ΠΑΝΕΠΙΣΤΗΜΙΟ ΘΕΣΣΑΛΙΑΣ
ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ ΜΕ ΕΦΑΡΜΟΓΕΣ
ΣΤΗ ΒΙΟΪΑΤΡΙΚΗ

ΠΤΥΧΙΑΚΗ ΕΡΓΑΣΙΑ: ΕΠΙΤΑΧΥΝΣΗ ΤΜΗΜΑΤΟΠΟΙΗΣΗΣ ΚΑΙ ΑΠΕΙΚΟΝΙΣΗΣ ΚΑΡΔΙΑΓΓΕΙΑΚΟΥ ΣΥΣΤΗΜΑΤΟΣ ΜΕΣΩ ΤΕΧΝΟΛΟΓΙΑΣ CUDA

ΝΙΚΟΛΑΟΣ ΒΕΡΒΕΡΗΣ

ΕΠΙΒΛΕΠΩΝ ΚΑΘΗΓΗΤΗΣ:

ΑΝΤΩΝΙΟΣ ΟΜ. ΑΛΕΤΡΑΣ

Contents

Introduction

There are several different types of medical imaging; Ultrasound, Computed Tomography (CT), nuclear imaging and Magnetic Resonance Imaging (MRI). L Wigstrom introduced three-dimensional Phase Contrast MRI (PC-MRI) with which one can gain three dimensional velocity data.

One of the most succesful ways to analyse such data and extract information is by manual segmentation. The downside of this option is that it is time consuming and operator dependent. Therefore, introducing automatic segmentation in images can be a huge assistance in analyzing medical data. Even an automated segmentation approach can require a lot of time.

The image analysis applications that are were used in this thesis visualize and analyze the cardiovascular tree in a three dimensional way by using flow properties in four dimensional phase contrast magnetic resonance imaging (4D PC-MRI) flow data. These applications are Segment and Fourflow. Segment analyzes and quantifies data from many different medical images and its range of tools include features like quantification of MRI flow and segmentation of the left ventricle. Fourflow is an open source software for quantification and visualization of 4D PC-MRI data that enables development of new quantitative analysis tools. Those applications' processes are time consuming and require a high computational power.

This is where CUDA is entailed. CUDA is a parallel programming C-like language that uses one or multiple graphics processing units of a machine. The idea is simple: if there is a job to be done, many workers can do it way faster than a single worker if they cooperate in an appropriate way. Likewise CUDA breaks down a complex job into smaller,easier tasks and distributes these tasks that have to be done to different threads or blocks of a GPU.

Therefore, the aim of this thesis is accelerating a high performance vessel segmentation algorithm that exploits the flow properties of 4D Phase Contrast MRI. Considering this purpose, accelerating performance of Fourflow through using Matlab Parallel Computing toolbox and CUDA C was attempted.

Background

Magnetic Resonance Imaging

Human body is consisted of about 75% water. Water includes one atom of oxygen and two atoms of hydrogen. Therefore, there is an abundance of hydrogen in the human body. Each hydrogen atom's nucleus is a positively charged proton and can be called as a spin (a small spinning magnet). This little magnet does not spin in a perfect alignment and there is certain angle at which the atom is spinning with respect to the axis of B_0 .

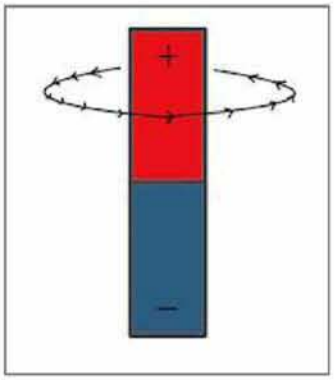


Figure 1: A spin

Spins reside in random positions and random angles inside the human body. When an MRI examination takes place, the patient is placed on a bed which is able to move inside a machine that generates an external magnetic field of multiple Tesla(ranging usually from 1.5 to 7). Higher Tesla values mean higher MRI scanner cost. When the patient is placed inside the static external field, the hydrogen spins allign with the direction of the external field in a parallel or anti parallel manner. This alignment results in a summation of the magnetic dipoles, into a net magnetization vector, M_0 that can create an image if it is handled in the right way.

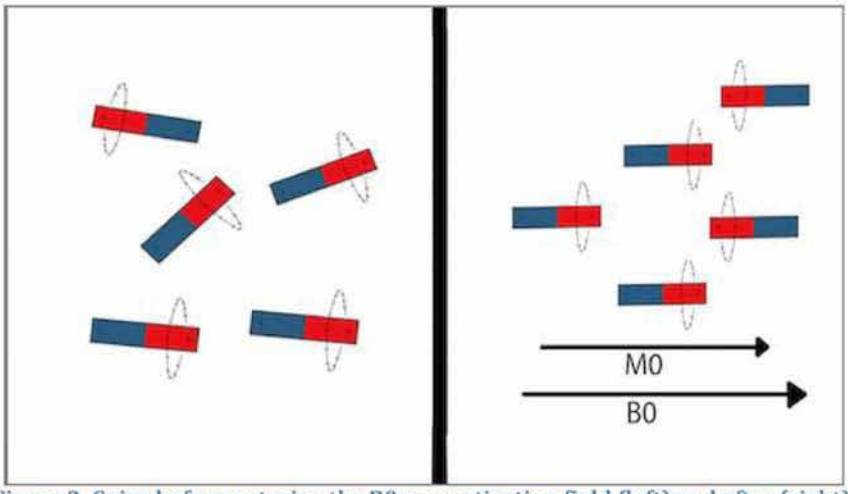


Figure 2: Spins before entering the B0 magnetization field (left) and after (right)

When spins are inside the magnetic field they precess with a frequency that is given by the Larmor equation:

$$f = \gamma_H B_0$$

where f stands for frequency, γ_H is for the gyromagnetic ratio of hydrogen and B_0 is the static magnetic field.

B_1 is an additional magnetic field of radiofrequency(RF) that is applied for a very short time on the x-y plane in order to rotate the magnetization vector M_0 90 degrees from its initial position(the z axis), so it is called a 90° pulse. In order for this "drop" to happen, B_1 has to rotate at the same frequency as the spins precess. Once the magnetization vector is on the x-y plane, the RF pulse is removed and M_0 rotates on the x-y plane and this way a MRI signal is generated. The spins at this time are affected by both of the fields. Their movement forms a spiral until it touches the x-y plane. This movement is being seen by someone who is using the Laboratory Frame of Reference.

If the observation could be done from the top of the the B_1 vector (somehow like the observer is sitting on the B_1 vector), then B_1 vector would appear still and the magnetization vector would be like it was decending until it drops completely on the x-y plane. This sight of view is called Rotating Frame of Reference point of view.

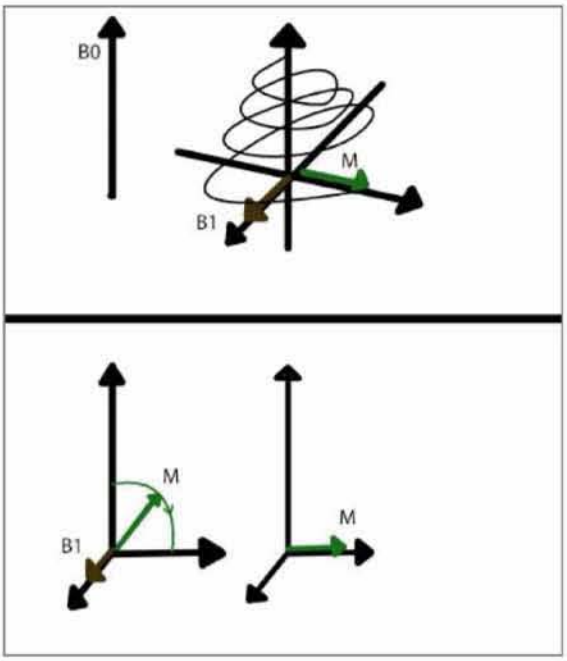


Figure 3: magnetization vector M drops to the x-y plane. Top: Laboratory Frame of Reference Bottom: Rotating Frame of Reference

When the magnetization M is on the x-y level, the Free Induction Decay signal is obtained. From the moment this signal gets created it starts to decrease until it becomes zero. This decrease of its value is because of the interactions of the spins that gain different rotational speeds and it is measured by the T2 reralaxation time. This relaxation time differs for each tissue. B_0 magnetic field is not completely homogenous cause of design imperfections and moreover disturbance to the field is added when the patient enters the machine. This is an additional reason for the signal to decay. The sum of these decay factors is represented by the T_2^* constant relaxation time which also depends on the body tissue. The magnetization on the x-y level M_{xy} is given by the following formula:

$$M_{xy}(t) = M_0 e^{-t/T_2^*}$$

where M_0 is the original magnetization on that starting latitude. The decrease which is caused by spin-spin interactions is calculated by the

following formula:

$$M_{xy}(t) = M_0 * e^{-t/T_2}$$

Silmutaneously with the signal decay, the magnetization vector M starts to grow along the x axis. This is because spins retrun the energy they have received to the lattice and it is called "spin-lattice" relaxation. The rebuilding of the magnetization on the z axis is given by the following formula:

$$M_z(t) = M_0 * (1 - e^{-t/T_1})$$

Spin-echo is the method used for the calculation of the T_2 value by isolating the decay reasons to spin-spin interaction only. This method starts when magnetizationvector lies on the y axis from the 90x pulse that was sent. In TE/2 time the spins dephase because of field heterogeneities and the magnetization starts to decay. On this point a $180^{\circ}x$ is applied andmakes the spins move towards the –y axis. After another TE/2 time the spins have been completely rebuilt to their initial status but this time they are placed in parallel to the –y axis. The result of this procedure is that now the dephasing that occurs to the magnetization is unaffected by field heterogeneities so the T_2 time constant can be calculated.

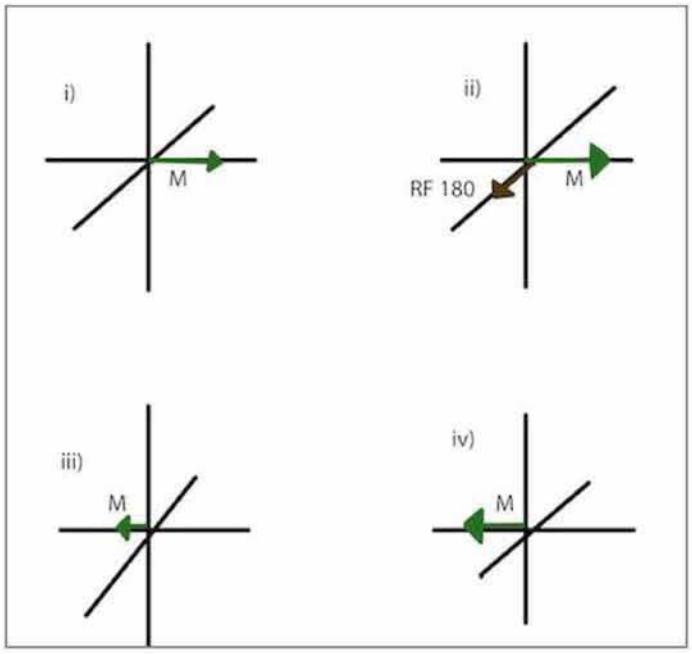


Figure 4: The spin-echo sequence

Gradients are additional changes in magnetic fields that are applied to the B_0 field and cause spins to change their frequency or direction on which they rotate based on their distance from the center of the magnet. When G_z gradient is applied, spins at the y axis precess at different frequencies based on their distance from the start of the y-axis. Since the strength of the gradient applied is known, if also the precessional frequency of a spin is also known, it is possible to calculate its position on the y-axis, as well as calculate the precessional frequency based on the position on the y-axis. Gradients can be applied in three axes, that is x, y and z. So the Larmor equation becomes:

$$f = \gamma_H (B_0 + G_X x + G_y y + G_z z)$$

where x,y and z are the distances from the center of the magnet.

When a G_z gradient is applied it only affects the rotational speeds of the spins on the z-axis. This way when a B_1 radiofrequency pulse is applied, spins of a certain distance from the z-axis'starting point can be forced to spin on the same frequency with the pulse and therefore only those spins will be forced to "drop" to the x-y plane and generate a signal. This is called "slice selection".

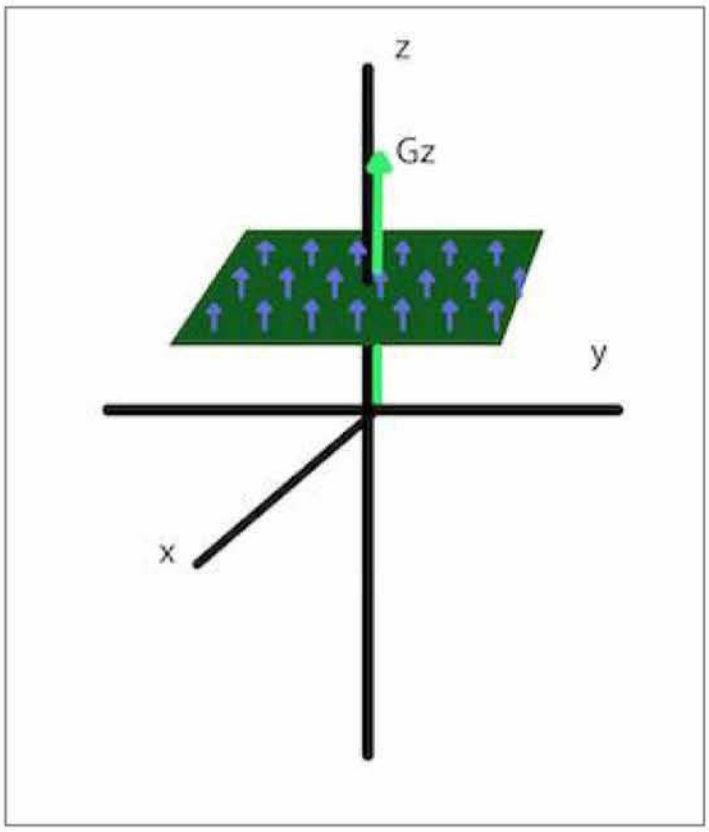


Figure 5: Slice selection process; a certain slice of spins is selected

G_x gradient encodes the frequency. That mean that when G_x is applied, spins at a specific distance on the x axis have exactly the same preciessional frequency and this frequency is different for each different location on the y axis. When the signal that contains these different frequencies is acquired, they can be retrieved by using the Fourier transform. Afterwards these frequencies can be translated with certain locations on the x-axis, since the precessional frequency and the x distance are linearly related as seen by the Larmor equation.

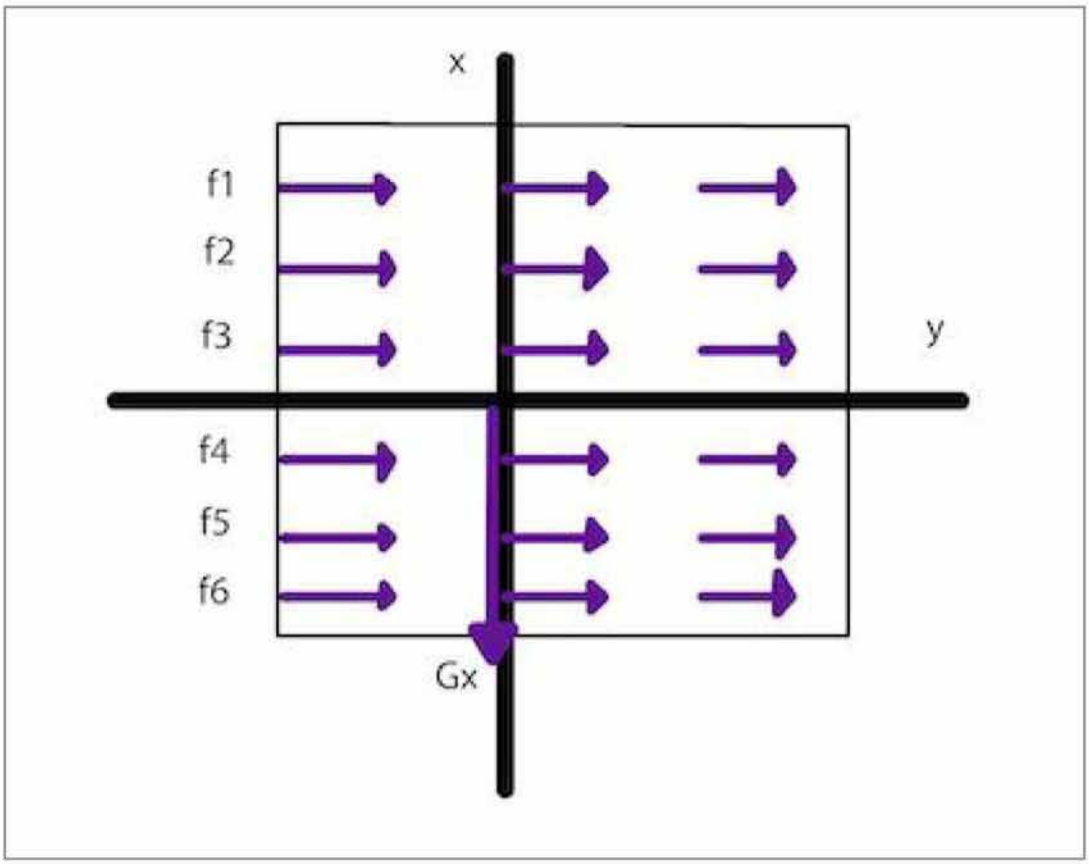


Figure 6: Gx applied, each row of magnetization vector has a different frequency value.

G_y gradient is applied as phase encoding in order to find the position of the spins on the y-axis. This happens exactly when RF pulse ends and before signal acquisition starts. While the G_y gradient application takes place the magnetization vectors on the y axis have different precessional frequencies. But this time, at the end of the G_y gradient pulse, every spin has the same precessional frequency but different phase

which depends from the distance from the center of the magnet. For every location on the y-axis, these different spin phases are summed. Fourier transform is not capable of extracting the original phases from these sums and it is not possible to identify the y location which they come from. This problem is solved by the repetition of the experiment several times with linearly increasing the strength of the G_y gradient. This way for every different experiment repetition there is a different "frequency" for every y location. Fourier transform can now be applied along the direction of each experiment, so that these frequencies can be told one from another and provide y position information.

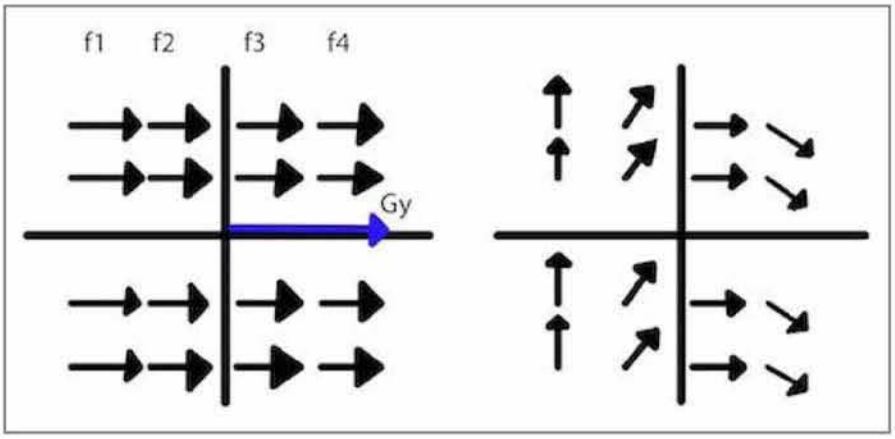


Figure 7: Gy gradient is applied when spins have different precessional frequencies .When it stops they have same frequencies but different phases

Information received by:[1] and [2]

Phase contrast MRI

Spins that are moving along the direction of a magnetic field gradient receive a phase shift φ.This change is proportional to the velocity of the tissue, u and creates a phase shift which is the loss of phase coherence in precessing spins.
Using this phase shift it is possible to construct an image detailing the velocity in any specified direction and slice. The phase of the signal taken from a single voxel is given by:

$$\varphi(r,T) = \gamma B_0 T + \gamma v \int_0^T G(r,t)t\,dt$$

$$= \gamma B_0 T + \gamma v \bar{G}$$

where γ is the gyromagnetic ratio, T is the time, B_0 is the external magnetic field and G(r,t) is the magnetic field gradient. In a phase contrast sequence two data sets with a different amount of flow sensitivity are acquired.

Then gradient pairs are applied that sequentially dephase and rephase continuously.By applying gradient pairs, two datasets with different phases dedending on how far the tissue has moved during the recording are acquired. By substracting these phases

$$\varphi 1 - \varphi 2 = \gamma \upsilon (\overline{G1} - \overline{G2})$$

gradient direction is attained. The gradient pairs are usually applied to produce velocity information in the x, y and z direction. This way the 3D velocity for each individual voxel in one slice is recorded simutaneously. The velocity is calculated with the following method; by comparing the phase of signals from each location in the two sequences the exact amount of motion induced phase change can be determined to have a map where pixel brightness is proportional to spatial velocity. As a result we get a multidiensional data set. For each different slice in this dataset there are three different phase images, each one corresponds to a gradient direction, while the amplitude of the image show the velocity. These images are time resolved, so there is a fourth dimension.

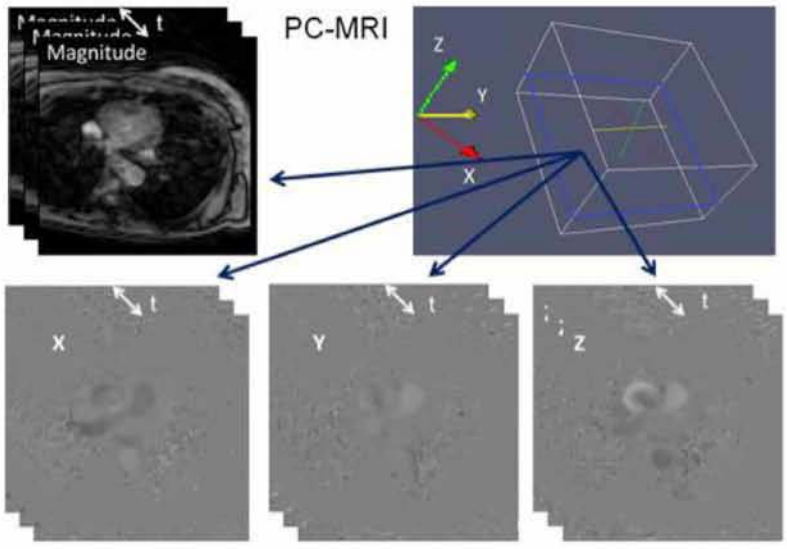


Figure 8: Data from PC-MRI scans and its components.
Source Ober P,(April 8, 2013) *Segmentation of cardiovascular tree in 4D from PC-MRI images*

Information received by:[3]

Blood vessel segmentation

Blood vessel segmentation is useful for analysing flow data. There are many different methods for segmentation, like level set functions, snakes and active contours but these are often limited by initialization parameters and are not completely automated, as well as they are limited to one or two blood vessels. By using steamlines, it is possible to visualize the flow, but not for the entire cardiovascular tree and also a lot of input is required.

A new vessel segmentation algorithm was introduced by Peter Oberg that benefited from PC-MRI images' velocity data. Through this algorithm the whole cardiovascular tree is portrayed.

The first step of this process is to remove the noise from the PC-MRI images.Local deviation is calculated for each pixel in each separate phase image. This way the erratic behaviour of noise is being identified and isolated. This process is repeated for each pixel.

Velocity is calculated as a product of the three different phase images G_1, G_2, G_3 in a vector form u= ($u_{G_1}, u_{G_2}, u_{G_3}$). Magnitude of the velocity is |u|= $\sqrt{u^2_{G_1} + u^2_{G_2} + u^2_{G_3}}$. Magnitude and direction of this velocity makes us able to separate what is coherent flow and what is not. Noise can be confused for coherent flow but generally the vector field in these areas shows larger angular speed and less coherent magnitude. So, coherent flow will have a more uniform vector field in both direction and magnitude than tissue and noise in the image, therefore three different coherence measures were tested.

Angular spread is one of these measures tested. First a 3x3x3 voxel neighbourhood is created around each voxel and the angle of each individual 3D vector is compared with the median angle of the 3D vector neighbourhood. The angles seen in figure 7 are calculated by taking the arcustangens of the x, y and z directions. The angular spread is defined by:

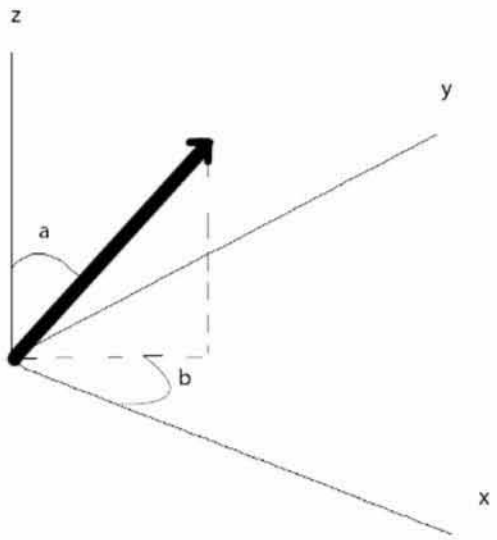


Figure 9: a and b angles

$$\vec{V}_g C = H * \vec{V}_g$$

$$\vec{V}_C = (\vec{u}_{G1C}, \vec{u}_{G2C}, \vec{u}_{G3C})$$

$$p_{ang}(x,y,t) = \frac{1}{n}\sum_{i=1}^{n} |ang(\alpha,\beta)|$$

Where H is a two dimensional filter, * denotes the convolution, v_C is the filter output for all three directions, n is the number of pixels in the neighbourhood, and ang is defined as the angles for each 3D vector within the neighbourhood as presented by Figure 7. This process is repeated for every pixel and every slice in the whole dataset. The normalized sum for systole of this neighbourhood around each voxel indicates whether or not the voxel is part of the coherent blood flow. The need of matching with the other features where a larger value means a larger probability for coherent flow the values of p_{ang} where modified in the following way:

$$p_{ang2}(x,y,t) = | p_{ang} - \sqrt{2}\pi |$$

Structure tensor is next measure tested. It is a matix and its eigenvalues indicate what direction of flow within a volume is most prominent. M_u is a positive semi definite matrix as defined by:

$$M_u = uu^T = \begin{pmatrix} u_{G_1} \\ u_{G_2} \\ u_{G_3} \end{pmatrix} [\, u_{G_1} u_{G_2} u_{G_3} \,] = \begin{pmatrix} u^2_{G_1} & u_{G_1}u_{G_2} & u_{G_1}u_{G_3} \\ u_{G_1}u_{G_2} & u^2_{G_2} & u_{G_2}u_{G_3} \\ u_{G_1}u_{G_3} & u_{G_2}u_{G_3} & u^2_{G_3} \end{pmatrix}$$

$$\hat{M}_u = H * M_u$$

And then M_u is convoluted with an average filter. Its convolution result has 3 eigenvalues; λ_1, λ_2, λ_3 one for each direction in space. The size of each eigenvalue is directly related to the amount of flow in the direction of its eigenvector. If all flow would flow in the same direction then there would only be one large eigenvalue $\lambda_1=1$ and its eigenvector would be in the direction of the flow.

In the case of no dominant direction in the neighbourhood, the value of all eigenvalues will be low, about 0.4. But if there there is coherent flow in the neighbourhood, a single direction should be more dominant than the others. This causes one eigenvalue to be much higher than the others. As a result voxels that have coherent flow, will have eigenvalues much greater than voxels with incoherent flow. Afterwards the values for these eigenvalues are averaged during systole for each pixel within the slice. This process is repeated for each pixel for each slice. A larger value of the feature indicates coherent flow.

$$\hat{p}_{vcs} = \frac{1}{n}\sum_{i=1}^{n} \max(eig(\hat{M}_u)) \quad \hat{p}_{vcs} = \frac{1}{n}\sum_{i=1}^{n} \max(eig(\hat{M}_u))$$

Projected velocity magnitude is the last but most important feature that uses a combination of angle and magnitude. When all 3D vectors within the neighbourhood project on the median vector of the same neighbourhood an expression for projected velocity magnitude $p_{proj}(x,y,t)$, is achieved which is an indication of coherent flow.

$$\vec{V}_{gC} = H * \vec{V}_g$$

$$\vec{V}_C = (\vec{u}_{G1C}, \vec{u}_{G2C}, \vec{u}_{G3C})$$

$$p_{proj(x,y,t)} = \frac{1}{n}\sum_{i=1}^{n} \vec{v}_i \vec{V}_C$$

H is the two dimensional mean filter, * denotes the convolution, v_{Cn} is the filter output for all three gradient directions, n is the number of pixels in the neighbourhood, v_i is the a vector in the neighbourhood, v_c is the median 3D vector and $v_i * v_c$ is the scalar product of the two vectors. Exactly like all the other measures,

the process is repeated for every pixel and averaged over systole. If the vectors within the neighbourhood are aligned in a similar direction, they will all project much of their magnitude or they will project poorly, in case they are aligned in different directions, onto the median vector direction. During systole the magnitude of the coherent flow within the vessels is large compared to its surrounding which further increases the effectiveness of the feature.

Throughout the previous steps, the product of the projected velocity magnitude feature and the actual magnitude image is used to finalize a probability map of the cardiovascular tree. It runs through all the slices of the input dataset and returns a new dataset.

Information received by:[3]

CUDA

The start of graphics processing unit (GPU) computing resides on a very simple but clever idea. GPUs in the early 2000s were built this way to produce a color for every pixel on the screen using programmable arithmetic units known as pixel shaders. What a pixel shader does is using its (x,y) position on the screen as well as some additional information like colors or texture coordinates to produce a final color. But because the whole process that was performed on the input colors and textures was completely controlled by the programmer, researchers found out that input colors could actually be any data. So if each color represents a number then calculations could be encoded into colors and their results could be as well translated from colours back to numbers. This way programmers could "trick" GPU into doing whatever calculation they wanted to do. But this model proved to be a bit too hard to handle for the widest part of the programming world, limitations on where the programmer could write to memory, strange usage of floating-point data and lack of debugging methods were only some of the problems that had to be faced.

The first GPU built with CUDA Architecture is GeForce 8800 GTX (http://www.geforce.com/hardware/desktop-gpus/geforce-8800-gtx/specifications). Cuda architecture aims to solve many of the programming issues that were faced initially by the first GPUs. This time GPU does not compute through pixel shaders but includes a unified shader pipeline that allows every arithmetic logic unit (ALU) on the chip to be handled by a program intending to perform general-purpose computations. Also single-precision floating-point arithmetic is now in and this time it can be used for general computation, not only for graphics. Moreover GPU could read and write to memory and access likewise a software-managed cache known as shared memory.

CUDA starts being used in 2007 assisting a wide variety of applications like medical imaging, computational fluid dynamics and enviromental science.

CUDA offers the opportunity of executing code on the GPU device that is currently selected instead of executing it on the CPU.

The following code includes a call to a function named kernel that involves angle brackets. Also the function has the identifier __global__ . This means that function kernel is being called and it will be executed on the GPU device that is currently selected instead of the CPU.

```
#include <iostream>
__global__ void kernel()
{

}
int main()
{
        kernel<<<1,1>>>();
        printf( "Hello, World!\n" );
        return 0;
}
```

What makes CUDA a parallel programming language is the option it gives to programmers to split a big task into smaller pieces and distribute the pieces to a GPU's "workers". These "workers" are called blocks and threads. Blocks are something like a grid. This grid can be one, two or three dimensional.

Block 00	Block01	Block02
Block10	Block11	Block12
Block20	Block21	Block22

Example of 2-dimensional Block Grid

Each block contains a grid of threads. Exactly like blocks, threads can also be one, two or three dimensional. Block00's threads would look like this:

Block00

Thread00	Thread01	Thread02
Thread10	Thread11	Thread12
Thread20	Thread21	Thread22

Let's examine a simple example to show how CUDA works and what can someone achieve by exploring the power of parallel computing. Addition of two vectors is the case.

A summary of this CUDA code is:
Two vectors(a and b) are created on the CPU and they are filled. Their values are copied to arrays that reside on GPU memory. A call to a function that runs in parallel on the GPU is made and the result is returned to the host. Finally the results are displayed and the memory that was used on the GPU is freed.

In a classic C approach code would look like this:

```
#define N 10
void add(int *a,int *b,int *c)
{
int spot=0;
        while(spot<N)
        {
                c[spot]=a[spot]+b[spot];
                spot++;
        }
}

int main()
{
int a[N],b[N],c[N];
         for(int i=0;i<N;i++)
         {
                a[i]=-i;
                b[i]=i*i;
         }

add(a,b,c);

for (i=0;i<N;i++)
{
  printf( "%d + %d = %d\n", a[i], b[i], c[i] );
}
}
```

CUDA C code to perform the same task is the following:

```
#define N 10

__global__ void add(int *a,int *b,int *c)
{
int tid=blockIdx.x;
            if (tid<N)
```

```
                c[tid]=a[tid] + b[tid];
}

int main()
{

int a[N],b[N],c[N];
int *dev_a,*dev_b,*dev_c;
cudaMalloc( (void**)&dev_a, N*sizeof(int));
cudaMalloc( (void**)&dev_b, N*sizeof(int));
cudaMalloc( (void**)&dev_c, N*sizeof(int));
for (int i=0;i<N;i++)
{
        a[i]=-i;
        b[i]=i*i;
}
cudaMemcpy (dev_a, a, N*sizeof(int), cudaMemcpyHostToDevice);
cudaMemcpy (dev_b, b, N*sizeof(int), cudaMemcpyHostToDevice);
add<<N,1>>(dev_a, dev_b, dev_c);
cudaMemcpy(c, dev_c, N* sizeof(int), cudaMemcpyDeviceToHost);
for(int i=0; i<N; i++)
{
      printf( "%d + %d = %d\n", a[i], b[i], c[i] );

}
cudaFree( dev_a );
cudaFree( dev_b );
cudaFree( dev_c);
}
```

The really interesting part of the code is:

```
add<<N,1>>(dev_a, dev_b, dev_c)
```

This is a call to a function named add. The call to the function is accompanied by brackets that include some parameters as well as the add function has a __global__ identifier. The combination of this identifier and these brackets means that the code that is written in the function will be executed on the GPU device instead of being executed on the CPU. The numbers inside the brackets refer to the number of blocks that will be launched and the number of threads that each block will include.

```
int tid=blockIdx.x;
```

blockIdx.x gives the index of the block that is currently launched. Since the grid of blocks is restricted to 10, it refers only to one dimension and in this case, this dimension is x(N could be representing more than just one number and it could for example be (10,4,2). In that case blockIdx.y and blockIdx.z would make sense and they would represent the indexing of these blocks on the y and z axis.).

So what's really happening is that there are 10 blocks running simultaneously and each one uses its index to compute the following code line:

```
int tid=blockIdx.x;
if (tid<N)
        c[tid]=a[tid] + b[tid];
```

In a figure it would look like this:

Block 1

```
__global__ void
add( int *a, int *b, int *c ) {
int tid = 0; if (tid < N)
c[tid] = a[tid] + b[tid];
}
```

Block 2

```
__global__ void
add( int *a, int *b, int *c ) {
int tid = 1; if (tid < N)
c[tid] = a[tid] + b[tid];
}
```

Block 3

```
__global__ void
add( int *a, int *b, int *c ) {
int tid = 2; if (tid < N)
c[tid] = a[tid] + b[tid];
}
```

Block 4

```
__global__ void
add( int *a, int *b, int *c ) {
int tid = 3; if (tid < N)
c[tid] = a[tid] + b[tid];
}
```

And this continues on the same way to Block 10.

The blocks are being executed in parallel so instead of computing C[10] array by using a while loop with 10 iterations, CUDA code launches 10 blocks where each one of them computes one spot in the C[10] array. If the assumption that the CPU being used and the GPU being used have the same computing power is made, then CUDA succeeds a 10x speedup.

Let's continue by explaining the main() code. The first line that needs explanation is

cudaMalloc((void**)&dev_a, N*sizeof(int)).

This works exactly like standard C malloc function with the difference that it does not allocate memory on the computer's memory but it does allocate memory from the GPU device that is currently selected. So there are three arrays that are being prepared to take some input data and their memory is allocated on the GPU.

The next line that would look unfamiliar to someone who is not used with CUDA C code is

cudaMemcpy (dev_a, a, N*sizeof(int), cudaMemcpyHostToDevice);

This function works like standard C memcpy function but the third parameter explains if the copy will be done from the GPU device to the host device (host device is called the CPU side of the machine being used) or if the copy will be done from the host device to the GPU (in this case the third parameter would be MemcpyDeviceToHost). In this particular scenario array a that resides on host memory is being copied into array dev_a that resides on device memory.

cudaFree(dev_a)

This is identical to C free function with the difference that cudaFree frees memory allocated on the GPU device and not on the host.

In this particular example threads were not used. Threads refer to a specific block, have their own indexing like blocks(threadIdx.x, threadIdx.y and threadIdx.z) and can share some common properties. For example threads within a block can use some shared memory. There are many comforts that CUDA provides to a programmer. Shared memory between threads, constant memory, texture memory, atomics and streams are only some of the benefits that CUDA offers.

Information gained by [4]

MATLAB Parallel Computing Toolbox

Parallel computing toolbox makes accelerating applications with GPUs an easy job. It gives the opportunity to any programmer to enjoy CUDA GPU parallel computing benefits without the need of in-depth knowledge of CUDA programming language or GPU architectures.

Combining the ease of MATLAB and the computing power of CUDA one can achieve great results from using this toolbox. When CUDA code is needed calls to kernels are made; kernels are one or more functions written in CUDA woven together in order to be used later. The following is a simple example:

```
k=parallel.gpu.CUDAKernel("simpleEx.ptx","simpleEx.cu");
```

where simpleEx.cu is:

```
__global__ void add ToVector(float *pi,float c,int vecLen)
int idx=blockIdx.x * blockDim.x+ threadIdx.x;
if (idx<vecLen)
{
    pi[idx]+=c;
}
```

and simpleEx.ptx is the result from compiling simpleEx.cu

The number of threads and blocks that will be launched on this kernel is set by the following commands:

```
k.GridSize=[5 1];
k.threadBlockSize=[5 1];
```

k now acts as a "call object" that someone can use in a feval function in order to produce results in a MATLAB code:

```
Out(25)=0;
Table(25)=0;
D_Table=gpuArray(single(Table));
Tablelength=25;
D_Tablelength=gpuArray(Tablelength);
A=8;
```

```
D_A=gpuArray(A);
[Out]=feval(k,D_Table, D_A, D_Tablelength);
```

Out keeps the result of this kernel call which is array Table with the value 8 added on each one of its initial values. As it can be easily figured out by the .cu file the calculations in the array's positions are done in parallel.

gpuArray is a function that saves an array of data on the GPU. Then this data can be used either for kernel calls or direct calculations.

As mentioned before, the kernel could include more than just one function. The only difference on declaring the kernel object then would be that it'd take a third parameter to explain to the system which function to execute(k=parallel.gpu.CUDAKernel(ptxfile, cufile , function_name)).

Methods

In this thesis, a previously developed method from Van Pelt et all and Peter Ober was improved by implementing GPU technology. The GPU implementation involved calculation of the propability map for vessel visualization based on 4D PC-MRI data. The core of this calculation resides on a MATLAB code.

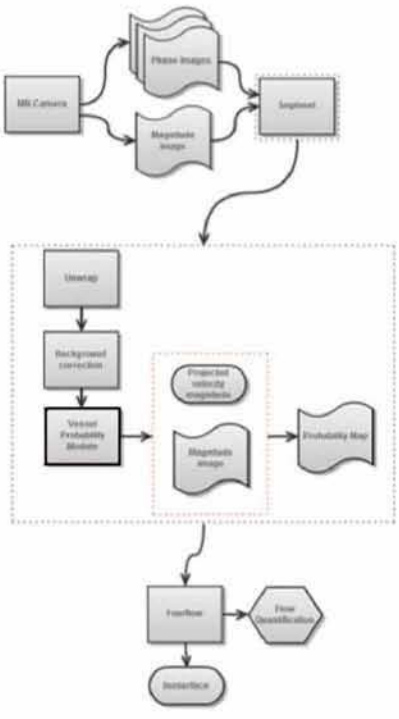


Figure 10:Schematic figure showing the data flow from start to finish. This thesis is being focused on accelerating the calulations of the Probability Map.
Source: Oberg P. (April 8, 2013) *Segmentation of cardiovascular tree in 4D from PC-MRI images*

In this MATLAB code three median images are created for each step on the t-plane(according to (x,y,t,s) coordinates where t comes from tloop and s from slice). Next, the vect3D table is being created which is the result of the stacked columns of the previously created median images. Following, a cscals table is being created that is the result of the multiplication of vect3D and median's tables' elements. The mean value of this cscals table is being added to a Magnitude Image table which sums up all the previous Magnitude Image values. Moreover, the temp table is being calculated that is the product of the normalized Magnitude Image table and Magnitude mean table which holds all the mean values of the third dimension (t) of each image. Finally, the newDataset table is created through repeating the absolute and normalized value of temp table.

```
for slice=1:slices
 magmean = mean(squeeze(SET(nom).IM(:,:,:,slice)),3);
 MagIM = zeros(SET(nom).YSize, SET(nom).XSize);

 for tloop = 1:tsize

  %Creating  medianimage for all directions:
  medianPhase = medfilt2(SET(NOphase).IM(:,:,tloop,slice), [neigm neign]);
  medianX = medfilt2(SET(NOphaseX).IM(:,:,tloop,slice), [neigm neign]);
  medianY = medfilt2(SET(NOphaseY).IM(:,:,tloop,slice), [neigm neign]);

  for X = (diffX+1):xsize2
   for Y = (diffY+1):ysize2

     vect3D = zeros(3, neigm*negn);
     vect3D(1,:) = colstack(squeeze(SET(NOphase).IM((Y-diffY):(Y+diffY), (X-
diffX):(X+diffX), tloop , slice)))'-0.5;
     vect3D(2,:) = colstack(squeeze(SET(NOphaseX).IM((Y-diffY):(Y+diffY), (X-
diffX):(X+diffX), tloop , slice)))'-0.5;
     vect3D(3,:) = colstack(squeeze(SET(NOphaseY).IM((Y-diffY):(Y+diffY), (X-
diffX):(X+diffX), tloop , slice)))'-0.5;

     cscals = [medianPhase(Y, X), medianX(Y, X), medianY(Y, X)] * vect3D;
     MagIM(Y,X) = MagIM(Y,X)+mean(cscals);

        end
  end
 end
 MagIM = MagIM/(max(MagIM(:)));
 temp = MagIM.*magmean;
 norm = abs(temp/max(temp(:)));
 newDataset(:,:,:,slice) = repmat(norm,[1 1 tsize]);
end
```

Two different computational systems were used mainly for this thesis. Both computational systems included Tesla C2075 GPUs. (http://www.nvidia.com/docs/IO/43395/NV-DS-Tesla-C2075.pdf)
Our algorithm was also tested and its execution times were calculated on two more computational systems including a GeForce770(http://www.geforce.com/hardware/desktop-gpus/geforce-gtx-770/specifications) and a Quadro K5000M(http://www.nvidia.com/object/quadro-k5000.html)

First Parallelization Approach

The first approach was to execute in parallel the two most inner loops of this code, that is X and Y for-loops. Considering this approach a CUDA kernel was launched for each of the slices*tsize steps of this procedure. In this kernel code the X and Y dimensions of the for-loops were replaced by thread and block indexing. This way instead of having X*Y loops running, now there are multiple blocks containing threads that execute in parallel that aim to calculate a final Magnitude Image element. The MATLAB code of this approach can be seen on Appendix I MATLAB code a). The cuda kernel code can be seen on Appendix II CUDA kernel code a).

Second Parallelization Approach

After collecting results from this approach, further improvement was attempted by further parallelization of the code. In this step, we added into the kernel call the third loop of the code (tloop). Now threads and blocks that are launched, calculate one slice of the last output Dataset that is being caclulated. The MATLAB code can be seen on Appendix I MATLAB code b). The CUDA kernel code can be seen on Appendix II CUDA Kernel code b).

Third Parallelization Approach

On the final approach the outer slice loop was distributed on multiple GPUs. This way two levels of parallelization are achieved. The first level derives from the parallelization that occurs from this GPU distribution implementation, while the second level derives from the parallelization that occurs from the CUDA kernel code that each GPU calls for each different slice of the dataset.

Moreover, a function was created containing all the above, in order to automate the whole process. The user can determine the number from the available GPU devices that he wants to utilize. A dataset is given to the function as input and the corresponding Propability Map and calculation time is the output. In this version of the code, the input datasets needed to create the output, were divided in smaller parts and then sent to the GPUs. The actual number of parts that each table is split to, comes from the available GPU devices that they will be assigned to. Through this process better speedup times are achieved because a lot of time is wasted on

transferring the data from and to the GPU.

In addition, a simple algorithm was implemented in order to determine for every different dataset, the number of threads and blocks that each kernel will launch. The available number of threads for each GPU is checked and then the largest possible number of threads is selected in order to obtain a certain blocks*threads combination that will result in a number equal to x*y*t elements. The MATLAB code can be seen in Appendix I MATLAB code c).The CUDA kernel code remains almost the same as the previous version with some minor changes.

This last parallelization approach was implemented but where were some serious drawbacks in using it for so small datasets as the ones that were used in this thesis. Dividing the iterations of the last outer loop on to multiple GPUs includes some calculation time overhead which is too high for these relatively small datasets. For example, the whole parallelization procedure for one dataset needed 1-1.5 seconds when the overhead for this procedure was more than a couple of seconds, making this method not acceptable to use. This means that we had to abandon this approach and return to a more simplified method. Therefore, we returned to the previous approach which included two levels of parallelization, but this time it is implemented in an automated away which could be used for any available dataset. Moreover we removed some unecessary transfers to the GPU that we figured out that existed in our code.The MATLAB code for this approach can be seen in Appendix I MATLAB code c). The changed kernel code without the redundant GPU transfers can be seen on Appendix II CUDA kernel code c).

Results

In this thesis we accelerated the MATLAB code of the computational core of Fourflow application's tool. We achieved speedup times up to 2500x decreasing the computational time from decades of minutes to a couple of seconds or even less. More precisely the CPU MATLAB code execution time and the corresponding parallel approach that were using GPU calculation times for each dataset used in this thesis are presented here. The results are being presented for the code execution on different computational systems that included different GPUs. These are a TeslaC2075, a Quadro K5000M and a GeForce 770.

First dataset: 80*80*40*52 Elements

Tesla C2075

CPU MATLAB Code computational time: ~1900 seconds
Parallelized approach computational time :~ 1.15 seconds
Speedup achieved~1650x

GeForce 770

CPU MATLAB Code computational time: ~2650 seconds
Parallelized approach computational time :~ 0.5 seconds
Speedup achieved:~ 5000x

Quadro K5000M

CPU MATLAB Code computational time:~1400 seconds
Parallelized approach computational time:~ 0.75 seconds
Speedup achieved:~ 5000x

Second dataset: 96*96*40*44 Elements

Tesla C2075

CPU MATLAB Code computational time: ~ 2300 seconds
Parallelized approach computational time: ~ 1.4 seconds
Speedup achieved:~1650x

GeForce 770

CPU MATLAB Code computational time: ~3000 seconds
Parallelized approach computational time :~ 0.6 seconds
Speedup achieved:~ 5000x

Quadro K5000M

CPU MATLAB Code computational time:~1700 seconds
Parallelized approach computational time:~ 0.9 seconds
Speedup achieved:~ 2000x

Third dataset: 144*144*40*44 Elements:

Tesla C2075

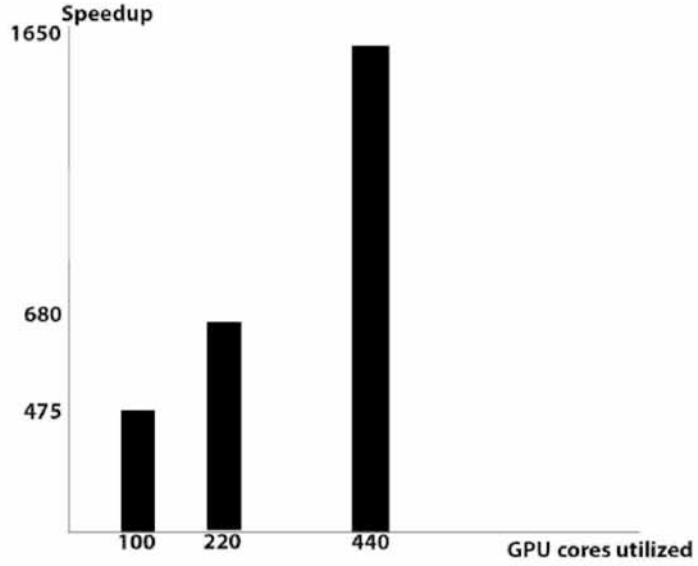
CPU MATLAB Code computational time: ~ 4900 seconds
Parallelized approach computational time: ~ 3 seconds
Speedup achieved:~1650x

GeForce 770

CPU MATLAB Code computational time: ~7500 seconds
Parallelized approach computational time :~ 1.5 seconds
Speedup achieved:~ 5000x

Quadro K5000M

CPU MATLAB Code computational time:~3900 seconds
Parallelized approach computational time:~ 1.9 seconds
Speedup achieved:~ 2000x



GPU Cores and execution time

The number of GPU cores that are utilized everytime a CUDA kernel is called derives from the number of threads and blocks that each kernel is specified to be launched with. In all the previous sections in this thesis the maximum number of GPU cores were used in order to achieve maximum speedup results. Computational times for for lower number of used GPU cores tested on a computational system with Tesla C2075 are being presented here:

First dataset:
Maximum possible number of threads and blocks (~440 GPU cores used)
Parallelized approach computational time: 1.15 seconds

Medium number of threads and blocks (~220 GPU cores used)
Parallelized approach computaitonal time: 2.80 seconds

Low number of threads and blocks (~100 GPU cores used)
Parallelized approach computational time: 4 seconds

Second dataset:
Maximum possible number of threads and blocks (~440 GPU cores used)
Parallelized approach computational time: 1.4 seconds

Medium number of threads and blocks (~220 GPU cores used)
Parallelized approach computational time: 3 seconds

Low number of threads and blocks (~100 GPU cores used)
Parallelized approach computational time: 4.4 seconds

Third dataset:

Maximum possible number of threads and blocks (~440 GPU cores used)
Parallelized approach computational time: 3 seconds

Medium number of threads and blocks (~220 GPU cores used)
Parallelized approach computational time: 6.5 seconds

Low number of threads and blocks (~100 GPU cores used)
Parallelized approach computational time: 12.8 seconds

MEX Code compared to CUDA Code

The main goal of this thesis was to accelerate the execution time of a MATLAB code. As it was explained in the speedup results, this acceleration was achieved by implementing calls to CUDA kernels from MATLAB code through MATLAB Parallel Computing Toolbox. Therefore speedup results were so high due to one more reason. MATLAB code hides some latencies compared to a C code. When turning MATLAB code into CUDA C code, the acceleration derives from two different levels. The first level is the acceleration from spreading the work that needs to be done on different threads on the GPU and the second level is the acceleration that occurs from calling C code instead of MATLAB code which is always faster. A MATLAB function is always more time consuming compared to a C function that performs the same tasks. MATLAB is not as low level programming language as C and offers some ease to use, but includes some overhead time in execution.

In order to present the speedup difference that would occur if C code was accelerated into CUDA C code, the implementation of a MEX file was needed. MEX functions are MATLAB functions that are used for caling C, C++ or Fortran files. A C file was created that executed the same task as the CUDA C code and was called through the MATLAB code. Now execution times to compare were from a MATLAB code that called a C function and a MATLAB code that scalled a CUDA function.

Speedup results that were accomplished on the computational system including a Tesla C2075 are the following:

First dataset

MATLAB code that calls MEX C code computational time: ~ 500 seconds
MATLAB code that calls CUDA C code computational time: ~1.15 seconds

Second Dataset

MATLAB code that calls MEX C code computational time: ~ 580 seconds

MATLAB code that calls CUDA C code computational time: ~ 1.4 seconds

Third Dataset

MATLAB code that calls MEX C code computational time:~ 1280 seconds
MATLAB code that calls CUDA C code computational time: ~3 seconds

Matlab code calling the MEX function and the correspoding MEX function can be seen in Appendix III MATLAB and MEX C code.

Discussion

In this thesis, we implemented CUDA code and MATLAB Parallel Toolbox in order to develop a high performance, fully automatic vessel segmentation algorithm that exploits the flow properties of 4D Phase Contrast MRI (PC-MRI) in the absence of gadolinium administration.

We succeeded in accelerating the performance of such an algorithm by having impressive speedup results, reducing the execution time of the initial algorithm in certain situations from more than half an hour to less than a second. Moreover the results that the GPU based code produced, in comparison with the results that the CPU based code produced never had a maximum error that was significant enough to be considered.

Based on the aforementioned results, we consider that now the algorithm is significantly more efficient and much closer to clinical use.

Future work

There are several ways that someone can use CUDA in order to achieve greater speedup results.

- The implementation of texture and shared memory and maybe even the design of a faster algorithm could lead into better acceleration of the initial MATLAB code.

- Furthermore, nVIDIA Profiler (https://developer.nvidia.com/nvidia-visual-profiler) was used in order to evaluate the performance of our CUDA kernel code algorithm. nVIDIA profiler is a toolkit that evaluates the performance of your code and suggests possible solutions for better results.The results show that there can be better performance if shared and texture memory are used.

- A more efficient and sophisticated way to calculate the threads and blocks for each thread can be implemented on the MATLAB scrip that calls the kernel since our implementation is rather simplistic.

Appendix I MATLAB code

a) First approach MATLAB code:

```
t = cputime;
load work.mat
k=parallel.gpu.CUDAKernel('loop.ptx','loop.cu');
k.GridSize=[threads 1 1];
k.ThreadBlockSize=[blocks 1 1];
MagIMR(6400)=0;
MagIMR=single(MagIMR);
D_NoPhase=gpuArray(reshape(NoPhase,1,13312000));
D_NoPhaseX=gpuArray(reshape(NoPhaseX,1,13312000));
D_NoPhaseY=gpuArray(reshape(NoPhaseY,1,13312000));

for slice=1:slices
 magmean = mean(I(:,:,:,slice),3);
 MagIMR(:)=0;
 for tloop = 1:tsize
  medianPhaseR = gpuArray(reshape(medfilt2(NoPhase(:,:,tloop,slice), [neigm
neign]),1,6400));
  medianXR = gpuArray(reshape(medfilt2(NoPhaseX(:,:,tloop,slice), [neigm
neign]),1,6400));
  medianYR = gpuArray(reshape(medfilt2(NoPhaseY(:,:,tloop,slice), [neigm
neign]),1,6400));

[MagIMR]=feval(k,MagIMR,D_NoPhase,D_NoPhaseX,D_NoPhaseY,diffX,diffY,sizex,siz
ey,tloop,slice,medianPhaseR,medianXR,medianYR);
 end
 MagIM=reshape((gather(MagIMR)),80,80);
 MagIM = MagIM/(max(MagIM(:)));
temp = MagIM.*magmean;
norm = abs(temp/max(temp(:)));
newDataset(:,:,:,slice) = repmat(norm,[1 1 tsize]);
end
e = cputime-t;
```

b)Second approach MATLAB code:

```
t=cputime;
load secondwork.mat;
k=parallel.gpu.CUDAKernel('completeKernel2A.ptx','completeKernel2A.cu');
k.GridSize=[threads];
k.ThreadBlockSize=[blocks];
MagIM=gpuArray(MagIM);
for slice=1:52;
magmean = mean(I(:,:,:,slice),3);
[MAG]=feval(k,MagIM,D_medianPhase,D_medianX,D_medianY,D_NoPhaseR,D_NoPh
aseXR,D_NoPhaseYR,slice,sizeX,sizeY,tsize,diffX);
MagIM2=reshape( gather(MAG),sizeX,sizeY);
MagIM2=double(MagIM2);
MagIM2 = MagIM2/(max(MagIM2(:)));
temp = MagIM2.*magmean
 norm = abs(temp/max(temp(:)))
 newDataset(:,:,:,slice) = repmat(norm,[1 1 tsize]);
 end
 q=cputime-t;
 disp(num2str(q));
```

c) Third approach MATLAB code:

```
function [data,total_time] = par3(setstruct,correction_value)
%Setting up the variables

I     = setstruct(1).IM;
sizeX  = setstruct(1).XSize;
sizeY  = setstruct(1).YSize;
tsize  = setstruct(1).TSize;
slices = setstruct(1).ZSize;

mediansize = sizeX*sizeY*tsize;

file = fopen('size.h','w');
fprintf(file,'int const mediansize=%d;',mediansize);
fclose(file);

system('nvcc -arch=sm_20 -ptx completeKernel3_1.cu');
disp('COMPILER CALLED')

k = parallel.gpu.CUDAKernel('completeKernel3_1.ptx','completeKernel3_1.cu');
```

```
diffX = 1;
D_NoPhaseR  = reshape(setstruct(2).IM,1,sizeX*sizeY*tsize*slices);
D_NoPhaseXR = reshape(setstruct(3).IM,1,sizeX*sizeY*tsize*slices);
D_NoPhaseYR = reshape(setstruct(4).IM,1,sizeX*sizeY*tsize*slices);
MagIM(sizeX*sizeY) = 0;
MagIM      = single(MagIM);
totalthreads = sizeX*sizeY*tsize;
threads    = k.MaxThreadsPerBlock;

result     = mod(totalthreads,threads);
while(result~=0)

   threads = threads-1;
   result  = mod(totalthreads,threads);

end

blocks     = totalthreads/threads;

k.GridSize         = blocks;
k.ThreadBlockSize   = threads;

disp(['Blocks: ',num2str(blocks),' - Threads: ',num2str(threads)])

D_NoPhaseR  = gpuArray(single(D_NoPhaseR));
D_NoPhaseXR = gpuArray(single(D_NoPhaseXR));
D_NoPhaseYR = gpuArray(single(D_NoPhaseYR));
MagIM      = gpuArray(single(MagIM));

total_kernel_time = 0;
gputime = tic;
for slice=1:slices

   magmean = mean(I(:,:,:,slice),3);

   % Kernel call
   kernel_time = tic;  [MAG]     = feval(k,MagIM,D_NoPhaseR,D_NoPhaseXR,...
      D_NoPhaseYR,uint32(slice),uint32(sizeX),uint32(sizeY),uint32(tsize),...
      uint32(diffX),single(correction_value));

   % See the last paragraph in this link
   % http://www.mathworks.se/support/solutions/en/data/1-
HSZ26C/?product=DM&solution=1-HSZ26C
   wait(gpuDevice);  % This work only for MATLAB version higher than 2012

   total_kernel_time = total_kernel_time + toc(kernel_time);
```

```
    % END Kernel call

    MagIM2    = reshape( gather(MAG),sizeX,sizeY);

    MagIM2    = MagIM2/max(((MagIM2(:))));
    temp      = MagIM2.*magmean;
    norm      = abs(temp/max(temp(:)));

    % replicate for all timeframes....
    newDataset(:,:,:,slice) = repmat(norm,[1 1 tsize]);

end
total_time = toc(gputime);
data = newDataset;

disp(['Kernel time (clean): ',num2str(total_kernel_time),'sec'])
```

Appendix II CUDA Kernel Code

a) First approach CUDA kernel code:

```
__global__ void function(float *MagIM,float *NoPhase,float *NoPhaseX,float
*NoPhaseY,int diffx,int diffy,int xsize2,int ysize2,int tloop,int slice,float
*medianPhase,float *medianX,float *medianY)
{
int i,j,ystart,xstart,xend,idx,idy,spot;
int yend,yrange,jlimit,starting_spot,counter,sizey;
float mean,sum,vect3D[27],cscals[9];

  sizey=ysize2+diffy;
  idx=blockIdx.x;
  idy=threadIdx.x;

  if(((idx>(diffx-1))&&(idx<xsize2))&&((idy>(diffy-1))&&(idy<ysize2)))
  {
    sum=0;

     xstart=idx-diffx;
    ystart=idy-diffy;
    xend=idx+diffx;
    yend=idy+diffy;
    yrange=yend-ystart;
    jlimit=xend-xstart;
    starting_spot=(slice-1)*80*80*40+(tloop-1)*80*80+(xstart)*80+ystart;
    counter=0;
    for(j=0;j<=jlimit;j++)
    {
        for (i=starting_spot+j*(sizey);i<=starting_spot+j*(sizey)+(yrange);i++)
        {
            vect3D[0*9+counter]=NoPhase[i]-0.50;
            vect3D[1*9+counter]=NoPhaseX[i]-0.50;
            vect3D[2*9+counter]=NoPhaseY[i]-0.50;
            counter++;
        }
    }
    spot=((slice-1)*80*80*40+(tloop-1)*80*80+(idx)*80+idy)%6400;
    counter=0;
   for(i=0;i<9;i++)
    {

cscals[i]=medianPhase[spot]*vect3D[i]+medianX[spot]*vect3D[i+9]+medianY[spot]
*vect3D[i+18];
```

```
        sum=sum+cscals[i];
      }
      mean=sum/9;
      MagIM[spot]=MagIM[spot]+mean;
    }
}
```

b) Second approach CUDA kernel code:

```
__global__ void function(float *MagIM,float *medianPhase,float *medianX,float
*medianY,float *NoPhase,float *NoPhaseX,float *NoPhaseY,int slice,int sizeX,int
sizeY,int tsize,int diffX)
{
  int id;
  int k,j,i,spot;
  float B1[9],B2[9],B3[9],swap,sum,mean,cscals[9];
  float vect3D[27];

  id=threadIdx.x + blockIdx.x * blockDim.x; //1000 blocks 256 threads
  id=id+(slice-1)*sizeX*sizeY*tsize; //ayta einai ta sizex sizey tsize

  if(id<sizeX*sizeY*tsize*slice)
  {
  spot=id%(sizeX*sizeY);
  if(spot%sizeX==0)
  {
    if(spot/sizeY==0)
    {
      B1[0]=0;
      B1[1]=0;
      B1[2]=0;
      B1[3]=0;
      B1[4]=0;
      B1[5]=NoPhase[id];
      B1[6]=NoPhase[id+sizeX];
      B1[7]=NoPhase[id+1];
      B1[8]=NoPhase[id+sizeX+1];

      B2[0]=0;
      B2[1]=0;
      B2[2]=0;
      B2[3]=0;
      B2[4]=0;
      B2[5]=NoPhaseX[id];
      B2[6]=NoPhaseX[id+sizeX];
      B2[7]=NoPhaseX[id+1];
```

```
        B2[8]=NoPhaseX[id+sizeX+1];

        B3[0]=0;
        B3[1]=0;
        B3[2]=0;
        B3[3]=0;
        B3[4]=0;
        B3[5]=NoPhaseY[id];
        B3[6]=NoPhaseY[id+sizeX];
        B3[7]=NoPhaseY[id+1];
        B3[8]=NoPhaseY[id+sizeX+1];
}
else if(spot/sizeY==sizeY-1)
    {
        B1[0]=0;
        B1[1]=0;
        B1[2]=0;
        B1[3]=0;
        B1[4]=0;
        B1[5]=NoPhase[id];
        B1[6]=NoPhase[id-sizeX];
        B1[7]=NoPhase[id-(sizeX-1)];
        B1[8]=NoPhase[id+1];

        B2[0]=0;
        B2[1]=0;
        B2[2]=0;
        B2[3]=0;
        B2[4]=0;
        B2[5]=NoPhaseX[id];
        B2[6]=NoPhaseX[id-sizeX];
        B2[7]=NoPhaseX[id-(sizeX-1)];
        B2[8]=NoPhaseX[id+1];

        B3[0]=0;
        B3[1]=0;
        B3[2]=0;
        B3[3]=0;
        B3[4]=0;
        B3[5]=NoPhaseY[id];
        B3[6]=NoPhaseY[id-sizeX];
        B3[7]=NoPhaseY[id-(sizeX-1)];
        B3[8]=NoPhaseY[id+1];
    }
    else
    {
```

```
        B1[0]=0;
        B1[1]=0;
        B1[2]=0;
        B1[3]=NoPhase[id-sizeX];
        B1[4]=NoPhase[id];
        B1[5]=NoPhase[id+sizeX];
        B1[6]=NoPhase[id-(sizeX-1)];
        B1[7]=NoPhase[id+1];
        B1[8]=NoPhase[id+(sizeX+1)];

        B2[0]=0;
        B2[1]=0;
        B2[2]=0;
        B2[3]=NoPhaseX[id-sizeX];
        B2[4]=NoPhaseX[id];
        B2[5]=NoPhaseX[id+sizeX];
        B2[6]=NoPhaseX[id-(sizeX-1)];
        B2[7]=NoPhaseX[id+1];
        B2[8]=NoPhaseX[id+sizeX+1];

        B3[0]=0;
        B3[1]=0;
        B3[2]=0;
        B3[3]=NoPhaseY[id-sizeX];
        B3[4]=NoPhaseY[id];
        B3[5]=NoPhaseY[id+sizeX];
        B3[6]=NoPhaseY[id-(sizeX-1)];
        B3[7]=NoPhaseY[id+1];
        B3[8]=NoPhaseY[id+(sizeX+1)];
}
   }
   else if(spot/sizeY==0)
   {
      if(spot%sizeX==(sizeX-1))
      {
        B1[0]=0;
        B1[1]=0;
        B1[2]=0;
        B1[3]=0;
        B1[4]=0;
        B1[5]=NoPhase[id];
        B1[6]=NoPhase[id-1];
        B1[7]=NoPhase[id+(sizeX-1)];
        B1[8]=NoPhase[id+sizeX];

        B2[0]=0;
```

```
        B2[1]=0;
        B2[2]=0;
        B2[3]=0;
        B2[4]=0;
        B2[5]=NoPhaseX[id];
        B2[6]=NoPhaseX[id-1];
        B2[7]=NoPhaseX[id+(sizeX-1)];
        B2[8]=NoPhaseX[id+sizeX];

        B3[0]=0;
        B3[1]=0;
        B3[2]=0;
        B3[3]=0;
        B3[4]=0;
        B3[5]=NoPhaseY[id];
        B3[6]=NoPhaseY[id-1];
        B3[7]=NoPhaseY[id+(sizeX-1)];
        B3[8]=NoPhaseY[id+sizeX];

}
    else
    {
        B1[0]=0;
        B1[1]=0;
        B1[2]=0;
        B1[3]=NoPhase[id];
        B1[4]=NoPhase[id-1];
        B1[5]=NoPhase[id+1];
        B1[6]=NoPhase[id+(sizeX-1)];
        B1[7]=NoPhase[id+sizeX];
        B1[8]=NoPhase[id+sizeX+1];


        B2[0]=0;
        B2[1]=0;
        B2[2]=0;
        B2[3]=NoPhaseX[id];
        B2[4]=NoPhaseX[id-1];
        B2[5]=NoPhaseX[id+1];
        B2[6]=NoPhaseX[id+(sizeX-1)];
        B2[7]=NoPhaseX[id+sizeX];
        B2[8]=NoPhaseX[id+sizeX+1];

        B3[0]=0;
        B3[1]=0;
        B3[2]=0;
```

```
      B3[3]=NoPhaseY[id];
      B3[4]=NoPhaseY[id-1];
      B3[5]=NoPhaseY[id+1];
      B3[6]=NoPhaseY[id+sizeX-1];
      B3[7]=NoPhaseY[id+sizeX];
      B3[8]=NoPhaseY[id+sizeX+1];
       }
}
else if(spot%sizeX==sizeX-1)
{
   if(spot/sizeY==sizeY-1)
   {
      B1[0]=0;
      B1[1]=0;
      B1[2]=0;
      B1[3]=0;
      B1[4]=0;
      B1[5]=NoPhase[id-1];
      B1[6]=NoPhase[id];
      B1[7]=NoPhase[id-sizeX];
      B1[8]=NoPhase[id-(sizeX+1)];

      B2[0]=0;
      B2[1]=0;
      B2[2]=0;
      B2[3]=0;
      B2[4]=0;
      B2[5]=NoPhaseX[id-1];
      B2[6]=NoPhaseX[id];
      B2[7]=NoPhaseX[id-sizeX];
      B2[8]=NoPhaseX[id-(sizeX+1)];

      B3[0]=0;
      B3[1]=0;
      B3[2]=0;
      B3[3]=0;
      B3[4]=0;
      B3[5]=NoPhaseY[id-1];
      B3[6]=NoPhaseY[id];
      B3[7]=NoPhaseY[id-sizeX];
      B3[8]=NoPhaseY[id-(sizeX+1)];
       }
   else
   {
      B1[0]=0;
      B1[1]=0;
```

```
      B1[2]=0;
      B1[3]=NoPhase[id-1];
      B1[4]=NoPhase[id-sizeX];
      B1[5]=NoPhase[id-(sizeX+1)];
      B1[6]=NoPhase[id+(sizeX-1)];
      B1[7]=NoPhase[id];
      B1[8]=NoPhase[id+sizeX];


      B2[0]=0;
      B2[1]=0;
      B2[2]=0;
      B2[3]=NoPhaseX[id-1];
      B2[4]=NoPhaseX[id-sizeX];
      B2[5]=NoPhaseX[id-(sizeX+1)];
      B2[6]=NoPhaseX[id+(sizeX-1)];
      B2[7]=NoPhaseX[id];
      B2[8]=NoPhaseX[id+sizeX];

      B3[0]=0;
      B3[1]=0;
      B3[2]=0;
      B3[3]=NoPhaseY[id-1];
      B3[4]=NoPhaseY[id-sizeX];
      B3[5]=NoPhaseY[id-(sizeX+1)];
      B3[6]=NoPhaseY[id+(sizeX-1)];
      B3[7]=NoPhaseY[id];
      B3[8]=NoPhaseY[id+sizeX];

   }
}
else if(spot/sizeY==sizeY-1)
{
   B1[0]=0;
   B1[1]=0;
   B1[2]=0;
   B1[3]=NoPhase[id-(sizeX+1)];
   B1[4]=NoPhase[id-1];
   B1[5]=NoPhase[id];
   B1[6]=NoPhase[id+1];
   B1[7]=NoPhase[id-(sizeX-1)];
   B1[8]=NoPhase[id-sizeX];

   B2[0]=0;
   B2[1]=0;
   B2[2]=0;
```

```
    B2[3]=NoPhaseX[id-(sizeX+1)];
    B2[4]=NoPhaseX[id-1];
    B2[5]=NoPhaseX[id];
    B2[6]=NoPhaseX[id+1];
    B2[7]=NoPhaseX[id-(sizeX-1)];
    B2[8]=NoPhaseX[id-sizeX];

    B3[0]=0;
    B3[1]=0;
    B3[2]=0;
    B3[3]=NoPhaseY[id-(sizeX+1)];
    B3[4]=NoPhaseY[id-1];
    B3[5]=NoPhaseY[id];
    B3[6]=NoPhaseY[id+1];
    B3[7]=NoPhaseY[id-(sizeX-1)];
    B3[8]=NoPhaseY[id-sizeX];
}
else
{
    B1[0]=NoPhase[id];
    B1[1]=NoPhase[id-1];
    B1[2]=NoPhase[id+1];
    B1[3]=NoPhase[id-(sizeX-1)];
    B1[4]=NoPhase[id-sizeX];
    B1[5]=NoPhase[id-(sizeX+1)];
    B1[6]=NoPhase[id+(sizeX-1)];
    B1[7]=NoPhase[id+sizeX];
    B1[8]=NoPhase[id+sizeX+1];

    B2[0]=NoPhaseX[id];
    B2[1]=NoPhaseX[id-1];
    B2[2]=NoPhaseX[id+1];
    B2[3]=NoPhaseX[id-(sizeX-1)];
    B2[4]=NoPhaseX[id-sizeX];
    B2[5]=NoPhaseX[id-(sizeX+1)];
    B2[6]=NoPhaseX[id+(sizeX-1)];
    B2[7]=NoPhaseX[id+sizeX];
    B2[8]=NoPhaseX[id+sizeX+1];

    B3[0]=NoPhaseY[id];
    B3[1]=NoPhaseY[id-1];
    B3[2]=NoPhaseY[id+1];
    B3[3]=NoPhaseY[id-(sizeX-1)];
    B3[4]=NoPhaseY[id-sizeX];
    B3[5]=NoPhaseY[id-(sizeX+1)];
    B3[6]=NoPhaseY[id+(sizeX-1)];
```

```
      B3[7]=NoPhaseY[id+sizeX];
      B3[8]=NoPhaseY[id+sizeX+1];

   }
for(k=0;k<8;k++)
   {
      for(j=0;j<8-k;j++)
      {
          if(B1[j]>B1[j+1])
        {
           swap=B1[j];
           B1[j]=B1[j+1];
           B1[j+1]=swap;
        }
        if(B2[j]>B2[j+1])
        {
           swap=B2[j];
           B2[j]=B2[j+1];
           B2[j+1]=swap;
        }
        if(B3[j]>B3[j+1])
        {
           swap=B3[j];
           B3[j]=B3[j+1];
           B3[j+1]=swap;
        }
       }
   }


   medianPhase[id]=B1[4];
   medianX[id]=B2[4];
   medianY[id]=B3[4];

   if( (spot%sizeX!=0) && (spot/sizeY!=0) && (spot%sizeX!=(sizeX-1)) &&
(spot/sizeY!=(sizeY-1)) )
   {
      vect3D[0]=NoPhase[id-(sizeX+1)]-0.5;
      vect3D[1]=NoPhaseX[id-(sizeX+1)]-0.5;
      vect3D[2]=NoPhaseY[id-(sizeX+1)]-0.5;

      vect3D[3]=NoPhase[id-sizeX]-0.5;
      vect3D[4]=NoPhaseX[id-sizeX]-0.5;
      vect3D[5]=NoPhaseY[id-sizeX]-0.5;

      vect3D[6]=NoPhase[id-(sizeX-1)]-0.5;
```

```
        vect3D[7]=NoPhaseX[id-(sizeX-1)]-0.5;
        vect3D[8]=NoPhaseY[id-(sizeX-1)]-0.5;

        vect3D[9]=NoPhase[id-1]-0.5;
        vect3D[10]=NoPhaseX[id-1]-0.5;
        vect3D[11]=NoPhaseY[id-1]-0.5;

        vect3D[12]=NoPhase[id]-0.5;
        vect3D[13]=NoPhaseX[id]-0.5;
        vect3D[14]=NoPhaseY[id]-0.5;

        vect3D[15]=NoPhase[id+1]-0.5;
        vect3D[16]=NoPhaseX[id+1]-0.5;
        vect3D[17]=NoPhaseY[id+1]-0.5;

        vect3D[18]=NoPhase[id+(sizeX-1)]-0.5;
        vect3D[19]=NoPhaseX[id+(sizeX-1)]-0.5;
        vect3D[20]=NoPhaseY[id+(sizeX-1)]-0.5;

        vect3D[21]=NoPhase[id+sizeX]-0.5;
        vect3D[22]=NoPhaseX[id+sizeX]-0.5;
        vect3D[23]=NoPhaseY[id+sizeX]-0.5;

        vect3D[24]=NoPhase[id+(sizeX+1)]-0.5;
        vect3D[25]=NoPhaseX[id+(sizeX+1)]-0.5;
        vect3D[26]=NoPhaseY[id+(sizeX+1)]-0.5;
cscals[0]=medianPhase[id]*vect3D[0]+medianX[id]*vect3D[1]+medianY[id]*vect3
D[2];

cscals[1]=medianPhase[id]*vect3D[3]+medianX[id]*vect3D[4]+medianY[id]*vect3
D[5];

cscals[2]=medianPhase[id]*vect3D[6]+medianX[id]*vect3D[7]+medianY[id]*vect3
D[8];

cscals[3]=medianPhase[id]*vect3D[9]+medianX[id]*vect3D[10]+medianY[id]*vect3
D[11];

cscals[4]=medianPhase[id]*vect3D[12]+medianX[id]*vect3D[13]+medianY[id]*vect
3D[14];

cscals[5]=medianPhase[id]*vect3D[15]+medianX[id]*vect3D[16]+medianY[id]*vect
3D[17];

cscals[6]=medianPhase[id]*vect3D[18]+medianX[id]*vect3D[19]+medianY[id]*vect
3[20];
```

```
cscals[7]=medianPhase[id]*vect3D[21]+medianX[id]*vect3D[22]+medianY[id]*vect
3D[23];

cscals[8]=medianPhase[id]*vect3D[24]+medianX[id]*vect3D[25]+medianY[id]*vect
3D[26];

    for(i=0;i<9;i++)
    sum=sum+cscals[i];

    mean=sum/9;

    atomicAdd(&(MagIM[spot]),mean);
}
  }
}
```

c) Changes that happened to last CUDA code version

```
#include "size.h"

__global__ void function(float *MagIM,float *NoPhase,float
*NoPhaseX,float *NoPhaseY,int slice,int sizeX,int sizeY,int
tsize,int diffX,float value)
{
    int id;
    int k,j,i,spot;
    float
B1[9],B2[9],B3[9],swap,sum,mean,cscals[9],medianPhase[mediansize]
,medianX[mediansize],medianY[mediansize];
    float vect3D[27];

    id=threadIdx.x + blockIdx.x * blockDim.x;
    id=id+(slice-1)*sizeX*sizeY*tsize;  // so that id is updated
about how many slices have passed

    sum=0;
    spot=id%(sizeX*sizeY); //masks the initial id into the right
coordinates

    if(spot%sizeX==0)
    {
        if(spot/sizeY==0)  //upper left corner
        {
            B1[0]=0;
            B1[1]=0;
            B1[2]=0;
            B1[3]=0;
            B1[4]=0;
            B1[5]=NoPhase[id];
            B1[6]=NoPhase[id+sizeX];
            B1[7]=NoPhase[id+1];
```

```
    B1[8]=NoPhase[id+sizeX+1];

    B2[0]=0;
    B2[1]=0;
    B2[2]=0;
    B2[3]=0;
    B2[4]=0;
    B2[5]=NoPhaseX[id];
    B2[6]=NoPhaseX[id+sizeX];
    B2[7]=NoPhaseX[id+1];
    B2[8]=NoPhaseX[id+sizeX+1];


    B3[0]=0;
    B3[1]=0;
    B3[2]=0;
    B3[3]=0;
    B3[4]=0;
    B3[5]=NoPhaseY[id];
    B3[6]=NoPhaseY[id+sizeX];
    B3[7]=NoPhaseY[id+1];
    B3[8]=NoPhaseY[id+sizeX+1];



}

else if(spot/sizeY==sizeY-1) //upper right corner
{
    B1[0]=0;
    B1[1]=0;
    B1[2]=0;
    B1[3]=0;
    B1[4]=0;
    B1[5]=NoPhase[id];
    B1[6]=NoPhase[id-sizeX];
    B1[7]=NoPhase[id-(sizeX-1)];
    B1[8]=NoPhase[id+1];

    B2[0]=0;
    B2[1]=0;
    B2[2]=0;
    B2[3]=0;
    B2[4]=0;
    B2[5]=NoPhaseX[id];
    B2[6]=NoPhaseX[id-sizeX];
    B2[7]=NoPhaseX[id-(sizeX-1)];
    B2[8]=NoPhaseX[id+1];

    B3[0]=0;
    B3[1]=0;
    B3[2]=0;
    B3[3]=0;
    B3[4]=0;
    B3[5]=NoPhaseY[id];
    B3[6]=NoPhaseY[id-sizeX];
    B3[7]=NoPhaseY[id-(sizeX-1)];
```

```
            B3[8]=NoPhaseY[id+1];


        }
        else   // any upper spot but not corners
        {
            B1[0]=0;
            B1[1]=0;
            B1[2]=0;
            B1[3]=NoPhase[id-sizeX];
            B1[4]=NoPhase[id];
            B1[5]=NoPhase[id+sizeX];
            B1[6]=NoPhase[id-(sizeX-1)];
            B1[7]=NoPhase[id+1];
            B1[8]=NoPhase[id+(sizeX+1)];

            B2[0]=0;
            B2[1]=0;
            B2[2]=0;
            B2[3]=NoPhaseX[id-sizeX];
            B2[4]=NoPhaseX[id];
            B2[5]=NoPhaseX[id+sizeX];
            B2[6]=NoPhaseX[id-(sizeX-1)];
            B2[7]=NoPhaseX[id+1];
            B2[8]=NoPhaseX[id+sizeX+1];

            B3[0]=0;
            B3[1]=0;
            B3[2]=0;
            B3[3]=NoPhaseY[id-sizeX];
            B3[4]=NoPhaseY[id];
            B3[5]=NoPhaseY[id+sizeX];
            B3[6]=NoPhaseY[id-(sizeX-1)];
            B3[7]=NoPhaseY[id+1];
            B3[8]=NoPhaseY[id+(sizeX+1)];


        }

    }
    else if(spot/sizeY==0)
    {
        if(spot%sizeX==(sizeX-1))  //lower left corner
        {
            B1[0]=0;
            B1[1]=0;
            B1[2]=0;
            B1[3]=0;
            B1[4]=0;
            B1[5]=NoPhase[id];
            B1[6]=NoPhase[id-1];
            B1[7]=NoPhase[id+(sizeX-1)];
            B1[8]=NoPhase[id+sizeX];

            B2[0]=0;
            B2[1]=0;
            B2[2]=0;
```

```
            B2[3]=0;
            B2[4]=0;
            B2[5]=NoPhaseX[id];
            B2[6]=NoPhaseX[id-1];
            B2[7]=NoPhaseX[id+(sizeX-1)];
            B2[8]=NoPhaseX[id+sizeX];

            B3[0]=0;
            B3[1]=0;
            B3[2]=0;
            B3[3]=0;
            B3[4]=0;
            B3[5]=NoPhaseY[id];
            B3[6]=NoPhaseY[id-1];
            B3[7]=NoPhaseY[id+(sizeX-1)];
            B3[8]=NoPhaseY[id+sizeX];



        }
        else      // any left spot but not corner
        {
            B1[0]=0;
            B1[1]=0;
            B1[2]=0;
            B1[3]=NoPhase[id];
            B1[4]=NoPhase[id-1];
            B1[5]=NoPhase[id+1];
            B1[6]=NoPhase[id+(sizeX-1)];
            B1[7]=NoPhase[id+sizeX];
            B1[8]=NoPhase[id+sizeX+1];


            B2[0]=0;
            B2[1]=0;
            B2[2]=0;
            B2[3]=NoPhaseX[id];
            B2[4]=NoPhaseX[id-1];
            B2[5]=NoPhaseX[id+1];
            B2[6]=NoPhaseX[id+(sizeX-1)];
            B2[7]=NoPhaseX[id+sizeX];
            B2[8]=NoPhaseX[id+sizeX+1];

            B3[0]=0;
            B3[1]=0;
            B3[2]=0;
            B3[3]=NoPhaseY[id];
            B3[4]=NoPhaseY[id-1];
            B3[5]=NoPhaseY[id+1];
            B3[6]=NoPhaseY[id+sizeX-1];
            B3[7]=NoPhaseY[id+sizeX];
            B3[8]=NoPhaseY[id+sizeX+1];


        }
    }
    else if(spot%sizeX==sizeX-1) // lower right corner
```

```
{
    if(spot/sizeY==sizeY-1)
    {
        B1[0]=0;
        B1[1]=0;
        B1[2]=0;
        B1[3]=0;
        B1[4]=0;
        B1[5]=NoPhase[id-1];
        B1[6]=NoPhase[id];
        B1[7]=NoPhase[id-sizeX];
        B1[8]=NoPhase[id-(sizeX+1)];

        B2[0]=0;
        B2[1]=0;
        B2[2]=0;
        B2[3]=0;
        B2[4]=0;
        B2[5]=NoPhaseX[id-1];
        B2[6]=NoPhaseX[id];
        B2[7]=NoPhaseX[id-sizeX];
        B2[8]=NoPhaseX[id-(sizeX+1)];

        B3[0]=0;
        B3[1]=0;
        B3[2]=0;
        B3[3]=0;
        B3[4]=0;
        B3[5]=NoPhaseY[id-1];
        B3[6]=NoPhaseY[id];
        B3[7]=NoPhaseY[id-sizeX];
        B3[8]=NoPhaseY[id-(sizeX+1)];

    }
    else   //  any lower spot but not corner
    {
        B1[0]=0;
        B1[1]=0;
        B1[2]=0;
        B1[3]=NoPhase[id-1];
        B1[4]=NoPhase[id-sizeX];
        B1[5]=NoPhase[id-(sizeX+1)];
        B1[6]=NoPhase[id+(sizeX-1)];
        B1[7]=NoPhase[id];
        B1[8]=NoPhase[id+sizeX];


        B2[0]=0;
        B2[1]=0;
        B2[2]=0;
        B2[3]=NoPhaseX[id-1];
        B2[4]=NoPhaseX[id-sizeX];
        B2[5]=NoPhaseX[id-(sizeX+1)];
        B2[6]=NoPhaseX[id+(sizeX-1)];
        B2[7]=NoPhaseX[id];
        B2[8]=NoPhaseX[id+sizeX];

        B3[0]=0;
```

```
            B3[1]=0;
            B3[2]=0;
            B3[3]=NoPhaseY[id-1];
            B3[4]=NoPhaseY[id-sizeX];
            B3[5]=NoPhaseY[id-(sizeX+1)];
            B3[6]=NoPhaseY[id+(sizeX-1)];
            B3[7]=NoPhaseY[id];
            B3[8]=NoPhaseY[id+sizeX];


        }
    }
    else if(spot/sizeY==sizeY-1) // any right spot but not corner
    {
        B1[0]=0;
        B1[1]=0;
        B1[2]=0;
        B1[3]=NoPhase[id-(sizeX+1)];
        B1[4]=NoPhase[id-1];
        B1[5]=NoPhase[id];
        B1[6]=NoPhase[id+1];
        B1[7]=NoPhase[id-(sizeX-1)];
        B1[8]=NoPhase[id-sizeX];

        B2[0]=0;
        B2[1]=0;
        B2[2]=0;
        B2[3]=NoPhaseX[id-(sizeX+1)];
        B2[4]=NoPhaseX[id-1];
        B2[5]=NoPhaseX[id];
        B2[6]=NoPhaseX[id+1];
        B2[7]=NoPhaseX[id-(sizeX-1)];
        B2[8]=NoPhaseX[id-sizeX];

        B3[0]=0;
        B3[1]=0;
        B3[2]=0;
        B3[3]=NoPhaseY[id-(sizeX+1)];
        B3[4]=NoPhaseY[id-1];
        B3[5]=NoPhaseY[id];
        B3[6]=NoPhaseY[id+1];
        B3[7]=NoPhaseY[id-(sizeX-1)];
        B3[8]=NoPhaseY[id-sizeX];
    }
    else                //any other spot not near the table limits
    {
        B1[0]=NoPhase[id];
        B1[1]=NoPhase[id-1];
        B1[2]=NoPhase[id+1];
        B1[3]=NoPhase[id-(sizeX-1)];
        B1[4]=NoPhase[id-sizeX];
        B1[5]=NoPhase[id-(sizeX+1)];
        B1[6]=NoPhase[id+(sizeX-1)];
        B1[7]=NoPhase[id+sizeX];
        B1[8]=NoPhase[id+sizeX+1];

        B2[0]=NoPhaseX[id];
```

```
    B2[1]=NoPhaseX[id-1];
    B2[2]=NoPhaseX[id+1];
    B2[3]=NoPhaseX[id-(sizeX-1)];
    B2[4]=NoPhaseX[id-sizeX];
    B2[5]=NoPhaseX[id-(sizeX+1)];
    B2[6]=NoPhaseX[id+(sizeX-1)];
    B2[7]=NoPhaseX[id+sizeX];
    B2[8]=NoPhaseX[id+sizeX+1];

    B3[0]=NoPhaseY[id];
    B3[1]=NoPhaseY[id-1];
    B3[2]=NoPhaseY[id+1];
    B3[3]=NoPhaseY[id-(sizeX-1)];
    B3[4]=NoPhaseY[id-sizeX];
    B3[5]=NoPhaseY[id-(sizeX+1)];
    B3[6]=NoPhaseY[id+(sizeX-1)];
    B3[7]=NoPhaseY[id+sizeX];
    B3[8]=NoPhaseY[id+sizeX+1];

}


for(k=0;k<8;k++) //sorting the tables
{
    for(j=0;j<8-k;j++)
    {


        if(B1[j]>B1[j+1])
        {
            swap=B1[j];
            B1[j]=B1[j+1];
            B1[j+1]=swap;
        }
        if(B2[j]>B2[j+1])
        {
            swap=B2[j];
            B2[j]=B2[j+1];
            B2[j+1]=swap;
        }
        if(B3[j]>B3[j+1])
        {
            swap=B3[j];
            B3[j]=B3[j+1];
            B3[j+1]=swap;
        }



    }

}


medianPhase[id]=B1[4]; //taking each median value
medianX[id]=B2[4];
medianY[id]=B3[4];
```

```
    if( (spot%sizeX!=0) && (spot/sizeY!=0) &&
(spot%sizeX!=(sizeX-1)) && (spot/sizeY!=(sizeY-1)) )
    {
        vect3D[0]=NoPhase[id-(sizeX+1)]-value;
        vect3D[1]=NoPhaseX[id-(sizeX+1)]-value;
        vect3D[2]=NoPhaseY[id-(sizeX+1)]-value;

        vect3D[3]=NoPhase[id-sizeX]-value;
        vect3D[4]=NoPhaseX[id-sizeX]-value;
        vect3D[5]=NoPhaseY[id-sizeX]-value;

        vect3D[6]=NoPhase[id-(sizeX-1)]-value;
        vect3D[7]=NoPhaseX[id-(sizeX-1)]-value;
        vect3D[8]=NoPhaseY[id-(sizeX-1)]-value;

        vect3D[9]=NoPhase[id-1]-value;
        vect3D[10]=NoPhaseX[id-1]-value;
        vect3D[11]=NoPhaseY[id-1]-value;

        vect3D[12]=NoPhase[id]-value;
        vect3D[13]=NoPhaseX[id]-value;
        vect3D[14]=NoPhaseY[id]-value;

        vect3D[15]=NoPhase[id+1]-value;
        vect3D[16]=NoPhaseX[id+1]-value;
        vect3D[17]=NoPhaseY[id+1]-value;

        vect3D[18]=NoPhase[id+(sizeX-1)]-value;
        vect3D[19]=NoPhaseX[id+(sizeX-1)]-value;
        vect3D[20]=NoPhaseY[id+(sizeX-1)]-value;

        vect3D[21]=NoPhase[id+sizeX]-value;
        vect3D[22]=NoPhaseX[id+sizeX]-value;
        vect3D[23]=NoPhaseY[id+sizeX]-value;

        vect3D[24]=NoPhase[id+(sizeX+1)]-value;
        vect3D[25]=NoPhaseX[id+(sizeX+1)]-value;
        vect3D[26]=NoPhaseY[id+(sizeX+1)]-value;

        //vectors multiplication


cscals[0]=medianPhase[id]*vect3D[0]+medianX[id]*vect3D[1]+medianY
[id]*vect3D[2];

cscals[1]=medianPhase[id]*vect3D[3]+medianX[id]*vect3D[4]+medianY
[id]*vect3D[5];

cscals[2]=medianPhase[id]*vect3D[6]+medianX[id]*vect3D[7]+medianY
[id]*vect3D[8];

cscals[3]=medianPhase[id]*vect3D[9]+medianX[id]*vect3D[10]+median
Y[id]*vect3D[11];

cscals[4]=medianPhase[id]*vect3D[12]+medianX[id]*vect3D[13]+media
nY[id]*vect3D[14];
```

```
cscals[5]=medianPhase[id]*vect3D[15]+medianX[id]*vect3D[16]+media
nY[id]*vect3D[17];

cscals[6]=medianPhase[id]*vect3D[18]+medianX[id]*vect3D[19]+media
nY[id]*vect3D[20];

cscals[7]=medianPhase[id]*vect3D[21]+medianX[id]*vect3D[22]+media
nY[id]*vect3D[23];

cscals[8]=medianPhase[id]*vect3D[24]+medianX[id]*vect3D[25]+media
nY[id]*vect3D[26];

        for(i=0;i<9;i++)
        sum=sum+cscals[i];

        mean=sum/9;

        atomicAdd(&(MagIM[spot]),mean);


    }

}
```

Appendix III MATLAB and MEX C code

a) MATLAB code that called the MEX C code

```
load data.mat;

diffX=1;
diffY=1;
neigm=3;
neign=3;
newDatasetA(80,80,40,52)=0;
```

```
I=setstruct(1).IM;
NoPhase1=setstruct(2).IM;
NoPhaseX1=setstruct(3).IM;
NoPhaseY1=setstruct(4).IM;
sizeX=setstruct(1).XSize;
sizeY=setstruct(1).YSize;
sizeT=setstruct(1).TSize;
slices=setstruct(1).ZSize;
NoPhase=reshape(NoPhase1,1,sizeX*sizeY*sizeT*slices);
NoPhaseX=reshape(NoPhaseX1,1,sizeX*sizeY*sizeT*slices);
NoPhaseY=reshape(NoPhaseY1,1,sizeX*sizeY*sizeT*slices);

t=cputime;

for slice=1:slices;
  magmean = mean(I(:,:,:,slice),3);
  MagIMout=zeros(sizeX*sizeY);

[MagIM2]=changed(MagIMout,NoPhase,NoPhaseX,NoPhaseY,slice,si
zeX,sizeY,sizeT,diffX);

  MagIM2=reshape(MagIM2,sizeX,sizeY);
  MagIM1 = MagIM2/(max(MagIM2(:)));
  temp = MagIM1.*magmean;
  norm = abs(temp/max(temp(:)));
  newDatasetA(:,:,:,slice) = repmat(norm,[1 1 sizeT]);
%replicate for all timeframes....
end
timeA=cputime-t;
```

b) MEX C code that is being called from MATLAB

```
#include <stdint.h>
typedef uint16_t char16_t;
#include <mex.h>
#include <matrix.h>
#include <stdlib.h>
#include <stdio.h>

void mexFunction(int nlhs, mxArray *plhs[], int nrhs, const
mxArray *prhs[])
{
```

```
    float
*MagIM,*medianPhase,*medianX,*medianY,*NoPhase,*NoPhaseX,*NoPhase
Y,*MagIMout;

    int slice,sizeX,sizeY,tsize,diffX;

    int id,start,end,sizeMagIM;
    int k,j,i,spot,position;
    float B1[9],B2[9],B3[9],swap,sum,mean,cscals[9],vect3D[27];

    medianPhase=(float *)malloc(sizeX*sizeY*tsize);
    medianX=(float *)malloc(sizeX*sizeY*tsize);
    medianY=(float *)malloc(sizeX*sizeY*tsize);

    MagIM=(float *)(mxGetPr(prhs[0]));
    NoPhase=(float *)(mxGetPr(prhs[1]));
    NoPhaseX=(float *)(mxGetPr(prhs[2]));
    NoPhaseY=(float *)(mxGetPr(prhs[3]));

    slice=(int)mxGetScalar(prhs[4]);
    sizeX=(int)mxGetScalar(prhs[5]);
    sizeY=(int)mxGetScalar(prhs[6]);
    tsize=(int)mxGetScalar(prhs[7]);
    diffX=(int)mxGetScalar(prhs[8]);

    plhs[0]=mxCreateNumericMatrix(sizeX*sizeY, 1, mxSINGLE_CLASS,
mxREAL);

    MagIMout=(float *)mxGetPr(plhs[0]);

    start=(slice-1)*sizeX*sizeY*tsize;
    end=slice*sizeX*sizeY*tsize;

    for (id=start;id<end;id++)
    {

        position=id%sizeX*sizeY*tsize;

        spot=id%(sizeX*sizeY);

        if(spot%sizeX==0)
        {
            if(spot/sizeY==0)
            {
                B1[0]=0;
                B1[1]=0;
                B1[2]=0;
                B1[3]=0;
                B1[4]=0;
                B1[5]=NoPhase[id];
```

```
        B1[6]=NoPhase[id+sizeX];
        B1[7]=NoPhase[id+1];
        B1[8]=NoPhase[id+sizeX+1];

        B2[0]=0;
        B2[1]=0;
        B2[2]=0;
        B2[3]=0;
        B2[4]=0;
        B2[5]=NoPhaseX[id];
        B2[6]=NoPhaseX[id+sizeX];
        B2[7]=NoPhaseX[id+1];
        B2[8]=NoPhaseX[id+sizeX+1];

        B3[0]=0;
        B3[1]=0;
        B3[2]=0;
        B3[3]=0;
        B3[4]=0;
        B3[5]=NoPhaseY[id];
        B3[6]=NoPhaseY[id+sizeX];
        B3[7]=NoPhaseY[id+1];
        B3[8]=NoPhaseY[id+sizeX+1];
    }
    else if(spot/sizeY==sizeY-1)
    {
        B1[0]=0;
        B1[1]=0;
        B1[2]=0;
        B1[3]=0;
        B1[4]=0;
        B1[5]=NoPhase[id];
        B1[6]=NoPhase[id-sizeX];
        B1[7]=NoPhase[id-(sizeX-1)];
        B1[8]=NoPhase[id+1];

        B2[0]=0;
        B2[1]=0;
        B2[2]=0;
        B2[3]=0;
        B2[4]=0;
        B2[5]=NoPhaseX[id];
        B2[6]=NoPhaseX[id-sizeX];
        B2[7]=NoPhaseX[id-(sizeX-1)];
        B2[8]=NoPhaseX[id+1];

        B3[0]=0;
        B3[1]=0;
        B3[2]=0;
        B3[3]=0;
        B3[4]=0;
```

```
            B3[5]=NoPhaseY[id];
            B3[6]=NoPhaseY[id-sizeX];
            B3[7]=NoPhaseY[id-(sizeX-1)];
            B3[8]=NoPhaseY[id+1];
        }
        else
        {
            B1[0]=0;
            B1[1]=0;
            B1[2]=0;
            B1[3]=NoPhase[id-sizeX];
            B1[4]=NoPhase[id];
            B1[5]=NoPhase[id+sizeX];
            B1[6]=NoPhase[id-(sizeX-1)];
            B1[7]=NoPhase[id+1];
            B1[8]=NoPhase[id+(sizeX+1)];

            B2[0]=0;
            B2[1]=0;
            B2[2]=0;
            B2[3]=NoPhaseX[id-sizeX];
            B2[4]=NoPhaseX[id];
            B2[5]=NoPhaseX[id+sizeX];
            B2[6]=NoPhaseX[id-(sizeX-1)];
            B2[7]=NoPhaseX[id+1];
            B2[8]=NoPhaseX[id+sizeX+1];

            B3[0]=0;
            B3[1]=0;
            B3[2]=0;
            B3[3]=NoPhaseY[id-sizeX];
            B3[4]=NoPhaseY[id];
            B3[5]=NoPhaseY[id+sizeX];
            B3[6]=NoPhaseY[id-(sizeX-1)];
            B3[7]=NoPhaseY[id+1];
            B3[8]=NoPhaseY[id+(sizeX+1)];
        }
    }
    else if(spot/sizeY==0)
    {
        if(spot%sizeX==(sizeX-1))
        {
            B1[0]=0;
            B1[1]=0;
            B1[2]=0;
            B1[3]=0;
            B1[4]=0;
            B1[5]=NoPhase[id];
            B1[6]=NoPhase[id-1];
            B1[7]=NoPhase[id+(sizeX-1)];
            B1[8]=NoPhase[id+sizeX];
```

```
        B2[0]=0;
        B2[1]=0;
        B2[2]=0;
        B2[3]=0;
        B2[4]=0;
        B2[5]=NoPhaseX[id];
        B2[6]=NoPhaseX[id-1];
        B2[7]=NoPhaseX[id+(sizeX-1)];
        B2[8]=NoPhaseX[id+sizeX];

        B3[0]=0;
        B3[1]=0;
        B3[2]=0;
        B3[3]=0;
        B3[4]=0;
        B3[5]=NoPhaseY[id];
        B3[6]=NoPhaseY[id-1];
        B3[7]=NoPhaseY[id+(sizeX-1)];
        B3[8]=NoPhaseY[id+sizeX];

    }
    else
    {
        B1[0]=0;
        B1[1]=0;
        B1[2]=0;
        B1[3]=NoPhase[id];
        B1[4]=NoPhase[id-1];
        B1[5]=NoPhase[id+1];
        B1[6]=NoPhase[id+(sizeX-1)];
        B1[7]=NoPhase[id+sizeX];
        B1[8]=NoPhase[id+sizeX+1];


        B2[0]=0;
        B2[1]=0;
        B2[2]=0;
        B2[3]=NoPhaseX[id];
        B2[4]=NoPhaseX[id-1];
        B2[5]=NoPhaseX[id+1];
        B2[6]=NoPhaseX[id+(sizeX-1)];
        B2[7]=NoPhaseX[id+sizeX];
        B2[8]=NoPhaseX[id+sizeX+1];

        B3[0]=0;
        B3[1]=0;
        B3[2]=0;
        B3[3]=NoPhaseY[id];
        B3[4]=NoPhaseY[id-1];
        B3[5]=NoPhaseY[id+1];
```

```
            B3[6]=NoPhaseY[id+sizeX-1];
            B3[7]=NoPhaseY[id+sizeX];
            B3[8]=NoPhaseY[id+sizeX+1];
        }
    }
    else if(spot%sizeX==sizeX-1)
    {
        if(spot/sizeY==sizeY-1)
        {
            B1[0]=0;
            B1[1]=0;
            B1[2]=0;
            B1[3]=0;
            B1[4]=0;
            B1[5]=NoPhase[id-1];
            B1[6]=NoPhase[id];
            B1[7]=NoPhase[id-sizeX];
            B1[8]=NoPhase[id-(sizeX+1)];

            B2[0]=0;
            B2[1]=0;
            B2[2]=0;
            B2[3]=0;
            B2[4]=0;
            B2[5]=NoPhaseX[id-1];
            B2[6]=NoPhaseX[id];
            B2[7]=NoPhaseX[id-sizeX];
            B2[8]=NoPhaseX[id-(sizeX+1)];

            B3[0]=0;
            B3[1]=0;
            B3[2]=0;
            B3[3]=0;
            B3[4]=0;
            B3[5]=NoPhaseY[id-1];
            B3[6]=NoPhaseY[id];
            B3[7]=NoPhaseY[id-sizeX];
            B3[8]=NoPhaseY[id-(sizeX+1)];
        }
        else
        {
            B1[0]=0;
            B1[1]=0;
            B1[2]=0;
            B1[3]=NoPhase[id-1];
            B1[4]=NoPhase[id-sizeX];
            B1[5]=NoPhase[id-(sizeX+1)];
            B1[6]=NoPhase[id+(sizeX-1)];
            B1[7]=NoPhase[id];
            B1[8]=NoPhase[id+sizeX];
```

```
            B2[0]=0;
            B2[1]=0;
            B2[2]=0;
            B2[3]=NoPhaseX[id-1];
            B2[4]=NoPhaseX[id-sizeX];
            B2[5]=NoPhaseX[id-(sizeX+1)];
            B2[6]=NoPhaseX[id+(sizeX-1)];
            B2[7]=NoPhaseX[id];
            B2[8]=NoPhaseX[id+sizeX];

            B3[0]=0;
            B3[1]=0;
            B3[2]=0;
            B3[3]=NoPhaseY[id-1];
            B3[4]=NoPhaseY[id-sizeX];
            B3[5]=NoPhaseY[id-(sizeX+1)];
            B3[6]=NoPhaseY[id+(sizeX-1)];
            B3[7]=NoPhaseY[id];
            B3[8]=NoPhaseY[id+sizeX];

        }
    }
    else if(spot/sizeY==sizeY-1)
    {
        B1[0]=0;
        B1[1]=0;
        B1[2]=0;
        B1[3]=NoPhase[id-(sizeX+1)];
        B1[4]=NoPhase[id-1];
        B1[5]=NoPhase[id];
        B1[6]=NoPhase[id+1];
        B1[7]=NoPhase[id-(sizeX-1)];
        B1[8]=NoPhase[id-sizeX];

        B2[0]=0;
        B2[1]=0;
        B2[2]=0;
        B2[3]=NoPhaseX[id-(sizeX+1)];
        B2[4]=NoPhaseX[id-1];
        B2[5]=NoPhaseX[id];
        B2[6]=NoPhaseX[id+1];
        B2[7]=NoPhaseX[id-(sizeX-1)];
        B2[8]=NoPhaseX[id-sizeX];

        B3[0]=0;
        B3[1]=0;
        B3[2]=0;
        B3[3]=NoPhaseY[id-(sizeX+1)];
        B3[4]=NoPhaseY[id-1];
        B3[5]=NoPhaseY[id];
        B3[6]=NoPhaseY[id+1];
```

```
    B3[7]=NoPhaseY[id-(sizeX-1)];
    B3[8]=NoPhaseY[id-sizeX];
}
else
{
    B1[0]=NoPhase[id];
    B1[1]=NoPhase[id-1];
    B1[2]=NoPhase[id+1];
    B1[3]=NoPhase[id-(sizeX-1)];
    B1[4]=NoPhase[id-sizeX];
    B1[5]=NoPhase[id-(sizeX+1)];
    B1[6]=NoPhase[id+(sizeX-1)];
    B1[7]=NoPhase[id+sizeX];
    B1[8]=NoPhase[id+sizeX+1];

    B2[0]=NoPhaseX[id];
    B2[1]=NoPhaseX[id-1];
    B2[2]=NoPhaseX[id+1];
    B2[3]=NoPhaseX[id-(sizeX-1)];
    B2[4]=NoPhaseX[id-sizeX];
    B2[5]=NoPhaseX[id-(sizeX+1)];
    B2[6]=NoPhaseX[id+(sizeX-1)];
    B2[7]=NoPhaseX[id+sizeX];
    B2[8]=NoPhaseX[id+sizeX+1];

    B3[0]=NoPhaseY[id];
    B3[1]=NoPhaseY[id-1];
    B3[2]=NoPhaseY[id+1];
    B3[3]=NoPhaseY[id-(sizeX-1)];
    B3[4]=NoPhaseY[id-sizeX];
    B3[5]=NoPhaseY[id-(sizeX+1)];
    B3[6]=NoPhaseY[id+(sizeX-1)];
    B3[7]=NoPhaseY[id+sizeX];
    B3[8]=NoPhaseY[id+sizeX+1];

}

for(k=0;k<8;k++)
{
    for(j=0;j<8-k;j++)
    {
        if(B1[j]>B1[j+1])
        {
            swap=B1[j];
            B1[j]=B1[j+1];
            B1[j+1]=swap;
        }
        if(B2[j]>B2[j+1])
        {
            swap=B2[j];
            B2[j]=B2[j+1];
```

```
                    B2[j+1]=swap;
                }
                if(B3[j]>B3[j+1])
                {
                    swap=B3[j];
                    B3[j]=B3[j+1];
                    B3[j+1]=swap;
                }
            }
        }

        medianPhase[position]=B1[4];
        medianX[position]=B2[4];
        medianY[position]=B3[4];

        if( (spot%sizeX!=0) && (spot/sizeY!=0) &&
(spot%sizeX!=(sizeX-1)) && (spot/sizeY!=(sizeY-1)) )
        {
            sum=(float)0;

            vect3D[0]=NoPhase[id-(sizeX+1)]-0.5;
            vect3D[1]=NoPhaseX[id-(sizeX+1)]-0.5;
            vect3D[2]=NoPhaseY[id-(sizeX+1)]-0.5;

            vect3D[3]=NoPhase[id-sizeX]-0.5;
            vect3D[4]=NoPhaseX[id-sizeX]-0.5;
            vect3D[5]=NoPhaseY[id-sizeX]-0.5;

            vect3D[6]=NoPhase[id-(sizeX-1)]-0.5;
            vect3D[7]=NoPhaseX[id-(sizeX-1)]-0.5;
            vect3D[8]=NoPhaseY[id-(sizeX-1)]-0.5;

            vect3D[9]=NoPhase[id-1]-0.5;
            vect3D[10]=NoPhaseX[id-1]-0.5;
            vect3D[11]=NoPhaseY[id-1]-0.5;

            vect3D[12]=NoPhase[id]-0.5;
            vect3D[13]=NoPhaseX[id]-0.5;
            vect3D[14]=NoPhaseY[id]-0.5;

            vect3D[15]=NoPhase[id+1]-0.5;
            vect3D[16]=NoPhaseX[id+1]-0.5;
            vect3D[17]=NoPhaseY[id+1]-0.5;

            vect3D[18]=NoPhase[id+(sizeX-1)]-0.5;
            vect3D[19]=NoPhaseX[id+(sizeX-1)]-0.5;
            vect3D[20]=NoPhaseY[id+(sizeX-1)]-0.5;

            vect3D[21]=NoPhase[id+sizeX]-0.5;
            vect3D[22]=NoPhaseX[id+sizeX]-0.5;
            vect3D[23]=NoPhaseY[id+sizeX]-0.5;
```

```
            vect3D[24]=NoPhase[id+(sizeX+1)]-0.5;
            vect3D[25]=NoPhaseX[id+(sizeX+1)]-0.5;
            vect3D[26]=NoPhaseY[id+(sizeX+1)]-0.5;


cscals[0]=(medianPhase[position]*vect3D[0]+medianX[position]*vect
3D[1]+medianY[position]*vect3D[2]);

cscals[1]=(medianPhase[position]*vect3D[3]+medianX[position]*vect
3D[4]+medianY[position]*vect3D[5]);

cscals[2]=(medianPhase[position]*vect3D[6]+medianX[position]*vect
3D[7]+medianY[position]*vect3D[8]);

cscals[3]=(medianPhase[position]*vect3D[9]+medianX[position]*vect
3D[10]+medianY[position]*vect3D[11]);

cscals[4]=(medianPhase[position]*vect3D[12]+medianX[position]*vec
t3D[13]+medianY[position]*vect3D[14]);

cscals[5]=(medianPhase[position]*vect3D[15]+medianX[position]*vec
t3D[16]+medianY[position]*vect3D[17]);

cscals[6]=(medianPhase[position]*vect3D[18]+medianX[position]*vec
t3D[19]+medianY[position]*vect3D[20]);

cscals[7]=(medianPhase[position]*vect3D[21]+medianX[position]*vec
t3D[22]+medianY[position]*vect3D[23]);

cscals[8]=(medianPhase[position]*vect3D[24]+medianX[position]*vec
t3D[25]+medianY[position]*vect3D[26]);


sum=sum+cscals[0]+cscals[1]+cscals[2]+cscals[3]+cscals[4]+cscals[
5]+cscals[6]+cscals[7]+cscals[8];

            mean=sum/9;


            MagIM[spot]=MagIM[spot]+mean;
            MagIMout[spot]=MagIM[spot];

            free(medianPhase);
            free(medianX);
            free(medianY);
        }

    }
}
```

References

[1] Class notes introduction to Biomedical Engineering, course offered by the Department of Computer Science and Biomedical Informatics, 2009.
[2] Angelos Chalkias, Human Torso Modeling for MRI Simulations, 2012
[3] Oberg P. (April 8, 2013) *Segmentation of cardiovascular tree in 4D from PC-MRI images*
[4] Jason Sanders, Edward Kandrot, CUDA by Example