IJAMEC
e-ISSN: 2147-8228

INTERNATIONAL JOURNAL OF APPLIED MATHEMATICS ELECTRONICS AND COMPUTERS

www.dergipark.org.tr/ijamec

International Open Access

Volume 08
Issue 01

March, 2020

Research Article

CUDA Based Computation of Quadratic Image Filters

Devrim Akgun [a,*], ***Süleyman Uzun*** [b]

[a]*Software Engineering, Sakarya University, Sakarya, 54187, Turkey*
[b]*Computer Engineering, Bilecik Şeyh Edebali University, Bilecik, 42002, Turkey*

ARTICLE INFO

Article history:
Received 28 November 2019
Accepted 25 January 2020

Keywords:
CUDA
GPU computing
Quadratic Image Filter
Volterra filter

ABSTRACT

Image processing applications usually requires nonlinear methods due to the nonlinear characteristics of images. Quadratic image filter which is a class of nonlinear image filters are widely used in practice such as noise elimination edge detection and image enhancement. On the other hand, second order products of the pixels make quadratic image filters computationally expensive to implement when compared to linear convolution. In the last decade, CUDA accelerated computing has been widely used in image processing applications to reduce computation times. In this study, an efficient method for the CUDA acceleration of the quadratic image filter has been implemented. For this purpose, alternative algorithms were examined comparatively since the performance of the GPU is sensitive to memory utilization. Because quadratic filter has a large number of coefficients and quadratic terms, the algorithm which utilizes the shared memory for storing image blocks provided the best throughput among the examined methods. Comparative results that were obtained using various images in different sizes show significant accelerations over sequential implementation.

1. Introduction

Image processing applications such as noise filtering, edge detection, and image enhancement are usually implemented using convolution or correlation operations. Convolution based image filtering involves moving a filtering kernel over the whole image to compute the pixels of output image. Each pixel is computed by multiplying the selected window of pixels from the input image with a filtering kernel. Then the multiplication results are summed and the resulting value is written to target pixel. [1]. Due the nonlinear structure of image contents, performance of linear filters may not be satisfactory in some applications. In such cases, nonlinear filters can be preferred over linear filters [2]. Theoretically, Volterra series approach is usually used to model nonlinear systems using infinite elements. In practice, Volterra model of a nonlinear system is defined by truncating it to a reasonable size. Usually, the preferred approach is to truncate it to include up to second order terms to reduce computational complexity and these filters are usually called as Quadratic Image Filter (QIF) [3]. QIFs are utilized in various research fields such as Gaussian noise or impulsive noise removal due to its edge preserving features [4]–[6]. QIFs are successful in edge detection applications [7], [8], medical image processing applications such as enhancement or noise reduction for mammogram images [9]–[15]. QIFs are utilized for feature extraction to use in face recognition applications [16]–[18]. QIFs requires to compute second order multiplications for computing a filtered pixel [19]–[21]. Hence, required computations considerably larger than a convolutional filter of the same window size. For example, a QIF application using 3×3 window requires additional 45 number of multiplications for obtaining all possible second degree of multiplications of the input pixels. Also, another 45 number of multiplications for window weights are required. On the other hand linear convolution only requires 9 number of multiplications with windows weights. Recent GPU (Graphics Processing Unit) products provides a good mean to accelerate computation of image processing applications. For this purpose, NVIDIA provides CUDA (Compute Unified Device Architecture) model to utilize

* Corresponding author. E-mail address: *dakgun@sakarya.edu.tr*
DOI: 10.18100/ijamec.652564

GPU efficiently. CUDA enables writing programs for high performance parallel applications. Various image processing algorithms that can be parallelized utilizes the GPU technology for acceleration [22]–[28]. Since pixel the operations of QIF are independent, the algorithm can be parallelized to run on GPU environment.

The focus of the present study is to develop an efficient algorithm based on CUDA for the GPU accelerated computing of QIFs. Sequential implementation of the QIFs usually require long execution durations when running times are investigated on an average desktop processor. Hence, an efficient CUDA based implementation may help utilization of QIFs practical. GPU acceleration is highly dependent on the programming approach and the utilization of GPU memory. Efficient approaches usually requires more complicated algorithm designs and programming approaches. Hence, three alternative implementations from simple to complex were discussed for comparison. As will be shown by experimental results the proposed method provides significant reductions in computation time when compared to sequential execution. Organization of the paper is as follows; in the second section background information for QIFs was given. In the third section, implementation of the alternative algorithms using CUDA kernels were explained in detail. In the fourth section, experimental execution times using sequential implementation and CUDA implementations were presented. Finally conclusions about the method and the results were given.

2. Background

QIFs, which are in the subclass of Volterra filters are the important alternatives of the linear filters[3], [29]. Eq-1 describes the output equation of the QIF. In this study, only the terms up to second degree were used and the constant term is excluded;

$$o(m,n) = o_1(m,n) + o_2(m,n) \tag{1}$$

In above equation, m and n show the pixel coordinates to be filtered and $o(m,n)$ shows the filtered output pixel. The linear component of the output is represented by $o_1(m,n)$ and the quadratic part is represented by $o_2(m,n)$. Eq-2 shows the expressions for the outputs $o_1(m,n)$ and $o_2(m,n)$.

$$\left.\begin{array}{c} o_1(m,n) = \sum_{i=-M}^{M}\sum_{j=-M}^{M} w_{i,j}^1\, x_{m+i,n+j} \\ o_2(m,n) = \sum_{i=-M}^{M}\sum_{j=-M}^{M}\sum_{k=-M}^{M}\sum_{l=-M}^{M} \\ w_{i,j,k,l}^2 x_{m+i,n+j} x_{k+i,l+j} \end{array}\right\} \tag{2}$$

Where M means that a window size of $(2M+1)\times(2M+1)$ is used for filtering. $x_{m+i,n+j}$ represents a pixel taken from the input image, $w^1_{i,j}$ and $w^2_{i,j,k,l}$ represents first order and second order filter weights respectively. The equations for linear and quadratic components can be expressed simpler. For this purpose, linear part of the expression can be redefined using one dimensional summation as given by Eq-3.

$$o_1(m,n) = \sum_{i=0}^{N\times N-1} W_1(i) X_{m,n}^1(i) \tag{3}$$

Where N is equal to $2M+1$. W_1 describes weights, $X_{m,n}$ is the selected window of pixels as given by Eq-4.

$$\left.\begin{array}{l} W_1 = [w_0^1\ \ w_1^1\ \ w_2^1\ \ \dots\ w_{NxN-1}^1] \\ X_{m,n}^1 = [x_0\, x_1\, x_2 \dots\ x_{NxN-1}] \end{array}\right\} \tag{4}$$

For further simplification, above expression can be written as a dot product as below;

$$o_1(m,n) = W_1 {X_{m,n}^1}^T \tag{5}$$

Similar to above arrangements, the quadratic component can be simplified. Initially it is useful to redefine it in two dimensional form as below;

$$o_2(m,n) = \sum_{i=0}^{N\times N}\sum_{j=0}^{N\times N} w_{i,j}^2 X_{m,n}^1(i) X_{m,n}^1(j) \tag{6}$$

Most of the multiplications of input pixels in Eq-6 is symmetric. This is shown by Eq-7 where the quadratic input terms forms a matrix of $(N\times N)\times(N\times N)$ size.

$$X_{m,n}^1 {X_{m,n}^1}^T = \begin{bmatrix} x_0x_0 & x_0x_1 & \dots & x_0x_{N\times N-1} \\ x_1x_0 & x_1x_1 & \dots & x_1x_{N\times N-1} \\ \vdots & \vdots & \vdots & \vdots \\ x_{N\times N-1}x_0 & x_{N\times N-1}x_1 & \dots & x_{N\times N-1}x_{N\times N-1} \end{bmatrix} \tag{7}$$

In order to eliminate symmetric terms in Eq-6, the initial value of the j is set to i as shown by Eq-8;

$$o_2(m,n) = \sum_{i=0}^{N\times N}\sum_{j=i}^{N\times N} w_{i,j}^2 X_{m,n}^1(i) X_{m,n}^1(j) \tag{8}$$

The second order products of input given by Eq-8 do not contain symmetric multiplications. Therefore, the unique multiplications can be rewritten in vector form;

$$X_{m,n}^2 = [x_0x_0\ \ x_0x_1\ \cdots\ x_{NxN-1}x_{NxN-1}] \tag{9}$$

Also the corresponding weights used in Eq-8 can be written in vector form as below,

$$W_2 = [w_0^2\ \ w_1^2\ \ w_2^2\ \dots\ w_B^2] \tag{10}$$

Therefore, the quadratic part of the Eq-1 can be expressed as a dot product as below;

$$o_2(m,n) = W_2 {X_{m,n}^2}^T \tag{11}$$

The number of elements in W_2 and $X^2_{m,n}$ depends on the size of the filter kernel. The number of terms in quadratic vector excluding the symmetric terms for $N\times N$ size of kernel can be calculated by Eq-12;

$$B = N \times N(1 + N \times N)/2 \tag{12}$$

The sum of linear and quadratic part gives the total equation to filter a pixel as shown by Eq-13.

$$o(m,n) = W_1 {X_{m,n}^1}^T + W_2 {X_{m,n}^2}^T \tag{13}$$

3. Proposed implementation

In the present study, three alternative method for the CUDA implementation of the QIF were investigated. Before addressing the most efficient method described in this study, the two alternative approaches were also discussed for comparison. First method which is the most straightforward implementation shown by Algorithm 1.

Algorithm 1: IMPLEMENTATION OF METHOD 1 FOR QIF KERNEL

Input: *inputImage*: Image to be filtered, *W*: Weights
Output: *outputImage* : Filtered image

```
Get thread id to determine pixel Id
pixelId ← blockDim.x * blockIdx.x + threadIdx.x
Get 3x3 mask to thread local memory
threadLocInImg ← get3x3frame(inImg, pixelId)
Apply filtering to selected pixel
y ← quadFilt(W, threadLocInImg, pixelId)
Write results to output image (repeat for RGB)
outputImage(pixelId) ← y
```

Every CUDA thread executes the kernel given by Algorithm 1 and filters a pixel selected according to the block and thread number. As explained in the previous section, once GPU initialized the data that kernel uses transferred to global memory. Before filtering function is executed, the neighborhood of selected pixel from the global memory is copied to global memory to reduce repeated reads from the global memory. Then, kernel filters a pixel selected according to *pixelId* variable using *quadraticFilter*() function. In a simpler implementation, this operation can also be discarded image data can be used directly. But due to the computation of quadratic terms, it is obvious that repeated reads of image data from global memory will decrease the performance. Filter functions normalizes the input pixels to 0-1 interval. After linear and quadratic components are computed, it is normalized back to 0-255 interval and it is written to back to pixel location in global memory area where the output image is defined.

Algorithm 2: IMPLEMENTATION OF METHOD 2 FOR QIF KERNEL

Input: *inputImage*: Image to be filtered, *W*: Weights, *NW*: Number of weights
Output: *outputImage* : Filtered image

```
Define a block shared array for filter weights
__shared__ float sharedW[NW];
Get thread id to determine pixel Id
pixelId ← blockDim.x * blockIdx.x + threadIdx.x
Get 3x3 frame to thread local memory
threadLocInImg ← get3x3frame(inImg, pixelId)
Copy all weigths to shared memory
if threadIdx.x < NW then
    sharedW[threadIdx.x] ← W[threadIdx.x]
Wait for all threads to complete copy operations
__syncthreads()
Apply filtering to selected pixel
y ← quadFilt(sharedW, threadLocInImg, pixelId)
Write results to output image (repeat for RGB)
outputImage(pixelId) ← y
```

Algorithm 2 shows the kernel implementation using Method 2 where block shared memory is utilized to improve the throughput. In addition to input pixels, each threads uses weights from global memory during filtering. On the other hand copying weights to thread local memory is not useful, since each thread reads weights one time and therefore this doesn't change the number of global memory reads. In this case block shared memory were used to reduce the number of global memory reads. Prior to filtering operation, the weights are copied to block shared memory by each thread as shown by the line 11 of Algorithm 2. Therefore the number of global memory accesses is reduced to the number of blocks. Similar to Algorithm 1, this method is also uses thread local memory for storing the neighborhood of pixel to be filtered. Once all weights are copied to shared memory, all threads in the same block read the weights from block shared memory. However, Method 2 requires synchronization of threads in a block to ensure copy operation completed before filtering starts. Above methods store an input pixel and its neighborhood in the local thread memory to reduce the memory reads from global memory. However, each thread reads neighborhood of a pixel to be filtered repeatedly. It is desired that once a pixel is read from global memory, it doesn't required to be read repeatedly. For this purpose an approach that use block shared memory for storing pixels of a block size is used. Method 3 is based on partitioning the input image into sub-blocks and storing every block in the block shared memory. Therefore repetitive accesses to global memory is eliminated. Although the pixels at the edges of blocks are read two times from the global memory, the other pixels are read one time from the global memory. After each of the sub-images are carried to block shared memories, no access to global memory is required to implement filtering.

Algorithm 3: IMPLEMENTATION OF METHOD 3 FOR QIF KERNEL

Input: *inputImage*: Image to be filtered, *W*:Weights,
BlockLength:Block height and width, *BlockSize*: Number of threads in a block,
BlockCols: Number of blocks in a column, *NW*: Number of weights
Output: *outputImage* : Filtered image

```
Define a block shared array for filter weights
__shared__ float blockW[NW];
Allocate a block shared array for
a block of input pixels (repeat for RGB)
__shared__ float blockSharedSubImage[BlockSize]
Copy all weigths to shared memory
if threadIdx.x < NW then
    blockW[threadIdx.x] ← W[threadIdx.x]
Determine the thread's block coordinates
rowBlock ← blockIdx.x / BlockCols
colBlock ← blockIdx.x % BlockCols
Determine the current thread coordinates in block
rowThread ← threadIdx.x / BlockLength
colThread ← threadIdx.x % BlockLength
Determine pixel coordinates
for current thread to apply filter
row ← rowBlock * (BlockLength − 2) + rowThread
col ← colBlock * (BlockLength − 2) + colThread
Copy a block of pixels to shared memory and scale to 0-1
blockSubImage[rowBlock * BlockLength + colBlock] ← inputImage(row, col)/255.0
Wait for all threads to complete copy operations
__syncthreads()
If selected pixel is not on the edge of a block
and the edge of image pixel start filtering
if not a pixel on the edge of a block and the edge of image then
    Apply filtering to selected pixel
    y ← quadFilt(blockW, blockSubImage, pixelId)
    Write results to output image (repeat for RGB)
    outputImage(pixelId) ← y
```

Method 1 requires reading a mask size of pixels and the filter weights from global memory for filtering a pixel. In Method 2, the weights are copied to block shared memory for each block and therefore reading the weights from the global memory is reduced to the number of blocks. Method 3 also reduces the necessity of reading the neighbor of each pixel for filtering by carrying the sub-part of image to shared memory. The number of sub-blocks is determined by the number of threads in a block as shown by Eq16. In this

equation *splitSize* shows the width and height of a block, *borderLines* is the number of additional lines on all sides of a sub-block. In the present study, the number of threads in a block is set to 256 and therefore image is split into sub-images of the size of 16×16. Since, sub-image in a block also contains the border lines, the sub-image size to be filtered in a 16×16 block is 14×14 for a 3×3 filtering mask. Threads in border lines were used for copying the image. Once *subImageSize* variable is determined in Eq. 16, *blockCols* and *blockRows* which are the number of columns and the number of rows respectively can be determined.

$$\left.\begin{array}{l} blockLength = 16 \\ borderLines = 1 \\ subImgSize = blockLength - 2 \times borderLines = 14 \\ threadsPerBlock = blockLength \times blockLength = 256 \\ blockCols = (nCols + subImgSize - 1) / subImgSize \\ blockRows = (nRows + subImgSize - 1) / subImgSize \\ numOfBlocks = blockCols \times blockRows \end{array}\right\} \quad (14)$$

After the parameters defined by Eq. 16 are set, the kernel shown by Algorithm 3 is called to filter image. Before filtering starts, block shared arrays are allocated for weights and sub-image and then weights are copied to block shared memory as in Method 2. After block coordinates and thread coordinates and the corresponding filter coordinates are determined, a sub-image of block size is copied to block shared memory. Each pixel is copied by a thread and all threads are synchronized to ensure that all pixels in a sub-block are copied before filtering starts. Both transferring filter weights and a block of input image to shared block memory reduce the global memory accesses significantly.

4. Experimental Results

Experimental results were obtained by measuring running durations of sequential implementations on CPU and parallel implementation on GPU using the described approaches in the previous section. Sequential execution durations were obtained using a computer that has Intel(R) Core(TM) i7-4710MQ CPU@2.50GHz processor. CUDA based execution durations were also obtained on the same computer which has NVIDIA GeForce GTX 960M video card. The algorithms were written using VC++ Visual Studio 2015™ environment. Color test images used in the performance measurements were read using OpenCV library[30]. Before processing, image data is normalized to 0-1 interval and stored in float data type for both CPU and GPU implementations. The weights of the quadratic filter were determined by optimization approach using training and test images [31]. An example filtering result for QIF using the example weights were given in Figure 1 which also contains original image, noisy image and other filtering results using Gaussian filter, Average filter and Median filter for comparison.

Table 1 shows the experimental measurements for execution durations repeated 8 times for the realized methods using a test image 1024×768 and 3×3 filter mask size. GPU execution times for the first run is relatively slow when compared to the rest of the results. In GPU computing this is usually called as warm-up run and discarded in performance measurements [32]. Additional kernel launches can be used to remove warm-up overheads. Because the size of the image has significant effect on the computation durations, the performance results were obtained for various sizes of images.

Table 1. Example running times (milliseconds) for CPU, GPU naïve and GPU SMM for 1024×768 image and 3×3 filter mask

	Method 1		**Method 2**		**Method 3**	
CPU	kernel + init.	kernel	kernel + init.	kernel	kernel + init.	kernel
86.47	***679.7***	*5.630*	***692.5***	5.264	***665.1***	*1.172*
87.13	9.294	5.680	8.613	5.254	4.185	1.182
85.19	8.825	5.623	8.430	5.262	3.852	1.175
82.98	9.177	5.594	8.729	5.299	4.033	1.184
83.99	9.681	5.592	8.795	5.268	4.028	1.198
83.79	9.177	5.632	8.524	5.265	4.430	1.207
83.04	8.902	5.611	8.927	5.268	4.207	1.192
86.32	9.039	5.633	8.974	5.278	4.452	1.216

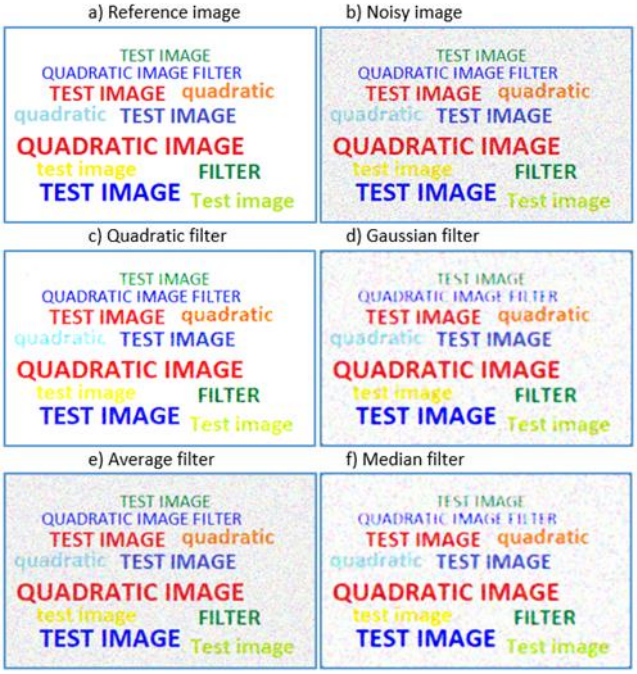

Figure 1. Example filtering results using a synthetic reference image

Table 2. Running durations (milliseconds) for CPU and GPU implementations

Image size	**CPU**	**Method 1**		**Method 2**		**Method 3**	
		kernel + init.	kernel	kernel + init.	kernel	kernel + init.	kernel
640×480	18.89	2.967	1.265	2.631	1.121	1.406	0.296
800×600	50.25	6.898	3.447	6.130	3.212	2.707	0.736
1024×768	88.41	9.849	5.650	9.074	5.259	4.120	1.190
1280×720	103.8	11.06	6.718	10.39	6.391	4.586	1.395
1440×900	137.0	15.04	9.487	13.95	9.030	5.982	1.941
1920×1080	220.6	22.43	15.11	20.85	14.52	8.595	2.949
2560×1440	391.3	37.18	26.10	34.66	24.67	14.45	5.306
3840×2160	938.7	83.33	64.08	81.10	62.32	30.14	11.81

Table 2 shows average computation times for 30 runs using all approaches and various sizes of images ranging

from 640×480 to 3840×2160. According to results, both kernel and kernel+initialization durations are considerably shorter than the sequential execution durations. While Method 1 and Method 2 provide close computation durations, the best acceleration was obtained by using Method 3. Kernel times where the filtering take place are even smaller than the total time which includes initialization times. Method 2 produced always smaller computation times than compared Method 1. This is because, the weights are read from block shared memory in Method 2 while weights are read from global memory in Method 1. Because weights are used one time by each thread for filtering, this doesn't provide significant acceleration like Method 3 where a block of image transferred to block shared memory. All threads uses the image data stored in shared memory. This significantly reduces the number of accesses to global memory and because each thread reads only a pixel from global memory to store to shared memory. On the other hand, the pixels at the edges are shared with blocks and they are read two times by neighbor blocks.

5. Conclusion

Quadratic filters are computationally expensive to implement due to the computation of all possible second degree multiplications. In addition, these terms are multiplied with weights and linear component is also computed for determining an output pixel. According to experimental evaluations, sequential implementation produces long execution times resulting as the image size is increased. In the present study, a GPU based method for the acceleration of the quadratic filter was introduced. For this purpose three alternative implementations from simple to complex were discussed. The first method simply reads a frame of pixels from global memory to thread local memory and reads weights from global memory and apply quadratic filtering to produce filtered pixel. Method 2 reads filter weights global memory and keep them in shared memory for the threads within the block. Therefore threads in a block reads weights from shared memory instead of global memory. Because each of the weights are read one time, this doesn't provide much improvement over Method 1. Method 3 requires the image be separated into a number of blocks determined by a thread block size. Within each block, a number pixels that is equal to the size of a block is filtered by the threads of that block. The image to be filtered is transferred to shared memory of each block to obtain better performance results. Although Method 3 increases the complexity of code it provides better management of memory for reducing access times, and therefore produces better results than the simpler implementations. The results show that, Method 3 considerably reduces the running duration of QIF when compared to CPU and other implementations with GPU. In the future studies, performance can further be improved using alternative algorithmic and programming approaches.

References

[1] J. C. Russ, *The image processing handbook*. CRC press, 2016.

[2] I. Pitas and A. N. Venetsanopoulos, *Nonlinear digital filters: principles and applications*, vol. 84. Springer Science & Business Media, 2013.

[3] G. F. Ramponi, G. L. Sicuranza, and W. Ukovich, "A computational method for the design of 2-D nonlinear Volterra filters," *Circuits Syst. IEEE Trans.*, vol. 35, no. 9, pp. 1095–1102, 1988.

[4] L. Thomas, G. Krishnan, R. A. Mol, and A. Roy, "Removal of Impulsive Noise from MRI Images using Quadratic Filter," *Int. J. Eng. Res. Technol.*, vol. 3, no. 4, pp. 2220–2223, 2014.

[5] M. Meenavathi and K. Rajesh, "Volterra Filtering techniques for removal of Gaussian and mixed Gaussian-Impulse noise," *Int. J. Electr. Comput. Eng.*, vol. 1, no. 2, pp. 184–190, 2007.

[6] J. Zhang and Y. Pang, "Pipelined robust M-estimate adaptive second-order Volterra filter against impulsive noise," *Digit. Signal Process.*, vol. 26, pp. 71–80, Mar. 2014.

[7] V. S. Hari, V. P. Jagathy Raj, and R. Gopikakumari, "Quadratic filter for the enhancement of edges in retinal images for the efficient detection and localization of diabetic retinopathy," *Pattern Anal. Appl.*, vol. 20, no. 1, pp. 145–165, Feb. 2017.

[8] V. S. Hari, V. P. Jagathy Raj, and R. Gopikakumari, "Unsharp masking using quadratic filter for the enhancement of fingerprints in noisy background," *Pattern Recognit.*, vol. 46, no. 12, pp. 3198–3207, Dec. 2013.

[9] Y. Zhou, K. Panetta, and S. Agaian, "Mammogram enhancement using alpha weighted quadratic filter," in Proceedings of the 31st Annual International Conference of the IEEE Engineering in Medicine and Biology Society: Engineering the Future of Biomedicine, EMBC 2009, 2009, pp. 3681–3684.

[10] A. Pandey, A. Yadav, and V. Bhateja, "Design of new volterra filter for mammogram enhancement," in *Advances in Intelligent Systems and Computing*, 2013, vol. 199 AISC, pp. 143–151.

[11] M. Kanamadi, V. Waghamode, and S. Bandekar, "Alpha Weighted Quadratic Filter Based Enhancement for Mammogram," in Proceedings of International conference on "Emerging Research in Computing, Information, Communication and Applications" (ERCICA), 2013, pp. 68–74.

[12] V. Bhateja, M. Misra, S. U.-C. methods and programs in, and undefined 2016, "Non-linear polynomial filters for edge enhancement of mammogram lesions," *Comput. Methods Programs Biomed.*, no. 129, pp. 125-134., 2016.

[13] V. S. Hari, R. V. P. Jagathy, and R. Gopikakumari, "Enhancement of calcifications in mammograms using Volterra series based quadratic filter," in *Proceedings - 2012 International Conference on Data Science and Engineering, ICDSE 2012*, 2012, pp. 85–89.

[14] G. Jothilakshmi and E. Gopinathan, "Mammogram Enhancement Using Quadratic Adaptive Volterra Filter A Comparative Analysis In Spatial And Frequency Domain," *ARPN J. Eng. Appl. Sci.*, vol. 10, no. 13, pp. 5512–5517, 2006.

[15] V. Bhateja, M. Misra, S. Urooj, and A. Lay-Ekuakille, "A robust polynomial filtering framework for mammographic image enhancement from biomedical sensors," *IEEE Sens. J.*, vol. 13, no. 11, pp. 4147–4156, 2013.

[16] A. Chakrabarty, H. Jain, and A. Chatterjee, "Volterra kernel based face recognition using artificial bee colonyoptimization," *Eng. Appl. Artif. Intell.*, vol. 26, no. 3, pp. 1107–1114, 2013.

[17] G. Feng, H. Li, J. Dong, and J. Zhang, "Face recognition based on Volterra kernels direct discriminant analysis and effective feature classification," *Inf. Sci. (Ny).*, vol. 441, pp. 187–197, 2018.

[18] G. Feng, H. Li, J. Dong, and J. Zhang, "Direct discriminant analysis using volterra kernels for face recognition," in *Communications in Computer and Information Science*, 2016, vol. 662, pp. 404–412.

[19] G. Sicuranza and G. Ramponi, "Adaptive nonlinear digital filters using distributed arithmetic," *IEEE Trans. Acoust.*, 1986.

[20] G. Ramponi, "Edge extraction by a class of second-order nonlinear filters," *Electron. Lett.*, vol. 9, no. 22, pp. 482–484, 1986.

[21] S. Mitra, "Image processing using quadratic volterra filters," in

Computers and Devices for Communication (CODEC), 2012 5th International Conference on, 2012, pp. 1–2.

[22] T. Kalaiselvi, P. Sriramakrishnan, and K. Somasundaram, "Survey of using GPU CUDA programming model in medical image analysis," *Informatics Med. Unlocked*, vol. 9, pp. 133–144, Jan. 2017.

[23] M. Soua, R. Kachouri, and M. Akil, "GPU parallel implementation of the new hybrid binarization based on Kmeans method (HBK)," *J. Real-Time Image Process.*, vol. 14, no. 2, pp. 363–377, Feb. 2018.

[24] A. HajiRassouliha, A. J. Taberner, M. P. Nash, and P. M. F. Nielsen, "Suitability of recent hardware accelerators (DSPs, FPGAs, and GPUs) for computer vision and image processing algorithms," *Signal Process. Image Commun.*, vol. 68, pp. 101–119, Oct. 2018.

[25] Y. Zhou, F. He, and Y. Qiu, "Accelerating image convolution filtering algorithms on integrated CPU–GPU architectures," *J. Electron. Imaging*, vol. 27, no. 03, p. 1, May 2018.

[26] O. Green, "Efficient scalable median filtering using histogram-based operations," *IEEE Trans. Image Process.*, vol. 27, no. 5, pp. 2217–2228, 2017.

[27] F. Bozkurt, M. Yaganoglu, and F. B. Günay, "Effective Gaussian Blurring Process on Graphics Processing Unit with CUDA," *Int. J. Mach. Learn. Comput.*, vol. 5, no. 1, p. 57, 2015.

[28] P. S. Battiato, "High Performance Median Filtering Algorithm Based on NVIDIA GPU Computing," in *International Symposium for Young Scientists in Technology, Engineering and Mathematics*, 2016, pp. 1–10.

[29] W. Ling, Nonlinear digital filters: analysis and applications. Academic Press, 2010.

[30] G. Bradski and A. Kaehler, *Learning OpenCV: Computer vision with the OpenCV library*. " O'Reilly Media, Inc.," 2008.

[31] S. Uzun and D. Akgün, "An Accelerated Method for Determining the Weights of Quadratic Image Filters," *IEEE Access*, vol. 6, pp. 33718–33726, 2018.

[32] J. Cheng, M. Grossman, and T. KcKercher, *Professional CUDA C programming*. Wrox, 2014.