Effective Gaussian Blurring Process on Graphics Processing Unit with CUDA

Ferhat Bozkurt, Mete Yağanoğlu, and Faruk Baturalp Günay

***Abstract*—Single threaded applications cannot catch up with parallel systems when it comes to scalability mainly because of the clock frequency limitation of modern CPUs and economical reasons. Using shared memory or distributed memory architectures parallel systems can provide tremendous speed up compared to single threaded systems. CUDA, a shared memory parallel software, is considered to be a powerful language because of its easy thread management aspect and support for GPUs. Gauss blurring is a well-known image processing technique which reduces image noise and detail. Because of the high computation requirement of this technique, single threaded applications exhibit poor performance. In this paper we show that orders of magnitude speed up can be achieved by carrying out this operation on CUDA architecture with the help of high parallelism.**

***Index Terms*—CUDA, GPU, gaussian blur, parallel processing, parallel computing.**

I. Introduction

High performance computers gained significant popularity in the last decade because of the unprecedented growth rate in data sizes. Despite the rapid developments in computer architectures, more powerful systems are needed to solve complex problems and to deal with huge data sets. In order to alleviate this problem, parallel computer systems are used. Compute Unified Device Architecture (CUDA), designed by NVIDIA, is a parallel processing architecture which allows significant performance improvement with the help of Graphics Processing Unit (GPU) [1]. CUDA enabled GPU has been increasingly used in a number of areas such as image and video processing in chemistry and biology, fluid dynamic simulations, computerized tomography (CT) etc. by software developers, scientists and researchers [2]. Gauss blurring is a common technique to reduce details and image noise.

In this study, we apply this technique for different image sizes by running a parallel GPU based CUDA software on a shared-memory architecture and compare the performance with traditional CPU based programs. We show that GPU based CUDA can provide order of magnitude speed up for derivations of Gaussian blurring methods compared to other architectures where there is no parallelism.

II. Contributions and Related Work

CUDA Software Development Kit (SDK), developed by NVIDIA, has been publicly available in February 2007. Alexander Kharlamov and Victor Podlozhnyuk in [3] worked on applying image noise reduction methods on CUDA and adapted some well-known techniques. They achieved 500 frame per second (fps) image processing rate on GeFOrce8800 GTX graphic card for a sample image which has dimensions 320×408.

In this study, we compare application of naive sequential Gaussian blurring process with parallel CUDA architecture with multi-threading for various images with different sizes. Based on our observations such as time and speed-up, an optimal number for grid, block and thread is determined. For instance, using CUDA for fast blurring and filtering in MR application provides significant performance improvement.

III. NVIDIA, CUDA Architecture and Threads

Many studies show that CUDA applications run faster than traditional CPU based applications [2], [4]-[7]. The basic intuition behind this fact is the benefit of extreme parallelism provided by GPU architecture. The GPUs do not require a flow control mechanism as in CPUs since the GPUs are primarily used for repeating image processing for every single pixel, collecting 3D, signal processing applications and complex calculations. Therefore the GPUs are designed to improve parallel data processing capability rather than an improved caching and flow control mechanism as in CPUs.

Installation of distributed systems and usage of parallel programming software systems such as PVM, MPI have always been a challenging task. Moreover, current shared memory parallel software libraries, called as OpenMp, POSIX threads and Win 32/64 threads, cannot divide one software into many parallel threads. The aforementioned shared memory software libraries can work only on CPU. CUDA can overcome those difficulties since it is a powerful parallel programming software that runs on graphical processors. CUDA can run any program written in C on GPU in a multi-threaded fashion since it is a C based language [8].

In CUDA architecture, threads execute on streaming processors (SP). Streaming multiprocessor (SM) is a hardware structure contains eight SP which forms the graphic card itself [2], [9]. CUDA creates parallelism automatically and therefore handles many difficulties. A program written in CUDA is essentially considered to be a serial software called “kernel”. CUDA runs thousands of copies of these kernels in parallel. CUDA is an extension of C language and therefore there is no need to direct programs to CUDA or to change their architectures for making them multi-threaded [10].

Parallel processing on GPU is accomplished through the driver software developed by vendors. When the software is

Manuscript received July 15, 2014; revised October 14, 2014.

The authors are with the Computer Engineering Department, University of Ataturk, Erzurum, 25240 Turkey (e-mail: fbozkurt@atauni.edu.tr, yaganoglu@atauni.edu.tr, baturalp@atauni.edu.tr).

operated, CPU processes the serial part of the code and GPU processes the CUDA code which requires parallel computation. GPU part of code is called “Kernel”. Kernel describes some processes that will be applied to certain data sets. GPU creates a separate kernel copy for each data set. These sets are called thread. A group of 512 threads forms a single block and 65536 threads forms a grid respectively [10].

Each thread has a program counter, register and state register. For image and data processing, millions of parallel threads are created and executed. CUDA technology exploits the parallel computation power of GPUs with many parallel threads. Each thread owns its own decision mechanism. Each thread owns an ID number and can operate on same code in parallel fashion [8], [10].

IV. Gaussian Blurring

The process of blurring an image via a gauss function is called Gaussian blurring. In general, this method is used to reduce image noise and details on graphic software, computer vision and image processing applications. In this technique the main idea is to update the value of a pixel with the average of neighbour pixels. Instead of computing the average of all neighbour pixels, we calculate a weighted average of them. If this value converges on related blur pixel, this shows average weight of each pixel has largest value. Simply, Gaussian blurring method helps to find the weight of each neighbour pixel.

Two dimensional Gaussian functions are shown in Fig. 1. Blur pixel has the peak value when it is compared with other neighbour weighted average pixels. The figure tells us that the maximum distance is five pixels for a picture and it is divided into equal parts. Thus, one can conclude that the window size is ten which is also called kernel size [11]-[13].

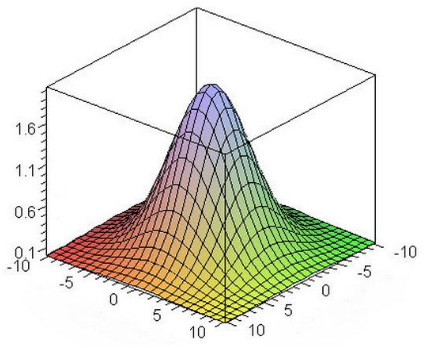

Fig. 1. The graphical representation of the 2-dimensional Gaussian function [12].

The weight of main pixel and the other weights of less weighted pixels can be calculated as in (1) Gaussian equation [11], [13].

$$G(x,y)=\frac{1}{2\prod\sigma^2}e^{-\frac{x^2+y^2}{2\sigma^2}} \tag{1}$$

In this equation the parameters are explained as follows:

- σ blur factor: If this value increase, image will blur.
- e : euler number
- x : Horizontal distance to centre pixel
- y :Vertical distance to centre pixel

According to this equation, x and y distances will be zero for centre pixel. When distance increases from centre pixel x^2+y^2 value will increase and weights will decrease. If this formula is applied to two dimensions, bell shape distribution appears from contour centre point and forms homocentric circle surfaces. These distribution values are used for making a convolution matrix. That matrix is applied to original image. Every new value of pixels can be computed by weighted averaging of itself and neighbour pixels. Centre pixel takes highest average weight. Nonetheless, the weight of the neighbour pixels is directly proportional to the distance to the centre. This process comes to a conclusion with a blur, conserves borders and edges [11].

We must apply kernel matrix to every pixel for obtaining blur image. For instance, if we want blur twenty five valued pixel, we will change twenty five with averaged value of that pixel and its neighbours. We acquire the new weight factors by multiplying changed values and kernel matrix. We maintain this process by cycling and getting a blur image. Expression of time complexity in this algorithm is [11]:

O(rows * cols * kernelwidth * kernelheight)

While applying gauss blur to rows and columns, same results are acquired. Thus, it does not need to pass every pixel. In this case time complexity is [11]:

O(rows * cols * kernelheight + rows * cols* kernelwidth)

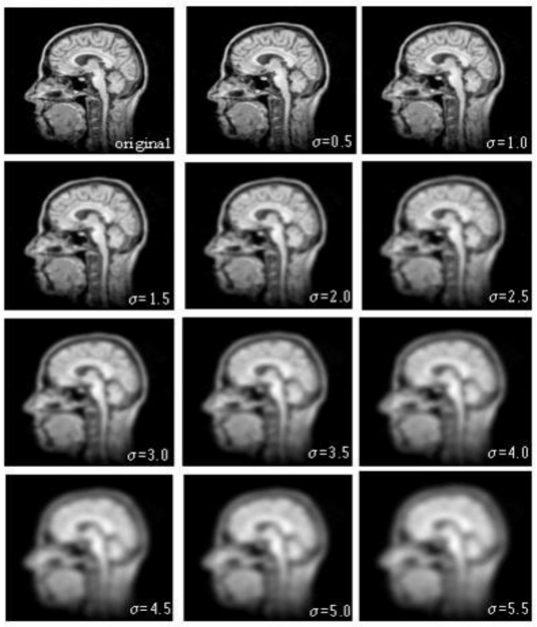

Fig. 2. A Gaussian scale-space of the sagittal view of a MR head image.

We can see application of Gaussian blurring algorithm to MR head image in Fig. 2. The general purpose of this operation is: in order to reduce image noise and details for detecting edges of MR head image [14]. This is to ensure that spurious high-frequency information does not appear in the image that affects the causes of different signals to become indistinguishable. Gaussian blurring is an appropriate solution for MR head images which have no sharp edges.

In Fig. 2, the original image is shown in the upper left (sigma=1). Varying s from 0.5 to 5.5 in steps of 0.5 blurs the other images from left to right and top to bottom [14].

V. Parallel Implementation

This study is implemented in C language in CUDA 5.0, OpenCV 2.4.3 and Visual Studio 2012 environment with the machine configuration parameters in Table I.

TABLE I: Test Machine Configuration

Technical Features	Version and Calculating Capacity
Device	GeForce GT 650M
CUDA Driver Version / Runtime Version	5.0
Total amount of global memory	4096 MByte
(2) Multiprocessors × (192) CUDA Cores/MP	384 CUDA Cores
Warp size	32
Max number of threads per multiprocessor	2048
Maximum number of threads per block	1024
Maximum sizes of each dimension of a block	1024 × 1024 × 64
Maximum sizes of each dimension of a grid	2147483647 × 65535 × 65535

Moreover, programming code in this study works with RGBA pictures. Every channel (Red, Green, Blue and Alpha) symbolizes with one byte (8 bit). Channels are arranged with sum of four bytes and between 0 and 255 (28-1). Gaussian blurring process has been implemented in various sizes images, with maximum 322 =1024 block size parallelization in 384 CUDA CORE GPU’s. The same blurring algorithm was implemented and run on CPU in order to compute the speed up based on execution times.

Some output’s explanation for performance metrics as below:

- **Grid Size:** Number of blocks
- **Thread Block Size:** Number of thread per block
- **GPU Time:** Execution time of algorithm on GPU
- **CPU Time:** Execution time of algorithm on CPU
- **Initialize Kernel Time:** Total time of loading image from the disk and allocating memory for original and modifying images on the host and the device
- **Save Image Disk Time:** Time of copying modified image from device to the host and saving it to disk
- **Speed-up:** Execution time of CPU/ Execution time of GPU

Following steps are used for calculating Gaussian blur [13]:

- **Step 1:** Separating RGBA image to red, green and blue channels.
- **Step 2:** Applying Gaussian blurring method for each time.
- **Step 3:** Taking red, green and blue values into RGBA again.

Algorithm Steps [13]:

- **Step 1:** Allocating memory for RGBA pictures on host and device.
- **Step 2:** Working of blue kernels.
- **Step 3:** Copying blurring picture from device to CPU and saving to disk.

CUDA C extends standard C language and provides describing C functions that are called kernel to users. When kernel functions run N times, they work in parallel on N channels. This property of kernel separates from familiar C functions [15], [16]. The primary parallel construct is a data-parallel, SPMD kernel function. A kernel function invocation explicitly creates many CUDA threads [17]. CUDA thread organization is shown as Fig. 3.

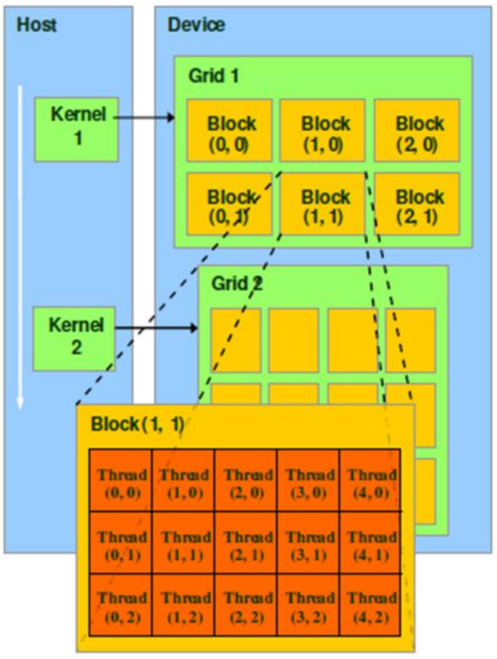

Fig. 3. Example of CUDA grid, block and channel structure [7].

Gaussian blurring process is made on image in the following host-device structure by way of effecting kernel functions and determining thread numbers in parallelization process by changing grid, block sizes. Results of this process and best performance results are compared as shown in Table II. Optimum grid, block and thread numbers are found with these functions.

```
__global__ void KernelFunc(...);
dim3   DimGrid(100, 50);    // 5000 thread blocks
dim3   DimBlock(4, 8, 8);   // 256 threads per block
```

VI. Experimental Results

The experiments are repeated with different image sizes. These are 1024 ×680 = 475KB, 1920 x 1080 = 815KB, 2560 ×1920 = 1.92MB, 4105 ×2727 = 10.2MB and 6245 ×3935 = 17.4MB. In order to observe the impact on execution time, tests are conducted on high resolution images. As shown in Table II, Gaussian blurring process on non-parallel method takes much more time than that on parallel method. This is particularly valid for images with larger sizes.

As shown in Table II, a speed up of 116.07 can be achieved for an image of 1024×680=475KB. The Gaussian blurring process takes about 155 ms on GPU whereas the same operation takes 1780 ms on CPU. We take the best result in 256 thread (16×16×1) and 2752 grid size (64×43×1).

TABLE II: COMPARING PARALLEL AND NON-PARALLEL EXECUTION TIME FOR VARIOUS SIZE IMAGES IN ORDER TO DECIDE OPTIMUM GRID, BLOCK, THREAD SIZE

Image Size	Grid size	Thread block size	CPU Time (ms)	GPU Time (ms)	Initialize Kernel Time(ms)	Save Image Disk Time(ms)	Speed-Up
1024x680 =475Kb	32x22x1	32x32x1	17825	158.473	557	55	112,47
1024x680 = 475Kb	64x43x1	16x16x1	17980	154.901	573	53	116,07
1024x680 = 475Kb	128x85x1	8x8x1	17774	302.274	560	53	58,8
1024x680 = 475Kb	256x170x1	4x4x1	17807	1115.56	559	53	15,96
1024x680 = 475Kb	512x340x1	2x2x1	17776	4275.18	563	53	4,15
1920x1080=815Kb	60x34x1	32x32x1	52960	427.021	606	150	124,02
1920x1080=815Kb	120x68x1	16x16x1	53039	427.65	613	149	124,02
1920x1080=815Kb	240x135x1	8x8x1	52956	838.08	614	150	63,18
1920x1080=815Kb	480x270x1	4x4x1	53030	3254.83	604	151	16,29
1920x1080=815Kb	960x540	2x2x1	Exceed Size	Exceed Size	Exceed Size	Exceed Size	Exceed Size
4105x2727=10.2Mb	129x86x1	32x32x1	285916	2228.74	1507	754	128,28
4105x2727=10.2Mb	257x171x1	16x16x1	290359	2214.13	1519	756	131,12
4105x2727=10.2Mb	514x341x1	8x8x1	287921	4423.99	1519	776	65,08
4105x2727=10.2Mb	1024x682x1	4x4x1	Exceed Size	Exceed Size	Exceed Size	Exceed Size	Exceed Size
4105x2727=10.2Mb	2048x1364	2x2x1	Exceed Size	Exceed Size	Exceed Size	Exceed Size	Exceed Size
6245x3935=17.4Mb	196x123	32x32x1	627881	4785.58	2263	1775	131,2
6245x3935=17.4Mb	391x246x1	16x16x1	632626	4797.69	2275	1780	131,86
6245x3935=17.4Mb	781x492x1	8x8x1	Exceed Size	Exceed Size	Exceed Size	Exceed Size	Exceed Size
6245x3935=17.4Mb	1562x984x1	4x4x1	Exceed Size	Exceed Size	Exceed Size	Exceed Size	Exceed Size
6245x3935=17.4Mb	3124x1968	2x2x1	Exceed Size	Exceed Size	Exceed Size	Exceed Size	Exceed Size

Fig. 4 shows the GPU time and Fig. 5 shows the speed-up graphics for 1024×680=475KB size image. When the number of threads is set to 4=2×2×1, the program triggers an error and cannot accomplish the task as the grid size exceeds the supported size of our current display card.

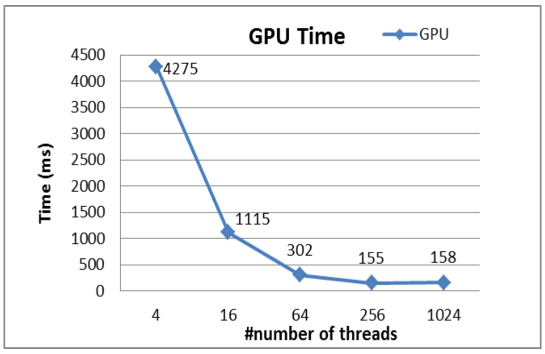

Fig. 4. GPU execution time for 1024×680 resolution and 475 KB size image.

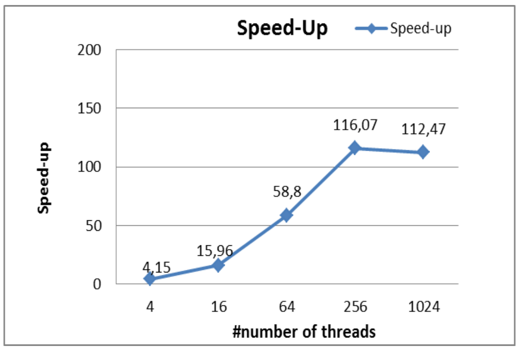

Fig. 5. Speed-up for 1024×680 resolution and 475 KB size image.

VII. CONCLUSION

Gaussian blurring process can be conducted by using CUDA architecture and multi-thread support. In this study we applied this technique on various sizes of images and compared the results. We considered performance metrics such as time and speed-up in order to choose optimal grid, block and thread number. This study proves that height and width values of the desired image is directly related to grid, block, and channel values. We show that mistuning the grid, block and channel parameters can cause fatal errors as it can cause memory access violation. Even if the application runs successfully the observed results would not be correct.

The results show that high performance processing that is typically obtained by expensive hardware can be achieved by using shared-memory parallel CUDA software in a cost-effective fashion. For instance, using CUDA architecture on MR images for blurring and filtering will be a fast and cost effective solution.

REFERENCES

[1] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, and K. Skadron, "A performance study of general-purpose applications on graphics processors using CUDA," *Journal of Parallel and Distributed Computing*, vol. 68, no. 10, pp. 1370-1380, 2008.

[2] M. Garland, L. S. Grand, J. Nickolls, J. Anderson, J. Hardwick, S. Morton, E. Phillips, Y. Zhang, and V. Volkov, "Parallel computing experiences with CUDA," *Micro*, vol. 28, no. 4, pp. 13-27, 2008.

[3] A. Kharlamov and V. Podlozhnyuk. (2007). Image Denoising. [Online]. Available: http://developer.download.nvidia.com/compute/DevZone/C/html/C/src/imageDenoising/doc/imageDenoising.pdf

[4] Z. Yang, Y. Zhu, and P. Yong, "Parallel image processing based on CUDA," in *Proc. International Conference on Computer Science and Software Engineering*, pp. 198-201, 2008.

[5] V. A. Prisacariu and I. Reid, "fastHOG – A real-time GPU implementation of HOG," Technical Report, No. 2310/09, Department of Engineering Science, University of Oxford, 2009.

[6] R. Membarth, H. Dutta, F. Hanning, and J. Teich, "Efficient mapping of streaming applications for image processing on graphics cards," *Transactions on HiPEAC*, vol. 5, no. 3, 2011.

[7] D. B. Kirk and W.W. Hwu, *Programming Massively Parallel Processors*: *A Hands-on Approach*, p. 54, 2010.

[8] S. Ryoo, C. I. Rodrigues, S. S. Baghsorkhi, S. S. Stone, D. B. Kirk, and W. W. Hwu, "Optimization principles and application performance evaluation of a multithreaded GPU using CUDA," in *Proc. PPoPP '08 13th ACM SIGPLAN Symposium on Parallel Programming*, pp. 73-82, 2008.

[9] P. D. Michailidis and K. G. Margaritis, "Accelerating kernel density estimation on the GPU using the CUDA framework," *Applied Mathematical Sciences*, vol. 7, no. 30, pp. 1447–1476, 2013.

[10] M. Akçay, B. Şen, G.M. Orak, and A. Çelik, "Paralel Hesaplama ve CUDA" *6. Uluslar arası İleri Teknolojiler Sempozyumu (İATS'11)*, 16 18 Mayıs 2011, Elazığ, Türkiye.

[11] R. C. Gonzalez and R. E. Woods, *Digital Image Processing*, 2nd ed. Pearson Education, ISBN 81-7808-629-8.

[12] F. M. Waltz and J. W. Miller, "Efficient algorithm for Gaussian blur using finite-state machines," in *Proc. SPIE Conf on Machine Vision Systems for Inspection and Metrology VII*, vol. 3521, pp. 334-341, 1998.

[13] The MSDN website. (2014). [Online]. Available: http://code.msdn.microsoft.com/windowsdesktop/Gaussian-blur-with-CUDA-5-df5db506

[14] The MIPAV Wiki website. (2014). [Online]. Available: http://mipav.cit.nih.gov/pubwiki/index.php/Filters_(Spatial):_Gaussian_Blur

[15] I. K. Park, N. Singhal, M. H. Lee, S. Cho, and C.W. Kim, "Design and performance evaluation of image processing algorithms on GPUs," *IEEE Transactions on Parallel and Distributed Systems*, vol. 22, issue 1, pp. 91-104, 2011.

[16] P. P. Shete, P. P. K. Venkat, D. M. Sarode, M. Laghate, S. K. Bose, and R. S. Mundada, "Object oriented framework for CUDA based image

processing," in *Proc. 2012 International Conference on Communication, Information & Computing Technology (ICCICT)*, pp. 1-6, 2012.

[17] J. A. Stratton, S. S. Stone, and W. W. Hwu, "MCUDA: An efficient implementation of CUDA kernels for multi-core CPUs," in *Lecture Notes in Computer Science*, J. N. Amaral, Ed. Springer-Verlag Berlin, Heidelberg, 2008, vol. 5335, pp. 16-30.

Ferhat Bozkurt is a research assistant in the Department of Computer Engineering, at the Ataturk University, Erzurum. He received his B.S. degree in the Department of Computer Engineering from the Dokuz Eylül University. He received his M.S. in the Department of Computer Engineering, at the Ataturk University. He is doing his Ph.D. in the Department of Computer Engineering, at the Karadeniz Technical University. His current research interests are in the fields of software, parallel computer architectures, high-performance computing, parallel algorithms, computer graphics and visualization and database applications.

Mete Yağanoğlu is a research assistant in the Department of Computer Engineering, at the Ataturk University, Erzurum. He received his B.S. in the Department of Computer Engineering from the Erciyes University. He received his M.S. in the Department of Computer Engineering, at the Ataturk University. He is doing his Ph.D. in the Department of Computer Engineering, at the Karadeniz Technical University. His current research interests are in the fields of software, human-computer interface, parallel computer architectures, data mining and database applications.

Faruk Baturalp Günay is a research assistant in the Department of Computer Engineering, at the Ataturk University, Erzurum. He received his B.S. degree in the Department of Electrical-Electronic Engineering from the Erciyes University. He received his M.S. degree in the Department of Electrical-Electronic Engineering, at the Ataturk University. He is doing his M.S. in the Department of Computer Engineering, at the Karadeniz Technical University. His current research interests are in the fields of hardware, parallel and distributed systems, wireless communications, optical communications, wireless sensor networks.