# PERFORMANCE ANALYSIS OF MORPHOLOGICAL OPERATIONS IN CPU AND GPU FOR ACCELERATING DIGITAL IMAGE APPLICATIONS

T.Kalaiselvi, P.Sriramakrishnan and K.Somasundaram

Department of Computer Science and Applications,
Gandhigram Rural Institute –Deemed University
Gandhigram

***ABSTRACT***

*In this paper, we evaluate the performance of morphological operations in central processing unit (CPU) and graphics processing unit (GPU) on various sizes of image and structuring element. The languages selected for algorithm implementation are C++, Matlab for CPU and CUDA for GPU. The parallel programming approach using threads for image analysis is done on basic entities of images. The morphological operations namely dilation and erosion are purely depends upon local neighborhood information of each pixel and thus independent. GPU capable to create more number of threads. Here thread per pixel of the image is created to execute the algorithms with the neighbors of relative pixels. Finally the speed performance of all algorithms on conventional processor CPU and parallel processor GPU are computed and compared. Dilation operation in GPU is up to 5 times faster than CPU C++ code and up to 4 - 11 times faster than CPU MATLAB code, likewise Erosion operation in GPU is up to 2 times faster than CPU C++ code and up to 6- 12 times faster than CPU MATLAB code when image size varying from 256 × 256 to 1024 × 1024. Further it shows that the performance of GPU implementation is gearing up when the image size is increased. While changing the structuring element size with 1024 × 1024 image the Dilation operation in GPU is up to 3-5 times faster than CPU C++ code and up to 10 - 35 times faster than CPU MATLAB code, likewise Erosion operation in GPU is up to 2 – 6 times faster than CPU C++ code and up to 12- 46 times faster than CPU MATLAB code.*

***KEYWORDS***

*Graphics processing unit, CUDA, Dilation and Erosion, Parallel processing, Morphological operations*

## 1.INTRODUCTION

Graphics processing unit (GPU) is a multi-core computer chip that performs rapid mathematical calculations, allowing very efficient manipulation of large blocks of data, primarily for rendering images, animations and video for the computer's screen [1]. A GPU is able to render images more quickly than a central processing unit (CPU) as it performs multiple calculations at the same time using its parallel processing architecture. It is also used for smooth decoding and rendering of 3D animations and video. GPU's have lot of computational hardware resources. Most of the time the large resource of GPU are unused when graphics applications like games are not run. Computation using GPU has a huge edge over CPU in speed. Hence it is one of the most interesting areas of research in the field of modern industrial research and development for accelerating computing speed. Image processing algorithms in general are good candidates for exploiting GPU capabilities. The parallelization is naturally provided by basic entity of an image.

DOI :10.5121/ijcsity.2016.4102 

To process these images, parallel processing capabilities in GPU can be used. The most common GPU programming model is compute unified device architecture (CUDA).

Ghorpade et al., [2] gave an analysis on the architecture of CUDA, CPU and GPU performance. In 2014, Thurley and Danell [3] worked in open source CUDA with morphological operations. They tested 8 bit and 32 bit images with morphological operations in CUDA. They realized CUDA-GPU does not have sufficient shared memory for process the 32 bit images. Jagannathan et al., [4] implemented vHGW algorithm for dilation and erosion independent of structuring element size has been implemented for different types of structuring elements of an arbitrary length and along arbitrary angle on CUDA programming environment with GPU hardware as GeForce GTX 480. They showed the results with maximum performance gain of 20 times than the conventional serial implementation of algorithm in terms of execution time. Harish and Narayanan [5] has implemented breadth first shortest path and single source shortest path algorithm in GPU. Their result showed that GPU is 20-50 times faster than CPU.

In MRI processing, Somasundaram and Kalaiselvi [6] showed that it requires more time to implement an algorithm for automatic segmentation of MRI scans of human brain in CPU. They realized CPU computation power not sufficient to process the large volume of medical data with complex algorithms. Yadav et al., [7] implemented texture based similarity function using GPU. They found that GPU is 30 times faster than CPU. ELEKS [8] has developed post processing of MRI images using GPU. Additionally they used parallel imaging methods to reduce the scan time in singular value decomposition (SVD). They achieved 155X than CPU. In 2015, Jing et al., [9] developed a fast parallel implementation of group independent component analysis (PGICA) for functional magnetic resonance imaging (FMRI). This proposed work demonstrated the speed accuracy of their experiments. But they realized the device memory constraints for large amounts of subject's data processing.

In this paper, we evaluate the performance of morphological operations in CPU and GPU. The languages selected for algorithm implementation are C++, Matlab for CPU and CUDA for GPU. The parallel programming approach using threads for image analysis is done on basic entities of images. This analysis is based on the nature of information required to process the image. Local processing includes a single pixel and its neighbors. Dilation and erosion comes under this category, because these algorithms purely depend upon the local neighborhood information of each pixel and thus independent. A thread for every pixel of the image is created to execute the algorithms with the neighbors of relative pixels. Here the number of threads creation depends on the nature of processor architecture. GPU has more cores and threads then conventional CPU. When this GPU model invoked with the possible threads, it's execute the threads in parallel and produce the results for the images quickly. Finally the speed performance of all algorithms on conventional processor CPU and parallel processor GPU on various size of images and structuring elements are computed and compared.

The remaining part of this paper is organized as follows. Section 2 describes the feature of GPU-CUDA programming then section 3 explains the implementation details of the morphological operations in CPU and GPU. Section 4 gives experimental results and discussion and section 5 concludes the paper.

## 2.GPU-CUDA PROGRAMMING

There are varieties of GPU programming models available in the market for accelerating computational capability of a complex system. CUDA is one such model. It is a parallel computing platform and programming model introduced by NVIDIA in late 2006. NVIDIA

released the documentation about CUDA programming. CUDA is an open source and minimal extension of the C and C++ programming languages [10]. CUDA is a new hardware and software architecture for issuing and managing computations on the GPU as a data-parallel computing device without the need of mapping them to a graphics API. Other GPU programming languages are openCL (open Computing Library) and DirectCompute.

## 2.1.CUDA Programming Model

For the programmer the CUDA model is a collection of threads running in parallel and its architecture as shown in Figure 1. A kernel is a function or routine that executes in parallel across a set of parallel threads. The programmer organizes these threads into a hierarchy of grids of thread blocks. A thread block is a set of concurrent threads that can cooperate among themselves through barrier synchronization and shared access to a memory space private to the block. A grid is a set of thread blocks that may each be executed independently and thus may execute in parallel. When invoking a kernel, the programmer specifies the number of threads per block and the number of blocks making up the grid. Each thread is given a unique thread ID number threadIdx within its thread block, numbered 0, 1, 2, ..., blockDim–1, and each thread block is given a unique block ID number blockIdx within its grid. CUDA supports thread blocks containing up to 1024 threads. For convenience, thread blocks and grids may have one, two, or three dimensions, accessed via .x, .y, and .z index fields.

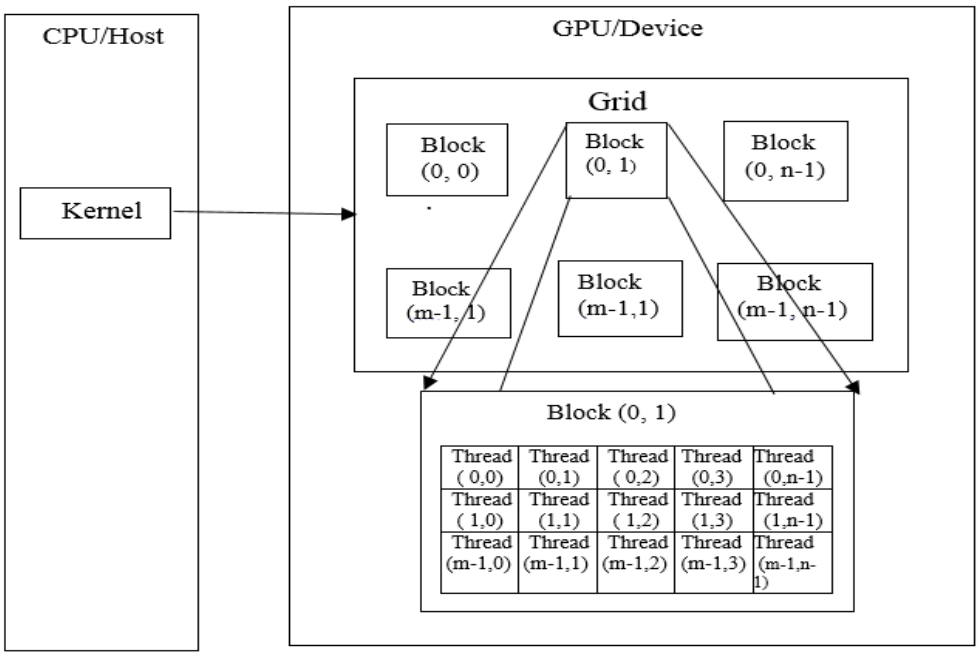



Figure 1: Programming model of GPU- CUDA

## 2.2.CUDA Memory Model

The GPU-CUDA memory model is given in Figure 2. Threads may access data from multiple memory spaces during their execution. Each thread has a private local memory. CUDA uses this memory for thread-private variables that do not fit in the thread's registers, as well as for stack frames and register spilling. Each thread block has a shared memory visible to all threads of the

block that has the same lifetime as the block. Finally, all threads have access to the same global memory. GPU have some special set of memories are cache, texture, constant memory [11]. GPU have small cache memory compared than CPU. Texture memory speclilized in fetching and cacheing data from 2D and 3D textures. Constant memory useful for storing data that remains unchanged. GPU have faster memory hierarchical as follows registers, shared memory, cache memory, constant memory, global memory and finally host memory (CPU). The execution flow of GPU-CUDA is a five step process given in Figure 3.

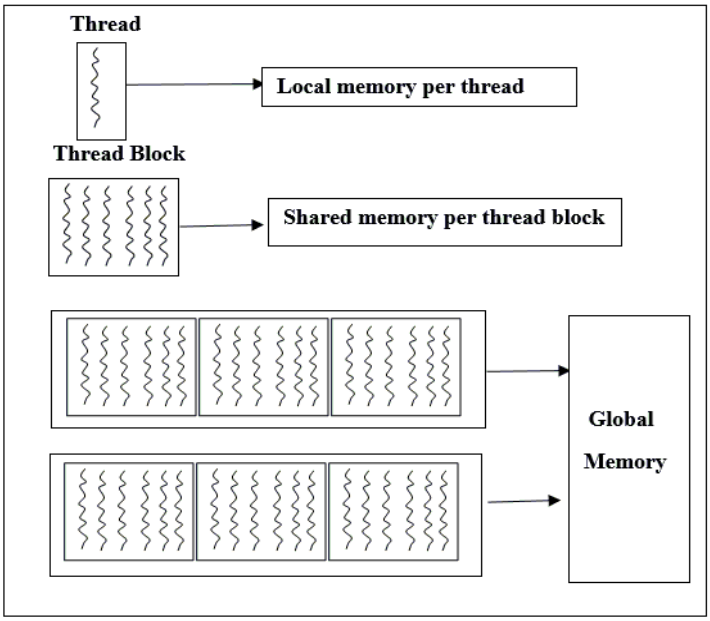



Figure 2: Memory model of GPU- CUDA

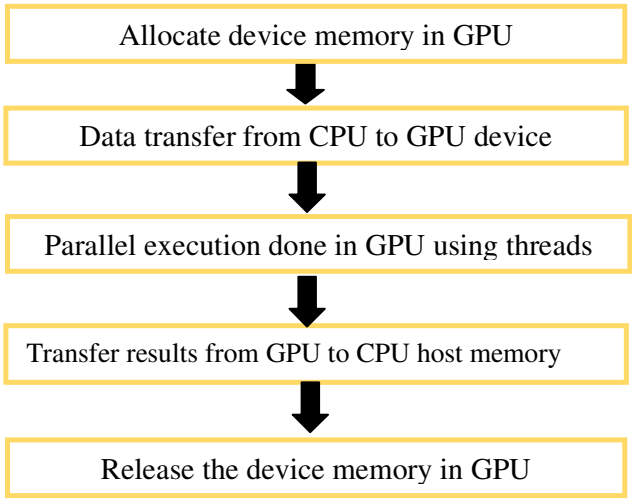



Figure 3. Execution Flow of GPU- CUDA

## 3.MORPHOLOGICAL OPERATIONS

Morphological operations are methods for processing images based on their shapes [12] [13]. It is very often used in applications where the shape of the object is an issue. In mathematical morphology, shape is controlled by "structuring element". The structuring element determines the precise details of the effect of the operators on the image. The structuring element consist of a pattern, specified as the coordinates of a number of discrete points relative to some origin. The structuring element shape may varying based on the applications [14]. Some different shapes of structuring elements are given in Figure 4. They are diamond, disk, octagon, square, arbitrary and line. The rounded cell in the structuring element denoted as origin. The changes in the size or shape of the structuring element gives reflection in the output image.

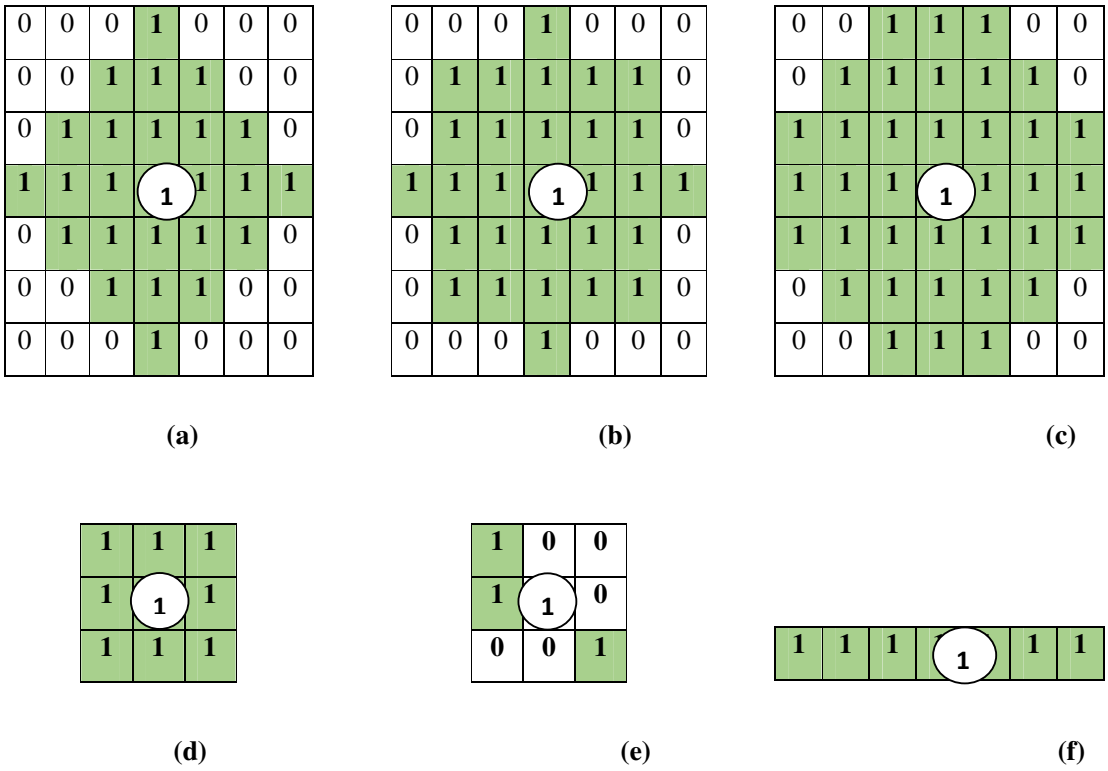



Figure 4. Types of structuring elements (a) Diamond (b) Disk (c) Octagon (d) Square (e) Arbitrary (f) Line

The primary morphological operations are dilation and erosion. Dilation and erosion are dual operations in that they have opposite effects in the resulting image. Dilation adds pixels to the boundaries of the objects while erosion removes a larger of pixels on the object boundaries. Neither erosion nor dilation is an invertible transformation. The implementation details of morphological operations in CPU and GPU is given below.

### Dilation

Dilation is the operation used to extent the thickness is controlled by a shape refereed by various structuring elements. In mathematically, dilation $\oplus$ combines two sets using vector addition [15]. The two sets referred to be image and corresponding structuring elements. The dilation $X \oplus B$ is

the point set of all possible vector additions of pairs of elements, one from each of the sets X and B,

$$X \oplus B = \{p \in \varepsilon^2 : p = x + b, x \in X \text{ and } b \in B\} \quad (1)$$

where, X is the image with M × N size and B is the structuring element. It is typically applied to binary images, but work also on grayscale images. The basic effect of the operator on a binary image is to gradually enlarge the boundaries of regions of foreground pixels. Thus areas of foreground pixels grow in size while holes within those regions become smaller.

## Erosion

Erosion is an operation used to shrink the object of the binary image. In mathematically, Erosion $\ominus$ combines two sets using vector subtraction of set elements [15]. The erosion $X \ominus B$ is the point set of all possible vector subtractions of pairs of elements, one from each of the sets X and B

$$X \ominus B = \{p \in \varepsilon^2 : p = x + b \in X \text{ for every } b \in B\} \quad (2)$$

where, X is the image with M × N size and B is the structuring element or kernel. The basic effect of the operator on a binary image is to erode away the boundaries of regions of foreground pixels. Thus areas of foreground pixels shrink in size, and holes within those areas become larger.

## CPU Implementation

Morphological CPU implementation done by using C++ and Matlab. Matlab is an interpreted language so it is usually slower than compiled languages like C and C++. That is the reason behind we take this two languages for CPU implementation. This operations implemented in CPU using conventional single threaded approach with the configurations of CPU as given below:

| | |
|---|---|
| Processors Name | : Intel - I5 2500 |
| Speed | : 2.9 GHz |
| RAM | : 4 GB |
| Languages | : Matlab and Visual C++ |

## GPU Implementation

The GPU implementation of morphological operations done by CUDA. CUDA is capable to create more number of threads per image. Here we can create thread per pixel and each thread have unique thread id [16]. Each thread executes the kernel code simultaneously and produce the output. CUDA 7.0 is used for this implementation. GPU hardware Nvidia Quadro K5000 is used and its complete specification is given below:

| | |
|---|---|
| Processors name | : NVIDIA Quadro K5000 |
| Speed | : 1.4GHz |
| Multiprocessor count | : 8 |
| Number of cores | : 1536 (8 x 192) |
| Memory | : 4GB |
| Threads | : 1024 per Block |

Language : CUDA 7.0
Memory bandwidth : 173 GB/s
Memory clock : 5.4GHZ
Transistor count : 3540 million
Register per block : 49152
Compute capability : Version 3.0
Max thread dimension : (1024, 1024, 64)
Max grid dimension : (2147483647, 65535, 65535)

## CUDA_Dilation

Read a binary image size of ROW × COL and store the two dimensional values to one dimensional IN input array row wise. Create same size array OUT to store the resultant image. Copy binary values of IN to OUT array. Define M × N structuring elements SE for further process. Allocate IN, OUT arrays in GPU memory. Transfer all data from CPU to GPU. Create ROW×COL threads and call kernel function. Execute all threads parallel using thread id. If value of SE array equal to 1 then corresponding pixel positions of OUT array is set to one. Get back resultant OUT array from GPU to CPU. CUDA implementation of dilation is given in Algorithm 1.

**Algorithm 1:** CUDA_Dilation with M × N Structuring Element

CUDA_DILATION (IN, OUT, SE)

```
Create a one dimensional intensity arrays IN and OUT with the size of ROW × COL
Read a binary image with ROW × COL pixels and copy to IN in row wise
Define M × N structuring element SE
Transfer the IN, OUT, SE array from host to device memory
Create ROW × COL threads and call kernel for parallel execution
for all ROW × COL  do in parallel
    tid = get thread id
        if tid < ROW × COL then do
            if IN [tid] equal to 1 then
                if SE [0] equal to 1 then
                            OUT [tid - COL-1] = 1;
                  end
                if SE [1] equal to 1 then
                            OUT [tid - COL] = 1;
                  end
                if SE [2] equal to 1 then
                            OUT [tid - COL + 1] = 1;
                  end
                if SE [3] equal to 1 then
                            OUT [tid-1] = 1;
                  end
                if SE [4] equal to 1 then
                            OUT [tid] = 1;
                  end
                if SE [5] equal to 1 then
                            OUT [tid+1] = 1;
                  end
                if SE [6] equal to 1 then
                            OUT [tid + COL-1] = 1;
```

```
                end
              if SE [7] equal to 1 then
                        OUT [tid + COL] = 1;
                end
              if SE [8] equal to 1 then
                        OUT [tid + COL + 1] = 1;
                end
        end
      end
end
```

Transfer OUT array from device to host memory.

### CUDA_Erosion

Create ROW × COL size of one dimensional IN array. Read a binary image with the size of ROW × COL to IN row wise. Create same size OUT array for resultant image. Copy binary values of IN to OUT array. Define M × N structuring elements SE with all 1's and Mat array for further processing. Allocate same size memory to IN, OUT, SE and Mat arrays in GPU. Transfer the data from CPU memory to GPU memory. Create ROW × COL threads and call kernel function. Execute all threads parallel using thread id. If the value of IN array to thread id equal to 1 then assign corresponding M × N neighbouring pixels to Mat array. Next compare Mat with SE if each value equals then set OUT [tid] to zero. Finally copy the resultant OUT array from GPU to CPU. CUDA implementation of erosion is given in Algorithm 2.

**Algorithm 2:** CUDA_Erosion with M × N Structuring Element

CUDA_EROSION (IN, OUT, SE, Mat)

```
Create a one dimensional intensity arrays IN and OUT with the size of ROW × COL
Read a binary image with ROW × COL pixels and copy to IN in row wise
Define M× N structuring element SE and initialize Mat array with zero
Transfer the IN, OUT, SE, Mat arrays from host to device
Create ROW × COL threads and call kernel for parallel execution
for all ROW × COL do in parallel
   tid = get thread id();
     if tid < ROW × COL then do
        if IN [tid] equal to 1 then
            assign values (IN [tid - COL-1], IN [tid - COL], IN [tid – COL + 1], IN [tid - 1], IN
[tid], IN [tid+1],
             IN [tid + COL-1], IN [tid + COL], IN [tid+COL+1]) to Mat
                if Mat equal to SE then do
                    OUT [tid] =0;
                end
          end
      end
 end
```

Transfer OUT array from GPU device memory to CPU host memory

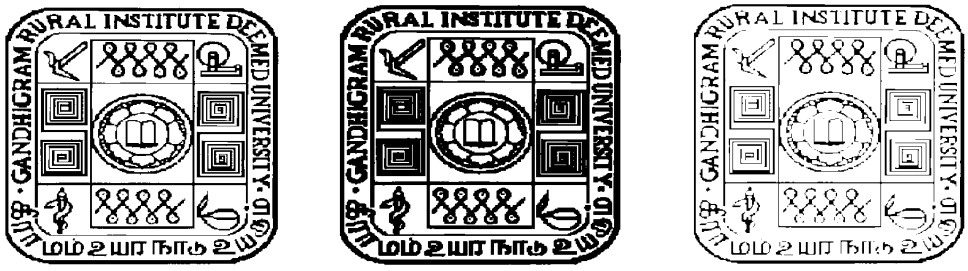
## 4.RESULTS AND DISCUSSION

The qualitative results of dilation and erosion using 3 × 3 structuring elements on a sample image as shown in Figure 5. We carried out GPU implementation in Algorithm 1 and 2. The square shaped structuring element (Figure 4 d) used for this CPU and GPU implementation. The execution time taken by morphological operations are recorded and are given in Table 1. Table 1

shows the time taken by each processor for performing pixel based local approaches namely dilation and erosion. Here the performance is given by three sizes of image 256 × 256, 512 × 512 and 1024 × 1024 on 3 × 3 square shaped structuring element. Computing times of CPU implementation algorithms in C++ (A) and Matlab (B) also compared with GPU (C). For comparison, results obtained in this proposed work by GPU- CUDA are shown in the Figure 6.

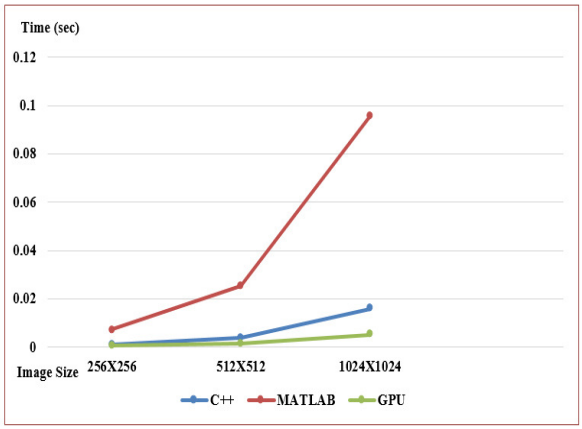
Table 1. Execution time for CPU and GPU implementation using 3 × 3 square shaped structuring element on various size of images

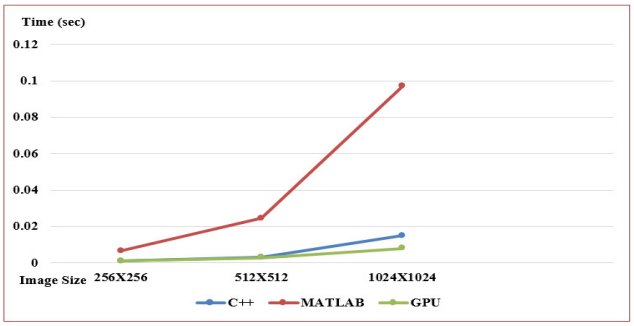
| Operations | Execution Time in sec | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | Image size= 256 × 256 | | | | | = 512 × 512 | | | | | = 1024 × 1024 | | | | |
| | Processor | | | Speed up | | Processor | | | Speed up | | Processor | | | Speed up | |
| | CPU | | GPU (C) | A/C | B/C | CPU | | GPU (C) | A/C | B/C | CPU | | GPU (C) | A/C | B/C |
| | C++ (A) | Matlab (B) | | | | C++ (A) | Matlab (B) | | | | C++ (A) | Matlab (B) | | | |
| **Dilation** | 0.001 | 0.00349 | 0.00073 | **1.3** | **4.7** | 0.004 | 0.0146 | 0.00146 | **2.7** | **10** | 0.016 | 0.059158 | 0.0056 | **3** | **10.5** |
| **Erosion** | 0.001 | 0.006507 | 0.00101 | **1** | **6.4** | 0.003 | 0.0246 | 0.00281 | **1** | **9** | 0.015 | 0.09684 | 0.005 | **2** | **12** |



**(a)** **(b)** **(c)**

Figure 5. Morphological operations using 3 × 3 square shaped structuring element (a) Sample image (b) Dilation (c) Erosion






**(a)** **(b)**

Figure 6. Results for CPU and GPU implementation a) Dilation b) Erosion on various sizes of images using 3 × 3 structuring element

The experiment of morphological operations done on various sizes of structuring element on fixed image size. Table 2 shows the time taken by various sizes of square shaped structuring element on image size 1024 × 1024. Computing times of CPU implementation algorithms in C++ (A) and Matlab (B) also compared with GPU (C). For comparison, results obtained in this proposed work by GPU- CUDA are shown in the Figure 7.

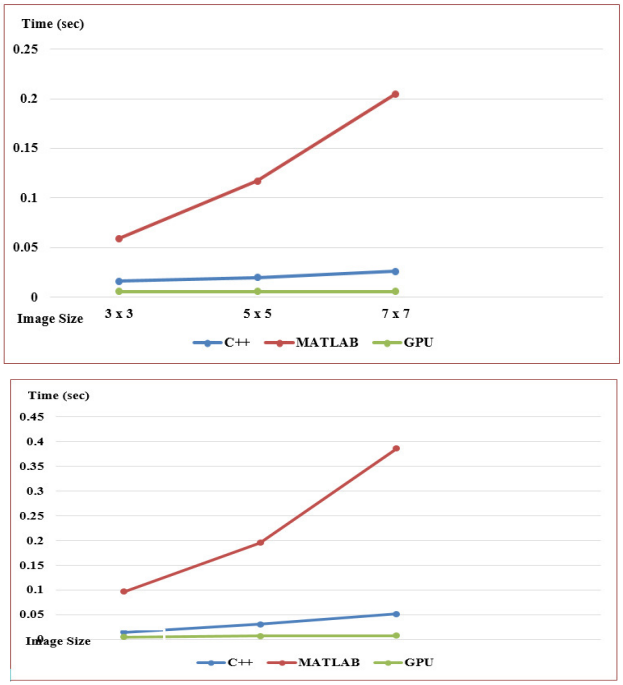
From Table 1 and Figure 6, we observed that GPU implementation can access every pixel in parallel and handling the ROW × COL number of threads as independent process. GPU is 4 - 10 times faster than CPU Matlab code and up to 3 times faster than CPU C++ code in dilation operation when image sizes are different on static size structuring element. Erosion gives 6 – 12 times faster than CPU Matlab and up to 2 times better than CPU C++. Further it shows that the performance of GPU implementation is gearing up when the image size is increased.

From Table 2 and Figure 7, CPU C++ and GPU are almost static time taken when handling various sizes of structuring element on same size image. But Matlab taking more time for various sizes of structuring elements. GPU in dilation gives 3 – 5 times faster than C++ and 10 – 35 times faster than Matlab. GPU in erosion gives 2 – 6 times faster than C++ and 12 – 46 times faster than Matlab. GPU is much faster than Matlab compared with C++ because Matlab is an

interpreted language so its usually slower than C++. More number of data parallelization gives attractive results. GPU not suitable for very less number of threads. Here we used light weight mathematical calculations and thus GPU gives very less speedup (up to 46 times). If we use heavy complex mathematical calculations like brain tumour extraction and visualization then GPU will give better results.

Table 2. Execution time for CPU and GPU implementation on 1024 × 1024 size image using various size of structuring element.

| Operations | Execution Time in sec | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | Image 1024 × 1024 | | | | | | | | | | | | | | |
| | Structuring element = 3 × 3 | | | | | = 5 × 5 | | | | | = 7 × 7 | | | | |
| | Processor | | | Speed up | | Processor | | | Speed up | | Processor | | | Speed up | |
| | CPU | | GPU (C) | A/C | B/C | CPU | | GPU (C) | A/C | B/C | CPU | | GPU (C) | A/C | B/C |
| | C++ (A) | Matlab (B) | | | | C++ (A) | Matlab (B) | | | | C++ (A) | Matlab (B) | | | |
| **Dilation** | 0.016 | 0.059158 | 0.00564 | **3** | **10.4** | 0.020 | 0.117 | 0.00576 | **3.5** | **20** | 0.026 | 0.205 | 0.005807 | **4.5** | **35** |
| **Erosion** | 0.015 | 0.09684 | 0.005 | **2** | **12.1** | 0.031 | 0.19568 | 0.007473 | **4** | **26.1** | 0.052 | 0.386 | 0.00837 | **6.2** | **46.1** |




(a) (b)

Figure 7. Results for CPU and GPU implementation a) Dilation b) Erosion on various sizes of structuring element on image size 1024 × 1024.

## 5. CONCLUSION

In this work, we have implemented morphological operations using GPU - CUDA model. Independent pixels can be accessed parallel for morphological operations. The GPU based coding yielded speedup values in the order of 46 times compared to conventional processor CPU with its conventional single threaded approach. Further it showed that the performance of GPU implementation is gearing up when the image size as well as structuring element size is increased.

## ACKNOWLEDGEMENTS

We gratefully acknowledge the support of NVIDIA Corporation Private Ltd, USA with the donation of the Quadro K5000 GPU used for this research.

## REFERENCES

[1] S. Tariq, (2011) "An Introduction to GPU Computing and CUDA Architecture", *NVIDIA Corporation*, Vol. 6, No. 5.

[2] J Ghorpade, J. Parande, M. Kulkarni & A. Bawaskar, (2012) "GPGPU Processing in CUDA Architecture", *Advance Computing: An international Journal*, Vol. 3, No. 1, pp. 105-119.

[3] Matthew J. Thurley & Victor Danell, (2012) "Fast Morphological Image Processing Open-Source Extensions for GPU processing with CUDA", *IEEE Journal of Selected Topics in Signal Processing*, Vol. X, NO. X, pp. 1-7.

[4] G. Jagannathan, R. Subash & K. Senthil Raja, (2014) "A Novel Open Source Morphology Using GPU Processing With LTU-CUDA", *International Journal for Advance Research in Engineering and Technology*, Vol. 2, No. 1, pp.1 - 5.

[5] P. Harish & P. J. Narayanan, (2007) "Accelerating large graph algorithms on the GPU using CUDA", *Proceedings of the International Conference on High performance computing*, pp. 197-208.

[6] K. Somasundaram & T. Kalaiselvi, (2011) "Automatic Brain Extraction Methods for T1 magnetic Resonance Images using Region Labelling and Morphological", *Computers in Biology and Medicine*, Vol. 41, No. 8, pp. 716-725.

[7] K. Yadav, A. Srivastava & M. A. Ansari, (2011) "Parallel Implementation of Texture based Medical Image Retrieval in Compressed Domain using CUDA", *International Journal on Computer Applications*, Vol. 1, pp. 53-58.

[8] ELEKS, "CUDA-Accelerated Image Processing for Healthcare", http://www.eleks.com, Last accessed on 20th Jan 2016.

[9] Y. Jing, W. Zeng, N. Wang, T. Ren, Y. Shi, J. Yin & Q. Xu, (2015) "GPU-based parallel group ICA for functional magnetic resonance data", *computer methods and programs in biomedicine*, Vol. 119, pp. 9-16.

[10] NVIDIA CUDA Compute Unified Device Architecture Programming Guide, Version 0.8, *NVIDIA Corporation*, 2007.

[11] Erik Smistad, Thomas L. Falch, Mohammadmehdi Bozorgi, Anne C. Elster, Frank Lindseth, Medical image segmentation on GPUs – A comprehensive review, Medical Image Analysis, 20 (1) (2015), 1-18.

[12] A. K. Jain, (1989) "Fundamentals of Digital Image Processing", *PHI*.

[13] Rafael C. Gonzalez & Richard E. Woods, (2002) Digital Image Processing, Second edition, *Prentice Hall*.

[14] T. Kalaiselvi, (2011) "Brain Portion Extraction and Brain Abnormality Detection from Magnetic Resonance Imaging of Human Head Scans"*, Pallavi Publications South India Pvt Ltd*.

[15] Milan Sonka, Vaclav Hlavac, Roger Boyle, (2013) "Image Processing, Analysis, and Machine Vision", Third edition, *Cengage Learning*.

[16] T. Kalaiselvi, P. Sriramakrishnan and K. Somasundaram, (2015) "Morphological Operations by Parallel Processing using GPU – CUDA Model*", Proceedings of National Conferences on New Horizons in Computational Intelligence and Information Systems (NHCIIS),* Vol.1, pp. 3 - 7.

## Authors

**T.Kalaiselvi** is currently working as an Assistant Professor in Department of Computer Science and applications, Gandhigram Rural Institute, Dindigul, Tamilnadu, India. She received her Bachelor of Science (B.Sc.) degree in Mathematics and Physics in 1994 & Master of Computer Applications (M.C.A) degree in 1997 from Avinashilingam University, Coimbatore, Tamilnadu, India. She received her Ph.D. degree from Gandhigram Rural University in February 2010. She has completed a DST sponsored project under Young Scientist Scheme. She was a PDF in the same department during 2010-2011. An Android based application developed based on her research work has won First Position in National Student Research Convention, ANVESHAN-2013, organized by Association of Indian Universities (AUI), New Delhi, under Health Sciences Category. Her research focuses on MRI of Human Brain Image Analysis to enrich the Computer Aided Diagnostic process, Telemedicine and Teleradiology Technologies.

**P.Sriramakrishnan** received his Bachelor of Science (B.Sc.) degree in 2011 from Bharathidasan University, Trichy, Tamilnadu, India. He received Master of Computer Application (M.C.A) degree in 2014 from Gandhigram Rural Institute-Deemed University, Dindigul, Tamilnadu, India. He worked as Software Developer in Dhvani Research and Development Pvt. Ltd, Indian Institute of Technology Madras Research Park, Chennai during May 2014 – March 2015. He is currently pursuing Ph.D. degree in Gandhigram Rural Institute- Deemed University. His research focuses on Medical Image Processing and Parallel Computing. He has qualified UGC-NET for Lectureship in June 2015.

**K.Somasundaram** received his Master of Science (M. Sc) degree in Physics from the University of Madras, Chennai, India in 1976, the Post Graduate Diploma in Computer Methods from Madurai Kamaraj University, Madurai, India in 1989 and the Ph.D degree in theoretical Physics from Indian Institute of Science, Bangalore, India in 1984. He is presently working as Professor at the Department of Computer Science and Applications, Gandhigram Rural Institute, Dindigul, India. From 1976 to 1989, he was a Professor with the Department of Physics at the same Institute. He was senior Research Fellow of Council Scientific and Industrial Research (CSIR) Govt. of India, in 1983. He was previously a Researcher at the International Centre for Theoretical Physics, Trieste, Italy and Development Fellow of Commonwealth Universities, at Edith Cowan University, Perth, Australia. His research interests are image processing, image compression and Medical imaging. He is Life member of Indian Society for Technical Education, India and Life member in Telemedicine society of India. He is also a member of IEEE USA.