AUTOMATYKA • 2009 • Tom 13 • Zeszyt 3

Sebastian Stoliński*, Szymon Grabowski*, Wojciech Bieniecki*

On Efficient Implementations of Median Filters in Theory and in Practice

1. Introduction

The median filter is a classic tool for impulse noise reduction in grayscale images. It replaces the central pixel in a mask (often square-shaped) with the pixel with the median intensity within this neighborhood. This rank (and thus non-linear) approach is robust to significant distortions (noise) on single pixels, which would strongly affect e.g. the mean value in the mask.

The neighborhood window W (also called a "window") usually contains an odd number of pixels, which guarantees that the median will be the value of some pixel existing in the neighborhood. This is undoubtedly beneficial. Other assets of a median filter are easiness of implementation and high speed.

Multi-channel (e.g. RGB) images require a different definition for median filtering, because there is no natural ordering of elements in vector space. The observation that logically motivated the vector median filter (VMF) idea was that the (scalar) median of the neighborhood may alternatively be pointed out with the formula: $med = \arg\min_{f_i \in W} \sum_j \left| f_i - f_j \right|$. It is then enough to replace the absolute values of the differences between scalars with distances between vector according to chosen metric (e.g., the Euclidean one).

There are many alternatives of VMF but almost all the work is dedicated to improving the resulting image quality. We are not aware of any serious considerations on how to implement color median-like filters efficiently.

In this paper we present a theoretical algorithm for worst-case optimized scalar median finding and an efficient implementation of VMF. Section 2 discusses various implementations for scalar (grayscale) median filtering. Section 3 briefly surveys color median filtering algorithms. Section 4 presents our novel implementation of VMF. Section 5 contains experimental tests. The last section concludes and points out avenues for future research.

* Computer Engineering Department, Technical University of Łódź

2. The scalar median filter – theory and practice

In this paper we consider square masks only. We assume the input image $I = \{l_{x,y} : 0 \le x \le n-1, 0 \le y \le m-1, 0 \le l_{x,y} \le L-1\}$. In plain words, the image has *nm* pixels and the intensity has L levels. Assume the point (0, 0) is the top left corner, so the top line will be called 0th line etc. Let the radius of the mask be denoted with *r*, so $|W| = (2r + 1)^2$. The most straightforward implementation of the median filter is to sort all $|W|$ intensities and select the median one, for each pixel of the image. It can be achieved in $O(|W|\log|W|) = O(r^2 \log r)$ time using e.g. merge sort. Note however that replacing a comparison based sort with counting sort yields $O(r^2 + L)$ time. For large enough masks we have $L = O(r^2)$ and then counting sort complexity reduces to $O(r^2)$. In the opposite case, $L >> r^2$, we can replace counting sort with e.g. 2-pass radix sort, and obtain $O(r^2 + \sqrt{L})$ complexity. In general, *k*-pass radix sort leads to $O(k(r^2 + L^{1/k}))$ time, which is optimized for $k = \log_{r^2} L$, and the time then becomes $O(r^2 \log_{r^2} L) = O(r^2 \log_r L)$.

An alternative to sorting is using a worst-case linear-time selection algorithm, which is only a theoretical option: very complicated [3] and/or accompanied with an impractically high constant [2]. This way, however, the filtering time gets $O(r^2)$.

We can achieve this complexity with simpler and more practical means. The current mask of size $|W|$ shares $(2r+1)*2r$ pixels from the previous mask. Assume that the image is scanned along the horizontal lines and also that we remember the sorted intensities for the previous mask. For the current pixel we sort the $2r + 1$ intensities from the rightmost column of the mask, in $O(r \log r)$ time, and then merge the old mask with the new $2r + 1$ values, "dropping" on the way the $2r + 1$ values corresponding to the leftmost column of the previous mask. This has $O(r \log r + r^2) = O(r^2)$ time, as promised.

Note that with no approach presented so far processing a single pixel cannot be achieved in time sublinear in the mask size. Yet in 1979 Huang [11] presented a surprisingly simple and efficient median filter implementation, based on incremental updates of the intensity histogram. When moving from one pixel to its successor (e.g. along a horizontal line), only $2r + 1$ additions and $2r + 1$ subtractions need to be done to update the histogram. Now, the median can be found from the histogram via performing at most L additions, which results in $O(r + L)$ worst-case time per pixel but typically it works much faster, as the median values for two successive masks are usually close and it is enough to trace the histogram bins close to the one containing the median for the previous mask. Another speedup idea is to use wide machine words (SSE) (applied for example in [13]), in order to inspect several ($\Theta(w/\log r)$ in theory) histogram bins in parallel. Still, if we want a more radical improvement in worst-case time complexity, we may represent the histogram with a balanced binary search tree, with intensity levels as keys and the size of each subtree stored in the corresponding root (maintaining those counters does not deteriorate the logarithmic complexities of the standard balanced BST operations). This idea is mentioned e.g. in [7]. The achieved time complexity is $O(r \log(r^2)) = O(r \log r)$ and can be improved to $O(r \log \min(r^2, L))$. Interestingly, instead of using a balanced BST, we can make use of two binary heaps (one for values smaller than the median, one for values greater than or equal the median), preserving the complexity but making the implementation simpler. This idea seems to belong to programming folklore [9].

Now we show how a balanced BST may on a RAM machine be replaced with the van Emde Boas (vEB) structure [4], with which keys from a universe $U = \{0, 1, ..., |U| - 1\}$ can be searched for in time (this idea is not novel either, a similar one can be found at [10]). Since $|U|$ is L in our case, this translates to overall $O(r \log\log L)$ time per pixel. We maintain the vEB structure and additionally an array $A[0, L - 1]$ of L counters. Assume that for a given mask centered at pixel (i, j) we know the median, its location ℓ in A and the cumulative sums $A[0, \ell - 1]$ and $A[\ell + 1, L - 1]$ (note that the latter can be trivially calculated as $r^2 - A[0, \ell - 1] - A[\ell]$). Now, if we proceed to the next mask, centered at $(i + 1, j)$, for $2r + 1$ added pixels we perform the same operations: one insertion into the vEB ($O(\log\log L)$ time), one increment to A ($O(1)$ time), maintaining ℓ and the cumulative sum $A[0, \ell - 1]$ ($O(1)$ time) and possibly finding the next ℓ' to replace ℓ, which requires the successor or the predecessor query in the vEB ($O(\log\log L)$ time). Similar operations are performed for the $2r + 1$ pixels to remove from the mask. Overall, the median is calculated in $O(r \log\log L)$ time per pixel.

Interestingly, the $O(r)$ factor may be decreased. The breakthrough achievement belongs to Gil and Werman [7], who gave an $O(\log^2 r)$-time algorithm, with no dependence on L. Finally, Perreault and Hébert [13] presented an $O(L)$-time algorithm, where the lack of dependence on r was achieved due to the incremental build of the mask in two directions: both horizontally and vertically.

Recent years have witnessed a progress in practical implementations. Weiss [18] presented two algorithms, one working in $O(\log^2 r)$ time for “arbitrary-depth images”, but no analysis with respect to L is given. Another algorithm is declared to need $O(\log r)$ time for 8-bit images, but again the L term is not taken into the complexity (and seemingly it is $O(\log r + L)$ in the worst case). Still, also thanks to heuristics, his algorithm work much faster than Photoshop’s median filters, especially for larger masks.

The implementation from the cited work of Perreault and Hébert [13] wins with Weiss’ algorithm if r exceeds about 40, i.e., for huge masks. They applied SIMD (SSE2 on x86 CPUs) instructions, multilevel histograms and other practical tricks. In comparison with Photoshop their filter again wins easily, and its execution time remains fixed for growing r, as predicted by its $O(L)$ complexity.

3. Scalar median in $O(\log L * \log^2 r / \log\log r)$ time

In this section we present an algorithm which achieves similar worst-case time to the Gil and Werman one but uses different means, hence, we believe, may be quite interesting from a theoretical point. Our algorithm, with $O(\log L * \log^2 r / \log\log r)$ time, never dominates over both Gil and Werman and Perreault and Hébert algorithms, but matches them for some relation between L and r (discussion at the end of this section).

Our algorithm makes use of *range quantile queries*, a well-known problem. This problem is often discussed in a restricted from, as range median queries, but some existing algorithms can easily be generalized to replace, at query time, the median with any given quantile. The very recently proposed solution by Gagie *et al.* [6] will serve our purpose.

The problem that Gagie et al. solve can be stated as following. Given a list (e.g., an array) of numbers, preprocess it using possible little space (and preferably time), so as to efficiently answer queries of the form: for a given range (e.g., the indexes of two endpoints of a sublist) and a rank, return the number with that rank from the pointed sublist. The solution in the cited work allows to search for items from integer alphabet of size σ in $O(\log\sigma)$ time, where the extra space cost (paid in the preprocessing) is $n\log\sigma*(1+o(1))$ bits and the preprocessing time is $O(n\log\sigma)$. The key underlying idea of their solution is to use a *binary wavelet tree* [12].

We use the wavelet tree and the Gagie et al. search mechanism over the whole image, many times (to save space, we can divide the image into areas of $\Theta(r)$ contiguous lines and invoke our procedure separately on those image parts). Let $r' = 2r + 1$. We use a parameter $k \geq 2$, whose value will be settled later. Let b_0 be the greatest power of k not exceeding r' (e.g., if $r' = 73$ and $k = 4$, then $b_0 = 4^3 = 64$). In the first (conceptual) pass over the image we consider all stripes (horizontal bars) of the image with their height ib_0/k, for all $1 \leq i \leq k$, and their top line jb_0, for all $j \geq 0$ for which the stripe is fully contained in the image (here and later, just for presentation clarity, we ignore the standard issue with the topmost and bottom rows of the image). Note that there are $\Theta(m/b_0 \times k)$ such stripes in total.

Now, let $b_1 = b_0/k$. In the next pass over the image we consider all stripes with their height ib_1/k, for all $1 \leq i \leq k$, and their top line jb_1, for all $j \geq 0$ for which the stripe is fully contained in the image. There are $\Theta(m/b_1 \times k)$ such stripes, which is (approximately) $b_1/b_0 = k$ times more than in the previous pass.

We continue in the same manner, until $b_{last\_indx} = 1$. Note that $last\_indx = \Theta(\log_k m)$.

Consider now all the stripes obtained in all the passes. The pixels in each stripe are ordered column-by-column taken from left to right, and in that order are (conceptually) copied to an 1D array, one per stripe. Those arrays $C_{v,\,u}$ are identified by two indicators: the number of image rows they cover v and the index u in the image of the top row they cover. For each array we build a wavelet tree.

This is all that we do in the preprocessing stage. The wavelet tree build time is $O(n\log\sigma)$ for a sequence of length n whose symbols are taken from an integer alphabet of size σ. In our case, we note that each image pixel (except possibly the topmost and the bottom rows, which, as mentioned above, we ignore in those considerations) belongs to at most $k\log_k m$ stripes, and the average number of stripes a pixel belongs to is $(k\log_k m)/2$. Our alphabet size is L. From that we directly obtain that the overall preprocessing time is $O(nmk * \log_k m * \log L)$.

Now, we need to notice (and this is the key observation of our algorithm) that each $r' \times r'$ mask can be composed as the union of disjoint contiguous subsequences (ranges) from $O(\log_k r') = O(\log_k r)$ different arrays C. Thanks to the prepared wavelet trees and the Gagie et al. algorithm we can find e.g. the median of each selected range in $O(\log L)$ time but how to find the median of the union of those ranges?

The solution to this problem can be found in the work by Frederickson and Johnson [5] from 1980, where they show an algorithm for finding the median over K sorted arrays working in $O\left(K + \sum_{i=0}^{K-1} \log n_i\right)$, where n_i are the array sizes. Note that in our scenario we do not have *physically* sorted arrays, but the value with any given rank in a given array (i.e., selected range) can be returned in merely $O(\log L)$ time, thanks to the wavelet tree associated with each array C. Hence the time complexity of the Frederickson and Johnson algorithm can be multiplied by $\log L$ to use it for median finding over our data.

Now we calculate the time to find the median of the union of our ranges. We have $K = O(\log_k r)$, the array sizes (n_i in the Frederickson and Johnson formula) are between r' (mask width) and $r'b_0$, i.e., logarithms of those sizes are always $\Theta(\log r)$. Remembering about the $\log L$ multiplier, we obtain $O(\log_k r * \log r * \log L)$ time complexity per pixel, which implies $O(nm\log_k r * \log r * \log L)$ time for the whole image.

We have $O(nmk * \log_k r * \log L)$ preprocessing time and $O(nm\log_k r * \log r * \log L)$ time for the main phase of the algorithm; what is the optimal k? Obviously, we should set $k = \log r$, which gives $O(nm \log r * (\log r / \log\log r) * \log L) = O(nm * \log L * \log^2 r / \log\log r)$ total time in the worst case, or $O(\log L * \log^2 r / \log\log r)$ time per pixel, and it ends the analysis.

Now, if we compare this result to the Gil and Werman one, which is $O(\log^2 r)$, we conclude that our algorithm is better if $\log L = o(\log\log r)$, and not worse if $L = \Theta(\log\log r)$. Unfortunately, in the former case the $O(L)$-time algorithm of Perreault and Hébert is a clear winner. The case of $\log L = \Theta(\log\log r)$ needs a more careful consideration: if $L = \Theta(\log^{2+\varepsilon} r)$, for any constant $\varepsilon > 0$, then our algorithm overcomes the Perreault and Hébert one and matches the complexity of the algorithm of Gil and Werman.

4. Color median filtering

Although there exists no natural ordering of elements in a vector space, the notion of a vector median may be logically introduced with regard to the definition of a scalar median. In formula (1), the differences between scalars may be replaced by distance between vectors, according to a specified (usually Euclidean or city-block) metric. So now:

$$R = \min_{F_i \in W} \sum_{j=0}^{N-1} d\left(F_i, F_j\right),$$

where R is the median (filter's output), N – the number of pixels in the window, and F_i, $i = 0 \ldots N-1$, pixels from W. Let also F_0 be the currently inspected pixel. This formula constitute the *vector median filter* (VMF) by Astola *et al.* [1].

Since that work there have been many ideas prompted to improve the image restoration capabilities of median-like filters for color images. They include directional filters [17] where the angle between vectors of color pixels is taken into account in order to eliminate vectors "atypically" located in the vector (color) space, conditional substitutions of the central pixel [16,8] and many others. A brief survey of existing algorithms can be found in [15]. We mention here yet a speed-oriented idea of mapping 3-dimensional vectors onto a single dimension with space-filling curves [14]. Alas, the algorithm RVMF (reduced vector median filter) from the cited work achieved somewhat worse signal-to-noise ratio compared to the original VMF.

5. Efficient implementation of VMF

The standard implementation of the color median filter requires $O(r^4)$ distance computations per pixel, which is enormously high for all but very small masks. We are not aware of any speed-optimized implementations of those filters, hence we propose our own algorithm for the classic VMF algorithm.

In the standard implementation many distances are computed several times, as the corresponding pixels fall within many different, yet spatially close, masks. The idea that we are going to exploit is to cache some computations. What is cached are the sums of distances from the current pixel (x, y) to all columns of the mask and also the total sum for the mask and the pixel (x, y).

We remind that r is the radius of the mask, hence the square side length is $2r + 1$. We consider scanning the image with the mask separately for each horizontal line. When the pixel (x, y) gets into the sliding mask for the first time during this scan, the sums of distances between (x, y) and the columns $[y - 2r, y]$ are calculated, and additionally the sum of those $2r + 1$ sums is computed. The successive mask is shifted right by one pixel, and now from the previous sum the sum of distances between (x, y) and the column $y - 2r$, which is dropped from the mask, is subtracted, in constant time. Since one column has just been removed, one column must also be added, this is the column $y + 1$, for which the sum of distances is calculated and added to the total sum. This is continued until (x, y) falls outside the mask in the current horizontal scan. Note that with each shift of the mask there are $2r + 1$ pixels that are covered by it for the first time and the described computations pertain to all of them taken individually.

This procedure is applied separately for each line of the image. The number of distance computations per pixel has been decreased to $O(r^3)$.

6. Experimental results

The algorithms were implemented in C++, using MS Visual Studio 2008. The tests were run on three workstations:

- Athlon64 XP 3000+ (1.81 GHz) with 2 GB RAM,
- Intel Core2Duo E5400 (2 × 2.2 GHz) with 2 GB RAM,
- Intel Core2Quad E9400 (4 × 2.67 GHz) with 8 GB RAM.

The first two machines worked under 32-bit Windows XP, while the last one under 64-bit Windows XP.

The test on each machine consisted on filtering three pictures: Lena (512×512 pixels), Baboon (256×256) and Frog (3456×2592). Each picture was filtered for masks of sizes: 3×3, 5×5, 7×7, 9×9, 11×11, 13×13, 15×15 and 17×17. There was one exception – the Athlon machine was too slow to finish tests on Frog in reasonable time. Test results, in seconds, are given in Tables 1–6.

The first three tables compare the baseline ("naïve") implementation against ours ("buffered"). All codes are single-threaded, i.e., make no use of many CPU cores. As it can be seen, for the smallest mask, 3×3, there is no gain from the variant with "buffering". In one case (Athlon64, Frog image) there was even >10% speed loss.

Things change radically when large masks are used. Even for 5×5 we observe speedup from about 1.6x for Frog to about 1.8x on the two other images. When the mask size goes to 13 × 13, the speedup raises to about 4.6–5.1x. Generally, the gains are slightly larger on the Intel CPUs.

As expected, the fastest in those tests was the Core2Quad (C2Q) machine. When the naïve implementation was used, the processing speed of C2Q on the largest image, Frog, reaches ~3.7 MB/s (1 pixel = 3 bytes) with the mask 3×3 and drops to mere 6.4 KB/s (!) with the mask 17×17. In the proposed implementation, those speeds are 3.6 MB/s and 29.5 KB/s, respectively. We assume here that 1 KB = 10^3 bytes and 1 MB = 10^6 bytes. On Lena, which is a much smaller image and, once read, can easily fit in the L2 cache of modern CPUs, those speeds were higher: 4.9 MB/s and 43.6 KB/s, respectively, for the proposed implementation.

Table 1
Filtering time with different masks, test machine: Athlon64 3000+

Image	Filter	Mask size							
		3	5	7	9	11	13	15	17
Frog	naïve	11.55	61.55	221.66	587.53	1301.64	2576.85	–	–
	buff-ered	12.89	39.05	93.56	194.22	349.75	539.31	–	–
Baboon	naïve	0.08	0.45	1.66	4.11	9.016	16.92	29.75	47.45
	buffered	0.06	0.25	0.64	1.34	2.44	3.66	5.58	7.77
Lena	naïve	0.34	1.97	6.50	16.91	36.92	70.70	123.50	201.11
	buffered	0.28	1.05	2.66	5.45	9.91	15.27	22.80	32.52

Table 2
Filtering time with different masks, test machine: Intel Core2Duo E5400

Image	Filter	Mask size							
		3	5	7	9	11	13	15	17
Frog	naïve	7.30	41.52	154.20	415.73	907.12	1833.24	3963.05	5211.00
	buffered	7.47	25.89	62.81	125.50	221.45	359.61	545.09	912.25
Baboon	naïve	0.06	0.33	1.06	2.83	6.11	11.72	20.45	32.56
	buffered	0.05	0.17	0.42	0.84	1.50	2.41	3.66	5.23
Lena	naïve	0.22	1.20	4.33	11.80	25.52	49.20	86.58	138.98
	buffered	0.20	0.70	1.72	3.48	6.20	10.03	15.23	22.06

Table 3
Filtering time with different masks, test machine: Intel Core2Quad E9400

Image	Filter	Mask size							
		3	5	7	9	11	13	15	17
Frog	naïve	5.81	33.88	126.41	337.52	747.45	1447.65	2552.21	4207.23
	buffered	5.61	19.95	50.61	101.92	180.55	293.11	446.53	648.34
Baboon	naïve	0.05	0.25	0.86	2.34	5.03	9.67	16.72	27.06
	buffered	0.05	0.14	0.34	0.69	1.22	1.98	2.98	4.28
Lena	naïve	0.16	0.98	3.53	9.63	20.97	40.59	70.83	116.20
	buffered	0.16	0.56	1.39	2.84	5.06	8.19	12.47	18.05

The results from the multi-thread implementation (Tabs. 4–6) are no surprise. The one-core Athlon64 was practically indifferent to the number of run threads. The Core2Duo and Core2Quad improve significantly their results up to 2 and 4 threads, respectively, which corresponds to their number of cores. It was also noticed that when the number of threads increased beyond those values, the speed occasionally grew too, by up to 5–6%. Using e.g. 4 threads at C2Q probably means one thread per core (no thread switching across the cores) and then the total time is limited by the "slowest" core, i.e. the one that is hampered by other light (system) tasks run in the background. This may be not the case when the number of threads is greater and the cores that end their work faster are able to intercept some threads previously assigned to the "slower" core. In other words, the workload may be more balanced across the cores when the number of threads is sufficiently large.

Table 4
Filtering time with different masks and different number of threads, test machine: Athlon64 3000+

Image	Threads	Mask size							
		3	5	7	9	11	13	15	17
Frog	1	12.89	39.05	93.56	194.22	349.75	539.31	–	–
	2	12.86	39.03	92.86	196.14	336.81	549.31	–	–
	3	12.84	39.02	92.83	196.78	335.52	541.22	–	–
	4	12.83	38.97	92.74	197.22	335.53	539.23	–	–
Baboon	1	0.06	0.25	0.64	1.34	2.44	3.66	5.58	7.77
	2	0.06	0.27	0.64	1.36	2.36	3.70	5.58	7.89
	3	0.08	0.27	0.63	1.36	2.33	3.67	5.53	7.94
	4	0.08	0.25	0.63	1.39	2.33	3.66	5.52	7.78
Lena	1	0.28	1.05	2.66	5.44	9.91	15.27	22.80	32.52
	2	0.27	1.03	2.64	5.50	9.53	15.45	23.16	32.94
	3	0.27	1.03	2.59	5.50	9.50	15.31	22.92	32.63
	4	0.27	1.00	2.58	5.52	9.52	15.28	22.89	32.56

Table 5
Filtering time with different masks and different number of threads, test machine: Intel Core2Duo E5400

Image	Threads	Mask size							
		3	5	7	9	11	13	15	17
Frog	1	7.469	25.891	62.812	125.498	221.451	359.605	545.086	912.245
	2	4.641	13.578	32.359	64.171	112.904	188.795	279.356	404.948
	3	4.593	13.547	32.156	64.249	112.874	186.982	277.606	405.339
	4	4.672	13.515	32	64.062	112.358	182.638	276.606	402.104
	5	4.594	13.563	32.468	63.89	112.42	183.403	277.997	409.948
	6	4.609	13.531	32.063	63.718	112.421	182.373	275.778	401.635
	8	4.515	13.453	32.046	63.561	111.905	182.326	276.2	401.307
Baboon	1	0.047	0.172	0.422	0.844	1.5	2.406	3.656	5.234
	2	0.047	0.109	0.219	0.454	0.797	1.297	1.953	2.859
	3	0.047	0.125	0.235	0.484	0.812	1.297	2.031	2.828
	4	0.047	0.094	0.25	0.453	0.797	1.281	1.891	2.703
	5	0.047	0.125	0.266	0.5	0.781	1.235	1.86	2.688
	6	0.032	0.109	0.234	0.453	0.781	1.234	1.859	2.657
	8	0.047	0.109	0.234	0.485	0.766	1.25	1.874	2.656
Lena	1	0.204	0.703	1.718	3.484	6.203	10.031	15.234	22.062
	2	0.109	0.39	0.906	1.828	3.204	5.296	8.031	11.562
	3	0.078	0.407	0.907	1.782	3.281	5.125	7.922	11.484
	4	0.125	0.375	0.906	1.781	3.187	5.172	7.781	11.219
	5	0.141	0.406	0.922	1.781	3.156	5.14	7.875	11.187
	6	0.125	0.391	0.89	1.797	3.156	5.172	7.844	11.219
	8	0.125	0.36	0.906	1.797	3.187	5.078	7.782	11.171

Table 6
Filtering time with different masks and different number of threads, test machine: Intel Core2Quad E9400

Image	Threads	Mask size							
		3	5	7	9	11	13	15	17
Frog	1	5.61	19.953	50.609	101.921	180.547	293.11	446.531	648.344
	2	3.203	10.468	25.859	51.485	90.969	147.594	224.875	326.282
	3	2.359	7.25	17.359	34.468	60.719	98.5	150.156	218.172
	4	2	5.782	13.219	25.968	45.75	74.188	116.094	163.547
	5	2.204	6.64	15.719	30.875	54.672	88.156	133.594	195.781
	6	1.985	5.672	13.235	25.953	45.688	81.172	122.719	163.562
	8	2	5.672	13.203	27.922	45.735	73.828	113.906	163.453
	12	1.984	5.782	13.093	25.906	45.594	73.812	113.219	162.657
Baboon	1	0.047	0.141	0.344	0.687	1.219	1.984	2.984	4.281
	2	0.031	0.093	0.188	0.375	0.656	1.062	1.609	2.313
	3	0.031	0.063	0.141	0.266	0.453	0.703	1.078	1.546
	4	0.031	0.047	0.11	0.203	0.343	0.547	0.829	1.172
	5	0.032	0.062	0.125	0.234	0.406	0.641	0.969	1.375
	6	0.032	0.062	0.109	0.203	0.375	0.531	0.843	1.172
	8	0.031	0.047	0.109	0.203	0.343	0.531	0.781	1.125
	12	0.031	0.047	0.109	0.204	0.328	0.516	0.781	1.094
Lena	1	0.156	0.562	1.391	2.843	5.062	8.187	12.469	18.047
	2	0.11	0.297	0.734	1.469	2.61	4.219	6.468	9.344
	3	0.078	0.203	0.5	0.985	1.734	2.813	4.281	6.219
	4	0.063	0.156	0.375	0.75	1.312	2.125	3.235	4.703
	5	0.063	0.172	0.438	0.891	1.563	2.531	3.859	5.594
	6	0.079	0.172	0.375	0.75	1.312	2.125	3.265	4.672
	8	0.062	0.157	0.375	0.75	1.312	2.094	3.203	4.594
	12	0.062	0.172	0.391	0.735	1.282	2.078	3.157	4.531

7. Conclusions

We discussed the classic 2D median filter, both for scalar data (grayscale images) and vector ones (color images). We review existing worst-case optimized algorithms for scalar median filtering and presented our algorithm along these lines, which matches the best results for some relation of the mask radius r and the number of intensity levels L. As our tools were rather different than in known algorithms, we think this achievement has some value in theory.

Later, we presented a simple idea to speed up the classic vector median filter, VMF. Surprisingly, it seems that little effort has been made for speed-optimized color median filters, which it should be of utmost importance as they are significantly slower than their scalar counterparts. We focus on the original idea only [1] and reduced $O(r^4)$ distance computations per mask in the naïve version to $O(r^3)$. Experiments show that for the mask of size 13×13 this version is about five times faster than the naïve one (and of course the speedup grows for even larger masks). We also demonstrated that median filters are easily scalable across many CPU cores (tests included a 2-core and a 4-core machine).

Future work should focus on analysis of other median-like filters from the implementation point. Also we believe other speedup ideas for VMF are possible and we are going to explore them soon.

Acknowledgement

We thank Kimmo Fredriksson for interesting discussion concerning Section 3.

References

[1] Astola J., Haavisto P., Neuvo Y., *Vector median flters*. Proceedings of the IEEE, vol. 78, no. 4, 1990, 678–89.

[2] Blum M., Flyod R. W., Pratt V., Rivest R., Tarjan R., *Time Bounds for Selection*. J. Comput. Syst. Sci., vol. 7, no. 4, 1973, 448–461.

[3] Dor D., Zwick U., *Selecting the Median*. SODA 1995, 28–37.

[4] van Emde Boas P., *Preserving Order in a Forest in Less Than Logarithmic Time and Linear Space*. Inf. Process. Lett., vol. 6, no. 3, 1977, 80–82.

[5] Frederickson G.N., Johnson D.B., *Generalized Selection and Ranking.* (Preliminary Version). STOC 1980, 420–428.

[6] Gagie T., Puglisi S.J., Turpin A., *Range Quantile Queries: Another Virtue of Wavelet Trees*. The Computing Research Repository (CoRR), CoRR abs/0902.0133, v5 (May 21, 2009), available at http://arxiv.org/abs/0903.4726.

[7] Gil J., Werman M., *Computing 2-D Min, Median, and Max Filters*. IEEE Trans. Pattern Anal. Machine Intell., vol. 15, no. 5, 1993, 504–507.

[8] Grabowski S., Bieniecki W., *A two-pass median-like filter for impulse noise removal in multichannel images*. III Konferencja „Komputerowe Systemy Rozpoznawania" KOSYR 2003 (3rd Conf. on Computer Recognition Systems), Miłków k/Karpacza, 2003, 195–200.

[9] Harter R., comp.programming forum, 2004, available at http://coding.derkeiler.com/Archive/General/comp.programming/2004-10/0289.html.

[10] Harter R., comp.programming forum, 2004, available at http://coding.derkeiler.com/Archive/General/comp.programming/2004-10/0664.html.

[11] Huang T., Yang G. J., Tang, G. Y.: *A Fast Two-Dimensional Median Filtering Algorithm*. IEEE Trans. Acoust., Speech, Signal Processing, vol. 27, no. 1, 1979, 13–18.

[12] Navarro G., Mäkinen V., *Compressed full-text indexes*. ACM Comput. Surv., 39(1), 2007.

[13] Perreault, S., Hébert, P., *Median Filtering in Constant Time*. IEEE Trans. Image Processing, vol. 16, no. 9, 2007, 2389–2394.

[14] Regazzoni C. S., Teschioni A., *A New Approach to Vector Median Filtering Based on Space Filling Curves*. IEEE Transactions on Image Processing, vol. 6, no. 7, 1997, 1025–1037.

[15] Smołka B., Chydziński A., *Fast detection and impulsive noise removal in color images*. Real-Time Imaging, vol. 11, no. 5–6, 2005, 389–402.

[16] Smołka B., Szczepański M., Plataniotis K.N., Venetsanopoulos A.N., *New technique of impulse noise suppression in color images*. II Konferencja „Komputerowe Systemy Rozpoznawania" KOSYR 2001 (2nd Conf. on Computer Recognition Systems), Miłków k/Karpacza, 2001, 225–232.

[17] Trahanias P.E., Venetsanapoulos A., *Vector directional filters: A new class of multichannel image processing filters*. IEEE Transactions on Image Processing, vol. 2, no. 4, 1993, 528–534.

[18] Weiss B., *Fast Median and Bilateral Filtering*. ACM Transactions on Graphics (TOG), vol. 25, no. 3, 2006, 519–526.