Comparative Analysis of the Optimization Techniques of Image Filters Used to Improve Optical Character Recognition on Text Images

Iulia Ştirb
Computer and Software Engineering Department, "Politehnica" University of Timişoara, România

Abstract - *Image filtering is changing the appearance of the image by altering the shades and colors of the pixels. Increasing the contrast as well as adding a variety of texture, tones and special effects to images are some of the results of applying image filters for the purpose of improving the success rate of OCR (Optical Character Recognition) performed on text image. Still the main purpose of filtering is reducing the noise around characters in the image. Within Silicon and Software System Limited (S3Group) Company from Dublin, Ireland, the image filters described in this paper are used to improve a subsequent OCR which is part of the process of testing the menu options displayed on the output video of the set-top box under test. This paper focuses on the optimization techniques and on the analysis of which better suits to be applied to each of the image filters in order to make them perform faster. The optimization that produces the best improvement in terms of execution time for all filters above is done using a byte array to store the components i.e. Blue, Green, Red and Alpha of each pixel. Two other optimizations are described, one using C# predefined ColorMatrix class for improving Contrast filter and the other by computing just once the filter weight (i.e. sum of all template elements) for all pixels in the interior of the image in the case of Sharpen and Blur filters, which both use a template to compute the new value of the pixels.*

Keywords: image, filter, optimization, speedup, contrast, OCR

1 Introduction

The OCR of videos, involving first an automatic extraction of many frames (captures) in a second, as in [1], is proved to be very useful in digital television domain. Lately, in digital television, the need for a high number of OCR on text images in a short time interval is increasingly common. Filtering the images before passing them to OCR is becoming more and more frequent because of the need to remove the noise. Since the OCR speed must increase, so the filtering speed must do.

The optimization techniques in terms of execution time of Contrast, Sharpen, Blur, Invert, Color and Highlight filters are presented the first and second sections of this paper and the techniques will be compared in the third section. Snippets of initial and optimized (using byte buffer) implementation of image filters are listed in the fourth section. Conclusions section states the proper optimization technique to be used for each filter.

What this paper brings new is a case study of which optimization technique suits well to each filter separately. I have carried out many tries of applying the optimization techniques on all filters. For instance, Contrast filter's most optimized version is the one in which the implementation is done using C# `ColorMatrix` class.

2 Optimization Techniques

2.1 Usage of a byte buffer

The byte buffer stores byte representation of the image, namely Blue, Green, Red and Alpha components in this order for each pixel as in Fig. 1. An additional cloned buffer is required for Sharpen and Blur filters to avoid altering the original pixels, since the new value of the current pixel depends on its neighboring pixels.

First step in the optimization algorithm is to obtain an object of type `BitmapData` (more details can be found in [2]) using C# `LockBits` method of `Bitmap` class. In next step `Scan0` property is accessed on the obtained object in order to get the address of the first pixel in the image. Once we have this address, all the image data from it is copied to the byte buffer using the overloaded `Copy` method of C# `Marshal` class. The third step is different from a filter to another and represents the calculations done on the byte buffer in order to filter the image. Finally, the byte buffer will contain the byte representation of the filtered image. By using the buffer, we avoid calling `SetPixel` method on a `Bitmap` object each time the pixels needs to be set to the filtered value, which would be time consuming since the `Bitmap` object is accessed as many times as the number of pixels in the image. The next step is to copy the filtered buffer back to the address of the first pixel of the image using the overloaded `Copy` method. The last step is to unlock the `Bitmap` object, using `UnlockBits` method.

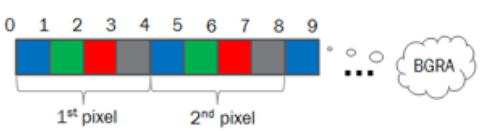

Figure 1. Image representation storage in the byte buffer

2.2 Usage of ColorMatrix class

Figure 2 present an example of how this technique performs for one pixel with e.g. Red component equal to 51 and scale down to 0.2. As in [3], the input vector represents the original pixel with its components Red, Green, Blue and Alpha in this order. The values of the components vary in a range from 0 (= 0 / 255) to 1 (= 255 / 255). The last element of the input vector is used for additional computations, if needed. After the calculation of the output array, the values are scaled again to the fit into the interval [0;255].

The matrix content is specific to each filter and exemplified in Fig. 3 for Contrast filter. *diff* is computed once for all pixels as in (1) and *scale* is the desired degree of Contrast (usually varies from 1 to 6). *diff* usage saves a lot of calculations per pixel, that should have normally been made.

$$diff = -0.5 * (scale - 1.0) - 0.5 / 255.0 \qquad (1)$$

2.3 Comparison Between the Optimization Techniques

Since using `ColorMatrix` implies having to perform the operation on the `Bitmap` object that encapsulates the image, using also a byte buffer would make no sense, so the two cannot be both applied to image filters. Details can be found in Table I.

TABLE I. Comparison between the optimization techniques improvements in terms of execution time of the image filters

Curr. No.	Image Filter	Optimization technique used			Speedup
		Byte Buffer	Color Matrix	Filter Weight	
1	Contrast	-	✓	-	8.7 times
2	Sharpen	✓	-	✓	4.7 times
3	Blur	✓	-	✓	4.7 times
4	Invert	✓	-	-	53 times
5	Color	✓	-	-	42 times
6	Highlight	✓	-	-	16 times

2.4 Initial Implementation of Filters and the One Optimized Using the Buffer

Code 1. Image filters implementation before being optimized (C#)

```
Bitmap clonedBitmap = new Bitmap(originalImage.Width,
                                 originalImage.Height);
UnsafeBitmap bitmap = new UnsafeBitmap(originalImage);
bitmap.LockBitmap();
…
// for each pixel with coordinates (x,y) set the filtered value
clonedBitmap.SetPixel(x,y,Color.FromArgb(r,g,b));
…
bitmap.UnlockBitmap();
return clonedBitmap;
```

Code 2. Image filters optimized implementation using the byte buffer technique (C#)

```
FilteredBitmap filteredBitmap = new
                                FilteredBitmap(originalImage);
byte[] buffer = filteredBitmap.LockBitmap();
…
```

$$\begin{bmatrix} 0.2 & 0.7 & 0.5 & 1.0 & 1.0 \end{bmatrix} \times \begin{bmatrix} 2 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0.5 & 0.1 & 0.2 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 0.9 & 0.8 & 0.7 & 1.0 & 1.0 \end{bmatrix}$$

Figure 2. `ColorMatrix` example of multiplication for one pixel

scale	0	0	0	0
0	scale	0	0	0
0	0	scale	0	0
0	0	0	scale	0
diff	diff	diff	diff	1

Figure 3. Color matrix for Contrast image filter

```
// here is the code which processes the buffer elements obtaining
// the filtered values; this part is different from a filter to another
…
// set the new value of all pixels, by accessing the image only once
filteredBitmap.SetPixels(buffer);
filteredBitmap.UnlockBitmap();
return filteredBitmap.Bitmap;
```

In Code 1, a reason for the high execution time when using `SetPixel` C# method is the assumption that the image representation is entirely loaded in memory each time a pixel is set with the new color (r,g,b).

The code which is optimized using byte buffer technique is presented in Code 2. `FilteredBitmap` is the class I have implemented which hides all the details. By using this technique, the image will be accessed only once for setting all pixels with the filtered values, considerably improving the execution time.

3 Conclusions

Filters can be combined in many ways to improve a subsequent OCR on text images. If there is a lot of noise in the image, a Blur filter is the proper solution, afterwards followed by Color filter. As in [4], this success rate of OCR also depends on whether the characters are broken or they touch each other.

Implementing the proper optimization technique for each filter depends on the particularities of the filters:

- The usage of a byte buffer is the technique with the largest applicability over image filters such as Invert, Color, or even Sharpen and Blur. The optimization is even greater as the filter perform a smaller number of computations, which is specific to the majority of image filters
- `ColorMatrix` is the most desired solution when the filter performs a lots of calculations such as Contrast filter and the calculations can be refactored and thus, reduced to a smaller and usually fixed number of computation involved when using `ColorMatrix`; this optimization cannot be applied for filters for which use a template to compute the filtered pixels

4 References

[1] T. Sato, T. Kanade, E.K. Hughes, M.A. Smith, *Video OCR for digital news archive*, Carnegie Mellon University, Pittsburg, United State, 1998

[2] "Bitmap Class", Microsoft, 2014, [Online: http://msdn.microsoft.com/en-us/library/system.drawing.bitmap(v=vs.110).aspx]

[3] "ColorMatrix Class", Microsoft, 2014, [Online: http://msdn.microsoft.com/en-us/library/system.drawing.imaging.colormatrix(v=vs.110).aspx]

[4] P. Stubberud[1], J. Kanai, V. Kalluri, "Adaptive image restoration of text images that contain touching or broken characters", [1]Nevada University, Las Vegas, United States of America, 1995