Tips & Tricks: Fast Image Filtering Algorithms

Alexey Lukin
Department of Computational Mathematics and Cybernetics
Moscow State University, Moscow, Russia
lukin@graphics.cs.msu.ru

Abstract

This paper highlights some fast algorithms for image filtering, specifically – box and Gaussian smoothing, Hann filtering, median filtering, and morphological operations. It is shown that some of these algorithms can be implemented with computational cost independent of a filter radius.

Keywords: *fast image filtering, Gaussian filter, Hann filter, median filter.*

1. INTRODUCTION

Image smoothing is one of the most widely used operations in image processing. A variety of well-known linear and non-linear smoothing algorithms exist, such as Gaussian or median filtering. They are used as parts of more complex algorithms including lightness equalization, noise reduction, contrast and sharpness enhancement.

A constant increase of resolution of digital images and photographs implies significant increase of calculations for image processing. The computational cost is not only affected by the increased image area, but also by the fact that required filter radiuses increase together with resolution. With straightforward O(r^2) filter implementations (r is a filter radius) the increase of resolution (image area) by a factor of k brings the increase of computational complexity by a factor of k^2. Many modern CPUs contain vector instructions (e.g. MMX technology) to increase performance of many filtering operations.

The purpose of this paper is to highlight some widely used and less known optimizations of image filtering algorithms. We also suggest a fast algorithm for Hann-window smoothing. Some of these algorithms achieve fixed computational cost per pixel, independent of a filter radius, and can significantly outperform straightforward implementations in case of large filter radiuses.

2. LINEAR SMOOTHING

2.1 Box blur

A box blur, also known as "moving average", is a simple linear filter with a square (or rectangular) kernel and all kernel coefficients equal. It is the quickest blur algorithm, but it lacks smoothness of a Gaussian blur.

A box blur allows implementation with a complexity independent of a filter radius. The algorithm is based on a fact that sum S of elements in the rectangular window can be decomposed into sums C of columns of this window:

$$S[i,j] = \sum_{k=-r}^{+r} C[i,j+k]$$

This allows a simple update rule when window S is moving from left to right:

$$S[i,j+1] = S[i,j] + C[i,j+r+1] - C[i,j-r]$$

Column sums C can be, in turn, effectively updated when window S is moving down to the next row:

$$C[i+1,j] = C[i,j] + x[i+r+1,j] - x[i-r,j]$$

Here $x[i,j]$ are image pixel values, $C[i,j]$ are sums of $(2r+1)$-pixel columns centered at $[i,j]$, and $S[i,j]$ are sums of $(2r+1)(2r+1)$-pixel square windows centered at $[i,j]$.

The result of the box filter is equal to

$$B[i,j] = \frac{1}{(2r+1)^2} S[i,j]$$

and it requires one multiplication per pixel after $S[i,j]$ is calculated. So, the overall per-pixel complexity of the box filter is 4 additions and 1 multiplication per pixel. Some additional overhead is required to calculate initial values of C and S at the image boundaries.

2.2 Triangular blur

A box blur turns impulsive image pixels into squares, step functions into linear ramps and its frequency response has high side lobes. So, its smoothing effect is often not sufficient. A simple improvement is an iterative application of a box filter to the image. Consecutive passes of filtering are equivalent to convolution of filter kernels. So, as the number of iterations increases, the superposition filter approaches a Gaussian.

A widely used triangular filter (or Bartlet window) can be constructed as a superposition of 2 box filters with the same radius.

The computational complexity of such filter is twice the complexity of a box filter, and the visual effect is very similar to Gaussian blur. The filter requires 2 passes through the image which can be overlapped due to filter locality.

2.3 Gaussian blur

Gaussian blur is considered a "perfect" blur for many applications, provided that kernel support is large enough to fit the essential part of the Gaussian. Indeed, the Gaussian filter's frequency response is also Gaussian, it has fast fall-off and no side lobes.

Gaussian filter on a square support is separable, i.e. 2D filtering it can be decomposed into a series of 1D filtering for rows and columns. When the filter radius is relatively small (less than few dozens), the fastest way to calculate the filtering result is direct 1D convolution. The filter symmetry can be exploited to reduce the number of multiplications by a factor of 2.

When a filter radius is large, direct convolution becomes expensive, and FFT-based OLA convolution is the algorithm of choice. A common mistake here is to transform the whole image row (or

column) with FFT, do the same with a zero-padded Gaussian kernel, multiply complex spectra and do the inverse transform.

First of all, it should be considered that the result of convolution has a length $N+M-1$, where N is the signal size and M is a filter kernel size (equal to $2r+1$), i.e. the output signal is longer than the input signal. Without proper padding (extension) of data, regular convolution will turn to a circular convolution, leading to problems near image boundaries.

Secondly, calculating FFT of the complete image row is not optimal, since the complexity of FFT is $O(N\log N)$. The complexity can be reduced by breaking the kernel into sections with an approximate length M and performing overlap-add (OLA) convolution section-wise. The FFT size should be selected so that circular convolution is excluded. Usually optimal performance is achieved when FFT size F is selected as the smallest power of 2 larger than $2M$, and signal section size is selected as $F-M+1$ for full utilization of FFT block. This reduces the overall complexity of 1D convolution to $O(N\log M)$.

So, the per-pixel complexity of Gaussian blur becomes $O(\log r)$. However, the constant is quite large, and for many practical purposes Gaussian blur can be successfully approximated with simpler filters.

2.4 Hann window

Hann window is a smooth function defined as

$$H(t) = 1 + \cos(t), \quad -\pi \le t \le \pi$$

It is characterized by a compact support and fast fall-off of side lobes in frequency response, i.e. good smoothing effect.

The algorithm that we propose for 1D Hann smoothing is based on modulation of the input signal with a complex exponent. Let's consider a discrete filtering with a Hann kernel:

$$h[t] = \frac{1}{2r+1}\sum_{k=-r}^{r}\left(1+\cos\frac{k\pi}{r+0.5}\right)x[t+k]$$

This can be rewritten as sum of a box filter and a cosine-modulated input signal. A box filter can be effectively calculated as described in section 2.1. Now we will deduce the update formula for fast calculation of a cosine modulated real-valued signal.

$$y[t+1] = \sum_{k=-r}^{r} x[t+1+k]\cos\frac{k\pi}{r+0.5} =$$

$$= \mathrm{Re}\sum_{k=-r}^{r} x[t+1+k]\exp\frac{k\pi i}{r+0.5} =$$

$$= \mathrm{Re}\sum_{k=-r+1}^{r+1} x[t+k]\exp\frac{(k-1)\pi i}{r+0.5} =$$

$$= \mathrm{Re}\left\{\left(\sum_{k=-r}^{r} x[t+k]\exp\frac{k\pi i}{r+0.5}\right)\exp\frac{-\pi i}{r+0.5} + \right.$$

$$\left. + x[t+r+1]\exp\frac{r\pi i}{r+0.5} - x[t-r]\exp\frac{(-r-1)\pi i}{r+0.5}\right\} =$$

$$= \mathrm{Re}\left\{\left(\sum_{k=-r}^{r} x[t+k]\exp\frac{k\pi i}{r+0.5}\right)\exp\frac{-\pi i}{r+0.5} + \right.$$

$$\left. + \big(x[t+r+1] - x[t-r]\big)\exp\frac{r\pi i}{r+0.5}\right\}$$

If we denote

$$z[t] = \sum_{k=-r}^{r} x[t+k]\exp\frac{k\pi i}{r+0.5}$$

then

$$y[t] = \mathrm{Re}\, z[t]$$

and

$$z[t+1] = \exp\frac{-\pi i}{r+0.5} z[t] + \big(x[t+r+1] - x[t-r]\big)\exp\frac{r\pi i}{r+0.5}$$

This update rule can be used for effective calculation of $y[t]$ and $h[t]$. So, the complexity of a 1D Hann filter is 7 real-valued multiplications and 9 additions per pixel independently of a filter radius.

The separable 2D Hann filter does not have strict radial symmetry, but it is close to symmetric. It allows a fast separable 2D algorithm.

2.5 Simple recursive filters

Another fast way to blur the image is to use 1st order recursive filters:

$$y[t] = (1-\alpha)y[t-1] + \alpha x[t]$$

This simple formula can be modified to include only 1 multiplication:

$$y[t] = y[t-1] + \alpha(x[t] - y[t-1])$$

This filter has a one-sided exponential infinite impulse response (IIR). To obtain a symmetric impulse response, the filter can be applied twice: in forward and backward direction, giving the complexity of 4 multiplications per pixel in a separable 2D variant.

2.6 Recursive approximations of Gaussian

Using higher orders of recursive filter allows a good approximation of Gaussian filtering. The following recursive 1D filter is suggested in [1]:

$$y[t] = Bx[t] + (b_1 y[t-1] + b_2 y[t-2] + b_3 y[t-3])/b_0$$

This filter should be applied twice: in forward and backward directions, yielding 12 multiplications per pixel in 2D case, independently of a filter radius. Filter coefficients can be calculated as follows:

$$B = 1 - \frac{b_1 + b_2 + b_3}{b_0}$$

$$b_0 = 1.57825 + 2.44413q + 1.4281q^2 + 0.422205q^3$$

$$b_1 = 2.44413q + 2.85619q^2 + 1.26661q^3$$

$$b_2 = -1.4281q^2 - 1.26661q^3$$

$$b_3 = 0.422205q^3$$

$$q = \begin{cases} 0.98711\sigma - 0.96330, & \sigma \geq 2.5 \\ 3.97156 - 4.14554\sqrt{1 - 0.26891\sigma}, & 0.5 \leq \sigma \leq 2.5 \end{cases}$$

A relative accuracy of this approximation increases as filter radius σ increases, but even with small σ the accuracy is good [1].

3. NON-LINEAR PROCESSING

3.1 Median filtering

Median is a non-linear local filter whose output value is the middle element of a sorted array of pixel values from the filter window. Since median value is robust to outliers, the filter is often used for reduction of impulse noise. Another useful property of a median is retention of edge sharpness while removing minor details from the image.

The straightforward implementation of median filter requires $O(r^2 \log r)$ operations per pixel to sort the array of $(2r+1)(2r+1)$ pixels in a window. However an optimization is possible when image data takes a limited range of discrete values, e.g. 8-bit pixel values. It is based on a fact that median value can be easily calculated from a histogram of pixel values in a window. For 8-bit pixel values such a histogram contains 256 bins and can be searched for a constant time (8 comparisons) independently of a filter radius. When a filter window shifts, this histogram can be effectively updated using the following algorithm: if the filter window shifts one pixel down, the pixels of upper window row are removed from the histogram ($2r+1$ operations), and pixels of a new lower window row are added to the histogram ($2r+1$ operations). During the course of 2D median filtering, the filter window can zigzag through the image allowing effective histogram updates on every step. Such a histogram-based algorithm requires only $O(r)$ operations per pixel.

To optimize the histogram search, a previously calculated median value can be used as a starting point in a search for a new median value.

A further optimization of median filtering is possible by maintaining several histograms as combining them in a certain way. In [2], the $O(\log r)$ algorithm is presented and ways to adapt this algorithm to 16-bit and floating-point data are discussed.

3.2 Binary morphological operations

A basic morphological operation is dilation. When a structuring element is defined inside a square window with a radius r, the dilation operation sets to 1 all the pixels from which the structuring element overlaps at least one non-zero pixel of the source image. A straightforward implementation of dilation requires $O(r^2)$ operations per pixel to check all the points of a structuring elements.

If we keep the number of non-zero pixels that are overlapped by a structuring element, an efficient update rule can be used for this number. When a structuring element window shifts one pixel to the right, some image pixels that can become overlapped are shifting in from the right border of a structuring element, and some image pixels can be shifting out of overlapping area through the left border of a structuring element. So, instead of counting a total number of overlapping pixels, we can increment the previous count by a number of pixels covered by the right border of the structuring element and decrement by the number of pixels that are lying to the left of the left border of a structuring element. The complexity of this optimized dilation is $O(r)$.

A similar optimization is possible for erosion operation. For erosion we will count the number of zero image pixels overlaid by a structuring element.

3.3 Min/Max filters

A max filter outputs a maximal pixel value from its rectangular window. A straightforward implementation requires $O(r^2)$ operations per pixel.

In case of small data bit depth, a histogram approach described in section 3.1 can be used. But when the bit depth is large, another approach based on a 1D running max filter appears more practical. Fast implementations of 1D running max filter are described in [3]. A simple and fast algorithm called MAXLINE2 is using a circular buffer of delayed input elements. The anchor points to the current maximal value. When the window is shifted, a new element is added to the delay line and compares against anchor element. If the new element is smaller, the maximum stays at the anchor. Otherwise anchor moves to a new element. When the anchor shifts out of the delay line, the whole delay line is scanned for a new anchor.

This algorithm works very fast on i.i.d. (independent identically distributed) data, but has a worst-case complexity of $O(r)$ for a monotonically decreasing data. An algorithm with a better worst-case complexity (although with a worse complexity on i.i.d. data) is also introduced in [3]. It has a complexity of $O(\log r)$.

This running max algorithm can be used for adding pixels to a 2D window of a 2D min/max filter with a worst-case complexity of $O(r \log r)$ operations per pixel.

3.4 Grayscale morphological operations

Grayscale morphological operations are based on min/max filters. When structuring element is rectangular, they can be optimized using min/max filters discussed in section 3.3.

4. CONCLUSION

Many image filtering algorithms can be effectively implemented with a reduced number of operations per pixel. This paper describes some optimizations that in many cases allow achieving computational complexity independent of a filter radius.

5. REFERENCES

[1] I.T. Young, L.J. van Vliet “Recursive implementation of the gaussian filter” // Signal Processing (44), pp. 139–151, 1995.

[2] B. Weiss “Fast Median and Bilateral Filtering” // ACM Transactions of Graphics (proceedings of the ACM SIGGRAPH'06 Conference), 2006.

[3] M. Brookes “Algorithms for max and min filters with improved worst-case performance” // IEEE Transactions on Circuits and Systems, Volume 47, Issue 9, Sep 2000, pp. 930–935.

The work has been supported by RFBR grant 06-01-39006-ГФЕН.

About the author

Alexey Lukin (Ph.D.) is a member of scientific staff of a Moscow State University, Department of Computational Mathematics and Cybernetics. His contact email is lukin@graphics.cs.msu.ru