Two-dimensional median filter algorithm for parallel reconfigurable computers

L. Hayat
M. Fleury
A.F. Clark

Indexing terms: Median filter algorithm, Image processing

Abstract: A fast and flexible median filter algorithm is presented which scales well with window size. It is based on the use of image histograming. Two accumulator arrays are used to determine the median value of a discrete sequence of numbers. Speed-up factors of 3 and 4 are achieved over conventional histograming methods (those using single accumulator arrays). Two approaches have been implemented with optimisation in mind. Worst- and best-case machine performance boundaries are defined. Timings are given for both types of parallel architecture.

1 Introduction

In many image processing tasks, particularly in the vision context, edges are of prime importance since they define the object boundaries from which features and other information can be extracted [18]. Median filtering is one procedure in which edges are preserved to an extent suitable for interpretation. Only straight lines are completely invariant to median filtering, so that subtracting a filtered image from the original will indicate the proximity of corners formed from curved lines [5, 6]. (For a mathematical justification see [15, 23]. Only symmetric filters which include the origin keep straight lines invariant. Lines with a width less than $\lceil P/2 \rceil$ may be removed, with P the width of a square window). No new grey-scale values are added to the image. Moreover, impulsive noise (often known as 'salt and pepper' noise) is removed.

For a median filter, the grey-level value of each pixel from the input image is replaced by the median of grey-level values in a neighbourhood of that pixel. If there is a window of size $P \times P$ with total number of entities equal to N (i.e. $N = P \times P$), then the median is defined to be that member for which $N - 1)/2$ entities are smaller or equal in value and $(N - 1)/2$ entities are larger or equal in value. Generally, windows are selected to be of odd size, if even the median is selected as the mean of the two middle values. In median filtering, sorting the pixel values into order (or, equivalently, searching through the pixel values [20]) is the time-critical stage. Arranging pixels according to their grey levels in ascending or descending order is time-consuming, and scales poorly as N increases: the time complexity is $O(N \log N)$ even with the most efficient sort algorithms. In part owing to this bottleneck, median filters are usually implemented with rather small values of P such as 3, 5 and 7. A number of techniques have been developed in an attempt to improve the speed of median filters [8, 12, 4, 2, 6, 9].

Unlike the conventional histograming method, based on a single accumulator array [5], our approach is based on multiple accumulator arrays. In the present approach, for imagery of 256 grey levels, one accumulator array is composed of 16 bins and the other 256 bins. In the smaller array, pixels are counted in by their four most significant bits (MSBs) of grey-level information, whereas in the larger array pixels are counted just as in the ordinary histograming process. The smaller array is used to determine where to search for the median in the larger array.

Earlier work has used the median of the previous window position as the starting point for the search [8, 12]. This approach, for 256 grey levels, has a worst-case search time of 127 steps, whereas the method presented here has a worst-case search time of 15 steps. Thus the present method will work even when the data are not strongly correlated or are corrupted by noise. The present method is at a disadvantage when internal arrangements make accessing the additional array take longer than updating a variable, but will be at an advantage over the running median method when larger word-length images are used.

It is also possible to use the mean as an estimator of the median but this is only reliable when the data within a window have a Gaussian distribution, when the filter might be used for corner detection [6]. In general, for discrete data, the complexity of histograming methods is $O(U + \Sigma_{i=1}^{n} d_i)$, where U is the set-up time for the first window and the d_i are the steps made for each pixel of an n-line image (ignoring edge effects). A comparative and comprehensive survey of a range of methods including various sorts is given in [14]. It is shown that for discrete, bounded data the quickest method is by histogram, which is just superior to the doubly-linked list, though a possible improvement for tree sort/heap sort methods [17] is given in [9], where a lower bound for the process of $O(n^2 \log P)$ for a square image is demonstrated. Histogram methods are easier to implement, particularly in regard to assembler programs, as there is less overhead in pointer manipula-

IEE Proceedings online no. 19952273

Paper first received 20th February and in revised form 14th August 1995

The authors are with the Image Processing Group, Department of Electronic Systems Engineering, University of Essex, Colchester, UK

tion, than in a linked list representation for tree-based sort methods. One might consider using an implicit tree as a means of reducing the overhead. (The values of an implicit tree are embedded in a linear array without requiring the use of pointers). Separable median filters, which are potentially quicker, only estimate the median; this is because the median filter is nonlinear [5, 21].

2 Median filtering algorithm

Let us call the two accumulator arrays A and B. In A pixels are counted according to their intensity or grey-level values. In B pixels are voted depending on the value of their four MSBs. Note that shifting left or right by four multiplies or divides by 16, respectively. This will usually take four machine cycles, considerably less than when integer arithmetic is used on many processors; for example, the transputer [13] takes 40 cycles for an integer divide. Where single-cycle access to look-up tables is possible then, as an alternative to shifting, one may be used to index the appropriate result. Either an assembly or a high-level language providing binary shift operations, such as C, should be used.

The whole process is completed in three different steps, as follows.

(i) Select a window size P_i, read one pixel and get its grey level. Using this grey level as the index of a bin in A, increment its contents by one. Simultaneously, fetch the four MSB value of the pixel, use this value as the index into B, and increment the contents of the corresponding bin by one. Repeat this sequence for all pixels from the selected window.

(ii) Read bins from B, adding their contents. Stop the addition when the summation equals or exceeds by half, the size of the selected window. Take off the value of the last-read bin, obtain the index of this last-read bin and shift it left by four bits. Use this shifted value as the index of accumulator A. Start reading bins, adding their contents to the previous sum. Stop adding when the summation equals, or exceeds by half, the size of the window. Obtain the index of the last-read bin, take this address as a grey level, and replace the central pixel from the window with this grey level.

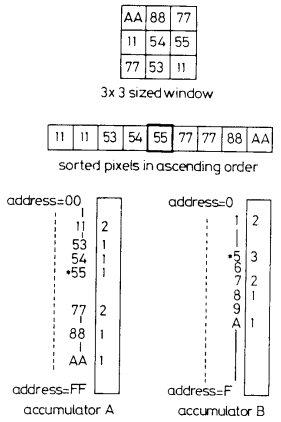

Fig. 1 *Histograming process for median filters*

(iii) Select the next successive overlapped window P_{i+1}, obtaining a column of new pixels from the right side and deleting a column of pixels from the left side. Return to the first step for these pixels, but decrement the arrays for the leftmost values and increment for the rightmost values. Where possible, any windowing operation will make savings from using an overlapping scan [5].

The three steps are repeated from pixel to pixel until the whole image has been processed.

The above three steps are further explained in the following example. Referring to Fig. 1, nine pixels with random grey-level values are taken in a 3×3 window pattern. There are two pixels with grey levels 11, and hence 2 votes are counted in A at location 11. Similarly, 2 votes are counted for pixels with grey level 77 at location 77. As the rest of the grey-level values appear only once, single votes are counted in their corresponding bins. In accumulator array B, 3 votes are counted at location 5 for pixels with grey-level values 53, 54, and 55. Two votes are counted at location 1 for the two pixels with grey-level value 11. Similarly, two votes are counted at location 7 for the two pixels with grey-level values 77. As the rest of the grey-level values appear only once, single votes are allocated to their corresponding bins. In total, there are nine votes for the nine pixels in each array.

In order to find the median, the second condition mentioned earlier is operated firstly on accumulator B and secondly on accumulator A. In B this condition is satisfied at location 5. This guarantees that the median of the given sequences of numbers lies in the range 50 – 5*F* (all numbers are in hexadecimal). This is shown in Fig. 1. Subtracting the last value to be added, 3 in this case, the summation process is switched to A starting from location 50 and terminating at location 55, where the sum is equal to 5, the mid-point of the size of the window pattern. Hence, 55 is the median.

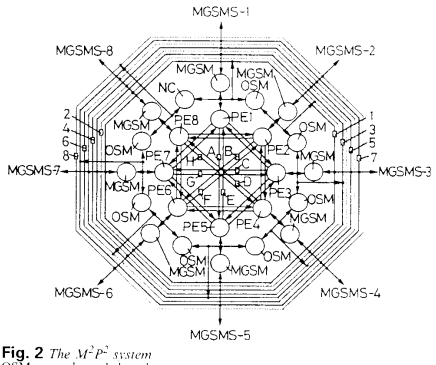

Fig. 2 *The M^2P^2 system*
OSM = overlapped shared memory
NC = not connected
MGSM = multiport global shared memory
MGSMS = multiport global shared memory segment
—1— 8-bit data memory bus PE-1 —A— 8-bit parallel I/O links PE-1
—2— 8-bit data memory bus PE-2 —B— 8-bit parallel I/O links PE-2
—3— 8-bit data memory bus PE-3 —C— 8-bit parallel I/O links PE-3
—4— 8-bit data memory bus PE-4 —D— 8-bit parallel I/O links PE-4
—5— 8-bit data memory bus PE-5 —E— 8-bit parallel I/O links PE-5
—6— 8-bit data memory bus PE-6 —F— 8-bit parallel I/O links PE-6
—7— 8-bit data memory bus PE-7 —G— 8-bit parallel I/O links PE-7
—8— 8-bit data memory bus PE-8 —H— 8-bit parallel I/O links PE-8

With this double-array method, 12 bins are summed up when accumulation begins from lower-addressed

bins and proceeds towards higher-addressed bins. On the other hand, using a single accumulator array method, 56 bins are needed to meet the second condition, providing the accumulation process begins from location 0 and proceeds towards location 255. This shows that the approach presented here significantly contributes towards the overall performance in comparison to a single accumulator array. An ideal-case performance could be achieved if an arrangement is made to read only relevant bins, rather than reading bins from one end until the summation exceeds half the size of the selected window [4], but this is possible only with dedicated hardware. If the dynamic range of grey-levels is larger, then there is no reason to stop at two arrays; thus the method is defined recursively.

3 Performance results

The new algorithm has been implemented on two parallel architectures, one using overlapped shared memory and the other using a localised address space. Both implementations use reconfigurable techniques [7].

3.1 Implementation using M^2P^2

The M^2P^2 system [11] is a dynamically reconfigurable, multi-dimensional, multi-directional, parallel, pipelined, image processing system (Fig. 2). The processors used in this system are Texas Instruments TMS320c25 digital signal processors. The architecture of this system retains the merits of both shared and distributed memory systems. Processing elements (PEs) are organised as a linear chain. Three types of communication are provided between the various processors: inter-processor communication between two adjacent processors is provided via overlapped shared memory; communication between non-adjacent processors is provided by global shared memory as well as through eight-bit wide parallel I/O links. Because of the provision of multiple communication paths, it can be dynamically reconfigured under software control, allowing the eight PEs to be organised in differently-sized, multichain stages from one PE per stage to four PEs per stage. Different stages may be configured with different number of PEs, depending on the task complexity and the desired level of computation. Furthermore, the eight PEs can be organised into other interconnection topologies such as bidirectional ring, hypercube and tree structures [10].

For comparative evaluation, both single- and double-accumulator array schemes were implemented on the M^2P^2 system. Based on experimental results, a double accumulator approach has shown a significant improvement over the single accumulator array. The

Table 1: Average execution timings for the new median filter algorithm and sort method on M^2P^2

Processors vs. execution times, s								
PEs	Image size	Pixel value	Med3×3	Med5×5	Med7×7	Sort3×3	Sort5×5	Sort7×7
1	128×64	0	0.3016	0.4263	0.5508	1.02	5.31	19.07
2			0.1509	0.2132	0.2757	0.5108	2.667	9.537
4			0.0755	0.1068	0.1380	0.2556	1.335	4.769
8			0.0738	0.0536	0.0696	0.1281	0.667	2.386
1		127	0.590	0.7144	0.835	1.02	5.31	19.07
2			0.2951	0.3573	0.4199	0.5108	2.667	9.537
4			0.1476	0.1789	0.2101	0.2556	1.335	4.769
8			0.0739	0.0896	0.1052	0.1281	0.667	2.386
1		256	0.694	0.8194	0.9436	1.02	5.31	19.07
2			0.347	0.4099	0.4720	0.5108	2.667	9.537
4			0.1738	0.2049	0.2361	0.2556	1.335	4.769
8			0.0871	0.1025	0.1184	0.1281	0.667	2.386
1		real image	0.4914	0.614	0.7425	1.02	5.31	19.07
2			0.2440	0.3065	0.3690	0.5108	2.667	9.537
4			0.1224	0.1538	0.1853	0.2556	1.335	4.769
8			0.0606	0.0768	0.0932	0.1281	0.667	2.386
Performance: processors vs. image size, s								
1	128×64	0	0.3016	0.4263	0.5508	1.02	5.31	19.072
2	128×128		0.3018	0.4263	0.5508	1.021	5.311	19.076
4	128×256		0.3019	0.4264	0.5510	1.023	5.314	19.079
8	256×256		0.3020	0.4266	0.5512	1.024	5.316	19.083
1	128×64	127	0.590	0.7144	0.835	1.02	5.31	19.072
2	128×128		0.5902	0.7145	0.835	1.021	5.311	19.076
4	128×256		0.5904	0.7147	0.8351	1.023	5.314	19.079
8	256×256		0.5905	0.7149	0.8353	1.024	5.316	19.083
1	128×64	257	0.694	0.8194	0.9436	1.02	5.31	19.072
2	128×128		0.6941	0.8195	0.9437	1.021	5.311	19.076
4	128×256		0.6942	0.8197	0.9439	1.023	5.314	19.079
8	256×256		0.6944	0.8198	0.9440	1.024	5.316	19.083
1	128×64	real image	0.4914	0.614	0.7425	1.02	5.31	19.072
2	128×128		0.4916	0.614	0.7426	1.021	5.311	19.076
4	128×256		0.4917	0.6143	0.7427	1.023	5.314	19.079
8	256×256		0.4919	0.6146	0.7430	1.024	5.316	19.083

programs were coded in assembly language. Execution timings measured for different sized windows using single accumulator array and processing real images were 1.66, 1.76, and 1.84s, whereas the execution timings measured for double accumulator arrays and processing the same real images were 0.491, 0.614, and 0.742s, respectively.

This approach was further studied on three different sample images, with all pixels having the same grey-level value, i.e. 0, 127 and 255. In each case, the accumulation process was started from lower-addressed bins and proceeded to higher-addressed bins. The worst performance occurred when the accumulation process terminated after going through all bins, 16 from accumulator array B and the corresponding 16 bins from A; this is the case when the number of pixels in a window with grey-level value 255 is more than half the size of the window. The simplest case was for an image in which all the pixels had value 0; this gave the minimum possible execution time. Average performance was achieved when all the pixels had grey level 127, resulting in maximum votes in the middle bins from both accumulator arrays. Therefore, for real images where multiple grey-level values are possible, the execution time must lie within the range of simplest- to worst-case performance.

Execution timings for these sample images with constant grey level (as well as a digitally-scanned calligraphic image [10]) are tabulated in Table 1. For comparison, a sorting process was also tested over the same sample images for all three differently-sized windows. A type of bubble sort was used, stopping when the mid-way value had been reached. In each case for the same size window, constant execution timings were observed. From these timings it is clear that the efficiency of this new approach is much better than the efficiency of its counterpart using sorting.

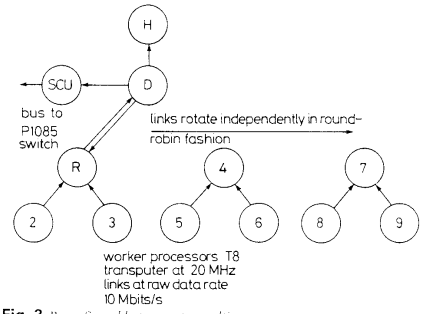

Fig. 3 *Reconfigurable transputer multi-tree*
H = host running Idris
D = data farmer reconfigurable manager
R = router and filter

3.2 Implementation using the Parsys supernode

The Parsys supernode [19] used for this work is a 24-transputer multi-user machine which can reconfigured by use of a programmable cross-bar switch (i.e. all-to-all switch) the P1085. Using customised software it is possible to achieve run-time reconfiguration which to some extent overcomes the limitations of a fixed-valency machine [7]. (A machine in which the number of communication connections between processors is limited to a fixed number or valency). Only ten of the processors were used to perform the median filter. A further transputer acted as host running Idris, a Unix-like operating system, and link rotation was performed by means of calls to the switch control unit (SCU) (see Fig. 3). Of the ten processors, nine were employed in active computation whilst the remaining processor acted as a data farmer.

The images were divided into strips, including border information, which were distributed to the processors using a load-balancing algorithm. An extra zeroed border was added to each image to cope with the boundary problem. On return, the image was reconstructed using a tagged bin sort. (A bin sort, which has time complexity $O(n)$, is only possible when the item keys form a known consecutive sequence. A tagged sort avoids data movements of the image strips by sorting only on strip identifiers or tags held in a reference array). 3L Parallel C was used to code the program [1], making use of library calls to implement the message-passing model of parallelism. Semaphores [3] were used to avoid excessive internal data movements. Inner loops were expanded to enhance performance (the advantage being that the branch test at each loop iteration is avoided). It may be profitable to use a look-up table for the 16-times table, which is needed to translate between the two arrays.

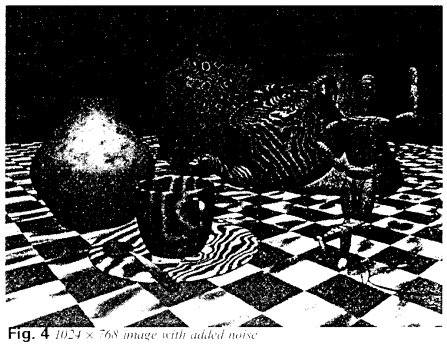

Fig. 4 *1024 × 768 image with added noise*

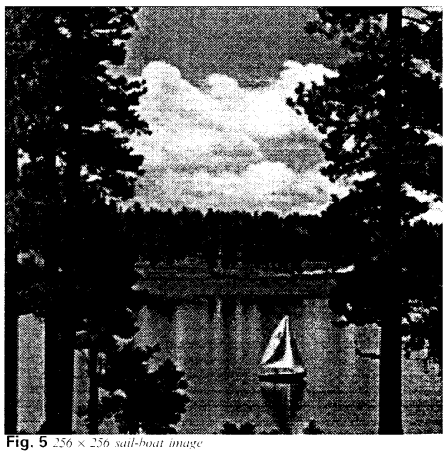

Fig. 5 *256 × 256 sail-boat image*

Timings for two images (Fig. 4 and the 'sail-boat' image [24] of Fig. 5) are given in Table 2. The resolution of the times is 15 625/s, which is the rate of the transputer low-priority clock. The timings were taken on the farmer at the completion of all processing, but they exclude input/output times. As in the case of the M^2P^2 results, best-case, average-case and worst-case figures are given by testing on images in which all the pixels had one particular value, effectively defining the performance boundaries. Our worst-case timing, using a 7×7 mask on a 256×256 test image, is 1.67s, which we believe is a significant improvement on comparable implementations.

Table 2: Execution timings for the new median filter algorithm on the Parsys supernode

Window size	Image size	Pixel value	Time, s
3×3	256×256	real image	0.87
5×5			1.06
7×7			1.27
3×3	1024×768	real image	8.3
5×5			10.66
7×7			13.23
3×3	256×256	0	0.58
		127	1.04
		255	1.20
5×5		0	0.79
		127	1.26
		255	1.46
7×7		0	0.99
		127	1.49
		255	1.67
3×3	1024×768	0	5.49
		127	11.93
		255	14.00
5×5		0	7.76
		127	14.17
		255	16.12
7×7		0	10.20
		127	16.61
		255	18.52

In [16] the performance of a 7×7 window is given for 16 transputers on a Telmat T.Node machine, which is very similar to the Parsys Supernode. Speed-up for the median filter using 16 transputers was 12.84; with 9 processors, it was 7.7, indicating that the performance had not saturated. The time for a 256×256 image was still 5.211s, excluding input/output. The difference, apart from the algorithm, may be due to the use of a mesh topology or the use of the Helios operating system, which puts a kernel on each machine. The authors also remark that Shell sorting was used, 'which is slow (on purpose)'. (In fact, the Shell sort has better worst-case performance than algorithms such as quick-sort [20]).

In one test we used a computer-generated image, to which 'salt' noise was artificially added. A 3×3 window was large enough to remove this noise. A larger filter might introduce extraneous pixels into the range of the offending noise: larger masks remove streaks of larger width but also remove fine detail lines.

4 Conclusion

The median filter is an important image-processing technique. A significant enhancement of existing median-filtering algorithms has been presented: this uses a multi-binned histogram method. To be able to perform fast median filtering, parallel computers are an obvious route. Hence, implementations on two classes of parallel-processing system have been described and evaluated. The timings for the M^2P^2 machine represent optimal timings, though from an exotic architecture, which is nevertheless relevant to this type of application. The timings for the Supernode represent optimal timings for a more accessible form of modular parallelism. In both cases, it was found necessary to tweak the hardware by the use of reconfiguration to improve performance. In addition, software techniques, such as the use of semaphores, tagged bin sorts, look-up tables and loop unrolling, contributed to the performance. A feature of our testing method for this data-dependent technique is the use of best- and worst-case results.

The algorithm presented gives creditable performance on both systems, though it is apparent that the use of DSPs confers an edge in performance over transputers, even though the hardware multiplier of the TMSC20c25 has not been used in this implementation [22]. The algorithm could be extended for larger word-length images: the 10-, 12- and 16-bit data commonly acquired from some sensors could be accommodated by using more than two arrays. For instance, if 12-bit data are used with 3 accumulator arrays, a maximum search path of 48 steps is needed, whereas the worst case for a running median would be 2048 steps.

5 References

1 3L Ltd, Peel House, Ladywell, Livingston, Scotland: 'Parallel C Version 2.2.2', 1991
2 AHMAD, M.O., and SUNDARARAJAN, D.: 'A fast algorithm for two-dimensional median filtering', *IEEE Trans. Circuits Syst.*, 1987, **34**, pp. 1364–1373
3 ANDREWS, G.R.: 'Paradigms for process interaction in distributed programs', *ACM Comput. Surveys*, 1991, **23**, pp. 49–90
4 ATAMAN, E., AATRE, V.K., and WONG, K.M.: 'A fast method for real-time median filtering', *IEEE Trans. Acoust. Speech Signal Process.*, 1980, **28**, pp. 415–421
5 DAVIES, E.R.: 'Machine vision: theory, algorithms and practicalities' (Academic Press, 1990)
6 DAVIES, E.R.: 'Simple fast median filtering algorithm, with application to corner detection', *Electron. Lett.*, 1992, **28**, pp. 199–201
7 FLEURY, M., and HAYAT, L.: 'Performance on a reconfigurable message passing parallel processor using the data farming paradigm', Technical report, Essex University, 1993
8 GARIBOTTO, G., and LAMBARELLI, L.: 'Fast on-line implementation of two-dimensional median filtering', *Electron. Lett.*, 1979, **15**, pp. 24–25
9 GIL, J., and WERMAN, M.: 'Computing 2-d min, median, and max filters', *IEEE Trans. Pattern Anal. Mach. Intell.*, 1993, **15**, pp. 504–507
10 HAYAT, L.: 'A dynamically reconfigurable parallel pipelined system for real-time image processing applications', PhD thesis, London University, 1992
11 HAYAT, L., and SANDLER, M.B.: 'An efficient multidimensional/multidirectional parallel pipelined architecture for image processing', *IEE International Conference on Digital Signal Processing*, UK, 1991, pp. 105–110
12 HUANG, T.S., YANG, G.J., and YANG, G.Y.: 'A fast two-dimensional median filtering algorithm', *IEEE Trans. Acoust. Speech Signal Process.*, 1979, **27**, pp. 13–18
13 INMOS LTD, 1000 Aztec West, Almondesbury, Bristol: 'The transputer databook', 1989
14 JUHOLA, M., KATAJAINEN, J., and RAITA, T.: 'Comparison of algorithms for standard median filtering', *IEEE Trans. Acoust. Speech Signal Process.*, 1991, **39**, pp. 204–208
15 JUSTUSSON, B.I.: 'Median filtering: statistical properties', in HUANG, T.S. (Ed.): 'Two-dimensional digital signal processing: transforms and median filters' (Springer, 1981)

16 KILINDRIS, T., and PITAS, I.: 'PARALLEL EIKONA: A parallel digital image processing package', in PITAS, I. (Ed.): 'Parallel algorithms' (J. Wiley and Sons, 1993), pp. 341–352
17 KNUTH, D.E.: 'The art of computer programming volume III: sorting and searching' (Addison-Wesley, 1974)
18 MARR, D., and HILDRETH, E.: 'Theory of edge detection', *Proc. Roy. Soc. London B*, 1980, pp. 187–217
19 PARSYS LTD, Boundary House, Boston Road, London, 'Hardware reference manual for the Parsys SN1000 series', 1989
20 PRESS, W.H., FLANNERY, B.P., TEUKOLSKY, S.A., and VETTERLING, W.T.: 'Numerical recipes in C: the art of scientific computing' (Cambridge University Press, 1992)
21 RANKA, S., and SAHNI, S.: 'Efficient serial and parallel algorithms for median filtering', *IEEE Trans. Signal Process.*, 1991, **39**, pp. 1462–1466
22 SANDLER, M.B., HAYAT, L., and COSTA, L.D.F.: 'Benchmarking processors for image processing', *Microprocess. Microsyst.*, 1990, **14**, pp. 583–588
23 TYAN, S.G.: 'Median filtering: deterministic properties', in HUANG, T.S. (Ed.): 'Two-dimensional digital signal processing transforms and median filters' (Springer, 1981)
24 WEBER, A.G.: 'Image data base', Technical report 101, University of Southern California Institute of Signal and Image Processing, Powell Hall 306, Los Angeles CA 90089-0272, 1986