Chapter 16
Acceleration of Retinex Algorithm for Image Processing on Android Device Using Renderscript

Duc Phuoc Phat Dat Le, Duc Ngoc Tran, Fawnizu Azmadi Hussin and Mohd Zuki Yusoff

Abstract The popularity and availability of Android devices have motivated researchers to implement their image processing systems on Android mobile platform. Retinex is considered as an effective method to restore the image's original appearance and used as a pre-processing step in many computer vision applications. It would give a lot of benefits to implement Retinex on such portable system and optimize it for real-time performance. This is a challenge because of limited computational power and memory of the portable system. This paper presents an implementation of rsRetinex, an optimized Retinex algorithm by using Renderscript technique. The experimental results show that rsRetinex could gain up to five times speedup when applied to different image resolution.

Keywords Retinex · Image enhancement · Image restoration · Android device Renderscript

16.1 Introduction

Retinex is a very popular and effective method to remove environmental light interferences which is used as a pre-processing step such as contrast enhancement in many image processing systems. It was introduced by Land and McCann [1, 2] and has a lot of updates, improvements by different researchers. The original Land and McCann work [3] described four steps for each iteration of Retinex calculation: ratio, product, reset and average [4]. Based on how the comparison pixels are chosen, the Retinex algorithm can be separated into three classes: path-based

D. Phuoc Phat Dat Le (✉) · D. N. Tran · F. A. Hussin · M. Z. Yusoff
Centre for Intelligent Signal and Imaging Research Laboratory, Electrical and Electronics Engineering Department, Universiti Teknologi Petronas, 31750 Bandar Seri Iskandar, Tronoh, Perak, Malaysia
e-mail: datduc007@ieee.org

Retinex algorithms [2, 3, 5], recursive Retinex algorithm [6], and center/surround Retinex algorithm [7]. Retinex processing of color image can be applied separately in R, G, and B spectral bands and combined together to get the final output image. The consumption cost of Retinex algorithm is very high for large image, which is especially taxing on mobile device. Therefore, this paper proposes a Renderscript acceleration of McCann99 Retinex called rsRetinex to achieve the real-time improvement on Android device. The experiment result shows that rsRetinex can achieve better real-time performance compared with Java code while achieving at least 4–5 times performance gain.

The organization of the rest of this paper is as follows: Sect. 16.2 presents a brief introduction of Retinex algorithm and Renderscript programming model. Section 16.3 presents detail of the rsRetinex implementation. The experimental results are reported and discussed in Sect. 16.4. Finally the conclusion is given in Sect. 16.5.

16.2 Background

16.2.1 Retinex Algorithm

The idea of the Retinex was conceived by Land [2] as a model of lightness and color perception of human vision. The basic form of multi-scale Retinex (MSR) [8–10] is given by

$$R_i(x_1,x_2) = \sum_{k=1}^{n} W_k(\log I_i(x_1,x_2) - \log[F_k(x_1,x_2) \otimes I_i(x_1,x_2)]) \quad i = 1,\ldots,N \tag{16.1}$$

where R_i is the output of the MSR process at the coordinates (x_1, x_2), the sub index i represents the ith spectral band, N is the number of spectral band, $N = 1$ for grayscale images and $N = 3$ for a typical color image. So in normal case, i $=$ 3 I_i is the input image in the spectral band ith, F_k represents the Kth surround function, W_k are the weights associated with F_k, k is the number of surround functions, or scales, and $\otimes$ represents the convolution operator. The basic principles of Retinex are: color is obtain from three lightness computed separately for each of color channels; the ratios of intensities from neighboring locations are assumed to be illumination invariant; lightness in a given channel is computed over large regions based on combining evidence from local ratios; the location with the highest lightness in each channel is assumed to have 100 % reflectance within that channel's band. On the other hand, the Retinex computation of lightness at a given pixel is the comparison of the pixel's value to that of other pixels. One version of Retinex is McCann99 Retinex [11], which has been given standardized definitions in terms of Matlab code. This paper proposes rsRetinex, an optimized implementation of McCann99 in Android OS.

16.2.2 Renderscript Programming Model

Renderscript is a new programming framework and API for Android. Renderscript code is called during the runtime in native level and communicates with the Android Virtual Machine (VM) Dalvik, so the way a Renderscript application is set up is different from a pure VM application. An application that uses Renderscript is still a traditional Java application and runs in the VM, but the developers write the script and indicate which parts of the program requires to be run with Renderscript. The developers do not need to target the multiple architectures because it is platform independent which means that any device that supports Renderscript will run the code properly regardless of the architecture. Renderscript gives the applications the ability to run operations with automatic parallelization across all available processor cores.

16.3 rsRetinex Implementation

McCann99 has three main functions as shown in the following pseudo code segment. They are ComputeLayer, ImageDownResolution and CompareWithNeighbor.

```
program Retinex_McCann99(){
  Global OPE RRE Maximum
  nLayers = ComputeLayer (nrows, ncols);
  Maximum = max(InputImage(:)) ;
  OP = Maximum*ones([nrows, ncols]);
  For each layer, do
    RR = ImageDownResolution(InputImage,2^(nLayers -
layer));
    RRE = Padded RR with zero
    OPE = Padded OP with zero
  For iter = 1:nIterations
    CompareWithNeighbor();
  End
  NP = OPE(2:(end-1), 2:(end-1));
}
```

In rsRetinex implementation, two main functions have been chosen to enhance the performance of the McCann99 Retinex algorithm in Renderscript: ImageDownResolution and CompareWithNeighbor because of their expensive pixel-wise computation. ImageDownResolution function averages image data to make a multi-resolution pyramid from the input image. Each pixel in the input image will be divided into many blocks; the number of pixels depends on the block size. After that the entire pixel in one block will be averaged to make a new pixel in the new layers. This task is pixel-wise, and the number of calculation steps depend on the block size, the smaller the block size is the more calculation steps to be performed.

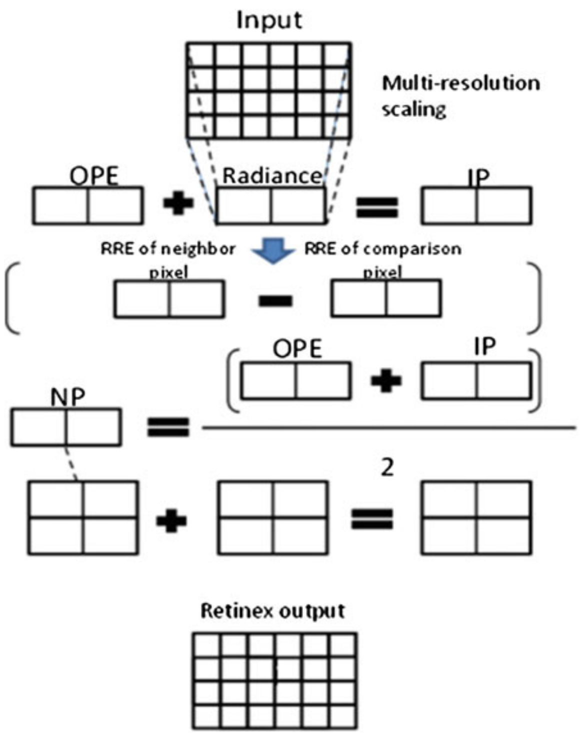
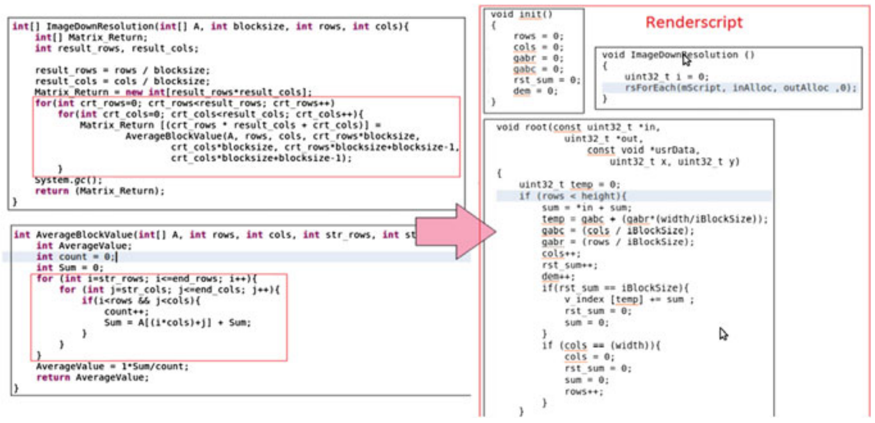


Fig. 16.1 The ratio-product-reset-average operation

By applying Renderscript technique to this function, the number of calculation steps will keep the same regardless the block size, because each pixel is calculated simultaneously. At each level, the new product will be calculated by visiting each of its eight neighboring pixels in clockwise order, which is implemented by the CompareWithNeighbor function. This operation calculates the New Product (NP) for the input pixel, for example (x′, y′). Figure 16.1 illustrates the ratio-product-reset-average operation. This operation starts with pixel (x, y) using the Old Product (OP). The entire OP is initialized with the maximum value for that wave band. It starts with subtracting the neighbor's log luminance called ratio step and then adding the result to the OP (the product step). If the radiance ratio, called Intermediate Product (IP), between pixel (x, y) and pixel (x′, y′) is greater than the maximum, the IP will be reset to the maximum (called reset step) and then averaged with the previous NP (called average step).

For each pixel in the current layer, the ratio-product-reset-average operation is performed on eight neighboring pixels in clockwise order. Because the Renderscript layer cannot access the memory directly, it needs to be allocated a memory location as a container to contain four data plane: OP, Radiance (RR), IP and NP by using allocation variables. In this implementation, only two allocation variables are used, inAlloc and outAlloc in order to reduce memory utilization. There are two main components in Renderscript code, init() and root(). The init() function is



```
int[] ImageDownResolution(int[] A, int blocksize, int rows, int cols){
    int[] Matrix_Return;
    int result_rows, result_cols;

    result_rows = rows / blocksize;
    result_cols = cols / blocksize;
    Matrix_Return = new int[result_rows*result_cols];
    for(int crt_rows=0; crt_rows<result_rows; crt_rows++)
        for(int crt_cols=0; crt_cols<result_cols; crt_cols++){
            Matrix_Return [(crt_rows * result_cols + crt_cols)] =
                    AverageBlockValue(A, rows, cols, crt_rows*blocksize,
                            crt_cols*blocksize, crt_rows*blocksize+blocksize-1,
                            crt_cols*blocksize+blocksize-1);
        }
    System.gc();
    return (Matrix_Return);
}
```

```
int AverageBlockValue(int[] A, int rows, int cols, int str_rows, int st
    int AverageValue;
    int count = 0;
    int Sum = 0;
    for (int i=str_rows; i<=end_rows; i++){
        for (int j=str_cols; j<=end_cols; j++){
            if(i<rows && j<cols){
                count++;
                Sum = A[(i*cols)+j] + Sum;
            }
        }
    }
    AverageValue = 1*Sum/count;
    return AverageValue;
}
```

```
void init()
{
    rows = 0;
    cols = 0;
    gabr = 0;
    gabc = 0;
    rst_sum = 0;
    dem = 0;
}
```

```
void ImageDownResolution ()
{
    uint32_t i = 0;
    rsForEach(mScript, inAlloc, outAlloc ,0);
}
```

```
void root(const uint32_t *in,
             uint32_t *out,
                const void *usrData,
                   uint32_t x, uint32_t y)
{
    uint32_t temp = 0;
    if (rows < height){
        sum = *in + sum;
        temp = gabc + (gabr*(width/iBlockSize));
        gabc = (cols / iBlockSize);
        gabr = (rows / iBlockSize);
        cols++;
        rst_sum++;
        dem++;
        if(rst_sum == iBlockSize){
            v_index [temp] += sum ;
            rst_sum = 0;
            sum = 0;
        }
        if (cols == (width)){
            cols = 0;
            rst_sum = 0;
            sum = 0;
            rows++;
        }
    }
```

Fig. 16.2 Code fragment of ImageDownResolution function

to initialize the local variable for the script. The same calculations to be performed on each pixel of the input image are described in root() function. When the rsForEach() function is invoked and passed the inAlloc as the input, Renderscript will automatically perform the calculation to all the pixels in the input and parallelize all the tasks across all available processor core, while the Java version only can perform sequentially by using a loop. Figure 16.2 shows the difference of ImageDownResolution implementations between the Java version with many loops and the loop-free Renderscript.

16.4 Experimental Result

To evaluate the performance of the proposed method, an experiment is performed on emulator, which settings are: ARM Cortex A9 1 GHz, RAM 1 GB, to compare the executing time between two implementations: java-only and Renderscript.

Figure 16.3 shows that with the 968 × 648 bitmap image, ImageDownResolution and CompareWithNeighbor function are invoked four times during execution. As shown in Fig. 16.3, the rsImageDownResolution gained around 1.6–6 times speedup for each run. The total execution time of rsImageDownResolution and jvImageDownResolution are 6,163 ms and 24,636 ms, which shows that rsImageDownResolution function gained almost four times. rsCompareWithNeighbor function gained around 4–25 times for each run. When applying rsRetinex to different image size, the system also achieved the significant performance improvement as shown in Table 16.1. With the 968 × 648 and 1,024 × 768 resolution, system speedups are about three and five times in real-time performance.

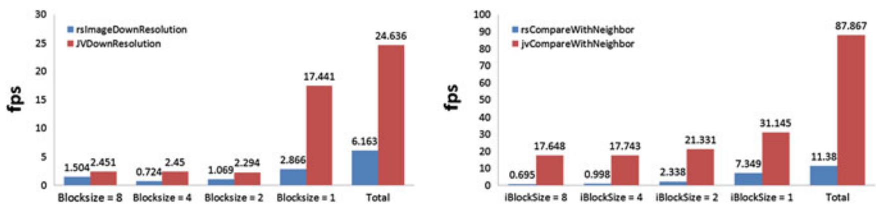


Fig. 16.3 Execution time of two parts in the Retinex algorithm applying to 968 × 648 bitmap image

Table 16.1 Experiment result with respect to image resolution

Image resolution	Java code execution time	rsRetinex execution time	Gain performance
284 × 177	18	4	4.5
290 × 439	43	10	4.3
340 × 148	17	3.8	4.4
968 × 648	150	32	4.6
1,024 × 768	446	121	3.68

16.5 Conclusion and Discussion

Building an image processing application on Android device is very useful because of mobility, portability and low cost. However the portable devices typically provide limited computational power, and memory capacity. In this paper, we have presented an optimization method using Renderscript. Two parts of McCann99 Retinex have been chosen for Renderscript implementation. The experimental results show that Renderscript can accelerate the McCann99 Retinex algorithm and rsRetinex gained at least 3–5 times speedup compared with the java-only implementation.

References

1. Ebner M (2007) Color constancy. Wiley, England, pp 143–153
2. Land E (1964) The Retinex. Amer Scient 52(2):247–264
3. Land EH, McCann JJ (1971) Lightness and Retinex theory. J Opt Soc Am 61:1–11
4. Land EH (1977) The Retinex theory of color vision. Sci Am 237:108–129
5. Land EH (1986) Recent advances in Retinex theory. Vision Res 26(1):7–21
6. Frankle J, McCann J (1983) Method and apparatus for lightness imaging. US Patent number 4384336
7. Land EH (1986) An alternative technique for the computation of the designator in the Retinex theory of color vision. Proc Nat Acad Sci 83:3078–3080

8. Wang Y-K, Huang W-B (2011) Acceleration of an improved Retinex algorithm. Computer vision and pattern recognition workshops (CVPRW). In: Conference on 2011 IEEE Computer Society, 20–25 June 2011, pp 72–77
9. Rahman Z, Jobson D, Woodell GA (1996) Multiscale retinex for color image enhancement. In: Proceedings of IEEE international conference on image processing, pp 1003–1006
10. Jobson DJ, Rahman Z, Woodell GA (1997) A multi-scale Retinex for bridging the gap between color images and the human observation of scenes. IEEE Trans Image Process Spec Issue Color Process 6(7):965–976
11. Funt B, Ciurea F, McCann J (2004) Retinex in Matlab. J Electron Imaging 13:48–57