J Real-Time Image Proc (2014) 9:407–425
DOI 10.1007/s11554-012-0301-6

SPECIAL ISSUE

A CUDA-enabled parallel algorithm for accelerating retinex

Yuan-Kai Wang · Wen-Bin Huang

Received: 6 March 2012 / Accepted: 6 November 2012 / Published online: 27 November 2012
© Springer-Verlag Berlin Heidelberg 2012

Abstract Retinex is an image restoration approach used to restore the original appearance of an image. Among various methods, a center/surround retinex algorithm is favorable for parallelization because it uses the convolution operations with large-scale sizes to achieve dynamic range compression and color/lightness rendition. This paper presents a GPURetinex algorithm, which is a data parallel algorithm accelerating a modified center/surround retinex with GPGPU/CUDA. The GPURetinex algorithm exploits the massively parallel threading and heterogeneous memory hierarchy of a GPGPU to improve efficiency. Two challenging problems, irregular memory access and block size for data partition, are analyzed mathematically. The proposed mathematical models help optimally choose memory spaces and block sizes for maximal parallelization performance. The mathematical analyses are applied to three parallelization issues existing in the retinex problem: block-wise, pixel-wise, and serial operations. The experimental results conducted on GT200 GPU and CUDA 3.2 showed that the GPURetinex can gain 74 times acceleration, compared with an SSE-optimized single-threaded implementation on Core2 Duo for the images with 4,096 × 4,096 resolution. The proposed method also outperforms the parallel retinex implemented with the nVidia Performance Primitives library. Our experimental results indicate that careful design of memory access and multithreading patterns for CUDA devices should acquire great performance acceleration for real-time processing of image restoration.

Y.-K. Wang (✉) · W.-B. Huang
Department of Electrical Engineering, Fu Jen Catholic University, 510, Zhongzheng Rd., Xinzhuang Dist., New Taipei City, 24205, Taiwan
e-mail: ykwang@mails.fju.edu.tw

Keywords Center/surround retinex · GPGPU · CUDA · Parallelization · Image restoration

1 Introduction

Retinex is an effective method for removing environmental light interference and has been used as a preprocessing step in many image and video applications, such as video compression [1], high dynamic range imaging [2], face recognition [3], and multisensor image fusion [4]. Retinex is a nonlinear mechanism that can achieve color constancy by simulating human color perception, which can adapt to varying light conditions and improve the visual quality of a perceived image. Land [5, 6] first proposed a model of the lightness and color perception of human vision. Various extensions of retinex have been developed and can be categorized into three classes: path-based algorithms, recursive algorithms, and center/surround algorithms.

Among the three classes, the center/surround retinex algorithm has no iterative process and is suitable for parallelization. The single-scale retinex (SSR) [7] algorithm is the first proposed center/surround algorithm that can either achieve dynamic range compression or color/lightness rendition, but not both. Both dynamic range compression and color/lightness rendition are achieved by multiscale retinex (MSR) [8] and multiscale retinex with color restoration (MSRCR) [9, 10] algorithms by combining small and large center/surround information. The computation of these retinex algorithms includes center/surround information extraction and log-domain processing, which have high time complexity.

The computation of image processing such as the center/surround retinex is intensive and time-consuming. Traditional sequential processing methods are often unable to

gain real-time performance. The multicore CPU, digital signal processor (DSP), field programmable gate array (FPGA), and general-purpose graphics processing unit (GPGPU) are hardware solutions to improve the performance of parallel processing. Using a multicore CPU involves applying large-core technologies with pipelines and hyper-threads to achieve instruction-level parallelism, but multicore CPU architecture is inappropriate for demanding computation applications. DSP programs, which are also parallelized at the instruction level, are effective for portable and mobile multimedia applications because of the low power consumption of DSPs; however, the level of their giga floating-point operations per second (Gflops) performance is inadequate for highly demanding applications. The FPGA involves adopting a customized hardware circuit to achieve superior parallelism, higher efficiency, and greater Gflops performance, compared to using multicore CPUs and DSPs.

Graphics processing units (GPUs) have traditionally been used to execute only graphics applications, and developing parallel processing algorithms on this platform is extremely difficult. In recent years, GPUs have evolved into GPGPUs and have been used as sources of massive computing power. The multithreads of GPGPUs and many-core processors are especially well suited for data parallel computation [11]. The power consumption and Gflops performance of GPGCUs are comparable with those of the FPGA. Cope et al. [12] systematically compared the FPGA and GPGPUs, finding that the FPGA outperforms GPGPUs for algorithms requiring large numbers of regular memory access, whereas GPGPUs are superior for algorithms with variable data reuse. Compute Unified Device Architecture (CUDA) is a software platform that can be used to achieve fine-grained parallelism in a GPGPU using Single Instruction Multiple Threads (SIMT). Developing GPGPU programs becomes more flexible and efficient because of their easy programmability when accelerating image processing in program-level parallelism.

However, designing parallel algorithms on GPGPUs for image processing is still challenging, especially in organizing thread hierarchy and using memory hierarchy to efficiently compute image tasks such as pixel- and block-wise operations. A GPGPU can accommodate hundreds or even thousands of threads being executed in parallel, but a massive multithreading pattern designed without considering constraints and latency degrades performance. In addition, register, cache, and global memory spaces in a GPGPU constituting a heterogeneous memory subsystem are exposed to program access for the flexibility of data parallel design. The performance of a parallelized image processing task is highly sensitive to the irregularity in access patterns of the heterogeneous memory subsystem, because memory access can be serialized if massive threads attempt to issue reads or writes without optimized memory access organization.

This paper proposes a data parallel algorithm, GPURetinex, to parallelize an improved MSRCR on a GPGPU. The GPURetinex includes a parallel contrast stretching method to the MSRCR for enhancing more challenging images with a high dynamic range, low key, and uneven illumination. The proposed method entails three stages: a Gaussian blur for center/surround information extraction, log-domain processing for color rendition and dynamic range compression, and alpha-trimmed contrast stretching to enhance contrast.

The three stages present three types of concerns for data-parallel design: block-wise, pixel-wise, and serial operations. A large convolution function in the Gaussian blur is necessary for extracting center/surround information, which is a block-wise operation and is extremely time consuming. Computing of the Gaussian blur is therefore approximated by separable convolution functions to reduce computing load and internal data transfer. The log-domain processing stage performs pixel-wise operations, which are inherently parallel problems that are easy to be parallelized. The serial calculation of histograms in the alpha-trimmed contrast stretching stage is inappropriate for GPGPU multithreading. A parallel reduction method [13] using a shared memory model is adopted.

This study proposes mathematical models of memory access patterns and multithreading patterns for the three types of operations to optimize the GPURetinex algorithm. Five different memory spaces in a GPGPU, including global memory, shared memory, texture cache, constant cache, and registers, are all exploited to place processing data appropriately, based on the distinct characteristics of each memory space and memory access pattern of the algorithm. The multithreading pattern of the GPURetinex, designed by considering SIMT characteristics and memory constraints of CUDA devices, is developed to improve the efficiency of parallelization.

The rest of this paper is organized as follows: Sect. 2 reviews recent works of parallel image processing that have used GPGPU and center/surround retinex algorithms. Section 3 presents the computational model of the proposed GPURetinex method. The data-parallel design with mathematical models of memory access patterns and massive multithreading is explained in Sects. 4 and 5. Section 6 provides implementation details of the GPURetinex algorithm based on the mathematical models. Experimental results are reported and discussed in Sect. 7. Finally, the conclusion of this study is provided in Sect. 8.

2 Related work

Two topics are reviewed in this section. The characteristics, advantages, and status of GPGPUs for improving

image processing algorithms are first presented. A unified view of current center/surround retinex methods is then provided.

2.1 Parallel image processing by GPU and GPGPU

There were many earlier works on GPU computing for image processing and computer vision before the born of GPGPU/CUDA. A fragment shader was used by Moreland and Angel [14] to compute fast Fourier transform on GPU and they gained 4 times faster, compared to CPU implementation. Strzodka and Garbe [15] presented a motion estimation algorithm with optical flow and achieved 2.8 times acceleration on a GeForce 5800 Ultra GPU than on an optimized CPU version. Shen et al. [16] implemented color space conversion for MPEG video encoding on GPU using DirectX to achieve 2–3 times acceleration. Many parallel algorithms for computer vision have been studied in GPU4Vision [17], such as real-time optical flow and total variation-based image segmentation.

Because a GPU is not a general-purpose and easily programmed platform, some open source libraries have been developed to aid the development of GPU algorithms. OpenVidia [18] provided a framework for video input and display. In addition, the implementation of feature detection and tracking, skin tone tracking, and real-time blink detection was also provided in OpenVidia. GpuCV [19] was designed to provide GPU acceleration with OpenCV interfaces. Another open source library called MinGPU [20] provided a set of useful image processing and computer vision functionalities.

GPGPU has been recently developed to enable developing data parallel algorithms for general-purpose applications. CUDA is a programming framework developed for nVidia GPGPUs. Because of its ease in programmability, GPGPU/CUDA has been increasingly adopted recently to accelerate computationally intensive tasks in image processing and computer vision domains. This study conducted a literature review of GPGPU parallelization in image processing and computer vision. Owens et al. [21] conducted a broad survey of general-purpose computation on graphics hardware. A parallel Canny edge detector under CUDA was demonstrated [22], including all algorithm stages. They achieved 3.8 times acceleration on the images of 3,936 × 3,936 resolution compared to an optimized OpenCV version running on a PC. A real-time 3D visual tracker with efficient Sparse-Template-based particle filtering was implemented by Lozano and Otsuka [23]. Their implementation can achieve 10 times performance improvements, compared to a similar CPU-only tracker. Accelerated wavelet-based image denoising was demonstrated by [24] with a speed improvement of 7.9 times. Parallelizing the boosting algorithm for object detection was proposed by [25], and it can outperform an optimized Core2 Duo CPU of 2.6 GHz by approximately 6–8×. A hybrid system sharing workloads between a CPU and a GPGPU was proposed by [26] for stereo vision and optical flow, and achieved a speed of up to 90 times.

Most previous implementations have not achieved superior acceleration because the parallel design of image processing requires exploiting hardware support for the principle of locality, and expresses the algorithm in a highly parallel fashion. A CUDA device is a GPGPU consisting of a set of multicore processors, referred to as streaming multiprocessors (SMs). Streaming processors (SPs) within a SM work in a single-instruction multiple data (SIMD) fashion. Although hundreds of cores in a GPGPU can run in parallel to improve the performance of the algorithm, a hierarchical memory structure with memory nearly as fast as a multiprocessor has to be exploited to exhibit spatial and temporal locality in particular programs. Off-chip memory has a much slower bandwidth but a larger capacity than that of on-chip memory. High-performance algorithms must exploit the characteristics of the non-uniform memory subsystem using as much fast memory and as little slow-access memory as possible.

Five memory spaces exist in CUDA: global, texture, constant, shared, and register memory. Global memory space, which is off-chip memory with a large capacity, has a high latency, typically two orders of magnitude greater than the accesses to on-chip memory. Despite its usage being the most flexible, its performance varies substantially depending on the associated memory access pattern. Data with a locality property accessed by global memory are expensive because the memory is not cached. Using global memory should be restricted to fully coalesced access. Coalescing groups of reads or writes of multiple data items into one operation improves memory bandwidth and reduces overhead.

Cache memory spaces including texture and constant caches store a copy of recently used data, removing the need for a fetch from off-chip if those data are required again. They are programmable using CUDA programs. Accessing constant and texture memory spaces is as fast as accessing registers on cache hits. Both types of cache memory space are read-only, and provide highly efficient memory accesses with locality properties. Constant cache can only accommodate a small number of constants for non-modifiable input data, but texture cache has plenty of capacity (as much as global memory) because it is bound to global memory.

Shared memory and register memory are types of on-chip memory. Accessing shared memory is as fast as accessing registers, as long as no bank conflict exists. Placing data in shared memory can improve the

performance significantly, especially when the data exhibit locality. Registers are on-chip resources that are equally divided across active threads in each multiprocessor. Each multiprocessor has a fixed number of registers, and excessive use of registers in a kernel limits the number of threads that can run simultaneously, thus exposing memory latency.

The parallel design in CUDA is based on the SIMT model. Three levels of parallel granularity are in CUDA: thread, warp, and thread block. A thread processes a single datum of a data stream, such as an image pixel. The SIMT simultaneously executes 32 parallel threads called "warps," which are batched in groups called "blocks." A thread block can be executed only on a single SM. However, time slicing enables a single SM to execute multiple blocks simultaneously. Each thread is executed independently with its own instruction address and register state. Threads within a block can be executed simultaneously and communicate with one another through shared memory space.

The warp is the fundamental unit of instruction dispatch within a single SM. When a multiprocessor is given one or more thread blocks to execute, it partitions them into warps and each warp is scheduled by a warp scheduler for execution. Warps of a block are concurrently executed by time slicing. Each SM manages a pool of warps, totaling thousands of threads. Individual threads composing a warp are of the same type and begin simultaneously at the same program address, but they are otherwise free to branch and be executed independently.

Application acceleration is highly dependent on being able to improve memory bandwidth and simultaneously ensure that all cores within the multiprocessors are busy. Enhancing the memory efficiency of applications on data-parallel architectures requires analyzing the characterization of memory access patterns in nested loops of the algorithm. Massive multithreading can maximize the use of all cores within the multiprocessors. However, resource constraints are the barrier to the increase of threading, because more threads may decrease performance because of resource contention. The algorithmic memory selection method in [27] can optimize the use of the non-uniform memory subsystem on nVidia GPGPUs mathematically.

2.2 A unified view of center/surround retinex algorithms

The path-based retinex algorithms are based on the multiplication of ratios between pixel values along a set of random paths in an image. Brainard and Wandell [28] examined the convergence properties, and a further development using Brownian motion [29] introduced a randomly distributed path. The primary drawbacks of path-based algorithms are high computational complexity, and too many parameters necessary to achieve satisfactory image restoration results. However, computational dependency in the path geometry causes it to become an inherently serial algorithm.

The recursive retinex algorithm, first developed by Frankle and McCann [30], is a 2D extension of the path-based version and replaces the path computation by performing a recursive matrix comparison [30–32]. The recursive retinex algorithm computes the ratios and products with long-distance iterations between pixels first, and then progressively moves to short-distance interactions. This algorithm is computationally more efficient than the path-based class. However, the recursive approach must be executed serially and cannot be parallelized.

Land [33] first proposed the center/surround retinex algorithm. Contrary to the sequential and iterative manners of the two classes, this new technique allows each pixel to be treated separately. New pixel values are obtained by computing the ratio between each treated pixel and a weighted average of its surrounding area. This new implementation suggests that the center/surround information can be computed from a blurred version of the input image. Thus, Rahman et al. [7] proposed the SSR that uses a Gaussian blur operation to compute the center/surround information. The MSR [8] is an extension of SSR and combines dynamic range compression and color/lightness rendition by weighting three SSRs of different spatial scales. In [9, 10], the MSRCR was proposed to compensate for the inherent loss of color using a color restoration factor.

The center/surround retinex algorithms are faster than those that are path-based, and the amount of parameters is greatly reduced. It has lower computational cost and overcomes the deficiency of the recursive retinex algorithm. In addition, the retinex algorithms of the path and recursive classes cannot be parallelized effectively because of the algorithmic characteristics of high data dependency among sequential steps, whereas the retinex algorithm of the center/surround class can be greatly parallelized. Therefore, the center/surround retinex algorithm is favorable for GPGPU/CUDA implementation.

The three center/surround algorithms are explained in a unified view. The MSRCR algorithm combines small center/surround information and large center/surround information. In addition, it adopts a color adaptation mechanism to improve color/lightness rendition of images that contain gray-world violations. The basic form of MSRCR is given in (1).

$$R_i(x,y) = C_i(x,y) \times \sum_{z=1}^{Z} W_z(\log I_i(x,y) - \log[F_z(m,n) I_i(x,y)]),\ i \in \{R,G,B\}, \quad (1)$$

where $R_i(x,y)$ is the MSRCR output at the coordinates (x, y) in the ith spectral band of RGB color images, $I_i(x, y)$ the

original image, F_z the zth center/surround function, W_z the weight of F_z, Z the number of center/surround functions, and "⊗" denotes a convolution operation. The center/surround function F_z (m, n) is given in (2).

$$F_z(m,n) = K e^{\frac{-(m^2+n^2)}{v_z}} \tag{2}$$

where v_z is the scale of the zth Gaussian center/surround function, and K is a constant satisfying the constraint $\iint F(m,n)\,\mathrm{d}m\mathrm{d}n = 1$. A smaller value of v_z leads to narrower surround, and a larger value of v_z leads to wider surround. Small scale plays a role in dynamic range compression, whereas large scale contributes to color/lightness rendition. The large scale can approach to the image size and is computationally intensive.

Equation (1) uses multiple center/surrounds and different weights to achieve a graceful balance between dynamic range compression and tonal rendition. A combination of three scales (i.e., $Z = 3$, representing narrow, medium, and wide center/surrounds) is usually adopted to achieve both dynamic range compression and tonal rendition simultaneously. The color restoration factor $C_i(x, y)$ in (1) is considered to offer color constancy and is given in (3).

$$C_i(x,y) = \beta \times \log\left(\gamma \times \frac{I_i(x,y)}{\sum_{i\in\{R,G,B\}} I_i(x,y)}\right) \tag{3}$$

where $C_i(x, y)$ is the color restoration coefficient in the ith spectral band, β is a gain constant, and γ controls the strength of nonlinearity. The MSRCR can provide synthesized color constancy, dynamic range compression, enhancement of contrast and lightness, and favorable color rendition.

The $R_i(x, y)$ becomes an MSR output if the color restoration factor $C_i(x, y)$ is not considered. The MSR can combine the dynamic range compression and color rendition, but it has drawbacks for color images [34]. Furthermore, if not only $C_i(x, y)$ is not considered but also the number of center/surround function n is 1, then (1) degenerates to be SSR. The SSR can only provide dynamic range compression on a smaller scale that makes the edge information become obvious. Conversely, when the larger scale is adopted, the SSR can only provide color rendition that increases color information. The SSR cannot simultaneously retain edge and color information.

The MSRCR is computationally the most intensive of the three center/surround retinex algorithms because it usually adopts three Gaussian blur convolutions on a large scale to compute the center/surround information in each spectral band.

The recent development of retinex has been devoted to the real-time improvement of performance. [35, 36] implemented the McCann and MSR algorithms using CUDA and obtained 5.4 times improvement. Our preliminary studies [37, 38] demonstrated the feasibility of high-performance throughput for the computing of MSRCR using GPGPU. However, the two implementation papers reported neither mathematical analysis nor exhaustive experiments for the results. This paper presents the theoretical design and complete experimental results for the GPURetinex.

3 Computational model of the algorithm

This section presents the proposed GPURetinex algorithm, which is an algorithmic improvement of MSRCR with a data-parallel design. The GPURetinex algorithm involves using the Gaussian convolution with a large kernel size to obtain the center/surround information. The 2D Gaussian filter is circularly symmetric, and can be approximated and separated into two 1D Gaussian filters: row filter r_z and column filter c_z. That is, $F_z = r_z \otimes c_z$. To convolve an image with separable filters, we first convolve each row of the image using c_z, resulting in an intermediate image. We then convolve each column of this intermediate image using r_z. The resulting image approximates the direct convolution of the original image and the 2D filter. Benefit of adopting the separable filters is the reduction of computation time. The convolution of an $M \times M$ image using an $N \times N$ filter kernel requires a time proportional to M^2N^2. However, the separable convolution only requires a time proportional to M^2N.

Equation (1) is decomposed into two operations because there are two types of computations: block-wise and pixel-wise. The two operations, Gaussian blur and log-domain processing, are therefore rewritten as follows:

$$G_{iz}(x,y) = F_z(m,n)I_i(x,y) \cong r_z(m,n)c_z(m,n)I_i(x,y), \tag{4}$$

$$R_i(x,y) = C_i(x,y) \times \sum_{z=1}^{Z} W_z(\log I_i(x,y) - \log G_{i,z}(x,y)), \tag{5}$$

However, the retinex output $R_i(x, y)$ may not improve the best color/lightness rendition and contrast, especially for images of high dynamic range, low-key, and uneven illumination. A statistically robust method, alpha-trimmed contrast stretching, is followed to improve its result:

$$O_i(x,y) = T(R_i(x,y)),$$
$$T(a) = \begin{cases} 0, & \text{if } a \le \hat{a}_{\min} \\ (a - \hat{a}_{\min}) \times \frac{255}{\hat{a}_{\max} - \hat{a}_{\min}}, & \text{if } \hat{a}_{\min} < a < \hat{a}_{\max} \\ 255, & \text{if } a \ge \hat{a}_{\min} \end{cases} \tag{6}$$

The contrast stretching uses the histogram of $R_i(x, y)$ to find the two extreme values $\hat{a}_{\min}$ and $\hat{a}_{\max}$ in a statistically robust manner before stretching is adopted. Let $h(i)$ and $H(i)$ be the histogram and cumulative histogram of $R_i(x, y)$ and be defined as follows:

$$h(i) = \frac{1}{N}\sum_{x,y}\delta(R(x,y),i), \quad 0 \le i \le L-1, \tag{7}$$

$$H(i) = \sum_{J=1}^{I} h(j), \tag{8}$$

where N is the number of pixels in $R_i(x,y)$, $\delta(\cdot,\cdot)$ is the Kronecker delta function, and L is the number of quantization levels. Then $\hat{a}_{\min}$ is statistically obtained by a predefined quantile α_{low} that is defined as the percentage of pixels with gray levels less than $\hat{a}_{\min}$. $\hat{a}_{\max}$ is also defined similarly with a quantile α_{high}. The values $\hat{a}_{\min}$ and $\hat{a}_{\max}$ are obtained as shown in (9) and (10):

$$\hat{a}_{\min} = \min\{i | H(i) \ge N \cdot \alpha_{\text{low}}\}, \tag{9}$$

$$\hat{a}_{\max} = \max\{i | H(i) \le N \cdot (1 - \alpha_{\text{high}})\}, \tag{10}$$

where $0 \le \alpha_{\text{low}}, \ \alpha_{\text{high}} \le 1$, and $\alpha_{\text{low}} + \alpha_{\text{high}} \le 1$. The upper and lower quantiles can be set to the same value (i.e., $\alpha_{\text{low}} = \alpha_{\text{high}} = \alpha$) with $\alpha \in [0.005, 0.015]$.

An illustration of the algorithm is shown in Fig. 1. The example image presents a low-key image of an indoor scene with non-uniform illumination at night. The low-key image contains a high dynamic range and a considerably dark region. One person is standing in the dark region, and it is not easy to perceive this person. Fig. 1c, e show the enhancement images of MSRCR and the modified MSRCR accompanied by alpha-trimmed contrast stretching. The person in the dark region becomes visible, and the details of the person are discernible in Fig. 1e.

The computational model provides the algorithmic aspect of the modified retinex. Equations (4)–(10) can be divided into three stages: Gaussian blur, log-domain processing, and alpha-trimmed contrast stretching. Equation (4) performs Gaussian blur and includes two convolution steps: row filtering and column filtering. The second stage log-domain processing is described in (5). The contrast stretching defined by (6)–(10) contains histogramming and stretching steps. A data-parallel design for the modified retinex algorithm, the GPURetinex algorithm, is proposed to improve its performance using GPGPU.

Additional analyses on the parallelization of (4) and (5) are necessary because the two stages cost 98.4 % of the execution time of the modified retinex. Gaussian blur and log-domain processing constitute an inherently parallel problem. Operations in these two stages can be parallelized at a fine-grained level and run as the SIMT kernels on CUDA. They are parallelized by distributing image data among concurrent threads, and then each thread runs the same kernel on its share of the data. However, the distribution of image and processing data to various CUDA memory spaces should greatly affect the performance because data transfer between low-speed memory and a multiprocessor costs time. Carefully designing data placement according to memory and algorithmic characteristics (i.e., a memory access pattern) is necessary. The multithreading pattern of these two stages is another performance obstacle to be considered. Although massive threading with thousands of threads in a multiprocessor is permissible in CUDA, a higher threading size may degrade the use of computing. Multithreading pattern according to hardware constraints in warp, register, and shared memory must be analyzed. The next two sections provide additional details of the data-parallel design of the algorithm according to the maximal use of memory access and multithreading patterns.

4 Analysis of memory access pattern

The memory access patterns of the three stages are analyzed using nested loop and array access. Array access in a specific nested loop exhibits one or many of six access patterns: linear, coalescing, temporal reuse, same-address-read, and chunkable. Proper selection of memory spaces for each data array can be achieved by analyzing its access patterns. We adopt the mathematical models and memory selection algorithm presented in [27] to map our data arrays to the most appropriate memory spaces in the CUDA device.

Although many nested loops are in (3)–(11), we analyzed only the most critical loops from (4), (5), and (7). Their loop structures are shown in Table 1. Equation (4) performs block-wise operations. The two filtering steps shown in Table 1a, b exhibit a loop structure of Depth 3. Their memory access patterns are analyzed individually because the two nested loops show different access characteristics for data arrays of I, I', r, and c. Table 1c includes a loop structure of Depth 3 for the pixel-wise operation in log-domain processing. The data arrays of W, I, G, R, and C correspond to the W_z, I_i, G_{iz}, R_i, and C_i in (5). They must be placed on difference memory spaces. The sequential loop operation of histogramming in listed Table 1d is a loop of Depth 2 and reveals the data dependency of array access and an irregular access pattern on the data array h, which is defined in (7). Data racing occurs when the two concurrent threads. The loops in the remaining equations are not analyzed because they are pixel-wise operations and have the same loop structure using (5).

The mathematical definition of an access pattern is described here. Assume that a loop of depth D accesses a P-dimensional array. We define the memory access pattern of the array in the loop as a memory access vector $\vec{p}$, which can be decomposed into the affine form: $\vec{p} = \boldsymbol{P}\vec{i} + \vec{o}$, where $\boldsymbol{P}$ is a memory access matrix of size PxD, $\vec{i}$ is an

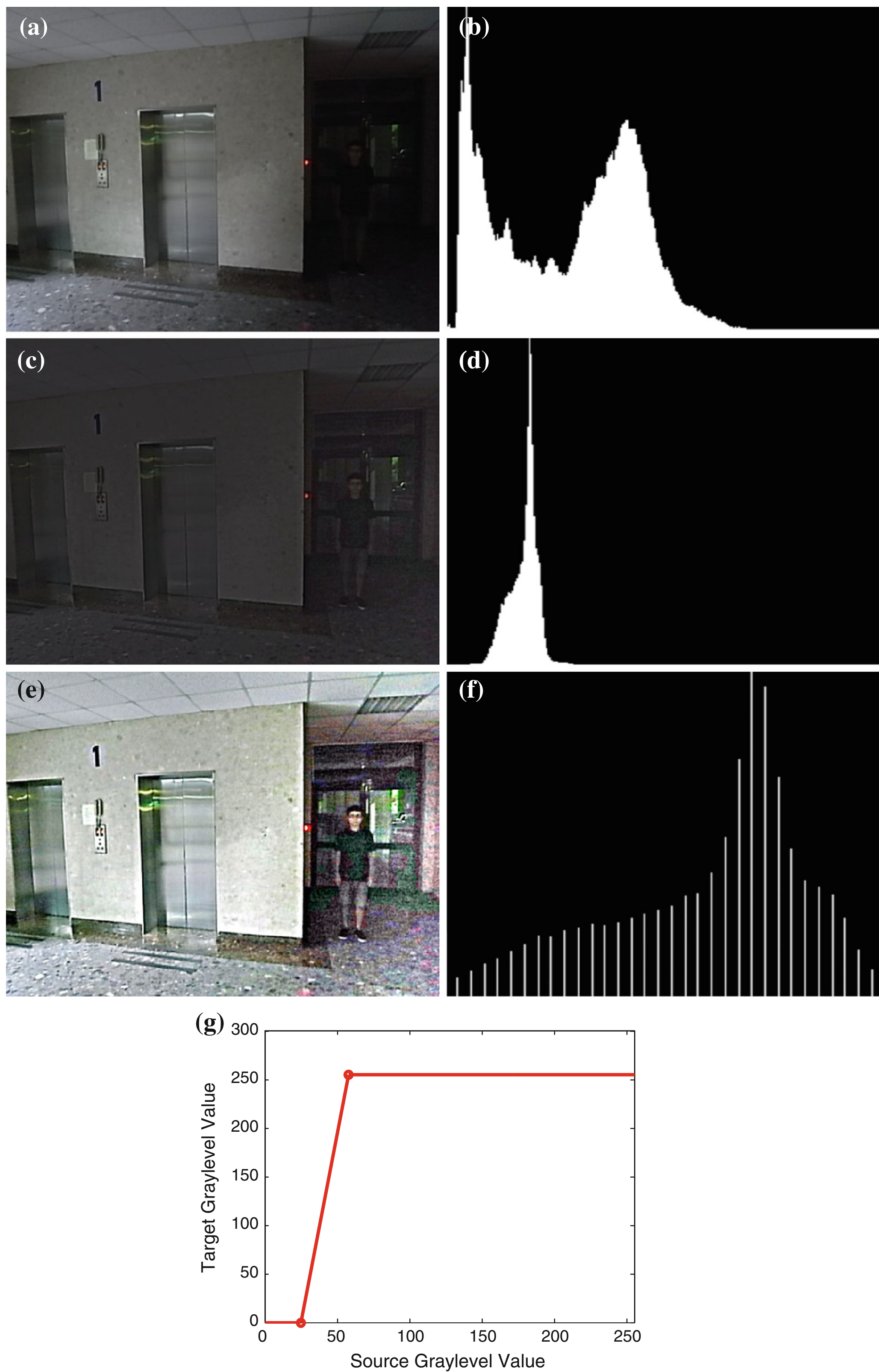


Fig. 1 Image enhancement using the two algorithms with $W_z = 1/3$, 1/3, 1/3, $v_z = 14, 69, 210$, and $\alpha = 0.015$ for the image size of $4{,}096 \times 4{,}096$. **a**, **b** The source image and its histogram, **c**, **d** MSRCR result and its histogram, **e**, **f** GPURetinex result and its histogram, **g** transfer function in the alpha-trimmed contrast stretching

Table 1 Pseudo codes of nested loops of (a) row filtering: $I' = I \otimes r$, (b) column filtering: $G = I' \otimes c$, (c) log-domain processing, (d) histogramming

(a)
```
for (int x=0; x<M; x++)
  for (int y=0; y<M; y++)
    for (int z=-s/2; z<s/2; z++)
      I'[x][y] = I[x][y+k] * r[z];
```

(b)
```
for (int x=0; x<M; x++)
  for (int y=0; y<M; y++)
    for (int z=-s/2; z<s/2; z++)
      G[x][y] = I'[y+z][x] * c[z];
```

(c)
```
for (int x=0; x<M;  x++)
  for (int y=0; y<M; y++)
  {  float temp = 0;
     for (int z=1; z<=Z; z++)
       temp = temp+W[z]*(log(I[x][y])–log(G[x][y][z]));
     R[x][y] = C[x][y] * temp;
  }
```

(d)
```
for (int i=0; i<Bins; i++)
  h[i] = 0;
for (int x=0; x<M;  x++)
  for (int y=0; y<M; y++)
    h[ R[x][y] ] ++;
```

The image size is $M \times M$ and filter size is s

iterator vector of size D iterating from the outermost to innermost loop, and $\vec{o}$ is an offset vector that is a column vector of size P that determines the starting point in a particular array.

The loops in Table 1(a), (b), and (c) display the nested loops of $D = 3$ for $P = 1$ and 2, as well as the nested loop in Table 1(d) of $D = 2$ for $P = 1$ and 2. Table 2 shows the access patterns and memory selections of the four loops. The first column in Table 2 shows the name of CUDA kernels corresponding to the four loops shown in Table 1. The second column presents the affine form of each data array, which is directly obtained by applying the mathematical definition to Table 1. Extracted from the affine form of each array, a pair of $(P, \vec{o})$ can represent the memory access pattern of the array.

For example, the input image I in row filtering has the affine form $\vec{m}_I$ and the memory access pattern $(\boldsymbol{P}_I, \vec{o}_I) = \left(\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 1 \end{bmatrix}, \begin{bmatrix} 0 \\ 0 \end{bmatrix} \right)$. The first two columns of $\boldsymbol{P}_I$ exhibit an inter-thread pattern and the third column intra-thread pattern. The inter-thread pattern $\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$ shows a linear memory access, which is a necessary condition for coalesced access in global memory. The intra-thread pattern $\begin{bmatrix} 0 \\ 0 \end{bmatrix}$ reveals temporal reuse; that is, a temporal locality property that is appropriate for selecting cache and shared memory spaces. The zero vector $\vec{o}_I$ means that no shifted access is in the array. Based on these properties combined with the knowledge that the image I is read-only and is of a large size, the memory selection algorithm in [27] suggests choosing texture memory for optimal performance. Therefore, access patterns of arrays can be directly obtained from the $(P, \vec{o})$ pair. From these characteristics, the proper selection of memory spaces can be achieved.

The third column of Table 2 shows the memory selection results for all arrays in Table 1. The three arrays, I in row filtering, I' in column filtering, and G in log-domain processing, are placed in texture memory because their patterns exhibit read-only and linear access with temporal locality. The three arrays, r in row filtering, c in column filtering, and W in log-domain processing, are placed in constant memory because they show the pattern of read-only, small, and same-address-read access. Global memory is chosen for six arrays because these arrays are either output data or linearly coalesced data with single access. Finally, shared memory is applied to the histogram array h because it exhibits a non-coalesced random access pattern.

In the parallel Gaussian blur operation, the texture memory and constant memory are used to improve efficiency because the readings of read-only data, including images I_i, I', and Gaussian blur functions r_z and c_z, obey the principle of temporal locality. Although both cache and

Table 2 Summary of memory access patterns and memory selection results

Kernel	Affine forms	Memory selections
Row filtering	$\vec{p_I} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 1 \end{bmatrix} \vec{i} + \begin{bmatrix} 0 \\ 0 \end{bmatrix}$	I: Texture
	$\vec{p_{I'}} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix} \vec{i} + \begin{bmatrix} 0 \\ 0 \end{bmatrix}$	I': Global
	$\vec{p_r} = [0 \quad 0 \quad 1] \vec{i} + [0]$	r: Constant
Column filtering	$\vec{p_{I'}} = \begin{bmatrix} 0 & 1 & 1 \\ 1 & 0 & 0 \end{bmatrix} \vec{i} + \begin{bmatrix} 0 \\ 0 \end{bmatrix}$	I': Texture
	$\vec{p_G} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix} \vec{i} + \begin{bmatrix} 0 \\ 0 \end{bmatrix}$	G: Global
	$\vec{p_C} = [0 \quad 0 \quad 1] \vec{i} + [0]$	c: Constant
Log-domain processing	$\vec{p_I} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix} \vec{i} + \begin{bmatrix} 0 \\ 0 \end{bmatrix}$	I: Global
	$\vec{p_G} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \vec{i} + \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix}$	G: Texture
	$\vec{p_C} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix} \vec{i} + \begin{bmatrix} 0 \\ 0 \end{bmatrix}$	C: Global
	$\vec{p_R} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix} \vec{i} + \begin{bmatrix} 0 \\ 0 \end{bmatrix}$	R: Global
	$\vec{p_W} = [0 \quad 0 \quad 1] \vec{i} + [0]$	W: Constant
Histogramming	$\vec{p_R} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \end{bmatrix}$	R: Global
	$\vec{p_h} = [\varepsilon \quad \varepsilon] \begin{bmatrix} x \\ y \end{bmatrix} + [0]$	h: Shared

The iteration vector $\vec{i}$ represents $[xyz]^T$. ε is a random number

shared memory spaces can provide the same shared access mechanism, cache memory is selected because of its optimization for read-only operation. Moreover, the access patterns of I and I' are different. Despite both having temporal reuse, I exhibits a linear access but I' non-unit stride access. The results of Gaussian blur G_{iz} are stored in global memory because of the capacity limitation of shared memory. For parallel log-domain processing, I_i, G_{iz}, and R_i are all read from and written to global memory using a coalescing technique. In the final step of parallel contrast stretching, global memory and shared memory are used to complete the computation of parallel histogramming.

The optimality of the memory selection results is demonstrated by conducting an experiment on row filtering with all possible combinations of memory selections. The memory space of output data array I' is fixed to global memory. The source image I can be placed on global or texture memory, but not shared and constant memory because of capacity limitation. The memory space of row filter r is flexible and has four choices. A combination of eight memory selections is tested for a 4,096 × 4,096 color image to obtain the execution time of the eight selections, as shown in Fig. 2. The experiment is performed on the Tesla 1060 GPGPU platform.

The experimental results in Fig. 2 show that the most favorable memory spaces for the I and r in row filtering are texture and constant caches, which matches the mathematical analysis presented in Table 2. The worst selection is global memory for both I and r. The improvement ratio is 5.45 for the worst case over the best. It means that the careful design of memory spaces can achieve more than five times improvement in the same GPGPU platform. The group of I being texture memory performs more favorably than the group of I being global memory. The result is reasonable because of the temporal reuse of source image pixels, and texture memory has the most effective support for the access of locality. Storing both I and r in texture memory could be a simple scheme with low execution time, but the experiment showed that execution time is improved when the convolution data r are stored in constant memory. This is mainly because the image is stored in texturing memory and is heavily accessed; therefore, offloading the access of the convolution data to constant memory relieves a bottleneck of the system.

5 Multithreading pattern design

Maximizing the use of the computing power in GPGPU requires increasing as much thread-level parallelism as possible and efficiently mapping this parallelism to multiprocessors to keep them busy. It is critical to parallelize threads and blocks with balanced computing applications across the multiprocessor that maximizes hardware use. Block size is defined as the number of threads within a

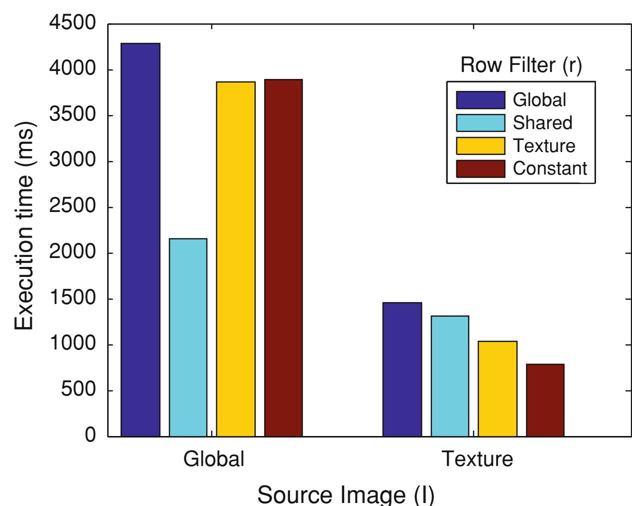


Fig. 2 Execution times of eight combinations of memory selections for row filtering

thread block. A large block size indicates that more threads are parallelized to maximize the use of computing power, but is inefficient when massive threads are serialized because of the limited capacity of register and shared memory. Block size is therefore a crucial parameter for multithreading design, and it can be derived using an index called "warp occupancy."

A warp is the fundamental scheduling unit of concurrency in GPGPUs because all threads within a warp execute concurrently. The multiprocessor becomes idle if a warp is not ready to execute its next instruction because the instructional input operands are not yet available or the warp is waiting at a memory fence or synchronization point. Having multiple resident warps and blocks per multiprocessor can help reduce idling in this case because warps from different blocks do not need to wait for each other at synchronization points. Usage is therefore directly related to the number of resident warps. The number of clock cycles required for a warp to be ready to execute its next instruction is called the "latency," and full usage is achieved when all warp schedulers always have an instruction to issue for some warp at every clock cycle during that latency period, which is also called "latency hiding." Executing other warps when one warp is paused or stalled is the only way to hide latencies and keep the hardware busy. The occupancy metric related to the number of active warps on a multiprocessor is, therefore, crucial in determining how effectively the hardware is kept busy.

Warp occupancy is the ratio of the number of active warps per multiprocessor to the maximal number of warps. Higher occupancy represents a higher chance of latency hiding. However, occupancy is subject to resource availability; that is, it should be determined by the tradeoff between the thread block size and on-chip memory usage. Threads consuming hardware resources (i.e., registers and shared memory) limit the number of threads that can be efficiently executed in a block.

Register availability is one constraint that determines warp occupancy. Register storage enables threads to keep local variables nearby for low-latency access. However, the set of registers is a limited commodity that all threads resident on a multiprocessor must share. Registers are allocated to an entire block simultaneously. Therefore, if each thread block uses many registers, the number of thread blocks that can be resident on a multiprocessor is reduced, thereby lowering the occupancy of the multiprocessor. Shared memory also acts as a constraint on warp occupancy. In many cases, the amount of shared memory required by a kernel is related to the chosen block size. In particular, each multiprocessor has a set of shared memory that is partitioned among the thread blocks.

Warp occupancy depends on the number of active warps per multiprocessor, which is implicitly determined according to block size along with resource (register and shared memory) constraints. Choosing a proper block size requires striking a balance between latency hiding (occupancy) and resource use.

The number of blocks and warps that can reside and be processed together on the multiprocessor depends on the amount of registers and shared memory used by the kernel and the amount of registers and shared memory available on the multiprocessor. There are also a maximal number of resident blocks and a maximal number of resident warps per multiprocessor. These limits, in addition to the amounts of register and shared memory available on the multiprocessor, are a function of the computing capability of the device.

We define a mathematical model of these criteria for the decision of block size based on resource constraints. The optimal block size $\hat{B}$ is determined according to maximal warp occupancy $O(B, \mathfrak{r}, \mathfrak{s}, K_w)$, which is a function of block size B, register number per thread $\mathfrak{r}$, bytes of shared memory per block $\mathfrak{s}$, and maximal number of warps K_w.

$$\hat{B} = \max_B O(B, \mathfrak{r}, \mathfrak{s}, K_w) = \max_B \frac{\min(W_B, W_\mathfrak{r}, W_\mathfrak{s})}{K_w} \quad (11)$$

where W_B, $W_\mathfrak{r}$ and $W_\mathfrak{s}$ represent three numbers of possible active warps within a multiprocessor determined according to assigned block size B and memory usages of $\mathfrak{r}$ and $\mathfrak{s}$. This minimal function presents the constraints that active warp number is restricted to register usage and shared memory usage.

The three possible warp numbers are decided according to seven hardware parameters of CUDA devices, $T = \{K_0, K_i, G_i | i \in \{w, \mathfrak{r}, \mathfrak{s}\}\}$. K_0 is the warp size (i.e., number of threads per warp). $K_w, K_\mathfrak{r}$, and $K_\mathfrak{s}$ are the maximal number of warps, registers, and shared memory per multiprocessor, respectively. $G_w, G_\mathfrak{r}$, and $\mathrm{G}_\mathfrak{s}$ are the allocation granularity of warps, registers, and shared memory per block, respectively. For a given block size B, which is a multiple of K_0, and preassigned design parameters using $\mathfrak{r}$ and $\mathfrak{s}$, we can derive $W_B, W_\mathfrak{r}$, and $W_\mathfrak{s}$ as follows:

$$W_B = n * \lfloor K_w / \Omega(n, G_w) \rfloor, \quad (12)$$

$$W_\mathfrak{r} = n * \lfloor K_\mathfrak{r} / \Omega(\mathfrak{r} * B, G_\mathfrak{r}) \rfloor, \quad (13)$$

$$W_\mathfrak{s} = n * \lfloor K_\mathfrak{s} / \Omega(\mathfrak{s}, \mathfrak{G}_\mathfrak{s}) \rfloor, \quad (14)$$

where $n = B/K_0$ is the number of warps within a block, $\Omega(X, Y)$ is equal to the roundup of X to the nearest multiple of Y, and Z is a floor operation that obtains the maximal integer less than Z.

For example, on a Tesla 1060 device, there are a maximum of 16K registers and shared memory per multiprocessor $(K_\mathfrak{r} = 16\,\mathrm{KB}, K_\mathfrak{s} = 16\,\mathrm{KB})$, and a maximum of

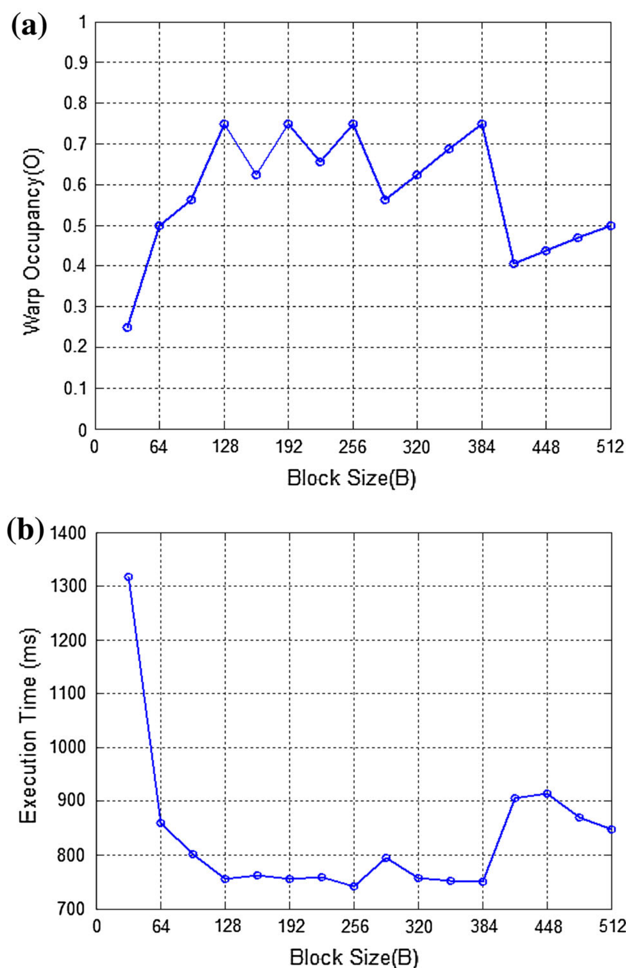


Fig. 3 Multithreading analysis taking the row filtering step as an example. **a** Relationship between block size and occupancy, **b** execution times versus block size

1,024 simultaneous residing threads (32 warps × 32 threads per warp, $K_{\rm w} = 32$, $K_0 = 32$). Warp allocations are rounded to the nearest two multiples ($G_w = 2$), register allocations are rounded to the nearest 512 registers per block ($G_{\mathfrak{s}} = 512$), and shared memory is rounded to 512 words per block ($G_{\mathfrak{r}} = 512$). A kernel with 128-thread blocks ($B = 128$) using 12 registers per thread ($\mathfrak{r} = 12$) and 40 bytes of shared memory ($\mathfrak{s} = 40$) results in an occupancy of 100 % because $O = \min\,(32, 40, 128)/32 = 1.0$. Whereas a kernel with 192-thread blocks ($B = 192$) using the same $\mathfrak{r} = 12$ and $\mathfrak{s} = 40$ brings about an occupancy of 94 %, because $O = \min\,(30, 36, 192)/32 = 0.94$.

The formulation is applied to the GPURetinex for the selection of block size. Consider row filtering and Tesla 1060 as examples. The design of the row filtering must use 20 registers ($\mathfrak{r} = 20$) and 40 bytes of shared memory ($\mathfrak{s} = 40$). The occupancy function $O(B, \mathfrak{r}, \mathfrak{s})$ reduces to $O(B) = \min\,(32, 3B/32, B)/32$ and is plotted in Fig. 3a.

An experiment exploring the execution times corresponding to B of the row filtering is conducted to verify $O\,(B)$ in Fig. 3a. The GPURetinex is executed using fixed B for all steps except that B of the row filtering is adjustable. The results are shown in Fig. 3b. The two graphs in Fig. 3 match each other perfectly because higher occupancy relates to lower execution time. However, peak values in Fig. 3a are not necessary the chosen block sizes. Higher occupancy may not demonstrate higher performance when $O > 0.6$, because latency hiding is satisfied when enough warps are running. We conclude that the range of block sizes from 128 to 384 is favorable for row filtering.

6 Implementation

We apply the analyses of memory access pattern and multithreading pattern to every step of the GPURetinex. We adopt the memory selection result in Table 2 for implementation, and the block size of 256 threads is chosen for all steps because the chosen block size achieves occupancy by more than 0.6 for all steps. The Gaussian blur stage includes the two convolution steps: row and column filtering. Each thread block contains 256 threads corresponding to each row. The convolutions of row and column filters use texture and constant caches to improve efficiency. Data are fetched from caches with higher memory bandwidth because spatial locality is fully exploited when threads in the same warp access nearby pixels in the image data, and temporal locality for the repetitions of reading the same data.

Moreover, an additional advantage of selecting texture memory to accommodate source and intermediate images in the parallel Gaussian blur stage is that addressing calculation of boundary condition outside the images is improved using dedicated hardware units. Texture memory provides an advantageous capability that is useful for block-wise image processing. It provides four addressing modes: mirror, clamp, wrap, and border, to handle coordinates that are out of range. The automatic handling of boundary cases refers to how a texture coordinate is resolved when it falls outside the valid addressing range. We adopt the automatic mirror addressing mode to reduce the overhead of handling out-of-range boundary using software. An opaque memory layout for texture memory, called the "CUDA array," optimized for texture fetching and supporting four out-of-boundary extrapolations, is also adopted.

This texturing technique differs from our prior previously proposed approach [38], in that transposing an image is performed before column filtering to prevent non-unit stride access. However, the transpose step is unnecessary

for texture cache because texture cache provides excellent bandwidth for 2D locality.

The parallel computing of log-domain processing includes subtractions, multiplication of weights and color restoration factor, and summation, as indicated in (5). These computations can be performed in parallel at a pixel-wise level with a thread block size of 256. Because each $R_i(x, y)$ value is calculated independently, data decomposition does not influence performance. The same horizontal stripe distribution and threading mechanism with a Gaussian blur is adopted here for simplicity. All data, including $G_{k,i}(x, y)$, $I_i(x, y)$, and $R_i(x, y)$, are read from and written to global memory because of the capacity limitation of international memory.

Alpha-trimmed contrast stretching includes the histogramming in (7) and the stretching in (6), which are two parallelization issues in this stage. Although the stretching step is an inherently parallel problem, the histogramming step is not. Histogramming fills a set of bins according to the occurrence of pixel values obtained from $R_i(x, y)$. Although histogramming is trivial to sequential processing, its computation on parallel processors is not straightforward.

Parallel histogramming usually adopts the reduction technique [39–41] to maximize the use of GPU hardware resources in both memory constraint and multithreading capability. To reduce a large vector of image values to a smaller bin vector serially using the sum operator, we must render buffers called "partial histograms" that are the histogram of a subimage. We then combine these partial histograms into the final histogram. Histogramming can be divided into two steps. The first step is to count the pixels of $N(x, y)$ in parallel into a partial histogram. Each warp of a thread block has its local histogram in shared memory. Each thread of a warp uses the atomic addition function of shared memory to sum the local histogram. The second step of parallel histogramming is to merge the partial histogram into a global histogram. In this step, the thread block size is 256, and each thread block sums a bin of a global histogram from the partial histogram. The reads of each thread block are uncoalesced, but this step requires only a fraction of total processing time. When a thread block writes all values of a bin in the partial histogram into its shared memory, recursive doubling procedures [13] are executed to merge all values of a bin into one.

Finally, for the cumulative histogram $H(i)$, $\hat{a}_{\min}$ and $\hat{a}_{\max}$ are computed sequentially from the small data array $h(i)$, and the stretching in (6) is performed in parallel at a pixel-wise level.

7 Experimental results

The performance of the GPURetinex has been tested on Tesla C1060 and CUDA 3.2. The GPGPU is combined with an Intel Core2 Duo of 3.0 GHz. The GPGPU has 240 streaming processors (SPs). In total, 240 color images were tested in these experiments to verify the enhancement and acceleration results of the GPURetinex.

There are four implementations of the method for experimental comparison. GPURetinex represents the proposed method presented in Sects. 3–6, that is, a parallel MSRCR with the parallel alpha-trimmed contrast stretching. There are two variants of the GPURetinex: GPURetinex_N and GPURetinex_NPP. GPURetinex_N is the GPURetinex without alpha-trimmed contrast stretching, or parallel MSRCR, which has been presented in [33]. GPURetinex_NPP [34] is the GPURetinex replacing the proposed convolution filter according to the functions in the NPP (nVidia Performance Primitive) 4.0 library [42] (functions nppiFilterRow_8u_C1R and nppiFilterColumn_8u_C1R). The fourth is a serial implementation of the proposed algorithm presented in Sect. 3 (i.e., CPURetinex), developed and run using a single thread on one core of the CPU. Gaussian blur filtering in the CPU version adopts the optimized SSE implementation using OpenCV.

7.1 Image enhancement test

The GPURetinex enhancement results with and without the alpha-trimmed contrast stretching (i.e., GPURetinex vs. MSRCR) are compared. Selective examples with challenging issues are chosen to demonstrate the enhancement results. Figure 4a, d, g is three original images. The first image is an outdoor scene of a non-uniform illumination problem that loses the color and detail in the tinted window on truck. The second image is the scene of low-key with loss of color and detail in the shadow. The third image is a low-contrast image with poor visibility condition caused by smoke. It is not easy to identify the objects in the scene.

Figure 4b, e, h shows the enhanced results of MSRCR, and Fig. 4c, f, i shows the enhanced results of GPURetinex. Figure 4c shows superior color/lightness rendition and contrast to that of MSRCR. The details of the tinted window on the truck have been enhanced. The lightness and contrast are improved in Fig. 4f. In Fig. 4i, dramatic improvements in overall visibility over the directly observed scene are apparent. The objects can be identified easily. The alpha-trimmed contrast stretching method can greatly improve color/lightness rendition and contrast.

7.2 Performance analysis

We next compare the execution time of the GPURetinex method with other algorithms on image resolutions $M \times M$, from 256 × 256 to 4,096 × 4,096. Three Gaussian filters of scales $v_z = 17$, 83, and 253 are adopted to compute the center/surround information in these

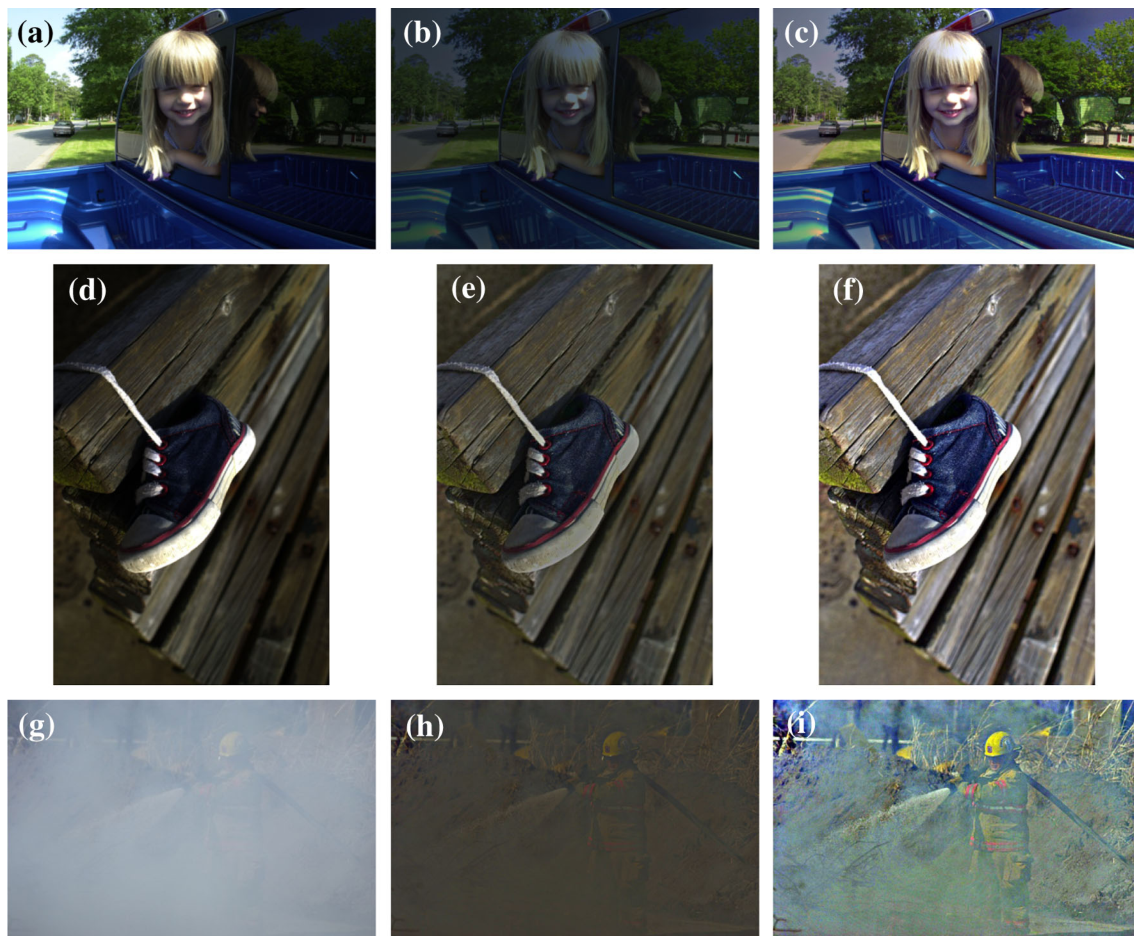


Fig. 4 The three sets of original and enhanced images. **a** An uneven illumination image with high dynamic range, **d** a low-key image, **g** a smoky image, **b**, **e**, **h** the results of MSRCR, **c**, **f**, **i** the results of GPURetinex. The retinex parameters for **a** is $W_z = 0.1, 0.2, 0.7$, $v_z = 5, 80, 1{,}040$, and $\alpha = 0.006$. The retinex parameters for **b** is $W_z = 0.2, 0.3, 0.5$, $v_z = 15, 80, 800$, and $\alpha = 0.015$. The retinex parameters for **c** is $W_z = 1/3, 1/3, 1/3$, $v_z = 15, 80, 250$, and $\alpha = 0.003$

experiments. The total execution time includes the computational times of (3)–(10). The time of every image resolution is a time average of 240 color images.

Figure 5 shows the total execution times between the GPURetinex and CPURetinex. The total execution time of both CPU and GPU versions increases proportionally according to image dimensions. The total execution times of the GPU versions are much less than those of CPU-based implementation. As the image size increases, the difference is more obvious. No difference exists between GPURetinex and GPURetinex_N because the additional alpha-trimmed contrast stretching stage required extremely little time. The GPURetinex is 1.7 times faster than GPURetinex_NPP on average, showing that our convolution filtering is more efficient than the convolution filtering performed using the GPGPU standard library.

Figure 6 shows the accelerations of the GPURetinex compared to the CPURetinex. Acceleration is measured

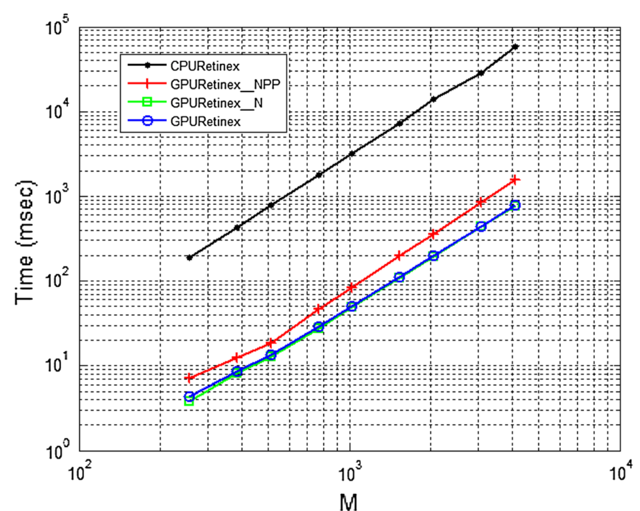


Fig. 5 Total execution time with different image sizes between the GPU and CPU versions

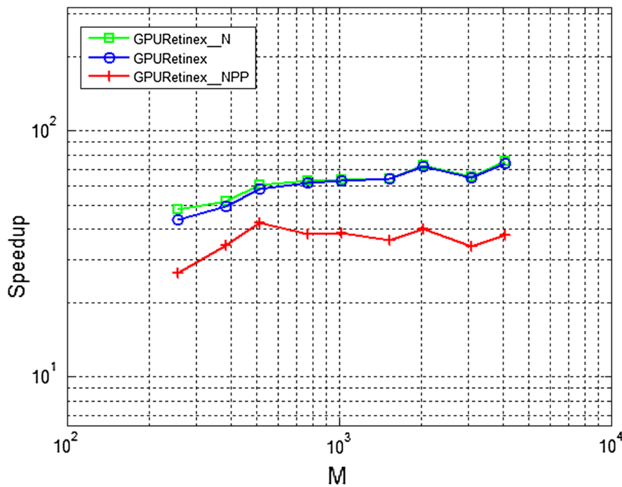


Fig. 6 The accelerations of the GPU algorithms over CPU versions

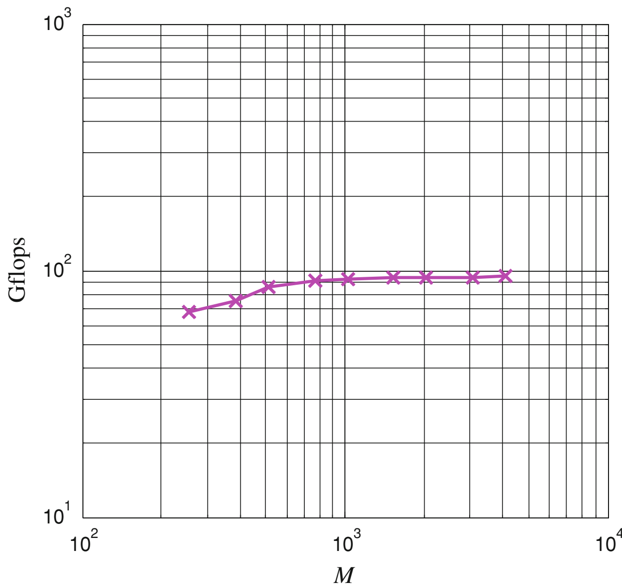
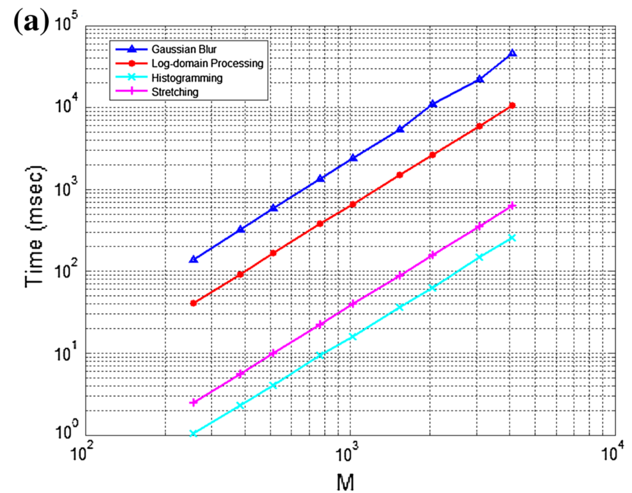


Fig. 7 The Gflops of GPURetinex



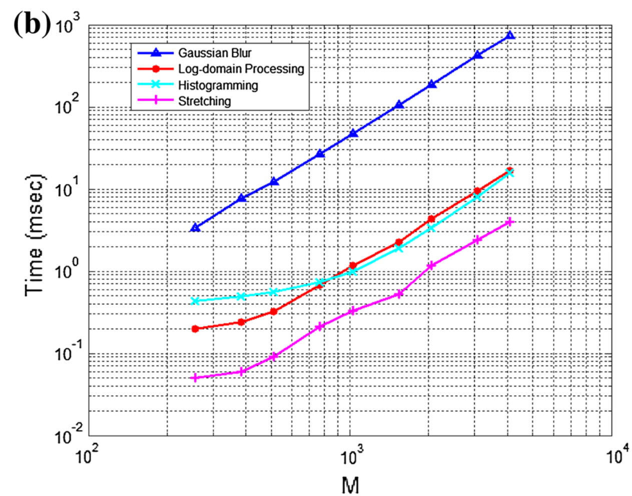


Fig. 8 The execution times of various steps for **a** CPURetinex, **b** GPURetinex

using the total execution time that does not include the cost of memory management and data transfer between the GPU and CPU. Relative acceleration is also linear according to the increase of image dimensions, thus indicating that the GPU implementation is memory bound rather than operationally bound. For the image of 4,096 × 4,096 resolution, the GPURetinex can gain 74× acceleration when compared with CPURetinex, and is twice as fast as GPURetinex_NPP. The experimental results demonstrate an excellent speed boost.

We further investigate the performance of the GPURetinex according to Gflops. The Gflops of the GPURetinex is defined as $(O/T) \times 10^{-9}$, where T is the total execution time of GPURetinex (s), and O is the number of floating-point operations in the modified retinex algorithm. As shown in Fig. 7, the performance converges to 95 Gflops when image sizes increase. It is 1/10 of theoretical peak Gflops of this GPGPU device because of the parallel overhead in the convolution stage. Furthermore, the access ratio and the bandwidth of global memory can affect the Gflops [43].

We illustrate the execution times of each operation in the modified retinex algorithm. The execution times of the four steps, including the Gaussian blur, log-domain processing, histogramming, and stretching steps, are compared according to image size. We first show the execution times for CPURetinex in Fig. 8a. The execution times of all steps increase in proportion to image sizes and the Gaussian blur always consumes the most processing time. The percentages of computational loading are 80.1 % for Gaussian blur, 18.4 % for log-domain processing, and 1.5 % for histogramming and stretching. Figure 8b shows the execution times for GPURetinex. The Gaussian blur still

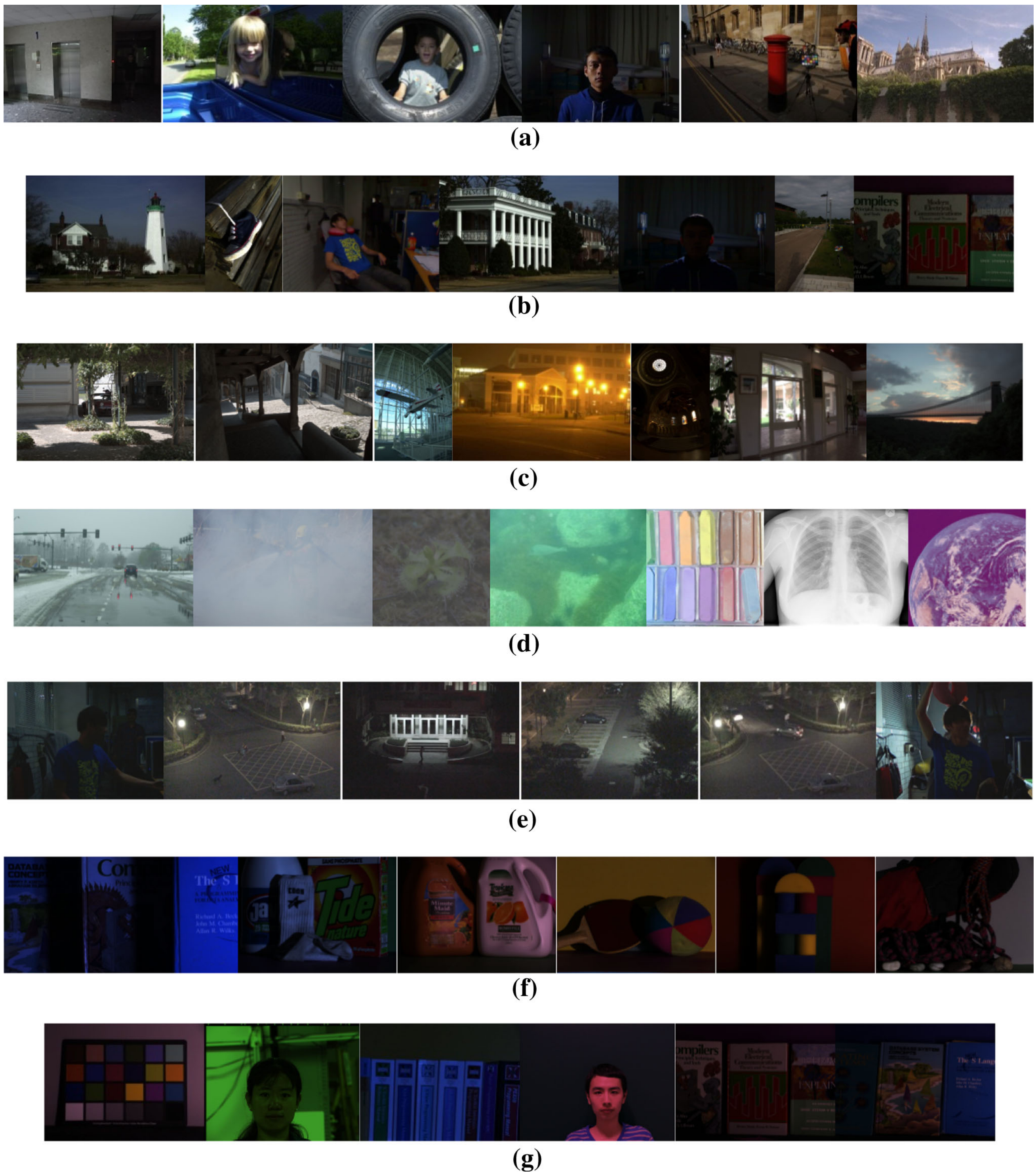


Fig. 9 Selective samples of test images. **a** The uneven illumination class, **b** low-key class, **c** high dynamic range class, **d** low-contrast class, **e** high noise class, **f** uneven illumination/color shift class, **g** even illumination/color shift class

consumes the most processing time. The execution times do not increase proportionally to small image sizes, possibly because the memory transfer of small image data corresponding to the high memory bandwidth of global memory exhibits a nonlinear effect.

7.3 Parallel histogramming

The performance of parallel histogramming can vary according to image characteristics, and it is further evaluated using as many as 240 test images. These images are

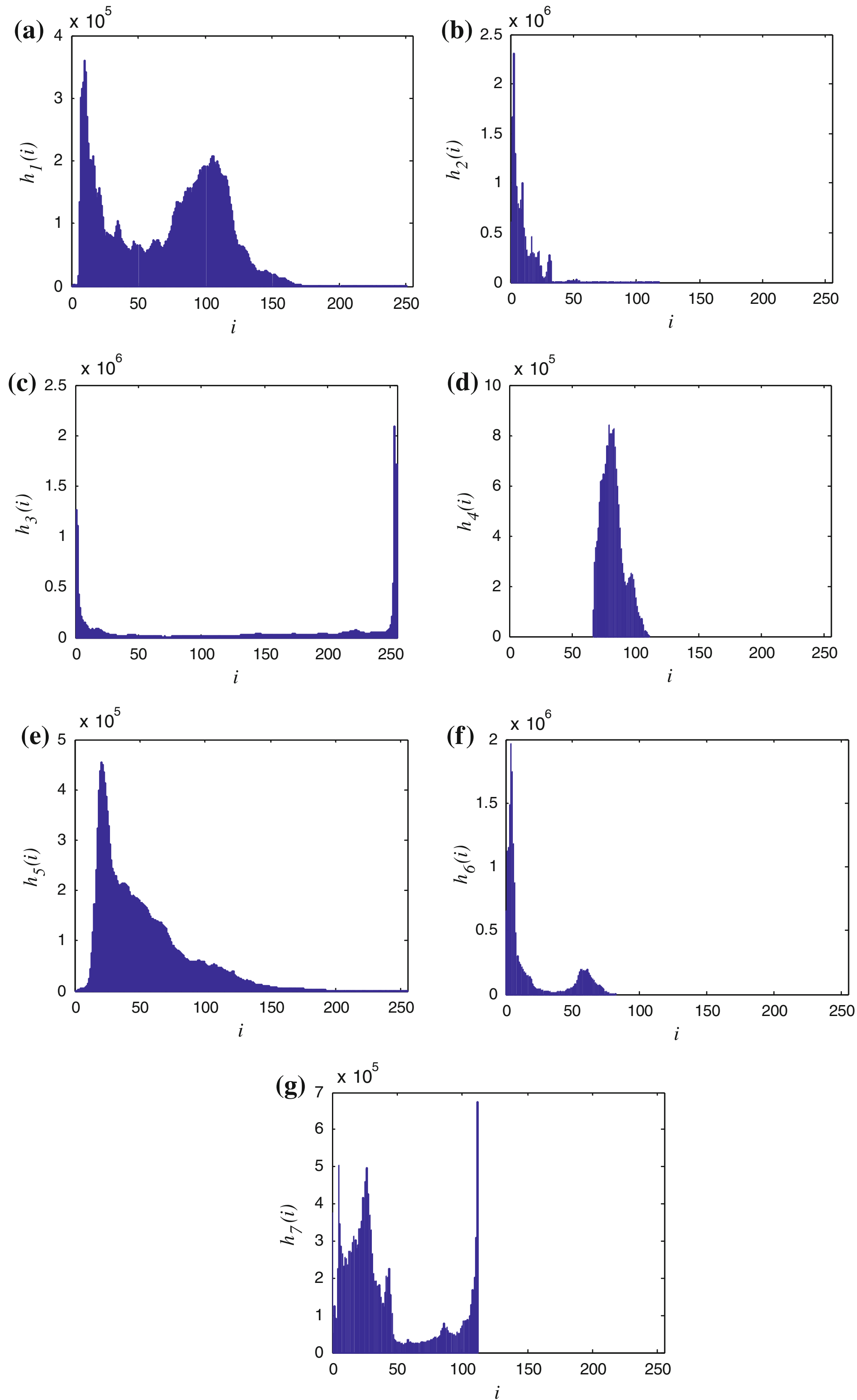


Fig. 10 Typical histograms of the seven classes. **a** Uneven illumination, **b** low-key, **c** high dynamic range, **d** low-contrast, **e** high noise, **f** uneven illumination/color shift, **g** even illumination/color shift

classified into seven classes: uneven illumination, low-key, high dynamic range, low contrast, high noise, uneven color shift, and even color shift. Selected samples of each image class are shown in Fig. 9.

The execution times of parallel histogramming are not the same for all of the images because the atomic operation adopted to avoid race condition can perform serially. If a bin of the histogram has more pixels, racing to access the same element is frequent, which induces more serialization operations and degrade the performance. Figure 10 shows the typical histograms of the seven classes. These unqualified images usually have a non-equalized histogram.

Figure 11 shows the average execution times of parallel histogramming for some of the test images that have a resolution of 4,096 × 4,096. The average execution times of the seven classes vary greatly. The high dynamic range class has the highest execution time because too many serial calculations occur to the pixels of the two relatively high peaks, as shown in Fig. 10c. Histogramming requires less than 1.5 % execution time and contributes an unnoticeable effect on the performance of the GPURetinex.

8 Conclusion

This paper presents a GPU-accelerated data parallel algorithm for the proposed retinex algorithm. In addition to the large-kernel-size convolution and log-domain operations, alpha-trimmed contrast stretching is devised to improve color rendition and contrast. The three popular types of image processing tasks: block-wise, pixel-wise, and serialized, all exist in the proposed retinex algorithm and the involved parallelization is challenging. The GPURetinex adopts a horizontal stripe data distribution to decompose the image data for parallelization, optimize memory usage and data alignment, and fully use hardware support. The GPURetinex gained 74 times acceleration when compared with CPU-based implementation on images with 4,096 × 4,096 resolution. The experimental results demonstrate that the performance advances of center/surround using retinex by GPGPU are definitely feasible.

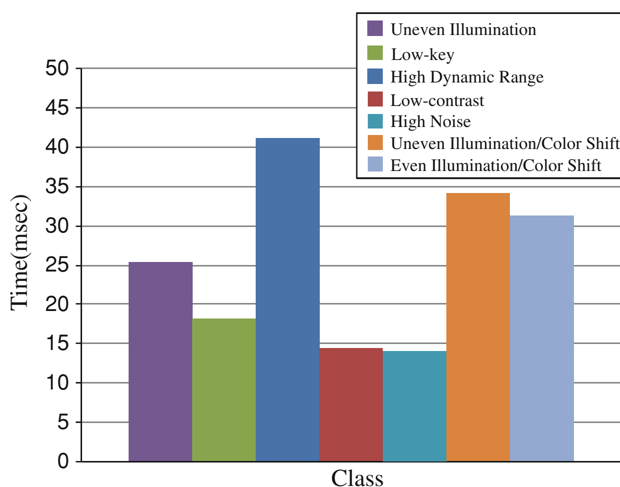


Fig. 11 Average execution time of parallel histogramming for the seven-class images

The acceleration of the GPURetinex is attained by mathematically analyzing memory access and multithreading patterns. Parallel design and CUDA implementation of the algorithm are examined closely by the parallelism concurrently operating on multiple data elements. Our experience shows that using CUDA enables moving complex image processing algorithms to a GPGPU with a significant acceleration against serialized implementation on a CPU. However, a complete understanding of the memory hierarchy and multithreading model is essential for parallelization. High processor occupancy is also critical for maximal performance on the GPGPU. Moreover, the data distribution for each thread block and data allocation in the memory hierarchy merits considerable attention.

Although the two mathematical models are validated on GT200 GPU and CUDA 3.2, the models can be universally applied to all the nVidia GPGPUs. However, selecting memory spaces and block sizes for a specific GPGPU is achieved in this paper manually before implementation, but not automatically within implementation. An algorithmic method realizing the mathematical models could be developed in the future to automate the selection of memory spaces and block sizes.

The efficiency of the proposed GPURetinex could be further enhanced. Further acceleration is possible using the data parallel techniques that are more sophisticated. Moreover, because the performance bottleneck is dominated by the Gaussian convolution, algorithms that are more efficient, such as the recursive Gaussian filter, can be studied further.

References

1. Marsi, S., Saponara, S.: Integrated video motion estimator with retinex-like pre-processing for robust motion analysis in automotive scenarios: algorithmic and real-time architecture design. J. Real-Time Image Proc. **5**(4), 275–289 (2010)
2. Meylan, L., Susstrunk, S.: High dynamic range image rendering with a retinex-based adaptive filter. IEEE Trans. Image Process. **15**(9), 2820–2830 (2006)
3. Park, Y., Kim, J.: Fast adaptive smoothing based on LBP for robust face recognition. Electronic Lett. **43**(24), 1350–1351 (2007)
4. Ra, J., Jang, J., Bae, Y.: Contrast-Enhanced Fusion of Multi-Sensor Images Using Subband-Decomposed Multiscale Retinex. IEEE Trans. Image Process. **21**(8), 3479–3490 (2012)
5. Ebner, M.: Color constancy. John Wiley & Sons Ltd, pp. 143–153. Chichester, England (2007)

6. Land, E.: The Retinex. Amer. Scient. **52**(2), 247–264 (1964)
7. Jobson, D.J., Rahman, Z., Woodell, G.A.: Properties and performance of a center/surround Retinex. IEEE Trans. Image Process. **6**(3), 451–462 (1997)
8. Rahman, Z., Jobson, D., Woodell, G. A.: Multiscale Retinex for color image enhancement. In: Proceedings of the IEEE International Conference on Image Processing, vol. 3, pp. 1003–1006. Lausanne, Switzerland (1996)
9. Jobson, D.J., Rahman, Z., Woodell, G.A.: A multi-scale Retinex for bridging the gap between color images and the human observation of scenes. IEEE Transactions on Image Processing: Special Issue on Color Processing **6**(7), 965–976 (1997)
10. Rahman, Z., Jobson, D., Woodell, G.A.: Retinex processing for automatic image enhancement. J. Electron. Imaging **13**(1), 100–110 (2004)
11. Che, S., Boyer, M., Meng, J., Tarjan, D., Sheaffer, J.W.: Skadron. K.: A performance study of general-purpose applications on graphics processors using CUDA. J. Parallel Distrib. Comput. **68**(10), 1370–1380 (2008)
12. Cope, B., Cheung, P.Y.K., Luk, W., Howes, L.: Performance comparison of graphics processors to reconfigurable logic: a case study. IEEE Trans. Comput. **59**(4), 433–448 (2010)
13. Siegel, H.J., Wang, L., So, J.E., Maheswaran, M.: Data parallel algorithms. Parallel and Distributed Computing Handbook, pp. 466–499. McGraw-Hill, New York (1996)
14. Moreland, K., Angel, E.: The FFT on a GPU. In: SIGGRAPH/Eurographics Work-shop on Graphics Hardware, pp. 112–119. Aire-la-Ville, Switzerland (2003)
15. Strzodka, R., Garbe, C.: Real-time motion estimation and visualization on graphics cards. In: Proceedings of IEEE Visualization, pp.545–552. Austin, USA (2004)
16. Shen, G., Gao, G.P., Li, S., Shum, H., Zhang, Y.: Accelerate video decoding with generic GPU. IEEE Trans. Circuits Syst. Video Technol. **15**(5), 685–693 (2005)
17. GPU4Vision, http://www.gpu4vision.org (2011)
18. Fung, J., Mann, S., Aimone, C.: OpenVIDIA: parallel GPU computer vision. In: Proceedings of ACM International Conference on Multimedia, pp. 849–852. Hilton, Singapore (2005)
19. Allusse, Y., Horain, P., Agarwal, A., Saipriyadarshan, C.: GpuCV: an opensource gpu-accelerated framework for image processing and computer vision. In: Proceedings of ACM International Conference on Multimedia, pp. 1089–1092. Vancouver, Canada (2008)
20. Babenko, P., Shah, M.: MinGPU: a minimum GPU library for computer vision. J. Real-Time Image Proc. **3**(4), 255–268 (2008)
21. Owens, J.D., Luebke, D., Govindaraju, N., Harris, M., Krger, J., Lefohn, A.E., Purcell, T.J.: A survey of general-purpose computation on graphics hardware. Comput. Graph. Forum **26**(1), 80–113 (2007)
22. Luo, Y., Duraiswami, R.: Canny edge detection on NVIDIA CUDA. In: IEEE Computer Society Conference on Computer Vision and Pattern Recognition Workshops, pp. 1–8. Anchorage, USA (2008)
23. Lozano, M., Otsuka, K.: Real-time visual tracker by stream processing. J. Sign. Process. Syst. **57**(2), 674–679 (2009)
24. Su, Y., Xu, Z.: Parallel implementation of wavelet-based image denoising on programmable PC-grade graphics hardware. Signal Process. **90**(8), 2396–2411 (2009)
25. Herout, A., Josth, R., Havel, J., Hradis, M., Zemcik, P.: Real-time object detection on CUDA. J. Real-Time Image Proc. **6**, 159–170 (2011)
26. Jensen, L.B.W., Kjar-Nielsen, A., Pauwels, K., Jessen, J.B., Hulle, M.V., Kruger, N.: A two-level real-time vision machine combining coarse- and fine-grained parallelism. J. Real-Time Image Proc. **5**, 291–304 (2010)
27. Jang, B., Schaa, D., Mistry, P., Kaeli, D.: Exploiting memory access patterns to improve memory performance in data-parallel architectures. IEEE Trans. Parallel Distrib. Comput. **22**(1), 105–118 (2011)
28. Brainard, D., Wandell, B.: Analysis of the Retinex theory of color vision. J. Opt. Soc. Am. A **3**(10), 1651–1661 (1986)
29. Rizzi, A., Gatta, C., Marini, D.: From Retinex to automatic color equalization issues in developing a new algorithm for unsupervised color equalization. J. Electron. Imag. **13**(1), 15–28 (2004)
30. Frankle, J., McCann, J.: Method and apparatus for lightness imaging. US Patent, 4384336 (1983)
31. McCann, J.: Lesson learned from mondrians applied to real images and color gamuts. In: Proceedings of IS&T/SID Seventh Color Imaging Conference, vol. 14, pp. 1–8. Scottsdale, USA (1999)
32. Sobol, R.: Improving the Retinex algorithm for rendering wide dynamic range photographs. J. Electron. Imag. **13**(1), 65–74 (2004)
33. Land, E.: An alternative technique for the computation of the designator in the Retinex theory of color vision. In: Proceedings of the National Academy of Science, vol. 83, pp. 3078–3080. USA (1986)
34. Tao, L., Asari, V.: Modified luminance based MSR for fast and efficient image enhancement. In: 32nd Applied Imagery Pattern Recognition Workshop, pp. 174–179. Washington, USA (2003)
35. Seo, H., Kwon, O.: CUDA implementation of McCann99 Retinex Algorithm. In: Proceedings of International Conference Computer Science and Convergence Information Technology, pp. 388–393 (2010)
36. Wang, Z. N., Liu, C. Z., Lu, Y., Wu, M., Zhang, P.: The implementation of multi-scale Retinex image enhancement algorithm based on GPU via CUDA. In: Proceedings of International Symposium Intelligent Signal Processing and Communications Systems, pp. 1–4. (2010)
37. Wang, Y. K., Huang, W. B.: Acceleration of an improved Retinex algorithm. In: IS&T/SPIE Electronic Imaging, Parallel Processing for Image Applications. Proceedings of SPIE, vol. 7872. California, USA (2011)
38. Wang, Y.K., Huang, W.B.: Acceleration of the Retinex algorithm for image restoration by GPGPU/CUDA. In: IEEE Computer Society Conference on Computer Vision and Pattern Recognition Workshops, pp. 72–77. Colorado, USA (2011)
39. Harris, M.: Optimizing parallel reduction in CUDA. In: NVIDIA Developer Technology (2007)
40. Podlozhnyuk, V.: Histogram calculation in CUDA. In: NVIDIA white paper (2007)
41. Shams, R., Kennedy, R.A.: Efficient histogram algorithms for NVIDIA CUDA compatible devices. In: Proceedings of International Conference Signal Processing and Communications Systems (ICSPCS), pp. 418–422. Gold Coast, Australia (2007)
42. NVIDIA Corporation: NVIDIA Performance Primitives (NPP), Version 4.0. (2011)
43. Kirk, D.B., Hwu, W.W.: Programming Massively Parallel Processors: A Hands-on Approach, pp. 77–94. Elsevier, Burlington (2010)

Author Biography

Yuan-Kai Wang Yuan-Kai Wang received the B.S. degree of electrical engineering in 1990 and Ph.D. degree of computer science and information engineering in 1995, from National Central University. He was a postdoctoral fellow in the Institute of Information Science of Academia Sinica from 1995–1999. From 1999 he joined

the department of electrical engineering, Fu Jen University as an associate professor.

He has been the Chair and Program Committee member of many international conferences, and reviewers of many journals and IEEE transactions. His research interests include computer vision, pattern recognition, and machine learning. He studied applications of the above methods on video surveillance, face recognition, biometrics, and robotic vision. Currently he is interested in machine learning and video analysis for visual surveillance, embedded computer vision, human–computer interface, robotics, and health care.