UNIVERSITY OF CALIFORNIA,
IRVINE

GPU – Accelerated Finite Element Forward Solver in Diffuse Optical Tomography

THESIS

submitted in partial satisfaction of the requirements
for the degree of

MASTER OF SCIENCE

in Biomedical Engineering

by

Turkay Kart

Thesis Committee:
Associate Professor Gultekin Gulsen, Chair
Professor Min – Ying Su
Professor Frithjof Kruggel

2016

© 2016 Turkay Kart

TABLE OF CONTENTS

Page

LIST OF FIGURES iii

LIST OF TABLES v

ACKNOWLEDGMENTS vi

ABSTRACT OF THE THESIS vii

CHAPTER ONE INTRODUCTION 1

1.1. Optical Properties of Tissue 4

1.2. Modeling of Photon Propagation in a Medium 5

1.3. Measurement Techniques for Diffuse Optical Imaging 7

1.4. Image Reconstruction Procedures 10

1.4.1. Numerical Solution of the Forward Problem 11

1.4.2. Inverse Problem in DOT 14

1.5. Motivation of Thesis 17

CHAPTER TWO GPU PARALLELISM WITH CUDA AND MATLAB 19

2.1. Parallel Computing with CUDA 19

2.1.1. Introduction 19

2.1.2. CUDA Programming Model 20

2.2. MATLAB's MEX Files with CUDA 25

2.2.1. Description of MEX Files 25

2.2.2. C MEX File Example 27

2.2.3. Building CUDA C MEX File 28

CHAPTER THREE IMPLEMENTATION OF THE FORWARD MODEL 30

3.1. MATLAB Implementation of the Forward Model 34

3.2. CUDA C Implementation 35

CHAPTER FOUR SIMULATION RESULTS AND DISCUSSION 46

4.1. Computational Accuracy of CUDA C Implementation 48

4.2. Computational Efficiency of CUDA C Implementation 55

CHAPTER FIVE CONCLUSION AND FUTURE WORK 62

5.1. Conclusion 62

5.2. Future Work 63

REFERENCES 65

LIST OF FIGURES

Page

Figure 1 Intensity changes of source and detected light over time in continuous wave (CW) technique 8

Figure 2 Intensity changes of source and detected light over time in frequency-domain technique 9

Figure 3 Intensity changes of source and detected light over time in time-domain technique....... 9

Figure 4 Representation of CPU and GPU architectures 20

Figure 5 Representation of thread hierarchy in CUDA programming model 22

Figure 6 Data processing flow for a basic CUDA C program 23

Figure 7 Data processing flow for a basic CUDA C MEX File 28

Figure 8 Example of the discretization of the domain Ω using a 2D FEM mesh of a 25 mm diameter medium 30

Figure 9 Example of a 3D FEM mesh of a 25 mm diameter cylinder 32

Figure 10 Data processing flow of the forward model in MATLAB 34

Figure 11 Data processing flow of CUDA C implementation for the forward model 36

Figure 12 An example of source and detector placement in a 3D mesh and its corresponding 2D slice at z=0. 48

Figure 13 Simulation results of photon density in logarithmic scale 51

Figure 14 Photon density profiles of CUDA C and MATLAB implementations over a line starting from the point (12.5, 0, 0) to the point (-12.5, 0, 0) in standard scale 52

Figure 15 Photon density profiles of CUDA C and MATLAB implementations over a line starting from the point (12.5, 0, 0) to the point (-12.5, 0, 0) in logarithmic scale 52

Figure 16 Results of the flux at the boundary which is measured by 16 equally distant detectors 54

Figure 17 Simulation time of CUDA C implementation for meshes with different number of nodes 56

Figure 18 Simulation time of MATLAB implementation for meshes with different number of nodes 57

Figure 19 The speed-up graphs with CUDA C implementation for 3D meshes with different number of nodes 58

Figure 20 Simulation time of CUDA C implementation for different number of sources 59

Figure 21 Simulation time of MATLAB implementation for different number of sources 59

Figure 22 The speed-up graphs with CUDA C implementation for different number of sources 61

LIST OF TABLES

Page

Table 1 Sample CUDA code that adds two vectors X and Y of size *N* and the resulting vector is Z of size *N*. 21

Table 2 Sample CUDA implementation of the processing flow after getting input vectors X and Y of the code in Table 1........ 24

Table 3 Example code in a MEX File to read the input data of various types 26

Table 4 Implementation of the C MEX File that finds the maximum element in the input vector. 27

Table 5 An implementation of the operation *atomicAdd()* for double precision floating point provided by CUDA Toolkit Documentation........ 39

Table 6 The features of the 3D meshes used in the simulation studies such as number of nodes, tetrahedral elements and triangular elements at the boundary........ 47

ACKNOWLEDGMENTS

I owe extreme gratitude to my thesis advisor Dr. Gultekin Gulsen. This thesis would not be possible without his knowledge and guidance. Thank you for being a true advisor who genuinely cares and encourages to succeed. I really appreciate that you worked with me to overcome problems that I faced in my research no matter how busy you were. Thank you for being a good friend and for giving insightful advice for my career and life.

I would like to thank my thesis committee member and our center director, Dr. Su. Thank you for creating productive and cooperative atmosphere for research. Her guidance and encouragement helped me to learn different areas of clinical research.

My special thanks to Dr. Frithjof Kruggel for agreeing to be on my thesis committee. Thank you for teaching me basics of numerical methods, which helped me during my research. His teaching skills are admirable and I really appreciate you for leading me to have excellent understanding of subjects.

I would like to deeply thank all my lab members. Thank you Farouk for training me on the Finite Element method for DOT and for preparing me 3D meshes. Thank you Alex for explaining me the theory of DOT and for always being there to help. Thank you Hakan for helping me to understand analytical methods used in DOT. Also, I would like to thank all other lab members for their contributions. Finally, thank you all for being good friends.

Last but not least, my deepest thanks to my family. Thank you for your unconditional love, patience, and emotional support throughout my entire educational path. I truly appreciate your guidance and continuous encouragement that help me to shape all stages of my life.

ABSTRACT OF THE THESIS

GPU – Accelerated Finite Element Forward Solver in Diffuse Optical Tomography

By

Turkay Kart

Master of Science in Biomedical Engineering

University of California, Irvine, 2016

Associate Professor Gultekin Gulsen, Chair

Diffuse Optical Tomography (DOT) is a recently emerging imaging modality that provides optical properties of a tissue. One of main challenges in DOT is the fact that it requires modeling of light propagation in tissue, which is a time consuming and computationally intensive task. However, this process can be accelerated by parallelizing the application. A graphics processing unit (GPU) has the suitable architecture for this task. Therefore, the main goal of this work is to implement a GPU-based solver for the forward problem of DOT. A Finite Element method is utilized to solve the diffusion approximation of the photon transport model in the project. CUDA's parallel architecture and MATLAB software are combined for the implementation of GPU-based forward solver. Several simulations are performed to test computational accuracy and efficiency of the solver. The results show that GPU-based implementation provides a significant speed-up with high accuracy.

CHAPTER ONE

INTRODUCTION

In clinical settings, there is a variety of non-invasive imaging techniques used for diagnostics. Nuclear imaging and optical imaging are the imaging modalities that provide functional and molecular information. Positron Emission Imaging (PET) and Single Photon Emission Computed Tomography (SPECT) are two of the most commonly used clinical nuclear imaging modalities. By injecting appropriate radioactive agents which bind to certain types of tissues, SPECT and PET scan their distributions and show where they are concentrated in the body [1]. However, both of them have potential health risks due to injected radioactive agents and exposure to ionizing radiation. In addition, SPECT and PET procedures are very expensive.

Compared to nuclear imaging, optical imaging has unique advantages of being non-ionizing and having low-cost instrumentation. Diffuse Optical Imaging (DOI) is one of emerging non-invasive optical imaging modalities that utilize near-infrared (NIR) light to probe biological tissues. Intrinsic optical properties of tissue such as absorption and scattering coefficients or refractive index spatially vary and DOI provides their spatial distribution with NIR measurements along or near the boundary of tissue [2]. Since tissue absorption is lowest in the NIR portion of the light spectrum, this window provides the highest depth penetration. When recovering spatial distributions of intrinsic optical properties of the tissue in DOI, different methods such as planar imaging, spectroscopic or tomographic techniques can be used. Although planar imaging and spectroscopic technique are mostly used due to their simplicity, tomographic techniques such as diffuse optical tomography (DOT) can be utilized for certain applications, especially for thick tissue imaging.

DOI has been applied to clinical applications in several areas such as breast imaging, functional brain imaging, muscle imaging, and joint imaging. One particular application area of DOI is breast imaging for detection of breast tumors. Breast cancer is one of the leading causes of cancer death for women in the United States. Early detection plays an important role to reduce the chance of death from breast cancer. Invasive techniques such as surgical biopsies provide effective breast cancer screening. However, 80 percent of U.S. women who undergo surgical biopsies do not have cancer [3]. There are also non-invasive techniques in screening breast cancer such as X-ray mammography, magnetic resonance imaging (MRI), ultrasound (US) and positron emission tomography (PET) [4-6].

X-ray mammography is widely accepted as a very effective method in detection of breast cancer. However, repeated screening with X-rays mammography may have serious effects due to ionizing radiation. In addition, the specificity of X-ray mammography is modest, which may result in false negatives [7]. Meanwhile, MRI provides high sensitivity [8] but it has a high cost. On the other hand, using ultrasound (US) for detection of breast cancer may result in poor screening due to the visualization difficulty of deeper lesions. In addition to these disadvantages of X-ray mammography, MRI and US, they provide limited information about quantitative tissue function and composition since they are mostly anatomical imaging modalities [9]. Lastly, nuclear imaging modalities such as SPECT and PET can provide metabolic information but has potential risks associated with injected radionuclides and exposure to ionizing radiation [6].

Compared to these conventional imaging modalities mentioned above, NIR DOI for breast imaging has several advantages. The first advantage of DOI is that, unlike X-ray mammography and PET, it uses non-ionizing radiation. This is very beneficial for repetitive breast cancer screening. The second advantage is that it can be very cost-effective and compact.

For example, a continuous wave (CW) breast imaging system can be relatively inexpensive and portable. Last but not least, NIR DOI provides functional properties of the breast with high sensitivity, which may help to detect breast cancer. However, high tissue scattering limits the resolution of DOI, which is the main hurdle for the clinical translation of this imaging technology. Extensive efforts have been spent to improve the resolution of DOI in recent years.

Another particular application area of DOI is functional brain imaging. NIR DOI can monitor cerebral oxygenation and hemodynamics non-invasively using neuronal and hemodynamic signals [3, 10-13]. DOI can measure hemodynamic signals by quantifying the concentration of Hb and HbO_2 [3]. This ability provides the blood volume and oxygen saturation changes that are associated with brain functions. Functional magnetic resonance imaging (fMRI) is also used for brain imaging. Although both fMRI and DOI utilizes similar signals, i.e. blood oxygenation levels to measure brain activity, DOI has its own merits of being cost-effective and portable, and most importantly of having high temporal resolution [3].

In the following sections, optical properties of tissue, light-tissue interaction principles and modeling of photon migration in turbid media will be described. Besides, three types of measurement techniques used in DOI, namely continuous wave (CW), frequency- and time-domain approaches will be introduced. In addition to these, image reconstruction procedures involving forward and inverse problems will be introduced. Lastly, motivation of this thesis will be presented.

1.1. Optical Properties of Tissue

When light interacts with tissue, there are two main events observed: absorption and scattering. Absorption occurs when the energy of a photon completely dissipated to tissue while scattering refers to deviation in the direction of photons when they interact with tissue [14]. In the NIR wavelength range, tissue intrinsic absorbers are mainly oxy- and deoxy-hemoglobin as well as water and fat [3, 15-19]. The absorption spectra of these chromophores differ from each other with respect to their spectral shape as well as the magnitude of the main absorption peaks [20, 21]. The total absorption of tissue measured by the DOI is a linear combination of the absorption of these chromophores for a particular wavelength. Hence, measurements at multiple wavelengths can be used to quantify absolute tissue chromophore concentrations. This allows estimation of total hemoglobin and oxygen saturation, which can be used to distinguish diseased and normal tissue.

In addition to absorption, scattering is the other fundamental quantity that describes the propagation of light. Cells in biological tissues contain a vast variety of molecules, which have different indices of refraction. When the light propagates through tissue, it may redirect due to the sharp transitions of the index of refraction. This is considered as scattering in the tissue. Tissue scattering may be used to detect morphological and pathophysiological changes in tissue. The scattering spectral features may vary due to a diseased breast tissue. Recent studies have shown that there is substantial contrast in breast lesions relative to normal tissue, and that the scattering contrast between malignant and benign processes appears to be significant [22].

On the other hand, DOI is greatly affected by the highly scattering nature of the biological tissues. Most medical imaging modalities utilize other parts of the electromagnetic spectrum, in which a sizeable portion of the photons propagate mainly in straight paths in the tissue. By utilizing different methods to eliminate the scattered photons, these modalities can provide high resolution images. Unfortunately, similar approaches cannot be applied to DOI of thick tissues due to overwhelmingly dominant scattering. Instead, modeling of light propagation in tissue is required to form an image from the measured optical data as described in the next section.

1.2. Modeling of Photon Propagation in a Medium

As mentioned earlier, photons can be absorbed or scattered as they propagate in a medium. If the medium is only absorbing, Beer – Lambert's law can describe the portion of the beam attenuated or absorbed as it propagates through this medium [14]. Beer-Lambert's Law expressed below states that the photon intensity decreases exponentially when the light travels in an absorbing medium.

$$I(x) = I_0 \times \exp(-\mu_a x) \tag{1}$$

where I is the photon intensity in the medium, I_0 is the incident photon intensity, μ_a is the absorption coefficient, and x is the distance travelled.

In biological tissues, the incident NIR light is not only absorbed but also heavily scattered [2]. Hence a more comprehensive modeling method is necessary for DOI. For this purpose, Radiative Transfer Equation (RTE) is used to describe photon propagation in tissue [23-28],

$$\frac{1}{c_n}\frac{\partial L(r,\hat{s},t)}{\partial t}+\nabla L(r,\hat{s},t)\hat{s}=-(\mu_a(r)+\mu_s(r))L(r,\hat{s},t)+\mu_s\iint_{4\pi}L(r,\hat{s}',t)f(\hat{s}\cdot\hat{s}')d\Omega+q(r,\hat{s},t) \quad (2)$$

where *L(r,ŝ,t)* is radiance, μ_a is the absorption coefficient, μ_s is the scattering coefficient, *q(r,ŝ,t)* is the source term, and c_n is the speed of the light in the tissue, which is typically c_0 / 1.4. Meanwhile, *f(ŝ·ŝ´)* is the phase function satisfying $\iint_{4\pi} f(\hat{s}\cdot\hat{s}')d\Omega' = 1$. The reduced scattering coefficient $\mu_s´$ is defined as $\mu_s' = (1-g)\mu_s$, where *g* is the average cosine of the phase function and typically 0.9 for biological tissues. The term *g* provides the degree of anisotropy of the phase function [29].

Unfortunately, RTE is difficult to solve both analytically and numerically [3]. When scattering coefficient is more dominant than absorption coefficient ($\mu_s' \gg \mu_a$) and the source term is considered as isotropic, the photon propagation can be approximated by the diffusion equation [28, 30-33]:

$$\nabla\cdot[\kappa(r)\nabla\Phi(r,t)]-\mu_a(r)\Phi(r,t)-\frac{1}{c_n}\frac{\partial\Phi(r,t)}{\partial t}=-q_0(r,t) \quad (3)$$

where Φ*(r,t)* is the angle – independent photon density, *q_0(r,t)* is an isotropic light source term and κ*(r)* is the diffusion coefficient expressed as:

$$\kappa(r) = \frac{1}{3}(\mu_a + \mu_s') \tag{4}$$

Also, apart from ($\mu_s' >> \mu_a$) condition, there is one more important condition required to be satisfied for the diffuse approximation, which is the source - detector separation $\rho >> 1 / \mu_s'$ [3].

Using Fourier transformation, the diffusion equation in frequency domain can be obtained from Eq. (3) and the diffusion equation for the continuous light source can be derived by setting modulation angular frequency ω to 0 [28]:

$$\nabla \cdot [\kappa(r)\nabla\Phi(r,\omega)] - [\mu_a(r) + \frac{i\omega}{c_n}]\Phi(r,\omega) + q_0(r,\omega) = 0 \tag{5}$$

$$\nabla \cdot [\kappa(r)\nabla\Phi(r)] - \mu_a(r)\,\Phi(r) + q_0(r) = 0 \tag{6}$$

1.3. Measurement Techniques for Diffuse Optical Imaging

Diffuse optical imaging measures the reflected and/or transmitted light through the tissue to obtain its optical properties. There are mainly three types of measurement techniques in DOI: continuous wave (CW), frequency-domain and time-domain techniques.

In continuous wave (CW) technique, intensity of the light source is constant and time independent; hence the detected light level is also constant as shown in Fig. 1. The advantage of CW technique is that it is a very straightforward technique and has low cost. Many different CW systems have been reported in the literature [34-36].

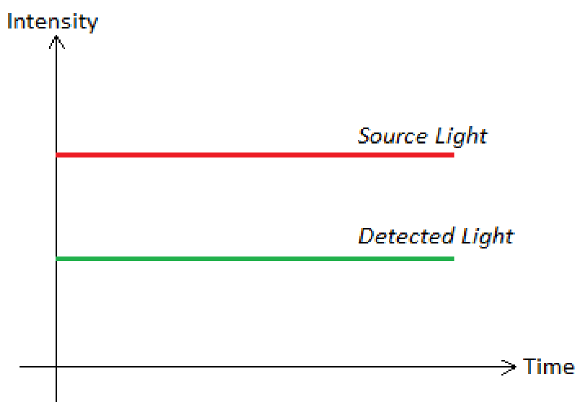


Figure 1 Intensity changes of source and detected light over time in continuous wave (CW) technique

Instead of a light source with a constant intensity, one with sinusoidally modulated amplitude is used in the frequency-domain technique. As seen in Fig. 2, the detected light signal has reduced amplitude and a phase shift compared to the source light. Although it is more comprehensive than the CW technique, when it is compared to time-domain technique, the frequency-domain technique is cheaper and its data acquisition time is shorter [37]. As the frequency increases, the phase shift difference between normal and tumor tissue increases; on the other hand, the sensitivity and accuracy of the detection system decreases [38]. As a final note, there are some applications which combine CW and frequency-domain techniques [39-42]. This improves the overall quantitative accuracy in chromophore concentration estimates [41].

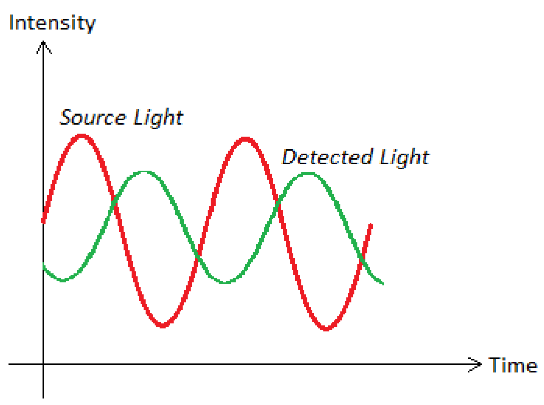


Figure 2 Intensity changes of source and detected light over time in frequency-domain technique

The last major measurement technique used in DOI is time-domain approach. It utilizes short laser pulses, namely sub-nanosecond laser pulses, and records flight time of photons as a function of time [43-45]. To do that, Temporal Point Spread Function (TPSF) is measured. Time-domain technique is very expensive and slow, but it provides richer data than frequency-domain technique and hence, leads to more accurate results [37].

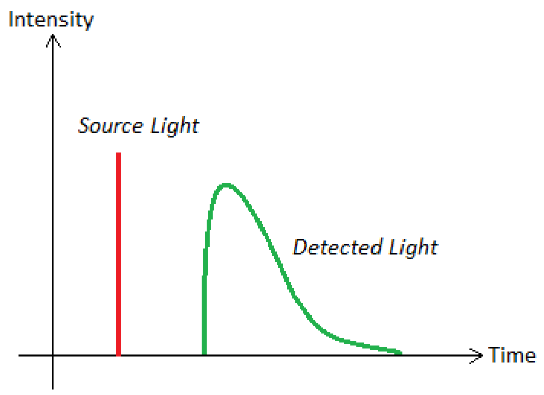


Figure 3 Intensity changes of source and detected light over time in time-domain technique

1.4. Image Reconstruction Procedures

The main goal of DOT is to find the distribution of optical properties in the tissue through image reconstruction. Image reconstruction in DOT is divided into two steps: forward and inverse problems. In the forward problem, a photon density map is obtained using the photon diffusion model. The forward problem can be used to estimate measurements for a medium with known optical properties and geometry. Three types of computational methods can be used to solve the forward problem of DOT:

- **Analytical Methods:** The Green's function is used to find optical properties of the tissue. The source is considered as a spatial and temporal delta-function [46]. The main drawback of analytical methods is that they provide solutions only for simple homogeneous geometry with a regular shape, or for media with a single spherical perturbation [33, 47].
- **Monte-Carlo Methods:** Monte-Carlo method is a computational method based on statistics [48-50]. While absorption and scattering events are simulated, all the events for each individual photon are recorded until the photon exits the medium or is completely absorbed. A large number of photons are traced as they propagate the medium and the sum of all events gives a statistical distribution. This method is very flexible and can be applied for complex heterogeneous geometry. Although Monte-Carlo method provides very accurate results, its main drawback is that it is computationally expensive.
- **Numerical Methods:** There are various numerical techniques: Finite Element Method (FEM), Finite Difference Method (FDM), Finite Volume Method (FVM), Boundary Element Method (BEM) [25, 39, 51, 52]. FEM is a very popular technique used to solve the forward problem since it is flexible, suitable for complex heterogeneous geometry

and provides a good approximation. More importantly, it is much faster than Monte-Carlo methods. The numerical solution in this work is obtained by FEM and will be explained later in detail.

To generate an image, an inverse problem should be solved. For this purpose, the forward problem is used. After assuming an initial optical property distribution, the difference between calculated and measured photon densities along the boundary is minimized by an iterative approach [2]. Reconstructing image in DOT is a challenging problem mainly due to inherently ill-posed inverse problem and non-linearity of the transport model. In addition, it can be a seriously ill-determined or under-determined problem, which means the number of independent equations is much less than the number of unknowns. In other words, the number of independent measurements is less than the number of pixels in the image of optical properties obtained in DOT [46, 53]. Therefore, there is no unique solution for given detected signals, which implies it is necessary to regularize the problem by iterative model-based linear algebra [53]. This is called regularization. Due to the reasons stated above, image reconstruction in DOT requires intense computations. As computers become more powerful, performing those computations become easier. This helps researchers to develop fast and reliable computational methods.

1.4.1. Numerical Solution of the Forward Problem

The forward problem starts with the photon diffusion model expressed as in Eq. 3. The photon diffusion model is a partial differential equation (PDE) and a numerical solution is obtained in this work by using a finite element method (FEM) framework.

In FEM framework, the photon diffusion equation can be written as [28]:

$$\int_{\Omega} dx\nabla\varphi \cdot (\kappa\nabla\Phi) + \int_{\Omega} dx\mu_a\varphi\Phi_x + \frac{1}{2\mathrm{A}}\int_{d\Omega} ds\varphi\Phi_x + \frac{i\omega}{c}\int_{\Omega} dx\varphi\Phi_x = \int_{\Omega} dx\varphi q_0 \quad (7)$$

where φ is an arbitrary test function. The function Φ can be expressed as $\Phi = \sum_{j=1}^{N} \xi_j\varphi_j$ where φ_j is a basis function, which is a piece-wise linear function, for the *j*th node. The distributions κ and μ_a can be written as $\kappa = \sum_{k=1}^{N} \kappa_k\varphi_k$ and $\mu_a = \sum_{k=1}^{N} \mu_{ak}\varphi_k$.

To simplify the above diffusion equation in FEM framework, it can be represented as [28]:

$$(A + B + C + \frac{i\omega}{c}D)\xi_x = Q \quad (8)$$

where

$$(A)_{ij} = \sum_{k=1}^{N} \kappa_k \int_{\Omega} dx\varphi_k(\nabla\varphi_i g\nabla\varphi_j) \quad (9)$$

$$(B)_{ij} = \sum_{k=1}^{N} \mu_{a,k} \int_{\Omega} dx\varphi_k(\varphi_i\varphi_j) \quad (10)$$

$$(C)_{ij} = \int_{d\Omega} ds(\varphi_i\varphi_j) \quad (11)$$

$$(D)_{ij} = \int_{\Omega} dx(\varphi_i\varphi_j) \quad (12)$$

$$(Q)_i = \int_{\Omega} dx(\varphi_i q_0) \quad (13)$$

$$\varphi = \sum_{x=1}^{N} \varphi_x \quad (14)$$

In the forward problem, the linear equations are solved to obtain the photon density distribution Φ in the medium. To obtain the photon flux at the boundary, the photon density Φ is needed. Robin boundary condition expressed below can be used to find the photon flux at the boundary [28]. According to the Robin boundary condition, the total inward directed photon current is zero on the boundary except at the regions of the source term.

$$\Phi(r,\omega) + 2A\kappa(r) \cdot \frac{\partial \Phi(r,\omega)}{\partial n} = 0 \tag{15}$$

where *n* is the direction perpendicular to the boundary surface. *A* in Eq. 15 is the boundary mismatch parameter related to light reflection at the boundary and determined by Fresnel's reflections [23]. The photon flux *J* at the boundary can be expressed as [28]:

$$J = -\kappa(r)\frac{\partial \Phi}{\partial n} = \frac{\Phi}{2A} \tag{16}$$

For frequency-domain technique, the photon flux calculated at the boundary is a complex number. There are mainly three types of data that can be obtained from the complex data at the boundary [28]:

- Type I shows real and imaginary parts of the calculated photon flux.
- Type II shows amplitude and phase data of the calculated photon flux.
- Type III shows amplitude in log scale and phase data of the calculated photon flux.

Type III is commonly used for the reconstruction purposes because it scales dynamic range of the amplitude data that can be in the range from 1 to 10^{-10} or even lower depending on measurement conditions. In log scale, the amplitude data is in the range from 0 to -10, which is similar to the level of dynamic range of the phase data. In addition, Type III makes it easy to see the relation between the amplitude / phase data and RF signal data recorded from instruments [28].

The diffusion equation provides reasonable solution accuracy for DOI owing to the assumption that reduced scattering coefficient is more dominant than absorption coefficient in most tissues, except in small regions of tissues such as cysts in the breast, cerebrospinal fluids in the brain, and cynovial fluids in the joints [2]. It is mentioned that diffusion equation is an approximation of Radiative Transfer Equation (RTE). Therefore, it is clear that RTE can provide a more accurate solution than diffusion equation but RTE has a high computational complexity. In addition, since the inverse problem explained in the next section has an iterative process, simulation time for DOT reconstruction can be substantially high. A better way to have a more accurate solution can be the use of high-order diffusion equations. However, as the order of the diffusion equation increases, the computational cost increases.

1.4.2. Inverse Problem in DOT

Inverse problem in DOT involves finding the distribution of optical properties of the tissue by minimizing difference between calculated and measured photon density at the boundary. A least-square method can be used to achieve the minimization. Newton's approach can be employed in least-square minimization of the following objective function [2]:

$$F(\Phi, \kappa, \mu_a) = \sum_{j=1}^{M} \left(\Phi_j^o - \Phi_j^c\right)^2 \tag{17}$$

where Φ_j^c and Φ_j^o are the computed and observed photon densities at the boundary for j=1, 2,…,M. The following nonlinear system of equations can be obtained by using least-square criteria [2]:

$$\begin{aligned}
\frac{\partial F}{\partial \chi_1} &=> -\sum_{j=1}^{M} \left(\Phi_j^o - \Phi_j^c\right) \frac{\partial \Phi_j^c}{\partial \chi_1} = 0 \\
\frac{\partial F}{\partial \chi_2} &=> -\sum_{j=1}^{M} \left(\Phi_j^o - \Phi_j^c\right) \frac{\partial \Phi_j^c}{\partial \chi_2} = 0 \\
&\vdots \\
\frac{\partial F}{\partial \chi_{2N}} &=> -\sum_{j=1}^{M} \left(\Phi_j^o - \Phi_j^c\right) \frac{\partial \Phi_j^c}{\partial \chi_{2N}} = 0
\end{aligned} \tag{18}$$

where N is the total number of κ or μ_a parameters, and the symbol χ expresses κ or μ_a. The nonlinear system of equations in Eq. 18 can be written as follows:

$$\hat{F} = (f_1, f_2, \dots, f_{2N})^T \tag{19}$$

where f_i is the left-hand side of each equation for *i* = 1, 2, …, 2N. Then, Newton's method gives:

$$G\Delta\chi = -\hat{F} \tag{20}$$

where $\Delta\chi = \chi^{(n)} - \chi^{(n-1)}$, n is the iterative step in Newton's method, and

$$G = \begin{pmatrix} \frac{\partial f_1}{\partial \chi_1} & \frac{\partial f_1}{\partial \chi_2} & \cdots & \frac{\partial f_1}{\partial \chi_{2N}} \\ \frac{\partial f_2}{\partial \chi_1} & \frac{\partial f_2}{\partial \chi_2} & & \frac{\partial f_2}{\partial \chi_{2N}} \\ \vdots & & \ddots & \vdots \\ \frac{\partial f_{2N}}{\partial \chi_1} & \frac{\partial f_{2N}}{\partial \chi_2} & \cdots & \frac{\partial f_{2N}}{\partial \chi_{2N}} \end{pmatrix} \tag{21}$$

After several calculations which are described in [2], $\hat{F}$ and G functions can be derived as follows:

$$-\hat{F} = \mathfrak{J}^{\mathrm{T}} \cdot \mathrm{b} \tag{22}$$

$$G = \mathfrak{J}^{\mathrm{T}}\mathfrak{J} \tag{23}$$

where $\mathfrak{J}$ is the Jacobian matrix and $\mathrm{b} = (\Phi_1^o - \Phi_1^c, \Phi_2^o - \Phi_2^c, \ldots, \Phi_M^o - \Phi_M^c)^{\mathrm{T}}$. Then, Eq. 20 can be rewritten as:

$$\mathfrak{J}^{\mathrm{T}}\mathfrak{J}\Delta\chi = \mathfrak{J}^{\mathrm{T}}(\Phi^o - \Phi^c) \tag{24}$$

Regularization methods are used to overcome ill-posedness of the inverse problem due to the matrix $\mathfrak{J}^{\mathrm{T}}\mathfrak{J}$. Hence, Eq. 24 becomes

$$(\mathfrak{J}^{\mathrm{T}}\mathfrak{J} + \lambda I)\Delta\chi = \mathfrak{J}^{\mathrm{T}}(\Phi^{o} - \Phi^{c}) \tag{25}$$

where I is the identity matrix and λ is the regularization parameter. There are many types of algorithms to implement the inverse part of image reconstruction. An algorithm based on Levenberg–Marquardt method is a common approach and it can solve the inverse problem iteratively until the stopping criteria are satisfied [54, 55].

1.5. Motivation of Thesis

Long simulation time is one of main challenges in DOT since it requires the processing of huge amount of data. Image reconstruction in DOT involves solving the forward model iteratively. Since the total number of iterations is large, the total simulation time decreases as the time to solve the forward model decreases. Thus, it is important to execute the DOT program efficiently. However, when the DOT program is run on a central processing unit (CPU), it is executed serially. One way to reduce the simulation time for the forward problem is to execute the DOT program in parallel. Although we have multi-core CPUs which provide parallelism, they are not enough to decrease this time-consuming procedure significantly. Graphics processing units (GPU) have a design that is aimed to handle large number of tasks simultaneously since they have a much larger number of cores. GPU's parallel execution of the forward solver can provide considerable speed-up compared to CPU's serial execution. In this work, we implement a GPU-based solver for the forward problem of DOT. To do that, CUDA's parallel architecture and MATLAB are combined and a Finite Element method is used.

We have two core goals for this project. Our first goal is to implement a Finite Element forward solver for DOT on a GPU device. We aim to show that computations in GPU-based forward solver are highly accurate when compared to a CPU-based forward solver. The second core goal is to achieve a significant speed-up in simulation time of the forward solver. Smaller simulation times for DOT can increase the applicability of DOT in clinical procedures.

It is worth to mentioning an important implied outcome of speed-up with a GPU-based implementation. Total simulation time of DOT is highly dependent on mesh complexity. For a CPU-based implementation, coarse meshes are used to reduce the simulation time. However, using coarse meshes may result in low accuracy. Therefore, mesh quality should be high so that the mesh could be closer to continuous form, which provides more realistic results. Owing to speed-up obtained with a GPU-based implementation, it is possible to use finer mesh for the same amount of simulation time. In addition, the number of sources used is another factor that increases the total simulation time. Hence, larger number of sources can be used in a GPU-based implementation than that can be used in a CPU-based implementation for the same simulation time.

CHAPTER TWO

GPU PARALLELISM WITH CUDA AND MATLAB

2.1. Parallel Computing with CUDA

2.1.1. Introduction

Over time, the need for parallelism increased with the demand of computationally intensive programs in several areas including the medical industry. As a result, programmers have moved from serial to parallel programming to satisfy the computational needs. CPU vendors started to produce multi-core CPUs to improve performance of applications by parallelism. However, this was not enough to satisfy the need. Therefore, graphics processor unit (GPU) have become a better candidate owing to larger number of execution cores. NVIDIA introduced the Compute Unified Device Architecture (CUDA) programming model to develop GPU parallelized applications [56-59]. Programmers can merge C programming language with CUDA to write their GPU parallelized codes.

Data-parallel programming model can be used for applications which process large blocks of data and GPU is a better platform for those applications since GPU architecture has more transistors to data processing rather than data caching when compared to CPU architecture.

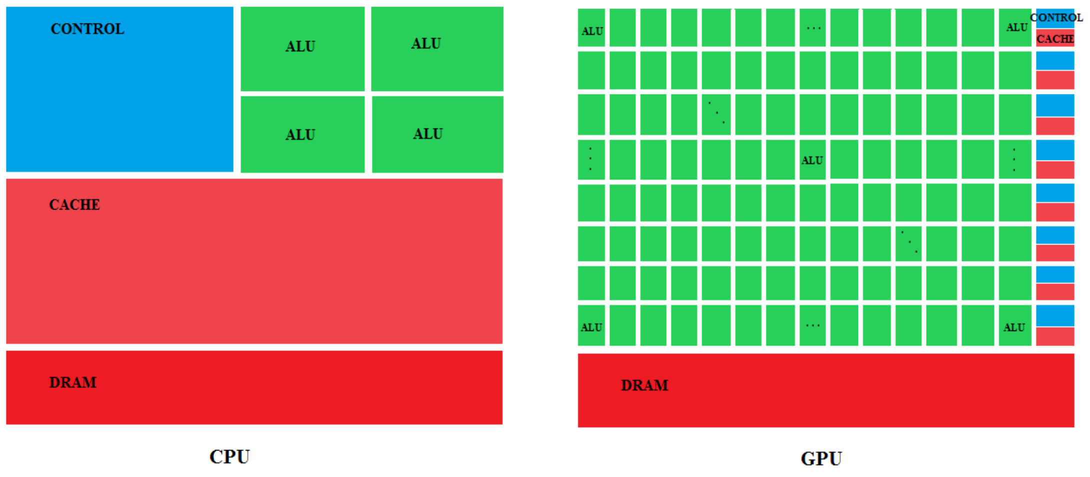


Figure 4 Representation of CPU and GPU architectures

The main concept of CUDA parallel programming is to achieve concurrency. For example, if an algorithm does the same calculation iteratively over a large number of steps, the individual calculation of each step can be found concurrently. Hence, the performance of the algorithm improves drastically. The CUDA parallel programming model partitions the problem into sub-problems that can be solved by grids of blocks, each of which contains multiple threads. In the following section, CUDA parallel programming model will be discussed in detail.

2.1.2. CUDA Programming Model

The fundamental building block of parallel computing is called a thread. To run multiple threads in a parallelized program, a function that runs on GPU is needed. This function is called a kernel. The total number of threads is specified when a kernel function is defined. Hence, kernels are executed *N* times in parallel by *N* different CUDA threads. Multiple parallel CUDA threads are logically grouped into a block. In addition, a set of blocks composes a grid. Using a number of threads and blocks, and a special notation: <<< ... >>>, the following syntax is used to invoke a kernel.

kernel_function <<<number_of_blocks, number_of_threads>>>(parameter1, parameter2, ...);

The total number of threads specifies the block size whereas the total number of blocks specifies the size of the grid. *__global__* declaration specifier and parameters are used to define a kernel. In addition, it is essential that a unique thread ID is assigned to each thread executing the kernel. The following example shows how to define and invoke a kernel that runs *N* multiple threads simultaneously:

Table 1 Sample CUDA code that adds two vectors X and Y of size *N* and the resulting vector is Z of size *N*.

```
// Defining Kernel
__global__ void VectorAdd(float *X, float *Y, float *Z){

        // Assigning thread ID
        int i = blockIdx.x * blockDim.x + threadIdx.x;

        // Adding two vectors X and Y
        Z[i] = X[i] + Y[i];
}
int main(){

        ...

        // Invoking kernel with N threads
        VectorAdd<<< 1, N >>> (d_X, d_Y, d_Z);

        ...

}
```

In the sample code above, the thread ID for each thread is calculated using block ID for each block and its dimension. This implies that threads are grouped according to the thread hierarchy in the CUDA parallel programming model. In addition, the thread index *threadIdx* is a 3-component vector which can be used to define one-dimensional, two-dimensional or three-dimensional block of threads. For example, in the above code, the threads are defined by one-dimensional thread index *threadIdx.x*. The following figure shows the thread hierarchy in the CUDA programming model:

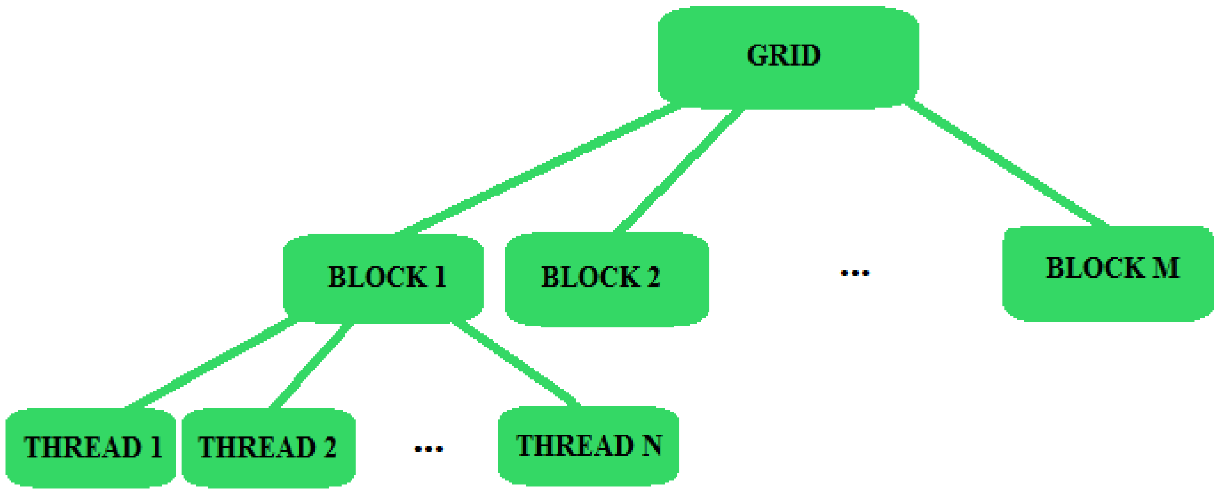


Figure 5 Representation of thread hierarchy in CUDA programming model

The GPU hardware limits the number of threads per block. This limit is 512 threads per block on the early hardware whereas it is 1024 threads per block on the later hardware. In addition, a kernel can have a grid of a total of 65,536 blocks, which means that 33.5 million threads in total can be scheduled on an early hardware. Therefore, a program can process up to 33.5 million array elements when one thread per array element is assumed.

Before starting, it is important to understand the processing flow of a basic CUDA program. In the first stage, the processing flow starts with allocating variables in the device. This memory allocation can be done by using the function *cudaMalloc().* Then, the function *cudaMemcpy()* allows the transfer of data between host and device memory. In the third stage, after all the necessary data is transferred to the device memory, the GPU processes it in parallel and the result is stored in the device memory. At the final stage, the result is copied from the device to the host memory and the device memory is freed by using the function *cudaFree().* The processing flow for a basic CUDA C program and its sample implementation for the code in Table 1 are given below:

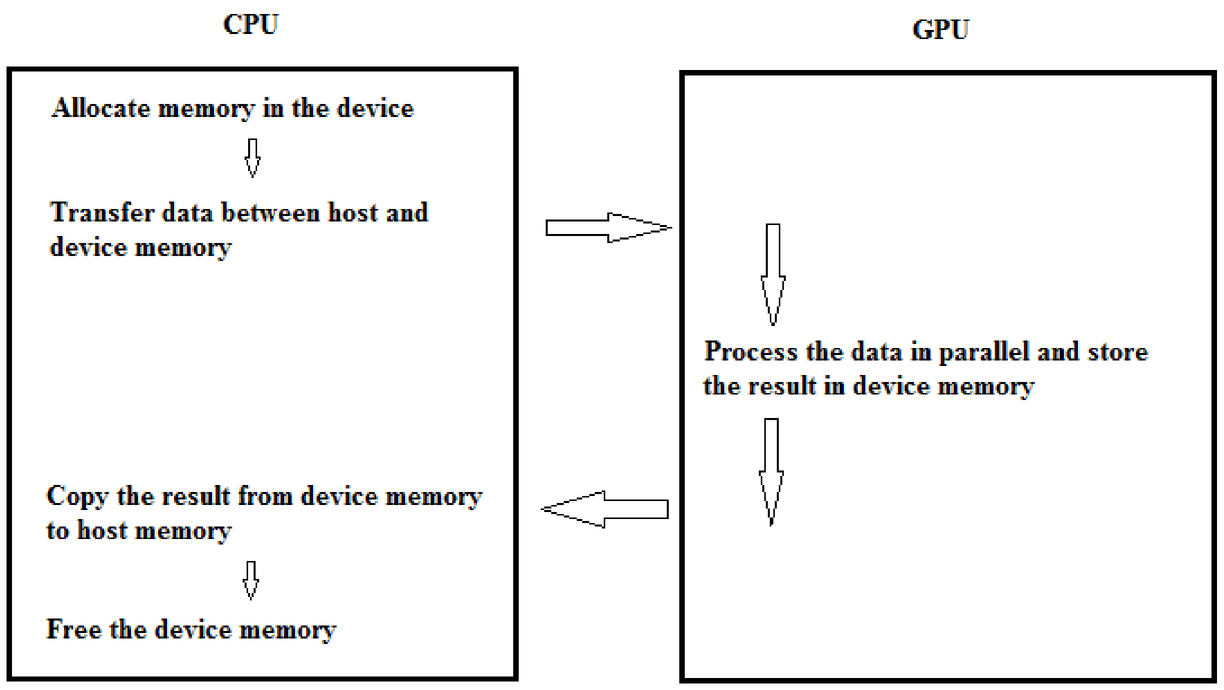

CPU
GPU
Allocate memory in the device
Transfer data between host and device memory
Process the data in parallel and store the result in device memory
Copy the result from device memory to host memory
Free the device memory

Figure 6 Data processing flow for a basic CUDA C program

Table 2 Sample CUDA implementation of the processing flow after getting input vectors X and Y of the code in Table 1

```
int main(){

        ...

        size_t size = N * sizeof(float);

        // Allocating vector Z in host memory
        float* Z = (float*) malloc(size);

        // Allocating vectors X and Y in device memory
        float* d_X;
        cudaMalloc(&d_X, size);
        float* d_Y;
        cudaMalloc(&d_Y, size);
        float* d_Z;
        cudaMalloc(&d_Z, size);

        // Copying vectors X and Y from host memory to device memory
        cudaMemcpy(d_X, X, size, cudaMemcpyHostToDevice);
        cudaMemcpy(d_Y,  Y, size, cudaMemcpyHostToDevice);

        ...

        // Copying the result from device memory to host memory
        cudaMemcpy(Z,  d_Z, size, cudaMemcpyDeviceToHost);

        // Free device memory
        cudaFree(d_X);
        cudaFree(d_Y);
        cudaFree(d_Z);

}
```

2.2. MATLAB's MEX Files with CUDA

2.2.1. Description of MEX Files

MATLAB Executable Files or MEX Files give the ability to use C/C++ or FORTRAN codes within the MATLAB environment [60, 61]. In addition, MEX Files can call MATLAB functions from C/C++ or FORTRAN code and the data between MEX Files and MATLAB environment can be transferred from one to another.

There are two advantages of using a MEX File over a MATLAB file:

- MEX Files can be used to increase speed of the application. Bottlenecks can be optimized to improve the performance.
- Custom code can be written for the MATLAB environment and MEX Files can integrate it to the environment.

The structure of C MEX Files is almost the same as typical C source files. However, instead of the *main()* function as the entry point, similar to a typical C source code, C MEX Files use the gateway routine *mexFunction()* as the entry point to be consistent with the MATLAB environment. The entry point of C MEX Files has the following syntax:

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

        /* C code here */

}
```

There are four input parameters at the gateway routine: *plhs* is the array of arguments at the left hand side, or the array of output arguments, *nlhs* is the number of arguments at the left hand side, or the size of the *plhs* array, *prhs* is the array of arguments at the right hand side, or the array of input arguments, and *nrhs* is the number of arguments at the right hand side, or the size of the *prhs* array.

We use MATLAB's MEX functions to read the input data in the MEX Files. *mxGetPr* is used to point the input matrix data of double type, *mxGetData* is used to point the input matrix data of other types such as *float* or *int*, *mxGetScalar* is used to get the value of scalar input, and *mxGetM* and *mxGetN* are used to get the size of the matrix. To create the output argument, *plhs[0]*, and to assign a double variable to *plhs[0]*, *mxCreateDoubleMatrix* and *mxGetPr* were used respectively.

Table 3 Example code in a MEX File to read the input data of various types

```
double* input1;
input1 = mxGetPr(prhs[0]);

float* input2;
input = (float*) mxGetData(prhs[1]);

double input3;
input3 = mxGetScalar(prhs[2]);

mwSize N;
N = mxGetN(prhs[0]);

double* outputMatrix;
plhs[0] = mxCreateDoubleMatrix(mRow, nCol, mxREAL);
outputMatrix = mxGetPr(plhs[0]);
```

2.2.2. C MEX File Example

As a reference, a C MEX File Example is presented in Table 4.

Table 4 Implementation of the C MEX File that finds the maximum element in the input vector.

```
#include "mex.h"

// The gateway function
void mexFunction(int nlhs, mxArray *plhs[], int nrhs, const mxArray *prhs[]){
        // Declaring variables
        int mRow, nCol;
        double *inVector, myOutput, maxElement;
        // Checking if the number of input argument is satisfied
        if(nrhs != 1){
                mexErrMsgTxt("There must be only one input.");
        }
        // Getting the size of the input
        mRow = mxGetM(prhs[0]);
        nCol = mxGetN(prhs[0]);
        // Checking if the input is a 1 by N vector.
        if(mRow != 1 || nCol < 1){
                mexErrMsgTxt("The input should be a 1 by N vector.");
        }
        // Reading the input data
        inVector = mxGetPr(prhs[0]);
        // Creating the output argument
        plhs[0] = mxCreateDoubleMatrix(1, 1, mxREAL);
        // Getting a pointer to the output
        myOutput = mxGetPr(plhs[0]);
        // C Code to find the maximum element in the input vector
        maxElement = inVector[0];
        if(nCol > 1){
                for(int i=1; i < nCol; i++){
                        maxElement = (maxElement < inVector[i]) ? inVector[i] : maxElement;
                }
        }
        myOutput = maxElement;
}
```

2.2.3. Building CUDA C MEX File

We mentioned that MEX Files could improve the performance of an application. In addition, the speed of a program can be further increased by combining CUDA and C MEX codes, which makes computationally expensive algorithms much faster. In addition, although MATLAB provides an extensive library of GPU-enabled functions, there are additional libraries in CUDA toolkit provided by NVIDIA. Hence, a function in one of these libraries can be called using CUDA C MEX Files.

For CUDA C MEX Files, the entry point *mexFunction()*, the input *prhs* and the output *plhs* arguments are the same with C MEX Files. However, the processing flow contains both CUDA and MEX routines. The following figure shows the processing flow for CUDA C MEX Files:

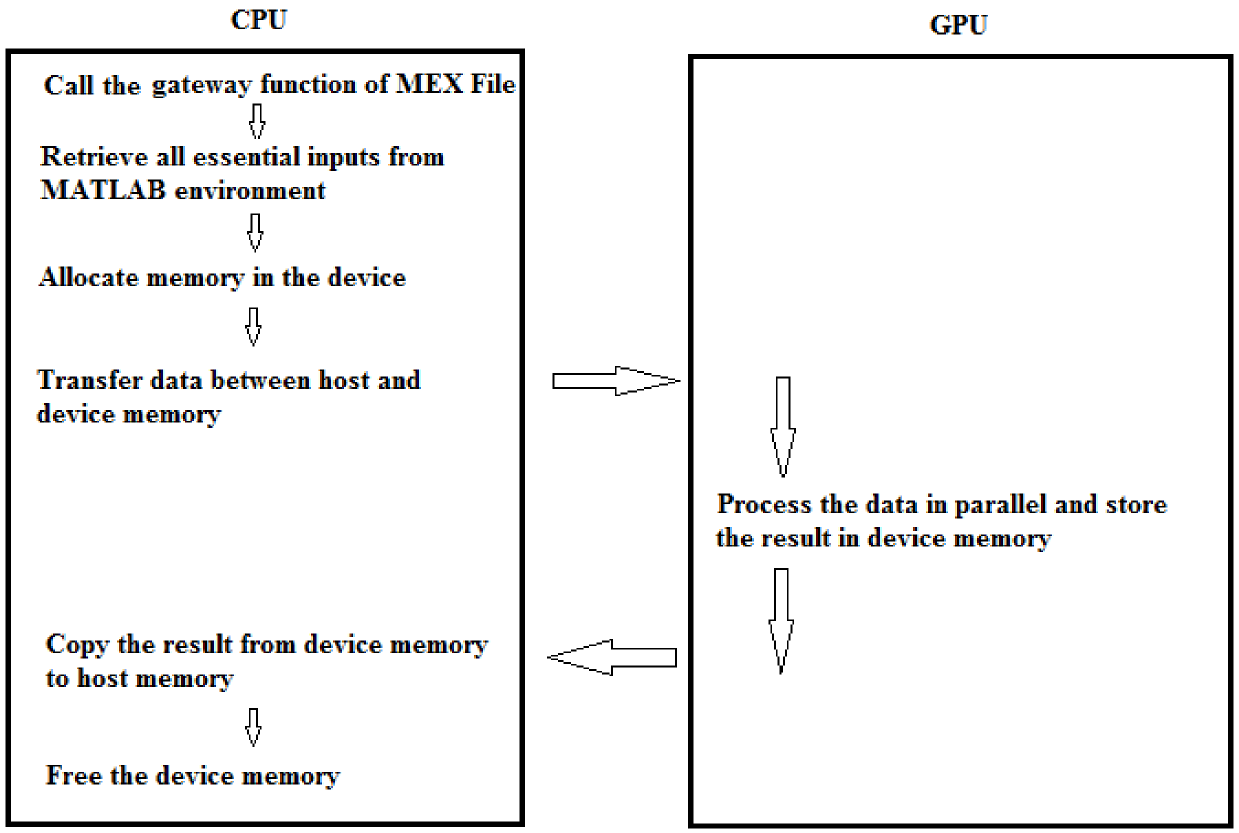

CPU
GPU
Call the gateway function of MEX File
Retrieve all essential inputs from MATLAB environment
Allocate memory in the device
Transfer data between host and device memory
Copy the result from device memory to host memory
Free the device memory
Process the data in parallel and store the result in device memory

Figure 7 Data processing flow for a basic CUDA C MEX File

Lastly, it is important to state two differences between MATLAB and CUDA C MEX Files. First, MATLAB stores data in column major order, whereas C/C++ or CUDA stores or accesses the data row major order. The second difference is the indexing scheme. MATLAB accesses the elements in a matrix using a one-based index. On the other hand, C/C++ accesses a matrix's elements using a zero-based index.

CHAPTER THREE

IMPLEMENTATION OF THE FORWARD MODEL

As we mentioned earlier, the photon propagation in turbid media is modeled by the Radiative Transfer Equation (RTE). However, the resolution of RTE is very complex; hence, it is usually approximated to the diffusion equation. We also mentioned that during the forward problem of DOT, the diffusion equation is solved using analytical or numerical methods. One of the most used numerical methods is called the Finite Element Method (FEM). In this work, we present the solution of the continuous wave (CW) DOT forward problem using FEM.

During the formulation of the CW DOT forward problem in FEM, the photon density Φ, the diffusion coefficient κ, and the absorption coefficient μ_a are spatially discretized by using the Galerkin weak form of the FEM diffusion equation and a Lagrangian basis function. Here, it is important to mention the trivial step in FEM, which consists of the discretization of the studied medium. Basically, the studied domain Ω is discretized or meshed into small elements connected at N nodes. Figure 8 shows an example of the discretization of the domain Ω.

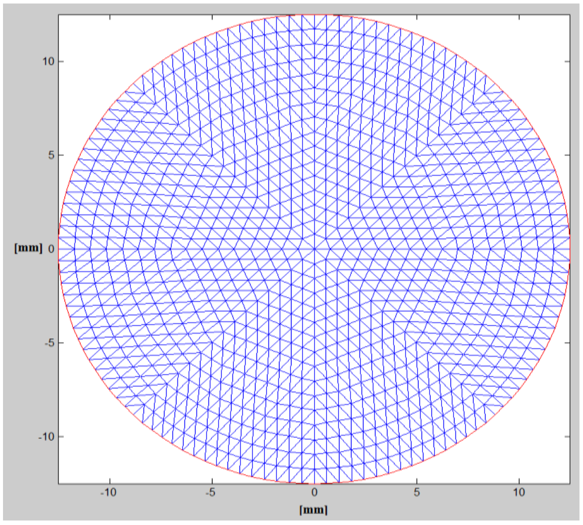


Figure 8 Example of the discretization of the domain Ω using a 2D FEM mesh of a 25 mm diameter medium

In two dimensions, the domain is divided into triangles and its boundary is discretized using segments. However, the domain is usually meshed using tetrahedral elements in 3D, and its boundary is divided into triangles. Therefore, in order to find the photon density Φ, we calculate nodal field photon density values for all the FEM mesh nodes within the domain Ω, that is $\Phi = \{ \phi_i \}$ for all nodes $i=1,2, ..,N$.

After spatially discretizing the diffusion equation, the forward problem can be expressed as a system of linear equations:

$$[A]\,\{\Phi\} = \{S\} \tag{26}$$

where A is the system matrix, Φ is a column vector of size *N*, S is a column vector of size *N*, and *N* is the number of nodes in the finite element mesh. The system matrix A is highly sparse and generally a banded matrix. It gathers the individual contributions from each mesh element characterized by a diffusion coefficient κ and an absorption coefficient μ_a at each of its nodes. Technically, solving the forward problem consists of first solving Eq. 26 to obtain the photon density Φ within the medium then obtaining the photon flux at the boundary.

Before explaining the implementation of the forward model, it is important to mention the 3D FEM mesh generation. In 3D modeling, the first step consists of defining the boundary of the studied medium. This is generally done by segmenting anatomical images of the medium.

Afterwards, the obtained boundary of the medium is transformed into a 3D closed geometry using any Computer-Aided Design (CAD) software. In some cases, the boundary of the 3D geometry requires some refinements and smoothening. Finally, the processed geometry is meshed using any 3D mesh generator software (Comsol, NIRFAST, etc.). Figure 9 shows an example 3D FEM mesh of a 25 mm diameter cylinder.

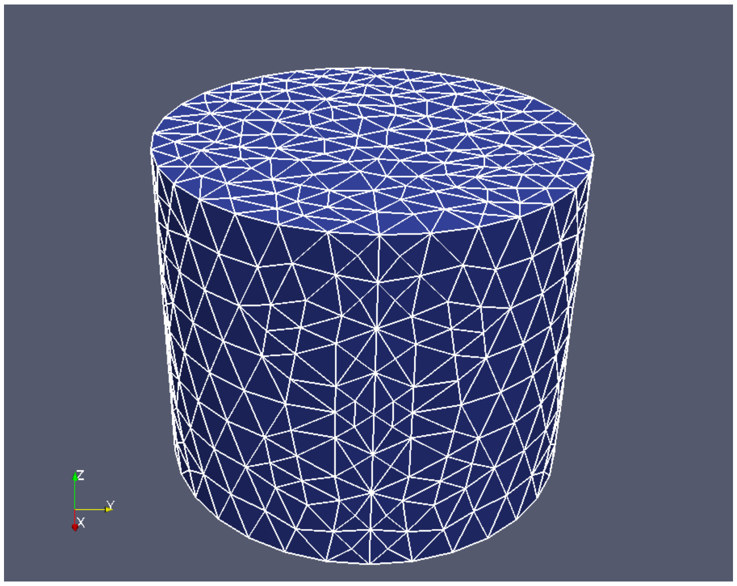


Figure 9 Example of a 3D FEM mesh of a 25 mm diameter cylinder

Mesh generation is a complex and time-consuming procedure in DOT. The computation time and space efficiency of this procedure highly depend on the mesh generation algorithm used. Iterative and recursive based methods, which have different time complexities, are generally used. Intuitively, it can be considered that 3D mesh generation algorithms have higher time and space complexities than the 2D ones; however, this is not always the case since time and space complexities are determined by the method used for mesh generation.

One of the most important factors affecting the computation time is the number of mesh elements. As imposed by the FEM, the medium must be discretized but the final solution has to be close to realistic continuous solution. It may be considered that using a very high number of elements seems to be the answer for this issue; however, the higher the number of elements is the longer the computation time is. Therefore, finding an optimum number of elements is one of the important steps when using FEM.

In this work, we used 3D meshes with different number of elements in order to investigate the relationship between the number of elements and the computation time of the CW DOT forward problem. Specifications of these meshes are given in the simulation and discussion section. Our meshes are characterized using the following notations:

- p (3xN): Each column contains the (x, y, z) 3D coordinates of a given mesh node.
- t (4xN_e): Each column contains the indices of the 4 nodes forming a tetrahedral element and N_e is the total number of the tetrahedral elements.
- e (3xN_b): Each column contains the indices of the 3 nodes forming a boundary triangular element and N_b is the total number of the boundary triangular elements.

Apart from the p, e, and t properties, we choose to add additional information such as the source and detector information into our 3D meshes. The source and detectors are represented by two properties: the node number for each source or each detector, and source or detector coordinates. In addition, information about how diffusion and absorption coefficients for each tetrahedral and boundary element affect the photon density is also included to our 3D meshes. This information is used to obtain the system matrix A.

3.1. MATLAB Implementation of the Forward Model

Earlier, we introduced that CW DOT forward problem involves in a system of linear equations, which can be represented in a matrix formulation. Here we present a processing flow of its implementation in MATLAB, which is a very strong software designed for matrix computations.

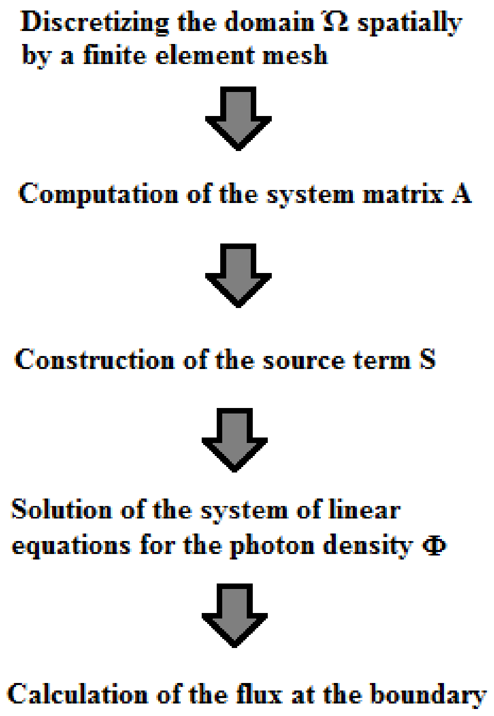

Discretizing the domain Ω spatially by a finite element mesh
Computation of the system matrix A
Construction of the source term S
Solution of the system of linear equations for the photon density Φ
Calculation of the flux at the boundary

Figure 10 Data processing flow of the forward model in MATLAB

After the mesh generation, the FEM forward problem solver implemented in MATLAB attributes the initial estimates of the optical properties of the studied medium to each mesh element. Once this initialization performed, the FEM forward solver calculates the first term of the left hand side of the system of linear equations, which is the system matrix A. This system matrix is assembled by considering the effect of each tetrahedral element on the neighboring tetrahedrons. Also, during the system matrix assembly, the boundary conditions of the diffusion equation are applied to the boundary triangular elements.

The next step is to initialize the vector S. In FEM, this vector is an Nx1 vector containing zeros everywhere except at the nodes to be used as sources. Usually, the values of vector S at these nodes are set to 1.

The photon density Φ is simply obtained by solving the system of linear equations. Once the photon density obtained, the boundary conditions are applied to find the photon flux at the detectors positioned at the boundary of the medium.

3.2. CUDA C Implementation

Although the processing flow of the CUDA C implementation for the forward model shown in Fig. 11 has some differences from the MATLAB implementation, the CUDA C implementation has the same core processing steps followed in the MATLAB version. We need to generate a Finite Element mesh to run the CUDA C implementation, and mesh generation and its features are the same as previously mentioned.

To run the CUDA C implementation of the forward model, we begin with the initialization of absorption and scattering coefficients, and calculation of the diffusion coefficient; then we import mesh features such as node coordinates, node numbers of tetrahedrons and triangles at the boundary, as well as source and detector information into the MATLAB environment. After that, we make the function call of the forward model's MEX File with the necessary parameters. The CUDA C implementation of the forward model calculates the photon density Φ and the flux at the boundary; and the MEX File function returns these two parameters into the MATLAB environment.

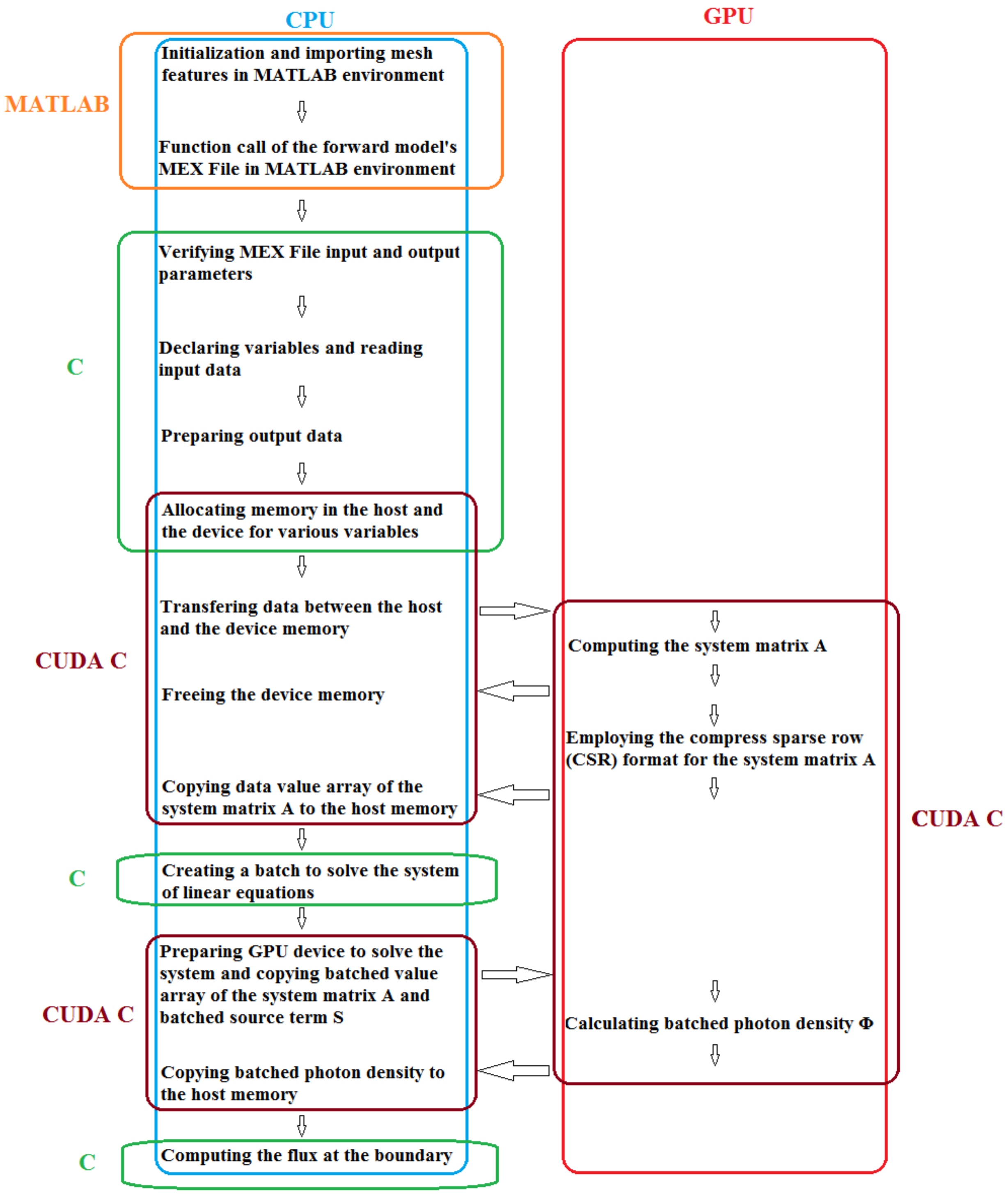

CPU
GPU
MATLAB
Initialization and importing mesh features in MATLAB environment
Function call of the forward model's MEX File in MATLAB environment
C
Verifying MEX File input and output parameters
Declaring variables and reading input data
Preparing output data
Allocating memory in the host and the device for various variables
CUDA C
Transfering data between the host and the device memory
Freeing the device memory
Copying data value array of the system matrix A to the host memory
Computing the system matrix A
Employing the compress sparse row (CSR) format for the system matrix A
CUDA C
C
Creating a batch to solve the system of linear equations
CUDA C
Preparing GPU device to solve the system and copying batched value array of the system matrix A and batched source term S
Copying batched photon density to the host memory
Calculating batched photon density Φ
C
Computing the flux at the boundary

Figure 11 Data processing flow of CUDA C implementation for the forward model

The CUDA C implementation of the forward model started with the gateway function of MEX Files: *mexFunction().* As stated earlier, the *mexFunction()* has four input parameters: array of input arguments, *prhs*; the size of *prhs* array, *nrhs*; array of output arguments, *plhs*; and the size of *plhs* array, *nlhs*. We need to verify the MEX File input and output parameters. To do that, we use *nrhs* and *nlhs* parameters. If the number of right hand side arguments, i.e. the number of input arguments *nrhs*, does not match with the total number of the necessary arguments to calculate the photon density, the MEX File gives an error message to indicate wrong number of input arguments included. In addition, if the number of left hand side argument, i.e. the number of output arguments *nlhs*, is different than 2, the MEX File shows that 2 output arguments are required.

The next step is to declare the variables and to read the input data. We define the same variables used in the MATLAB environment. To assign input arguments coming from the MATLAB environment, we use MEX functions that read the input data. There are various types of input data in the implementation; therefore, different MEX functions are used to read different types of data. For example, the absorption coefficient μ_a is a double variable; hence, the MEX function *mxGetPr* are utilized for the absorption coefficient to assign the input data. However, the variable to store nodes of tetrahedrons is an integer type; therefore, the MEX function *mxGetData* are used for that variable. In addition, the scalar inputs such as number of nodes, number of tetrahedrons, and number of triangles at the boundary are assigned by the MEX function *mxGetScalar*.

After reading the input data, we need to create the output arguments, the photon density Φ for all sources and the flux at the boundary. The first step is to calculate the size of the photon density matrix and the flux matrix. Then, the MEX function *mxCreateNumericMatrix* is used to create these output matrices with the arguments: number of rows, number of columns, identifier for the class of the array and the complexity flag. In the implementation, *mxDOUBLE_CLASS* and *mxREAL* are the class identifier and the complexity flag, respectively; since double precision in the output data is set for the continuous wave (CW) method used in these calculations. If we want to switch single precision, we can change the class identifier to *mxSINGLE_CLASS*. Furthermore, for frequency-domain calculations, the complexity flag can be changed to *mxCOMPLEX*. After creating the output matrices, the MEX function *mxGetPr* are used to assign them to the array of output arguments *plhs*.

At this point, the input arguments are read and our output arguments are prepared; so, we are ready to compute the system matrix A. Since the system matrix A is calculated on the GPU device, we firstly declared our GPU device variables and allocated some of the device memory for them by using the function *cudaMalloc()*. Then, the data in the host CPU is copied to the device GPU with the function *cudaMemcpy()*. The syntaxes for the functions *cudaMalloc()* and *cudaMemcpy()* are mentioned in the previous chapter. At this point, we have everything we need to compute the system matrix A in the GPU device. The kernel is then invoked with the declared variables in the device and the system matrix A is obtained. Here, there are three important concepts to mention. The first one is the difference between MATLAB's and C's indexing schemes. As stated earlier, MATLAB accesses elements in an array using a one-based index and C accesses them using a zero-based index. In the implementation, we utilized the node numbers of the tetrahedrons and the triangles at the boundary as the arrays' indices. The node numbers for both the tetrahedrons and the triangles at the boundary start with one. Since MATLAB has a one-based index scheme, the indexing is correct for arrays using indices that come from the node numbers starting with one. However, this is not the case for the arrays in C, since C has a zero-based index scheme. Therefore, we subtracted one from indices coming from the node numbers starting with one. The second one is the existence of atomic operations in computing the system matrix A. Same memory locations are read, modified and written by multiple threads at the same time when calculating some of the elements of the system matrix A. In this case, the behavior is undefined and the result is wrong. We use an atomic operation, namely *atomicAdd(),* to prevent this issue. The operation *atomicAdd()* reads the data for a node at some address in the device memory, adds the calculated value to update the data for that node, and writes the result back to the same address. The operation is guaranteed not to be interfered by any other thread that will update the data for the same node until the operation is complete. Therefore, calculations of some elements of the system matrix A can be time-consuming when compared to computing the other elements. CUDA Toolkit Documentation provides an implementation of the operation *atomicAdd()* for double-precision floating-point as follows:

Table 5 An implementation of the operation *atomicAdd()* for double precision floating point provided by CUDA Toolkit Documentation

```
__device__ double atomicAdd(double* address, double val){
   unsigned long long int* address_as_ull = (unsigned long long int*) address;
   unsigned long long int old = *address_as_ull, assumed;
   do {
       assumed = old;
       old = atomicCAS(address_as_ull, assumed, __double_as_longlong(val +
                              __longlong_as_double(assumed)));
   // Note: uses integer comparison to avoid hang in case of NaN (since NaN != NaN)
   } while (assumed != old);
   return __longlong_as_double(old);
}
```

The last important step for computing the system matrix A is the index calculations for 1D arrays converted from 2D arrays. The CUDA programming language enables to use 2D arrays; however, 1D arrays can be preferred due to their simplicity. In this case, indices for 1D arrays are needed to be calculated. If an element of 2D array with m rows and n columns is defined as *my2D [i][j],* its 1D version is *my1D [i * m + j]*.

The system matrix A is highly sparse *N* by *N* matrix, where *N* is the total number of nodes. To store the matrix A, we employ the compress sparse row (CSR) format. In the CUDA programming model, CSR format of an m by n matrix K is stored by 4 parameters:

- csrValK: The pointer to data value array of nonzero elements of the matrix in row major format
- csrRowPtrK: The pointer to the array that holds indices of the first nonzero element in the *i*th row for *i*=1,2, ... m. The length of this array is m+1, where the last element is nnz + csrRowPtrK(0).
- csrColIndK: The pointer to the array that holds column indices of the nonzero elements of the matrix.
- nnz: The total number of nonzero elements in the matrix.

The CSR format of the matrix K shown below is as follows:

$$K = \begin{bmatrix} 2 & 0 & 0 & 1 \\ 0 & 0 & 0 & 4 \\ 5 & 0 & 7 & 0 \\ 0 & 0 & 6 & 0 \end{bmatrix}$$

csrValK = [2 1 4 5 7 6]

csrRowPtrK = [0 2 3 5]

csrColIndK = [0 3 3 0 2 2]

To convert the system matrix A from its dense format to its CSR format, we use *cuSPARSE API* which is provided by NVIDIA and contains library routines of basic linear algebra for sparse matrices. Firstly, the *cuSPARSE* library is initialized and a handle on the *cuSPARSE* context is created by calling the *cuSPARSE* helper function *cusparseCreate()*. Hardware resources necessary for accessing the GPU device is allocated owing to this function. The *cuSPARSE* helper function *cusparseCreateMatDescr()* is called to initialize the matrix descriptor and *cusparseSetMatType()* is used to set the matrix type field of the matrix descriptor. In addition, the helper function *cusparseSetMatIndexBase* is called to set the index base field of the matrix descriptor. For conversion of the system matrix A from dense to CSR format, we use the *cuSPARSE* format conversion function *cusparseDdense2csr()*. This function has 10 parameters which need to be allocated before calling it. These parameters are as follows:

- handle: handle to the *cuSPARSE* library context
- m: the number of rows of matrix A
- n: the number of columns of matrix A
- descrA: the matrix descriptor for A
- A: the matrix A
- lda: the leading dimension of the matrix A
- nnzPerRow: the array containing the number of nonzero elements per row
- csrValA: the data value array of nonzero elements of the matrix
- csrRowPtrA: the array that holds the indices of the first nonzero element in the *i*th row for *i*=1,2, ... m
- csrColIndA: the array that holds the column indices of the nonzero elements of the matrix A.

We already have handle, descrA, m, n, lda, A. Next, we need to find the nnzPerRow parameter. The *cuSPARSE* function *cusparseDnnz* is used in the CUDA C implementation. In addition to the array containing the number of nonzero elements per row, the function *cusparseDnnz* gives the total number of nonzero elements. After the conversion, we obtain the arrays csrValA, csrRowPtrA, and csrColIndA.

The next step is to solve the system of linear equations. We create a batch of the system matrix A in CSR format for all sources to further parallelize the system solver. Therefore, we can solve the all linear equations simultaneously instead of solving them one by one.

$$\begin{bmatrix} A \\ \vdots \\ A \end{bmatrix} \begin{bmatrix} \Phi_1 \\ \vdots \\ \Phi_n \end{bmatrix} = \begin{bmatrix} s_1 \\ \vdots \\ s_n \end{bmatrix} \tag{27}$$

where n is the number of sources, s_i is the source vector for the *i*th source and Φ_i is the photon density vector for *i*th source.

To implement a solver for the system of linear equation, we use the *cuSOLVER API* which is provided by NVIDIA and contains library routines for dense and sparse matrix factorization and system solver. The library function to solve our batched system is *cusolverSpDcsrqrsvBatched(),* which is a low level function of *cuSolverSP*. This function solves a batched system shown below by using QR factorization.

$$A_j \times X_j = B_j \tag{28}$$

where each A$_j$ is a m×n sparse matrix in CSR storage format with the arrays csrValA, csrRowPtrA and csrColIndA, and j = 1, 2,..., batchSize. Each B$_j$ (mx1) and X$_j$ (nx1) are the right-hand-side and the solution vectors, respectively. QR factorization is basically the decomposition of a matrix, in this case the system matrix A, into an orthogonal matrix Q, and upper triangular matrix R.

$$A = Q\,R \tag{29}$$

There are two prerequisite to use the batched sparse QR:

1) All matrices A$_j$ must have the same sparsity pattern. For our case, they are same matrices; hence, they have same sparsity pattern.

2) The solution is valid only if A$_j$ is of full rank for all j = 1, 2, ..., batchSize, which is also satisfied for our case.

The implementation of the *cuSolverSP* function *cusolverSpDcsrqrsvBatched()* starts with creating a handle on the *cuSolverSP* context and initialization by *cusolverSpCreate()*. Then, the matrix descriptor is created, and the matrix type and the matrix index base are set. We also need to create an empty info structure by using the function *cusolverSpCreateCsrqrInfo()* with a parameter of data type *csrqrInfo*. The opaque data structure *csrqrInfo* keeps the intermediate data such as the orthogonal matrix Q and the upper triangular matrix R of QR factorization in the batched QR.

In addition, an analysis phase by the function *cusolverSpXcsrqrAnalysisBatched()* is implemented before solving the batched system. This function is used to obtain sparsity pattern of the orthogonal matrix Q and the upper triangular matrix R of QR factorization. Furthermore, the function provides the information for the working space to perform QR.

At this point, everything to solve the batched system is ready; so the function *cusolverSpDcsrqrsvBatched()* is called with the following parameters:

- handle: handle to the *cuSolverSP* library context
- m: the number of rows of each matrix A_j
- n: the number of columns of each matrix A_j
- nnz: The total number of nonzero elements in each matrix A_j.
- descrA: the matrix descriptor for each A_j
- csrValBatchedA: the data value array of nonzero elements of the batched system
- csrRowPtrA: the array that holds indices of the first nonzero element in the *i*th row for i = 1, 2, ..., m
- csrColIndA: the array that holds column indices of the nonzero elements of the matrix A.
- B: the batched source term S
- x: the batched solution returned
- batchSize: number of equations to be solved
- info: the opaque structure for QR factorization
- pBuffer: the buffer allocated by the user

Although it isn't required for our implementation, there is an important detail that needs to be addressed. It is possible that the device memory is not big enough to solve the batched system in one function call; therefore, it is necessary to choose a proper batch size. If the size of the batched system is too big, the calculation needs to be divided into parts; hence the function *cusolverSpDcsrqrsvBatched()* is called more than once.

After solving the batched system, we obtain an array of the photon densities for all sources. The photon density array is copied from the device memory to the host memory. This is the first output of our implementation. Furthermore, the second output, the flux at the boundary, needs to be calculated. To do that, we firstly calculate the boundary factor. Then, the flux at the boundary is computed using the boundary factor and the detector information defined in the mesh.

CHAPTER FOUR

SIMULATION RESULTS AND DISCUSSION

Simulation studies are performed to evaluate the performance of the forward model that is implemented using CUDA C language, which is a GPU parallelized application. Our main aim is to compare the computational accuracy and efficiency of our GPU-based forward solver with the CPU-based conventional forward solver developed in our lab using MATLAB.

Simulations are done on a computer that has 2.67 GHz. Intel Core i7 CPU – 920 and Windows 7 64-bit operating system. Our GPU card is a NVIDIA GeForce GTX 960 with 5.2 compute capability. GeForce GTX 960 GPU card has 1024 CUDA cores and 2 GB memory. Its memory clock is 7.0 Gbps and memory bandwidth is 112 GB/sec.

3D Finite Element meshes are used in the simulations. To make a good comparison between CPU and GPU, we use several 3D meshes with different number of nodes, tetrahedral elements and triangular elements at the boundary. However, dimensions of our 3D cylindrical meshes are always kept the same. The diameter and the height of the meshes are 25 mm and 20mm, respectively. The origin of the coordinate system is chosen at the center of cylindrical mesh for illustration purposes. Other features of our 3D meshes, namely the number of nodes as well as the number of tetrahedral and triangular elements, are given in Table 6.

Table 6 The features of the 3D meshes used in the simulation studies such as number of nodes, tetrahedral elements and triangular elements at the boundary

3D Meshes	Number of Nodes	Number of tetrahedral elements	Number of triangular elements at the boundary
1	2316	11393	1552
2	3370	16938	2068
3	4335	22058	2484
4	5299	27505	2680
5	6394	33437	3096
6	8359	44219	3736
7	9339	49806	3928

In addition to different meshes, we also investigate the effect of the number of sources during the simulations. The number of the sources is varied between 2 to 26. The maximum number of sources is determined by the memory need of the batched system solver explained in the previous chapter. If the GPU device memory is not large enough, it cannot handle the application for all sources. As mentioned earlier, it is possible to divide the solution of the system of linear equations into parts if there are a large number of sources, which makes it difficult to process them in one function call. However, we haven't utilized this approach during these simulations. In addition to this, the maximum number of the sources is limited by the 3D mesh used in the simulations. For example, maximum number of the sources is 26 for the mesh with 4335 nodes; on the other hand, it is only 4 for the mesh with 9339 nodes and it is much higher than 26 for the mesh with 3370 nodes. Apart from the sources, there are 16 detectors added into our 3D meshes. Both sources and detectors are placed on the z = 0 plane and they are equally spaced. Although 16 detectors and different number of sources are utilized in the simulations, Figure 12 shows only eight detectors and eight sources for visualization purposes.

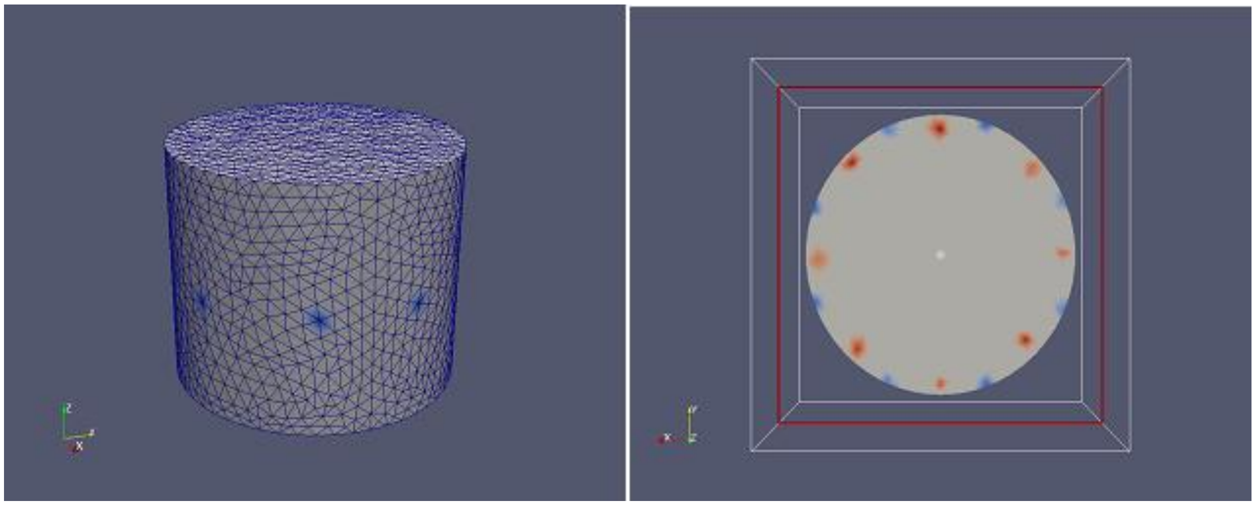


Figure 12 An example of source and detector placement in a 3D mesh and its corresponding 2D slice at z=0.

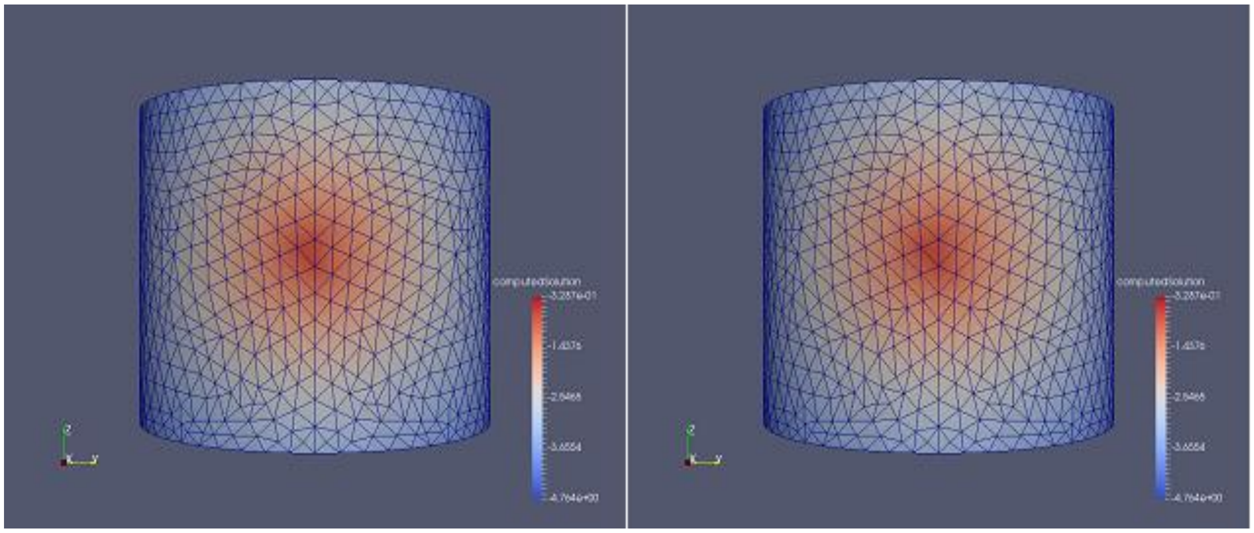
In the simulations, the absorption coefficient μ_a is set to 0.01 mm^{-1} and the scattering coefficient μ_s is set to 0.8 mm^{-1} to mimic tissue. Furthermore, since we use continuous wave (CW) measurement technique, the frequency ω is set to 0.

In the following sections, computational accuracy and efficiency of CUDA C implementation of the forward model will be presented. In addition, the simulation results will be discussed in various aspects.

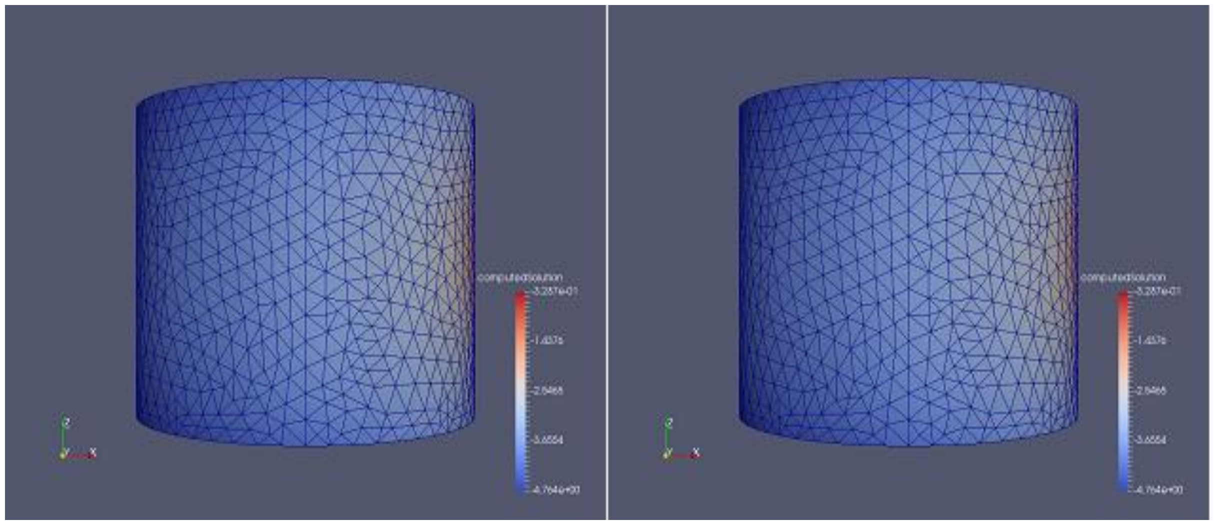
4.1. Computational Accuracy of CUDA C Implementation

Taking the advantage of parallelism and reducing simulation time are the main motivation behind GPU-based parallelization. However, besides time, accuracy of the algorithm is an important factor to consider. To test computational accuracy of our CUDA C implementation, we firstly calculate the photon density of our 3D homogeneous finite element mesh for one source with our new GPU-based forward solver. Our 3D mesh has 9339 nodes, 49806 tetrahedral elements and 3928 triangular elements at the boundary. The source is placed to near the point (12.5, 0, 0). Then, the simulation is carried out with the same parameters but using the conventional CPU-based forward solver.

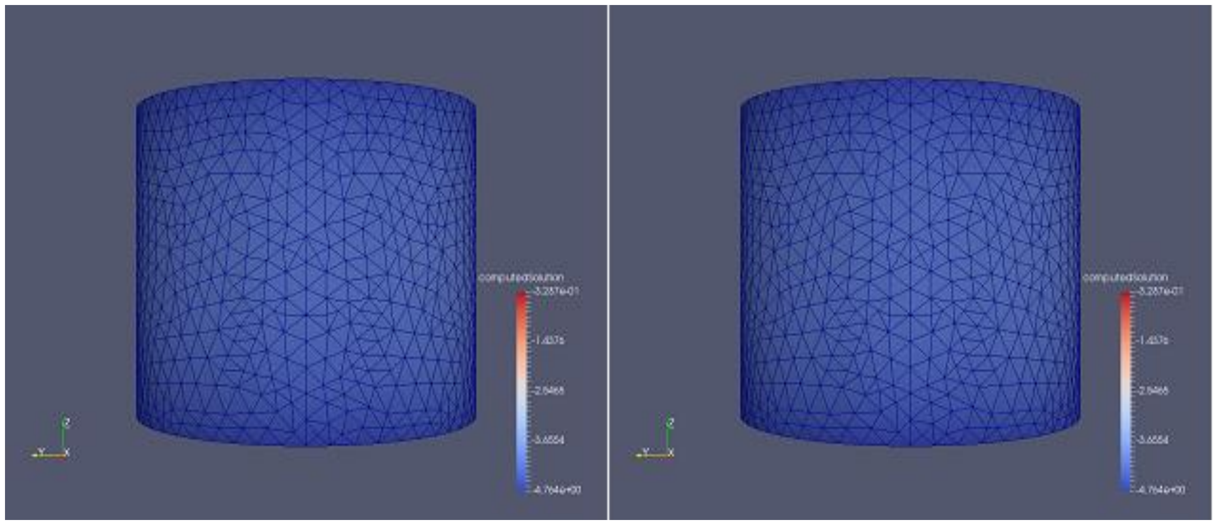
For visualization purposes, ParaView 5.0.0 is used to generate the figures below. The calculated photon density is presented in log scale format due to the vast dynamic range. This also better represent the diffuse nature of photon propagation in the medium. The results of the new CUDA C implementation and conventional MATLAB solver are given on the left and right hand of the Figure 13, respectively. In addition to views from 4 different angles, cross-sectional images of x-y, x-z planes are provided.



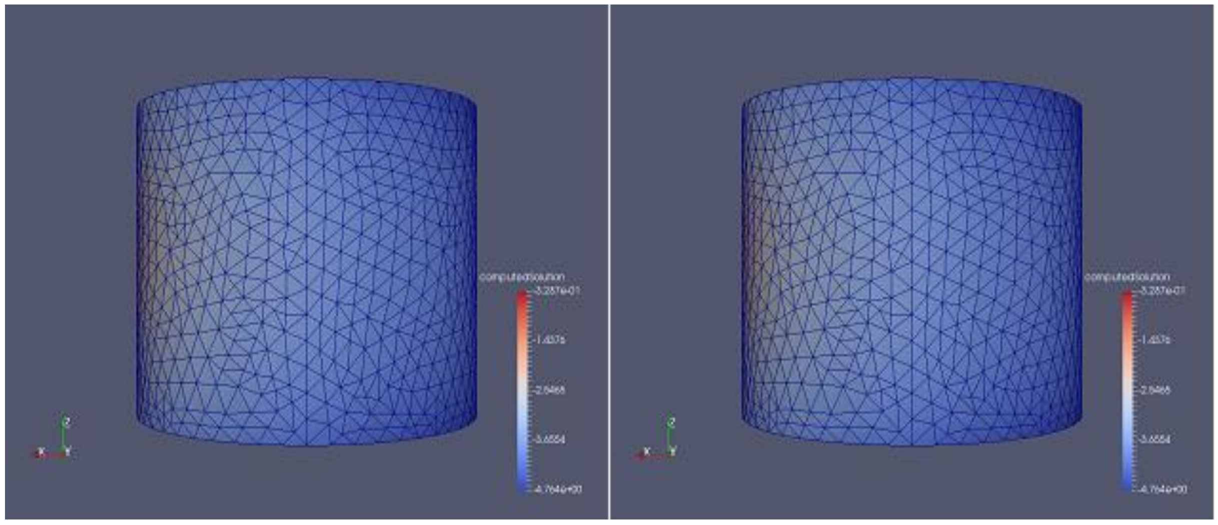
(a)


computedSolution
-3.257e-01
-1.4376
-2.5465
-3.6554
-4.764e+00
Z
Y
X

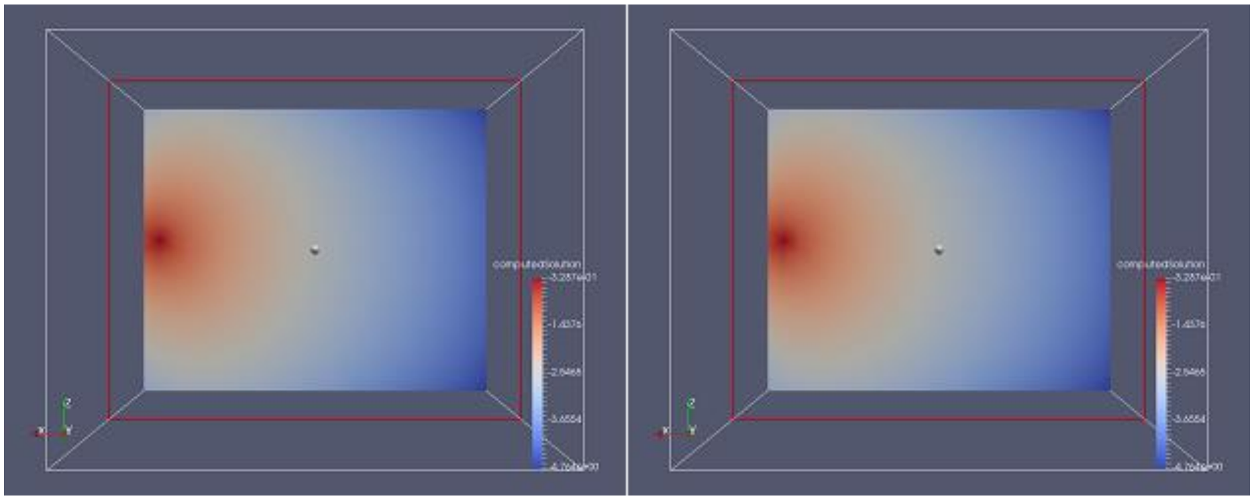
(b)


computedSolution
-3.257e-01
-1.4376
-2.5465
-3.6554
-4.764e+00
Z
Y
X

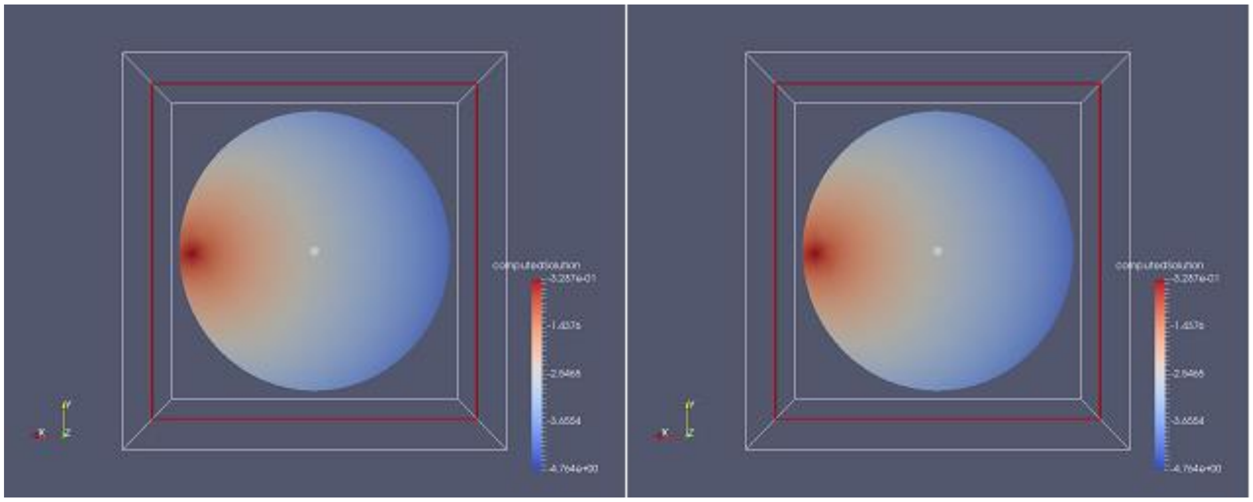
(c)


computedSolution
-3.257e-01
-1.4376
-2.5465
-3.6554
-4.764e+00
Z
X
Y

(d)



(e)



(f)

Figure 13 Simulation results of photon density in logarithmic scale
CUDA C and MATLAB implementation results are given on the left and right side, respectively
(a) The view from +x direction, (b) The view from –y direction, (c) The view from –x direction, (d) The view from +y direction, (e) 2D slice with a normal in y direction, (f 2D slice with a normal in z direction

As the figures above show nice diffusive patterns, it is important to obtain the photon density profile on a line between two points: (12.5, 0, 0) and (-12.5, 0, 0). Figure 14 and 15 show the profiles along this line for GPU and CPU implementations in standard and log scale.

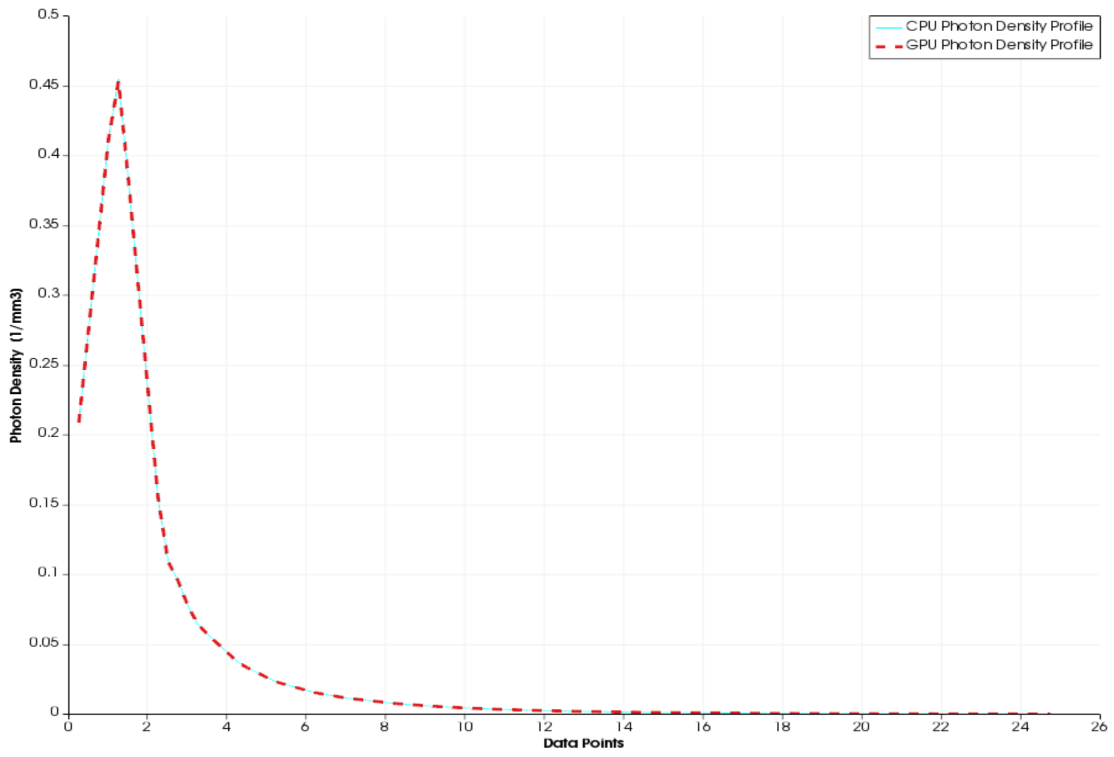


Figure 14 Photon density profiles of CUDA C and MATLAB implementations over a line starting from the point (12.5, 0, 0) to the point (-12.5, 0, 0) in standard scale

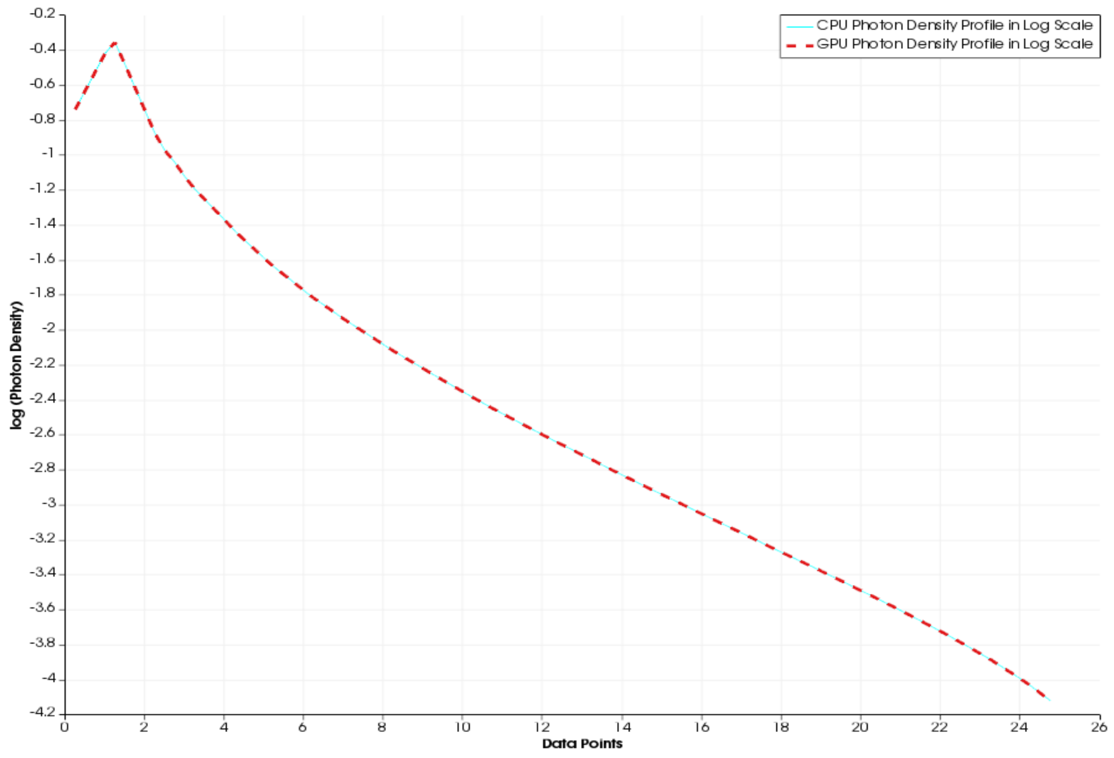


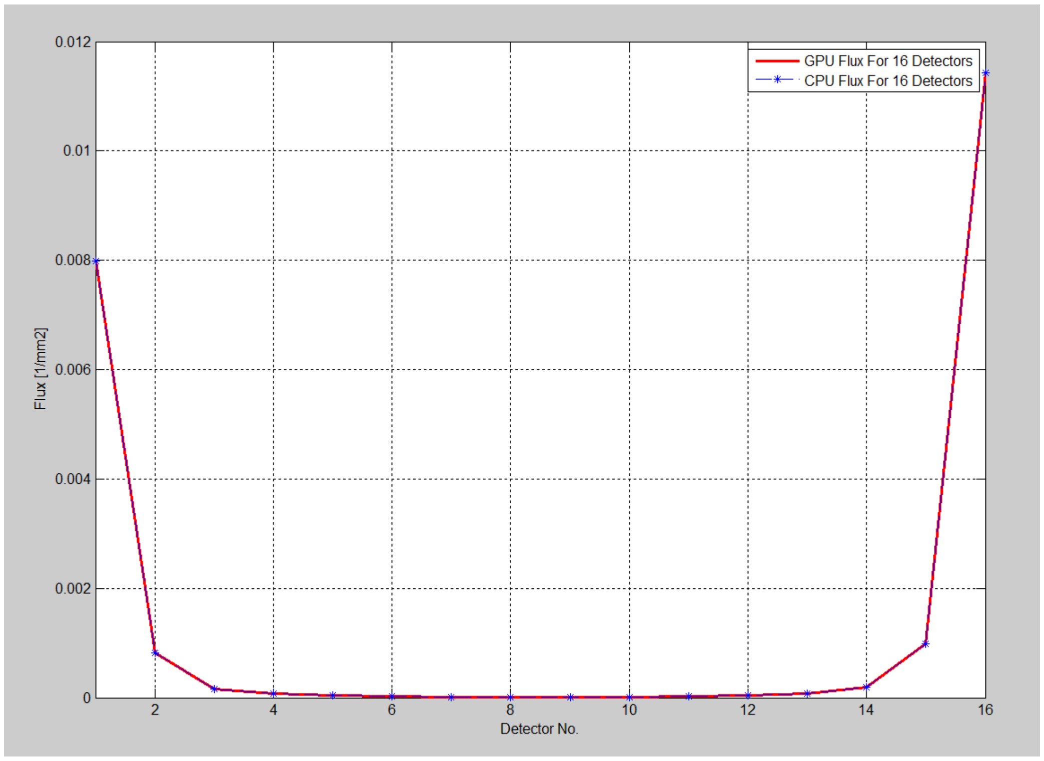
Figure 15 Photon density profiles of CUDA C and MATLAB implementations over a line starting from the point (12.5, 0, 0) to the point (-12.5, 0, 0) in logarithmic scale

As expected, the photon density profile decreases exponentially with depth. Both photon density profiles obtained by CPU- and GPU-based solvers match exactly with each other in both standard and log scale. As seen from figures, the photon density profile is not a perfectly smooth function. There are two reasons for this behavior. The first one is that elements of our 3D meshes are not perfectly shaped and aligned. The second reason is that the program interpolates the data points to show the profile. However, the important point is that photon density profiles that belong to both GPU and CPU version of the solver show the same pattern.

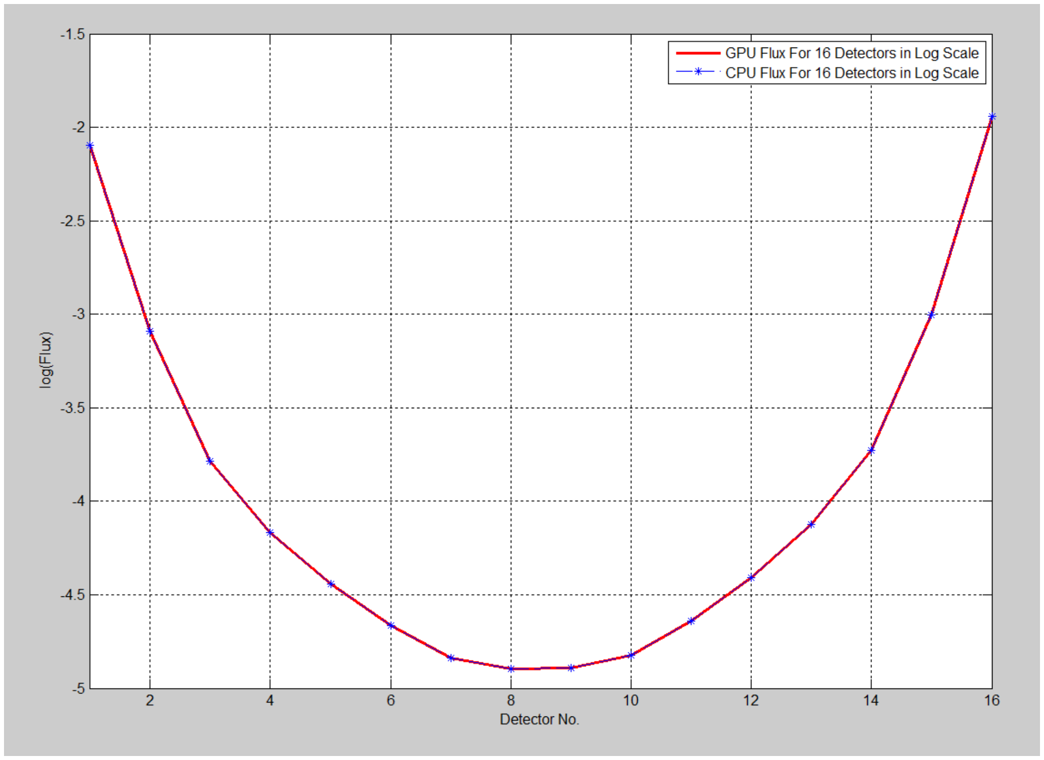
To analyze the difference between GPU- and CPU-based simulation results, we utilize root-mean-square error, or RMS error. The RMS error can be found by using the following formula:

$$RMS\ Error = \sqrt{\frac{\sum_{i=1}^{n}\left(\Phi_{GPU,i}-\Phi_{CPU,i}\right)^2}{n}} \qquad (30)$$

The RMS error is found 4.9 x 10^{-17} for the photon density of the 3D finite element mesh with 9339 nodes. As the final step of testing computational accuracy of our new CUDA C implementation of the forward solver, we compare its flux at the boundary with the conventional MATLAB implementation. As mentioned earlier, there are 16 detectors at the boundary which are located on the z = 0 plane. Figure 16 shows results for GPU- and CPU-based implementations and their comparison in both standard and log scale.



(a)



(b)

Figure 16 Results of the flux at the boundary which is measured by 16 equally distant detectors (a) The flux at the boundary obtained by the CUDA C and MATLAB implementations in standard scale, (b) The flux at the boundary obtained by the CUDA C and MATLAB implementations in logarithmic scale

Just as the photon density, we also calculate the RMS error between two implementations for the flux calculated at the boundary. The RMS error is 5.7 x 10^{-14} for the flux at the boundary.

The main reason of these RMS errors is the fact that calculations of floating points are not guaranteed to give identical results for different hardware. However, we can say that the RMS errors for both the photon density and the flux at the boundary are very low. The minimum value is around 10^{-5} for both calculated photon density and flux. Since the RMS errors for the photon density and the flux at the boundary are around 10^{-17} and 10^{-14} respectively, we can conclude that the GPU-based forward model works with high accuracy.

Before going into computational efficiency, there is one important issue left to be addressed. This issue is the precision level of the implementation. In our new CUDA C implementation, we use double precision floating points for comparison purposes; however, it is also possible to use single precision floating points. Using single precision will provide higher speed compared to using double precision.

4.2. Computational Efficiency of CUDA C Implementation

High computational efficiency is the main goal of the GPU parallelism; therefore, it is important to evaluate the computational efficiency of our CUDA C implementation. To test the computational efficiency, we conduct several simulations using various meshes with different number of nodes and elements. These simulations give us the relationship between the number of nodes in a mesh and speed-up of our new implementation over conventional one. In addition, we want to know how the number of sources affects the speed-up; hence we make additional simulations using different number of sources.

For the first part of testing computational efficiency of our CUDA C implementation, we use four sources with different types of 3D meshes, which have different features such as number of nodes, tetrahedral elements, and triangular elements at the boundary explained in the previous section. The Figure 17 and 18 show our simulation results for GPU- and CPU-based implementation.

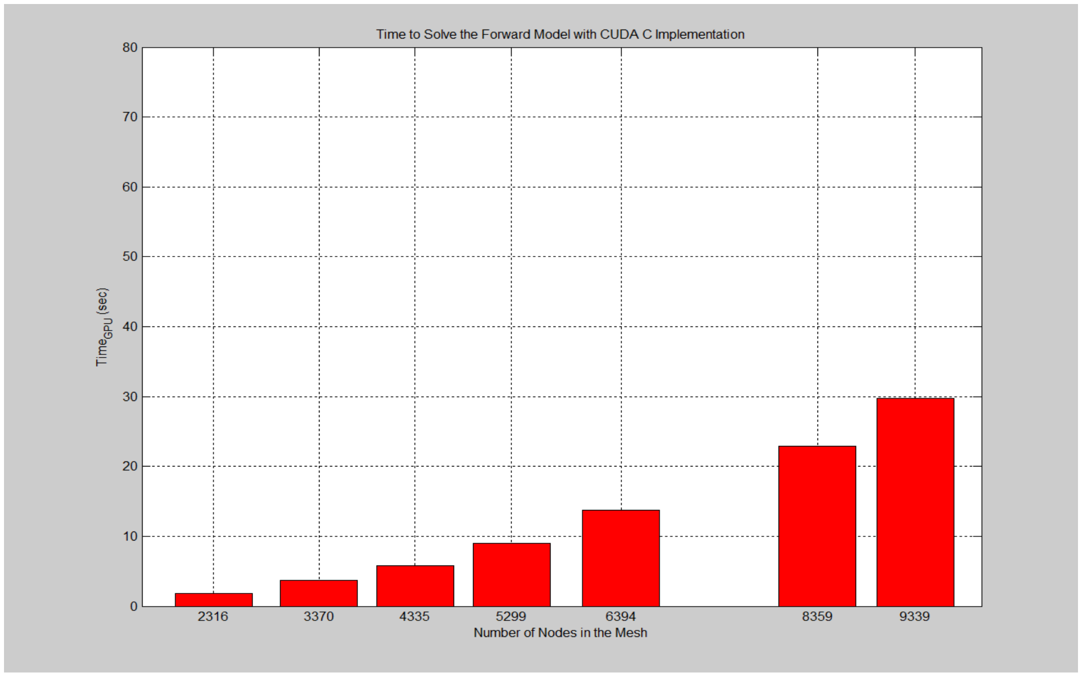


Figure 17 Simulation time of CUDA C implementation for meshes with different number of nodes

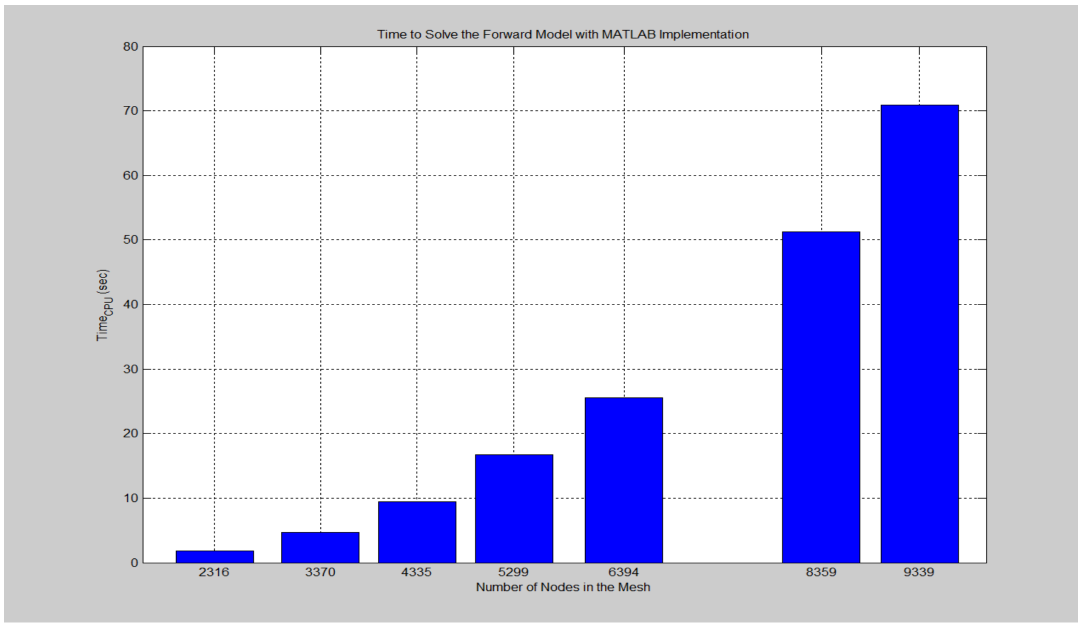
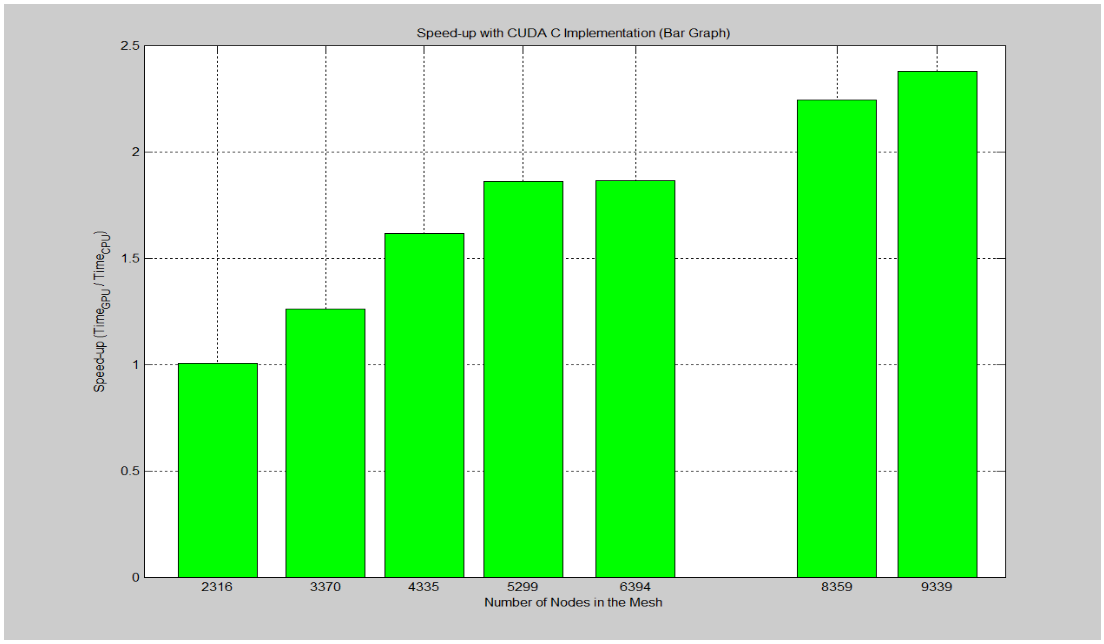
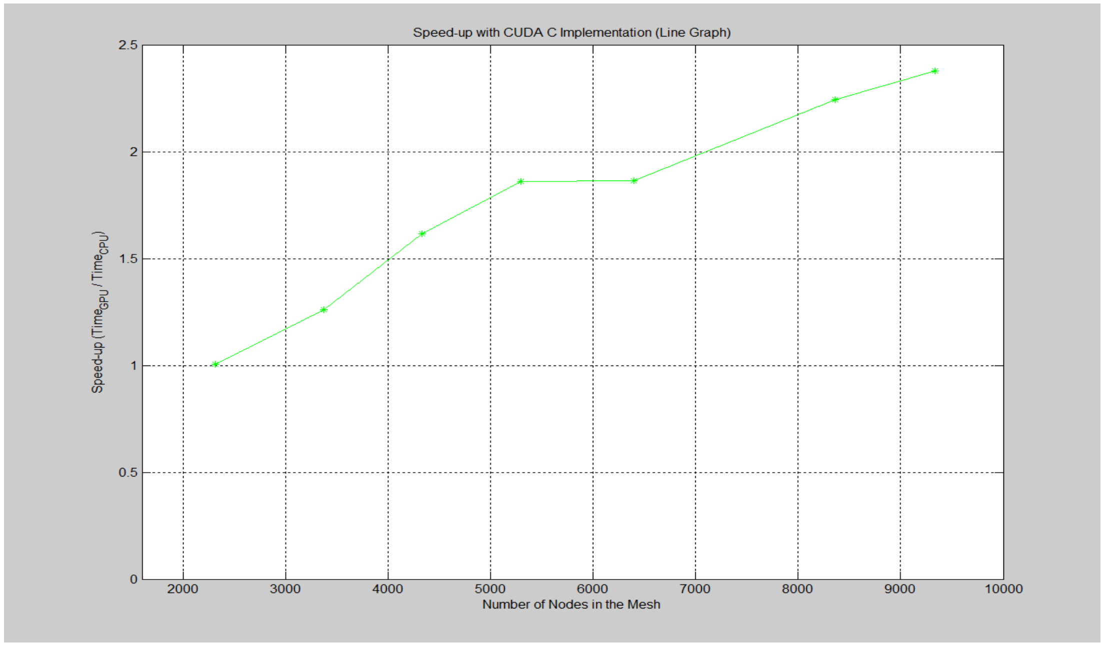


Figure 18 Simulation time of MATLAB implementation for meshes with different number of nodes

Using the time results from previous figures, we calculate the speed-up of using CUDA C implementation over MATLAB implementation. The speed-up bar and line graphs are given in Figure 19. For 3D meshes that have small number of nodes, there is a little speed-up; however, as the number of nodes in a mesh increases, the speed-up becomes significant. Here, it is important to mention that reasons of low speed-up for the small number of nodes are mainly high data transfer time between CPU and GPU and high memory allocation time. Unfortunately, we could not use a 3D mesh with very large number of nodes in our simulations because there is no enough memory in our computer to process that amount of data. Therefore, maximum observable speed-up for our simulations is near 2.5, which means the GPU application is approximately 2.5 times faster than the CPU application for a 3D mesh with 9339 nodes.



(a)



(b)

Figure 19 The speed-up graphs with CUDA C implementation for 3D meshes with different number of nodes (a) The bar graph of the speed-up and (b) The line graph of the speed-up

For the second part of the computational efficiency test, we use different number of sources with our 3D mesh of 4335 nodes. Figures 20 and 21 show our simulation results for CUDA C and MATLAB implementations.

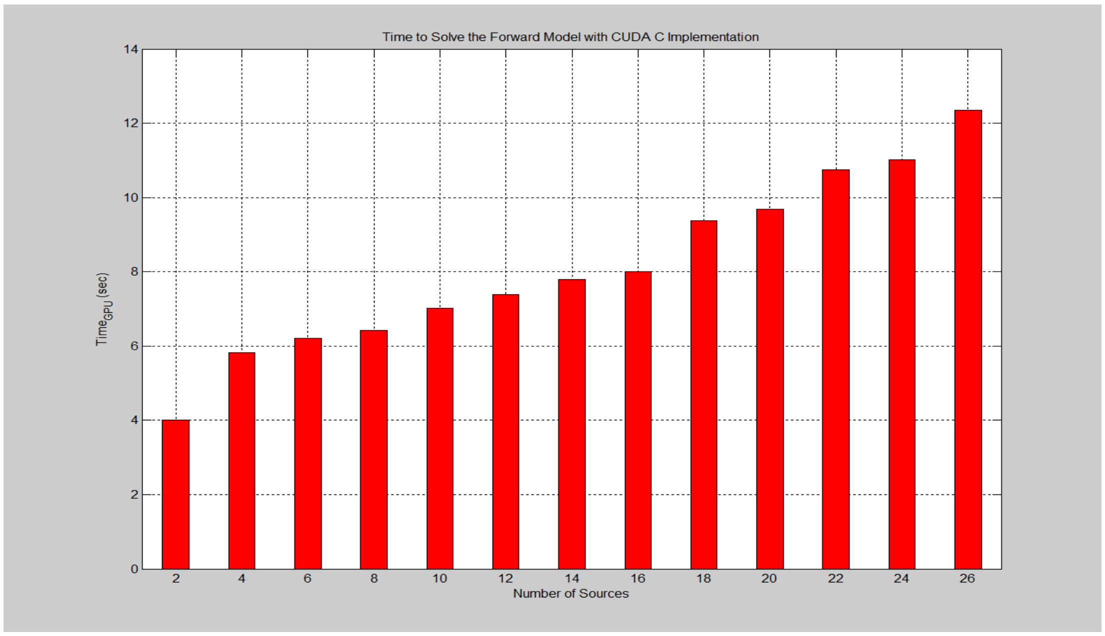


Figure 20 Simulation time of CUDA C implementation for different number of sources

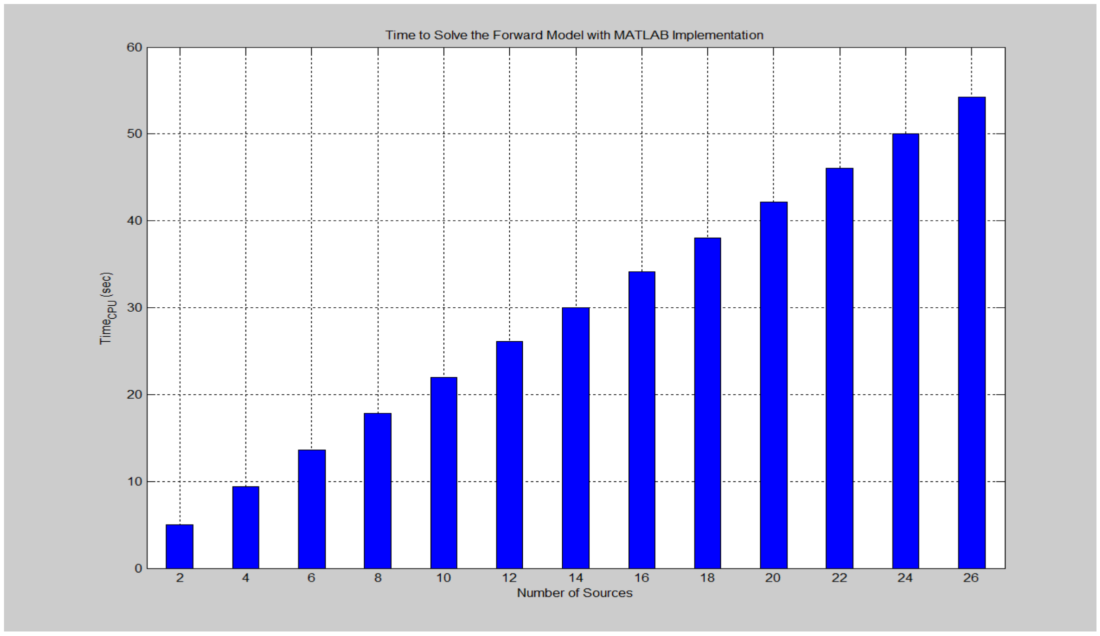
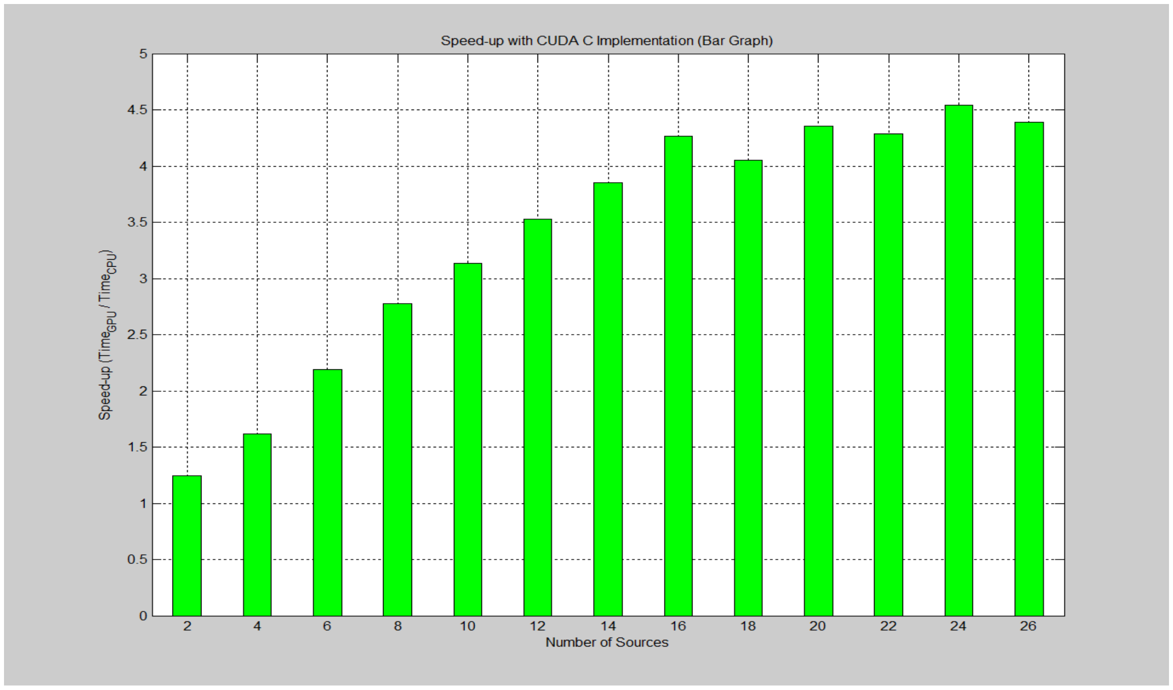
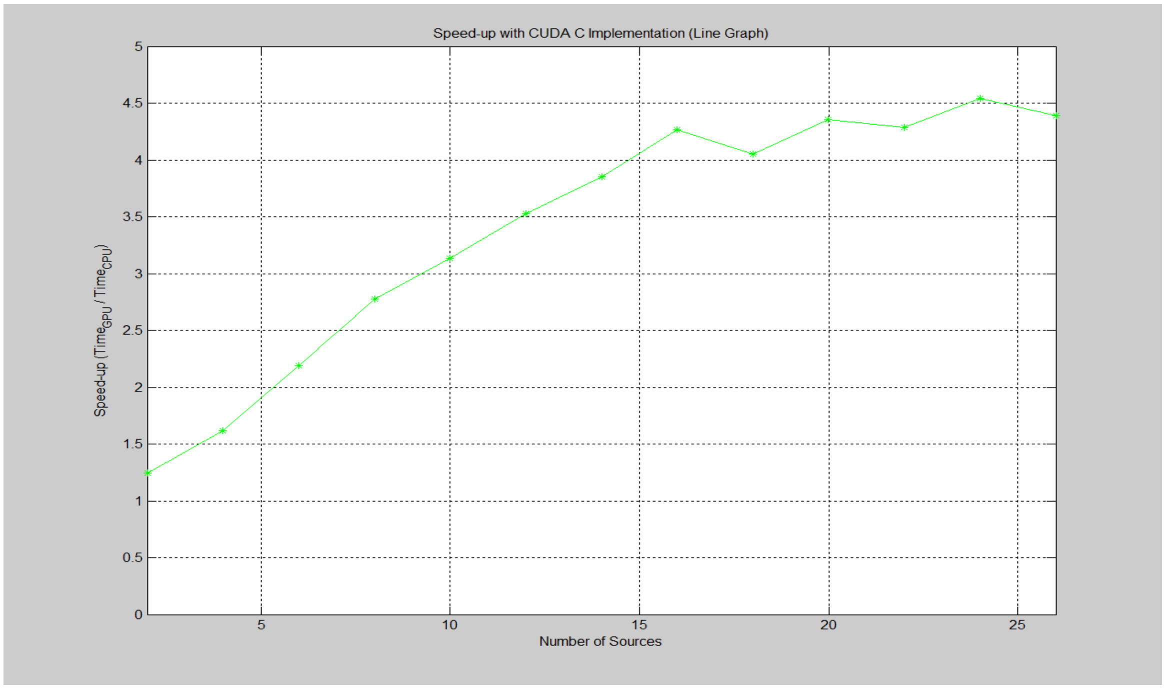


Figure 21 Simulation time of MATLAB implementation for different number of sources

There is an important reason behind why using different number of sources is a good measure to test computational efficiency. As we said earlier, the implementation of the forward model consists of two main parts: computation of the system matrix A and solving the system of linear equations. In terms of time, the computation of the system matrix A is insignificant compared to solving the system of linear equations. Solving the system is an iterative process for the MATLAB implementation and its time is dependent on the number of sources used. Therefore, the number of sources generally determines the overall time of the forward solver. In fact, time for solving of the system of linear equations is so dominant that CPU simulation time of the forward solver changes almost linearly with the number of sources. On the other hand, GPU simulation time of the forward solver increases with much smaller increments as the number of sources increases. Therefore, the speed-up which is shown in Figure 22 becomes higher with increasing number of sources. However, beyond a certain number of sources, the speed-up slows down. Just like the simulations of different 3D meshes, we cannot use large number of sources in the simulations due to memory requirements. For our simulations with range of number of sources from 2 to 26, we observe that maximum speed-up is approximately 4.5.



(a)



(b)

Figure 22 The speed-up graphs with CUDA C implementation for different number of sources (a) The bar graph of the speed-up and (b) The line graph of the speed-up

CHAPTER FIVE
CONCLUSION AND FUTURE WORK

5.1. Conclusion

In this work, we implemented a GPU-based solver for the forward problem of DOT by using a Finite Element method. In the implementation, CUDA's parallel architecture and MATLAB software were combined by utilizing MEX Files. We made several simulations to test relative computational accuracy and efficiency of our new CUDA C implementation when compared to the MATLAB implementation. For relative computational accuracy, we used a mesh with a high number of nodes and one source. We found that root-mean-square (RMS) errors between the CUDA C and the MATLAB implementation are in the order of magnitude of -17 for the photon density and in the order of magnitude of -14 for the flux. These RMS errors are very small compared to the minimum values of photon density and flux that are in the order of magnitude of -5. This shows that our new implementation has a high accuracy. For relative computational efficiency, we used several meshes with various nodes and elements, and utilized different number of sources. We computed that the maximum speed-up is around 2.5 for simulations with different meshes and around 4.5 for simulations with different number of sources.

In conclusion, we can say that we have significant computational speed-up with high computational accuracy. However, it is possible to accelerate the computational speed. One way to achieve that is to write a custom GPU code to solve the system of linear equations with a heterogeneous programming style. In a nutshell, we achieved the goal of this project with our new GPU-based implementation.

5.2. Future Work

As mentioned in the implementation chapter of the forward model, we used a library function provided by CUDA programming model to solve the system of linear equations. Using a library function limited our capabilities to optimize the system solver. Therefore, as a future work, we can write a custom code to minimize the effects of bottlenecks, which improves the performance of the system solver. We believe that running a custom GPU parallelized code on a more powerful NVIDIA Tesla GPU card will provide much higher speed-up when compared to current GPU-based implementation. Apart from the custom code, it is possible to decrease the simulation time by a heterogeneous programming style. We can optimize our code to run both on CPU and GPU simultaneously and have a higher speed-up.

We also mentioned that image reconstruction of DOT involves two parts, which are the forward and the inverse problem. In this project, we only worked on the forward problem; therefore, it is possible to extend the project to the inverse problem. Although image reconstruction can be done significantly faster with our current GPU-based implementation, we can also take advantage of parallelism of the inverse problem. Additionally, we can combine the forward and the inverse problem and obtain much higher speed-up with further parallelism. We can even go beyond the speed-up provided by the parallelized forward and inverse problems by using a heterogeneous programming style.

Lastly, we used the diffusion approximation of Radiative Transfer Equation (RTE) in this project; however, it is possible to utilize higher order approximation of RTE. As a future work, we can implement a GPU-based solver for the forward problem which uses higher order of approximation of RTE. Since complexity of the higher order approximations is much higher than diffusion approximation, the simulation time will be longer. However, we believe that the calculation time can be decreased to acceptable limits by a GPU-based implementation.

REFERENCES

[1] J.T. Bushberg and J.M. Boone, The essential physics of medical imaging, Lippincott Williams & Wilkins, 2011.

[2] H. Jiang, Diffuse Optical Tomography Principles and Applications, CRC Press Taylor and Francis Group, 2011.

[3] J.W. Tunnell, In vivo clinical imaging and diagnosis, The McGraw-Hill Companies, 2011.

[4] M. J. Cross, S. E. Harms, J. H. Cheek, G.N. Peters, R. C. Jones, New horizons in thediagnosis and treatment of breast cancer using magnetic resonance imaging, Am J Surg, 166, 749 – 753; discussion 753-755, 1993.

[5] P. Forouhi, J. S. Walsh, T. J. Anderson, U. Chetty, Ultrasonography as a method of measuring breast tumor size and monitoring response to primary systematic treatment, Br J Surg, 81, 223-225, 1994.

[6] M. Tatsumi, C. Cohade, K. A. Mourtzikos, E. K. Fishman, R. L. Walh, Initial experience with FDG-PET/CT in the evaluation of breast cancer, Eur J Nucl Med Mol Imaging, 33, 254-262, 2006.

[7] B. Bone, Z. Pentek, L. Perbeck, B. Veress, Diagnostic accuracy of mammography and contrast enhanced MR imaging in 238 histologically verified breast lesions, Acta. Radiol., 38, 489–496, 1997.

[8] T. Hata, H. Takahashi, K. Watanabe, M. Takahashi, K. Taguchi, T. Itoh, S. Todo, Magnetic resonance imaging for preoperative evaluation of breast cancer: a comparative study with mammography and ultrasonography, J. Am. Coll. Surg., 198, 190–197, 2004.

[9] A. E. Cerussi, A. J. Berger, F. Bevilacqua, N. Shah, D. Jakubowski, J. Butler, R. F. Holcombe, B. J. Tromberg, Sources of absorption and scattering contrast for near-infrared optical mammography, Acad Radiol, 8, 211-218, 2001.

[10] F. F. Jobsis, Noninvasive, infrared monitoring of cerebral and myocardial oxygen sufficiency and circulatory parameters, Science, 198, 1264-1267, 1977.

[11] A. Villringer, B. Chance, Non-invasive optical spectroscopy and imaging of human brain function, Trends Neurosci, 20, 435-442, 1997.

[12] M. Fabiani, D. D. Schmorrow, G. Gratton, Optical imaging of the intact human brain, IEEE Eng Med Biol Mag, 26, 14-16, 2007.

[13] G. Strangman, D. A. Boas, J. P. Sutton, Non-invasive neuroimaging using near-infrared light, Biol Psychiatry, 52, 679-693, 2002.

[14] R. Splinter, B. A. Hooper, An introduction to biomedical optics, CRC Press Taylor and Francis Group, 2007.

[15] T.O. McBride, B.W. Pogue, E.D. Gerety, S.B. Poplack, U.L. Osterberg, K.D. Paulsen, Spectroscopic diffuse optical tomography for quantitative assessment of hemoglobin concentration and oxygen saturation in breast tissue, Appl. Opt., 38, 5480-5490,1999.

[16] M. Cope, D.T. Delpy, System for long term measurement of cerebral blood and tissue oxygenation on newborn infants by near infra-red transillumination, Med Biol Eng Compt, 26, 289-294, 1988.

[17] V. Quaresima, S.J. Matcher, M. Ferrari, Identification and quantification of intrinsic optical contrast for near-infrared mammography, Photochem Photobiol, 67, 4-14, 1998.

[18] C. E. Cooper, M. Cope, R. Springett, P.N. Amess, J. Penrice, L. Tyszczuk, S. Punwani, R. Ordidge, J. Wyatt, D. T. Delpy, Use of mitochondrial inhibitors to demonstrate that cytochrome oxidase near-infrared spectroscopy can measure mitochondrial dysfunction noninvasively in the brain, J Cereb Blood Flow Metab, 19, 27-38, 1999.

[19] G. Gulsen, H. Yu, J. Wang, O. Nalcioglu, S. Merritt, F. Bevilacqua, A.J. Durkin, D.J. Cuccia, R. Lanning, B. J. Tromberg, Congruent MRI and near-infrared spectroscopy for functional and structural imaging of tumors, Technol Cancer Res Treat, 1, 497-505, 2002.

[20] R. L. P. van Veen, H. J. C. M. Sterenborg, A. Pifferi, A. Torricelli, E. Chikoidze, R. Cubeddu, Determination of visible near-IR absorption coefficients of mammalian fat using time- and spatially resolved diffuse reflectance and transmission spectroscopy, J Biomed Opt., 10 (5), 054004, 2005.

[21] S. Kukreti, A. E. Cerussi, W. Tanamai, D. Hsiang, B. J. Tromberg, E. Gratton, Characterization of metabolic differences between benign and malignant tumors: high-spectral resolution diffuse optical spectroscopy. *Radiology*, 254 (1), 277-84.

[22] B. W. Pogue, X. Song, S. Srinivasan, H. Dehghani, T. D. Paulsen, K. D. Tosteson, C. Kogel, S. Soho, S. Poplack, Near-infrared scattering spectrum differences between benign and malignant breast tumors measured in vivo with diffuse tomography, OSA Biomed. Optics Topical Meetings, 2004.

[23] R. C. Haskell, L. O. Svaasand, T. Tsay, T. C. Feng, M. S. McAdams, B. J. Tromberg, Boundary conditions for the diffusion equation in radiative transfer, J. Opt. Soc. Am. A Opt. Image Sci. Vis. 11, 2727, 1994.

[24] K. Ren, G. Bal, A. H. Hielscher, Transport- and diffusion-based optical tomography in small domains: a comparative study, Appl. Opt. 46, 6669-6679, 2007.

[25] S. R. Arridge, M. Schweiger, M. Hiraoka, D. T. Delpy, A finite element approach for modeling photon transport in tissue, Med. Phys. 20, 299-309, 1993.

[26] J. C. Rasmussen, A. Joshi, T. Pan, T. Wareing, J. McGhee, E. M. Sevick-Muraca, Radiative transport in fluorescence-enhanced frequency domain photon migration, Med. Phys. 33, 4685-4700, 2006.

[27] F. P. Bolin, L. E. Preuss, R. C. Taylor, R. J. Ference, Refractive index of some mammalian tissues using a fiber optic cladding method, Appl. Opt. 28, 2297-2303, 1989.

[28] Y. Lin, Quantitative fluorescence tomography using multi-modality approach, University of California, Irvine, 2009.

[29] V. V. Tuchin, Laser light scattering in biomedical diagnostics and therapy, J. Laser Appl. 5, 43-60, 1993.

[30] A. Ishimaru, Wave propagation and scattering in random media, New York Academic, 1978.

[31] M.S. Patterson, B. Chance, B.C. Wilson, Time resolved reflectance and transmittance for the non-invasive measurement of tissue optical properties, Appl Opt, 28, 2331-2336, 1989.

[32] J.B. Fishkin, E. Gratton, Propagation of photon-density waves in strongly scattering media containing an absorbing semi-infinite plane bounded by a straight edge, J Opt Soc Am A, 10, 127 -140, 1993.

[33] D. A. Boas, M. A. O'leary, B. Chance, A.G. Yodh, Scattering of diffuse photon density waves by spherical inhomogeneities within turbid media: analytic solution and applications, Proc Natl Acad Sci USA, 91, 4887-4891, 1994.

[34] X. Intes, J. Ripoll, Y. Chen, S. Nioka, A.G. Yodh, B. Chance, In vivo continuous-wave optical breast imaging enhanced with Indocyanine Green, Med Phys., 30 (6), 1039-1047, 2003.

[35] A. Siegel, J.J. Marota, D. Boas, Design and evaluation of a continuous-wave diffuse optical tomography system, Opt Express, 4 (8), 287-298, 1999.

[36] J. Su, H. Shan, H. Liu, M.V. Klibanov, Reconstruction method with data from a multiple-site continuous-wave source for three-dimensional optical tomography, J Opt Soc Am A Opt Image Sci Vis., 23 (10), 2388-2395, 2006.

[37] I. Nissila, J. C. Hebden, D. Jennions, J. Heino, M. Schweiger, K. Kotilahti, T. Noponen, A. Gibson, S. Jarvenpaa, L. Lipiainen, T. Katila, Comparison between a time-domain and a frequency-domain system for optical tomography, J. Biomed. Opt., 11, 064015:1-18, 2006.

[38] B. Chance, M. Cope, E. Gratton, N. Ramanujam, B. Tromberg, Phase measurement of light absorption and scatter in human tissue, Rev. Sci. Inst., 69 (10), 3457-3481, 1998.

[39] J. P. Culver, R. Choe, M. J. Holboke, L. Zubkov, T. Durduran, A. Slemp, V. Ntziachristos, B. Chance, A. G. Yodh, Three-dimensional diffuse optical tomography in the parallel plane transmission geometry: evaluation of a hybrid frequency domain/continuous wave clinical system for breast imaging, Med Phys., 30 (2), 235-247, 2003.

[40] F. Bevilacqua, A. J. Berger, A. E. Cerussi, D. Jakubowski, B. J. Tromberg, Broadband absorption spectroscopy in turbid media by combined frequency-domain and steady-state methods, Appl Opt., 39 (34), 6498-6507, 2000.

[41] J. Wang, B. W. Pogue, S. Jiang, K. D. Paulsen, Near-infrared tomography of breast cancer hemoglobin, water, lipid, and scattering using combined frequency domain and cw measurement, Opt Lett., 35 (1), 82-84, 2010.

[42] Q. Fang, S. A. Carp, J. Selb, G. Boverman, Q. Zhang, D. B. Kopans, R. H. Moore, E. L. Miller, D. H. Brooks, D. A. Boas, Combined optical imaging and mammography of the healthy breast: optical contrast derived from breast structure and compression, IEEE Trans Med Imaging, 28 (1), 30-42, 2009.

[43] H. Eda, I. Oda, Y. Ito, Y. Wada, Y. Oikawa, Y. Tsunazawa, M. Takada, Y. Tsuchiya, Y. Yamashita, M. Oda, A. Sassaroli, Y. Yamada, M. Tamura, Multichannel time-resolved optical tomographic imaging system, Review of Scientific Instruments, 70 (9), 3595-3602, 1999.

[44] F. E. W. Schmidt, M. E. Fry, E. M. C. Hillman, J. C. Hebden, D. T. Delpy, A 32-channel time-resolved instrument for medical optical tomography, Rev. Sci. Instrum., 71 256, 2000.

[45] X. Intes, Y. Jiangsheng , A. G. Yodh, B. Chance, Development and evaluation of a multi-wavelength multi-channel time resolved optical instrument for NIR/MRI mammography co-registration, Proc. IEEE 28th Annual Northeast Bioengineering Conf., 91–92, 2002.

[46] P. K. Yalavarthy, A generalized least-squares minimization method for near infrared diffuse optical tomography, Dartmouth College, 2007.

[47] S. R. Arridge, M. Cope, D. T. Delpy, The theoretical basis for the determination of optical path lengths in tissue: temporal and frequency analysis, Phys. Med. Biol., 37, 1531–1559, 1992.

[48] S. A. Parhl, M. Keijzer, S. L. Jacques, A. J. Welch, A Monte Carlo model of light propagation in tissue, Dosimetry of laser radiation in medicine and biology, SPIE Institute Series, IS 5, 102-111, 1989.

[49] B. C. Wilson, G. Adam, A Monte Carlo model for the absorption and flux distributions of light in tissue, Med. Phys., 10, 824–830, 1983.

[50] L. H. Wang, S. L. Jacques, L. Q. Zheng, MCML - Monte Carlo modeling of photon transport in multi-layered tissues, Comp. Meth. Prog. Biomed., 47, 131–146, 1995.

[51] K. Ren, G. S. Abdoulaev, G. Bal, A. H. Hielscher, Algorithm for solving the equation of radiative transfer in the frequency domain, Opt. Lett., 29, 578–580, 2004.

[52] S. Srinivasan, B. W. Pogue, C. M. Carpenter, P. K. Yalavarthy, K. Paulsen, A boundary element approach for image-guided near-infrared absorption and scatter estimation, Med. Phys., 34, 4545-4557, 2007.

[53] S. R. Arridge, Optical tomography in medical imaging, Inv. Problems, 15, R41–R93, 1999.

[54] K. Madsen, H. B. Nielsen, O. Tingleff, Methods for non-linear least squares problems, Technical University of Denmark, 2004.

[55] W. Sun, Y. X. Yuan, Optimization Theory and Methods, Springer, 2006.

[56] NVIDIA, Parallel Programming and Computing Platform, NVIDIA Corporation, http://www.nvidia.com/object/cuda_home_new.html.

[57] NVIDIA, CUDA Toolkit Documentation, NVIDIA Corporation, http://docs.nvidia.com/cuda/index.html.

[58] NVIDIA, CUDA C Programming Guide, NVIDIA Corporation, http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html.

[59] NVIDIA, CUDA C Best Practices Guide, NVIDIA Corporation, http://docs.nvidia.com/cuda/cuda-c-best-practices-guide/index.html.

[60] MATLAB & Simulink, MEX File Creation API, The Mathworks Inc., http://www.mathworks.com/help/matlab/call-mex-files-1.html.

[61] MATLAB & Simulink, GPU Computing, The Mathworks Inc., http://www.mathworks.com/help/distcomp/gpu-computing.html.