EURASIP Journal on Image and Video Processing
a SpringerOpen Journal

RESEARCH Open Access

Two-way partitioning of a recursive Gaussian filter in CUDA

Chang Won Lee, Jaepil Ko and Tae-Young Choe*

Abstract
Recursive Gaussian filters are more efficient than basic Gaussian filters when its filter window size is large. Since the computation of a point should start after the computation of its neighborhood points, recursive Gaussian filters are line oriented. Thus, the degree of parallelism is restricted by the length of the data image. In order to increase the parallelism of recursive Gaussian filters, we propose a two-way partitioned recursive Gaussian filter. The proposed filter partitions a line into two lines and a point, which is used for Gaussian blur effect across the two lines. This partition increases the parallelism because the filter is applied to the two blocks in parallel. Experimental results show that the process time of the proposed filter is half compared to the time of an one-way parallel recursive Gaussian filter while the peak signal-to-noise ratio is maintained within an acceptable rate of 26 to 33 dB.

Introduction

A Gaussian blur filter, or Gaussian filter, is one of the fundamental and widely used image processing techniques. A typical use of the filter is denoising. It is also used as a preprocessing step for down/up sampling, edge detection [1,2], or scale space representation [3]. Contrast enhancement techniques such as Retinex [4-6] or unsharp filters [7] are the other uses for Gaussian blur where it approximates the illumination component of an image at a large scale.

According to the definition of Gaussian filter, filtered value of a pixel in a two-dimensional image is computed using nearby pixel values. The range of the pixels to be used is determined by filter window size $N \times N$. It is known that the filtered value of a pixel can be computed by pixels in a horizontal line and a vertical line that pass through the pixel. Let us call the Gaussian filter that computes filtered value of a pixel using the crossing two lines as *finite impulse response* (FIR) filter. FIR filter has $4N \times width \times height$ computation steps if the size of given image is width $\times$ height.

Although FIR filter implements Gaussian blur filter in the exact discrete way, the processing time of the filter depends on the filter window size $N \times N$. Recursive Gaussian filters that implement Gaussian filter are developed in order to eliminate effect of the filter window size. A recursive Gaussian filter computes filtered value of a pixel using differential values of its neighborhood pixels [8]. Since the differential values of neighborhood pixels contain all approximated data within the range of the filter window size, the filter window size is not involved in the number of computation steps. Computation step of a recursive Gaussian filter proposed by van Vliet et al. is about $32 \times width \times$height. Thus, recursive Gaussian filters is faster than the FIR filter if a filter window size is greater than 8. Unfortunately, recursive Gaussian filters make dependence between pixels and restrict the degree of parallelism. Pixel $p[i]\,[j]$ must wait until the filtered value of pixel $p[i-1]\,[j]$ is computed in the row-oriented step.

As graphic processing unit (GPU) cores can be used for general purpose computation, many image processing algorithms have been implemented in general purpose GPU (GPGPU). NVIDIA supports Compute Unified Device Architecture (CUDA) as a GPGPU architecture and a development environment. Since recursive Gaussian filters make dependencies between pixels, it is conventional to allocate a line into a thread in a core processes. On the other side, the bitmap Gaussian filter can allocate 1 pixel per 1 thread, which fully utilizes available cores. Thus, recursive Gaussian filters shows better performance in the restricted area where the number of cores is small and the filter window size is large.

*Correspondence: choety@kumoh.ac.kr
Department of Computer Engineering, Kumoh National Institute of Technology, Gumi, Gyeongbuk 730-701, Korea

We propose a refined recursive Gaussian filter for GPGPU that partitions working domain into two ways. The proposed filter combines a recursive Gaussian filter and FIR filter in order to minimize error rate that occurs by splitting the working domain. The remainder of this paper is structured as follows: 'Problem environment and related work' section explains the problem environment and reviews related work. 'Proposed filter' section gives details of the proposed refined recursive Gaussian filter. 'Experimental results' section gives the experimental results of the proposed filter. Finally, 'Conclusions' section concludes with future works.

Problem environment and related work

Recursive Gaussian filter

Gaussian blur uses the Gaussian distribution function. Equation 1 shows a Gaussian distribution function for a two-dimensional space represented as two-dimensional array in$[x, y]$ [9]:

$$g(x,y) = \frac{1}{\sqrt{2\pi}\sigma} e^{-\frac{d^2}{2\sigma^2}}, \tag{1}$$

where $d = \sqrt{(x-x_c)^2 + (y-y_c)^2}$ is the distance of the neighborhood pixel in$[x, y]$ from the center pixel in$[x_c, y_c]$, and σ denotes the Gaussian half-width [10]. Since discrete two-dimensional space like image is filtered, integer value N is used instead of real value σ. For the simplicity, N is set to $2 \times \sigma$, and filter window size $N \times N$ is used. Since a basic Gaussian blur (FIR) filter is a type of a separable filter [11], the filter computes the filtered pixel out$[x, y]$, the discrete convolution of pixel in$[x, y]$ with the sampled Gaussian, as follows:

$$\begin{aligned} w[x,y] &= \sum_{k=x-N/2}^{x+N/2} \text{in}[k-x,y]\, g(k,y), \\ \text{out}[x,y] &= \sum_{l=y-N/2}^{y+N/2} w[x,l-y]\, g(x,l), \end{aligned} \tag{2}$$

where out[ ] is computed after the two-dimensional array w[ ] is computed. Thus, the time complexity of the basic Gaussian filter for 1 pixel depends on the filter window size and is about $2N + 1$ additions and $2N + 2$ multiplications.

Such computational complexity can be reduced to a constant if a recursive Gaussian filter is used [8,12]. Young and van Vliet proposed a recursive Gaussian filter, which we call it YVRG filter, using an approximation by the Fourier transform [8]. A recursive Gaussian filtering process requires two *steps* where each step is composed of two *passes* as follows:

1. Row-oriented step
   (a) Forward pass generates w[ ] using in[ ]
   (b) Backward pass generates out[ ] using w[ ]
2. Column-oriented step
   (c) Downward pass generates w[ ] using out[ ]
   (d) Upward pass generates in[ ] using w[ ]

where the two-dimensional array in[ ] stores the input image, and another two-dimensional array out[ ] stores the intermediate image after the row-oriented step finishes. A one-dimensional array w[ ] temporarily stores the intermediate data during each pass of the lines. After the filtering finishes, array in[ ] is overwritten by the filtered image. These steps are shown in Figure 1. Since the column-oriented step is the same as the row-oriented step except for its direction, only the row-oriented step will be discussed in the paper as shown in Figure 2. During a forward pass, input pixels are processed in the forward direction and the intermediate pixels are stored in a temporary array named w[ ]. During a backward pass, the pixels in w[ ] are processed in the backward direction. The resulting pixels are stored in data array out[ ] and used as input data in the column-oriented step.

Let us show the detailed step of YVRG filter. The resulting pixel out$[x, y]$ is computed from the following two recursive passes:

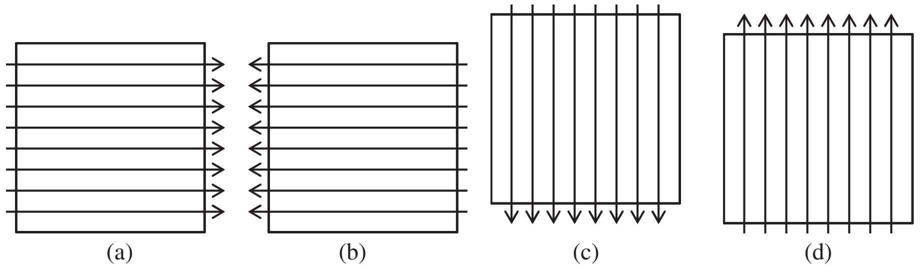

Figure 1 Passes of recursive Gaussian filter. Sequence of a recursive Gaussian filter: **(a)** row-oriented step forward pass, **(b)** row-oriented step backward pass, **(c)** column-oriented step forward pass, and **(d)** column-oriented step backward pass.

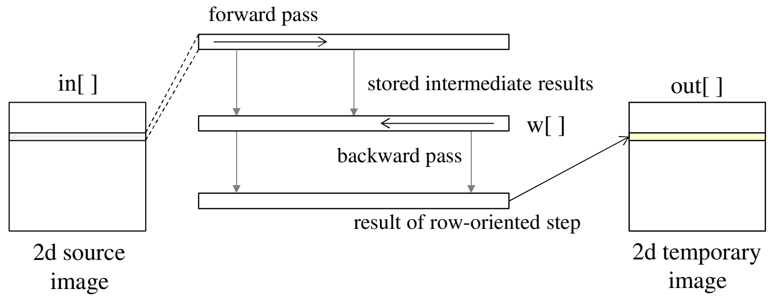

Figure 2 Line-oriented process of a recursive Gaussian filter. Array *w*[ ] is used for intermediate storage.

Forward pass:

$$\begin{aligned} w[x,y] &= B\cdot \mathrm{in}[x,y] + (b_1\cdot w[x-1,y] + b_2\cdot w[x-2,y] \\ &\quad + b_3\cdot w[x-3,y])/b_0, \end{aligned} \tag{3}$$

Backward pass:

$$\begin{aligned} \mathrm{out}[x,y] &= B\cdot w[x,y] + (b_1\cdot \mathrm{out}[x+1,y] \\ &\quad + b_2\cdot \mathrm{out}[x+2,y] + b_3\cdot \mathrm{out}[x+3,y])/b_0. \end{aligned} \tag{4}$$

The precalculated constants are

$$\begin{aligned} b_0 &= 1.57825 + 2.44413q + 1.4281q^2 + 0.422205q^3 \\ b_1 &= 2.44413q + 2.85619q^2 + 1.26661q^3 \\ b_2 &= -(1.4281q^2 + 1.26661q^3) \\ b_3 &= 0.422205q^3 \\ B &= 1 - ((b_1 + b_2 + b_3)/b_0) \\ q &= \begin{cases} 0.1147705018520355224609375 & \text{if } N < 0.5 \\ 3.97156 - 4.14554\sqrt{1 - 0.26891N} & \text{if } 0.5 \le N < 2.5 \\ 0.98711N - 0.96330 & \text{otherwise.} \end{cases} \end{aligned}$$

Since filter window size N is used to calculate constant q only, it does not affect the time complexity of the YVRG filter. Equations 3 and 4 show that pixels in a line are bound by precedence relations. The resulting value of pixel out$[x,y]$ needs the value of out$[x+1,y]$ and the intermediate pixel value $w[x,y]$ needs the value of $w[x-1,y]$. Thus all pixel values in a line are restricted by a linear precedence order.

While the recursive process needs three previous neighborhood pixels for each pixel according to Equations 3 and 4, there are no sufficient neighborhood pixels near the image boundaries for the process. For example, pixels $w[-1,y]$, $w[-2,y]$, and $w[-3,y]$ required by pixel $w[0,y]$ are not available in the image. Thus, boundary pixels are duplicated as shown in Figure 3. In the case of a row-oriented step, in$[x,0]$ is copied to $w[x,-3]$ through $w[x,-1]$ and these are used to calculate $w[x,0]$ through $w[x,2]$.

A forward pass uses five multiplications and three additions per pixel as same as the backward pass does. In order to compute the filtered value of a pixel, four passes are required. Thus 20 multiplications and 12 additions are required per pixel. A recursive Gaussian filter of any filter window size has a similar time complexity as a basic Gaussian filter with filter window size seven.

Unfortunately, the precedence relation of YVRG filter causes a disadvantage if the filter is applied to a small-sized image on massive parallel computers. For example, the YVRG filter uses a maximum of 512 processors in parallel in order to process a 512×512 image because the pixels in a line are bound by precedence relations. Most high-end NVIDIA graphic processors have more than 1,000 cores [13], of which about half are idle during the filtering process. This means that the YVRG filter cannot fully utilize graphic processors if the number of cores exceeds the image length. Although the basic Gaussian filter can fully utilize graphic processors, it requires a

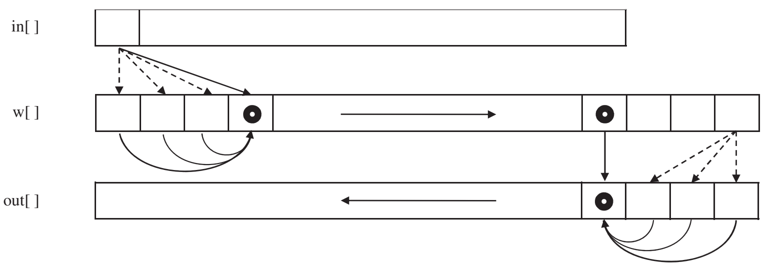

Figure 3 Border process in recursive Gaussian filter. The first element in[0] is duplicated and is used by first four elements of *w*[ ].

massive computation time when the filter window size is large. In order to solve such a dilemma, we refine the YVRG filter in order to increase the degree of parallelism by partitioning each line into three parts: two sub-lines and a point. The three parts execute in parallel in order to theoretically reduce process time by half and double the graphic processor's utilization.

System environments

Let us consider the hardware systems on which recursive Gaussian filters run. Currently, many types of parallel computers are available. At the standalone computer level, openCL based on multiple cores [14] and CUDA using NVIDIA graphic processors [15] are popular. At the networked computing system level, cluster computing, grid computing, and cloud computing systems are available [16]. Since image processing shares a heavy amount of data between nearby pixels, little communication overhead is required. Thus, standalone computer systems with multiple processing cores are preferable to network computing systems.

CUDA is a parallel computing platform. It requires GPUs produced by NVIDIA. CUDA is provided with libraries and a compiler called 'nvcc.' CUDA allows a programmer to make a general-purpose function called a kernel and run it in a core, a processing unit of an NVIDIA graphic card. The kernel function is mapped to a process unit called a thread that runs in a core. A block is composed of multiple threads and is mapped to a streaming multiprocessor (SM) that is composed of multiple CUDA cores. Since the number of cores in an SM is static, there should be a sufficient number of threads in a block to maximize GPU utilization.

In a traditional parallel computer, an optimal static mapping of threads into processors is possible. However, it is not a good idea to allocate a specific number of threads to an SM, since the CUDA cores' configuration in SM is different for each GPU model. CUDA provides an automatic scheduler for allocating threads to cores. Thus, a programmer only needs to allocate enough threads in a block in order to maximize the utilization of CUDA cores in SMs. The programmer also needs to take care of making enough number of blocks in order to keep SMs work as much as possible. After all, defining domain range per thread and the amount of threads per block determines the utilization of a CUDA GPU.

There are four levels of memory in CUDA: constant memory, local memory, shared memory, and global memory. Since the size and access times of memory levels are different, careful variable allocation is required.

Related work

If FIR filter is used in parallel computing systems, an input image can be partitioned into any types, for example, line-oriented, column-oriented, checkerboard-oriented partition [11,17], or bitwise allocation [2,18], because there is no precedence dependency between pixels. Although the FIR filter is highly suitable for parallelism, it requires large process time for big filter window because its processing step is linear to the filter window size. Ryu and Nishimura tried to reduce process time of FIR filter using a look-up table and integer-only operations [19]. Although the process time of the filter is reduced, the time complexity is still linear to the filter window size.

Recursive Gaussian filters exploit differences of previous nearby pixels in order to eliminate effect of filter window size from process time. Two representative recursive Gaussian filters are proposed by Deriche [20] and YVRG filter proposed by van Vliet et al. [12]. Since the filter window size is used not as an iteration number but as a parameter for calculating coefficients in the recursive computation, the filters have steady process time even if the filter window size becomes large. Process steps of the two recursive Gaussian filters are almost the same, except that Deriche's filter works better if the filter window size is less that 64 and YVRG filter works better otherwise [10].

However, the YVRG filter binds all pixels into a line because of the precedence dependencies of adjacent pixels. For example, $w[x, y]$ should be computed after $w[x-1, y]$ in a forward pass of a row-oriented step, and $\text{out}[x, y]$ should be computed after $\text{out}[x+1, y]$ in a backward pass.

Because of these dependencies, the parallel version of recursive Gaussian filters should partition an image into lines such that each processor computes a output line from an input line using an intermediate buffer $w[\ ]$, as shown in Figure 2. The degree of parallelism of the YVRG filter is $\min(v, h)$ where the image size is $v \times h$. If the number of available processors exceeds the degree of parallelism, the remainder of the processors become idle. For example, an NVIDIA graphic processor GTX780 has 2304 CUDA cores. Assume that a filter program processes a 512×512 image. While 512 CUDA cores are working, the other 1792 CUDA cores stay idle, leading to a 22.2% processor utilization.

A parallel version of a recursive Gaussian filter (one-way recursive Gaussian filter) is already included in the CUDA Toolkit. The version is a parallelized implementation of the YVRG algorithm proposed by Young and van Vliet [8]. This implementation partitions an image into lines and allocates each line to a thread. Thus, its parallelism is restricted by the height or width of the input image. Other papers have partitioned image domains in a similar manner while implementing similar algorithms [17,21,22].

Gaussian KD-Trees algorithm was proposed for accelerating a broad class of nonlinear filters like bilateral filters

[23]. It adapts three schemes to achieve less computation. One is ignoring interactions further than three standard deviations apart. Another is taking important sampling in which values are averaged with other values that are considered nearby. The other is computing the filter at a lower resolution and then interpolating the result. This method is superior to the previous grid approach [24] in terms of memory size and processing time aspects. However, it has a tree building overload and it is also difficult to implement tree traversal due to its extremely irregular algorithm that results in debilitating the advantage of GPU implementation.

Podlozhnyuk proposed an efficient parallel image convolution method that uses shared memory in CUDA [11]. Since values in the boundary pixels of a block should be exchanged between neighborhood blocks, a careful communication schedule is required. The paper proposes a schedule that enables data loading and filtering process concurrently. Although the method works well on FIR filter, it does not show the same performance if recursive Gaussian filters are used. Recursive Gaussian filters prefer temporary memory access. Thus, local memory is more efficient than shared memory in the case of recursive filters.

Proposed filter

Two-way partitioning

In order to increase the parallelism of a recursive Gaussian filter, we propose a two-way recursive Gaussian filter. The filter partitions an image in line-based orientation and partitions each line into three blocks again. The first and third blocks use the recursive Gaussian filter, while the second block uses a general Gaussian filter. One major problem is that all pixels in a line are related by a precedence dependency, as shown in Equations 3 and 4. However, we note two facts that circumvent the dependencies in a line:

- When out$[x, y]$ is computed in backward pass, out$[x+1, y]$ through out$[x+3, y]$ are required. Pixels out$[x+1, y]$ through out$[x+3, y]$ are the results of a row-oriented step. If the pixels have already been computed using the basic Gaussian filter, out$[x, y]$ has no precedence dependency on out$[x+k, y]$ where $k > 3$.
- There is no priority between forward and backward passes. Thus, any pass can start in any order.

These facts motivate us to partition a line into the following three parts:

- B_l: the left half of the line from index 0 to width/2$-$1, where *width* is the number of horizontal pixels in the picture.
- P_c: the pixel located at width/2.
- B_r: the right half of the line from index width/2 + 1 to width$-$1.

The proposed filter uses two passes for each step. In the first pass, the filter in block B_l works similarly to a one-way recursive Gaussian filter. It starts from the leftmost pixel in$[0, y]$ and generates intermediate data $w[\ ]$ until it arrives at index width/2 $-$ 1, as shown in Figure 4. The filter for pixel P_c computes the horizontal value of the basic Gaussian filter w[width/2, y] for pixel in[width/2, y] with its filter window size. Thus, the time complexity of P_c is depend on the filter window size. The filter in block B_r works similarly to the filter in block B_l but in the reverse direction. It starts from the rightmost pixel in[width $-$ 1, y] and generates

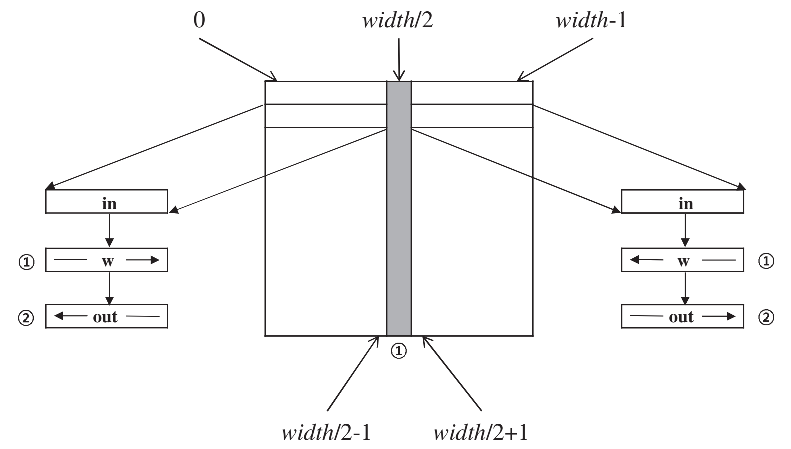

Figure 4 Partition to three blocks. Each line is divided into three blocks B_l (index range 0 $\sim$ width/2 $-$ 1), P_c (index width/2), and B_r (index range width/2 + 1 $\sim$ width $-$ 1).

intermediate data $w[\ ]$ until index width/2 + 1. In other words, the filter in block B_r starts from a backward pass.

Since forward and backward passes are used to reflect the left-side and right-side neighborhood pixels, respectively, the execution order of the forward and backward passes does not affect the filtering results. After finishing the first pass, that is, computation of $w[]$, each block starts a reverse directed pass. It is important to notice the filtering boundary pixels. Boundary pixels like $w[0, y]$, $w[1, y]$, and $w[2, y]$ need values $w[-3, y]$, $w[-2, y]$, and $w[-1, y]$. Since these values are not available, $w[0, y]$ is used instead. When the direction of computation is changed, pixel out[width/2 − 1, y] needs value out[width/2, y], out[width/2 + 1, y], and out[width/2 + 2, y]. Although these indexes have been computed in B_r, they are not the same values as in Equation 4 because those values are generated by the second pass while the values in B_r are generated by the first pass.

In order to solve this mismatch, pixel P_c, computed by the basic Gaussian filter, is used as a buffer zone. The computed value of pixel P_c, that is, out[width/2, y], is used instead of the three values out[width/2, y], out[width/2 + 1, y], and out[width/2 + 2, y] as shown in Figure 5. This is the reason for computing P_c concurrently with the other two blocks. In addition, block B_r uses pixel P_c in the same manner but in the inverse order. Figure 6 shows the parallelism of the proposed filter. Three processors can execute in parallel in the first pass and two processors can execute in parallel in the second pass.

Algorithm 1 shows the detailed process of row-oriented step of the proposed two-way filter. Three threads are assigned to each row calculated in line 1. Thus a block takes charge of adjacent lines in an image. For example, if four lines are allocated to each block (blockDim.x = 4), the third block filters line 12 ∼ 15. The three threads are identified by pre-assigned value *threadIdx.y*. If a thread has index (blockIdx.x = 2, threadIdx.x = 2, threadIdx.y = 1), the thread tasks charge of central pixel P_c of line 10. Neighborhood values for boundary pixels are initialized at line 3, 9, 14, and 21. Intermediate array w[row] [] is divided to two sub arrays w_l and w_r in order to prevent race condition in w[row] [width/2−2] ∼ w[row] [width/2+2],

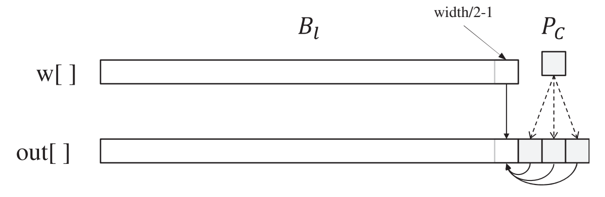

Figure 5 Start of reverse pass. Pixel out[width/2 − 1] is computed using pixel w[width/2 − 1], out[width/2], out[width/2 + 1], and out[width/2 + 2] which are copied from P_c.

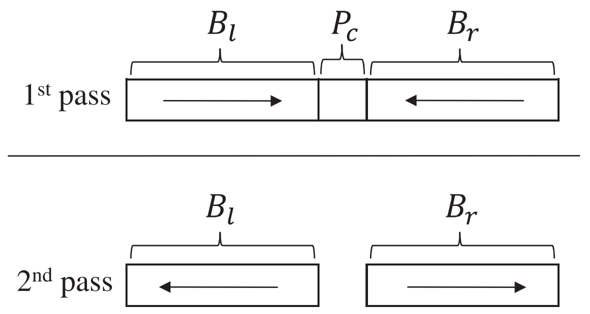

Figure 6 Parallelism of the row-oriented step. In the first pass, B_l, P_c, and B_r are computed in parallel. In the second pass, inverse directions of blocks B_l and B_r are computed in parallel.

which are overlapped and are accessed concurrently by two threads.

The time complexity of the proposed filter is determined by two factors: line width and filter window size. Assuming that there are enough processors, three processors are allocated to each line. If l = max(width, height), then $3l$ processors are required in order to maximize the performance. A processor for block B_l or B_r computes half of the $l/2$ pixels in a pass. Since each pixel requires eight multiplications and four additions, $6l$ operations are required for each pixel. The center pixel P_c requires $3N$ operations, where $N \times N$ is the filter window size. Thus, max($6l$, $3N$)+$6l$ operations are required per step and max($12l$, $6N$)+$12l$ steps are required during the proposed filter. In short, the process time of the filter is halved or speedup is doubled compared to those of a one-way recursive Gaussian filter if the number of cores is equal to or greater than $3l$.

Double speedup with the cost of triple cores is not highly efficient. The proposed two-way recursive Gaussian filter is useful if lots of small-sized images compared to the number of cores should be processed in sequence. An typical example is the application that filters multiple different windows of an image in order to find the best parameters.

Table 1 Experimental environment

	Description
CPU	Intel core i5 750 (2.67 GHz)
GPU	Geforce GTX 670 (GK104)
	Compute capability, 3.0
	Number of CUDA cores 1,344
	Graphic clock, 915 MHz
	Processor clock, 980 MHz
Compiler	Visual Studio 2010
Library	CUDA 5.0
Image	512 × 512 bitmap

Compile and execution environment.

Algorithm 1: TWO-WAY RECURSIVE GAUSSIAN FILTER, row-oriented step

```
Input: An 2-d image array in[ ][ ] with $height \times width$, filter window size $N$
Output: An filtered 2-d image array out[ ][ ] with the same size of in[ ][ ]
row ← $blockDim.x * blockIdx.x + threadIdx.x$
if $threadIdx.y = 0$ then                                   /* left side block $B_l$ */
    $w_l[row][2], w_l[row][1], w_l[row][0] \leftarrow in[row][0]$
    for $i \leftarrow 3$ to $width/2 - 1$ do
        $w_l[row][i] \leftarrow B * in[row][i] + (b_1 * w_l[row][i-1] + b_2 * w_l[row][i-2] + b_3 * w_l[row][i-3])/b_0$
else if $threadIdx.y = 1$ then                              /* center pixel $P_c$ */
    $w_l[row][width/2], w_r[row][width/2] \leftarrow bitwiseRowGaussianFilter(in, row, width/2, N)$
else                                                        /* right side block $B_r$ */
    $w_r[row][width], w_r[row][width+1], w_r[row][width+2] \leftarrow in[row][width-1]$
    for $i \leftarrow width - 1$ downto $width/2 + 1$ do
        $w_r[row][i] \leftarrow B * in[row][i] + (b_1 * w_r[row][i+1] + b_2 * w_r[row][i+2] + b_3 * w_r[row][i+3])/b_0$
synchronize threads
if $threadIdx.y = 0$ then                                   /* left side block $B_l$ */
    $w_l[row][width/2+1], w_l[row][width/2+2] \leftarrow w_l[row][width/2]$
    for $i \leftarrow width/2 - 1$ downto $0$ do
        $w2_l[row][i] \leftarrow B * w_l[row][i] + (b_1 * w_l[row][i+1] + b_2 * w_l[row][i+2] + b_3 * w_l[row][i+3])/b_0$
        $out[row][i] = w2_l$
else if $threadIdx.y = 1$ then                              /* center pixel $P_c$ */
    do nothing
else                                                        /* right side block $B_r$ */
    $w_r[row][width/2-1], w_r[row][width/2-2] \leftarrow w_r[row][width/2]$
    for $i \leftarrow width/2 + 1$ to $width - 1$ do
        $w2_r[row][i] \leftarrow B * w_r[row][i] + (b_1 * w_r[row][i-1] + b_2 * w_r[row][i-2] + b_3 * w_r[row][i-3])/b_0$
```

Experimental results

The proposed two-way recursive Gaussian filter was implemented in the C programming language using the CUDA library. The experimental environment is shown in Table 1. The experiment used the image commonly known as Lena, shown in Figure 7. For comparison, a parallel version of the one-way recursive Gaussian filter proposed by Young and van Vliet was also implemented.

Gaussian filters usually have been used to de-noise in differential operations for image processing or to achieve image blurring effects. For the differential operations, the typical filter window size is 3×3. To obtain a proper blurring effect, we should carefully determine the filter window size according to image sizes. One of the thumb rules suggests 1%∼5% of the image size for the Gaussian half-width σ and two times of σ for the filter window size [25]. If we take 3% of σ for a 512×512 resolution image, the filter window size becomes about 30. Bilateral filters require 10% of σ, then the mask size becomes 102 for the same image resolution. The larger image size requires much larger mask size. In this paper, we take 3×3 and 30×30 mask size for Lena image.

Two filter window sizes, 3 and 30, were used, as shown in Figure 8. Filter window size 3 is the minimum size for

Figure 7 Data image. 512×512-sized image.

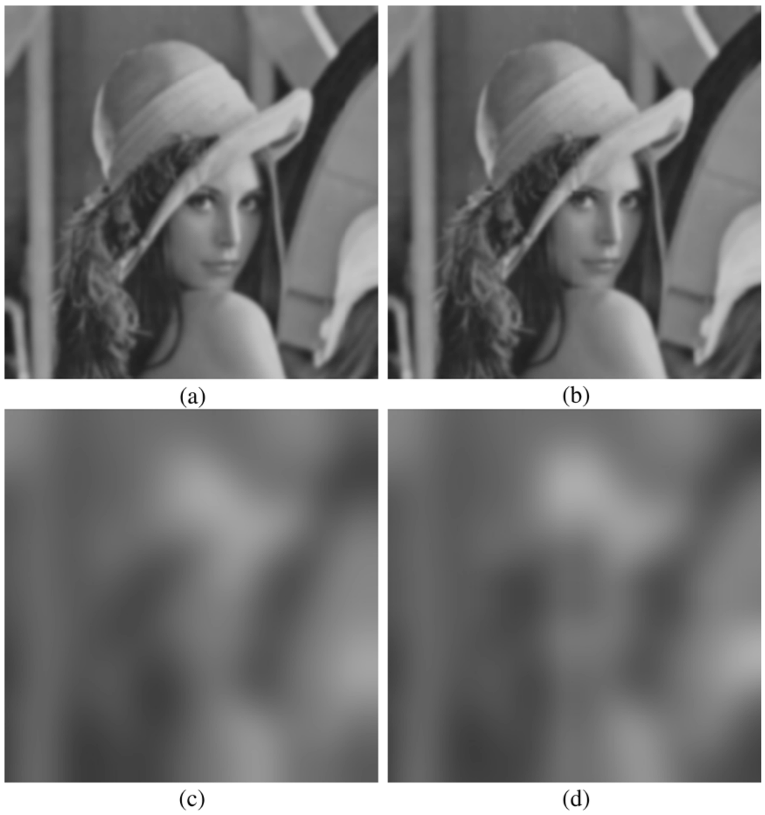

Figure 8 Filtered images. Comparison of the recursive Gaussian filter and proposed two-way recursive Gaussian filter: **(a)** the recursive Gaussian filter with filter window size 3, **(b)** the proposed two-way recursive Gaussian filter with filter window size 3, **(c)** the recursive Gaussian filter with filter window size 30, and **(d)** the proposed two-way recursive Gaussian filter with filter window size 30.

Gaussian filter because only adjacent pixels are used for filtering. Hale proposed 64 as a boundary filter window size for choosing among different two recursive Gaussian functions: smaller window size for Deriche's algorithm [20] and bigger window size for van Vliet et al.'s YVRG algorithm [12]. Since the proposed algorithm is based on van Vliet et al.'s algorithm, filter window size 30 is chosen as a medium stable value. The original image was changed to gray scale in order to analyze the results quantitatively.

Validity of the proposed filter

The peak signal-to-noise ratio (PSNR) was used for quantitative analysis. PSNR is computed as follows:

$$\mathrm{MSE} = \frac{1}{mn}\sum_{i=0}^{m-1}\sum_{j=0}^{n-1}\left[I(i,j) - K(i,j)\right]^2$$

$$\mathrm{PSNR} = 10\log_{10}\frac{\mathrm{MAX}_I^2}{\mathrm{MSE}}, \qquad (5)$$

where MSE is the mean squared error, m is the width of the images, n is the height of the images, I and J are the two compared images, and MAX_I is the maximum value of a pixel in image I. Since the images are converted to gray scale for the comparison, MAX_I is 255. The smaller the difference between the two images is, the smaller the MSE is. Thus, a large PSNR indicates that the two images are similar. Table 2 shows the PSNR values between a one-way recursive, two-way recursive, and basic Gaussian filter. For image comparison, a tolerable PSNR range is between 30 and 50 dB. It is known that two images are not easy to distinguish with the naked eye if the PSNR is 30 dB or more. Although the PSNR between the results of one-way and two-way recursive Gaussian filters is smaller than 30 dB when the filter window size is 3, the one-way recursive Gaussian filter has a worse PSNR value when it is compared to the basic Gaussian filter. The table shows that the result of a two-way recursive Gaussian filter is closer to that of the basic Gaussian filter. Thus, the proposed two-way recursive Gaussian filter is usable in the general case.

Performance comparison

Measure using cudaEventRecord()

Since the parallel CUDA code runs in a GPU, general clock measure functions like `gettimeofday()` or `clock()` cannot measure its process time correctly.

Table 2 PSNR comparison

Image size	Filter window size	One-way vs. two-way	FIR vs. one-way	FIR vs. two-way
SD	3	29.4	26.2	29.7
	30	31.1	32.8	31.4
	90	29.5	33.4	29.4
HD	3	33.0	30.9	33.6
	30	30.6	39.7	30.8
	90	24.7	33.0	26.3
Full HD	3	30.5	26.6	29.2
	30	32.9	36.7	33.2
	90	30.1	34.8	31.4

PSNR between three images: non-recursive Gaussian filter (the basic Gaussian filter), one-way recursive Gaussian filter, and proposed two-way recursive Gaussian filter. Unit of the PSNR value is dB (decibel).

CUDA provides the cudaEvent family of functions in order to measure process time in a GPU.

Before the process times are measured, the allocation of threads to blocks should be considered in order to maximize parallelism. A *thread* is a logical process unit in CUDA. The *kernel code* is a description of a thread and a *block* is a set of threads. The maximum number of threads in a block is 1,024, in the case of the GeForce GTX 670 GPU. However, this does not mean that all 1,024 threads run concurrently in a block. Although a SM is mapped to a block in CUDA and each SM contains 32 processing cores [26], this does not mean that 32 threads in a block will maximize the parallelism. Because of this uncertainty, we decided to allocate various numbers of threads to blocks to find an allocation that minimizes the process time. Figure 9 shows the process times of different numbers of allocated threads. The x-axis indicates the number of allocated lines per block, and the y-axis indicates the process time in the GPU kernel. One-way recursive Gaussian filter allocates one thread per line, and the proposed two-way recursive Gaussian filter allocates three threads per line. The one-way recursive Gaussian filter has the minimum process time when the number of threads per block is 8 as shown in Figure 9a. Filter window size $N \times N$ does not affect the process time in the recursive Gaussian filters, as we expected. Thus, filter window size 30×30 is used for the following experiments.

The proposed two-way recursive Gaussian filter has the minimum process time when the number of threads per block is 16 as shown in Figure 9b. In Figure 9a, the process times increase as the lines per block exceed 8. Figure 9b shows the similar anomaly from 32 lines per block. The main reason of the larger process time in the case of the smaller number of lines per block is that the number of threads is not sufficient to fully utilize cores allocated to each block. As the number of lines per block increases, utilization of cores in each block increases, but the number of blocks decreases. If the number of blocks is not sufficient, blocks are not scheduled evenly. If 32 lines of an image with 512 lines are allocated to each block, 16 blocks should be allocated 7 SMs of NVIDIA GPX670 graphic card. Then a SM runs 2 or 3 blocks, which makes imbalance between SMs.

From the two process times, the speedup of the proposed two-way filter over the one-way filter is shown in Figure 10a, where the speedup value is double if the number of threads per block is greater than or equal to 16. Also, the figure shows that overheads invoked by sequential global memory access and FIR filter computation in the center point P_c do not affect performance of the two-way filter.

Measure using Nsight

Nsight is a development tool provided by NVIDIA and it tells occupancy of parallel applications in GPU [27]. Thus, Nsight can be used to investigate GPU utilization.

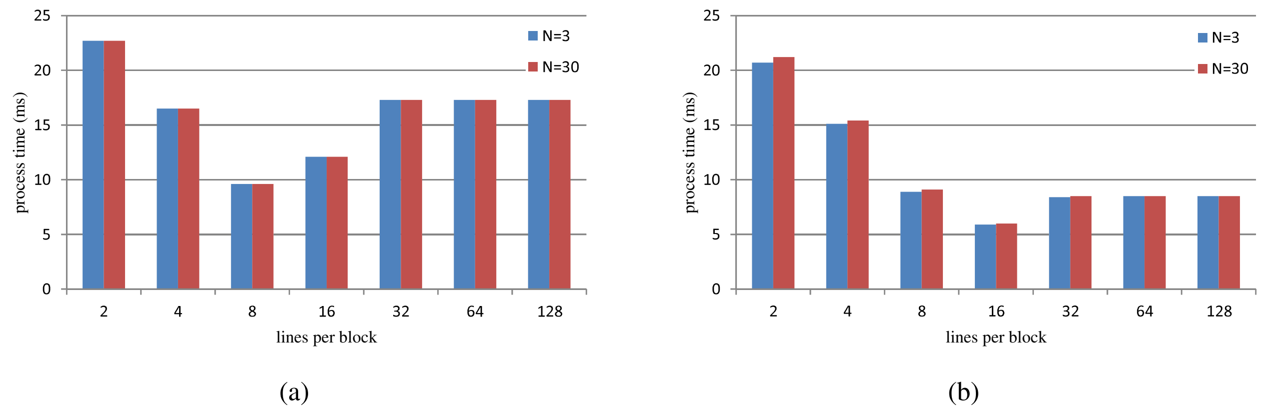

Figure 9 Process times of recursive Gaussian filters and speedup. Process time measured by cudaEventRecord() function. **(a)** One-way recursive Gaussian filter and **(b)** proposed two-way recursive Gaussian filter.

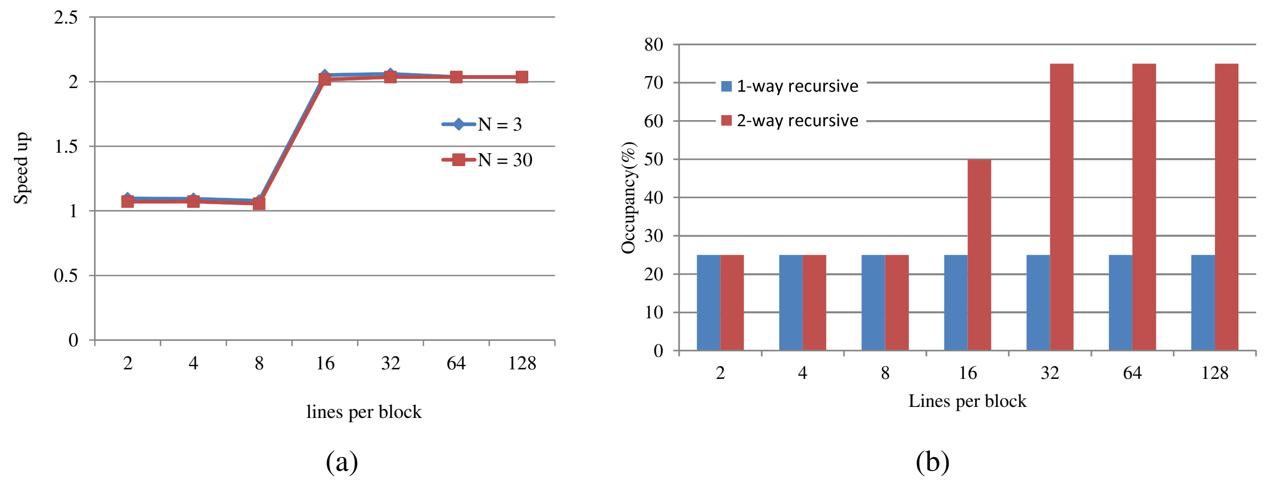

Figure 10 Speed up and occupancy comparison of recursive Gaussian filters. (a) Speed up of the proposed two-way recursive Gaussian filter against one-way recursive Gaussian filter. **(b)** Occupancies of recursive Gaussian filters.

Occupancy is the percentage of active warps against the maximum active warps, where a warp means a group of 32 threads. The amount of active warps is decided by the number of allocated threads per block, amount of available registers, and amount of shared memory. Allocating more threads to block increases occupancy.

Figure 10b shows an occupancy comparison between one-way recursive Gaussian filter and two-way recursive Gaussian filter. Since the total number of threads is fixed to data image width or height, that is, 512 in the case of Lena image, many CUDA cores are idle and it makes the CUDA occupancy at the low value. On the other side, two-way recursive Gaussian filter increases its occupancy as the number of lines per block increases over 16 as shown in Figure 10b.

Improvement using local memory

Figure 9 shows that performances of recursive Gaussian filters degrade as the number of lines per block exceeds 16. The imbalanced block allocation to SMs and sequential global memory access are assumed to be the main reasons of the degradation. Since the balanced allocation policy changes by different graphic card models, finding efficient memory access is focused in order to improve the proposed filter. CUDA provides following four-level hierarchical memory model: constant memory, per-thread local memory, per-block shared memory, and global memory [28]. Constant memory is used to store constant values. Thus, constants in Equations 3 and 4 are stored in constant memory. Image data is stored in the global memory in order to be accessed by all threads. During a forward pass, each pixel in array in[ ] is read once or twice according to filter window size. Thus it is not highly required to copy it in shared or local memory. Since array $w[\ ]$ is generated and frequently accessed by each thread, it is good target for locating in local memory. Pixel P_c is read six times by adjacent two threads. Since P_c is stored in cache of the thread after being read, its memory location does not highly affect the performance of the filtering. As the result, major variables are located in each memory as follows:

- Constant memory: constants for Equations 3 and 4
- Per-thread local memory: $w[\ ]$, temporary memory for thread
- Global memory: in[ ], out[ ] (including P_c)

Figure 11 shows the improved process time by moving temporary array $w[\ ]$ from global memory to local memory. Note that the process time do not increase

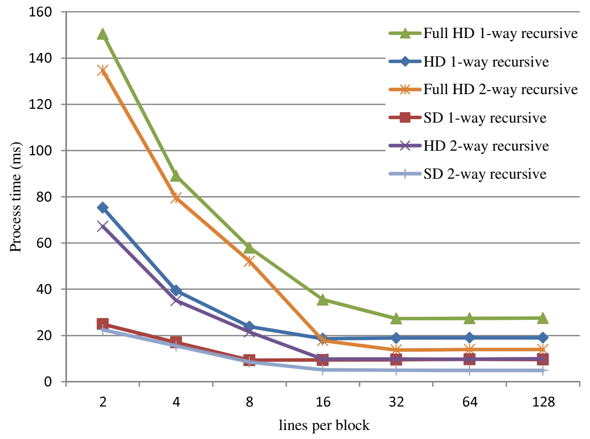

Figure 11 Process time improvement using local memory. Improved process time by allocating intermediate array variable $w[\ ]$ to local memory. Filter window size is set to 30 × 30. SD sized image is 512 × 512 Lena.bmp file, HD sized image is 1280 × 720 Mountain.bmp file, and Full HD sized image is 1920 × 1080 Lake.bmp file.

although the number of lines per block increases, which is a different tendency compared to the case of global memory access shown in Figure 9. The imbalanced block allocation to SMs still exists even in the case of the local memory access. However, local memory can be accessed in parallel by each thread while global memory should be accessed sequentially by blocks. As the result, the effect of imbalance allocation is assumed to be reduced by the local memory access. In order to prove the assumption, process time in each thread is analyzed as shown in Table 3. A process time is mainly composed of data request time on memory, execution time of instructions, and synchronization time. If the data request time is subtracted from the process time, remaining execution and synchronization times become similar for both memory access cases.

Figure 11 includes process time of HD and full HD-sized images. One-way recursive Gaussian filter is also improved by local memory access. Process times are stabilized after 32 lines per block. Two-way recursive Gaussian filter has 1.96, 1.90, and 1.98 speedup compared to one-way recursive Gaussian filter in the case of SD, HD, and full HD images, respectively. The figure shows the proposed two-way recursive Gaussian filter generates steady speedup and performance against bigger images.

Figure 12 shows performance comparison between a CPU-based implementation and the GPU-based implementation. GPU-based implementation configures the number of lines per block as 32. Multi-threaded two-way recursive Gaussian filter is implemented on a PC with Intel Quad Core i5-750 Processor 2.67 GHz. Although the CPU-based implementation has better performance on SD-sized image, the situation is reversed when HD and full HD-sized image is used.

Conclusions

In this paper, we have noticed that line-oriented recursive Gaussian filter can be partitioned more and proposed a two-way recursive filter that increases CUDA GPGPU utilization. The proposed filter divides each line into two sub-lines and a central point. The central point is used to compensate mismatches occurred by dividing a line into two parts. PSNR shows that the quality of the filter locates between non-recursive Gaussian filter and the one-way line-oriented recursive Gaussian filter. The process time of the proposed filter is reduced to half by setting the number of lines per block to the same or greater than 16.

The research can be expanded by considering following various concerns. Starting from the proposed two-way recursive Gaussian filter, a line can be partitioned to three or more blocks, where quality and speedup are major concerns. Another consideration is the central points. The central points or boundary points between blocks in the partitioned line can be designed differently. The central points are required as boundary points in each block. If recursive equations are designed carefully, it could be possible to partition a line without any central point or boundary point.

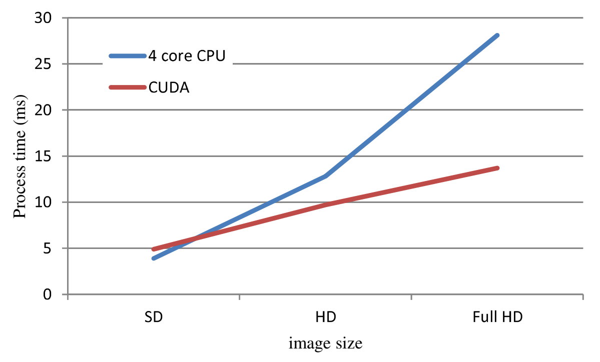

Figure 12 Performance comparison between 4 core CPU and 1344 core GPU. The proposed two-way recursive Gaussian filter is executed. Intel i5 with four cores uses four threads that fully utilize the multi-core CPU. CUDA configures 64 lines per block.

Table 3 Kernel process time components

Lines per block	Process time (ms)		Data request percentage		Execution and sync (ms)	
	Local	Global	Local	Global	Local	Global
4	15.2	15.1	1.2%	12.3%	15.0	13.5
8	8.4	8.9	2.9%	25.2%	8.2	6.8
16	5.1	5.9	6.1%	41.9%	4.8	3.5
32	4.9	8.4	10.2%	48.9%	4.4	4.3
64	5.1	8.5	10.2%	48.9%	4.4	4.3

Extraction of execution times and synchronization time among entire processing time in the case of two-way recursive Gaussian filter: 'Data request' means the percentage of memory access time among the entire filtering process. 'Execution and Sync' means the processing time where memory access time is subtracted.

Competing interests
The authors declare that they have no competing interests.

Authors' contributions
The work presented here was carried out in collaborated process of all authors. JPK and TYC defined the research theme. CWL and TYC designed methods, algorithm, and implementation, and carried out the laboratory experiments. CWL, JPK, and TYC analyzed the data, interpreted the results, and wrote the paper. All authors have contributed to inspection and approved the manuscript.

Received: 15 November 2013 Accepted: 10 June 2014
Published: 30 June 2014

References
1. J Canny, A computational approach to edge detection. IEEE Trans. Pattern Anal. Mach. Intell. **8**(6), 679–698 (1986)
2. Y Luo, R Duraiswami, Canny edge detection on NVIDIA CUDA, in *IEEE Computer Society Conference on Computer Vision and Pattern Recognition Workshops, 2008. CVPRW'08* (IEEE Anchorage, AK, USA, 2008), pp. 1–8
3. DG Lowe, Distinctive image features from scale-invariant keypoints. Int. J. Comput. Vis. **60**(2), 91–110 (2004)
4. EH Land, The Retinex theory of color vision. Sci Am. **237**(6), 108–128 (1977)
5. DJ Jobson, Z Rahman, GA Woodell, Properties and performance of a center/surround retinex. IEEE Trans. Image Process. **6**(3), 451–462 (1997)

6. DJ Jobson, Z-U Rahman, GA Woodell, A multi-scale retinex for bridging the gap between colour images and the human observation of scenes. IEEE Trans. Image Process. **6**(7), 965–976 (1997)
7. R Haralick, L Shapiro, *Computer and Robot Vision*. (Addison-Wesley, Boston, USA, 1992)
8. IT Young, LJ van Vliet, Recursive implementation of the Gaussian filter. Signal Process. **44**, 139–151 (1995)
9. LG Shapiro, GC Stockman, Image smoothing, in *Computer Vision* (Prentice Hall Upper Saddle River, NJ, USA, 2001), p. 137
10. D Hale, Recursive gaussian filters. CWP-546 (2006). http://www.cwp.mines.edu/Meetings/Project06/cwp546.pdf
11. V Podlozhnyuk, Image convolution with CUDA. NVIDIA Corporation White Paper, June. **2097**(3) (2007). https://cluster.earlham.edu/trac/bccd-ng/export/2037/branches/cuda/trees/software/bccd/software/cuda-0.2.1221/sdk/projects/convolutionSeparable/doc/convolutionSeparable.pdf
12. LJV Vliet, IT Young, PW Verbeek, Recursive Gaussian derivative filters, in *Proceedings of the Fourteenth International Conference on Pattern Recognition, 1998*, vol. 1 (IEEE Brisbane, Queensland, Australia, 1998), pp. 509–514
13. NVIDIA Corporation, White Paper NVIDIA GeForce GTX 680 (2012). http://www.geforce.com/Active/en_US/en_US/pdf/GeForce-GTX-680-Whitepaper-FINAL.pdf
14. P Jaaskelainen, CS de La Lama, P Huerta, JH Takala, OpenCL-based design methodology for application-specific processors, in *2010 International Conference on Embedded Computer Systems* (IEEE Samos, Greek, 2010), pp. 223–230s
15. CUDA Parallel computing platform. http://www.nvidia.com/object/cuda_home_new.html, Jan. 2014
16. I Foster, Y Zhao, I Raicu, S Lu, Cloud computing and grid computing 360-degree compared, in *Grid Computing Environments Workshop, 2008. GCE'08* (IEEE, 2008), pp. 1–10
17. Y Su, Z Xu, X Jiang, GPGPU-based Gaussian filtering for surface metrological data processing, in *IV* (IEEE Computer Society, 2008), pp. 94–99. http://dx.doi.org/10.1109/IV.2008.14
18. A Trabelsi, Y Savaria, A 2D Gaussian smoothing kernel mapped to heterogeneous platforms, in *2013 IEEE 11th International New Circuits and Systems Conference (NEWCAS)* (IEEE Paris, France, 2013), pp. 1–4
19. J Ryu, TH Nishimura, Fast image blurring using Lookup Table for real time feature extraction, in *IEEE International Symposium on Industrial Electronics, 2009. ISIE 2009* (IEEE Seoul, Korea, 2009), pp. 1864–1869
20. R Deriche, Recursively implementating the Gaussian and its derivatives (1993). http://hal.archives-ouvertes.fr/docs/00/07/47/78/PDF/RR-1893.pdf
21. Y Ma, K Xie, M Peng, A parallel Gaussian filtering algorithm based on color difference, in *IPTC* (IEEE, 2011), pp. 51–54. http://ieeexplore.ieee.org/xpl/mostRecentIssue.jsp?punumber=6099690
22. D Nehab, A Maximo, RS Lima, H Hoppe, GPU-efficient recursive filtering and summed-area tables. ACM Trans. Graph. **30**(6), 176:1–176:11 (2011)
23. A Adams, N Gelfand, J Dolson, MsLevoy, Gaussian KD-trees for fast high-dimensional filtering, in *ACM Transactions on Graphics (TOG), Volume 28* (ACM, 2009), p. 21
24. S Paris, F Durand, A fast approximation of the bilateral filter using a signal processing approach, in *Computer Vision–ECCV 2006* (Springer Graz, Austria, 2006), pp. 568–580
25. SK Park, BS Kim, EY Chung, KH Lee, A new illumination estimation method based on local gradient for retinex, in *IEEE International Symposium on Industrial Electronics, 2009. ISIE 2009* (IEEE Seoul, Korea, 2009), pp. 569–574
26. J Luitjens, S Rennich, CUDA warps and occupancy. GPU Computing Webinar (2011). http://on-demand.gputechconf.com/gtc-express/2011/presentations/cuda_webinars_WarpsAndOccupancy.pdf
27. NVIDIA Nsight Visual Studio Edition. https://developer.nvidia.com/nvidia-nsight-visual-studio-edition June 2014
28. NVIDIA, Programming model, in *CUDA C Programming Guide, Design Guide* (NVIDIA, 2013), pp. 12–13. http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#axzz34o3nNnUF June 2014

doi:10.1186/1687-5281-2014-33
Cite this article as: Lee *et al.*: Two-way partitioning of a recursive Gaussian filter in CUDA. *EURASIP Journal on Image and Video Processing* 2014 **2014**:33.