Median Filtering in Constant Time

Simon Perreault* and Patrick Hébert, *IEEE member*

***Abstract*— The median filter is one of the basic building blocks in many image processing situations. However, its use has long been hampered by its algorithmic complexity of $O(r)$ in the kernel radius. With the trend toward larger images and proportionally larger filter kernels, the need for a more efficient median filtering algorithm becomes pressing. In this correspondence, a new, simple yet much faster algorithm exhibiting $O(1)$ runtime complexity is described and analyzed. It is compared and benchmarked against previous algorithms. Extensions to higher-dimensional or higher-precision data and an approximation to a circular kernel are presented as well.**

I. Introduction

THE median filter [1] is a canonical image processing operation, best known for its salt and pepper noise removal aptitude. It is also the foundation upon which more advanced image filters like unsharp masking, rank-order processing, and morphological operations are built [2]. Higher-level applications include object segmentation, recognition of speech and writing, and medical imaging. Figure 4 shows an example of its application on a high-resolution picture.

However, the usefulness of the median filter has long been limited by the processing time it requires. Its nonlinearity and non separability make it unsuited for common optimization techniques. A brute-force approach simply builds a list of the pixel values in the filter kernel and sorts them. The median is then the value situated at the middle of the list. In the general case, this algorithm's per-pixel complexity is $O(r^2 \log r)$, where r is the kernel radius. When the number of possible pixel values is a constant, as is the case for 8-bit images, one can use a bucket sort, which brings the complexity down to $O(r^2)$. This is still unworkable for any but the smallest kernels.

The classic algorithm [3], used in virtually all publicly available implementations, exhibits $O(r)$ complexity (see Algorithm 1). It makes use of a histogram for accumulating pixels in the kernel. Only a part of it is modified when moving from one pixel to another. As illustrated in Figure 1, $2r+1$ additions and $2r+1$ subtractions need to be carried out for updating the histogram. Computing the median from a histogram is done in constant time by summing the values from one end and stopping when the sum reaches $(2r+1)^2/2$. For 8-bit images, a histogram is made of 256 bins and therefore 128 comparisons and 127 additions will be needed on average. Note that any other rank-order statistic can be computed in the same way by changing the stopping value.

Efforts were made to improve the complexity of the median filter beyond linear. A complexity of $O(\log^2 r)$ was attained by Gil et al. [4] using a tree-based algorithm. In the same paper they claimed a $O(\log r)$ lower bound for any 2-D median filter algorithm. Our work is most similar to that of [5], where sorted lists were used instead of histograms, which resulted in a $O(r^2)$ complexity and was relatively slow. More recently, Weiss [6] developed a method whose runtime is $O(\log r)$ using a hierarchy of histograms. In his approach, even though *complexity* has been lowered, *simplicity* has been lost. We strive for a simple and efficient algorithm, with applicability to both CPU and custom hardware.

The authors are with the Computer Vision and Systems Lab, Université Laval, Québec, Canada. G1K 7P4. Phone: (418) 656-2131 #4479. Fax: (418) 656-3594. E-mail: {perreaul,hebert}@gel.ulaval.ca

EDICS: FLT-MRPH (Rank and morphological filtering techniques)

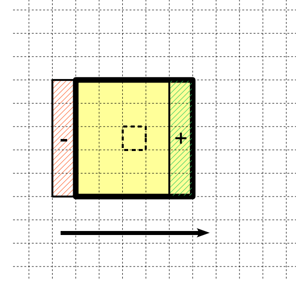

Fig. 1: In Huang's $O(n)$ algorithm, $2r+1$ pixels must be added to and subtracted from the kernel's histogram when moving from one pixel to the next. In this figure, $r = 2$.

Algorithm 1 Huang's $O(n)$ median filtering algorithm.

Input: Image X of size $m \times n$, kernel radius r
Output: Image Y of the same size as X

```
Initialize kernel histogram H
for i = 1 to m do
  for j = 1 to n do
    for k = -r to r do
      Remove X_{i+k, j-r-1} from H
      Add X_{i+k, j+r} to H
    end for
    Y_{i,j} ← median(H)
  end for
end for
```

In this correspondence we propose a simple $O(1)$ median filtering algorithm similar in spirit to Huang's. We show a few straightforward optimizations which enable it to become much faster than the classic algorithm. We take the opportunity to examine why the Gil-Werman lower bound of $O(\log r)$ does not seem to hold. Then we explore extensions to the new filter, namely application to higher-precision or higher-dimensional data as well as a circular kernel approximation. Finally, timing results are shown, asserting the practicality of our approach.

II. From $O(r)$ to $O(1)$ Complexity

To best understand our approach, it is helpful to first point out the inefficiency in Huang's algorithm. Specifically, notice that no information is retained between rows. Each pixel will need to be added and removed to $2r+1$ histograms over the course of processing the whole image, which causes the $O(r)$ complexity. Intuitively, we can guess that we will need to accumulate each pixel at most a constant number of times to obtain $O(1)$ complexity. As we will see, this becomes possible when information is retained between rows.

Let us introduce one property of histograms, that of distributivity [6]. For disjoint regions A and B,

$$H(A \cup B) = H(A) + H(B).$$

Notice that summing histograms is a $O(1)$ operation with respect to the number of accumulated pixels. It depends only on the size of the histogram, which is itself a function of the bit depth of the

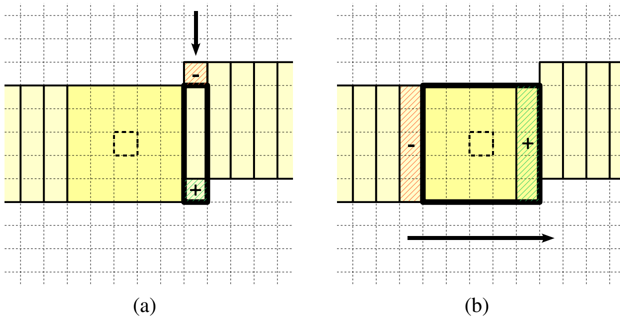

Fig. 2: The two steps of the proposed algorithm. (a) The column histogram to the right is moved down one row by adding one pixel and subtracting another. (b) The kernel histogram is updated by adding the modified column histogram and subtracting the leftmost one.

image. Having made this observation, we can move on to a new $O(1)$ algorithm.

The proposed algorithm maintains one histogram for each column in the image. This set of histograms is preserved across rows for the entirety of the process. Each column histogram accumulates $2r + 1$ adjacent pixels and is initially centered on the first row of the image. The kernel histogram is computed by summing $2r + 1$ adjacent column histograms. What we have done is break up the kernel histogram into the union of its columns, each of which maintains its own histogram. While filtering the image, all histograms can be kept up to date in constant time with a two-step approach.

Consider the case of moving to the right from one pixel to the next. The column histograms to the right of the kernel are yet to be processed for the current row, so they are centered one row above. The first step consists of updating the column histogram to the right of the kernel by subtracting its topmost pixel and adding one new pixel below it (Figure 2a). The effect of this is lowering the column histogram by one row. This first step is clearly $O(1)$ since only one addition and one subtraction, independent of the filter radius, need to be carried out.

The second step moves the kernel histogram, which is the sum of $2r+1$ column histograms, one pixel to the right. This is accomplished by subtracting its leftmost column histogram and adding the column histogram lowered in the first step (Figure 2b). This second step is also $O(1)$. As stated earlier, adding, subtracting, and computing the median of histograms comprise a number of operations depending on the image bit depth, not on the filter radius.

The net effect is that the kernel histogram moves to the right while the column histograms move downward. We visualize the kernel as a zipper slider bringing down the zipper side represented by the column histograms. Each pixel is visited only once and is added to only a single histogram. The last step for each pixel is computing the median. As stated earlier, this is $O(1)$ thanks to the histogram.

All of the per-pixel operations (updating both the column and kernel histograms as well as computing the median) are $O(1)$. Now we address the issue of initialization, which consists of accumulating the first r rows in the column histograms and computing the kernel histogram from the first r column histograms. This results in an $O(r)$ initialization. In addition, there is overhead when moving from one row to another which accounts for another $O(r)$ term. However, since the $O(r)$ initialization only occurs once per row, the cost per pixel is insignificant for arbitrarily large images. In particular, the *amortized* cost drops to $O(1)$ per pixel when the dimensions of the image are proportional to the kernel radius, or if the image is processed in tiles of dimensions $O(r)$. When the tile size is limited by the dimensions of the image, the redundancy of information outside the image (e.g. solid color, or repeated edge pixels) correspondingly simplifies the initialization, allowing $O(1)$ computation on any size image, for any kernel radius.

Algorithm 2 The proposed $O(1)$ median filtering algorithm.

Input: Image X of size $m \times n$, kernel radius r
Output: Image Y of the same size
  Initialize kernel histogram H and column histograms $h_{1\dots n}$
  for $i = 1$ to m **do**
    for $j = 1$ to n **do**
      Remove $X_{i-r-1,j+r}$ from h_{j+r}
      Add $X_{i+r,j+r}$ to h_{j+r}
      $H \leftarrow H + h_{j+r} - h_{j-r-1}$
      $Y_{i,j} \leftarrow \text{median}(H)$
    end for
  end for

To summarize, the operation rundown for an 8-bit grayscale pixel is as follows:

- 1 addition for adding the new pixel to the column histogram to the right of the kernel.
- 1 subtraction for removing the old pixel from that same column histogram.
- 256 additions for adding the new column histogram to the kernel histogram.
- 256 subtractions for subtracting the old column histogram from the kernel histogram.
- 128 comparisons and 127 additions, on average, for finding the median of the kernel histogram.

This may seem excessive. However, most of these operations are naturally vectorizable, which lowers the time constant considerably. More importantly, many optimizations can be applied to reduce the number of operations. They are discussed in the next section.

III. Implementation Notes

This section describes some optimizations that can be applied to increase the efficiency of the proposed algorithm. They all depend on the particular CPU architecture on which the filter is executed. As such, their effect can vary greatly (even reducing the efficiency in some cases) from one kind of processor to another. Note also that optimizations of sections III-C and III-D are data-dependent.

A. Vectorization

Modern processors provide SIMD instructions that can be exploited to speed up our algorithm. The operation rundown shows that most of the time is spent in adding and subtracting histograms. This can be sped up considerably with MMX, SSE2 or Altivec instruction sets by processing multiple bins in parallel. To maximize the number of histogram bins that we can add in one instruction, each bin is represented with only 16 bits. Thus, the kernel size is limited to 2^{16} pixels, which is acceptable for typical uses. This limit is not intrinsic to our algorithm: it is only a means for optimization.

Another area where parallelism can be exploited is the reading of pixels from the image and their accumulating in column histograms. Instead of alternating between updating column and kernel histograms, as described in Section II, we can process the column histograms for a whole row of pixels first. Using SIMD instructions, we can update multiple column histograms in parallel. We then proceed with the kernel histogram as usual.

B. Cache Friendliness

The constant-time median filtering algorithm needs to keep in memory one histogram for each column. For the whole image, this may easily amount to hundreds of kilobytes, often exceeding the cache size of today's processors. This leads to inefficient repeated access to the main memory and negates the usefulness of the cache. One way to circumvent this limitation is to split the image in vertical stripes that are processed independently. The width of each stripe is chosen to be such that the histograms fill up the cache but do not exceed its capacity. One disadvantage of this modification is that it amplifies the border effects. In practice, it usually causes a huge decrease in processing time. Note that simultaneously processing stripes on different processors is an easy way to parallelize the proposed algorithm.

C. Multilevel Histograms

Multilevel histograms have been analyzed in [7] and shown to be a very effective optimization. The idea is to maintain a parallel set of smaller histograms accumulating only the higher order bits of pixels. For example, it is common to use two tier histograms for 8-bit images, where the higher tier is 4-bit wide while the lower tier contains the full 8-bit data. It is customary to name them the *coarse* and *fine* levels, respectively. The coarse level would contain 16 bins (2^4) and each one would be the sum of its corresponding 16-element segment of the fine level.

There are two advantages to multilevel histograms, the first being the acceleration of the computation of the median. Instead of examining the entire 256 bins, we can now make 16-element hops by finding the median at the coarse level. This gives us the segment of the fine histogram that contains the median. Instead of an average of 128 additions and comparisons, we now only need 16 (8 at each level) to reach the median. The second advantage is related to addition and subtraction of histograms. One can skip a 16-element segment of the fine histogram when its corresponding coarse value is zero. When r is small, column histograms are sparsely populated and so the added branching may be worthwhile.

D. Conditional Updating of the Kernel

The separation of histograms in coarse and fine levels enables a slightly less obvious but very effective optimization. Notice that up to this point, most of the processing time was spent in adding and subtracting column histograms to and from the kernel histogram. With conditional updating, this time is lowered by keeping up to date only the kernel histogram's coarse level while its fine level is updated on-demand.

Recall that computation of the median is done by first scanning at the coarse level, which indicates the 16-element segment of the fine level that contains the median. Since a column histogram contributes to at most $2r + 1$ computations of the kernel's median, at most $2r + 1$ of its fine level segments will ever be useful. When pixel values vary smoothly in the image, the actual figure is much lower because the same segment is accessed repetitively. Updating the kernel histogram's fine level with segments that will never be used can be skipped.

To do so, we need to maintain a list of the column index at which each segment was last updated. When moving from one pixel to the next, we update both levels of the new column histogram but only the coarse level of the kernel histogram. Next, we compute the kernel histogram's median at the coarse level and determine in which segment of the fine level the median resides. We then bring that segment up to date by processing column histograms starting from its last updated column. If that column is offset by more than $2r + 1$ pixels from the current one, then there is no overlap between the kernel at the old location and the current one. We therefore update the histogram segment from scratch, skipping columns in the process. It is in that case, by skipping columns, that we make up in reduced processing time for the additional branching and bookkeeping.

It is also advantageous to interleave the layout of histograms in memory so that segments of adjacent columns are also adjacent in memory. That is, histogram bins should be arranged first by segment index, then by column index, and finally by bin index. That way, updating a segment of the kernel's fine level corresponds to summing a contiguous block of memory.

IV. Refutation of the Gil-Werman Lower Bound

A theoretical lower bound of $\Omega(\log r)$ for the complexity of the 2-D median filter was introduced in [4]. They state that "*any algorithm for computing the r-sized median filter for an $n \times n$ input with $n \geq (3r-1)/2$ runs in $\Omega(\log r)$ amortized time per element.*" This seems to be in direct contradiction with our findings: we have proven that our algorithm is in $O(1)$ and $\Omega(\log r) \cap O(1) = \emptyset$ by definition.

Although their reasoning is correct, it is based on reduction from sorting. They show that the 2-D median filter has the power of sorting arbitrary input. They then argue that since the output has been sorted, the runtime must have been $\Omega(\log r)$ per element. This is true *as long as one uses a comparison sort algorithm*. This is avoidable when the number of possible signal values is a constant.

It is well known that non-comparison sort algorithms, of which bucket and radix sort are examples, are not subjected to this lower bound [8]. The histogramming process our algorithm makes use of is analogous to sorting data with a non-comparison sort. The counterexample of a median filter exhibiting $O(1)$ runtime per element disproves the $\Omega(\log r)$ lower bound. It is also readily recognized as the true lower bound since per-element processing time cannot be lower than constant. It would nevertheless be possible to diminish this constant with new optimization ideas.

V. Extensions

The proposed algorithm can be extended to new situations. We explore some of the more common ones in this section.

A. Higher Precision

Images having a bit depth other than 8 bits can be processed just as easily by our algorithm. A single change needs to be made: the number of histogram buckets must be scaled accordingly. This implies that histogram addition and subtraction as well as the search for the median will take accordingly more time. It would be useful at some point to make use of three-tier (or more) histograms as their size increases.

Larger histograms will also occupy more space in memory, which will result in smaller stripes (see Section III-B). If this becomes a problem, the ordinal transform [6] could be of some use. However, as the size of histograms scales in $O(2^b)$, where b represents the image bit depth, high bit depths are a fundamental weakness of any histogram-based algorithm.

B. Higher Dimensionality

Median filtering data in more than two dimensions is common in fields like medical imaging [9], [10] and video processing [11]. The proposed algorithm can handle N-dimensional data in $O(1)$ runtime complexity at the cost of increased memory usage. As an example, Algorithm 3 shows how 3-D median filtering is accomplished. Extending this to higher dimensions is straightforward.

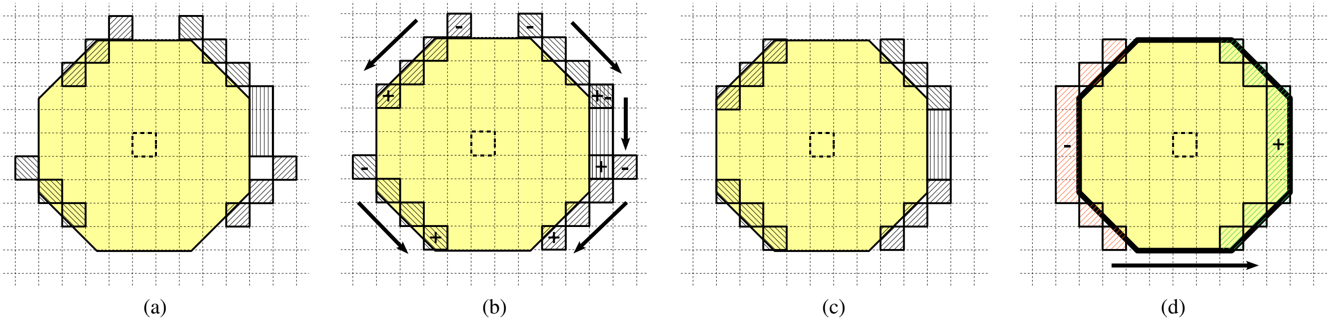

Fig. 3: Movement of diagonal and column histograms in the circular kernel approximated by an octagon. (a) Layout of the five side histograms one row above the current one. (b) Histograms being lowered to the current row. (c) Position of histograms when centered on the current row. (d) Octagonal kernel moving horizontally.

Algorithm 3 Median filtering in constant time for 3-D data.

Input: Image X of size $m \times n \times o$, kernel radius r
Output: Image Y of the same size
  Initialize kernel histogram H, planar histograms $h^1_{1\ldots o}$ and column histograms $h^2_{1\ldots n,1\ldots o}$
  for $i = 1$ to m **do**
    for $j = 1$ to n **do**
      for $k = 1$ to o **do**
        Remove $X_{i-r-1,j+r,k+r}$ from $h^2_{j+r,k+r}$
        Add $X_{i+r,j+r,k+r}$ to $h^2_{j+r,k+r}$
        $h^1_{k+r} \leftarrow h^1_{k+r} + h^2_{j+r,k+r} - h^2_{j+r,k-r-1}$
        $H \leftarrow H + h^1_{k+r} - h^1_{k-r-1}$
        $Y_{i,j,k} \leftarrow \text{median}(H)$
      end for
    end for
  end for

Each new dimension requires a new set of histograms whose size is the product of the sizes of lower-order dimensions. This exponential scaling will quickly make it impractical for higher dimensions. However, it would always be possible to process the data in hyper-stripes (see Section III-B), which would let one impose an arbitrary limit on memory usage at the cost of increasing the importance of the linear terms caused by border effects. As for the runtime, one can see that the inner loop is still $O(1)$ in the kernel size. It scales with the number of dimensions as $O(N)$, as each added dimension requires one more histogram summation. It is interesting to notice that a single new pixel from the source image is accessed for each pixel of the destination image to be computed.

C. *Circular Kernel Approximation*

A popular extension to many filters is an approximately circular kernel. Such a kernel shape is closer to the theoretical perfectly circular kernel as defined on continuous data and minimizes the artifacts caused by a square kernel. The proposed algorithm can be extended to an octagonal kernel, as shown in Figure 3. Five histograms, each one corresponding to one side of the octagon, are used instead of a single column histogram in the square kernel case. They can be preserved across rows in much the same way. Instead of one set of column histograms, five sets are now needed: one for the vertical sides to the right and the left, and four for the diagonals. Note that diagonals need separate sets for the left and right side because they have differing orientations.

The algorithm is still $O(1)$, albeit with a higher constant. Five sets of histograms must be kept up to date instead of one in the square kernel case. Moving the kernel histogram from one pixel to another now requires three histogram summations and three histogram subtractions instead of one of each in the square kernel case.

Although our implementation of the octagonal filter is not optimized, we can expect a fast one to be about 5 times slower than the square kernel. It should be possible to devise better geometric constructions which could possibly lower the constant while keeping the algorithm $O(1)$. As another example, hexagonally-tessellated CCDs such as those made by Fuji could benefit from a hexagonal kernel, which would be built following a similar reasoning. What should be pointed out is that the constant scaling of the proposed algorithm does not depend on the kernel being rectangular.

VI. Results

The new algorithm was compared against Photoshop CS 2, which features an implementation of Huang's classic $O(r)$ algorithm, and against Pixfoliate, a Photoshop plugin distributed by Weiss[1], implementing his $O(\log r)$ algorithm. The latter is the fastest 2-D median filter algorithm known to the authors. Timing was conducted on a PowerMac G5 1.6 GHz running Mac OS X 10.4. An 8-megapixel (3504×2336) RGB image with typical content (shown in Figure 4) was filtered with a varying filter radius. The results are displayed in Figure 5. One can see the flat trace generated by the proposed algorithm, indicative of its constant runtime complexity.

The optimizations of sections III-C and III-D are data-dependent and there exist pathological cases designed to disrupt the hypotheses on which those optimizations rely. One such case is the rainbow image shown in Figure 5e, for which per-pixel processing time was measured to be about twice as long as for the sails image, with a peak ratio of 2.33 at $r = 100$. Figure 5f shows the resulting output featuring an unusual smoothly varying signal, defeating optimization III-D. As for the the best case (solid black), it was processed twice as fast. Timing with an assortment of stock photographs produced results identical to those of Figure 5. Table I shows different processors' affinity for the proposed algorithm. All four optimizations were used on those processors.

It may appear surprising that the $O(1)$ algorithm is so much faster than the $O(r)$ algorithm for small kernel sizes. From experience, a reduction in complexity often comes at the price of an increase in the associated constant. In this case, our algorithm is both less complex and more efficient by a large margin at all kernel sizes compared with the classic algorithm. It could also be argued that it is simpler by comparing Algorithms 1 and 2 side by side.

[1] http://www.shellandslate.com/pixfoliatemacdemo.html

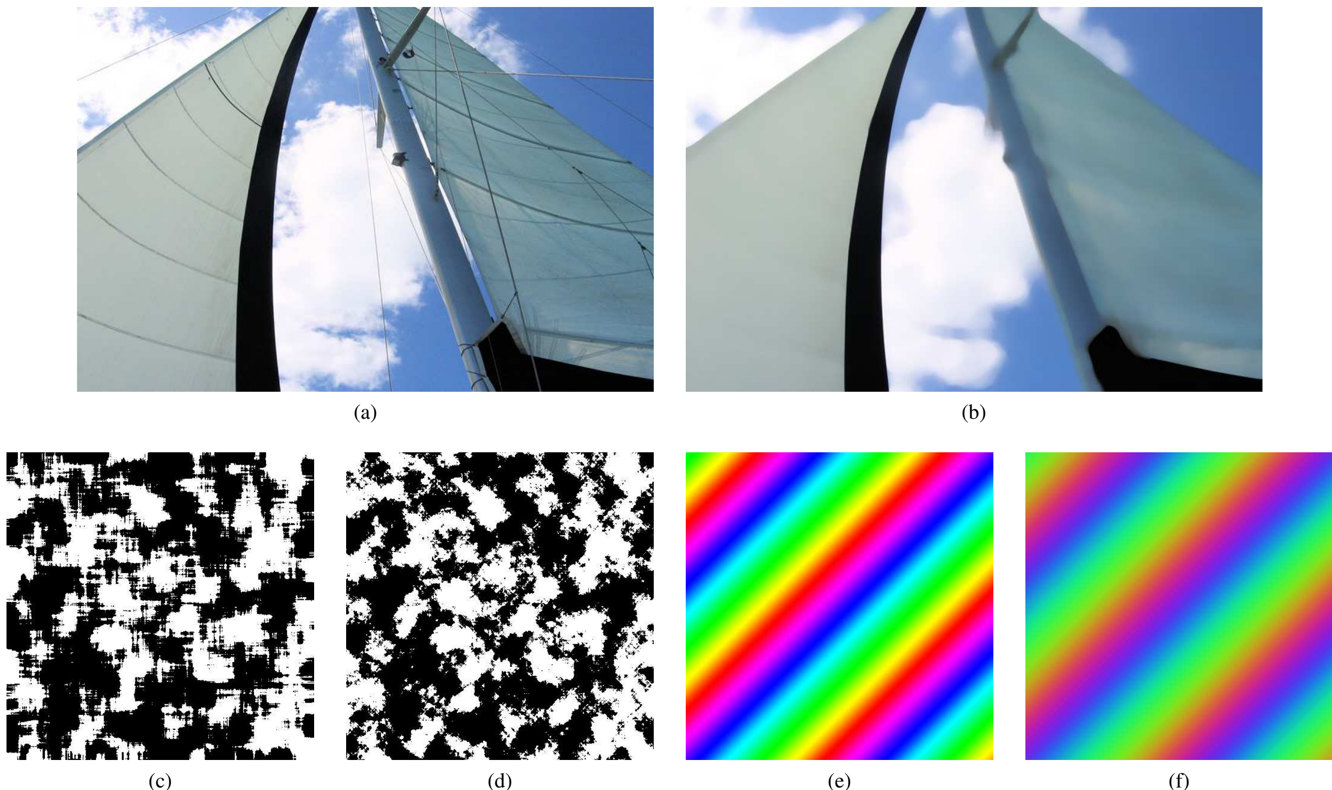

Fig. 4: Effect of median filtering. (a) Original 8-megapixel RGB image. (b) The same image after applying the median filter with a square kernel of 50 pixels radius. Notice sharpness of edges while small structures have been lost. (c) 512×512 image of uniform binary noise filtered with a square kernel of 20 pixels radius. (d) Same image filtered with an octagonal kernel of 20 pixels radius. Notice disappearance of horizontal and vertical line artifacts. (e) Closeup of pathological case which tries to defeat the optimizations of sections III-C and III-D. It is a 512×512 region of an 8 megapixel image of a periodic truncated triangle. (f) Output after filtering at $r = 100$.

TABLE I: Comparison of the proposed method's efficiency on different processors. ($r = 50$ on an 8-megapixel RGB image.)

Processor	SIMD instruction set	L2 cache size	Clock cycles per output element
PowerMac G5 1.6 GHz	AltiVec	512 kB	102
Intel Core 2 Duo E6600	SSE2	4096 kB	153
AMD Sempron 2400+	MMX	256 kB	296
Intel Pentium 4 1.8 GHz	SSE2	256 kB	628

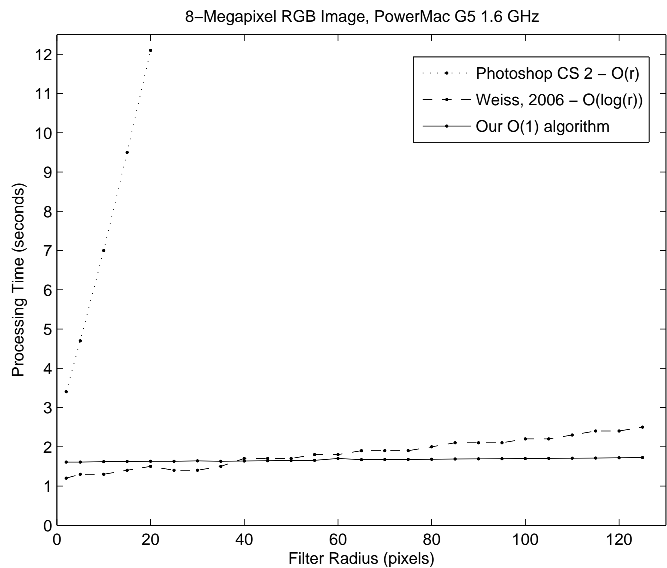

Fig. 5: Timing of the proposed algorithm.

The new algorithm performs better or worse than Weiss' $O(\log r)$ depending on the value of r. The crossover is at $r = 40$, although the traces are fairly parallel. This gives the crossover point a high sensitivity to experimental conditions. Since Weiss' Pixfoliate software is only available on the PowerPC architecture, comparison was only carried out on this platform. A slight reduction of the constant on current or future architectures would greatly lower the crossover radius.

Given the similar timings of the two best algorithms, the difference lies in two places. First, the tree of histograms in Weiss' algorithm makes the implementation fairly convoluted and generally unsuitable for custom hardware. Also, as a higher tree is required for greater radii, different implementations need to be generated, each one handling a portion of the radius range. In contrast, our implementation of the proposed algorithm, including optimization, totals about 275 lines of C code and handles all of the radius range. Second, the proposed algorithm has the advantage of constant complexity. This means that it will perform better than one of higher complexity as the kernel radius increases. Given the trend toward higher-resolution images, requiring correspondingly higher filter kernel radii, this makes the proposed algorithm future-proof. Faster hardware with better vectorization capabilities will also contribute to lowering its time constant.

VII. Conclusion

We have presented a fast and simple median filter algorithm whose runtime and storage scale in $O(1)$ as the kernel radius varies. We have proposed a few optimizations that make this algorithm as fast as the fastest currently available, to the extent of our knowledge, while remaining much simpler. With its straightforward instruction-level parallelism, it is suitable for CPU-based as well as custom hardware implementation.

Significant issues regarding the extensibility of the algorithm to new situations have been addressed. In particular, filtering data of higher dimensionality or precision, as well as an approximation to a circular kernel, have been discussed. We have also shown why the Gil-Werman theoretical lower bound of $\Omega(\log r)$ on the complexity of the 2-D median filter does not apply to traditional algorithms making use of histograms for sorting the data.

An implementation in C of the proposed algorithm is available freely on the authors' website[2] and has been included in the popular and free OpenCV[3] computer vision library. We are confident that new, clever optimizations will further lower its time constant. We hope the simplicity, speed and adaptability of this new algorithm will make it useful across a wide range of applications.

References

[1] J. Tukey, *Exploratory Data Analysis*. Addison-Wesley Menlo Park, CA, 1977.

[2] P. Maragos and R. Schafer, "Morphological Filters–Part II: Their Relations to Median, Order-Statistic, and Stack Filters," *IEEE Trans. Acoust., Speech, Signal Processing*, vol. 35, no. 8, pp. 1170–1184, 1987.

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

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

[5] B. Chaudhuri, "An Efficient Algorithm for Running Window Pel Gray Level Ranking 2-D Images," *Pattern Recognition Letters*, vol. 11, no. 2, pp. 77–80, 1990.

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

[7] L. Alparone, V. Cappellini, and A. Garzelli, "A Coarse-to-Fine Algorithm for Fast Median Filtering of Image Data With a Huge Number of Levels," *Signal Processing*, vol. 39, no. 1-2, pp. 33–41, 1994.

[8] D. Knuth, *The Art of Computer Programming Volume 3: Sorting and Searching*, 2nd ed. Addison Wesley Longman Publishing Co., Inc. Redwood City, CA, USA, 1998.

[9] T. Nelson and D. Pretorius, "Three-Dimensional Ultrasound of Fetal Surface Features," *Ultrasound in Obstetrics and Gynecology*, vol. 2, no. 3, pp. 166–174, 1992.

[10] P. Carayon, M. Portier, D. Dussossoy, A. Bord, G. Petitpretre, X. Canat, G. Le Fur, and P. Casellas, "Involvement of Peripheral Benzodiazepine Receptors in the Protection of Hematopoietic Cells Against Oxygen Radical Damage," *Blood*, vol. 87, no. 8, pp. 3170–3178, 1996.

[11] G. Arce, "Multistage Order Statistic Filters for Image Sequence Processing," *IEEE Trans. Signal Processing*, vol. 39, no. 5, pp. 1146–1163, 1991.

[2] http://nomis80.org/ctmf.html

[3] http://www.intel.com/technology/computing/opencv/