Frequency Domain Image Filtering Using CUDA

MUHAMMAD AWAIS RAJPUT*, UMAIR ALI KHAN*, AND NISAR AHMED MEMON**

RECEIVED ON 08.04.2014 ACCEPTED ON 16.07.2014

ABSTRACT

In this paper, we investigate the implementation of image filtering in frequency domain using NVIDIA's CUDA (Compute Unified Device Architecture). In contrast to signal and image filtering in spatial domain which uses convolution operations and hence is more compute-intensive for filters having larger spatial extent, the frequency domain filtering uses FFT (Fast Fourier Transform) which is much faster and significantly reduces the computational complexity of the filtering. We implement the frequency domain filtering on CPU and GPU respectively and analyze the speed-up obtained from the CUDA's parallel processing paradigm. In order to demonstrate the efficiency of frequency domain filtering on CUDA, we implement three frequency domain filters, i.e. Butterworth, low-pass and Gaussian for processing different sizes of images on CPU and GPU respectively and perform the GPU vs. CPU benchmarks. The results presented in this paper show that the frequency domain filtering with CUDA achieves significant speed-up over the CPU processing in frequency domain with the same level of (output) image quality on both the processing architectures.

Key Words: **Frequency Domain Filtering, Compute Unified Device Architecture, Parallel Processing.**

1. INTRODUCTION

With the advancements in the field of computer hardware, more throughput in terms of higher computations per second is always desirable. While the requirement of more processing speed still persists, the processing power of modern general-purpose processors does not keep up pace with the advancements in the state-of-the-art complex algorithms pertaining to different fields of science. Although, the modern processors comprise multiple cores for parallel processing, they are still not sufficient for high-end processing requirements. Parallel processing is performed by a number of techniques including multiplicity of functional units in processors, distributed computing, multi-processing, vector processing, and grid computing (to name a few). However, these techniques are complex, expensive and/or require large-scale implementation as well as significant modifications in the processing hardware. Therefore, a general-purpose, cheaper and more scalable parallel computing paradigm is required.

NVIDIA introduced CUDA in 2007 for general-purpose (parallel) processing on graphics processing units. Although several graphics libraries such as OpenGL, DirectX and OpenCL are also used for general-purpose processing on GPUs, they are not as rich in programming features and flexibility as CUDA (e.g. profiling, debugging, efficient and convenient high-level APIs, etc.). CUDA makes use of the massive data parallelism capabilities of a GPU's

* Assistant Professor, and ** Professor,
Department of Computer Systems Engineering, Quaid-e-Awam University of Engineering, Science & Technology, Nawabshah

multiple streaming processors for performing general-purpose (parallel) processing. Additionally, CUDA provides increased programmer productivity as well as built-in support for OpenCL and OpenGL. CUDA finds its applications in various fields like medical imaging, weather, space, computational finance, computational flood dynamics, simulation and modeling, to name a few.

Image processing is one of the fields which have remained of particular interest for the researchers since a long time. Since most of the image processing operations operate on individual image pixels, they are compute-intensive in terms of computations per second. However, image processing operations are inherently parallel and most suitable for SIMD (Single Instruction Multiple Data) [1] Parallel processing architecture. A CPU (processor), due to limited number of cores and a specific architecture, offers limited capabilities of processing large sets of data in a parallel fashion. On the other hand, CUDA transfers the data parallel part of an algorithm to the GPU where hundreds of GPU cores simultaneously operate on the data using the GPU's SIMD architecture. Many image processing algorithms have been implemented on CUDA [2][3][4][5] and are under extensive research for further improvements.

A number of image and video processing operations have been investigated and analyzed using CUDA which include motion tracking [6], face tracking [7], feature extraction and tracking [8], object detection and classification [9-10], and medical image segmentation [11-12], to name a few. The relevant literature demonstrates CUDA's efficacy and huge potential for ultra-fast image processing by performing massively parallel operations with an economically effective hardware. Apart from this, image and signal filtering using CUDA has also been extensively studied in the relevant literature [13-16]. Image filtering is required for performing a number of tweaks on images, e.g. blurring, sharpening, de-noising, and tone mapping of high dynamic range images, etc. Since image filtering is compute-intensive and presents a dedicated challenge in real-time operations, the acceleration using CUDA's parallel processing paradigm is highly desirable. However, image filtering using CUDA is mostly implemented in spatial domain using convolution operations. Filtering in spatial domain is more computationally complex and slower than its counterpart; the filtering in frequency domain. In frequency domain, convolution takes the form of multiplication which is much simpler than convolution. Especially for larger filters, the cost of frequency domain filtering using FFT becomes relatively small compared to spatial domain filtering. The complexity of the filtering in spatial domain is $O(N^2)$. Whereas, in the frequency domain filtering, FFT is applied in the form of multiplication which has complexity of O(N). Another advantage of FFT is that its execution time does not increase significantly when multiple filters are used. Whereas, multiple filters have drastic effects on the performance of convolution operation [17].

In this paper, we investigate the benefit of frequency domain image filtering using CUDA for low-pass image filters namely Butterworth, ideal and Gaussian filters. For this purpose, we first convert the image from spatial domain to frequency domain by applying discrete Fourier transform. The low-pass frequency domain filters are then applied to the image. The following step transforms the image into spatial domain by applying IFFT (Inverse Fast Fourier Transform) to give the output image. These steps are performed on the CPU-based algorithm and the GPU-based algorithm respectively and the bench- marks are performed with respect to execution time, image quality and speed-up. The benchmark results obtained after the comparison of our implemented CUDA code with that of the CPU code shows a significant speed-up of up to 5x.

The rest of the paper is organized as follows. Section 2 provides an overview of the related work. Section 3 gives an introduction of CUDA framework. Section 4 gives a detailed overview of our proposed method of frequency domain image filtering using CUDA. In section 5, we discuss our experimental setup and results. Section 6 concludes the paper.

2. RELATED WORK

The implementation of FFT and IFFT for frequency domain image filtering is first performed in [18]. This work shows how a commodity graphics card can be used to perform image filtering. However, this work does not leverage CUDA, but rather makes use of OpenGL and Cg runtime graphics libraries which are not as effective as CUDA.

In [17], Fialka et al. compare the GPU implementation of frequency and spatial domain filtering and identify the conditions under which the frequency domain filtering gives better performance than the corresponding spatial domain filtering. However, this analysis is based on shader programming which is a part of DirectX and is not as efficient and flexible as CUDA [19].

Despite of immense potential of image filtering using CUDA, little attention is paid on frequency domain image filtering using CUDA. In [20], Eklund, et. al. use frequency domain image filtering with CUDA in one of the steps of medical image registration. However, the GPU vs. CPU speed-up for the filtering operation is reported to be just 1.7x for large data. It is also not clear whether this speed-up includes the overhead time (time to transfer the data from CPU to GPU and back).

Mohammad Nazmul Haque et. al. [21] implement the FFT and IFFT image reconstruction algorithm using CUDA's cuFFT library. They emphasize the use of cuFFT library function to compute the FFT and IFFT of the image for the various frequency domain operations including image processing in frequency domain. Their work focuses on porting the whole workflow of image processing to the GPU instead of only FFT and IFFT part.

In a more recent work, Xi Chen et. al. [22] implement image filtering using CUDA and achieve significant speed-up. Their implementation is focused on frequency domain filters using CUDA and they show a huge speed-up achieved by porting parallel tasks on the GPU. However their reported execution time for the CUDA algorithm excludes the overhead time which has a significant impact on a GPU's computing performance.

3. COMPUTE UNIFIED DEVICE ARCHITECTURE

NVIDIA's CUDA programming model allows programmers to port the compute-intensive tasks to GPUs where they can be executed in parallel way using SIMD architecture. A typical CUDA program basically comprises a heterogeneous computing paradigm where the serial part of the program is executed by the CPU and the compute-intensive part is seamlessly ported to the GPU by the CUDA compute engine. CUDA framework has following advantages:

- Massively parallel hardware designed to utilize a graphics card to run generic (not essentially graphic) code.
- Ease of programming with a C-like language (CUDA C), eliminating the need of using pixel and vertex shaders to emulate general-purpose computers (OpenGL).
- SIMD based parallel hardware to execute a significantly larger number of operations per second than the CPU, at a fairly similar financial cost, yielding substantial performance improvements as compared to CPU.
- CUDA provides an easy way for programmers to write massively parallel programs by using an API (Application Programming Interface) which is a collection of functions specially built for GPUs.

In CUDA's terminology, the CPU is called host which works in coordination with the GPU (called device). In SIMD computing architecture used by the GPU, each function executed by the GPU is called a kernel which contains multiple threads of an instruction/program operating on different data. A CUDA program typically combines parts of the code which have the margin of data parallelism and those which have no or less possibility of data parallelism. The parts having more data parallelism are executed on the device and the parts having no or less data parallelism are executed on the host [23]. Therefore, an efficient implementation of a program on CUDA requires precise identification of the program's parts which can be ported to the GPU and executed in parallel way on its multiple cores using SIMD architecture.

A kernel comprises several threads executing the same program. These threads are grouped into blocks having unique identifiers. The threads in the same block can cooperate and communicate with each other [24]. Several blocks are combined to form a grid. The dimensions of the block and grid depend on the problem and data size. Fig. 1 shows a grid of thread blocks and the threads inside a block.

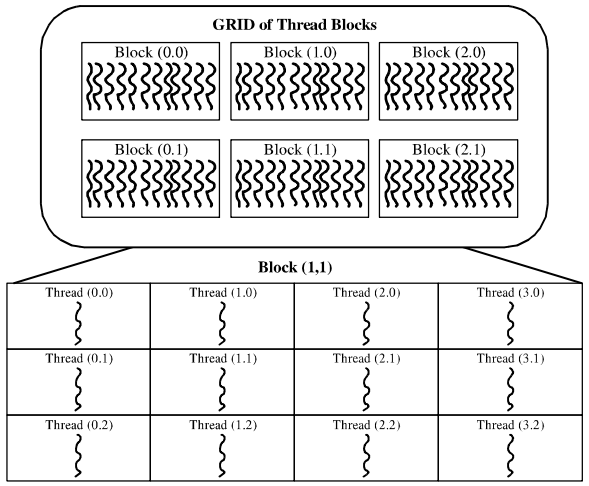


FIG. 1. GRID, BLOCKS AND THREADS IN CUDA FRAMEWORK

A typical CUDA program has the following workflow:

(1) The CPU allocates memory at the GPU to store data to be processed on the GPU and communicated back to the CPU after processing.
(2) The CPU transfers data from main memory to the GPU's (allocated) memory.
(3) The GPU processes the threads of a kernel on its multiple cores and stores the results in its memory.
(4) The results from the GPU's allocated memory are copied back to the main memory.
(5) The CPU frees the GPU's allocated memory.

One of the major issues actively researched in CUDA framework [25-26] is to reduce the memory transfer overhead to and from the GPU. The latest NVIDIA's GPUs which are targeted for higher-performance computing rather than graphics (e.g., NVIDIA Tesla C2050) are designed with high memory transfer bandwidth to cope with this problem.

4. FREQUENCY DOMAIN IMAGE FILTERING USING CUDA

Frequency domain image filtering focuses on the frequency spectrum of the image. Typical steps in this process are:

(i) Transforming the image from spatial domain to frequency domain using DFFT (Discrete Fast Fourier Transform).
(ii) Creating the filter in frequency domain using the transfer function specific to the filter.
(iii) Multiplying the frequency spectrum of the image with the filter function.
(iv) Transforming the resultant image spectrum to spatial domain using IFFT.

Since the above mentioned operations, i.e. DFFT and IFFT are massively parallel, they are potential candidates to be implemented on CUDA. Generally, image filtering is done for various purposes such as enhancing the visual appearance of an image, removing the noise from an image, edge detection, to name a few. Image filtering can be performed in either spatial domain or frequency domain. Spatial domain methods use convolution operations that are applied repeatedly on image pixels. These operations are complex and computationally expensive. In contrast, the frequency domain methods involve the Fourier transform of the image which is then multiplied with a filter function to obtain the resultant image spectrum. This spectrum is then transformed back into spatial domain. The base of frequency domain filtering is the convolution theorem which is defined as [27]:

$$f(x,y) * h(x,y) \Leftrightarrow F(u,v)H(u,v) \quad (1)$$

and conversely,

$$f(x,y)h(x,y) \Leftrightarrow F(u,v) * H(u,v) \quad (2)$$

Where the symbol "*" represents the convolution of the two functions. It implies that image/signal filtering by convolving an image/signal f(x,y) with a convolution mask h(x,y) in spatial domain is equivalent to the multiplication of image/signal in frequency domain F(u,v) by a filter transfer function H(u,v) [27-28]. Basically the idea in frequency domain filtering is to select a filter transfer function that modifies the frequency spectrum F(u,v) in a specified manner. For example, when the transfer function of a Butterworth low-pass filter H(u,v) is multiplied by a centered frequency domain image F(u,v), it attenuates high frequencies in F(u,v) while leaving the low frequencies relatively unchanged. The overall result is image blurring (smoothing) [29]. The steps involved in the frequency domain image filtering and its spatial domain counterpart are shown in Fig. 2.

4.1 Frequency Domain Image Filters

We selected three frequency domain image filters, i.e. Butterworth low-pass filter, ideal low-pass filter, and Gaussian low-pass filter for implementing their operation on CUDA. The basic operation of all these filters is to blur or smoothen an image with the pre-selected parameters. The transfer function H of a Butterworth low-pass filter of order n with cut-off frequency at distance D_0 from the origin is defined as:

$$H_{(u,v)} = \frac{1}{1+\left[\frac{D_{(u,v)}}{D_0}\right]^{2n}} \quad (3)$$

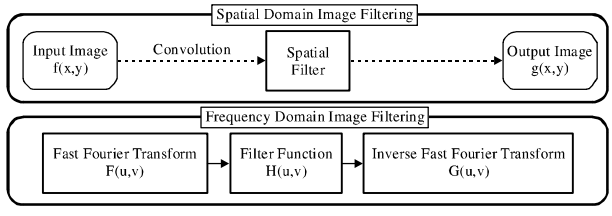


FIG. 2. FREQUENCY DOMAIN IMAGE FILTERING AND ITS SPATIAL DOMAIN COUNTERPART

where u and v are the coordinates of the transformed image (frequency domain). The advantage of using a Butterworth filter in practice is its low ringing effect. The transfer function of ideal low-pass function is given by Equation (4).

$$H_{(u,v)} = \begin{cases} 1 & if\ D_{(u,v)} < D_0 \\ 0 & otherwise \end{cases} \quad (4)$$

The ideal low-pass filter is the simplest of all the three filters mentioned in this paper. Its operation is to suppress all the frequencies which are higher than the cut-off frequency D_0 and to leave all other frequencies un-changed.

The transfer function of the third frequency domain filter, the Gaussian low-pass filter, which we consider for CUDA implementation is given by Equation (5).

$$H_{(u,v)} = e - \frac{\left(u^2 + v^2\right)}{2D_0^2} \quad (5)$$

4.2 Identifying the Compute-Intensive Parts

In order to accelerate the image filtering process with CUDA, it is particularly important to identify the compute-intensive parts of the filtering process which could be ported to GPU memory and have margin of parallel processing with SIMD architecture. We use Matlab's visual profiler to analyze the image filtering (CPU-based) code for the compute- intensive parts. Our investigation for the Butterworth low-pass filter code using Matlab profiler shows that the part of FFT is most time-consuming compared to rest of the program code. Fig. 3 shows the highlighted part of the code snippet in red identified as most compute-intensive by Matlab profiler. The time taken by each part of the code is shown on the left in red values. The blue values on the left show the number of times a code snippet has been executed. Based on this analysis, we select the FFT part to be ported to GPU by creating different CUDA kernels whose individual threads run parallel to each other on the GPU.

4.3 CUDA Work Flow for Frequency Domain Image Processing

We first load the image into host memory and then transfer it to the device memory using CUDA's library function cudaMemcpy. Our CUDA programmed kernels then operate on the image. One of these kernels computes the 2D fast Fourier transform of the image. In the following step, the filter matrix is multiplied with the frequency spectrum of the image by another kernel and the resultant image is temporarily held in the device memory. Subsequently, the resultant image spectrum is applied to the third kernel to transform it back into spatial domain (inverse Fourier transform). The resultant image pixels are finally transferred to the host using cudaMemCpy() function.

Fig. 4 shows the work flow of our frequency domain image filtering on CUDA. The host (CPU) only executes those parts of the code which are either not

```
 0.18       1   10 f=imread(filename);
 0.05       1   11 f=im2double(f);
            1   12 [M,N]=size(f);
            1   13 tic
                14 %FFT of the image after padding values of the same size as image
 1.89       1   15 F=single(fft2(f,2*M,2*N));
                16 %FFT shift
 0.10       1   17 Fc=single(fftshift(F));
                18 %deleting large matrix to release memory
 0.05       1   19 delete F
                20 %initializing the filter of the same size as the padded image
 0.06       1   21 H=single(zeros(2*M,2*N));
                22 %calculating filter values
            1   23 for u=1:2*M;
         4096   24 for v=1:2*N;
 8.84 16777216   25 radius = ((u-(M+1))^2 + (v=(N+1))^2)^.5;
10.91 16777216   26 H(u,v)= 1./(1.0 +  (radius./ 40).^(2*2));
 8.03 16777216   27 end;
         4096   28 end;
            1   29 filter=H;
                30 %multiply the filter with image spectrum
 0.34       1   31 G=H.*Fc;
                32 %deleting large matrix to release memory
            1   33 delete Fc
                34 %Inverse FFT shift
 0.10       1   35 Gsh=single(ifftshift(G));
                36 %deleting large matrix to release memory
            1   37 delete G H
                38 %Inverse FFT to transform the image in spatial domain
 1.79       1   39 Ginv=ifft2(Gsh,2*M,2*N);
                40 %deleting large martix to release memory
            1   41 delete Gsh
                42 %converting to real and unpadding
            1   43 Greal=single(real(Ginv(1:M,1:N)));
            1   44 resultImage=Greal;
                45 % t=toc*1000
                46 %   disp("Image  "& M & "x" & N);
            1   47   t_total(count)= toc *1000
            1   48   sum=sum+t_total(count);
            1   49     end
 0.05       1   50     end
```

FIG. 3. MATLAB PROFILER RESULT FOR BUTTERWORTH FILTER

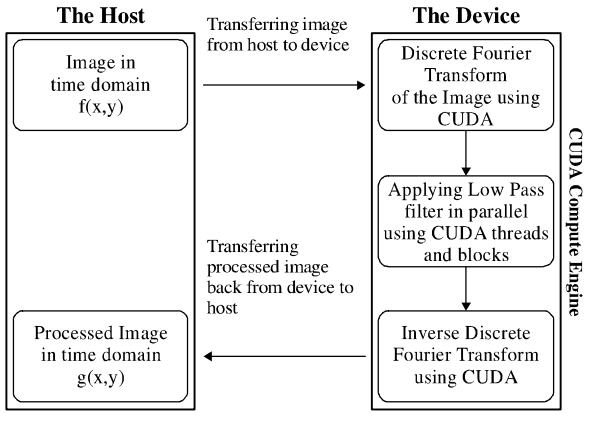


FIG. 4. WORK FLOW FOR FREQUENCY DOMAIN IMAGE FILTERING WITH CUDA

parallelizeable or are not related to processing the image (e.g. loading image pixels into the host memory, writing the filtered image to disk, etc.). The rest of the code such as transforming the image into frequency domain using discrete FFT, applying transfer function on the frequency spectrum of the image, and transforming the image back into spatial domain using inverse FFT are all performed by the GPU, thus leveraging the massive parallel processing power of the GPU cores for the compute-intensive parts of the code.

5. EXPERIMENTAL RESULTS

The dimension of the block size used in our experiments is selected to be $(B_h,B_w)=(16,16)$, where B_h and B_w represent the height and width of the block respectively (256 threads per block). It is worth mentioning that for discrete Fourier transform, the image needs to be padded with the same number of zeros as the original size of the image to obtain a finer sampling of the Fourier transform [30]. These zeros are unpadded after the inverse discrete Fourier transform of the image. Therefore, we have two possible sizes of the block grid, i.e. block grid for unpadded (original sized) image, and the block grid for padded image.

The dimensions of the block gird for unpadded and padded image, represented by D_u and D_p respectively, are calculated by the following rule:

$$D_u = (\frac{I_h}{B_h}, \frac{I_w}{B_w}) \quad (6)$$

$$D_p = (\frac{P_h}{B_h}, \frac{P_w}{B_w}) \quad (7)$$

Where I_h and I_w are the dimensions of the unpadded image, P_h and P_w are the dimensions of the padded image with $P_h = 2\times I_h$, $P_w = 2\times I_w$, and B_h and B_w represent the dimensions of each block.

The execution times of the frequency domain filters were measured for CPU and GPU implementation separately for different image sizes. Fig. 5 shows the graph of the average execution times of the CPU and the GPU implementations of Butterworth filters with increasing image sizes. It is clear from Fig. 5 that there is a significant speed-up achieved by our GPU implementation of image filtering. However, it is also interesting to measure the speed-up with and without the memory transfer overhead. This comparison is shown in Fig. 6. The memory transfer overhead refers to the time consumed in allocating memory on the GPU and transferring the data from CPU to GPU and then back. It can be noted that as the size of the input data increases, the overhead also increases proportionally and the speed-up starts dropping for the larger image sizes. The execution times of the CUDA kernels for image filtering and the respective memory transfer times are shown in Fig. 7.

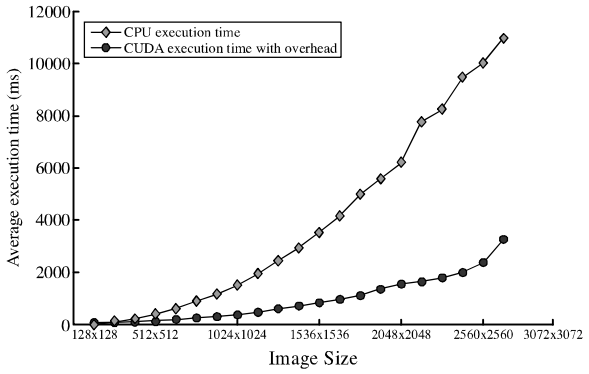


FIG. 5. AVERAGE EXECUTION TIME OF CPU- AND GPU-BASED BUTTERWORTH FILTER

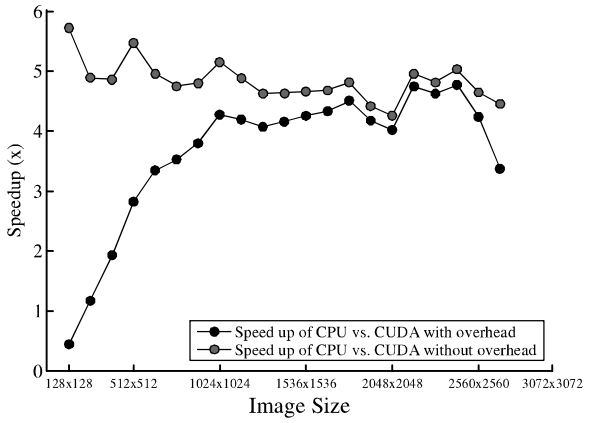


FIG. 6. SPEED-UP WITH AND WITHOUT OVERHEAD FOR BUTTERWORTH FILTER

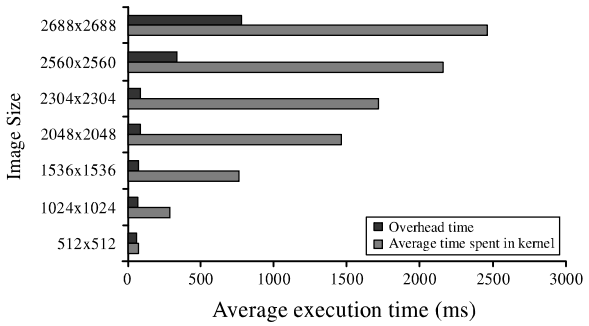
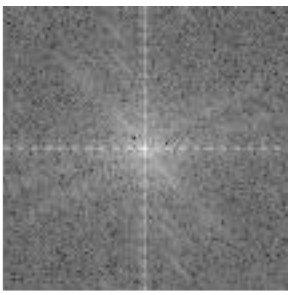


FIG. 7. KERNEL EXECUTION TIMES AND OVERHEAD TIMES FOR VARIOUS IMAGE SIZES

Fig. 8 (a-c) shows the image filtering results with Butterworth, Gaussian and Ideal low-pass filters. For the sake of simplicity, we present the CPU filtered image of Butterworth filter only. The CUDA filtered images for the three filters are shown in Fig. 8(d-f). Although the image filtering results for GPU implementation and its CPU counterpart appear to be same, we also compare the filtered images of both the implementations by calculating the MSE (Mean Square Error) for the CPU filtered and the GPU filtered images of dimensions $M \times N$ by the following rule:

$$MSE = \frac{1}{MN} \sum_{i=0}^{M-1} \sum_{j=0}^{N-1} [C(i,j) - G(i,j)]^2 \quad (8)$$

(a) ORIGINAL IMAGE



(b) FREQUENCY SPECTRUM OF THE IMAGE

(c) CPU BUTTERWORTH

(d) CUDA BUTTERWORTH

(e) CUDA GAUSSIAN

(f) CUDA IDEAL LOW-PASS

FIG. 8. FREQUENCY DOMAIN IMAGE FILTERING FOR VARIOUS FILTERS

where $C(i, j)$ and $G(i, j)$ represent the CPU filtered image and the GPU filtered image respectively. The average MSE for all the filter implementations is found to be 0.00040 which shows a very satisfactory level of similarity between the CPU filtered and the GPU filtered images.

We also perform statistical analysis on the speed-up trend with respect to image sizes. By obtaining the speed-up statistics from different image sizes and using appropriate curve fitting techniques, we find an interesting relationship between the speed-up and the data size which is shown in Fig. 9. It is evident that the speed-up follows an exponential trend which can be defined by the following relation:

where a = 6.221, b = −0.0001475, c = −7.11 and d = −0.001641.

The detailed specifications of the hardware and the software used in our experiments are given in Tables 1-2.

6. CONCLUSION

In this paper, we have investigated the benefits of frequency domain image filtering using CUDA. We implemented three low-pass frequency domain filters, namely Butterworth, Ideal, and Gaussian filters. The results presented in this paper show that our CUDA/GPU implementation of the frequency domain image filtering achieves a speed-up of up to 5x as compared to the CPU implementation of the same filters. Apart from this, we also presented an important relationship between the speed-up and the data size.

In future, we aim to extend this work by applying the CUDA based frequency domain image filtering on real-time video and image processing systems. We

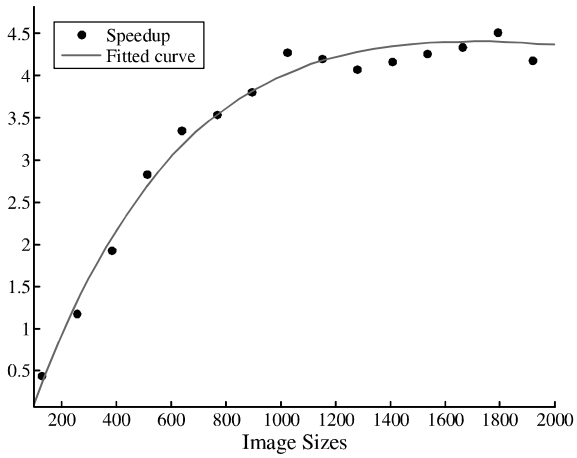


FIG. 9. GPU VS. CPU SPEED-UP TREND

further aim to improve the speed-up by utilizing other features of CUDA such as shared memory which operates much faster than the global memory. We also aim to transform the spatial domain image filters in frequency domain and then implement them on CUDA.

ACKNOWLEDGEMENT

This research is carried out in M.E. Thesis, Department of Computer Systems Engineering, Quaid-e-Awam University of Engineering, Science & Technology, Nawabshah, Pakistan.

REFERENCES

[1] NVIDIA Corporation, "NVIDIA CUDA C Programming Guide", 2011.

[2] Park, I.K., Singhal, N., Lee, M.H., Cho, S., and Kim, C., "Design and Performance Evaluation of Image Processing Algorithms On GPUs", IEEE Transactions on Parallel and Distributed Systems, Volume 22, No. 1, pp. 91-104, 2011.

[3] Palhano, F., Xavier, G., Andrade, P., and Hellier, P., "Real Time Ultrasound Image Denoising", Journal of Real-Time Image Processing, Volume 6, No. 1, pp. 15-22, 2011.

TABLE 1. HOST SPECIFICATIONS

Feature	Specification
CPU Model	Intel core i7
CPU clock speed	3.4 GHz
System Main Memory 8 GB	8 GB
Operating System	Windows 7 Home Premium 64-bit

TABLE 2. DEVICE SPECIFICATIONS

Feature	Specification
GPU processor	NVIDIA GeForce GT 610
CUDA cores	48
Core clock	810 MHz Memory data rate 1000 MHz Memory interface 64-bit
On-chip Memory	1 GB Memory Bandwidth 14.4 GB/sec
CUDA toolkit	version 6.0 [31]

[4] Wang, K., Huang, C., Y.J., Kao, Chou, C.Y., Oraevsky, A., and Anastasio, M.A., "Accelerating Image Reconstruction in Three-Dimensional Optoacoustic Tomography On Graphics Processing Units", Medical Physics, Volume 40, No. 2, pp. 23301, 2013.

[5] Brown, J., and Capson, D., "A Framework for 3D Model-Based Visual Tracking Using A GPU-Accelerated Particle Filter", IEEE Transactions on Visualization and Computer Graphics, Volume 18, No. 1, pp. 68-80, 2012.

[6] Huang, J., Ponce, S.P., Park, S.I., Cao, Y., and Quek, F., "GPU-Accelerated Computation for Robust Motion Tracking Using the CUDA Framework", Proceedings of 5th International Conference on Visual Information Engineering, pp. 437-442. Xi'an, China, 2008.

[7] Sharma, B., Thota, R., Vydyanathan, N., and Kale, A., "Towards a Robust, Real-Time Face Processing System Using CUDA-Enabled GPUs", Proceedings of IEEE International Conference on High Performance Computing, pp. 368-377, Kochi, India, 2009.

[8] Sinha, S.N., Frahm, J.M., Pollefeys, M., and Genc, Y., "Feature Tracking And Matching in Video Using Programmable Graphics Hardware", Machine Vision and Applications, Volume 22, No. 1, pp. 207-217, 2011.

[9] Zhang, L., and Nevatia, R., "Efficient Scan-Window Based Object Detection Using GPGPU", Proceedings of IEEE Computer Society Conference on Computer Vision and Pattern Recognition, pp. 1-7, Anchorage, USA, 2008.

[10] Mussi, L., Cagnoni, S., and Daolio, F., "GPU-Based Road Sign Detection Using Particle Swarm Optimization", Proceedings of 9th International Conference on Intelligent Systems Design and Applications, pp. 152-157, Pisa, Italy, 2009.

[11] Pan, L., Gu, L., and Xu, J., "Implementation of Medical Image Segmentation In CUDA", Proceedings of IEEE International Conference on Information Technology and Applications in Biomedicine, pp. 82-85, Shenzhen, China, 2008.

[12] Kauffmann, C., and Pich´e, N., "Seeded ND Medical Image Segmentation by Cellular Automaton on GPU", International Journal of Computer Assisted Radiology and Surgery, Volume 5, No. 3, pp. 251-262, 2010.

[13] Scherl, S., Keck, B., Kowarschik M., and Hornegger, J., "Fast GPU-based CT Reconstruction Using the Common Unified Device Architecture", Proceedings of IEEE Nuclear Science Symposium Conference Record, pp. 4464-4466, Honolulu, HI, 2007.

[14] Ogawa, K., Ito, Y., and Nakano, K., "Efficient Canny Edge Detection Using a GPU", Proceedings of IEEE 1st International Conference on Networking and Computing, pp. 279-280, Higashi-Hiroshima, Japan, 2010.

[15] Van, W.J., Jalba, A.C., and Roerdink, J.B., "Accelerating Wavelet Lifting on Graphics Hardware Using CUDA", IEEE Transactions on Parallel and Distributed Systems, Volume 22, No.1, pp. 132-146, 2011.

[16] Wilson, J.A., and Williams, J.C., "Massively Parallel Signal Processing Using the Graphics Processing Unit for Real-Time Brain-Computer Interface Feature Extraction", Frontiers in Neuro Engineering, Volume 2, No. 11, pp. 1-13, 2009.

[17] Fialka, O., and Cadik, M., "FFT and Convolution Performance in Image Filtering on GPU", Proceedings of 10th International Conference on Information Visualization, pp. 609-614, London, England, 2006.

[18] Moreland, K., and Angel, E., "The FFT on a GPU", Proceedings of the ACM SIG-GRAPH/EUROGRAPHICS Conference on Graphics Hardware, pp. 112-119, Aire-la-Ville, Switzerland, 2003.

[19] Guzhva, A., Dolenko, S., and Persiantsev, I., "Multifold Acceleration of Neural Network Computations Using GPU", Proceedings of International Conference on Artificial Neural Networks, pp. 373-380, Cyprus, 2009.

[20] Eklund, A., Andersson, M., and Knutsson, H., "Phase Based Volume Registration Using CUDA", Proceedings of IEEE International Conference on Acoustics Speech and Signal Processing, pp. 658-661, Dallas, TX, 2010.

[21] Haque, M.N., and Uddin M.S., "Accelerating Fast Fourier Transformation for Image Processing Using Graphics Processing Unit", Journal of Emerging Trends in Computing and Information Sciences, Volume 2, No. 8, pp. 367-375, 2011.

[22] Chen, X., Qiu Y., and Yi, H., "Implementation and Performance of Image Filtering on GPU", Proceedings of IEEE 4th International Conference on Intelligent Control and Information Processing, pp. 514-517, Beijing, China, 2013.

[23] Kirk, D.B., and Wen-mei, W.H., "Programming Massively Parallel Processors: A Hands-On Approach", Morgan Kaufmann, USA, 2010.

[24] Garland M., Grand, S., Nickolls, J., Anderson, J., Hardwick, J., Morton, S., Phillips, E., Zhang, Y., and Volkov, V., "Parallel Computing in CUDA", IEEE Micro, Volume 28, No. 4, pp. 13-27, 2008.

[25] Ahmed, F., Quirem, S., Shin, B.J., Son, D.J., Woo, Y.C., Lee, B.K., and Choi, W., "A Study of CUDA Acceleration and Impact of Data Transfer Overhead in Heterogeneous Environment", Proceedings of Workshop on Unique Chips and Systems UCAS-7, pp. 22-25, New Orleans, USA, 2012.

[26] Duato, J., Pena, A., Silla, F., Fernandez J., Mayo R., and Quintana-Orti, E., "Enabling CUDA Acceleration within Virtual Machines Using rCUDA", Proceedings of International Conference on High Performance Computing, pp. 1-10, Bangalore, India, 2011.

[27] Brigham, E., and Morrow, R., "The Fast Fourier Transform", IEEE Spectrum, Volume 4, No.12, pp. 63-70, 1967.

[28] Smith, S.W., "The Scientist and Engineer's Guide to Digital Signal Processing", California Technical Publication, San Diego, 1997.

[29] Gonzalez, R.C., Woods, R.C., and Eddins, S.L., "Digital Image Processing Using MATLAB", Gatesmark Publishing, USA, 2009.

[30] Lin, C.Y., Wu, M., Bloom, J.A., Cox, I.J., Miller, M.L., and Lui, Y.M., "Rotation, Scale, and Translation Resilient Watermarking for Images", IEEE Transactions on Image Processing, Volume 10, No. 5, pp. 767-782, 2001.

[31] https://developer.nvidia.com/cuda-downloads