FAST 8-BIT MEDIAN FILTERING BASED ON SEPARABILITY

David Cline *Kenric B. White* *Parris K. Egbert*

Brigham Young University Brigham Young University

ABSTRACT

We present a fast 8-bit median filter implementation based on a separability argument. Our strategy is to start with a naive separable implementation of the median filter which displays O(1) time complexity per pixel, and then optimize this implementation to reduce the time constant. The optimizations that we employ include (a) lowering the histogram memory requirements by using unsigned shorts or bytes as histogram elements, (b) consolidating operations by performing multiple short additions with a single integer addition, (c) employing SSE instructions to further speed up the histogram updates, and (d) updating only the part of the histogram that contains the median. We show that by employing these rather straightforward optimizations, we can achieve filter speeds that approach the performance of the fastest proprietary median filters currently in use.

Index Terms— Image processing, Median filters

1. INTRODUCTION

The median filter [1] is one of the most popular image processing filters in use today. It is defined as replacing each image pixel with the median value within a kernel mask surrounding the pixel. Applications of median filtering include image denoising, edge-preserving blur, background removal in handwriting recognition, image sharpening and painterly rendering effects (see figure 1).

The main drawback to the median filter, particularly for large kernel sizes, is filter speed. A brute force implementation involves gathering and sorting all of the values within the filter kernel, resulting in an $O(n^2 \log n)$ time complexity per pixel, where n is the width of the kernel. If the image data is integer-valued, sorting can be eliminated by using a histogram instead of a sorted list, reducing the time complexity to $O(n^2)$. This is still excruciatingly slow, however.

Huang [2] optimized the median filter, noting that the values within the filter kernel are mostly the same between neighboring pixels–the center of the kernel remains intact, but the leading and trailing edges of the kernel change. Huang took advantage of this fact by updating the kernel histogram from pixel to pixel rather than creating it from scratch each time. This lead to an $O(n)$ algorithm, which is quite fast for small kernel sizes, but becomes slow as the kernel size increases.

Fig. 1. Median filtering, with a kernel radius of 32, performed on a 6.4 megapixel image in less than one second using our algorithm.

Recently, Weiss [3] developed a patented median filter that works in $O(\log n)$ time per pixel. In Weiss's algorithm, a number of columns are processed at once, and the histogram for a given column is stored as a set of *partial histograms* with signed elements. By structuring calculations in this manner, Weiss's algorithm only needs to add each pixel to $\log n$ histograms rather than n as required in Huang's method.

In this paper we present an optimized median filter that works for 8-bit images and rectangular kernels. The new algorithm is based on separablility principles, and it runs in $O(1)$ time per pixel. Our algorithm works by keeping a separate histogram for each column of an image. The histogram for a particular kernel mask is incrementally assembled from multiple column histograms. While the obvious implementation of our algorithm results in a large time constant, we show that by applying a few straightforward optimizations to the initial implementation, we can achieve speeds near the performance of the fastest median filters currently in use.

2. A SEPARABLE IMPLEMENTATION OF THE MEDIAN FILTER

This section details our basic separable implementation of the median filter. Note that we are not talking about a "separable median" which computes the median of a set of medians. Rather, we produce the true median of the kernel mask, but optimize the implementation through separability principles.

As mentioned, our separable algorithm keeps a histogram for each column in the image. Before processing a line of pixels, we update all of the column histograms. This involves adding and subtracting one pixel value from each column his-

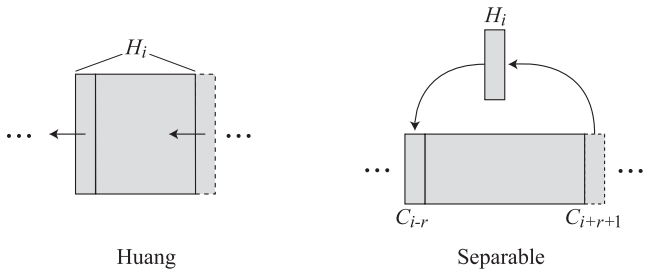

Fig. 2. Huang's method vs. our separable implementation. Huang updates the current histogram H_i by adding the values on the leading edge of the kernel to it and subtracting the values on the trailing edge from it. The result is an $O(n)$ algorithm per pixel. On the other hand, our separable implementation updates H_i by adding the column histogram on the leading edge of the kernel to it and subtracting the column histogram on the trailing edge of the kernel. This results in an $O(1)$ time complexity per pixel, with a time constant that is on the order of the histogram size.

togram. For column i, the update from row j to row $j+1$ can be expressed as follows:

$$C_{i,j+1} = C_{i,j} - I_{i,j-r} + I_{i,j+r+1}, \quad (1)$$

where $C_{i,j}$ is the histogram for column i with the kernel mask centered on row j, $I_{i,j}$ is the value of pixel (i, j) in the image, and r is the kernel radius.

After updating the column histograms, we initialize the kernel histogram for the left edge of the row, and then march across the row, using the column histograms to update the kernel histogram. This process is similar to Huang's method, except that we add and subtract complete column histograms rather than individual pixel values, as shown in figure 2. Thus, in its most basic form, our separable implementation requires 256 additions and 256 subtractions per pixel. Mathematically, we can express the kernel histogram update from position i to position $i+1$ as

$$H_{i+1} = H_i - C_{i-r} + C_{i+r+1}, \quad (2)$$

where H_i is the kernel histogram for position i, and once again C_i is the column histogram for column i.

After creating the kernel histogram, we must find the median within it. The simplest way to do this is to scan through the histogram starting at zero. In our code, we optimize the median search by scanning up from zero or down from 255 depending on whether the previous median was below or above 128. The graphs in figure 4 show the performance of the basic separable implementation, labeled as "separable (int)".

Notice that our basic separable algorithm has a complexity of $O(1)$ per pixel, though it does have a large time constant associated with it. Based on simply counting the number of atomic operations, one might expect Huang's method to outperform our separable implementation up to a kernel radius of about 128. The crossover point between the two methods is actually much lower, around 50, however. This extra speed can be explained by the natural vectorized nature of the separable implementation. The bulk of the processing in the separable algorithm lies in adding and subracting complete histograms (vector operations), whereas Huang's algorithm adds and subtracts single values (random access operations).

3. OPTIMIZING THE BASIC IMPLEMENTATION

This section describes a number of optimizations that take advantage of the inherent vector nature of the separable implementation. None of these optimizations are complicated, but taken together they nearly triple the performance of our implementation, pushing the crossover point between our method and Huang's to about radius 20.

Bottlenecks of the separable algorithm. It shouldn't be hard to see that updating the kernel histogram and finding the median are the bottlenecks of the separable implementation. In fact, updating the kernel histogram accounts for about 88% of the total run time. For this reason, it makes sense to concentrate optimization efforts on speeding up the kernel histogram updates.

Reducing column histogram memory usage. Assuming that we only care about kernel sizes up to 255×255, the column histograms can be stored as unsigned shorts or bytes rather than integers. Moving to unsigned bytes reduces storage and memory bandwidth requirements by a factor of 4. Figure 4 shows the result of making this change, labeled as "separable (byte)".

Performing two short adds with a single integer add. While storing the column histograms as unsigned shorts or bytes reduces memory throughput requirements, 512 operations are still needed to update the kernel histogram for each pixel. We can cut this number in half if we store both the kernel and column histograms as unsigned short integers. The idea is to store the histograms as unsigned shorts, but perform the update with unsigned integer additions. In essence, we are treating single integers as parallel registers that operate on two shorts simultaneously. This version of the algorithm is labeled "separable (multiadd)" in figure 4. Of course, we could perform four operations at once by storing the histograms as unsigned bytes, but that would restrict the maximum kernel radius to 7.

Combining operations using SIMD instructions. We can further consolidate the operations that must be performed during a histogram update by using SIMD instructions. SSE provides instructions to add or subtract 4 integers at a time, so by performing the histogram updates in SSE registers, the histogram updates reduce to 32 vector subtractions and 32 vector additions. Our timing results for the SSE version of the algorithm are labeled "separable (sse)" in figure 4. We did not attempt to optimize our code for the Power Mac, but we sus-

pect that similar speedups to those seen on the x86 platforms using SSE could be achieved on the Mac using Altivec.

At this point we have a fairly fast implementation for large kernels. For example, on our Pentium 4 test machine, the SSE optimized algorithm can filter a 6.4 megapixel image in about two seconds for kernels up to radius 127. This is about four times faster than our best implementation of Huang's algorithm at radius 127, and thirteen times faster than Photoshop 6.0 at radius 100. Nevertheless, the memory throughput of the algorithm remains very high, 1024 bytes per pixel. This high throughput stems from the fact that we add and subtract a complete column histogram for each pixel. In the next section, we will show how to reduce the throughput by as much as 8 times by only updating part of the histogram each time.

4. PARTIAL HISTOGRAM UPDATES

Up to now our optimization strategy has been built around pushing the same amount of data through the computer faster. In this section, we take a different tack, concentrating on finding the median while reducing the required data throughput. Our basic strategy is to identify a small segment of the histogram that is guaranteed to contain the median, and then update only this part of the histogram.

The coarse histogram. In our implementation, we divide the histogram into 16 segments, each containing 16 elements. To determine the histogram segment that contains the median, we define a "coarse" histogram that is the histogram of pixel values that have been divided by 16. As with the standard histograms, we store a coarse histogram for each column in the image. Updating the coarse histogram for a pixel only requires $1/16^{th}$ the data throughput of the full 256 element histogram.

Updating the histogram segment containing the median. An important property of the coarse histogram is that its median tells us which segment of the full histogram the true median lies in. In particular, if element s is the median of the coarse histogram, the median of the full histogram must lie between elements $16s$ and $16(s+1)$. Thus, we can find the median as follows: first, we update the coarse histogram and find its median. Next, we update the segment of the full histogram corresponding to the coarse median and search through this segment to find the true median.

We keep track of the last column in which the algorithm updated each histogram segment, and use this information to decide whether to update the segment incrementally or rebuild it from scratch. In the common case where the median lies in the same segment as the previous pixel, the segment update only requires 16 additions and 16 subtractions, which can be done in SSE instructions with just 2 additions and 2 subtractions, assuming that we store histogram elements as unsigned shorts. Even in the worst case, the partial update does not require more segment updates than updating the full histogram each time.

```
1.  Initialize the coarse column histograms.
2.  Initialize the full resolution column histograms.
3.  For j = 0 to imageheight − 1
4.      Initialize the coarse kernel histogram.
5.      Initialize the full resolution kernel histogram.
6.      Set the last updated column to 0 for all segments.
7.      Find the median and write it to the destination image.
8.      For i = 1 to imagewidth − 1
9.          Update the coarse kernel histogram.
10.         Find the coarse median and sum of elements below it.
11.         Update the hist. segment S containing the coarse median:
12.             Let c be the last column for which S was updated.
13.             If i − c > the kernel radius, r
14.                 Update S from scratch.
15.             Else
16.                 Update S incrementally.
17.         Find the median starting at the beginning of S.
18.         Write the median to the destination image.
19.         Set the last updated column for S to i.
20.     Update the coarse column histograms.
21.     Update the full resolution column histograms.
```

Fig. 3. Partial histogram update median filtering algorithm.

Finally, we note that because of cache issues it is much more efficient to store histogram segments as contiguous blocks of memory rather than storing complete histograms together. Thus, it is better to store the column histograms as 16 arrays with $16 \times imagewidth$ elements rather than one array with $256 \times imagewidth$ elements.

Figure 3 gives pseudocode for the partial update algorithm. Timing results for this method are labeled as "partial update" in figure 4.

5. TIMING RESULTS

This section gives timings for different variants of our median filtering algorithm on three test platforms, a 3.2 GHz Pentium 4, a 2.2 GHz Athlon 64 3500+, and a 2 GHz Power Mac G5. The Pentium 4 graph also shows timings for Photoshop version 6.0 for comparison. (This was the only platform on which we had access to Photoshop.) All of the timing results are for the 3000×2250 image shown in figure 1, although we recorded similar filter times for a number of other test images.

Huang's method. To create timing results for Huang's method, we coded several different variants of his algorithm, such as scanning vertically or horizontally over the image, and keeping track of a pivot or not. Then, for each kernel size we kept the best time of all the variants. Just as an aside, our experiments with Huang's algorithm yielded some interesting

results. Most interesting was the disparity between horizontal and vertical scanning. Our experiments showed that scanning horizontally is faster for small kernels up to about radius 50, but slower for large kernels. In fact, choosing the right scan direction resulted in a speedup of more than four times in some of our tests.

The separable methods. The timings labeled "separable..." show the results of the different optimizations described in section 3. Our basic separable implmentation, "separable (int)", is already appreciably faster than our best implmentation of Huang's algorithm for very large kernels, but the optimizations outlined in section 3 more than double the performance of the separable implementation on the Intel and AMD test machines, and nearly double the performance on the Power Mac even without employing SIMD instructions.

Partial histogram updates. Performing partial instead of complete histogram updates as described in section 4 more than doubled the already fast speed of the separable implementation on all three of our test platforms. Unfortunately, SSE instructions provided only a slight improvement ($\sim$7%) on the AMD test machine, and no improvement at all on the Intel test machine (so we did not graph "partial update (sse)" for the Intel platform). Nevertheless, the "partial update" timing results on all of our test machines are still quite fast, and the crossover point between our partial update algorithm and Huang's method is around radius 7 for all three platforms. Thus, we recommend using Huang's method (with a horizontal scan) up to kernel radius 7, and the partial update method for larger sizes.

Comparison with Weiss's method. We do not have the capability currently to test our method directly against Weiss's Photoshop plugin, but based on his published results, we believe that our algorithm is within two or three times the performance of his proprietary method for 8-bit data. (He filtered an 8 megapixel RGB image in about 1.4 seconds on a 2.5 GHz Power Mac G5, and we filtered a 6.4 megapixel grayscale image in 1.3 seconds on a 2.0 GHz Power Mac.)

Recent developments. While finishing this article, we discovered that simultaneous to our work, researchers at Université Laval in Quebec, Simon Perreault and Patrick Hébert, were working on a very similar median filter algorithm. Their work has since been accepted for publication in IEEE Transactions on Image Processing, and preliminary results can be found on the web at *http://nomis80.org/ctmf.html*.

6. REFERENCES

[1] John Tukey, "Exploratory Data Analysis," 1977, Addison-Wesley.

[2] T. S. Huang, "Two-Dimensional Signal Processing II: Transforms and Median Filters," 1981, pp. 209–211, Berlin: Springer-Verlag.

[3] Ben Weiss, "Fast median and bilateral filtering," in *ACM Trans. Graph.*, New York, NY, USA, 2006, vol. 25, pp. 519–526, ACM Press.

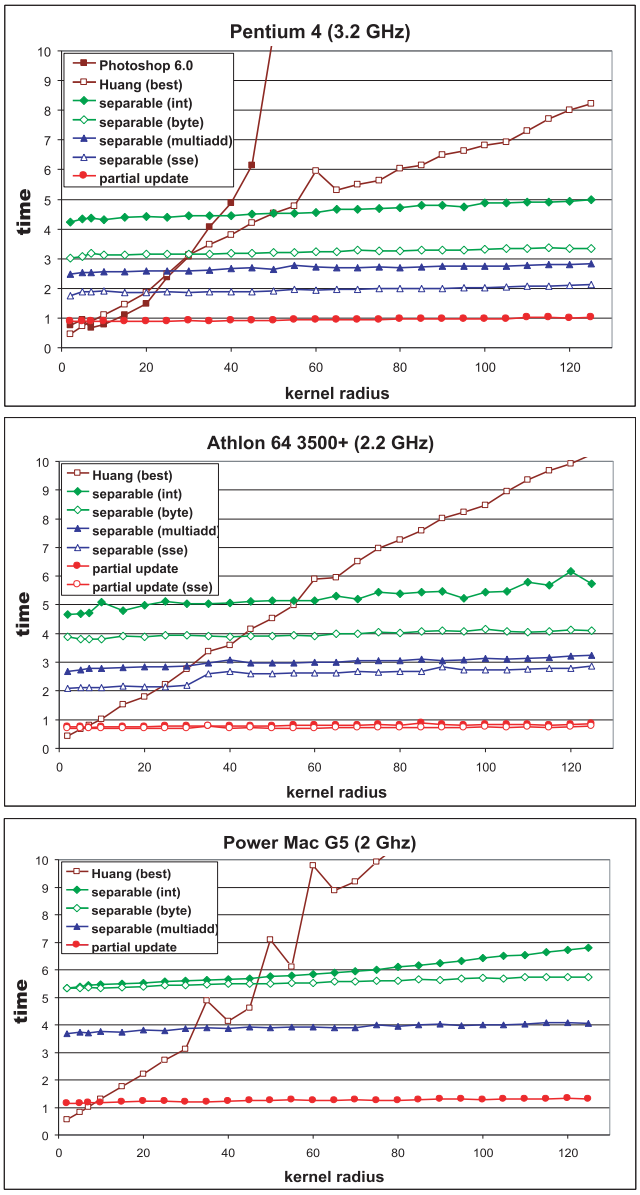

Fig. 4. Median filter timings for different hardware configurations. All timings are for the 6.4 megapixel test image from figure 1.