I.J. Intelligent Systems and Applications, **2018, 1, 58-68**
Published Online January 2018 in MECS (http://www.mecs-press.org/)
DOI: 10.5815/ijisa.2018.01.07

Performance of Medical Image Processing Algorithms Implemented in CUDA running on GPU based Machine

T. Kalaiselvi
Department of Computer Science and Applications,
The Gandhigram Rural Institute - Deemed University, Tamilnadu, India.
E-mail: kalaiselvi.gri@gmail.com

P. Sriramakrishnan and K. Somasundaram
Department of Computer Science and Applications,
The Gandhigram Rural Institute - Deemed University, Tamilnadu, India.
E-mail: sriram0210@gmail.com, ka.somasundaram@gmail.com

Received: 10 May 2017; Accepted: 06 July 2017; Published: 08 January 2018

***Abstract*—**This paper illustrates the design and performance evaluation of few algorithms used for analysing the medical image volumes on the massive parallel graphics processing unit (GPU) with compute unified device architecture (CUDA). These algorithms are selected from the general framework, devised for computer aided diagnostic (CAD) system. The CAD system used for analysing large medical image datasets are usually a pipeline processing that includes a variety of image processing operations. A MRI scanner captures the 3D human head into a series of 2D images. Considerable time spent in pre and post processing of these images. Noise filters, segmentation, image diffusion and enhancement are few such methods. The algorithms are chosen for study requires local information, available in few pixels or global information available in the entire image. These problems are best candidates for GPU implementation, since the parallelism is naturally provided by the proposed Per-Pixel Threading (PPT) or Per-Slice Threading (PST) operations. In this paper implement the algorithms for adaptive filtering, anisotropic diffusion, bilateral filtering, non-local means (NLM) filtering, K-Means segmentation and feature extraction in 1536 core's NVIDIA GPU and estimated the speed up gained. Our experiments show that the GPU based implementation achieved typical speedup values in the range of 3-338 times compared to conventional central processing unit (CPU) processor in PPT model and up to 30 times in PST model.

***Index Terms*—**Medical images, image processing, GPU, CUDA, parallel processing.

I. Introduction

Graphics processing units (GPUs) are evolving at a rapid rate in recent years, partly due to the increasing needs of active computer graphics designing community and parallel computing [1]. GPU is a multi-core computer chip that performs rapid mathematical calculations, making very efficient transfer of large blocks of data, primarily for rendering images for games, animations and video on a computer screen. 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 running. 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 medical research and development for accelerating computing speed. In the medical domain, GPU used to process the huge volumes of data for the medical diagnosis to reduce computational cost.

Image processing algorithms in general are good candidates for exploiting GPU capabilities. The parallelism is naturally provided by per-pixel (or per-voxel) operations. MRI processing of human head scans is one such area to effectively utilize the GPU resources. A MRI of human head scans are a stack of two dimensional images from a 3D volume. To process these slices, parallel processing capabilities in GPU can be used. By developing appropriate parallel algorithms, MRI processing can be accelerated to a greater extent. Some other languages or programming interfaces to support the parallel computing are OpenCL (Open Computing Library), DirectX Compute and FORTRAN. Some parallel models are available for parallel computation like bulk synchronous parallel model (BSP) and field programmable gate array (FPGA) [2] [3].

In this paper, we implement few commonly used image pre-processing algorithms and operations needed for MRI brain volume analysis pipeline using GPU programming in CUDA and estimate speedup in GPU and CPU. In CPU architecture, the algorithms are implemented in a sequential manner using C++ language. The parallelism in GPU is achieved by creating individual threads to a

pixel or a slice. We propose two parallel models and process all the threads simultaneously in a GPU. First one is Per-Pixel Threading (PPT) model and the second one is Per-Slice Threading (PST) model. This classification is based on the nature of information, local or global, required to process the image. Local processing involves only a single pixel and its neighbors. Global processing involves all the pixels in the image. The implementation of adaptive filter, anisotropic diffusion, bilateral filter and non-local means (NLM) filter falls under the first category, as these algorithms mainly deal with local neighborhood information of each pixel and thus independent. Here a thread for every pixel of the image is created to execute the algorithms with the neighbors or relative pixels. When PPT model is invoked it executes all the threads in parallel and produces the results for the image in a single run. K-Means segmentation and feature extraction falls under the second category. They need the global information from the entire pixels in a slice to produces the results. Here a thread is created for every slice of MRI volume to execute the algorithm. When this PST model is invoked, it processes all the slices in parallel by their respective threads and produces the result for the entire MRI volume in a single run. The speed performance of all algorithms using conventional processor CPU and GPU are computed respectively and compared.

The remaining part of this paper is organized as follows. Detailed survey of GPU implementations on various fields are given in the section II. Section III describes the feature of GPU-CUDA programming model. Section IV explains the implementation details of the algorithms in GPU. Section V discusses the results and section VI concludes the paper.

II. Related Works

Recently a huge research over the performance comparison among CPU and GPU for complex computer algorithms is actively going on. Ghorpade et al., gave an analysis on the architecture of CUDA and done a performance comparison on CPU and GPU [4]. Harish and Narayanan had implemented breadth first shortest path and single source shortest path algorithm in GPU [5]. Their result showed that GPU is 20-50 times faster than CPU. Yadav et al., implemented texture based similarity function using GPU [6]. They achieved the GPU 30 times faster than the CPU. Das repeatedly computed the square root of values in array N using both CPU and GPU [7]. It is showed speed up range from 56 to 197 times. Almazrooie et al. proposed a fast Fuzzy C Means algorithm using GPU [8]. The GPU based FCM tested in the brain simulated dataset to segment the brain tissues and achieved 245 fold speedup than the sequential processor. Hemert and Dickerson implemented CUDA based application which performs high-precision randomization tests using Monte Carlo sampling [9]. They performed experiments with large number of variables, such as microarrays, next generation sequencing read counts and chromatographical signals. They achieved 12X speedup in GPU than with CPU.

Nowadays, few more researches are going on testing the performance of these architectures over medical image analysis. In MRI processing, Somasundaram and Kalaiselvi showed that more time is required to implement an algorithm for automatic segmentation of brain [10]. Eklund et al., gave a valuable survey on implementing various medical image processing algorithms like filtering, interpolation, segmentation, registration, noise removal and reconstruction [11]. They stated that there still exists no freely available library for separable and non-separable convolution in 2D, 3D and 4D. They mention that any work is yet done on interpolation. Pratx et al., did a survey about GPU computing in medical physics like image reconstruction in CT and MRI images [12]. This article mentioned few GPU based image processing registration and segmentation. Smistad et al., presented a review about medical image segmentation on GPU [13]. They gave some basics of CUDA model and explain various segmentation methods like thresholding, region growing, watershed and active contours on GPU. Shams et al., discussed medical image registration process on a CPU and a GPU [14].

ELEKS software has developed for post processing of MRI images using GPU [15]. Additionally they used parallel imaging methods to reduce the scan time. Parallel imaging methods involve mathematical operation called singular value decomposition (SVD). SVD is an iterative algorithm and parallelization of a single iteration does not give any significant benefits, especially for smaller image sizes. Therefore, the only realistic way to accelerate this operation is to run SVD processing over all frames and coils in parallel. The developers solved this problem by porting SVD algorithm to GPU programming model and running it over all frames and coils in parallel. They achieved 155X speedup on GPU than with CPU.

In 2015, Jing et al., developed a fast parallel implementation of group independent component analysis (PGICA) for functional magnetic resonance imaging (fMRI) [16]. Their proposed work demonstrated the speed accuracy of their experiments. But they realized that the device memory constraints for large amounts of subject's data processing. Eklunda et al., implemented various preprocessing operations on fMRI like slice timing corrections, motion compensation and smoothing on GPU [17]. They described GPU implementation of two statistical approaches, the general linear model and canonical correlation analysis (CCA) for fMRI data. Zhu et al., implemented the perfusion image analysis on GPU in CT and MRI [18]. They reduced the processing time of 6 minutes in CPU to 65 seconds in GPU for CT images, and 35 minutes to 10 minutes in MRI images. The final speedup factors are 5.56X and 3.5X for CT and MR images respectively.

III. 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 developed by NVIDIA in late 2006. 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. CUDA programming model is specified in the document released by NVIDIA [19]. CUDA is an open source and extension of the C programming language.

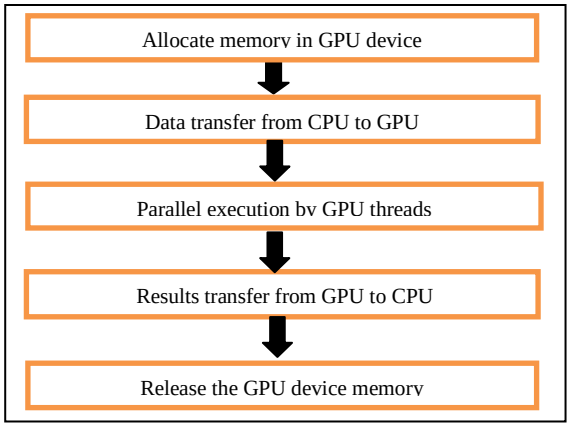

Fig.1. CUDA Execution Flow

The CUDA model supports collection of threads running in parallel. A kernel is a function or routine that executes on the GPU device [20]. A kernel 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. The execution flow of CUDA is a five step process see in Fig.1.

i) Allocate required memory in GPU device
ii) Transfer data from CPU to GPU
iii) Execute threads in parallel by GPU - CUDA
iv) Transfer results from GPU to CPU
v) Release the GPU device memory

IV. GPU Implementation of Medical Image Algorithms

CPU and GPU implementations are done using C++ and CUDA 7.5 respectively. Here we used single precision operations on both CPU and GPU implementation. In GPU implementation we used scalar based single instruction multiple threads (SIMT) model [20]. In SIMT, each thread has its own registers access those instructions that process different data simultaneously. One disadvantage of using a GPU coprocessor to accelerate computations is the cost of transferring data between main memory on the host system and the GPU's memory. This takes place over a PCI-Express bus, which having maximum transfer rate of 2GB/s, a factor of 87X less than the memory bandwidth of the on-board QUADRO GPU memory.

Image processing algorithms used for MRI volume analysis are parallel in nature. They can be implemented in local per-pixel or global per-slice operations. Hence we implemented the medical image algorithms in the following two models.

A. Per-Pixel Threading (PPT) Model
B. Per-Slice Threading (PST) Model

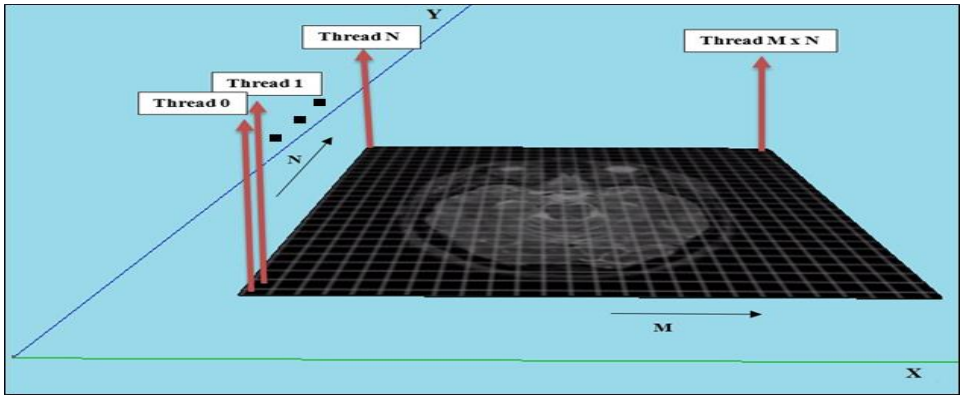

Fig.2. Per-Pixel Threading (PPT) model

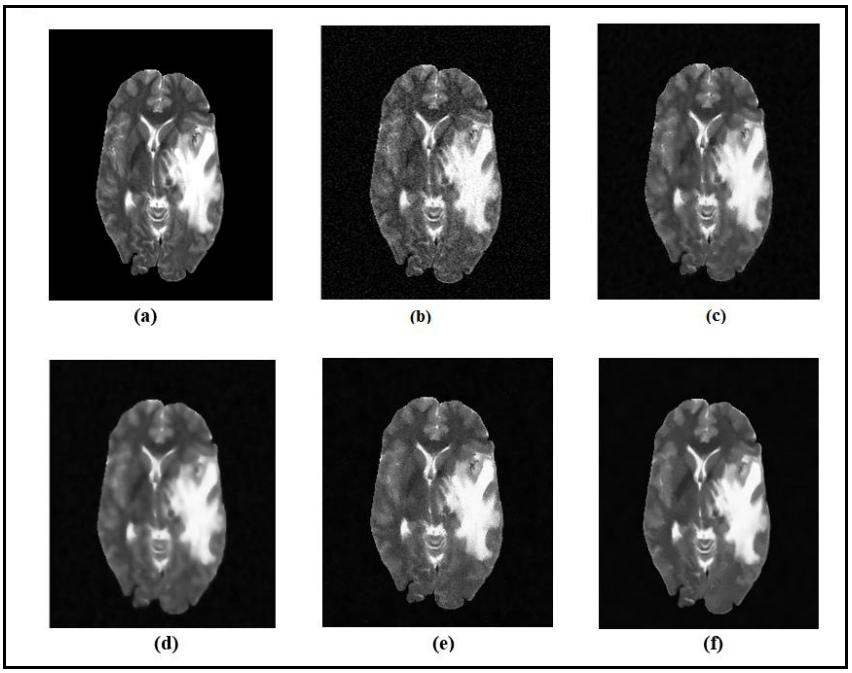

Fig.3. Results of image filtering techniques for MRI image (a) Original image, (b) Image with Gaussian noise (σ=0.005), (c) Adaptive filter, (d) Anisotropic Diffusion filter, (e) Bilateral filter, (f) Non-Local Means Filter

A. Per-Pixel Threading (PPT) Model

In PPT model, a thread is created for each pixel as shown in Fig. 2 and each thread have a unique thread ID. Each thread executes the algorithm in parallel and produces the results. We implemented few popular pre-processing filter algorithms for image enhancement. Image enhancement is used to improve the quality of the original image. Image filters are generally concentrate on a group of pixels in a neighborhood of each pixel. This process is known as convolution method. Here we take computationally challenging filters like adaptive filter, anisotropic diffusion filter, bilateral filter and non-local means filter for GPU implementation. All the arrays are defined in GPU is 1D pattern, which helps to calculate the number of CUDA blocks and threads. For the qualitative analysis, a sample slice was taken from a MRI dataset and Gaussian noise with σ =0.005 was introduced as shown in Fig. 3(a) and 3(b).

Adaptive Filter

Adaptive filter is a linear filter applied on a degraded image that contains noise. Adaptive filters are very effective in removing additive white noise, speckle noise and impulse noise. This filter depends on three statistical measures, mean and local variance with a defined M × N neighborhood region and a global variance on entire image. Adaptive filter done as [21]:

$$f(x,y)=g(x,y)-((\sigma_n)^2/(\sigma_L)^2)[g(x,y)-M_L] \quad (1)$$

where, $g(x, y)$ is the noisy pixel, $f(x, y)$ is the filtered pixel, M_L and σ_L^2 are local mean and variance computed for a window of size M×N and σ_n^2 is the global variance for the entire pixels in the image. For our experiment we have taken window of size 5×5. The filtered image using adaptive filter is shown in Fig. 3(c). The proposed CUDA implementation of Adaptive Filtering algorithm is given in the following Algorithm 1.

Algorithm 1: CUDA_ Adaptive_Filter

CUDA_Adaptive_Filter (NOI_IMAGE, DE_IMAGE, L_MEAN, L_VAR, W, ROW, COL)

Create a one dimensional intensity array NOI_IMAGE with the size of ROW × COL
Read the MRI noisy image pixel and store it into the NOI_IMAGE in row wise
Create zero valued arrays DE_IMAGE, L_MEAN and L_VAR with the size of ROW × COL pixels
Allocate GPU global memory for all parameters
Transfer all parameters from CPU to GPU memory
Create ROW × COL number of threads and call the kernel for parallel execution
for all ROW × COL number of threads do in parallel
tid=get thread ID
 if tid<ROW × COL then do
  Invoke code for Adaptive filter algorithm to each pixel.
  Reduction sum technique used for G_VAR calculation
  Filtered pixel value stored into DE_IMAGE
 end
end
Transfer DE_IMAGE from GPU global memory to CPU host memory

Anisotropic Diffusion Filter

Anisotropic diffusion filter is a technique introduced by Perona and Malik aimed at reducing image noise without removing significant parts of the image content, typically edges, lines or other details that are important for the interpretation of the image using non-linear partial differential equation [22]. Anisotropic diffusion is useful for edge detection, image smoothing, image segmentation and enhancement. The basic idea by Perona and Malik

was that diffusion is maximal within uniform regions and stopped across edges. The kernel used for diffusion is:

$$g(\nabla I) = \frac{1}{1+\left(\frac{\|\nabla I\|}{K}\right)^2} \quad (2)$$

Perona and Malik discretized their anisotropic diffusion as:

$$I_{t+1}(S) = I_t(S) + \frac{\lambda}{\eta_S}\sum_{p\in\eta_S} g\left(\left|\nabla I_{S,P}\right|\right)\nabla I_{S,P} \quad (3)$$

where, g is the conductance function, I is a the input noisy image, symbol ∇ which in the continuous form is used for gradient operator, K is the gradient threshold parameter, t denotes the iteration step, S denotes the pixel position in the discrete 2D grid, $\lambda \in (0,1)$ is rate of diffusion and η_S is 8 neighborhood of pixel S, η_S = {E, W, N, S, NE, SE, NW, SW}. Here it represents a scalar defined as the difference between neighboring pixels in each direction:

$$\nabla I_{S,P} = I_t(P) - I_t(S), P \in \eta_S = \{E, W, N, S, NE, SE, NW, SW\} \quad (4)$$

In our implementation, Eq.(2) is chosen for $g(x)$ with K=30, iterations t=8, integration constant $\frac{\lambda}{\eta_S}$ =1/8. The filtered image using anisotropic diffusion is shown in Fig. 3(d). The proposed CUDA implementation of Anisotropic Diffusion algorithm is given in the following Algorithm 2.

Algorithm 2: CUDA_ Anisotropic_Diffusion

CUDA_Anisotropic_Diffusion (NOI_IMAGE, DE_IMAGE, NUM_ITER, K, INT_CONS, ROW, COL)

```
Create a one dimensional intensity array NOI_IMAGE with the size of ROW × COL
Read the MRI noisy image and store it into the NOI_IMAGE in row wise
Create zero valued arrays DE_IMAGE with the size of ROW × COL pixels
Define NUM_ITER, K and INT_CONS values
Allocate GPU global memory for all parameters
Transfer all data from CPU to GPU memory
Create ROW × COL number of threads and call the kernel for parallel execution
  for all ROW × COL number of threads do in parallel
    tid=get thread ID
      if tid < ROW × COL then do
          Invoke code for diffusion algorithm to each pixel.
          Filtered pixel value stored into DE_IMAGE
       end
    end
Transfer DE_IMAGE from GPU device memory to CPU host memory
```

Bilateral Filter

Bilateral filter is initially presented by Tomasi and Manduchi in 1998 [23]. A bilateral filter is a non-linear, edge-preserving and noise-reducing filter. The bilateral filter takes a weighted sum of the pixels from the local neighborhood. The weights depend on both the spatial distance and the intensity distance. The spatial and intensity distances are done by the domain and range filter respectively. Bilateral filter is combination of both domain and range filters. Intensity difference between center and neighborhood pixels is calculated based on Euclidean distance. In this way, edges are preserved well while noise is averaged out. Mathematically, at a pixel location *x*, the output of a bilateral filter combined form of shift invariant domain filter with Gaussian range filter is given by [21]:

$$I(x) = \frac{1}{C}\sum_{y\in N(x)} e^{\frac{-\|y-x\|^2}{2\sigma_d^2}} e^{\frac{-|I(y)-I(x)|^2}{2\sigma_r^2}} I(y) \quad (5)$$

where, σ_d and σ_r are domain and range variance, $N(x)$ is a spatial neighborhood of pixel $I(x)$ and C is the normalization constant. In our implementation, we set the value of the parameters are $\sigma_d = 3, \sigma_r = 0.1$ and C = 1. The filtered image using bilateral filter is shown in Fig. 3(e). The proposed work of CUDA implementation of Bilateral filtering algorithm is given in the Algorithm 3.

Algorithm 3: CUDA_ Bilateral_Filter

CUDA_ Bilateral_Filter (NOI_IMAGE, DE_IMAGE, W, SIGMA_D, SIGMA_R, ROW, COL)

```
Create a one dimensional intensity array NOI_IMAGE with the size of ROW × COL
Read the MRI image and store it into the NOI_IMAGE in row wise
Create zero valued arrays DE_IMAGE with the size of ROW × COL pixels
Define W, SIGMA_D, and SIGMA_R
Allocate GPU global memory for all parameters
Transfer all variables from CPU to GPU memory
Create ROW × COL number of threads and call the kernel for parallel execution
for all ROW × COL number of threads do in parallel
tid=get thread ID
 if tid<ROW × COL then do
    Invoke code for bilateral filter algorithm to each pixel.
    Denoise image stored to DE_IMAGE
  end
end
Transfer DE_IMAGE from GPU global memory to CPU host memory
```

Non-Local Means Filter

Non-local means (NLM) is a nonlinear filter, takes a mean of all pixels in the image, weighted by how similar these pixels are to the target pixel [24]. Each pixel *p* of the NLM filtered image is computed with the following formula:

$$NL(p) = \sum w(p,q)v(q) \quad (6)$$

where, *v* is the noisy image, and weights *w(p, q)* meet the following conditions $0 \le w(p,q) \le 1$ and $\sum_q w(p,q)$.

The weights are based on the similarity between the pixels *p* and *q*. The Euclidean distance used to calculate similarity between the pixels p and q is given by:

$$d(p,q) = \left\| v(N_p) - v(N_q) \right\|_{2,F}^2 \quad (7)$$

where, $F > 0$ is a standard deviation of Gaussian kernel and N_k denotes a square neighborhood of fixed size and centered at a pixel *k*.

The weights are computed as:

$$w(p,q) = \frac{1}{Z(p)} e^{-\frac{d(p,q)}{h^2}} \quad (8)$$

$Z(p)$ is the normalizing constant defined as $Z(p) = \sum_q e^{-\frac{d(p,q)}{h^2}}$ and h is the weight-decay control parameter. We used parameter values h=20, F=2 and search window size 5 × 5 to check the similarity between p and q. The filtered image using NLM filter is shown in Fig. 3(f). The proposed of CUDA implementation of NLM algorithm is given in the Algorithm 4.

Algorithm 4: CUDA_ Non-Local_Means

CUDA_ Non-Local_Means (NOI_IMAGE, DE_IMAGE, H, F, W, ROW, COL)

Create one dimensional intensity array NOI_IMAGE and DE_IMAGE with the size of ROW × COL
Read the MRI image and store it into the NOI_IMAGE in row wise
Create zero valued array DE_IMAGE with the size of ROW × COL pixels
Allocate GPU global memory for all parameters
Transfer all parameters NOI_IMAGE, DE_IMAGE, H, F and W from host to device memory
Create ROW × COL number of threads and call the kernel for parallel execution
for all ROW × COL number of threads do in parallel
  tid=get thread ID
  if tid< ROW × COL then do
    Invoke code for NLM algorithm to each pixel.
    Denoise pixel value stored into DE_IMAGE
  end
end
Transfer resultant DE_IMAGE from GPU device memory to CPU host memory

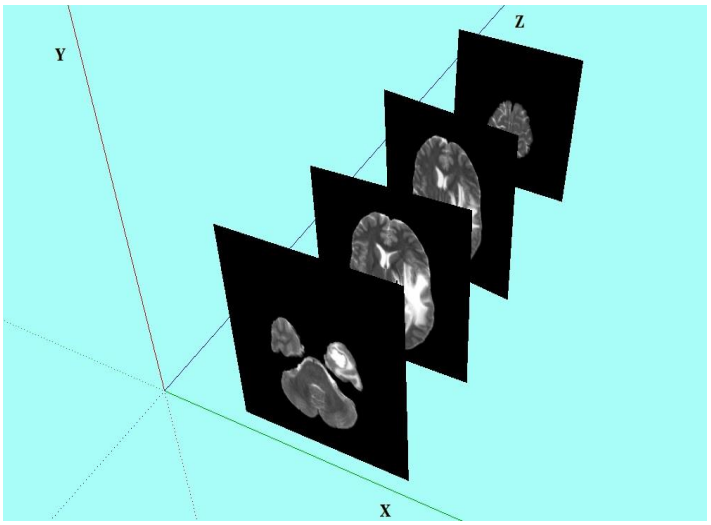

Fig.4. Per-Slice Threading model

B. Per-Slice Threading (PST) Model

In PST model, thread is created for each slice of the MRI brain volume and all thread is executed in parallel. The PST model is shown in Fig. 4. Each thread executes the algorithm in the kernel function. The kernel executes the algorithms by taking the pixels in serial manner. We implemented this model to K-Means for image segmentation and feature extraction for image classification.

Image Segmentation

Image segmentation is a process to split a digital image into meaningful multiple regions. Segmentation is mostly used to separate the object from background [25]. Segmentation techniques are broadly classified into two types: supervised and unsupervised. Supervised methods require the user interaction and are known as semi-automatic. Unsupervised techniques are completely automatic and segment the regions in feature space with a high density. The popular unsupervised classification techniques are K-Means (KM), Fuzzy C-Means (FCM) and Expectation - Maximization (EM) methods.

K- Means (KM) Algorithm

KM is one of the simplest unsupervised algorithms to classify a given data set through a certain number of clusters (assume k clusters) fixed a prior. KM is a hard segmentation procedure that generates a sharp classification [26]. It assigns each data either to a class or does not.

This algorithm consists of the following steps with a data set x_j, $i=1, 2, \ldots, n$.

Step 1: Initialize the centroids c_j, *j=1, 2, …., k.*

Step 2: Assign each data point to the group that has the closest centroids.

Step 3: When all points have been assigned, calculate the positions of the *k* centroids.

Step 4: Repeat Steps 2 and 3 until the centroids no longer move. This produces a separation of the data points into groups from which the metric to be minimized can be calculated.

This algorithm aims at minimizing an objective function, in this case a squared error function. The objective function is given by:

$$J = \sum_{j=1}^{k} \sum_{i=1}^{n} \left\| x_i^{(j)} - c_j \right\|^2 \quad (9)$$

where, $\left\| x_i^{(j)} - c_j \right\|^2$ is a measure of intensity distance between a data point x_j and the cluster centre c_j. For simplicity, the Euclidean distance is used as the dissimilarity measure. The proposed work of CUDA implementation of K-Means algorithm is given in the Algorithm 5.

Algorithm 5: CUDA_K-Means for 4 tissue segmentation of MRI Image

CUDA_K-MEANS (IN, S1, S2, S3, S4, ROW, COL, n)

```
Create a one dimensional intensity array IN with the size of n ×ROW × COL
Read n MRI images with ROW × COL pixels
Create zero valued arrays S1, S2, S3 and S4 with the size of ROW × COL pixels
Copy all pixel details in a size of n × ROW × COL to IN in row wise
Transfer the IN, S1, S2, S3 and S4 arrays from CPU to GPU device memory
Create n threads and call the kernel for parallel execution
for all n threads do in parallel
        tid = get thread ID
    if tid < n then do
      Invoke code for K-Means algorithm to each image.
      Four segments results are stored into S1, S2, S3, and S4.
    end
end
Transfer S1, S2, S3 and S4 from GPU device memory to CPU host memory
```

Image classification

Image classification is perhaps the most important part of digital image analysis. Image classification analyses the numerical properties of various image features and organizes data into categories. One of the famous image classification methods is feature extraction.

Feature Extraction

Feature extraction plays a major role in classification systems [27]. The image based classification system depends on certain features of images ranging from simple statistical properties to complex shape properties. We demonstrate the extraction of few simple statistical features for an image. Histogram of the image gives summary of the statistical information about the image. So first order statistical information of the image can be obtained using histogram of the image. Mean (μ) is the average value of the intensity of the image. Variance (σ^2) tells the intensity variation around the mean. Skewness (μ_3) is the measure which tells the symmetries of the histogram around the mean. Kurtosis (μ_4) is the flatness of the histogram. Energy (*E*) is defined based on a normalized histogram of the image. Energy shows how the gray levels are distributed. Uniformity of the histogram is represented by the entropy (*H*).

Probability density $p(i)$ can be obtained by dividing the value of intensity level histogram $h(i)$ with total number of pixels in the image.

$$p(i) = \frac{h(i)}{XY}, i = 0,1,\ldots,G \quad (10)$$

where, X is number of pixels in the horizontal spatial domain and Y is the number of pixels in the vertical spatial domain. G is the total gray level of an image. Following is the list of features obtained using histogram of the image.

Mean: $$\mu = \sum_{i=0}^{G-1} i\ p(i) \quad (11)$$

Variance: $$\sigma^2 = \sum_{i=0}^{G-1} (i-\mu)^2\ p(i) \quad (12)$$

Skewness: $$\mu_3 = \sigma^{-3} \sum_{i=0}^{G-1} (i-\mu)^3\ p(i) \quad (13)$$

Kurtosis: $$\mu_4 = \sigma^{-4} \sum_{i=0}^{G-1} (i-\mu)^4\ p(i) \quad (14)$$

Energy: $$E = \sum_{i=0}^{G-1} \left[p(i) \right]^2 \quad (15)$$

Entropy: $$H = -\sum_{i=0}^{G-1} p(i) \log_2 \left[p(i) \right] \quad (16)$$

The proposed work of CUDA implementation of feature extraction algorithm is given in the Algorithm 6.

Algorithm 6: CUDA_1Dim features

CUDA_1Dim _FEATURES (IN, OUT, n, ROW, COL)

```
Create a one dimensional intensity arrays IN and OUT with the size of
n ×ROW × COL
Read n MRI images with ROW × COL pixels
Copy all pixel details in a size of N × ROW × COL to IN in row wise
Transfer the IN and OUT arrays from host to device
Create n threads and call kernel for parallel execution
for all n images do in parallel
        tid = get thread ID
        if tid < n then do
            Calculations of the formulae for each image.
            Store six outputs to OUT array.
        end
 end
Transfer OUT array from GPU device memory to host memory
```

V. Result and Discussions

The image dataset used for the testing purpose were collected from whole brain atlas (WBA) maintained by Harward medical school. The configurations of CPU and GPU system used in our experiment are given in Table 1.

We carried out our experiments by executing the algorithms explained in the section IV. First we carried out PPT model given in Algorithms 1 to 4. The execution time taken by each filtering algorithm is recorded and is given in Table 2 for three different sizes of images 256 × 256, 512 × 512 and 1024 × 1024 pixels. The plot of image size versus execution time for PPT model is shown in Fig. 5. From the Table 2 and Fig. 5 we observe that the GPU is 3-338 times faster than CPU. Further it shows that the speed up in GPU increases as the image size increases.

The performance of adaptive filter showed in Fig. 5(a) shows very less speedup because this algorithm needs global variance of pixels in the entire image. This global variance is calculated by using sum reduction technique such as sum the adjacent thread value using thread ID. It is a reduction process and stopping criteria depends on image size. This increases the execution time in GPU when image size increases. Fig. 5(b) and 5(c) shows the performance of anisotropic diffusion and bilateral filter. The time taken by GPU is almost constant and by CPU it is increasing exponentially when the image size increases.

Table 1. Configurations of CPU and GPU Hardware

Features	CPU	GPU
Processors name Speed Count Number of cores	Intel - I5 2500 3.4 GHZ 1 4	NVIDIA Quadro K5000 1.4GHz 8 1536 (8 × 192)
Memory	4 GB	4GB
Threads	4	1024 per Block
Operating system	Windows 8 64 bit	Windows 8 64 bit
Programming language	C++	CUDA 6.5
Graphics clock	810MHz	706MHz
Memory bandwidth	21GB/s	173GB/s
Power consumption	95W	122W (Auxiliary power required)
Transistor count	1400 million	3540 million
Others	----	Compute capability Version 3.0 Memory clock 5.4GHZ Max grid dimension (2147483647, 65535, 65535) Max thread dimension (1024,1024,64) Register per block 49152

Table 2. Execution Time in PPT Model for Various Image Sizes

Algorithm	Execution Time in sec								
	Image size= 256 × 256			= 512 × 512			= 1024 × 1024		
	Processor		Speed up	Processor		Speed up	Processor		Speed up
	CPU	GPU		CPU	GPU		CPU	GPU	
Adaptive filter	0.0050	0.0016	3X	0.0190	0.0051	3.8X	0.0800	0.0152	5.2X
Anisotropic Diffusion	0.1720	0.0048	35.X	0.7020	0.0125	56X	3.0250	0.0480	62.7X
Bilateral filter	0.1330	0.0780	1.7X	0.5140	0.0940	5.5X	2.1570	0.1100	19.6X
Non Local means	1.9030	0.0072	264X	7.7200	0.0240	322X	31.2310	0.0924	338X

The NLM filter is very computationally challenging algorithm in CPU as value of the central pixel is calculated from its neighborhood pixels in a serial manner. In GPU, this computation is done simultaneously. Hence the performance of NLM algorithm in GPU gives very high speed up to 338X as shown in Fig 5(d). The

peak signal to noise ratio (PSNR) value of all four filters are given in Table 3. The PSNR values are computed between the original image and filtered image by each filtering algorithm. The NLM filter gives higher PSNR value than by other filtering techniques.

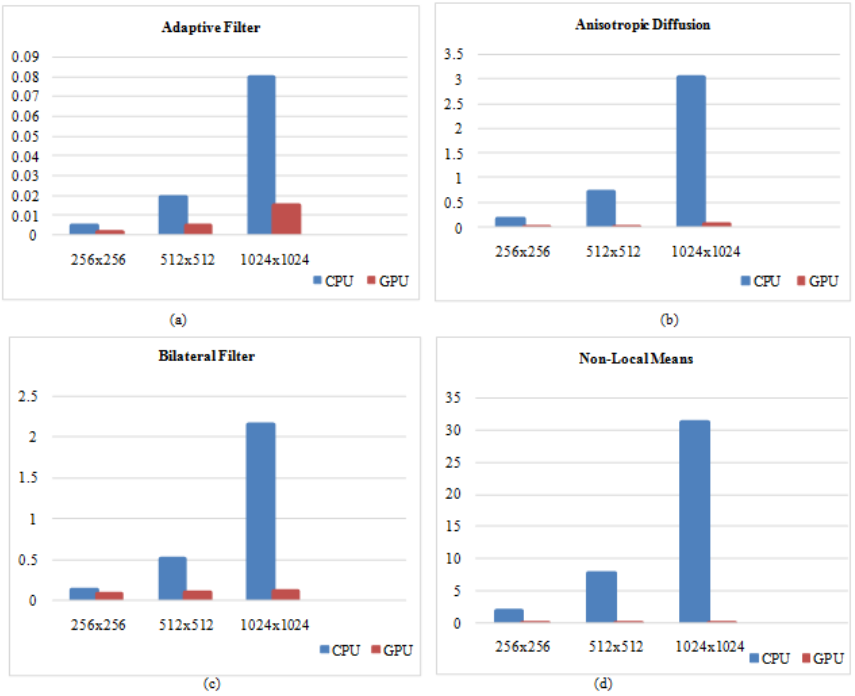

Fig.5. Results for PPT model a) Adaptive filtering, (b) Anisotropic diffusion, (c) Bilateral filtering, (d) Non-Local Means.

Table 3. PSNR Value of Sample MRI Brain Slice

Filter name	PSNR
Adaptive filter	28.90
Anisotropic diffusion	28.42
Bilateral filter	29.18
Non-local means	30.12

Table 4. Execution Time in PST Model for Various MRI Volume Sizes

Algorithm	Execution Time in sec																	
	No of slices= 1			= 10			= 20			= 50			= 100			= 180		
	Processor		Speed Up	Processor		Speed up	Processor		Speed up	Processor		Speed up	Processor		Speed up	Processor		Speed up
	CPU	GPU		CPU	GPU		CPU	GPU		CPU	GPU		CPU	GPU		CPU	GPU	
Segmentation	0.017	0.29	17X CPU	0.17	0.30	1.8X CPU	0.345	0.305	1.1X	0.898	0.321	2.8X	1.98	0.343	5.7X	3.252	0.365	8.9X
Feature Extraction	0.012	0.02	1.6X CPU	0.134	0.022	6X	0.268	0.024	11.2X	0.684	0.0289	23.7X	1.37	0.057	24.2X	2.488	0.083	30X

We then carried out experiments by implementing Algorithm 5 and 6 in PST model. The experiments were done with MRI volumes of size 1, 10, 20, 50, 100 and 180 slices. Table 4 shows the CPU and GPU performance on K-Means segmentation and feature extraction for various MRI volume sizes. The plot of number of slice per volume versus execution time for PST model is shown in Fig. 6. Table 4 shows that GPU performance is up to 30 times faster than CPU. But for a single image or volume with less slices, CPU yields better result than GPU as shown in Fig. 6(a) and 6(b). The time taken for transfer of data between CPU and GPU is an overhead in total computation time. When the slices are less, the overhead time is higher than the CPU. Hence the advantage of GPU cannot be realized. This shows that PST model of GPU is not suitable for parallel processing with less number of slices. PST model is well suited for large dataset.

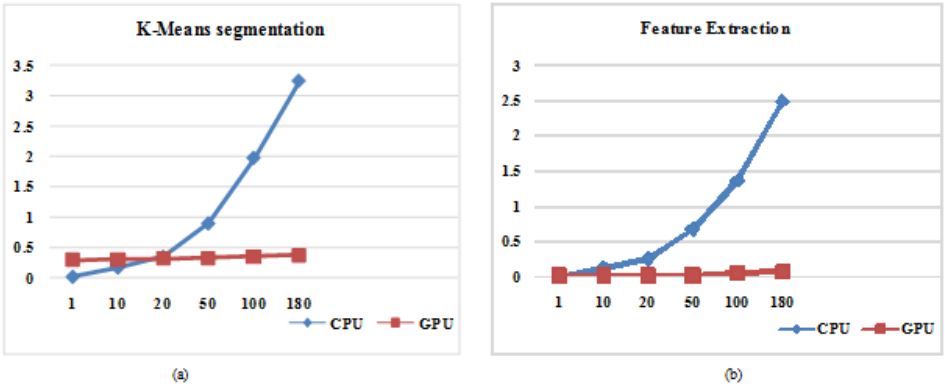

Fig.6. Results for PST model a) K-Means segmentation, b) Feature extraction

VI. Conclusion and Future Work

In this paper we have proposed per-pixel threading (PPT) for processing a slice and per-slice threading (PST) for an MRI volume that can be implement in a GPU. Using these models, we have implemented some of the general and advanced image pre-processing algorithms for accelerating the CAD systems in MRI volume analysis using GPU CUDA model. The adaptive filter, anisotropic diffusion, bilateral filter and non-local means filter depend on neighborhood information and thus implemented in PPT model. The algorithms like K-Means and Feature extraction are usually depend on entire slice information and thus implemented in PST model. The GPU based coding yielded speedup values in the range of 3-338 times compared to conventional processor CPU for PPT model and up to 30 times to PST model. Further work is under progress for classification of brain abnormality and tumor detection in MRI using GPU.

Acknowledgement

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, "An Introduction to GPU Computing and CUDA Architecture", Computing and CUDA Architecture, *NVIDIA Corporation*, vol. 6, no.5, 2011.

[2] A. Mohan and G. Remya, "A Review on Large Scale Graph Processing using Big Data Based Parallel Programming Models", *International Journal of Intelligent Systems and Applications*, vol. 9, no. 2, pp. 49-57, 2017.

[3] T. Praveen T and P. Arun Raj Kumar, "Multi-Objective Memetic Algorithm for FPGA Placement using Parallel Genetic Annealing", *International Journal of Intelligent Systems and Applications*, vol. 8, no. 4, pp. 60-66, 2016.

[4] J Ghorpade, J. Parande, M. Kulkarni and A. Bawaskar, "GPGPU Processing in CUDA Architecture", *Advance Computing: An International Journal*, vol. 3, no.1, pp.105-120, 2012.

[5] P. Harish and P. J. Narayanan, "Accelerating large graph algorithms on the GPU using CUDA", *Proceedings of the International Conference on High Performance Computing*, pp. 197-208, 2007.

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

[7] A. Das, "Process Time Comparison between GPU and CPU", *Tech. Report*, 2011.

[8] M. Almazrooie, R. Abdullah, and M. Vadiveloo, "GPU-Based Fuzzy C-Means Clustering Algorithm for Image Segmentation", CoRR, abs/1601.00072, 2016.

[9] J. L. Van Hemert and J. A. Dickerson, "Monte Carlo randomization tests for large-scale abundance datasets on the GPU", *Computer Methods and Programs in Biomedicine*, vol. 101, pp.80-86, 2011.

[10] K. Somasundaram and T. Kalaiselvi, "Automatic Brain Extraction Methods for T1 magnetic Resonance Images using Region Labelling and Morphological Operations", *Computers in Biology and Medicine*, vol. 41, no. 8, pp.716-725, 2011.

[11] A. Eklund, P. Dufort, D. Forsberg and S. M. LaConte, "Medical image processing on the GPU- Past, present and future", *Medical Image Analysis*, vol. 17, no.8, pp.1073-1094, 2013.

[12] G. Pratx and L. Xing, "GPU computing in medical physics: A review", *The International Journal on Medical Physics and Practice*, vol. 38, no. 5, pp. 2685-2697, 2011.

[13] E. Smistad, T. L. Falch, M. Bozorgi, A. C. Elster and F. Lindseth, "Medical image segmentation on GPUs – A comprehensive review", *Medical Image Analysis*, vol. 20, no. 1, pp. 1-18, 2015.

[14] R.Shams, P. Sadeghi, R. A. Kennedy and R. I. Hartley, "A Survey of Medical Image Registration on Multicore and the GPU", *IEEE Signal Processing Magazine*, vol. 27, no. 2, pp. 50-60, 2010.

[15] ELEKS, "CUDA-Accelerated Image Processing for Healthcare", http://www.eleks.com. Last accessed on 19th June 2016.

[16] Y. Jing, W. Zeng, N. Wang, T. Ren, Y. Shi, J. Yin and Q. Xu, "GPU-based parallel group ICA for functional magnetic resonance data", *Computer Methods and Programs in Biomedicine*, vol. 119, no. 1, pp.9-16, 2015.

[17] A. Eklunda, M. Andersson and H. Knutsson, "fMRI analysis on the GPU - Possibilities and challenges", *Computer Methods and Programs in Biomedicine*, vol. 105, vol.2, pp. 145-161, 2012.

[18] F. Zhu, D. R. Gonzalez, T. Carpenterb, M. Atkinsona and J. Wardlaw, "Parallel perfusion imaging processing using GPGPU", *Computer Methods and Programs in*

Biomedicine, vol. 108, pp. 1012-1021, 2012.
[19] CUDA C Programming Guide, Version 8.0, *Tech. report*, NVIDIA, June 2017.
[20] D. B. Kirk and W. W. Hwu, *Programming Massively Parallel Processor: A Hands-on Approach*, 3rd Ed., Elsevier, pp. 1-576, 2016.
[21] S.O. Haykin, *Adaptive Filter Theory*, 5th Ed. Prentice Hall, 2013.
[22] P. Perona and J. Malik, "Scale-Space and Edge Detection using Anisotropic Diffusion", *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 12, no. 7, pp. 629-639, 1990.
[23] C. Tomasi and R. Manduchi, "Bilateral filtering for gray and colour images", *Proceedings of IEEE International Conference on Computer Vision*, pp. 839-846, 1998.
[24] A. Buades, B. Coll and J. M. Morel, "A Non-Local Algorithm for Image Denoising", *Computer Vision and Pattern Recognition*, vol. 2, pp. 60-65, 2005.
[25] A. K. Jain, *Fundamentals of Digital Image Processing*, Prentice-Hall, Englewood Clis, New Jersey, 1989.
[26] M. Mam, G. Leena and N. S. Saxena, "Improved K-means Clustering based Distribution Planning on a Geographical Network", *International Journal of Intelligent Systems and Applications*, vol. 9, no. 4, pp. 69 – 75, 2017
[27] R. W. Conners and C. A. Harlow, "A Theoretical Comparison of Texture Algorithms", *IEEE Transaction on Pattern Analysis and Machine Intelligence*, vol. 2, pp. 204-222, 1980.

Authors' Profiles

T. Kalaiselvi is currently working as an Assistant Professor in Department of Computer Science and Applications, The 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 The 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.

Sriramakrishnan P. is a Research Scholar (Full-time) in the Department of Computer Science and Applications, Gandhigram Rural Institute - Deemed University, Dindigul, India. He received his Bachelor of Science (B.Sc.) degree in 2011 from Bharathidasan University, Trichy, Tamilnadu, India. He received Master of Computer Applications (M.C.A) degree in 2014 from The Gandhigram Rural Institute- Deemed University, Dindigul, Tamilnadu, India. He worked as Software Engineer in the Dhvani Research and Development Pvt. Ltd, Indian Institute of Technology Madras Research Park, Chennai during January 2014 – March 2015. He is currently pursuing Ph.D. degree in The 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.

Somasundaram K. 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. 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 also a member of IEEE USA.

How to cite this paper: T. Kalaiselvi, P. Sriramakrishnan, K. Somasundaram, "Performance of Medical Image Processing Algorithms Implemented in CUDA running on GPU based Machine", International Journal of Intelligent Systems and Applications(IJISA), Vol.10, No.1, pp.58-68, 2018. DOI: 10.5815/ijisa.2018.01.07

