image, corrupted by a 25% salt and pepper noise (a) and then processed by both full and separable 15×15 filters. a) shows the whole image, b) focus on a 681×681 pixel square around the last letters "US" on the right side. Images c) and are filtered using respectively the full and the separable version. For each image, results are given through PSNR(dB)/MSSIM and are computed on the whole images.

5.2 Comparison of the corruption generated by separable and full filters

The percentages of non-noisy pixels corrupted by only one of the median filters, separable or full, are plotted on Fig. 12. It confirms that in the range of our experiments, the full filter always alters more pixels than the separable version, though the difference tends to decrease when the window size increases. The maximum difference of 6.68 points is observed for the smallest size while the minimum of 2.64 points is reached at the largest size of the experiment, *ie.* 131×131. It represents between 17,500 and 6,900 pixels for the 512×512 images of the database.

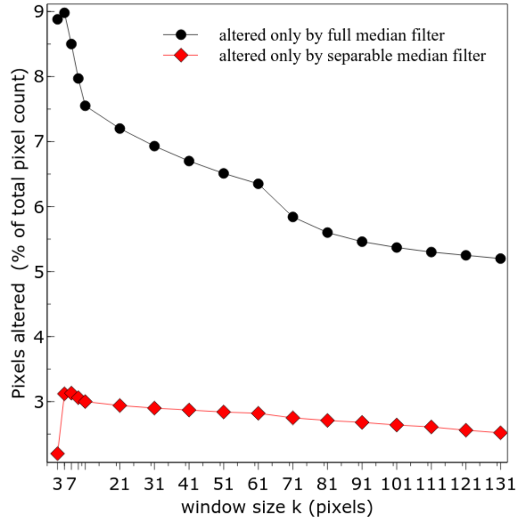


Fig. 12 Percentage of non-noisy pixels corrupted by only one of the median filter versions and not the other. The percentages represent the total pixel count, over the 10,000 natural images of BOSS database, for filtering window sizes ranging from 3×3 to 131×131.

5.3 Computational complexity analysis

As presented above, computations are mostly achieved by means of multiple calls to a simple swapping function, that actually swaps its two input parameters if their values are not in ascending order. A good approach to the complexity is given by the number of times each median filter calls this function. It actually represents the worst case, as each call will not necessarily result in an actual swap.

For each processed pixel, the proposed separable filter realizes respectively 3, 10 and 16 calls for the smallest sizes 3×3, 5×5 and 7×7 because no overlapping is exploited. For sizes ranging from 9×9 to 19×19 the packet size is set to $m = 4$ and the number of calls is given by:

$$W_{sep}(k) = 18 + \frac{1}{2}\sum_{i=6}^{i=6+\frac{k-9}{2}} sw(i) \tag{7}$$

$$\text{where} \quad sw(i) = \begin{cases} 3\frac{i-1}{2} + 1 & , i \ odd \\ 3\frac{i}{2} & , i \ even \end{cases}$$

Above 21×21, the packet size is set to $m = 8$ and the number of calls is then given by:

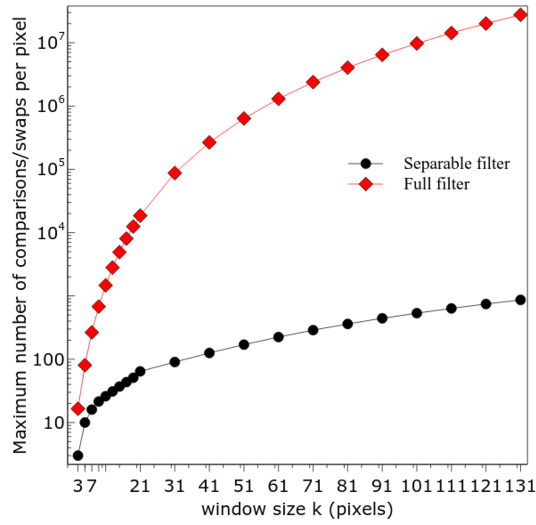


Fig. 13 Complexity of both PRMF (full) and PRSMF (separable) implementations, for filtering window sizes ranging from 3×3 to 131×131. The complexity is displayed as the total number of calls to the swapping function, according to real executions.

$$W_{sep}(k) = 64.25 + \frac{1}{4} \sum_{i=10}^{i=12+\frac{k-21}{2}} sw(i) \tag{8}$$

According to Equations 7, 8 and results from [10], the complexity of processing a **single pixel** for the proposed separable median filter is $O(\frac{1}{4}klog(k)^2)$, which is close to k^3, as opposed to $O(\frac{1}{10}k^4)$ for the PRMF full version.

It is illustrated in Fig. 13 where the proposed PRSMF implementation is compared to the PRMF regarding the real (i.e. during executions) total number of calls to the swapping function made by both filters. Except for a flat area in the PRSMF results, curves perfectly match the given complexities.

5.4 Performances of the proposed implementation

Kernel throughput values of the proposed implementation called PRSMF[11] are presented in Fig. 14 for 1,024×1,024 and 8,192×8,192 pixel images and compared with Nvidia's NPP, Green's new GPU 2D filter [21] and our previous PRMF 2D filter [10]. Green's new GPU implementation represents one of the most recent and most effective 2D median filter but does not run in a separable way. As we do not have access to the source code, the timings

[11] Parallel Register-only Separable Median Filter

on V100 have been extrapolated from [21], where a comparison on Titan-X against PRMF is detailed; we trusted author's results and used the constant $\times 2.5$ speed factor measured between Titan-X and Volta (see 4.2) to produce a relevant comparison. Nvidia's NPP is a closed code library which appears to be currently the only one that allows to run in the separable way with reasonable throughputs.

The maximum throughputs of PRSMF are reached by the 3×3 filter, processing $m = 4$ pixel per thread, with around 125 billion pixels per second on V100. That represents around 62.5% of the peak throughput (for both GPUs). The optimal value for m depends on the window size k but only takes two different values: $m = 4$ when $k < 21$ and $m = 8$ when $k \geq 21$. This behavior confirms that the computation load has a higher impact than the parallelism level for relatively small windows, as the smallest size for which $m = 8$ filters use less registers per pixel than $m = 4$ filters is $k = 33$, according to section 3.3.

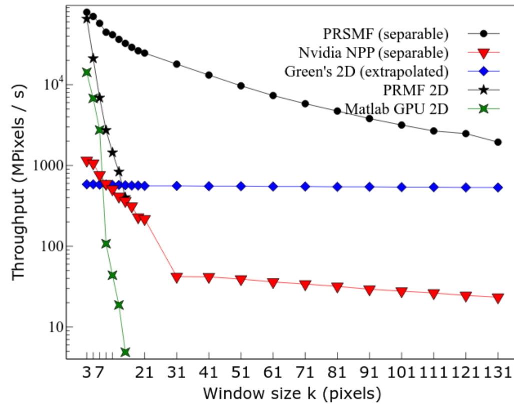
The performances of PRSMF are between 9 and 24 times faster than the Nvidia NPP library, which seems to implement two different methods depending on the filtering size: below $k = 21$, the plot looks like a sorting-based implementation while from $k = 31$ and above, the plot seems to reveal a histogram-based implementation (near constant time). As for Green's implementation, the achieved throughput is almost constant whatever the filtering and image sizes, while PRSMF's throughput increases with the image size. Still, Green's implementation achieves lower throughputs whatever the filtering size, at least up to $k = 131$ where the lowest speedup brought by PRSMF is around $\times 5$ and reaches around $\times 200$ as its highest value (3×3, $8{,}192\times8{,}192$ pixel image).

It is important to notice that the performances of these kernels on 16-bit images remain the same as when run on 8-bit images.

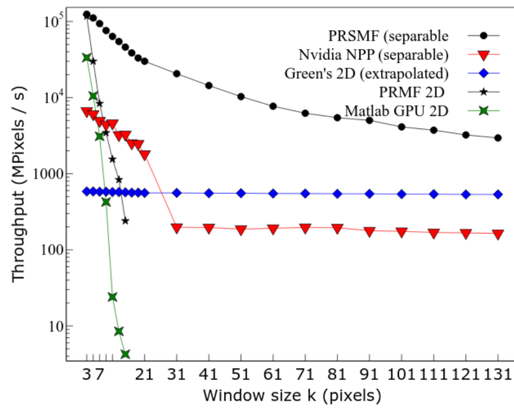
Additionally, it is worth noting that the proposed implementation also optimizes data transfers by allocating aligned memory on the GPU side and pinned memory on the CPU side. As a reference value on both Titan-X and V100, 9.0 ms are needed to copy a $8{,}192\times8{,}192$, 8 bit depth image from CPU to GPU's global memory, and 5.2 ms are necessary to copy back the same image to CPU's pinned-memory. The same transfer from GPU to CPU, using a classical pageable-memory allocation on CPU side(malloc) will cost 40 ms on both GPU models. Notice that transfer times vary linearly with pixel depth and are thus doubled for 16-bit images with respect to 8-bit images.

6 Conclusion

In this paper, extensive study and theoretical considerations have proved that statistically, better 2-D median filtering is obtained through a separable version than classical full 2-D filtering. In addition, a very high speed implementation has been designed, which makes it possible to process up to 50 billion pixels per seconds on a very common Titan-X GPU, which represents for example, more



(a) 1024×1024 input image.



(b) 8192×8192 input image.

Fig. 14 Pixel throughput value comparison, in million pixels per second, of Nvidia's NPP, Green's and PRMF implementations against the proposed PRSMF, on Volta V100 GPU.

than 6000 Ultra High Definition (or 4K) images per second. An unprecedented throughput of 125 billion pixels per second has been achieved on a high-end V100 GPU model.

Let us also note that the throughput values achieved by 3×3 window filters come quite close to the measured peak effective pixel throughput allowed by our development platforms. Hence, the search for further improvement seems to be conditioned by new GPU architecture capabilities.

Indeed, new perspectives have been opened with the release of Volta and Turing family GPUs and not yet investigated in this work because of its very recent availability. For example, independent thread scheduling and collaborative groups could be promising features to be utilized.

The generalization of the comparison of the separable 2D median filter and the full 2D median in presence of additive white Gaussian noise has already

been conducted and confirms the average superiority of the separable median (not discussed here due to the limited number of pages). Likewise, it would be relevant to study the same situation with the 3D median filter and other noise types. Finally, we plan to revisit other non-linear filters with the pseudo-separability in mind.

Source codes of the proposed implementation can be generated online and downloaded for testing purposes at:
http://info.iut-bm.univ-fcomte.fr/staff/perrot/convomed/ In addition, full definition images and more measurements can be found at:
http://info.iut-bm.univ-fcomte.fr/staff/perrot/separable-median/

Acknowlegments

This work was partially supported by the EIPHI Graduate School (contract ANR-17-EURE-0002). Computations have been performed on the supercomputer facilities of the "Mésocentre de Franche-Comté".

References

1. Batcher KE (1968) Sorting networks and their applications. In: Proceedings of the April 30–May 2, 1968, spring joint computer conference, ACM, New York, NY, USA, AFIPS '68 (Spring), pp 307–314, DOI 10.1145/1468075.1468121, URL http://doi.acm.org/10.1145/1468075.1468121
2. Adams, Andrew (2021) Fast Median Filters Using Separable Sorting Networks. In: ACM Transactions and graphics, August 2021, Association for Computing Machinery, New York, NY, USA, DOI 10.1145/3450626.3459773, URL https://doi.org/10.1145/3450626.3459773
3. Chen W, Beister M, Kyriakou Y, Kachelries M (2009) High performance median filtering using commodity graphics hardware. In: Nuclear Science Symposium Conference Record (NSS/MIC), 2009 IEEE, pp 4142–4147, DOI 10.1109/NSSMIC.2009.5402323
4. Huang TS (1981) Two-Dimensional Digital Signal Processing II: Transforms and Median Filters. Springer-Verlag New York, Inc., Secaucus, NJ, USA
5. Kachelriess M (2009) Branchless vectorized median filtering. In: Nuclear Science Symposium Conference Record (NSS/MIC), 2009 IEEE, pp 4099 –4105, DOI 10.1109/NSSMIC.2009.5402362
6. Sanchez R, Rodriguez P (2012) Highly parallelable bidimensional median filter for modern parallel programming models. Journal of Signal Processing Systems pp 1–15, DOI 10.1007/s11265-012-0715-1, http://dx.doi.org/10.1007/s11265-012-0715-1
7. Tukey JW (1977) Exploratory Data Analysis. Addison-Wesley
8. Weiss B (2006) Fast median and bilateral filtering. In: ACM SIGGRAPH 2006 Papers, ACM, New York, NY, USA, SIG-

GRAPH '06, pp 519–526, DOI 10.1145/1179352.1141918, http://doi.acm.org/10.1145/1179352.1141918
9. Perreault, S., & Hébert, P. (2007). Median filtering in constant time. IEEE transactions on image processing, 16(9), 2389-2394.
10. Perrot, G., Domas, S., & Couturier, R. (2014). Fine-tuned High-speed Implementation of a GPU-based Median Filter. Journal of Signal Processing Systems, 75(3), 185-190.
11. Cline, D., White, K. B., & Egbert, P. K. (2007, September). Fast 8-bit median filtering based on separability. In Image Processing, 2007. ICIP 2007. IEEE International Conference on (Vol. 5, pp. V-281). IEEE.
12. Tukey, J. W. (1978). The ninther, a technique for low-effort robust (resistant) location in large samples. Contributions to Survey Sampling and Applied Statistics in Honor of HO Hartley, 251-258.
13. Narendra, P. M. (1981). A separable median filter for image noise smoothing. IEEE Transactions on Pattern Analysis and Machine Intelligence, (1), 20-29.
14. Cover database used to generate the Testing database: http://boss.gipsa-lab.grenoble-inp.fr/Warming/Materials/BOSSRank-covers.tar.bz2
15. Nividia Kepler architecture: https://en.wikipedia.org/wiki/Kepler_(microarchitecture)
16. Nvidia Maxwell architecture: https://developer.nvidia.com/maxwell-compute-architecture
17. Nvidia Pascal architecture: https://developer.nvidia.com/pascal
18. Nvidia Volta architecture: https://devblogs.nvidia.com/inside-volta/
19. Nvidia NPP library: https://developer.nvidia.com/npp
20. Paulius, Micikevicius, Performance Optimization: Programming Guidelines and GPU Architecture Reasons Behind Them, GPU Technology Conference, 2013. http://on-demand.gputechconf.com/gtc/2013/presentations/S3466-Programming-Guidelines-GPU-Architecture.pdf
21. Green, O. (2018). Efficient Scalable Median Filtering Using Histogram-Based Operations. IEEE Transactions on Image Processing, 27(5), 2217-2228.
22. Battiato, P. S. (2016). High Performance Median Filtering Algorithm Based on NVIDIA GPU Computing.
23. Pei-Eng Ng and Kai-Kuang Ma, "A switching median filter with boundary discriminative noise detection for extremely corrupted images," in IEEE Transactions on Image Processing, vol. 15, no. 6, pp. 1506-1516, June 2006. DOI 10.1109/TIP.2005.871129
24. Zhou Wang, A. C. Bovik, H. R. Sheikh and E. P. Simoncelli, "Image quality assessment: from error visibility to structural similarity," in IEEE Transactions on Image Processing, vol. 13, no. 4, pp. 600-612, April 2004. DOI 10.1109/TIP.2003.819861

Declarations

Funding

The Graphical Processing Units used in the described experiments have been provided by the Mésocentre de calcul de Franche-Comté, the regional computing facility.

Conflicts of interest/Competing interests

Authors declare that they do not have no conflict of interest of any sort.

Availability of data and material

Source codes of the proposed implementation can be generated online and downloaded for testing purposes at: http://info.iut-bm.univ-fcomte.fr/staff/perrot/convomed/
In addition, full definition images and more measurements can be found at:
http://info.iut-bm.univ-fcomte.fr/staff/perrot/separable-median/

Code availability

Sample source codes are available via the above URL, but the source code of the kernel generator is beyond the scope of this paper and is right protected.