Hardware Acceleration of a Monte Carlo Simulation for Photodynamic Therapy Treatment Planning

by

William Chun Yip Lo

A thesis submitted in conformity with the requirements
for the degree of Master of Science
Graduate Department of Medical Biophysics
University of Toronto

Abstract

Hardware Acceleration of a Monte Carlo Simulation for Photodynamic Therapy Treatment Planning

William Chun Yip Lo

Master of Science

Graduate Department of Medical Biophysics

University of Toronto

2009

Monte Carlo (MC) simulations are widely used in the field of medical biophysics, particularly for modelling light propagation in biological tissue. The iterative nature of MC simulations and their high computation time currently limit their use to solving the forward solution for a given source configuration and optical properties of the tissue. However, applications such as photodynamic therapy treatment planning or image reconstruction in diffuse optical tomography require solving the inverse problem given a desired light dose distribution or absorber distribution, respectively. A faster means for performing MC simulations would enable the use of MC-based models for such tasks. In this thesis, a gold standard MC code called MCML was accelerated using two distinct hardware-based approaches, namely designing custom hardware on field-programmable gate arrays and programming commodity graphics processing units (GPUs). Currently, the GPU-based approach is promising, offering approximately 1000-fold speedup with 4 GPUs compared to an Intel Xeon CPU.

Acknowledgements

This thesis is truly a journey. Along its path, I met a number of individuals who have assisted me and taught me a great deal, including different ways to approach problems. Through this interdisciplinary research project, I had the pleasure of working with and being instructed by experts from different research areas. First, I am indebted to both of my supervisors, Prof. Lothar Lilge and Prof. Jonathan Rose, for providing me with an excellent learning environment as well as their guidance and mentorship throughout my project. Together, they have helped realize my goal of applying my engineering skills in the medical field. Furthermore, Dr. David Jaffray's insights into clinical treatment planning have been instrumental in the development of my thesis.

Through this interdisciplinary research, I also had the chance to work with fellow graduate students in the Department of Electrical and Computer Engineering. Specifically, I would like to acknowledge Jason Luu and Keith Redmond's assistance with the FPGA-based hardware design. In addition, Prof. Chow, my instructor for a number of computer hardware design courses, has offered me insightful advice during this initial phase of my thesis. During the second phase of my thesis, David Han's expertise in the NVIDIA GPU architecture and his assistance with optimizing the CUDA program have been invaluable. The friendship we have established is something that I value very much.

Finally, I would like to thank my parents for bringing me to Canada so that I can meet with all these great individuals. Without their sacrifice, none of this would have been possible today.

This thesis is funded by the Natural Sciences and Engineering Research Council of Canada (NSERC) through an NSERC postgraduate scholarship.

Contents

List of Tables

List of Figures

List of Abbreviations

AAPM	American Association of Physicists in Medicine
ALA	5-aminolevulinic acid
CAD	Computer-aided design
CPU	Central processing unit
CPU-MCML	CPU-based MCML
CUDA	Compute Unified Device Architecture
DSP	Digital signal processing
ENIAC	Electronic numerical integrator and computer
FBM	FPGA-based MCML
FEM	Finite element method
FPGA	Field programmable gate array
GPU	Graphics processing unit
GPU-MCML	GPU-based MCML
IC	Integrated circuit
I/O	Input/output
LE	Logic element
MC	Monte Carlo
MCML	Monte Carlo for Multi-Layered media (name of software package)
IPDT	Interstitial photodynamic therapy
PDP	Power-delay product
PDT	Photodynamic therapy
RTE	Radiative transport equation
SIMT	Single-instruction, multiple-thread
SP	Scalar processors
TM-4	Transmogrifier 4 (A prototyping platform with 4 Stratix FPGAs)

List of Symbols

D^*	Photodynamic dose [ph/g]
D	Concentration of the photosensitizer drug [mol/L]
ϕ	Light fluence rate [W/cm^2]
T	Light exposure time [s]
ε	Extinction coefficient [cm^{-1}/(mol/L)]
ρ	Density [g/cm^3]
h	Planck's constant (6.6×10^{-34} J s)
c	Speed of light (3.0×10^{10} cm/s)
λ	Wavelength [nm]
D^*_{th}	Threshold dose [ph/g]
ψ_{th}	Threshold fluence [J/cm^2]
μ_a	Absorption coefficient [cm^{-1}]
μ_s	Scattering coefficient [cm^{-1}]
g	Anisotropy factor [dimensionless]
n	Refractive index [dimensionless]
$A[r][z]$	Absorption probability density (or absorption array) [cm^{-3}]
dr, dz	Resolution of absorption grid in the r and z directions [cm]
nr, nz	Number of absorption grid elements in the r and z directions
ξ	Uniform random variable
(μ_x, μ_y, μ_z)	Direction cosines of a photon packet
θ	Deflection angle
ψ	Azimuthal angle
W	Current weight of a photon packet
ΔW	Absorbed weight

Chapter 1

Introduction

Photodynamic therapy (PDT) is an emerging, minimally invasive treatment modality in oncology and other fields. For treating oncologic conditions, the key steps in PDT include the uptake of a light-sensitive drug in the patient's tumour and the local activation of the drug by delivering a sufficient light dose selectively to the region. To effectively target the therapy at the tumour, while sparing the healthy tissue nearby, accurate light dosimetry is critical during treatment planning. Among other techniques for computing light dose, the Monte Carlo (MC) method is considered the gold standard approach in terms of accuracy and flexibility in modelling complex 3-D geometries. However, the use of MC-based models for solving iterative, inverse optimization problems such as PDT treatment planning is currently hindered by its long computation time. Accelerating MC simulations would enable their use for solving such computationally intensive inverse problems. Specifically, this thesis explores two different hardware-based approaches to accelerate an MC simulation for computing light dose in multi-layered biological tissue.

The following sections review the progress in PDT and the clinical dosimetric concepts unique to PDT. In particular, the notion of light dosimetry is introduced to explain how the MC method can be used for PDT treatment planning.

1.1 Progress in Photodynamic Therapy

Initially developed for the local destruction of solid tumours, today photodynamic therapy (PDT) has been applied to a wide range of clinical conditions. The fundamental mechanism of PDT involves the accumulation of a light-sensitive compound, called a photosensitizer, in the treatment target and the irradiation of this target volume with light (typically in the visible to near-infrared range) to generate reactive oxygen species [1,2,3,4]. The biological effects of these reactive oxygen species include tissue destruction through necrosis or apoptosis, vascular damage resulting in further cell death, and immune modulation.

In terms of non-oncologic conditions, PDT has become a standard treatment for age-related macular degeneration [5]. It is also being investigated for localized infections such as periodontitis [6] and for other conditions including rheumatoid arthritis [7]. As for oncologic applications, PDT has demonstrated high efficacy for the treatment of basal cell carcinoma [8], which is a superficial skin tumour. In addition, PDT has been approved for treating refractory superficial bladder cancer [9], early-stage bronchial cancer [10], and high-grade dysplasia in Barrett's oesophagus which is an important risk factor for developing oesophageal carcinoma [11]. Clinical trials are underway to investigate the use of this minimally invasive modality for deep-seated tumours, such as malignant brain tumours [12], prostate cancer [13], and head and neck cancers [14]. However, special light delivery systems are required to adequately cover the complex organ geometries in these cases. For example, in a prostate PDT trial, multiple interstitial fibres are surgically implanted using a modified stabilizing system originally designed for brachytherapy [15]. Since the inter-patient variations in optical properties, 3-D geometry, and biological response of the tumour can be significant, pre-treatment optimization or treatment planning for each patient is especially critical in interstitial PDT (IPDT) [15]. Although PDT can, in theory, be repeated multiple times without inducing apparent resistance in the tumour (since DNA is not the major target in PDT [16]), treatment planning

helps physicians and medical physicists avoid under-dosing or over-dosing, particularly in critical structures or organs. The biological response and clinical outcome can also be more accurately correlated with the prescribed dose. By building up this knowledge base, tissue response models can be developed. Overall, treatment planning is a key step in the development of IPDT as a reliable treatment modality. The next section introduces the fundamental dosimetry concepts required to understand the unique challenges in treatment planning for PDT.

1.2 Clinical Dosimetry for PDT Treatment Planning

Compared to radiation therapy treatment planning, PDT treatment planning is still a nascent field. PDT dosimetry requires the consideration of at least three key parameters: the concentration of photosensitizing drug, the power density of light delivered, and the partial pressure of oxygen in the tissue. Unfortunately, these three quantities are intricately linked together and they can also vary over the course of the treatment due to photobleaching (which decreases the effective concentration of photosensitizers present), changes in optical properties within the necrotic tissue (which affects the amount of light actually delivered), or vascular shutdown (which decreases the concentration of oxygen) [17]. For practical dosimetry in a clinical setting, the American Association of Physicists in Medicine (AAPM) recommended the definition of a more practical dosimetric parameter called photodynamic dose [18]. Under this definition (Eq. 1.1), the photodynamic dose D^* [measured in ph/g or the number of photons absorbed by photosensitizer per gram of tissue] is primarily a function of the light fluence rate ϕ [W/cm^2], light exposure time T [s], and concentration of the photosensitizer drug D [mol/L or moles of drug per litre of tissue] accumulated in the target site:

$$D^* = \varepsilon D \phi T \frac{\lambda}{hc\rho} \tag{1.1}$$

where ε is the extinction coefficient of the photosensitizing drug [$cm^{-1}/(mol/L)$], ρ is the density of the tissue [g/cm^3], h is Planck's constant which equals 6.6×10^{-34} J s, c is the speed of light or 3.0×10^{10} cm/s, and λ is the wavelength of the photon expressed in centimetres. Note that λ/hc also represents the number of photons per Joule of energy.

Based on this photodynamic dose definition, the threshold dose (D^*_{th}) to achieve the desired biological effect (namely cell death) can be determined experimentally through the delineation of necrotic zones, which have been shown to have sharp boundaries in PDT [19]. Typical values of D^*_{th} range from 10^{18} to 10^{19} photons/gram depending on the photosensitizing drug used and the intrinsic sensitivity of the target tissue to PDT [20].

A more convenient definition is based on the threshold fluence ψ_{th} [J/cm^2] since fluence is typically monitored throughout PDT. Rearranging Eq. 1.1 yields an expression for the threshold fluence, as shown below:

$$\psi_{th} = \phi T = (\frac{D^*_{th}}{\varepsilon D})(\frac{hc\rho}{\lambda}) \tag{1.2}$$

Note that a different threshold value is required for each wavelength.

The above definitions of photodynamic dose do not consider the effect of tissue deoxygenation, the quantum yield in the generation of oxidative radicals, and the fraction of generated radicals that succeed in causing cellular damage [18]. The reason for this simplification is that the parameters used in the above definitions, such as the optical power density and the photosensitizer concentration, are easier to control under experimental or clinical conditions. In particular, light dosimetry has been widely accepted to be very important in PDT treatment planning. Ongoing clinical trials have focused on the quantification of fluence both within and around the tumour as the fluence distribution can be varied (within limits) even during the therapy [15, 21, 22]. Selective tumour necrosis is largely dependent on reaching a sufficiently high light dose or fluence within the tumour while not exceeding the threshold level for necrosis in surrounding normal tissues. Therefore, a successful PDT treatment relies on the accurate computation of the fluence distribution throughout the tumour and in surrounding healthy tissues or

organs at risk. Improvements in PDT efficacy, particularly for interstitial applications, require the development of fast and accurate computational tools to enable efficient light dosimetry for PDT treatment planning and for real-time adjustment of the optical power density during the therapy. To achieve this goal, the treatment planning software should employ an accurate model of light propagation in turbid media that can take into account the complex geometry of the tumour and the heterogeneity in the tissue's light interaction coefficient and responsivity to PDT, for clinically robust treatment planning.

The focus of this thesis is the acceleration of a gold standard light dose computation method, to be described next. For clinical relevance, the geometric uncertainty in the computation of the isofluence contours, particularly at the threshold fluence level, must be within acceptable limits in the accelerated version. Typically, an acceptable level of uncertainty is +/- 1 to 2 mm in the position of the isofluence contours around the threshold fluence level, considering that a safety margin of 2 mm is currently used in treatment planning for an ongoing prostate PDT clinical trial [15].

1.3 Light Dosimetry Models

The inputs to a light dosimetry model include a set of measured optical properties of the tissue. Tissue optical properties are specified by four key parameters: the absorption coefficient μ_a [in units of mm^{-1} or cm^{-1}], scattering coefficient μ_s [mm^{-1} or cm^{-1}], anisotropy factor g [dimensionless], and refractive index n [dimensionless]. The absorption coefficient is defined as the product of the concentration of chromophores (or light-absorbing molecules) within the tissue and their molecular absorption cross-section. The scattering coefficient is similarly defined. Anisotropy refers to the average cosine of the scattering angle, which ranges from -1 to 1. Isotropic scattering is represented by a value of 0, while backward-directed scattering and forward-directed scattering are indicated by $-1 < g < 0$ and $1 > g > 0$, respectively. Typically, in the therapeutic or spectral win-

dow for PDT (which ranges from 600 to 800 nm), scattering dominates over absorption ($\mu_s >> \mu_a$) and scattering is forward-directed ($1 > g > 0$). Finally, the refractive index of biological tissue ($n = 1.33 - 1.5$) is close to that of water ($n = 1.333$).

A widely accepted mathematical model of light transport in tissue is based on the radiative transport equation (RTE) [23]. The time-dependent RTE is shown in Eq. 1.3.

$$\frac{1}{v}\frac{\partial}{\partial t}L(\mathbf{r},\mathbf{\Omega},t) + \mathbf{\Omega}\cdot\nabla L(\mathbf{r},\mathbf{\Omega},t) + [\mu_a(\mathbf{r}) + \mu_s(\mathbf{r})]L(\mathbf{r},\mathbf{\Omega},t) = \int_{4\pi} L(\mathbf{r},\mathbf{\Omega'},t)\mu_s(\mathbf{r},\mathbf{\Omega'}\rightarrow\mathbf{\Omega})\,d\mathbf{\Omega'} + S(\mathbf{r},\mathbf{\Omega},t) \tag{1.3}$$

The key quantity in the RTE is the radiance [W m^{-2} sr^{-1}] or $L(\mathbf{r},\mathbf{\Omega},t)$, defined as the radiant power [W] crossing an infinitesimal area at location $\mathbf{r}$ perpendicular to the direction $\mathbf{\Omega}$ per unit solid angle. Note that $\mu_s(\mathbf{r},\mathbf{\Omega'}\rightarrow\mathbf{\Omega})$ is the differential scattering coefficient, where $\mathbf{\Omega'}$ represents the propagation direction before elastic scattering while $\mathbf{\Omega}$ represents the new direction after scattering. Therefore, the total scattering coefficient is given by $\int_{4\pi}\mu_s(\mathbf{r},\mathbf{\Omega'}\rightarrow\mathbf{\Omega})\,d\mathbf{\Omega'}$ and the term $\int_{4\pi} L(\mathbf{r},\mathbf{\Omega'},t)\mu_s(\mathbf{r},\mathbf{\Omega'}\rightarrow\mathbf{\Omega})\,d\mathbf{\Omega'}$ accounts for the gain in radiance into $\mathbf{\Omega}$ as a result of scattering from all directions $\mathbf{\Omega'}$. The RTE also contains a source term called $S(\mathbf{r},\mathbf{\Omega},t)$, which may be used to describe the light emitted from the implanted source fibres. Finally, v is the speed of light in the tissue.

Exact analytical solutions for the RTE only exist for simple geometries and various approximations are employed in practice. A common first-order approximation is called the diffusion approximation, which has several important limitations [24]. First, the diffusion approximation fails close to light sources. This distance is less than one transport mean free path, defined as $1/[\mu_a + \mu_s(1-g)]$, from the source. Within the spectral window for PDT, one transport mean free path is 1-2 mm. This limitation also extends to photon sinks or strongly absorbing objects as well as boundaries between different tissue types. Second, the diffusion approximation is not as accurate in strongly absorbing tissue, meaning that for this approximation to be valid, the scattering coefficient has to be much greater than the absorption coefficient. This condition is typically

satisfied if $\mu_s(1-g) > 10\mu_a$. To model heterogeneity in tissue optical properties, numerical approaches such as the finite element method (FEM) [25] are commonly used to solve the diffusion equation. In FEM, the complex organ geometry is discretized into a mesh of elements such as 4-noded tetrahedral elements, and a system of equations is constructed to calculate the fluence rate inside each element by applying the diffusion approximation locally. Although this approach has been applied to clinical light dosimetry for prostate IPDT due to its relatively low computation time (2-5 h of total treatment planning time [15]), the limitations of the diffusion approximation are still present. To overcome these limitations, Monte Carlo modelling can be used. In fact, MC simulations are widely used as the gold standard in radiotherapy treatment planning and there is a clear trend towards adopting the MC method for clinical radiotherapy dose calculations in commercial treatment planning systems [26, 27]. The next section presents MC-based light dosimetry for PDT treatment planning.

1.4 MC-based Light Dosimetry

Compared to other techniques for computing light dose, the Monte Carlo (MC) method is more flexible in modelling complex 3-D geometries with heterogeneous tissue optical properties and it is considered the gold standard approach in terms of accuracy [28, 29]. Unfortunately, MC simulations are not yet routinely used in clinical dosimetry for PDT treatment planning because they are computationally intensive and very time-consuming [30]. Although different efficiency-enhancing methods or *variance reduction techniques* are traditionally introduced to reduce the computation time (and similar variance reduction techniques are used in MC-based dosimetry in radiotherapy [31]), the computation time for MC remains high for iterative forward solutions of light transport that optimize the source geometry and emission profile to achieve a desired light dose distribution. Considering that hundreds of iterations are typically required to search for

the optimum source configuration (by changing parameters such as the position, length, and power of each diffuser) [32] and each iteration takes 20-30 minutes using MC-based light dose computation (estimated with the commercial light modelling software called ASAP [33]), MC-based PDT treatment planning would take days to weeks to complete. Accelerating MC simulations would enable the use of MC-based models for solving these iterative, inverse problems, including light dosimetry for treatment planning in PDT and other therapies (such as laser interstitial thermal therapy [34]) or for image reconstruction in diffuse optical tomography [35].

Attempts to accelerate MC simulations for modelling light propagation in tissues have been limited to software parallelization schemes. For example, one such scheme involved dividing the simulation into many independent groups, each of which was executed on a different computer or processor in parallel [36, 37]. One potential problem with the software parallelization approach is the need to have dedicated access to a computer cluster in order to achieve the desired performance. This approach is not easily accessible as the capital and maintenance costs of a large, dedicated networked cluster of servers are substantial, thus hindering the deployment of complex MC-based models in iterative optimization problems.

This thesis explores two distinct hardware-based approaches to accelerate MC simulations for computing light dose in PDT. The first approach involves the creation of custom hardware *de novo* on programmable logic devices called field-programmable gate arrays (FPGAs) [38,39]. The second approach exploits the high performance of commodity graphics processing units (GPUs). To demonstrate the feasibility of the hardware approach, the widely accepted MC code called Monte Carlo for Multi-Layered media (MCML) [40] was used as a gold standard for the computation of light dose distributions. With modifications to model more complex scenarios, it can be used for MC-based light dosimetry in PDT treatment planning.

1.5 Organization of Dissertation

The next chapter provides further background on modelling light propagation in tissue using the MC method and presents the MCML algorithm in detail to facilitate the discussions in subsequent chapters. Chapter 3 and Chapter 4 illustrate how two different hardware-based approaches, namely the FPGA-based approach and GPU-based approach, were used to accelerate the MCML code. In each chapter, the relevant programming paradigms and related work are introduced before describing the final solution as well as its accuracy and performance. Finally, Chapter 5 concludes with the contributions of this thesis, the implications of the current work, and the future work required to enable MC-based PDT treatment planning for complex 3-D geometries.

Chapter 2

The MCML Light Dose Computation Method

In this chapter, the general Monte Carlo method is introduced, followed by how it can be used for modelling photon transport in biological tissue. The key computational steps in the MCML algorithm are reviewed to show how the MC technique can be applied to the computation of light dose in multi-layered tissue.

2.1 The Monte Carlo Method

The Monte Carlo method is a statistical sampling technique that has been widely applied to a number of important problems in medical biophysics and many other fields, ranging from photon beam modelling in radiation therapy treatment planning [41] to protein evolution simulations in biology [42]. The name *Monte Carlo* is derived from the resort city in Monaco which is known for its casinos, among other attractions. As its name implies, the key feature of the MC method involves the exploitation of random chance or the generation of random numbers with a particular probability distribution to model the physical process in question [43]. Since the MC method inherently relies on repeated sampling to compute the quantity of interest, the development of the MC method has

parallelled the evolution of modern electronic computers. In fact, initial interests in MC-based computations stemmed from von Neumann's vision of using the first electronic computer - the ENIAC [44] - for the modelling of neutron transport [45], which was later adopted for the development of the atomic bomb in World War II.

Despite the increased variety and sophistication of MC-based simulations today, most MC-based models still retain the same essential elements, including the extensive use of random numbers and repeated sampling. For example, in the case of photon transport, random numbers are used to determine the distance of photon propagation and the direction of scattering, among other interactions. Each photon is tracked for hundreds of iterations and typically thousands to millions of photons are required to accurately compute the quantity of interest, such as the light dose distribution for the case of PDT. Due to the large number of iterations required, different variance reduction techniques [46] have been introduced to reduce the number of samples required to achieve a similar level of statistical uncertainty or variance in MC-based computations. Conversely, variation reduction schemes allow more equivalent samples to be computed within the same amount of time. (Several relevant techniques are discussed in the next section.) Unfortunately, the simulation time remains high for solving complex optimization problems such as those for treatment planning, which require many of these MC simulations. It is important to note that while this thesis focuses on the acceleration of the MCML code for modelling light propagation only, a similar approach may be used to accelerate other interesting MC-based simulations in the biophysics, including those for radiotherapy treatment planning. The MCML algorithm was chosen as the basis of this initial exploration due to its widespread acceptance and its relevance to light dosimetry in PDT treatment planning.

2.2 The MCML Algorithm

The MCML algorithm [40] provides an MC model of steady-state light transport in multi-layered media. With modifications, it can form the basis for light dose computation in PDT treatment planning. The use of the MCML code package as a gold standard in this thesis is based on the widespread acceptance of the code package and the agreement of the simulation results with tissue phantom-based as well as in vivo measurements [47, 48, 49]. It has also been extended and applied to numerous interesting investigations, ranging from reflectance pulse oximetry studies [50] to the 3-D modelling of light propagation in a human head [51].

The MCML implementation assumes infinitely wide layers, each of which is specified by its thickness and its optical properties, comprising the absorption coefficient, scattering coefficient, anisotropy factor, and refractive index (as described in Section 1.3). A diagram illustrating the propagation of photon packets in a multi-layered skin model [52] is shown in Fig. 2.1 (a), using ASAP as the MC simulation tool to trace the paths of photons [33].

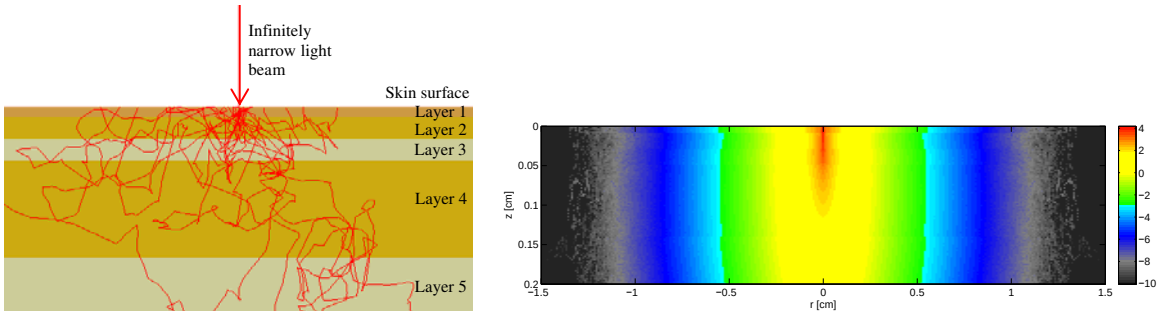
Three physical quantities are scored, in a spatially-resolved fashion, in the MCML code – absorption, reflectance, and transmittance. Note that for the purpose of PDT treatment planning, absorption by the photosensitizer is the quantity of interest (as noted in Eq. 1.1). Therefore, reflectance and transmittance are henceforth not considered. In the MCML code, absorption is recorded in a 2-D absorption array called $A[r][z]$, representing the photon absorption probability density [cm^{-3}] as a function of radius r and depth z. Fig. 2.1 (b) shows the computed absorption probability density after tracing 100 million photon packets from an infinitely narrow light beam (or a point source) perpendicular to the top layer of the same skin model. Through the simulation input parameters, the size of each absorption element (specified by dr and dz) and the number of elements (specified by nr and nz) can be changed. The simulation volume of interest or the extent of the detection grid is specified by the total radius $nr \times dr$ and total depth

$nz \times dz$. A photon packet at positions beyond the extent of the detection grid continues to propagate, but its absorption events are recorded at the boundary elements of the $A[r][z]$ array. As such, these values are incorrect as warned by the authors of the MCML code [40]. Absorption probability density can be converted into more common units, such as photon fluence [measured in cm^{-2} for the impulse response of a point source]. Fig. 2.2 (a) shows this conversion: the fluence distribution was obtained by dividing the absorption probability density, plotted in Fig. 2.1 (b), by the local absorption coefficient for each layer. A common type of plot for visualizing the same fluence distribution in treatment planning is the isofluence contour plot, as shown in Fig. 2.3. To model finite-sized sources, the photon distribution obtained for the impulse response can be convolved with tools such as the CONV program [53]. An example is shown in Fig. 2.2 (b), which plots the fluence distribution resulting from a Gaussian beam.

The simulation of each photon packet consists of a repetitive sequence of computational steps and can be made independent of other photon packets by creating separate absorption arrays and different random seeds. Therefore, a conventional software-based acceleration approach involves processing multiple photon packets simultaneously on multiple processors. Figure 2.4 shows a flow chart of the key steps in an MCML simulation, which includes photon initialization, position update, direction update, fluence update, and photon termination. The following sections provide a brief summary of the computations performed in each of these steps.

2.2.1 Photon Initialization

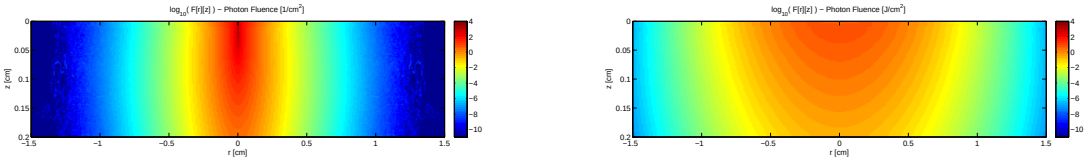
To begin the MCML simulation, a new photon packet is launched vertically downwards (which is assigned the $+z$ direction) into the multi-layered media from the origin. Note that the infinitely narrow light beam irradiates the top layer at the origin. The weight of the photon packet is also initialized to 1.



(a) Tracing multiple photon packets with ASAP

(b) Logarithm of absorption probability density

Figure 2.1: Monte Carlo simulation of photon propagation in a 5-layer skin model from an infinitely narrow beam at 633nm



(a) Infinitely narrow beam

(b) 1 J Gaussian beam (1/e radius=0.5 cm)

Figure 2.2: Logarithm of fluence distribution in the skin model for the impulse response [measured in units of $1/\text{cm}^2$] and for a Gaussian beam [J/cm^2].

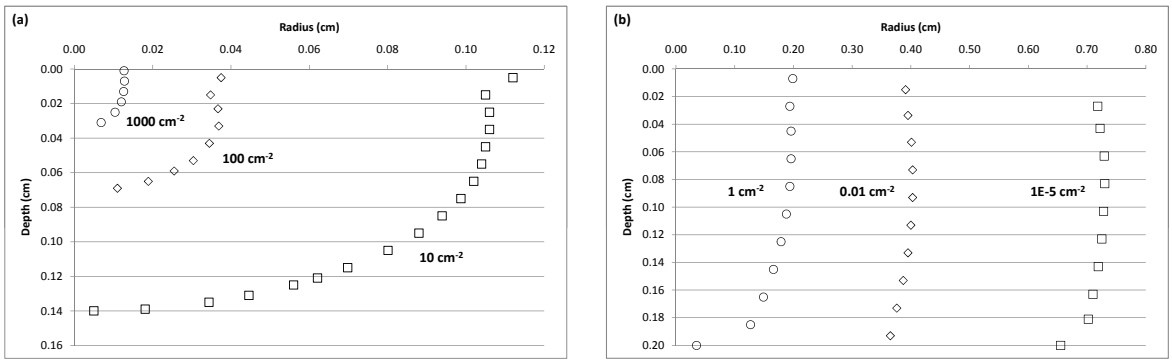


Figure 2.3: Isofluence contour lines for the impulse response in the skin model. Note that Fig. 2.2 (a) shows the same fluence distribution plotted in a different format.

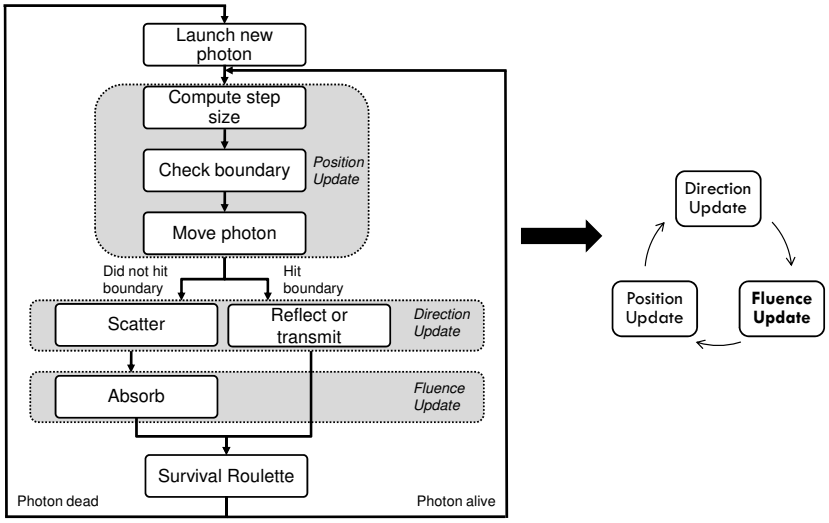


Figure 2.4: Left: Flow-chart of the MCML algorithm. Right: Simplified representation used in subsequent sections.

2.2.2 Position Update

The position update step moves the photon packet along its current direction vector, given by the direction cosines (μ_x, μ_y, μ_z), to its next interaction site. The distance of propagation, called the step size, is computed by sampling a probability density function. Since the probability of a photon travelling a distance s is proportional to $e^{-(\mu_a+\mu_s)s}$ [54], the step size s [cm] can be calculated using Eq. 2.1:

$$s = \frac{-ln(\xi)}{\mu_a + \mu_s} \tag{2.1}$$

where ξ is a random variable uniformly distributed between 0 and 1, while μ_a and μ_s are the absorption and scattering coefficients [cm^{-1}], respectively.

This step size is used to check if the photon packet will encounter a boundary or the interface between two layers. This condition is called a *hit* and is determined by Eq. 2.2:

$$hit = \begin{cases} 1 & \text{if } (s \text{ - } dl\_b/\mu_z) <= 0 \\ 0 & \text{if } (s \text{ - } dl\_b/\mu_z) > 0 \end{cases} \tag{2.2}$$

where dl_b is the distance [cm] to the closest boundary in the direction of photon propa-

gation and μ_z is the direction cosine along the z direction. Note that all boundaries are perpendicular to the z axis.

If the photon packet crosses a boundary, the step size is reduced so that the photon packet arrives at the boundary. The difference between the original step size and the reduced step size is called *sleft* and is calculated using Eq. 2.3. If *sleft* is not zero, it will be used as the step size in the next iteration.

$$sleft = \begin{cases} 0 & \text{if } hit = 0 \\ s - (dl\_b/\mu_z) & \text{if } hit = 1 \end{cases} \tag{2.3}$$

The new position for the photon packet (x', y', z') is determined by first multiplying the step size by the direction cosines in the x, y, and z directions (μ_x, μ_y, μ_z respectively) to obtain the vector components and then adding these values to the old position (x, y, z).

2.2.3 Direction Update

The direction update step performs two mutually exclusive operations depending on whether or not the photon packet has encountered a boundary during the position update step. If the photon packet has not hit a boundary, the Henyey-Greenstein function [55], originally developed for modelling diffuse radiation in the galaxy, is used to model scattering in the tissue and the new direction cosines (μ'_x, μ'_y, μ'_z) are determined according to Eqs. 2.4 - 2.6.

$$\mu'_x = \frac{sin(\theta)[\mu_x\mu_z cos(\psi) - \mu_y sin(\psi)]}{\sqrt{1-\mu_z^2}} + \mu_x cos(\theta) \tag{2.4}$$

$$\mu'_y = \frac{sin(\theta)[\mu_y\mu_z cos(\psi) + \mu_x sin(\psi)]}{\sqrt{1-\mu_z^2}} + \mu_y cos(\theta) \tag{2.5}$$

$$\mu'_z = -sin(\theta)cos(\psi)\sqrt{1-\mu_z^2} + \mu_z cos(\theta) \tag{2.6}$$

where θ is the deflection angle, ψ is the azimuthal angle, and (μ_x, μ_y, μ_z) represents the old direction cosines.

If the photon packet has hit a boundary, the direction update step determines whether it is reflected back or traverses through and updates the direction based on Fresnel's formula [56] and Snell's law. Whether or not a photon packet is reflected back is determined by generating a random number ξ and comparing it with the internal reflectance $R(\theta_i)$ computed using Eq. 2.7.

$$R(\theta_i) = \begin{cases} 1 & \text{if } \theta_i > sin^{-1}(n_t/n_i) \\ \frac{1}{2}\left[\frac{sin^2(\theta_i-\theta_t)}{sin^2(\theta_i+\theta_t)} + \frac{tan^2(\theta_i-\theta_t)}{tan^2(\theta_i+\theta_t)}\right] & \text{otherwise} \end{cases} \tag{2.7}$$

where θ_i is the angle of incidence, θ_t is the angle of transmission, while n_i and n_t are the refractive indices of the incident medium and transmitted medium at the modelled wavelength, respectively. $R(\theta_i)$ represents the average for the two orthogonal polarization directions. (While polarization is not modelled in the MCML code, other code packages such as polmc [57] support polarization-dependent effects.)

If ξ is greater than $R(\theta_i)$, the photon packet is transmitted. Otherwise, it is internally reflected. At incident angles greater than the critical angle or $sin^{-1}(n_t/n_i)$, the photon packet is completely internally reflected. Once the photon packet's path is determined, the direction cosines are updated as follows:

$$(\mu_x', \mu_y', \mu_z') = \begin{cases} (\mu_x, \mu_y, -\mu_z) & \text{if reflected} \\ (\mu_x n_i/n_t, \mu_y n_i/n_t, SIGN(\mu_z)cos\theta_t) & \text{if transmitted} \end{cases} \tag{2.8}$$

where $SIGN(\mu_z)$ gives the sign of μ_z. The direction cosines for the transmitted case are derived based on Snell's law.

2.2.4 Fluence Update

The fluence update step adjusts the photon packet's weight to simulate absorption at the site of interaction. The concept of a photon packet weight is introduced here to simulate absorption in the tissue more efficiently. This is a variance reduction technique known as implicit photon capture [58], which allows a photon packet to be absorbed

or captured multiple times instead of terminating a photon after each absorption event. The differential weight ΔW to be absorbed is computed according to Eq. 2.9 and is accumulated in the raw absorption array $A_{raw}[r][z]$ at the location of absorption.

$$\Delta W = W \frac{\mu_a}{\mu_a + \mu_s} \tag{2.9}$$

where μ_a and μ_s are the absorption and scattering coefficients of the current layer while W is the current weight of the photon packet.

Since the number of photon packets launched (denoted N_{photon}) and the volume of the absorption grid elements (ΔV measured in cm^3) can differ for each simulation, the accumulated weights in the raw absorption array $A_{raw}[r][z]$ must be normalized, as follows:

$$A_{normalized}[r][z] = \frac{A_{raw}[r][z]}{N_{photon}\Delta V} \tag{2.10}$$

where $A_{normalized}[r][z]$ is the absorption probability density measured in units of cm^{-3}.

To obtain the fluence [cm^{-2}], the absorption probability density is divided by the local absorption coefficient μ_a [cm^{-1}] per layer. For the fluence distribution resulting from a finite-sized beam [J/cm^2], the CONV program described earlier can be used.

2.2.5 Photon Termination

The MCML algorithm terminates a photon packet when it exits the tissue or through a Russian roulette [59] that is activated when the weight of the photon packet has reached a predefined threshold value, as further simulation has a minimal effect on the variance of the results. When the weight reaches this threshold, the roulette generates a uniform random number between 0 and 1. If the random number is above 1/10, the photon packet is terminated; otherwise, the weight of the photon packet is increased by a factor of 10 to maintain the conservation of energy in the system. Note that this is another variance reduction scheme implemented in the MCML code to reduce computation time. Other

variance reduction methods also exist, including a plethora of schemes introduced in the MCNP code from the Los Alamos National Laboratory [60].

2.3 Summary

Now that the MCML algorithm has been described, two different hardware-based approaches to accelerate the MCML computations will be presented in the next 2 chapters.

Chapter 3

FPGA-based Acceleration of the MCML Code

This chapter presents the design of custom computer hardware on field-programmable gate arrays (FPGAs) to accelerate the MCML algorithm. An overview of how hardware design on an FPGA differs from software programming on a general-purpose processor is included. For readers interested in the performance and validation results, please refer to Section 3.5 and Section 3.6. The work described in this chapter has been published in the Journal of Biomedical Optics [61] and presented at the IEEE FCCM 2009 conference [62].

3.1 Field-Programmable Gate Arrays

An FPGA is a prefabricated silicon chip that can be programmed electrically to implement virtually any digital design, including custom computer hardware for accelerating simulations. Its flexibility is derived from an underlying *programming technology*, which is typically implemented using a specific type of programmable electrical switch [64]. An FPGA consists of an array of programmable blocks interconnected by a programmable routing fabric as shown in Fig. 3.1 [63]. These programmable blocks include the *soft logic* blocks (also called *logic elements*) that can perform binary computation or store

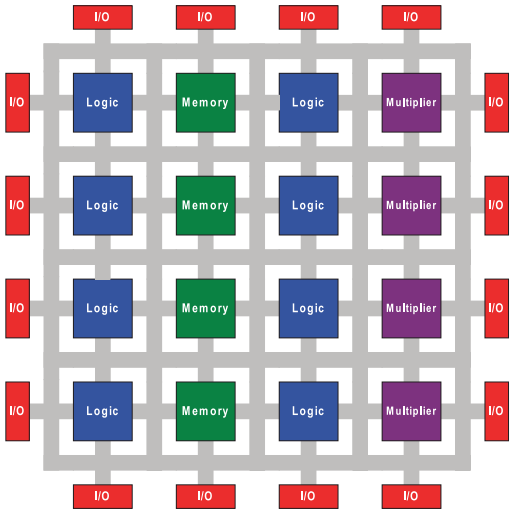


Figure 3.1: Key features of a basic FPGA - a programmable routing fabric (coloured in grey) that interconnects different blocks, such as *soft/programmable logic* (blue), *on-chip memory* (green), and *hard multipliers* (purple). The programmable input/output or I/O blocks (red) connect the FPGA to the outside world [63].

data, *on-chip memory* blocks (with typically only Mbits of space) that allow fast access to larger data structures such as arrays, and *I/O* blocks that connect to other FPGAs, external memory modules (for extra storage space), or a host computer communication interface. Additionally, modern FPGAs contain *hard multipliers* that perform multiplication in faster, non-programmable circuitry since multipliers are costly in terms of area to implement in soft logic blocks. (Note that multiple *digital signal processing (DSP) elements* may be required to implement a hard multiplier, depending on its size as defined by the bit widths of the operands.)

An FPGA enables the design of dedicated custom hardware, providing increased performance for computationally intensive applications, without the high power consumption and maintenance costs of networked computer clusters. FPGAs further offer the flexibility of customizing the underlying hardware architecture for a specific application. Therefore, the FPGA-based approach was explored first to create dedicated computer hardware that was tailored to the MCML algorithm.

3.2 Related Work

There is prior work on FPGA-based acceleration for related MC simulations. Gokhale et al. presented an FPGA-based implementation of an MC simulation for radiative heat transfer that achieved a 10.4-fold speedup on a Xilinx Virtex II Pro FPGA compared to a 3 GHz processor [65]. (Note that *speedup* refers to the ratio between the sequential execution time in software and the parallel execution time in hardware.) A convolution-based algorithm used in radiation dose calculations achieved a 20.7-fold speedup [66]. This group adopted a design flow involving a programming language called Handel-C [67], which was designed to ease hardware development by providing a C-like environment for coding. However, Handel-C does not generate efficient hardware and this group's design was too large to fit on the Altera Stratix FPGA they had available. As a result, their speedup values were projected using results from an emulated version of the hardware using the ModelSim tool [68]. Similarly, Fanti et al. showed only a partial implementation of an MC-based computation for radiotherapy without providing any speedup figures [69]. A working FPGA implementation of an MC-based electron transport simulation was shown by Pasciak et al., reporting a speedup between 300 and 500-fold compared to their custom software running on a 64-bit AMD Opteron 2.4 GHz machine [70]. Their work focused on radiation transport computations, which have some similarities to this work, but involve fundamentally different physical interactions, such as electron impact ionization events due to high-energy beams.

The design presented in this chapter is a working implementation of an MC-based photon migration simulation in biological tissue, based on the MCML code described in Chapter 2, on FPGA hardware. Also, a systematic design flow, involving an intermediate hardware modelling stage, was adopted to reduce development time, as described in the next section.

3.3 Hardware Design Method

3.3.1 Overview

Hardware design requires the explicit handling of two concepts that are normally abstracted from software design: cycle-accurate design and structural design. Cycle-accurate design requires a hardware designer to specify precisely what happens in each hardware clock cycle. Structural design requires a hardware designer to specify exactly what resources to use and how they are connected. In contrast, a typical software designer will not be concerned with the number of clock cycles consumed in a processor for a section of code although they do profile the code to determine and reduce performance bottlenecks. Also, the underlying architecture and the internal execution units of a processor are not specified by the program and are typically not considered by the programmer.

To ease the design flow in FPGA-based hardware development, specific computer-aided design (CAD) tools are used, which are analogous to the compiler used by the software programmer. These CAD tools typically accept a hardware description language, which is a textual description of the circuit structure. To determine the precise logic implementation, location and connectivity routing for a digital hardware implementation on FPGAs, the CAD software performs a number of sophisticated optimizations [71].

To implement a large hardware design, the designer must break down the system into more manageable sub-problems, each of which is solved by the creation of a module that is simulated in a cycle-accurate manner to ensure data consistency. Due to the vast amount of information gathered, a full system simulation cycle-by-cycle for large designs such as the FBM hardware is time-consuming.

To simulate the full system more efficiently, an intermediate stage involving the use of a C-like language that models the cycle-accurate hardware design, without details on the exact implementation, is employed. This stage also allows for the testing and debugging of the additional complexity of cycle-accurate timing before considering structural design

necessary in the final hardware design.

The design of the FPGA-based digital hardware for the MCML computations followed the hardware design stages described above, including the intermediate, cycle-accurate timing stage. A C-based hardware modelling language, called SystemC [72], was used to develop the intermediate hardware design. Verilog [73] was selected as the hardware description language, and the Altera Quartus II 7.2 CAD tool [74] was used to synthesize the Verilog code into hardware structures as well as to configure the FPGA.

3.3.2 Hardware Acceleration Techniques

An FPGA can implement any digital circuit including those with significant amounts of computation. Such implementation has the potential to be significantly faster than software-based implementations on a general-purpose processor for two reasons: first, an FPGA can implement many computational units in parallel and second, it allows exact organization of the data flow to effectively utilize those computational units.

A key factor limiting the amount of parallelism and hence the speed of an FPGA-based solution is the number of logic elements available on the device. Therefore, minimizing the number of logic elements required for binary logic computation can lead to the maximization of the performance per FPGA through replicating computational units to enable parallel execution.

To achieve the goal of maximizing parallelism and computational throughput, three hardware acceleration techniques are commonly applied. First, to greatly reduce the size of a computational unit, the conversion from floating point to fixed point data representation is used, although careful design and modelling are essential to ensure that the proper precision is maintained. Second, look-up tables can be created in the on-chip memory to pre-compute values for expensive computations such as trigonometric functions, thereby eliminating the need for a large number of logic elements. The third key technique is pipelining [75], which optimizes the computational throughput. The

pipelining approach, similar to an assembly line, breaks down a complex problem into simpler stages, each of which is responsible for performing a simple task. Since each stage can now perform its task simultaneously with other stages, the net throughput is increased, thereby speeding up the computation. An example of a pipeline is shown in Fig. 3.2, where the calculation $Y = aX^2 + b$ is broken down into three pipeline stages. Suppose the original calculation takes 300 ns to complete and further suppose that each stage, representing a sub-step in the whole computation, takes 100 ns in this balanced pipeline. Although it still takes 300 ns to compute the value Y from the time the input X enters the pipeline, a continuous stream of new input data can be fed into the pipeline. Therefore, once the pipeline is filled, a new value Y can be computed every 100 ns, thereby increasing the net throughput by a factor of 3 compared to the non-pipelined computation. This increased efficiency is alternatively explained by the fact that 3 computations, performed by 3 independent stages, are executed simultaneously in the pipeline. Note that the slowest stage also dictates the throughput of a pipeline. Therefore, an efficient pipeline design should be properly balanced by dividing the most time-consuming computations into more stages. While pipelining leads to significant performance gain, the complexity involved in designing and verifying the individual stages increases appreciably in sophisticated designs, such as the one for the MCML algorithm accelerated in this work.

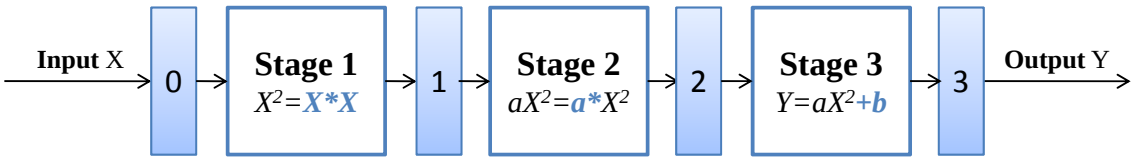


Figure 3.2: An example of a three-stage pipeline: stage 1, square the input X and feed the result X^2 into the next stage; stage 2, multiply X^2 by coefficient a; stage 3, add constant b to aX^2 from stage 2 to create the final result Y. Intermediate values are stored in temporary registers, represented by the numbered rectangles between consecutive stages.

3.4 FPGA-based Hardware Implementation

The FPGA-based hardware implementation is named here FPGA-based MCML or FBM. It was first implemented on a multi-FPGA platform called the Transmogrifier-4 (TM-4) [76]. This platform contains four FPGAs from the Altera Stratix device family (Altera Corporation, San Jose, CA) and has a host software package to communicate with a computer. The same design was also migrated to a newer platform called the DE3 board [77] with a modern Stratix III FPGA to show the implication of FPGA technology on performance.

3.4.1 Modifications to the MCML code

First, to reduce the hardware resource requirements of the design, the 64-bit double-precision floating-point operations used in the MCML software were converted to 32-bit (or 64-bit as required) fixed-point operations in hardware. This conversion had a significant impact on hardware resource usage (and hence the parallelism or performance achievable as discussed above) as floating-point hardware is resource-intensive on FPGAs [78]. However, the use of fixed-point data representation gives rise to other complexities, such as the possibility of overflow. To avoid overflow when accumulating absorption (Eq. 2.9) in fixed-point data representation, the 64-bit data type (which can store a maximum value of 2^{64}-1) was used to create the $A[r][z]$ array.

Similarly, logarithmic and trigonometric functions are very resource-intensive to implement on FPGAs. Therefore, lookup tables were created to store pre-computed values for trigonometric functions (as required in Eqs. 2.4-2.6) and logarithmic functions (required in Eq. 2.1). The FPGA on-chip memory was used to store these lookup tables.

To apply the pipelining technique to the hardware implementation, a SystemC model was created to model the individual stages in the computation. This was the most time-consuming and complex step as the individual stages and the relative timing of the

operations had to be specified. Every dependency in the computation must be carefully laid out to determine the appropriate division of the computation into stages.

PDT treatment planning requires only fluence quantification. Hence, only absorption is recorded in the on-chip memory in the hardware design, while the reflectance and transmittance are ignored to reduce memory usage. The dimensions of the absorption array are fixed at 256 by 256 and the number of tissue layers is restricted to five to further reduce memory usage.

3.4.2 System Overview

The overall system contains both hardware and software components. The hardware component, called the FBM hardware, resides on the FPGA device and performs the Monte Carlo simulation. The software on the host computer performs the pre-processing steps and post-processing steps. The former includes the parsing of the simulation input file and the initialization of the hardware system based on the simulation input file. The latter includes the transfer of the simulation results from the FPGA back to the host computer and the creation of the simulation output file containing the absorption array. The absorption array is then used to generate the fluence distribution. To illustrate the overall program flow from the user's perspective, the key steps are shown in Fig. 3.3. For the final system, the same hardware design is replicated across four FPGA devices on the TM-4 platform to show the scalability of the solution.

3.4.3 Overview of Hardware Design

The design of the the FBM hardware dictates the overall performance of the system. The architecture of the FBM hardware, which uses the pipelining acceleration technique, is shown in Fig. 3.4.

As illustrated in Fig. 3.4, the pipelined hardware consists of a series of hardware modules or cores, each further subdivided into pipeline stages, for the corresponding

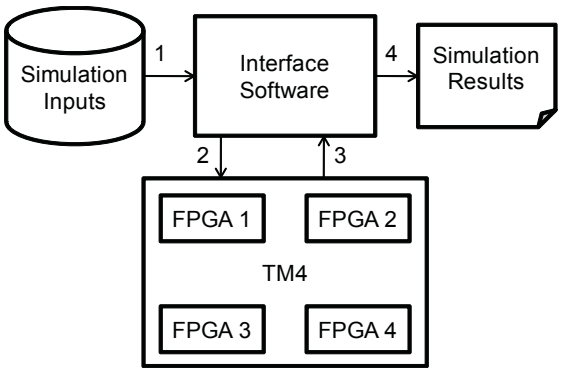


Figure 3.3: FPGA-based system overview: step 1, parsing of the simulation input file; step 2, transfer of initialization information to the FPGAs; step 3, transfer of simulation results from the FPGAs; step 4, creation of the simulation output file.

computations presented in Fig. 2.4. For clarity, the individual arithmetic blocks (such as multipliers), look-up tables, random number generators [79], and the individual pipeline stages are not shown. A single pass through the entire pipeline is equivalent to a single iteration in the key loop of the MCML program. The pipeline has 100 stages, meaning 100 photon packets at different stages of the simulation are handled concurrently once the pipeline is filled. The Step Size Core, Boundary Checker Core, and Movement Core are collectively called the Position Update Engine since they are responsible for updating the position of the photon packet. The Reflect/Transmit Core, Rotation Core (for modelling scattering), and Shared Arithmetic Core are grouped under the Direction Update Engine, which mainly determines the direction cosines of the photon packet. Comparing the hardware design to the software flow chart given in Fig. 2.4, scattering (computed with the Rotation Core), absorption (computed with the Fluence Update Core), and internal reflectance (computed with the Reflect/Transmit Core) are all simulated in parallel in the hardware. In fact, if these three hardware cores were connected in series instead, approximately 3 times the number of temporary storage registers (used to propagate intermediate results between consecutive stages) would be required. In the current design, these registers are shared across the parallel, concurrently executing modules. The final

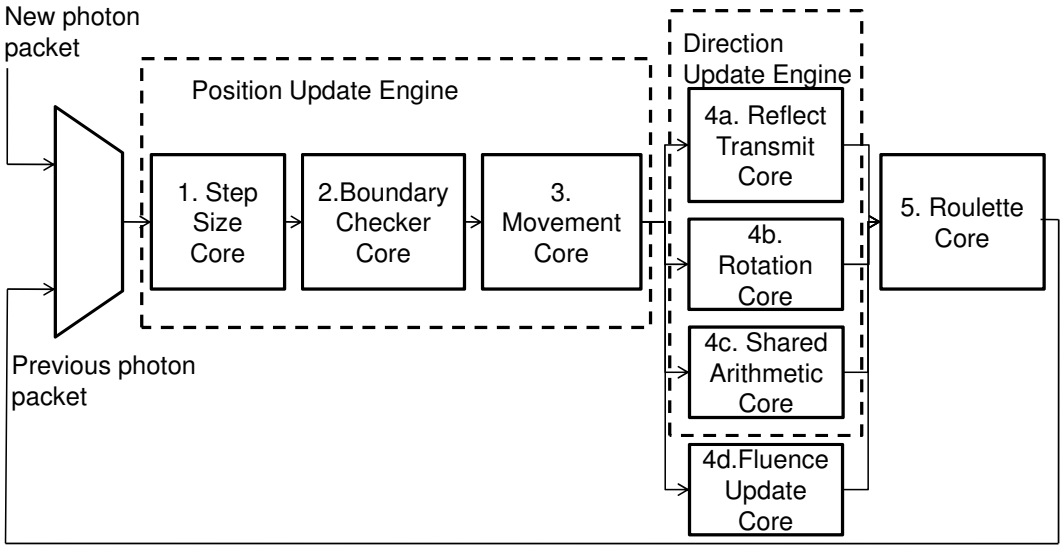


Figure 3.4: Pipelined architecture of the FBM hardware: Module 1 computes the step size using Eq. 2.1; Module 2 is based on Eq. 2.2 and Eq. 2.3; Module 3 uses the step size to compute the new position; Module 4a computes the internal reflectance (Eq. 2.7), determines whether the photon reflects or transmits, and updates the direction (Eq. 2.8) as well as the layer the photon resides; Module 4b models scattering and computes the new direction using Eqs. 2.4-2.6; Module 4c contains the arithmetic blocks shared across modules 4a and 4b; Module 4d computes the absorption (as described in Section 2.2.4) and records it in the on-chip memory; Finally, module 6 performs the survival roulette (Section 2.2.5).

stage (called the Roulette Core) determines whether a photon packet is still active, in which case it continues iterating at the beginning of the pipeline. Otherwise, a new photon packet is selected to immediately enter the pipeline.

Resource sharing is a key feature of this pipelined hardware design. To explain why the computational units between modules 4a (or the Reflect/Transmit Core) and 4b (Rotation Core) can be shared, notice that the computations in module 4a or module 4b are mutually exclusive, as discussed in Section 2.2.3. (Recall that if a photon hits a boundary, the scattering computations are skipped.) However, resource sharing along with parallel processing result in greater design complexity since the modules cannot be designed completely in isolation. For example, it is imperative that modules 4a,4b, and

4d all finish their operations within exactly 37 clock cycles to ensure data consistency. The number of stages was a major design decision as increasing this number leads to a higher overall clock speed, at the expense of greater register usage and greater design complexity.

The next two sections present further implementation details on the Fluence Update Core and the Direction Update Engine for illustrative purposes.

3.4.4 Pipeline Stages in the Fluence Update Core

This section shows how the individual pipeline stages were designed using the Fluence Update Core as an example. Note that the symbols/names used in the following diagrams correspond to the variable names used in the hardware description (or the Verilog code) shown in Appendix A. The original MCML code for the fluence update step is also included in the same Appendix for comparison.

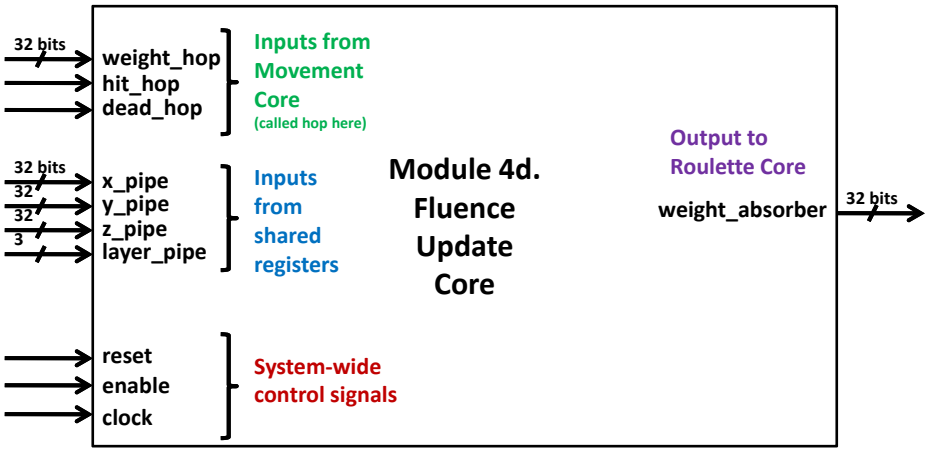


Figure 3.5: Simplified I/O interface for the Fluence Update Core

Interfacing with adjacent modules

Before specifying the individual pipeline stages, the inputs and outputs (I/O) of the hardware module must first be determined. A simplified I/O interface for the Fluence

Update Core is shown in Fig. 3.5. The memory interface for writing the results to the absorption array stored in the on-chip memory is omitted for clarity. Note that only the signal wires (similar to software variables) from the preceding Movement Core that are used by this module are connected as inputs. These include the current weight of the photon packet (called `weight_hop`), whether it has hit a boundary (`hit_hop`), and whether it has been terminated (`dead_hop`). Other inputs include the photon position (`x_pipe, y_pipe, and z_pipe`) and the current layer (`layer_pipe`) read from the shared pipeline registers, to be discussed next. There are also other control signals to run the circuit (`clock`), reset the circuit (`reset`) and enable/disable the circuit (`enable`). The output of this module is the updated weight of the photon packet (`weight_absorber`) and the differential weight absorbed which is written into the on-chip memory directly (not shown).

Shared and Internal Pipeline Registers

The key feature of a pipelined hardware design is a series of registers that store temporary, intermediate results between consecutive stages (as explained earlier in Fig. 3.2). These registers ensure that all stages can run concurrently.

Inside the Fluence Update Core, a total of 37 pipeline stages are present, as shown in Fig. 3.6. The choice of the number of stages is determined by many factors, including how the computations are laid out in the Direction Update Engine which runs synchronously with this core and must also have the same number of stages.

Outside the Fluence Update Core, a series of shared pipeline registers with the same pipeline depth (or 37 stages) are constructed to propagate the common photon data structure. Details on the data stored and propagated through these registers are provided in Table 3.1. The Fluence Update Core reads in the photon packet's position and the current layer from these registers. In addition, this module propagates its own modified data and intermediate results in a series of internal pipeline registers. These include the

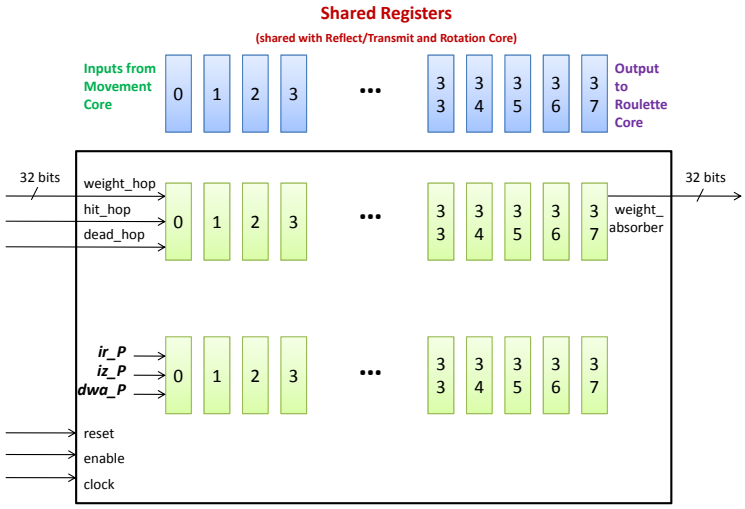


Figure 3.6: Overview of the pipeline inside the Fluence Update Core. Note that each numbered rectangle represents a pipeline register for storing intermediate values between stages.

location of absorption as specified by the indices to the absorption array (`ir_P, iz_P`) and the differential weight absorbed (`dwa_P`).

Stages 1 and 2 - Initialization and Computation of x^2 and y^2

The division of the computation into stages must take into account all potential dependencies (i.e., certain steps must be performed in a fixed order). For example, in the Fluence Update Core, the position of the photon packet must be converted from Cartesian into cylindrical coordinates before accumulating the absorbed weight into the $A[r][z]$ array. The radius is computed according to Eq. 3.2. However, the square root of r^2 shown in Eq. 3.2 cannot be performed until the sum of x^2 and y^2 is ready as shown in Eq. 3.1.

$$r^2 = x^2 + y^2 \tag{3.1}$$

$$r = \sqrt{x^2 + y^2} \tag{3.2}$$

With this in mind, the first two stages of the Fluence Update Core are responsible for initialization and for the computation of the square of x and y, as shown in Fig. 3.7.

Table 3.1: Examples of photon packet data stored in the shared pipeline registers

Name	Symbol	Bit Width
x coordinate	x	32
y coordinate	y	32
z coordinate	z	32
direction cosine (x)	μ_x	32
direction cosine (y)	μ_y	32
direction cosine (z)	μ_z	32
layer	*layer*	3
hit boundary?	*hit*	1
dead?	*dead*	1
weight	W	32

For stage 1, no computations are performed. For stage 2, the current x and y positions (`x_pipetemp` and `y_pipetemp`) are first read from the shared registers at the corresponding stage. Note that timing is critical and a mistake in coding can lead to mis-reading from the incorrect stage. Next, two multiplications are performed simultaneously within one clock cycle, with the use of two 32-bit by 32-bit multipliers. The 64-bit results for x^2 and y^2 are stored in two temporary registers named `x2_P` and `y2_P`, respectively. Finally, the data in the internal pipelined registers are propagated forward as before.

Stage 3 - Computation of r^2

To simplify the diagram, the propagation of internal register values is not explicitly drawn in Fig. 3.8. In Stage 3, a simple addition of x^2 and y^2 is performed using a 64-bit adder unit to calculate r^2, as shown earlier in Eq. 3.1. The result is stored in a register labelled `r2_P`. Note that as this is being performed, a new value from stage 2 is simultaneously computed. This is the basic mechanism or idea behind the pipelining technique.

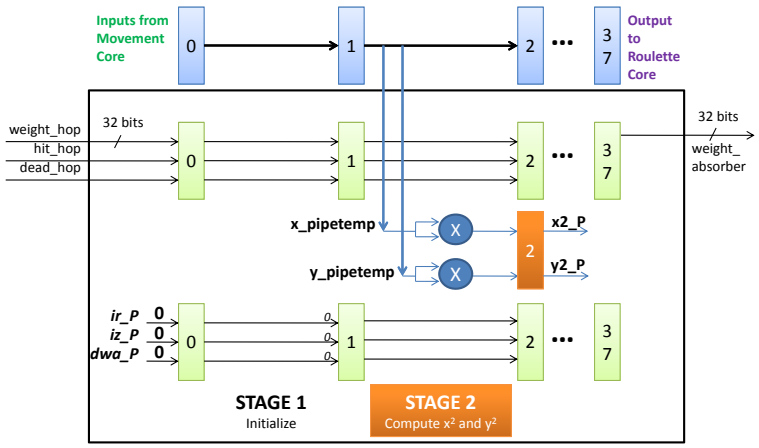


Figure 3.7: Pipeline Stages 1 and 2 inside the Fluence Update Core

Stage 4 - Computation of r and ΔW

Unlike previous stages, part of Stage 4 requires more than one cycle to complete due to the use of a square-root block to compute r from r^2, as illustrated in Fig. 3.9. Square-root blocks are relatively slow on FPGAs; therefore, to improve its clock speed, the pipelining technique is applied here as well. However, increasing the number of pipeline stages within the square-root block also increases resource usage. Based on empirical testing of this trade-off, a pipeline depth of 10 was chosen, which means that it takes 10 clock cycles before the result of the square-root operation (`r_P`) becomes ready. Correctly aligning this with the rest of the computation at the right stage is crucial as missing by 1 cycle can cause inconsistency in the simulation data.

This stage also concurrently computes ΔW (denoted `dwa`), using the formula below from Chapter 2:

$$\Delta W = W \frac{\mu_a}{\mu_a + \mu_s} \tag{3.3}$$

Note that $\frac{\mu_a}{\mu_a+\mu_s}$ is a constant for each layer and does not require re-computation every time. Hence, a look-up table (called `muaFraction` consisting of 5 entries for the 5 layers) was used instead to save a divider and an adder. To compute ΔW, the constant $\frac{\mu_a}{\mu_a+\mu_s}$ for the desired layer is first retrieved from the look-up table based on the current layer index

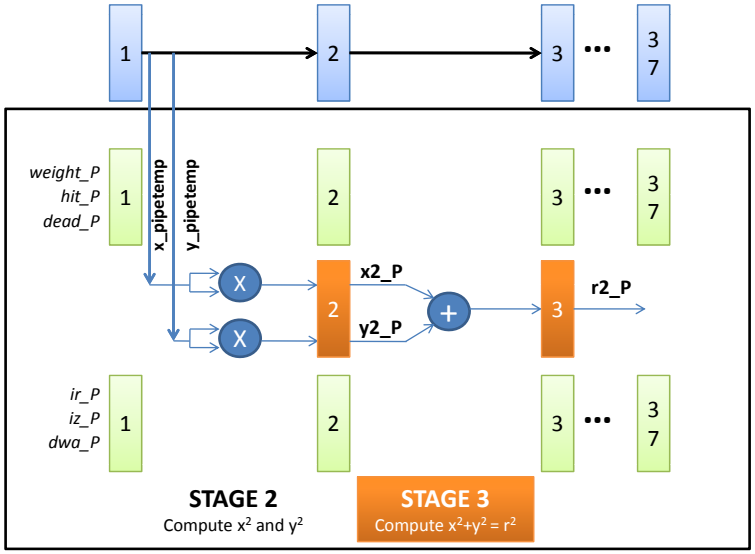


Figure 3.8: Pipeline Stages 2 and 3 inside the Fluence Update Core

(`layer_pipetemp`). It is then multiplied by the current weight W (named `weight_P` from the internal registers). The most significant 32 bits of the final product are stored in the internal pipeline registers called `dwa_P`. Finally, a number of corner cases also need to be considered. For example, if the photon has hit a boundary (as indicated by the signal `hit_P[3]`), then ΔW should be zero to ensure no weight is added to the array in the on-chip memory. Extra control circuitry or logic is required to handle such special cases throughout.

The stages described thus far illustrate several key differences with custom hardware design. The remaining stages, which involve handling a memory interface to the on-chip memory, follow a similar design methodology. The keen reader may refer to Appendix A for details on how the entire Fluence Update Core is implemented stage-by-stage.

3.4.5 Design Challenges for the Direction Update Engine

To further illustrate the complexity of the FBM hardware, the implementation of the Direction Update Engine (depicted in Fig. 3.4), which computes the scattering angle and

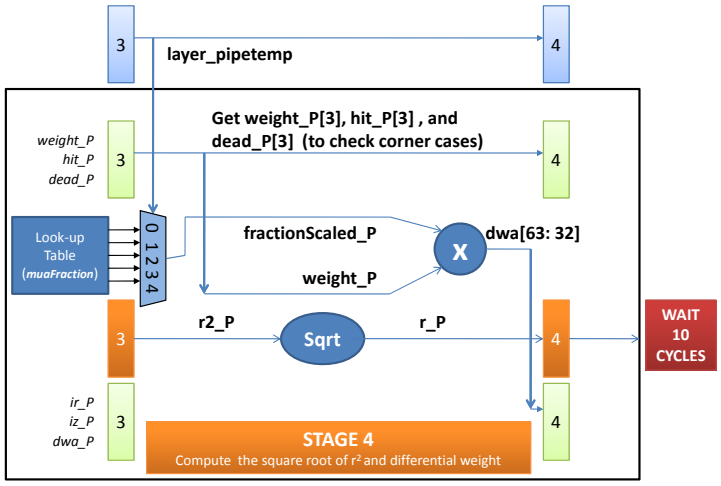


Figure 3.9: Pipeline Stage 4 inside the Fluence Update Core

updates the direction of the photon packet (Eqs. 2.4-2.6), is described as follows.

$$\mu_x' = \frac{sin(\theta)[\mu_x\mu_z cos(\psi) - \mu_y sin(\psi)]}{\sqrt{1-\mu_z^2}} + \mu_x cos(\theta) \tag{3.4}$$

A direct implementation of Eq. 3.4 would be very inefficient and would result in low clock speed and high resource usage for each of the three direction cosines (the formula for only one of which is reproduced above). The FPGA contains dedicated hard multipliers, but it does not contain other dedicated circuitry to perform division, square root or trigonometric functions. To efficiently evaluate trigonometric functions, look-up tables with pre-computed sine and cosine values are stored in the on-chip memory, which has a very fast (or 1 clock cycle) data access time. Division and square root operations are performed using Altera Quartus library blocks. As these computations are generally slow (as defined by the maximum clock speed), square root blocks are internally implemented as a 10-stage pipeline and dividers are split into 20 stages to increase the clock speed. The choice of the pipeline depth or the number of pipeline stages is a trade-off between logic usage and clock speed. A deeper pipeline generally translates to a higher clock speed (up to a certain limit), as shown by Fig. 3.10 using a 64-bit divider as an example; however, the additional stages increase logic usage (e.g., a 30-stage divider may require

~50% more logic compared to a 20-stage divider). A pipeline depth of 20 stages was chosen to achieve a desirable clock speed, while reserving resources for other modules.

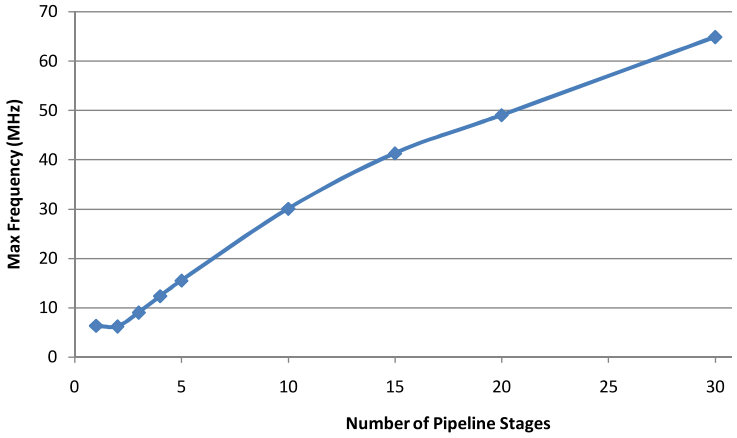


Figure 3.10: Effect of pipeline depth on the maximum clock speed of a 64-bit fixed point divider

3.4.6 Importance of Managing Resource Usage

Allocating the limited amount of FPGA on-chip resources effectively is critical to a high-speed digital hardware design. A small, compact design can be replicated multiple times to achieve more parallelism and can be tuned more easily for a higher clock speed. Together, these two factors can significantly impact the final performance of the hardware. Table 3.2 gives the resource usage statistics for each module in the pipelined hardware, including the logic element (LE) usage, key arithmetic blocks used, on-chip memory usage (in bits) and number of pipeline stages.

On-chip memory usage is critical since there are only 7.4 Mbits of space available on each Stratix FPGA for the TM-4. Most of this space is occupied by the absorption array $A[r][z]$. Although a larger $A[r][z]$ array (up to ~330 $\times$ 330 64-bit elements) can be stored in the on-chip memory, the array dimensions are fixed at 256 $\times$ 256 (which is a power of 2) to reserve some space for look-up tables and to reduce the amount of hardware needed in addressing the array. For example, a division by 256 can now be

implemented as a bit-shifting operation in binary representation, which is significantly less resource-intensive than a division.

Another important resource is the number of logic elements for implementing the computations in the design. Only the most resource-intensive arithmetic units are listed, including multipliers, dividers, and square root blocks. One important optimization, which cannot be readily implemented in software, is the sharing of resource-intensive computational units between the Reflect/Transmit Core (module 4a) and Rotation Core (module 4b). A total of 15 multipliers, 1 divider, and 1 square root block are shared between these two cores. A particularly important arithmetic block is the multiplier since there are only limited number of hard multipliers; additional multipliers are costly to build using soft logic. Therefore, multipliers and other costly arithmetic blocks such as dividers or square root blocks are shared as appropriate.

Finally, the number of stages within each module represents the number of clock cycles required for a photon packet to propagate through the module. Increasing the number of stages serves to decrease the complexity of each stage, thereby improving the clock speed. An example of using this technique is module 2, which lies in the critical path of the circuit. A deep pipeline with 60 stages was designed to divide the computation into finer steps, increasing the clock speed of this part of the circuit and that of the entire pipeline as a result. The key drawbacks include the need for many additional pipeline registers between stages which significantly increased the logic element usage and the extra design effort required due to the increased complexity involved in development and verification.

3.4.7 Trade-offs

Several important tradeoffs were made on the final hardware design. First, to maximize the on-chip memory space available to the look-up tables, the absorption array was limited to 256 by 256 elements in the r and z direction, respectively. Second, the number

Table 3.2: Allocation of on-chip resources on the TM-4 and the number of pipeline stages per module

Module	Logic Elements (LE)	Key Arithmetic Blocks	On-chip Memory (bits)	Number of Stages
1. Step Size Core	856	2 multipliers (16 DSP elements[b1])	32,768[c1]	1
2. Boundary Checker Core	31,868	3 multipliers and 1 divider (8,273 LE)	—	60[e]
3. Movement Core	3,905	3 multipliers (3,444 LE)	—	1
4a. Reflect/Transmit Core	1,355	—	65,536[c2]	37
4b. Rotation Core	3,190	—	589,824[c3]	37
4c. Shared Arithmetic Core	5,160	15 multipliers, 1 divider and 1 square root (120 DSP elements and 5,160 LE)	—	N/A
4d. Fluence Update Core	2,403	3 multipliers and 1 square root (24 DSP elements and 270 LE)	4,194,304[d]	37
5. Roulette Core	309	—	—	1
Total Used/ Total Available	64,147[a]/79,040	26 multipliers[b2], 2 dividers, 2 square root blocks (160/176 DSP elements and 17,147 LEs)	4.8/7.4 Mbits	100

[a] Other logic elements were used by 5 random number generators (557 LE), shared pipeline registers (9,545 LE), the TM-4 ports interface (170 LE), design wrappers and other control/arithmetic blocks (4,829 LE).

[b1] 8 DSP elements required per multiplier

[b2] 20 multipliers were implemented in 160 (out of 176) DSP elements while 6 multipliers were implemented in logic elements.

[c1] For a logarithm look-up table with 1,024 entries x 32 bits/entry = 32,768 bits

[c2] For 2 look-up tables used in the Fresnel calculations (also 1024 32-bit entries per look-up table)

[c3] For 4 trigonometric look-up tables (2 with 1,024 32-bit entries and another 2 with 8,192 32-bit entries)

[d] For the 64-bit $A[r][z]$ array with 256 x 256 elements

[e] Dividing this module into 60 stages resulted in high logic element usage (∼15,000 LE for pipeline registers alone)

of layers supported by the hardware was set to a maximum of five, also due to memory constraints. Even though the number of layers was fixed at a maximum of five, their optical properties and the dimensions of the voxels (dr and dz) can still be modified easily through the standard input simulation file format used in the MCML program.

3.5 Validation

For the purpose of validation and performance comparison, a skin model was selected as the simulation input to the MCML program. The tissue optical parameters presented in Table 3.3 are based on the light scattering study of tissues by Tuchin [52]. In Tuchin's model of the skin, five layers are used to represent the varying tissue optical properties from the epidermis to the dermis plexus profundus. The optical parameters for two wavelengths, λ=633 nm and 337 nm, were used. Also, the top and bottom ambient media were set to be air ($n = 1$, μ_a=0, and μ_s=0).

To test the accuracy and performance of the hardware system with different tissue optical parameters, the absorption coefficient and scattering coefficient were varied systematically in a separate experiment, as described in the next section. Note that the design was validated on both the TM-4 and DE3 board. Since only the communication interface was modified in the migration process, identical results were generated on both platforms. The results presented in this section can be reproduced on either platform.

3.5.1 FPGA System-Level Validation Procedures

System validation consisted of three phases. The first phase involved verifying the simulation outputs from the FBM hardware against the gold standard (or the original MCML software executed on an Intel Xeon processor). Since MC simulations are non-deterministic, it is important to separate the error introduced by the hardware implementation (including lookup tables and fixed-point conversion) from the statistical

Table 3.3: Optical properties of the five-layer skin tissue at λ=633 nm (and 337 nm in brackets) [52]

Layer	$\mu_a(cm^{-1})$	$\mu_s(cm^{-1})$	g	n	Thickness(cm)
1. epidermis (top)	4.3 (32)	107 (165)	0.79 (0.72)	1.5	0.01
2. dermis	2.7 (23)	187 (227)	0.82 (0.72)	1.4	0.02
3. dermis plexus superficialis	3.3 (40)	192 (246)	0.82 (0.72)	1.4	0.02
4. dermis	2.7 (23)	187 (227)	0.82 (0.72)	1.4	0.09
5. dermis plexus profundus	3.4 (46)	194 (253)	0.82 (0.72)	1.4	0.06

uncertainty inherent in an MC simulation. In other words, a fair comparison between the MCML software and the FBM implementation can only be obtained by considering the variance in the output of the MCML simulation, which is a 2-D absorption array scoring the photon absorption probability density [cm^{-3}] as a function of radius and depth. The resulting absorption probability density map is also a function of tissue optical properties. To quantify the difference between these arrays, the relative error $E[i_r][i_z]$ between corresponding elements was computed using the following formula:

$$E[i_r][i_z] = \frac{|A_s[i_r][i_z] - A_h[i_r][i_z]|}{A_s[i_r][i_z]} \tag{3.5}$$

where A_s is the gold standard absorption array produced by the original MCML software after launching 100 million photon packets and A_h contains the corresponding elements in the absorption array produced by the FBM hardware. To visualize the distribution of the relative error, a 2-D colour map showing the relative error (in percent) as a function of position (r,z) was generated. For comparison, a reference colour map depicts the relative error in the output from 2 gold standard absorption arrays to account for the statistical uncertainty between MCML simulation runs.

To summarize the effect of varying the number of photon packets, the mean relative error (Eq. 3.6) was computed by averaging the relative error in all elements in the

absorption array with values above a selected threshold of 0.00001 cm^{-3}:

$$E_{ave}[i_r][i_z] = \frac{\sum_{i_z=0}^{n_z} \sum_{i_r=0}^{n_r} E[i_r][i_z]}{n_r n_z} \tag{3.6}$$

where E_{ave} is defined as the mean relative error, $E[i_r][i_z]$ is the relative error for each element (as defined in Eq. 3.5), and n_r=256 and n_z=256. The threshold is necessary since relative error is undefined when $A_s[i_r][i_z]$ (or the software MCML output) is zero. (Note that as the same threshold is used for quantifying the mean relative error between the hardware and the gold standard and for calculating the error between two gold standard runs.) This analysis enables the quantification of the impact of look-up tables and fixed-point conversion in the hardware implementation. Photon packet numbers ranging from 10^5 to 10^8 were simulated.

To further characterize the hardware system with varying tissue optical parameters, the performance and relative error based on 10^8 photons were analyzed as a function of the target albedo. In a single-layer geometry, the target albedo, defined as $\mu_s/(\mu_a + \mu_s)$, was systematically varied from 0.50 to 0.96 in order to investigate the effects of tissue optical property on both the speedup and error.

The third phase for system-level validation of the FPGA-based hardware design involved analyzing the effect of the error within the context of PDT treatment planning. Isofluence maps were generated after launching 10^8 photon packets with the FBM hardware. The relative shift in the position of the isofluence lines was analyzed by comparing against the gold standard MCML output.

It is important to emphasize that in all phases of the validation, a much higher resolution (dr=0.01 cm and dz=0.002 cm, comparable to the dimension of single cells) than clinically required for PDT treatment planning (typically mm resolution) was used to increase the sensitivity for detecting any systematic errors. In addition, the voxel resolution was set based on the transport mean free path, defined as $1/[\mu_a + \mu_s(1-g)]$, which is around 0.01 to 0.04 cm using the optical properties from Table 3.3. While a vertical resolution (in the z direction) of 0.002 cm probably over-samples the spatial

distribution, it enables the visualization of changes within each tissue layer, particularly in the top layer which has a thickness of only 0.01 cm. A similar rationale for choosing voxel resolution was used in the multi-layered skin model for pulse oximetry by Reuss et al [80].

3.5.2 Results

Figures 3.11 and 3.12 show the distribution of the relative error for 10^5 and 10^8 photon packets respectively, using Tuchin's skin model at λ=633 nm as input. In both cases, the accuracy of the results produced by the FBM hardware was comparable to that of the MCML software, as demonstrated by the similarity between the two error distributions [Figs. 3.11(a) and 3.11(b)]. The statistical uncertainty decreased for the simulation that used 100 million photon packets, as indicated by the expansion of regions within the r, z plane showing less than 5% error in Fig. 3.12. This is expected as the variance in Monte Carlo simulations decreases by $1/\sqrt{n}$ where n represents the number of photon packets. Figure 3.12(a) also shows some slight differences of about 1-2% (manifesting as an S-shaped region with lower error) in the region within a radius of 0.5 cm (or the high fluence region). Further analysis revealed that this S-shaped pattern can be eliminated by replacing the random number generator in the original MCML software with the version implemented in hardware (namely the Tausworthe generator [79]). The disappearance of the S-shaped pattern with the use of the same random number generator (or the Tausworthe generator) shows that the minor deviation observed was due to the statistical differences in the random number sequence generated by two different random number generators [Figs. 3.12(c) and 3.12(d)].

To analyze the effect of photon packet number on the simulation accuracy, the mean relative error is plotted in Fig. 3.13. Figure 3.13(a) shows that the mean relative error of the results generated by the FBM hardware closely tracked the mean relative error of the MCML gold standard, both decreasing as the number of photon packets increased. Figure

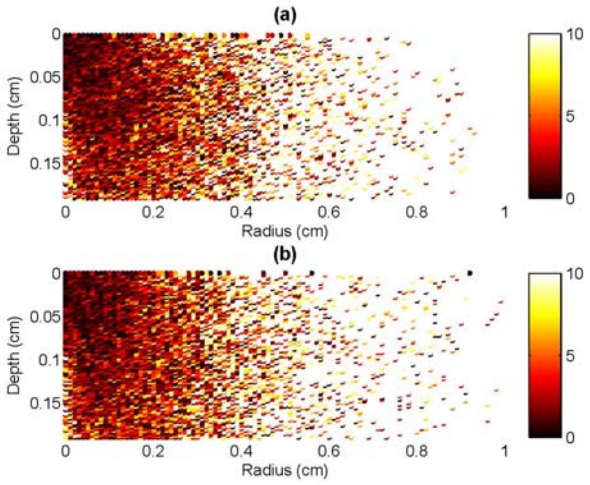


Figure 3.11: Distribution of relative error as a function of radius and depth using 10^5 photon packets (at λ=633 nm): (a) The FBM hardware (10^5 photon packets) versus the MCML software (10^8 photon packets) and (b) MCML (10^5 photon packets) versus MCML (10^8 photon packets). The bar represents percent error from 0 to 10%. Values above 10% are represented by the same colour as the maximum value in the colour scale.

3.13(b) shows the impact of converting from double-precision floating point operations to fixed point operations combined with the impact of the use of look-up tables on the relative error. As shown by the plot, the conversion introduced an increase in relative error of 0.2-0.5%, which has a negligible effect on PDT treatment planning as explained later in more details.

In the second phase of the validation, the mean relative error as a function of the albedo was plotted [Fig. 3.14(a)]. The results show that for albedo values between 0.7 and 1.0, the increase in error was only 0.5-1%, while for albedo values below 0.7, the added error was up to 2%. This increase was caused by the significant reduction in the number of non-zero absorption array elements. For example, at an albedo of 0.90, there were 11407/65536 non-zero elements, but only 351/65536 non-zero elements at an albedo of 0.5. This high proportion of zero elements is due to the small voxel size used (dr=0.01 cm and dz=0.002 cm).

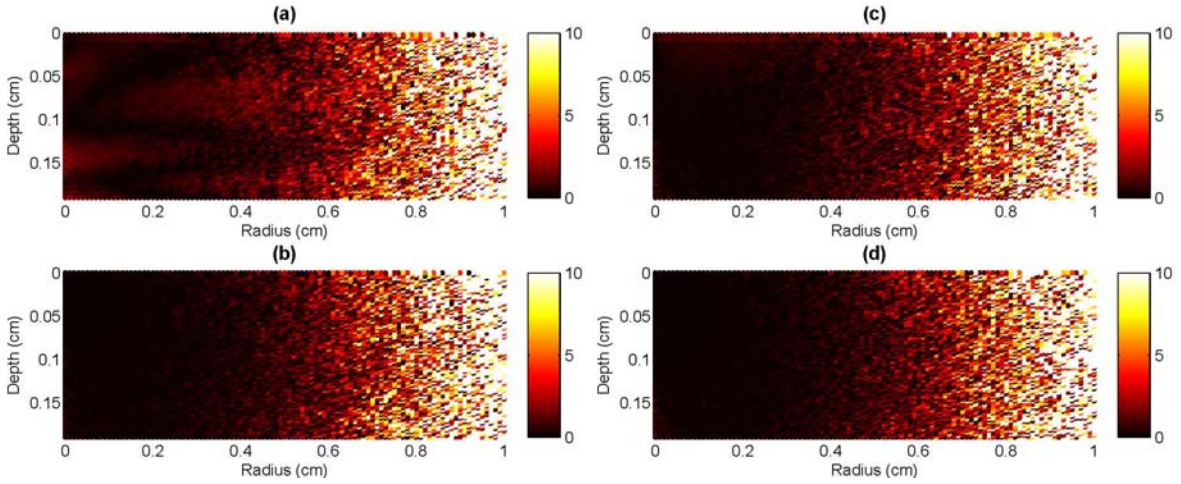


Figure 3.12: Distribution of relative error as a function of radius and depth using 10^8 photon packets (at λ=633 nm): (a) The FBM hardware versus the MCML software (run 2), (b) MCML (run 1) versus MCML (run 2), (c) FBM versus MCML with Tausworthe generator (run 2), and (d) MCML (run 1) versus MCML (run 2) both with Tausworthe generator. Colour bar represents percent error from 0% to 10%.

To investigate the impact of the 1-2 % additional error within the context of PDT treatment planning, the isofluence lines for the impulse response based on simulation input parameters from Table 1 were plotted (Fig. 3.15). The isofluence lines produced by the FBM hardware and the MCML software matched each other well. The shift in the position of the isofluence lines was only noticeable for fluence levels at 0.00001 cm^{-2}, which is 8 orders of magnitude smaller than the fluence near the centre – 1000 cm^{-2}. The detected shift was only around 0.1 mm, which is of little significance in PDT treatment planning.

3.6 Performance

A common metric for measuring the performance improvement of a parallel or accelerated implementation is the *speedup*, which is defined as the ratio between the *sequential*

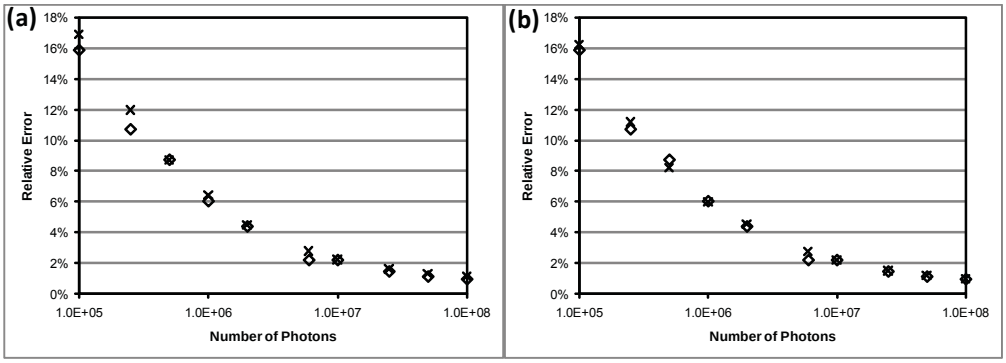


Figure 3.13: Mean relative error as a function of the number of photon packets simulated (at λ=633 nm). The horizontal axis is in logarithmic scale: $\Diamond$, mean relative error between two independent MCML runs. (a) $\times$, mean relative error comparing the results produced by the FBM hardware and the MCML software and (b) $\times$, mean relative error of the results produced by the C program modelling look-up tables and fixed-point operations compared to MCML which uses double-precision floating point operations. Each point represents the average obtained from four simulation runs.

execution time ($t_{sequential}$) and *parallel execution time* ($t_{parallel}$):

$$Speedup = \frac{t_{sequential}}{t_{parallel}} \tag{3.7}$$

The parallel execution time of the FBM hardware was measured on two different FPGA-based platforms: the TM-4 platform, which contains four Stratix FPGAs, and the DE3 board, which contains a single Stratix III FPGA. The detailed specifications for both are compared in Table 3.4. A key difference between the Stratix and Stratix III FPGA is the Integrated Circuit (IC) process technology used to fabricate each silicon chip. The Stratix III FPGA is manufactured using the 65 nm lithographic process, which results in a smaller transistor size and higher transistor density compared to the Stratix FPGA fabricated using the 130 nm process technology. The higher transistor density translates to a larger number of logic elements and a higher clock speed for designs on the Stratix III FPGA compared to the Stratix FPGA.

For fair comparison, a processor or CPU manufactured using a similar process tech-

Table 3.4: Specifications of the TM-4 and DE3 FPGA platforms

	TM-4	DE3-150
FPGA Device[a]	Four Stratix (EP1S80) FPGAs	One Stratix III (EP3SL150) FPGA
Logic Elements (LE)	79 K/FPGA	142 K[b]
DSP Elements[c]	176/FPGA	384
On-chip Memory	7.4 Mbits/FPGA	5.6 Mbits
IC Process Technology[d]	130 nm	65 nm

[a] Note that the Stratix FPGA used by the TM-4 is at the top of its class. A more advanced Stratix III EP3SL340 FPGA (available on the DE3-340 model) contains 338 K LEs, 576 DSP elements, and 16 Mbits of on-chip memory

[b] The Stratix III FPGA uses a new architecture called Adaptive Logic Module (ALM). This device contains 56,800 ALMs, but for easier comparison, the equivalent number of logic elements is reported.

[c] The DSP elements on Stratix and Stratix III FPGAs are different. For example, a 32-bit x 32-bit multiplier requires 8 DSP elements (9-bit x 9-bit) on Stratix FPGAs, but only 4 DSP elements (18-bit x 18-bit) on Stratix III FPGAs.

[d] Integrated Circuit (IC) process technology refers to the silicon chip fabrication technology, as defined by the "size" of transistors (in nm) created in the lithographic process. (The smaller it is, the higher the transistor density and clock speed.)

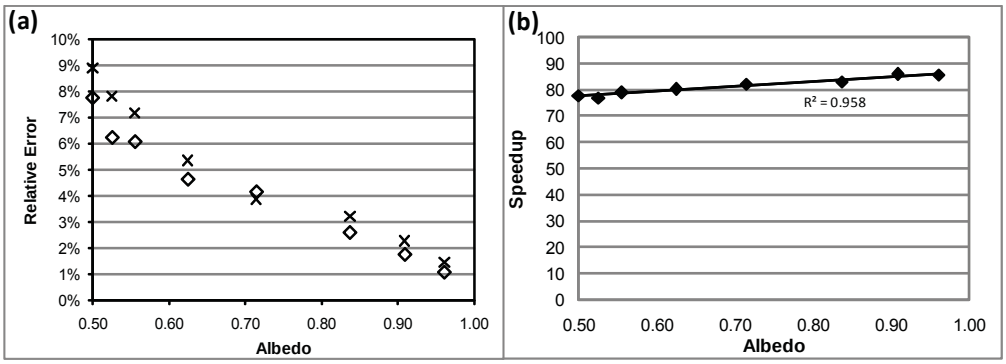


Figure 3.14: (a) Mean relative error with varying albedo (10^8 photon packets): ◊, mean relative error between two independent MCML runs; ×, mean relative error between the results produced by the FBM hardware and the MCML software. (b) Speedup at different albedo values (10^8 photon packets). Each point represents the mean obtained from four simulation runs.

nology was selected for measuring the sequential execution time. Namely, an Intel Xeon processor built with the 130nm process technology was compared against the 130 nm Stratix FPGA, while a 65nm Intel Xeon processor was compared against the 65 nm Stratix III FPGA. The specifications of each platform are listed in Table 3.5.

Table 3.5: Specifications of two Intel-based server platforms

	Platform 1 (130 nm processor)	Platform 2 (65 nm processor)
Processor	Intel Xeon 3.06-GHz CPU (Pentium 4)	Intel Xeon 5160 3-GHz CPU (Dual-Core)
CPU Cache[a]	512 kB	4 MB
Memory	2 GB of RAM	8 GB of RAM
C Compiler	gcc 3.2.2	gcc 4.1.2

[a] A larger cache (L2 cache) – a fast CPU memory – can reduce the number of expensive memory accesses by temporarily storing frequently used data.

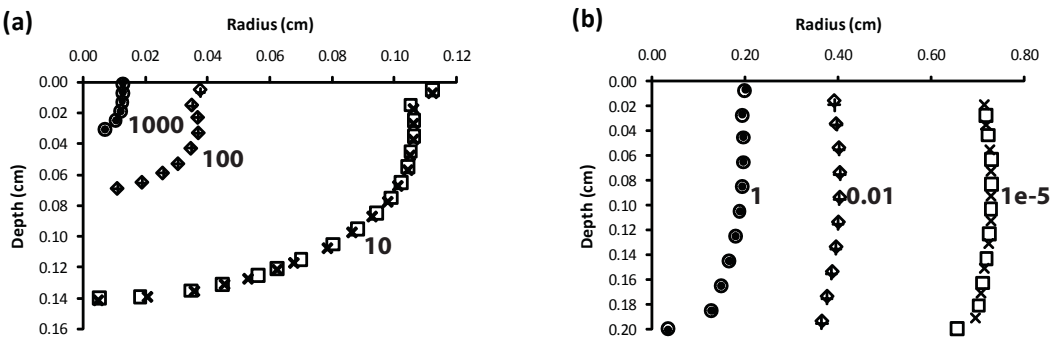


Figure 3.15: Comparison of the isofluence lines for the impulse response generated by the FBM hardware and the MCML software using 100 million photon packets (at λ=633 nm): $\circ$, $\Diamond$, and $\square$, results from MCML; and $\bullet$, $+$, and $\times$, results from FBM. (a) Isofluence lines for fluence levels at 1000, 100, and 10 cm^{-2}, as indicated on the figure and (b) isofluence lines for fluence levels at 1, 0.01, and 0.00001 cm^{-2}.

3.6.1 Multi-FPGA Platform: TM-4

The simulation time on the FBM hardware implemented using the TM-4 platform was compared to the original MCML software executed on a single 3-GHz Intel Xeon (Pentium 4) processor with a process technology of 130nm. This test platform is called Platform 1 in Table 3.5. For a complete end-to-end comparison, the runtime includes file I/O, system initialization, the MC simulation, and all pre-processing/post-processing operations to generate the final simulation output file. The MCML software was compiled using full compiler optimizations (gcc -O3 optimization flag [81]).

As shown in Table 3.6, the runtime of the MC simulation using 100 million photon packets was reduced from over 2.5 h in software to approximately 2 min in hardware for the skin model at λ=633 nm. The overall speedup was 78 times greater including a data transfer time of 8 s. Using the tissue optical properties at λ=337 nm from Table 3.7, the overall speedup was 66 times, mainly due to the much shorter execution time and hence the relative importance of the data transfer time. However, the data transfer rate was far from expected due to a known issue in the communication channel on the TM-4 prototyping system. Normally, the communication channel [host-to-FPGA PCI

Table 3.6: Runtime of the MCML software and the FBM hardware at 10^8 photon packets, averaged over four independent simulation runs. (Input from Table 3.3 at λ=633 nm.)

Device	Clock Speed	Simulation Time (s)	Data Transfer Time (s)	Overall speedup	Speedup excluding data transfer
Intel Xeon	3.06 GHz	9150	0	1	1
TM-4	41 MHz	109	8	78 +/- 1	84 +/- 1

Table 3.7: Runtime of the MCML software and the FBM hardware at 10^8 photon packets, averaged over four independent simulation runs. (Input from Table 3.3 at λ=337 nm.)

Device	Clock Speed	Simulation Time (s)	Data Transfer Time (s)	Overall speedup	Speedup excluding data transfer
Intel Xeon	3.06 GHz	3100	0	1	1
TM-4	41 MHz	39	8	66 +/- 1	80 +/- 1

(peripheral component interconnect) bus] supports a bandwidth of 266 MB/s for writes to the FPGA and 154 MB/s for reads from the FPGA to the host [76]. Currently, it takes 8 s to transfer 610 kB of data. Hence, the use of commercial prototyping platforms with fully functional communication channels should yield a net 84 times speedup for the λ=633 nm case and 80 times speedup for the λ=337 nm case without any modifications to the design. Fig. 3.14(b) shows that as the albedo increased, the speedup increased from 77 to 87 times, since the MCML software executes more expensive computations for calculating the scattering angle and computing internal reflectance at higher albedo. (A photon packet also takes a greater number of random walks before it is terminated by the survival roulette). The average speedup was 80 times with the current TM-4 platform running at a clock speed of 1/75 times compared to that of the Xeon processor.

Table 3.8: Performance comparison of Stratix, Stratix III, and 3 GHz Xeon 5160 processor for the MCML simulation with 10^8 photon packets. (Input parameters from Table 3.3 at λ=633 nm.)

Platform	Clock Speed	Simulation Time (s)	Speedup
Xeon 5160 processor	3 GHz	6102	1
Stratix III (DE3)	80 MHz	220	28
4 x Stratix (TM-4)	41 MHz	109	56[a]
1 x Stratix (TM-4)	41 MHz	436*	14*
4 x Stratix III (stacked DE3)	80 MHz	55*	112*

* Projected values

[a] This speedup is lower than in Table 3.6 due to the faster Xeon CPU used for baseline comparison.

3.6.2 Modern FPGA Platform: DE3 Board

On a single Stratix III FPGA, the FBM hardware can operate at a higher clock frequency (80 MHz), achieving a 28-fold speedup compared to a 3-GHz Xeon 5160 dual-core processor (utilizing one processor core). Note that this test platform (called Platform 2 in Table 3.5) contains an Intel processor with 65nm process technology, which is identical to that of the Stratix III FPGA. The on-chip simulation time was 6102 seconds for the processor and 220 seconds for the FPGA. A third-party host-to-FPGA communication software package that is still under development was used to migrate the TM-4 design over to the DE3. This software package has a performance problem and requires over three hours to transfer one megabyte of data (although the integrity of the data was not affected). The total amount of data that needs to be transferred to and from the device is less than a megabyte and should take less than one second given a well-designed communication interface. For this reason, the data-transfer overhead was not included in the reported simulation time.

Table 3.9: Resource utilization of one instance of the FBM design on a single Stratix (TM-4) and on a single Stratix III device (DE3).

FPGA Device	Logic Block Usage[a]	DSP Elements[a]	On-chip Memory	Clock Speed
Stratix (130nm)	64,147/79,040 LEs	160/176	4.8/7.4 Mbits	41 MHz
Stratix III (65nm)	31,185/56,800 ALMs	92/384	4.8/5.6 Mbits	80 MHz

[a] The Stratix III device uses a different FPGA architecture compared to the Stratix device. The Stratix III FPGA contains Adaptive Logic Modules (ALMs) instead of Logic Elements (LEs) for soft logic and it also contains different kinds of DSP elements for multipliers.

3.7 Resource Utilization

Table 3.9 shows the resource utilization (including the logic element, DSP element, and on-chip memory usage) and clock speed of the FBM hardware on a Stratix FPGA device (Altera part number: EP1S80F1508C6) and a modern Stratix III FPGA device (EP3SL150F1152C3). Note that the Stratix and Stratix III devices contain different types of logic blocks and DSP elements. Although the FBM implementation only occupies about 55% of the logic elements and 24% of the DSP elements on the Stratix III device, the on-chip memory usage is 86%.

The on-chip memory usage and the number of occupied logic elements currently limit the number of replicas of the design to one. The use of external memory modules, together with a larger FPGA with more logic elements (offered by more advanced models of the DE3 board family), will allow multiple instances of the pipelined MCML hardware to be created, which translates to a proportionate increase in speedup. Also, the DE3 board which houses the current Stratix III device was designed to be stacked, allowing multiple Stratix III FPGAs to be combined into a single system. This offers the possibility of potentially performing the same MCML computation in a cluster of FPGAs.

3.8 Power

While power consumption is not of much concern in a hospital setting with the proper infrastructure, power consumption is an important engineering metric that can be used to show the merit of an FPGA-based implementation, compared to a cluster of processors. Also, power consumption becomes an essential metric in a cluster configuration, especially when hundreds of FPGAs or CPUs are used for complex simulations such as those required for treatment planning. Specifically, the *power-delay product* (PDP) is commonly used to measure the efficiency of different implementations. To compute the power-delay product, the power consumption and the simulation time or *delay* are multiplied together.

In the following comparison, only the power consumed by the processor or FPGA was considered; off-chip memory, network, and disk power were ignored. The thermal design power of the dual-core Xeon 5160 processor is specified by Intel to be 80W [82], and half of this value is taken as the power consumption for a single core. Although 40 W may be pessimistic for the single core, this value was chosen because the processor will be heavily utilized during the MCML computation. The FPGA power consumption was determined using the PowerPlay Power Analyzer tool in Quartus II version 8.1 [83]. The default settings for PowerPlay were used, in which the input pin toggle rate was set at the default 12.5%. Note that although most of the inputs remain almost constant during the simulation of any large number of photon packets, the default settings were chosen to give a more conservative estimate. Table 3.10 shows that the FBM hardware on the Stratix III achieved a 716-fold better power-delay product than a single-core 3 GHz Xeon 5160 processor. The normalized PDP represents the energy efficiency ratio of the Stratix III implementation. Note that a hypothetical cluster of 28 processors that matches the performance of the Stratix III FPGA will have the same 716-fold worse power-delay product compared to the FPGA. In other words, the Stratix III implementation will remain 716-fold more energy efficient regardless of the number of processors present.

Table 3.10: Power-delay product (PDP) of Stratix III, 3 GHz Xeon processor, and CPU cluster for the MCML simulation using 100 million photon packets at λ=633 nm. Delay values are extracted from Table 3.8.

Machine	Power (W)	Delay (s)	PDP (kJ)	Normalized PDP
Stratix III FPGA	1.55	220	0.341	1
Xeon processor (using 1 core)	40.0	6102	244	716
CPU cluster (28 cores)	1120	218	244	716

3.9 Summary

Custom digital hardware with a 100-stage pipeline was designed and validated for the MCML computations on two FPGA-based platforms – the TM-4 platform and the DE3 board. The hardware performed the MC simulation on average 80 times faster on the TM-4 platform with four Stratix FPGAs than the MCML software executed on a 3-GHz Intel Xeon processor. On the modern DE3 board with a single Stratix III FPGA, a 28-fold speedup was achieved compared to a different 3-GHz Xeon processor (manufactured using a more advanced process technology), while reducing the energy consumption by 716 times compared to a CPU-based computing cluster. Multiple DE3 boards can also be stacked to form a modern multi-FPGA platform that allows for the replication of the pipelined hardware across the FPGAs. The scalability of the custom hardware solution was demonstrated using the TM-4 platform in this thesis. In terms of accuracy, the isofluence contours generated by the hardware matched the software counterpart well, showing only a 0.1 mm shift for fluence levels as low as 0.00001 cm^{-2} in a skin model.

Although the FPGA-based approach potentially offers a scalable, low-power solution for PDT treatment planning, a number of challenges remain due to the complexity of designing and modifying custom hardware. The next chapter explores an alternative route that has recently captured the interests of the scientific computing community.

Chapter 4

GPU-based Acceleration of the MCML Code

This chapter explores a second method of accelerating the MCML code: programming commodity graphics processing systems. The first two sections provide the motivation for using graphics hardware (GPU) and the obstacles faced by other groups in this area. The new GPU programming paradigm is introduced in Section 4.3, followed by the implementation details of the GPU-accelerated MCML program in Section 4.4. For readers interested in the performance and validation results, please refer to Section 4.5 and Section 4.6. Part of this chapter has been presented at the European Conferences on Biomedical Optics [84].

Note that this chapter must first introduce the key hardware terminology for understanding graphics processing hardware, since this understanding was instrumental to the successful acceleration of the MCML code. Similarly, for other related applications, this learning curve is required to fully utilize this emerging scientific computing platform.

4.1 Graphics Processing Units

The proliferating gaming and entertainment industries have driven the rapid evolution of graphics processing units (GPUs). Since the GPU is specialized for processing high frame rate, high-definition graphics, it is well-suited for computationally intensive, parallelizable applications. The significant increases in floating point computational power and memory bandwidth further make the GPU an attractive platform for scientific computing [85].

GPU-accelerated scientific computing is becoming increasingly popular with the release of an easier-to-use programming model and environment from NVIDIA (Santa Clara, CA), called CUDA, short for Compute Unified Device Architecture [85]. CUDA provides a C-like programming interface for NVIDIA GPUs and it suits general-purpose applications much better than traditional GPU programming languages. However, performance optimization of a CUDA program requires careful consideration of the GPU architecture to exploit the full computational potential of the GPU. In this chapter, the exciting performance achieved for the acceleration of the MCML code is reported. More importantly, the insights gained through experimenting with different parallelization and optimization schemes are discussed, which reveal the unique challenges in CUDA programming and the subtlety of the NVIDIA GPU architecture.

4.2 Related Work

There are a number of related works that have used GPUs to accelerate Monte Carlo simulations. In terms of previous attempts to use GPUs for MC-based photon simulations, Alerstam et al. reported ∼1000x speedup on the NVIDIA GeForce 8800GT graphics card, compared to an Intel Pentium 4 processor, for the simulation of time-resolved photon migration in a homogeneous, semi-infinite geometry [86]. A 1-D array was used to generate the histogram for the time of flight of photons. Simultaneous to my efforts, the same group released a CUDA-based implementation of the MCML code called CU-

DAMCML (beta version), reporting an order of magnitude lower speedup (or $\sim$ 50-60x compared to an Intel Core i7 CPU) when absorption is recorded in a multi-layered geometry. Note that absorption is scored in a 2-D array due to radial symmetry. Based on this trend, a 3-D absorption array along with a voxel-based 3-D tissue geometry could cause significant performance degradation without gaining a deeper understanding of the performance bottleneck.

The work presented in this chapter proposes a different approach to handle the inefficiency in the scoring of absorption and addresses the question of how various optimizations can dramatically affect the performance of MC-based simulations for photon migration on NVIDIA GPUs.

4.3 CUDA-based GPU Programming

This section reviews the fundamentals of CUDA programming on NVIDIA GPUs. Only the technical terms necessary for understanding the subsequent sections are introduced. For a full description on NVIDIA GPU and CUDA, readers may consult the CUDA programming guide [85].

4.3.1 GPU Hardware Architecture

The underlying hardware architecture of a NVIDIA GPU is illustrated in Fig. 4.1 [85]. In fact, this architecture is common in a growing list of CUDA-enabled NVIDIA graphics cards to ensure compatibility with CUDA code. The remainder of this section gives a brief overview of the architecture using the NVIDIA GeForce GTX 280 GPU as an example. Note that another graphics card used in this work, called NVIDIA GeForce GTX 295, contains 2 GPUs and each GPU is similar to that on GTX 280. To understand the programming paradigm for NVIDIA GPUs in the next section, two key aspects must be explained, including the unique layout of processors and the memory hierarchy.

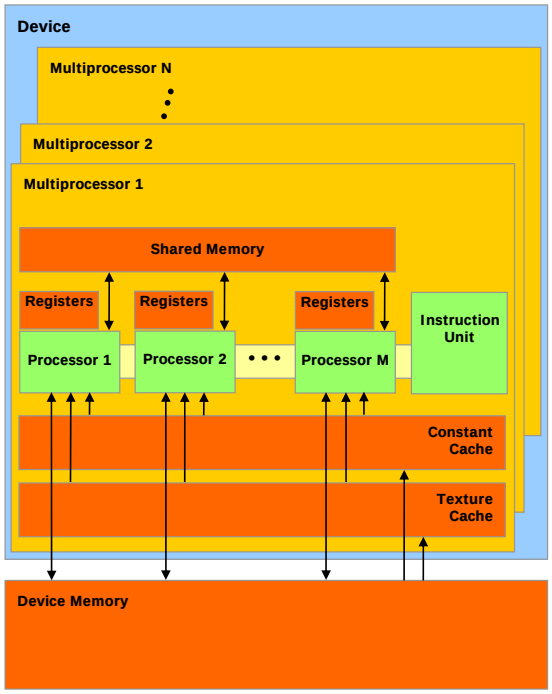


Figure 4.1: Hardware architecture of a NVIDIA GPU (N=30, M=8 for GTX 280) [85]

Layout of Processors

The GTX 280 GPU contains N=30 *multiprocessors*, each of which contains M=8 scalar processors (SPs). (GTX 295 has 2 GPUs and hence a total of 60 multiprocessors or 480 SPs.) The processor clock frequency or *clock speed* for each SP is 1296 MHz for GTX 280 and 1242 MHz for GTX 295. Typically, the clock speed for a modern CPU is around 3 GHz or 3000 MHz. However, apart from the clock speed, the performance of a processor is also determined by how efficiently instructions are executed or how much useful work can be executed in each hardware clock cycle. For the NVIDIA GPU, each multiprocessor executes instructions using a mode called *single-instruction, multiple-thread* (SIMT). A *thread* contains a parallel unit of work that is performed by a sequence of processor instructions. For example, a thread can process a group of pixels in a medical image or simulate a group of photon packets in the MCML algorithm. For the purpose of this discussion, SIMT means that all SPs within a multiprocessor always execute the

same instruction, but on potentially different data (or different photon packets for the MCML code). The main implication of the SIMT architecture is that a GPU with 240 SPs cannot be viewed as 240 independent processors; instead, it should be considered as 30 independent processors that can perform 8 similar computations at a time. This has a significant implication on how MC-based programs need to be written for high efficiency on NVIDIA GPUs. In contrary, this is typically not a key consideration when programming MC simulations on multi-processor computer clusters, as each CPU or processor *can* execute independently.

Memory Hierarchy

Second, the different layers and types of memory on the GPU must be understood by the programmer, as there is a significant difference in memory access time. The outermost layer, which is also the largest and slowest, is the off-chip *device memory* (also known as *global memory*). It can be used to communicate with the host computer processor or CPU and to store large quantities of data. It is typically at least 1 GB in size on modern GPUs, but it is relatively slow, requiring around 600 clock cycles. This is called the memory access latency, which is defined as the amount of time in clock cycles needed to retrieve the first segment of data from memory after an initial request. Note that the memory bandwidth, or the data transfer rate sustained after the initial access latency, is still considerably high. The peak memory bandwidth, though rarely attained, is claimed to be up to 141.7 GB/s for GTX 280, so that a single transfer of hundreds of megabytes of data to the GPU (such as for image processing applications) is usually not a big issue [85]. Closer to the GPU are the various kinds of fast, on-chip memories, including *registers* with typically single clock cycle of access latency, *shared memory* at close to register speed, and a low-latency cache for *constant memory* and *texture memory*, as shown in Fig. 4.1. Although on-chip memories are fast, they are limited in storage space. In total, there are 16,384 registers, each with 32 bits of storage, per multiprocessor for

Table 4.1: Mapping the key MCML variables to various GPU memories for high performance

Variable	GPU Memory	Rationale
Photon data structure[a]	Registers	Independently accessed (private to each thread)
Random number seeds	Registers	Small and frequently accessed
Absorption array $A[r][z]$	Global memory	Large array (accessed by all threads)
Selected region of $A[r][z]$*	Shared memory	Faster access to the absorption array
Layer data structure[b]	Constant memory	Read-only data
Input file data structure[c]	Constant memory	Read-only data

[a] Contains the position, direction cosines, layer, step size, weight, and other temporary variables.

[b] Contains read-only tissue optical properties per layer, including μ_a, μ_s, g, n, and the z-coordinates bounding each layer. A total of 20 layers can be supported with 8 kB of constant cache.

[c] Contains read-only simulation input parameters, such as dr, dz, nr, nz, and the number of photons.

* Note that the small size of the shared memory (only 16 kB) makes it crucial to choose the cached region carefully. This optimization is discussed in Section 4.4.

modern graphics cards (with compute capability 1.2 or above [85]). Although it may seem that there are many registers, these registers are distributed among up to a few hundred threads to store thread-private temporary variables. Therefore, if 512 threads are launched, each thread can only be allocated 32 registers. As for the shared memory, only 16 kB per multiprocessor are available and this space is useful for communication between the SPs. Finally, an 8 kB constant cache per multiprocessor is available to store exclusively read-only data.

Table 4.1 shows how the key variables in the MCML code are mapped onto different types of GPU memories for high performance. Note that there is also a region in device memory called *local memory* reserved for large data structures, including arrays, which cannot be mapped into registers by the compiler. *Local* memory is somewhat a misnomer for hardware acceleration since it is as slow as global memory. For the MCML code, arrays are used extensively in the original random number generator, which makes it inefficient

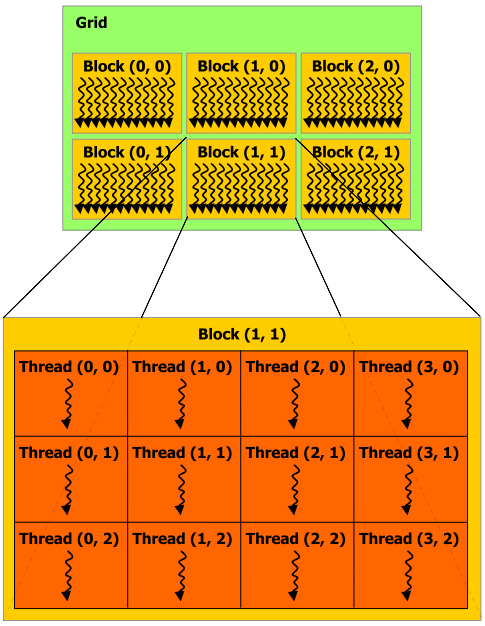


Figure 4.2: CUDA programming model - concept of a thread block consisting of multiple threads, each responsible for a subtask as defined by the programmer [85]

on GPUs since arrays are mapped to the slow, local memory. Therefore, a different random number generator, explained in Section 4.5.3, is employed for the GPU version.

4.3.2 Programming with CUDA

CUDA is a C-based programming language extension that has gained acceptance in the scientific computing community in recent years. By abstracting away some of the complexity in programming graphics hardware, CUDA reduces the learning curve and has made the GPU more accessible to a wider audience for general-purpose computing. Unfortunately, it is not trivial to optimize for high computational speed using CUDA, as this programming paradigm still exposes a significant portion of the underlying GPU architecture for the programmer to handle.

CUDA Programming Model

In CUDA, the programmer writes GPU code in the form of *kernels*, which are similar to regular C functions. Multiple copies are executed in parallel by the GPU *threads*. Therefore, the programmer must first divide the application into parallel units of execution, which are assigned to threads. (For the MCML code, each thread can be assigned a group of photon packets, which can be processed in parallel with other threads.) These threads are in turn organized into *thread blocks*, as shown in Fig. 4.2. Next, the programmer specifies a *kernel configuration* - the number of thread blocks and number of threads within each thread block. **This is an important decision as each thread block is executed on a single multiprocessor and the threads inside are mapped onto the individual SPs. Note that this is not a one-to-one mapping.** For high performance, NVIDIA suggests launching multiple thread blocks per multiprocessor and hundreds of threads within each thread block to fully occupy the GPU resources. For example, one might choose 60 thread blocks for 30 multiprocessors to allow 2 thread blocks to interleave execution on the same multiprocessor (handled by the thread scheduler). Within each thread block, 256 threads may be launched. (Using this configuration with 15360 threads in total and assuming 1 million photon packets need to be simulated, each thread would process ∼65 photon packets.) The choice of a proper kernel configuration involves more complicated trade-offs and is limited by the shared memory usage, register usage, and other thread scheduling considerations. Interested readers can refer to Chapter 5 of the CUDA programming guide for details [85].

Memory Access Restrictions

CUDA also requires the programmer to explicitly manage the storage of data on the GPU and the data transfer between the device memory on the GPU side and the host memory on the CPU side. The programmer must be aware of the size and access restrictions of each type of memory in order to properly write and launch a GPU kernel, as explained in

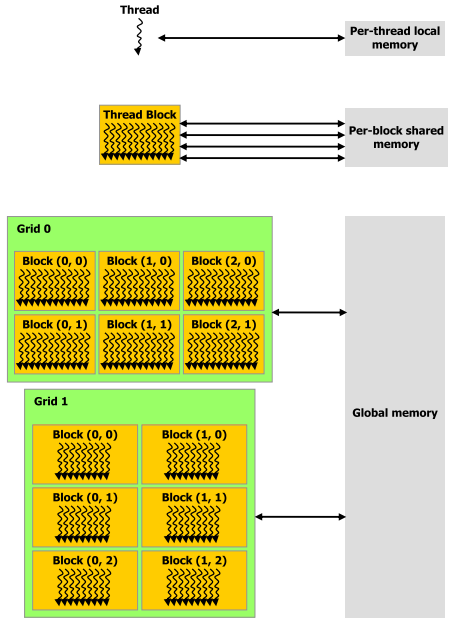


Figure 4.3: Memory access restrictions for CUDA programming - (1) local memory and registers are private to each thread, (2) shared memory is shared by threads within a thread block, and (3) global memory can be accessed by all threads from all thread blocks. [85]

Fig. 4.3. The scope of a variable differs depending on the specific type of memory used. For example, the global memory and the read-only constant memory can be accessed by all threads, while shared memory is only shared by threads within a thread block. By default, variables declared without specifying the type of memory are stored in registers, which are private to each thread and cannot be accessed by other threads. Notice that these restrictions limit how different variables in the MCML code are mapped onto the GPU memories, as shown earlier in Table 4.1. For example, if the photon data structure were to be shared across threads or if its size were too large, it would not have been possible to place it inside the fast registers.

Atomic Instructions

CUDA also provides ways to synchronize the execution of threads as many applications require some form of cooperation or *synchronization* between the subtasks assigned to

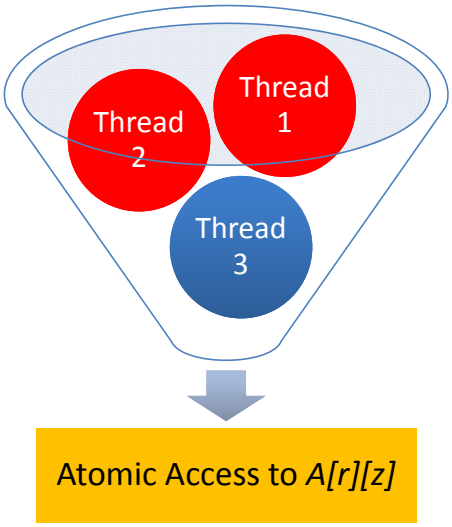


Figure 4.4: Concept of an atomic access represented by a funnel: as thread 3 is accessing the absorption array, threads 1 and 2 must wait. Atomic instructions can cause bottlenecks in a computation, especially with thousands of threads common in GPU programming.

each thread, as defined by the programmer. To synchronize execution, multiple threads may communicate through some shared variables. A common synchronization technique involves the use of *atomic instructions* to coordinate the sequential access to a shared variable, which can be stored in global or shared memory for NVIDIA GPUs. Atomic instructions guarantee data consistency by allowing only one thread to update the shared variable at any time; however, in doing so, it stalls other threads if they require access to the same variable. The concept of an atomic access is illustrated by the funnel in Fig. 4.4, where 3 threads are simultaneously attempting to access the shared $A[r][z]$ array to accumulate its absorbed weight, which is a key step in the fluence update part of the MCML algorithm.

Just as an atom in chemistry is considered the indivisible unit or the fundamental building block of all matter, an atomic instruction cannot be further divided. For example, an atomic instruction may read a piece of data, modify it, and then write the modified value back to memory all in one indivisible operation, which may be guaranteed

by locking the variable to avoid interruptions temporarily. Since this locking mechanism forces all other threads that might be simultaneously desiring to modify that memory location to wait, atomic instructions can give rise to performance bottlenecks. Although atomic instructions can sometimes be avoided by replicating the shared data structure in each thread, if the shared data structure is large (as is the case for the absorption array), the size of the memory will in turn limit the number of threads that can be launched. This may decrease the parallelism and the final performance of the application despite having avoided the use of atomic instructions. The problem of optimizing the use of atomic instructions for the GPU-based MCML program is further discussed in Section 4.4.2.

4.3.3 CUDA-specific Acceleration Techniques

For the highest performance, programmers must write CUDA code with the GPU architecture in mind. Two important objectives include maximizing parallelism and instruction throughput, both of which are also impacted by efficient memory usage.

Maximizing Parallelism and Instruction Throughput

There are a number of techniques to improve the efficiency of code execution on the NVIDIA GPU by increasing the level of parallelism and instruction throughput, several of which are listed below:

1. **Devise a parallelization scheme with minimum synchronization** by carefully identifying parallelizable portions in the CPU code.

2. **Minimize code divergence**, or threads taking different execution paths, by eliminating or grouping conditional statements, such as `if and else` constructs or common `for, while,` and `do while` loops.

3. **Use single-precision floating point operations** whenever possible as double-precision operations and integer operations are slower on current GPUs.

4. **Use the faster GPU-intrinsic math functions**, such as `__logf(x)` for faster logarithmic computations, by enabling the `-use_fast_math` compiler flag.

Efficient Memory Usage

Efficient memory usage is critical as the amount of time required for retrieving data from different types of memories can differ significantly, as explained in Section 4.3.1. Memory operations can also be significantly more time-consuming than arithmetic operations. For example, global memory access can take hundreds of clock cycles when only a few cycles are required for single-precision floating point multiplication. Therefore, global memory operations should be reduced as much as possible. There are mainly three ways to achieve this goal:

1. **Store or cache frequently accessed data in fast on-chip memories**, such as registers, shared memory, and the constant memory. (For the GPU-based MCML code, the photon data structure is stored in registers, part of the absorption array is cached in shared memory, and the optical properties of the tissue layers are stored in the constant memory.)

2. **Eliminate arrays to avoid the use of local memory**, which is as slow as global memory. (In the original MCML code, the random number generation algorithm depends heavily on large arrays, which make it inefficient for GPU implementation.)

3. **Increase the number of threads**, which gives the thread scheduler more opportunities to fully utilize the GPU resources and to hide global memory access latency (e.g., by executing computations from another thread while a thread is waiting for data from the global memory).

Balancing Trade-offs

Unfortunately, the optimizations above often compete with one another for GPU resources. For example, caching data in registers increases register usage per thread, which in turn limits the number of threads that can be launched. If maximum parallelism (or 512 threads per thread block) is desired, each thread can at most use 32 registers (since 32 x 512 = 16,384 registers, which is the maximum number of registers available). To complicate matters further, the number of registers required by a program is not easily predicted as the number of variables does not necessarily correspond to the number of registers needed due to compiler optimizations.

One of the key challenges in optimizing a CUDA program is to find a scheme of assigning resources that achieves the best performance. The strategies for accelerating the MCML code on CUDA-enabled GPUs are discussed in the next section.

4.4 GPU-accelerated MCML Code

In this section, the implementation details of the GPU-accelerated MCML program are presented, showing how a high level of parallelism is achieved, while avoiding memory bottlenecks. The development process is described to summarize the thought process and challenges encountered before arriving at the final solution. This may assist other investigators in related efforts since the MC method is widely applied in computational biophysics and most MC simulations share a set of common features. The final, optimized implementation was also tested on a multi-GPU system to show the possibility of using a cluster of GPUs for PDT treatment planning.

4.4.1 Parallelization Scheme

One key difference between writing CUDA code and writing a traditional C program (for sequential execution on a CPU) is the need to devise an efficient parallelization

scheme for the case of CUDA programming. Although the syntax used by CUDA is in theory very similar to C, the programming approach differs significantly. Fig. 4.5 shows an overview of the parallelization scheme used to accelerate the MCML code on the NVIDIA GPU. Compared to serial execution on a single CPU where only one photon packet is simulated at a time, the GPU-accelerated version can simulate many photon packets in parallel using multiple threads executed across many scalar processors. Note that the total number of photon packets to be simulated are split equally among all created threads.

The GPU program or kernel contains the computationally intensive part or the key loop in the MCML simulation (represented by the position update, direction update, and fluence update loop in the figure). Other miscellaneous tasks, such as reading the simulation input file, are performed on the host CPU. Each thread executes a similar sequence of instructions, except for different photon packets simulated based on a different random number sequence. To ensure that a different sequence is generated within each thread, the unique thread ID (obtained using the CUDA keywords `threadIdx.x` and `blockIdx.x`) is used to create a unique set of random seeds for each thread.

In the current implementation, the kernel configuration is specified as 30 thread blocks (Q=30), each containing 256 threads (P=256). As shown in Fig. 4.5, each thread block is physically mapped onto one of the 30 multiprocessors and the 256 threads interleave its execution on the 8 scalar processors within each multiprocessor. As discussed in Section 4.3.3, increasing the number of threads helps to hide the memory latency. However, this also increases competition for atomic access to the common $A[r][z]$ array. Therefore, the maximum number of threads, which is 512 threads per thread block on the graphics cards used in this work, was not chosen and only 256 threads were launched per thread block. A lower number would not be desirable since more than 192 threads are required to avoid delays in accessing a register (due to potential register read-after-write dependencies and register memory bank conflicts [85]). A similar reasoning applies

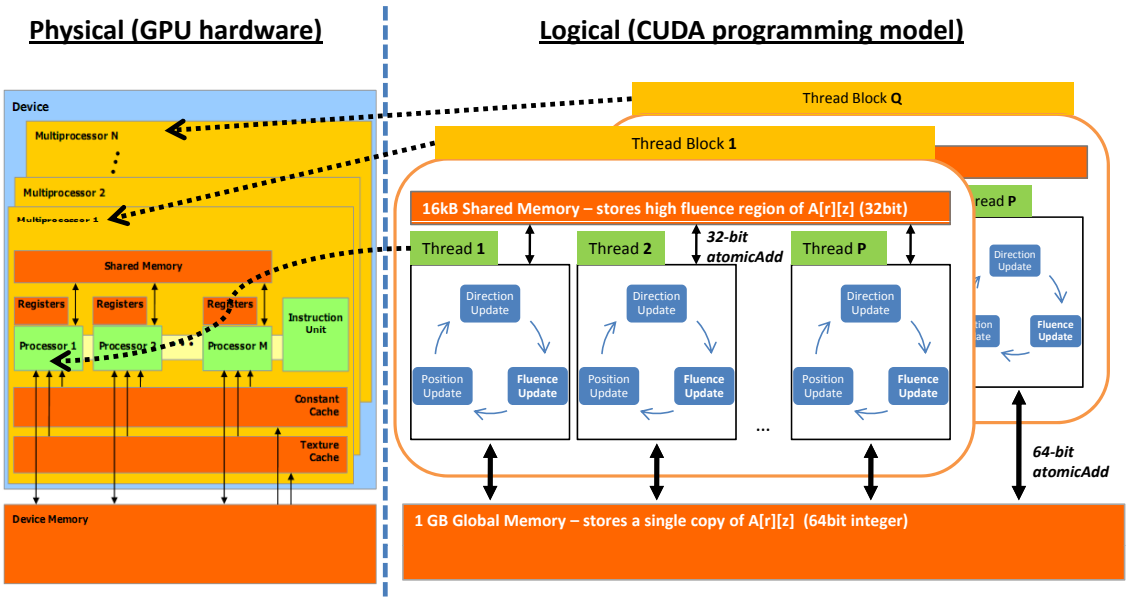


Figure 4.5: Parallelization scheme of the GPU-accelerated MCML code (Q=30 and P=256 for each GPU). Note the mapping of the threads to the GPU hardware. In general, P should be a multiple of M, while Q should be a multiple of N for high performance.

to the number of thread blocks chosen. A lower number than 30 thread blocks would under-utilize the GPU computing resources since there are 30 multiprocessors available. A larger number, such as 60 thread blocks, would decrease the amount of shared memory available for caching and also increase competition for access to the $A[r][z]$ array. The need to alleviate the competition for atomic access is discussed in detail next.

4.4.2 Key Performance Bottleneck

To understand further why atomic accesses to the $A[r][z]$ array could become a key performance bottleneck, notice that all threads add to the same absorption array in the global memory during the fluence update step. In CUDA, atomic addition is performed using the `atomicAdd()` instruction. However, using `atomicAdd()` instructions to access the global memory is particularly slow, both because global memory access is a few orders of magnitude slower than that of on-chip memories and because atomicity prevents

parallel execution of the code (by stalling other threads in the code segment where atomic instructions are located). This worsens with increasing number of threads due to the higher probability for simultaneous access to an element, also known as contention.

Note that although the $A[r][z]$ array could, in theory, be replicated per thread to completely avoid atomic instructions, this approach is limited by the size of the device memory, as discussed in Section 4.5.3, and would not be feasible in the general 3-D case with much larger absorption arrays. Therefore, a more general approach was explored to solve this performance problem.

4.4.3 Solution to Performance Issue

To reduce contention and access time to the $A[r][z]$ array, two memory optimizations, caching in registers and shared memory, were applied.

The first optimization is based on the idea of storing the recent write history, representing past absorption events, in temporary registers to reduce the number of atomic accesses to the global memory. It was observed that consecutive absorption events can happen at nearby, or sometimes the same, locations in the $A[r][z]$ array, depending on the absorption grid geometry and optical properties of the layers. Since the number of registers is limited, in the final solution, only the most recent write history is stored in 2 registers – one for the last memory location and one for the total accumulated weight. In each thread, consecutive writes to the same location of the $A[r][z]$ array are accumulated in these registers until a different memory location is computed. Once a different location is detected, the total accumulated weight in the temporary register is flushed into the global memory using an `atomicAdd()` operation and the whole process is repeated.

The second optimization, illustrated in Fig. 4.5, is based on the high access rate of the $A[r][z]$ elements near the photon source (or at the origin in the MCML model), causing significant contention when atomic instructions are used. Therefore, this region of the $A[r][z]$ array is cached in the shared memory. This optimization has two significant

implications. First, contention in the most competitive region of the $A[r][z]$ array is reduced by up to 30-fold since the shared memory copy of the array is updated atomically by only 256 threads within each thread block instead of 7680 threads across 30 blocks. Second, accesses to the shared memory are ∼100-fold faster than those to the global memory. Together, these two factors explain the significant improvement in performance observed after this optimization, as shown in Section 4.5.3.

To store as many elements near the photon source as possible in the shared memory, the size of each element in the $A[r][z]$ array was reduced to 32 bits (as opposed to 64 bits for the master copy in the global memory). Given the size of the shared memory is 16 kB, 3584 32-bit elements can be cached compared to only 1792 elements if 64-bit elements were used (3584 x 32 bits or 4 bytes = 14 kB, with the remaining shared memory space reserved by CUDA). However, this reduction also causes a far greater risk of computational overflow, which occurs when the accumulated value exceeds $\sim 2^{32}$ (instead of $\sim 2^{64}$ in the 64-bit case). To prevent overflow, the old value is always checked before adding. If overflow is imminent, the value is flushed to the absorption array in global memory, which still uses a 64-bit integer representation and overflow in this 64-bit array is not necessary to detect given it takes ∼ 1500 billion photon packets all dropping their weights (initially set to 12 million) into one single voxel to cause overflow in a 64-bit integer.

As an additional optimization to avoid atomic accesses, in the GPU version, photon packets at locations beyond the coverage of the absorption grid (as specified through the input parameters dr, dz, nr, and nz) no longer accumulate their weights at the perimeter of the grid, unlike in the original MCML code. Note that these boundary elements were known to give invalid values in the original MCML code [40]. This optimization does not change the correctness of the simulation, yet it ensures that performance is not degraded if the size of the detection grid is decreased, which forces photon packets to be absorbed at boundary elements (significantly increasing contention and access latency to these

elements in the $A[r][z]$ array).

4.4.4 Other Key Optimizations

Another major problem with the original MCML code for GPU-based implementation is its abundance of branches (e.g., `if` statements), leading to significant code divergence. To illustrate how the issue of divergence was tackled, the implementation of the `Reflect` function inside the direction update step is described as an example.

Looking inside the original `Reflect` function shown in Listing 4.1 [40], there are two nearly identical branches of execution (labelled *Branch 1* and *Branch 2*) with only slightly different assignments of variables. The condition used to determine which branch to take is the sign of `Photon_Ptr->uz` or μ_z, which is the direction cosine in the z direction. There is also a significant amount of computation in each branch, meaning that if this code were directly implemented on the GPU, the thread taking a different execution path would significantly slow down the other threads. In particular, the `RFresnel` function for computing the internal reflectance (defined in Eq. 2.7) contains another layer of conditional statements and expensive trigonometric calculations, which are not shown here for clarity.

In the CUDA implementation as shown in Listing 4.2, the `Reflect` code was significantly re-structured to remove or to reduce the size of a large number of branches. For example, the branch in the outer-most layer was replaced with a much smaller one that decides which variables to use by collecting all the information that depends on the value of μ_z, followed by the common piece of computation. This optimization almost halved the number of instructions executed in this step and reduced divergence significantly. The correctness of these transformations is shown by the validation results presented in Section 4.6.

```
1 void Reflect(InputStruct *In_Ptr, PhotonStruct *Photon_Ptr, OutStruct *Out_Ptr)
```

```
2  {
3    ////////////////////////////////////////////////////////////////////////
4    // Branch 1: If the photon packet is heading UPwards (-z direction)
5    ////////////////////////////////////////////////////////////////////////
6    if(Photon_Ptr->uz < 0.0) {
7      double uz = Photon_Ptr->uz;    /* z directional cosine. */
8      double uz1;                    /* temporary variable. */
9      double r=0.0;                  /* reflectance */
10     short  layer = Photon_Ptr->layer;
11     double ni = In_Ptr->layerspecs[layer].n;
12     double nt = In_Ptr->layerspecs[layer-1].n;
13     /* Get r. */
14     if( - uz <= In_Ptr->layerspecs[layer].cos_crit0)
15       r=1.0;                       /* total internal reflection. */
16     // Compute Internal Reflectance using RFresnel function
17     else
18       r = RFresnel(ni, nt, -uz, &uz1);
19
20     if(RandomNum() > r) {          /* transmitted to layer-1. */
21       if(layer==1) {
22         Photon_Ptr->uz = -uz1;
23         Photon_Ptr->dead = 1;
24       }
25       else {
26         Photon_Ptr->layer--;
27         Photon_Ptr->ux *= ni/nt;
28         Photon_Ptr->uy *= ni/nt;
29         Photon_Ptr->uz = -uz1;
30       }
31     }
32     else                           /* reflected. */
33       Photon_Ptr->uz = -uz;
34   }
35   ////////////////////////////////////////////////////////////////////////
36   // Branch 2: If the photon packet is heading DOWNwards (+z direction)
37   ////////////////////////////////////////////////////////////////////////
38   else {
39     // Exactly the same variables with slightly different assignments
40     double uz = Photon_Ptr->uz;
41     double uz1;
42     double r=0.0;
```

```
43      short  layer = Photon_Ptr->layer;
44      double ni = In_Ptr->layerspecs[layer].n;
45      double nt = In_Ptr->layerspecs[layer+1].n;      //Note that [layer+1] is used
46      /* Get r. */
47      if( uz <= In_Ptr->layerspecs[layer].cos_crit1)  //Note that cos_crit1 is used
48        r=1.0;
49      else
50        r = RFresnel(ni, nt, uz, &uz1);               //Note the sign of uz
51
52      if(RandomNum() > r) {                           // transmitted to layer+1.
53        if(layer == In_Ptr->num_layers) {
54          Photon_Ptr->uz = uz1;
55          Photon_Ptr->dead = 1;
56        }
57        else {
58          Photon_Ptr->layer++;                        //Note the change in layer
59          Photon_Ptr->ux *= ni/nt;
60          Photon_Ptr->uy *= ni/nt;
61          Photon_Ptr->uz = uz1;                       //Note the sign of uz1
62        }
63      }
64      else
65        Photon_Ptr->uz = -uz;
66    }
67  }
```

Listing 4.1: MCML C code for the Reflect() function [40]

```
1   __device__ void Reflect(float rand3, float *ux, float *uy,
2     float *uz, unsigned int *layer, unsigned int* dead) {
3       /* Collect everything that depends on the sign of uz. */
4       float cos_crit; // cosine of the critical angle
5       int new_photon_layer;
6       if (*uz > 0.0F) {
7           cos_crit = d_layerspecs[(*layer)].cos_crit1;
8           new_photon_layer = (*layer)+1;
9       } else {
10          cos_crit = d_layerspecs[(*layer)].cos_crit0;
11          new_photon_layer = (*layer)-1;
12      }
13      float ca1 = fabsf(*uz); // cosine of the incident angle
14      *uz = -(*uz); // Default move = reflect
```

```
15      // General Case - No total internal reflection
16      // if incident angle < the critical angle,
17      // or if (cosine of incident angle (ca1) > cosine of critical angle (cos_crit))
18      if (ca1 > cos_crit)
19      {
20          // refractive indices of incident and transmitted media
21          float ni = d_layerspecs[(*layer)].n;
22          float nt = d_layerspecs[new_photon_layer].n;
23          float ni_nt = __fdividef(ni, nt);    // reused later
24          /*Compute using RFresnel() - not shown. */
25          if (rand3 > rFresnel)
26          {
27              *layer = new_photon_layer; // Transmit into new layer
28              *dead = (*layer == 0 || *layer > d_In_Ptr.num_layers);
29              *ux *= ni_nt; //update direction
30              *uy *= ni_nt;
31              *uz = -copysignf(uz1, *uz);
32          }
33      }
34  }
```

Listing 4.2: CUDA code for the Reflect() function

Finally, this implementation also includes a number of other optimizations, such as using GPU-intrinsic math functions (namely __sincosf(x) and __logf(x)), reducing local memory usage by expanding arrays into individual elements, and storing read-only tissue layer specifications in constant memory. Further details on the effect of these optimizations and other optimizations not mentioned here are discussed in Section 4.5.3. For details on the CUDA source code, please refer to Appendix B.

4.4.5 Scaling to Multiple GPUs

To scale the single-GPU implementation to multiple GPUs, multiple host threads were created on the CPU side to simultaneously launch multiple kernels, to coordinate data transfer to and from each GPU, and to sum up the partial results generated by the GPUs for final output. The same kernel and associated kernel configuration were replicated N

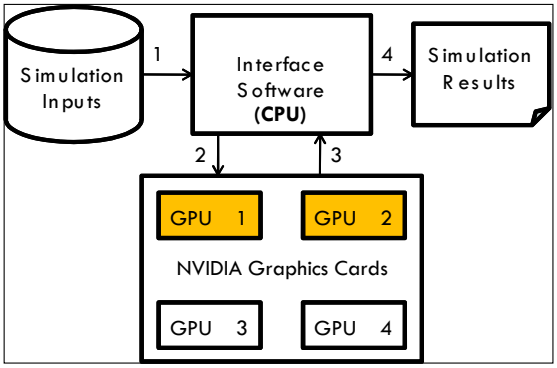


Figure 4.6: Multi-GPU system (GTX 295 contains GPU 1 and GPU 2, highlighted to indicate they belong to the same graphics card, while the two GTX 280 cards contain GPU 3 and GPU 4): step 1, parsing of the simulation input file; step 2, transfer of initialization information to the GPUs; step 3, transfer of simulation results from the GPUs; step 4, creation of the simulation output file. Inside each GPU, the same parallelization scheme from Fig. 4.5 is used.

times where N is the number of GPUs, except that each GPU initializes a different set of seeds for the random number generator and declares a separate absorption array. This allows the independent simulation of photon packets on multiple GPUs, similar to the approach taken in CPU-based cluster computing.

4.5 Performance

4.5.1 GPU and CPU Platforms

The execution time of the GPU-accelerated MCML program (named here GPU-MCML) was first measured on a single GPU — the NVIDIA GTX 280 graphics card — with 30 multiprocessors. The code was migrated to a Quad-GPU system consisting of two NVIDIA GTX 280 graphics cards and a NVIDIA GTX 295 graphics card with 2 GPUs, as shown in Fig. 4.6. This Quad-GPU system contains a total of 120 multiprocessors. The final GPU-MCML was compiled using the CUDA Toolkit 2.2 and was tested in both

a Linux and Windows environment. The number of GPUs used can be varied at run-time and the simulation is split equally among the specified number of GPUs.

For baseline performance comparison, the same Intel Xeon processor (or Platform 2 in Table 3.5) was selected due to its high performance. Note that this processor was manufactured using the 65 nm process technology (defined in Section 3.6), which is similar to that of the GTX 280 graphics card. The original, CPU-based MCML program (named here CPU-MCML) was compiled with the highest optimization level (gcc -O3 flag) and its execution time was measured on one of the two available cores on the Xeon processor, as in Chapter 3. All execution times included the main simulation and all pre-/post-processing operations.

4.5.2 Speedup

For performance comparison, the same five-layer skin model (at λ=633 nm) from Table 3.3 was used. Table 4.2 shows the execution time of GPU-MCML as the number of GPUs and the associated number of scalar processors was increased. The kernel configuration was fixed at 30 thread blocks, each with 256 threads. The performance of the solution was roughly proportional to the number of GPUs used. Using all 4 GPUs or equivalently 960 scalar processors, the simulation time for 100 million photon packets in the skin model was reduced from approximately 1.7 h on an Intel Xeon processor to 5.8 s on 4 GPUs. This represents an overall speedup of 1052x !

4.5.3 Effect of Optimizations

The impressive performance was the result of a series of optimizations, some of which have been described earlier in Section 4.4. This section presents the effect of these optimizations as well as several other ones, not mentioned previously, with significant implications.

Table 4.2: Speedup as a function of the number of GPUs for simulating 10^8 photon packets in the skin model

Platform (Configuration)	Total Number of Scalar Processors	Simulation Time (s)	Speedup	Normalized Speedup
Intel 3-GHz Xeon 5160 processor	–	6102	1	0.0037
GTX 295 (using 1 GPU)	240	22.7	269	1.0
GTX 295 (2 GPUs)	480	12.5	488	1.8
GTX 295 + GTX 280 (3 GPUs)	720	7.7	792	2.9
GTX 295 + 2 x GTX 280 (4 GPUs)	960	5.8	1052	3.9

The Unoptimized MCML Kernel

The initial MCML kernel, which did not employ any optimizations, yielded a very low performance – 4x on the NVIDIA 8800 GTX graphics card with 128 scalar processors or equivalently 16 multiprocessors. Because the 8800 GTX graphics card does not support atomic operations, this initial version employed a very basic parallelization scheme, in which a private copy of the $A[r][z]$ array was created for each thread in the global memory. The final result was obtained by summing the partial results stored in the private copies. At this point, the kernel configuration has not yet been fine-tuned and was set to 32 thread blocks x 32 threads/block. (The effect of changing this configuration will be discussed later.) However, the speedup was far from satisfactory, given that a significant amount of time has been invested in converting the original C code to CUDA code.

Reducing Local Memory Usage

To investigate the performance issue further, the CUDA compiler intermediate files (with the `.cubin` file extension) were inspected. These files explained the usage statistics of registers, shared memory, and local memory for the initial MCML kernel. According to the usage statistics, each thread required 292 bytes of local memory, mostly for storing

Table 4.3: Effect of local memory usage on the simulation time for 10^8 photon packets in the skin model

Optimization	Local Memory Usage (byte)	Simulation Time[a] (s)	Speedup
1. Original, unoptimized version	292	1460	4
2. Use the Tausworthe random number generator [79]	68	759	8
3. Expand arrays into individual elements	28	578	11
4. Double the number of threads	28	338	18

[a] Measured on the NVIDIA 8800GTX graphics card

the arrays used by the random number generator. Since local memory is as slow as global memory, a series of optimizations were applied to reduce local memory usage and their effects on performance are shown in Table 4.3. First, a more efficient random number generator called the three-component Tausworthe generator [79] (with a period length of $\approx 2^{88}$) was adopted as it does not use arrays. This optimization led to a two-fold speedup compared to the previous version. Second, another array used to store nine random numbers was expanded into individual elements, allowing the compiler to allocate them in registers. This optimization resulted in another 1.3x speedup.

The last optimization involved fine-tuning the kernel configuration and doubling the number of threads to 2048 (16 thread blocks x 128 threads/thread block), which resulted in an additional 1.7x speedup. The determination of the optimum kernel configuration was based on the performance guidelines suggested in the CUDA programming guide [85] as well as empirical testing. After a whole series of optimizations, the total speedup was only 18x on the 8800 GTX card.

Table 4.4: Effect of optimizations on the simulation time for 10^8 photon packets in the skin model

Optimization	Simulation Time[a] (s)	Speedup
1. Baseline[b] (30 blocks, 128 threads/block)	466	13
2. Use atomic instructions (30 blocks, 256 threads/block)	158	39
3. Use fast intrinsic math functions	157.9	39
4. Cache recent fluence updates in registers	112	54
5. Cache high fluence regions of the $A[r][z]$ array in shared memory	46	133
6. Reduce divergence, instruction count, and optimize overflow handler	22	277

[a] Measured on the NVIDIA GTX 280 graphics card

[b] This is the version after optimization 4 from Table 4.3.

Key Limitation of the First MCML Kernel

Although the initial approach of using a private copy of the $A[r][z]$ array per thread allowed the parallel processing of photon packets without synchronization, this approach suffered from a fundamental problem. The size of the device memory, which is 768MB in 8800 GTX, limited the total number of threads that could be launched, which in turn limited the amount of parallelism and hence speedup achieved.

Migrating to New GPU with 64-bit Atomic Support

The migration to the new NVIDIA GTX 280 graphics card is a key milestone since this card supports 64-bit atomic instructions, unlike the old 8800 GTX. Atomic operations allow multiple threads to share access to a common data structure. The size of the device memory is no longer the determining factor in the number of threads that can be launched.

Before making any changes, a baseline measurement was made using the same code on the new card. A rather disappointing speedup of ∼13x was obtained, as shown in

Table 4.4. Next, a different parallelization scheme was attempted: a single copy of the $A[r][z]$ array was created in the global memory to be atomically accessed by all threads using the 64-bit integer `atomicAdd()` operation. A total of 7680 threads were launched using a kernel configuration of 30 thread blocks x 256 threads/block. The device memory size no longer restricted the parallelism since only one copy of the absorption array is required for any number of threads that may be launched. The increase in the number of threads led to a 3-fold performance improvement.

Unfortunately, the next optimization, involving the use of the GPU intrinsic math functions where possible (such as integer division, trigonometric, and logarithmic functions), showed only marginal improvement (< 1 % change in execution time). This was mainly due to the presence of another performance bottleneck, which highlights a major challenge in optimizing CUDA code - the identification of the main performance bottleneck.

Minimizing Atomic Accesses to Global Memory

As discussed in Section 4.4, one particularly crucial bottleneck is the use of atomic instructions to access global memory. To solve this problem, the first approach involved caching the most recent fluence update history in registers (as described in Section 4.4.3), which increased the speedup to ∼54x. This scheme was then expanded to cache writes to multiple locations of the $A[r][z]$ array (in each thread) by using the shared memory, which further improved the performance to ∼90x. The final approach proposed in this thesis, which involved caching the high fluence regions of the $A[r][z]$ array in shared memory, led to significantly better performance (∼133x) due to the great reduction in the number of expensive, atomic accesses to global memory. To maximize the number of high-fluence voxels that can be stored in the small shared memory, the previous scheme of caching recent writes to multiple locations of the $A[r][z]$ array was not implemented simultaneously.

Table 4.5: Effect of grid geometry on simulation time for 10^7 photon packets in a thick homogeneous slab

dr and dz (cm)	nr	nz	Simulation Time in GPU-MCML[a] (s)	Simulation Time in CPU-MCML[b] (s)	Speedup
0.001	500	200	14.5	5658	390
0.01	500	200	15.2	5670	373
1	500	200	14.4	5460	379
1	5	2	14.3	5473	383
0.001	5000	2000	35.5	6546	184

[a] Measured on the NVIDIA GTX 280 graphics card

[b] Measured on the Intel Xeon 5160 processor

Reducing Code Divergence and Instruction Count

Once the atomic accesses were optimized, the reduction of divergence and instruction count throughout the code as well as the optimization of the overflow handler led to an additional ~2x performance. Specifically, the number of branches in the `Reflect` function was greatly reduced as discussed in Section 4.4.4. The number of instructions was reduced by cleaning up the code throughout. Finally, the overflow handler was optimized so that once overflow is detected in a single element in the shared memory, a group of elements are flushed to global memory to avoid frequent (or element-by-element) writes to global memory. Please see Appendix B for details.

4.5.4 Effect of Grid Geometry

To test the effect of grid resolution and the number of voxels on execution time, a thick, homogeneous slab (μ_a=0.1 cm^{-1}, μ_s=90 cm^{-1}, n=1.4, g=0.9, thickness=100 cm) was used. These optical properties are based on the test case used by Alerstam et al. [86] in their validation of CUDAMCML.

As shown by the first three rows in Table 4.5, the grid resolution (dr and dz) only changed the simulation time of GPU-MCML slightly. The difference was within one second. Interestingly, in the homogeneous slab, the simulation is even faster on the GPU, compared to the speedup obtained for the skin geometry from Table 4.2. This is likely due to the more prominent effect of divergence in thread execution when interfaces are present. The most significant difference in simulation time was observed when the number of voxels in the radial direction (nr) and in the z direction (nz) was increased from 500 $\times$ 200 (or 10^5 voxels) to 5000 $\times$ 2000 (or 10^7 voxels). The latter configuration increased the simulation time of both CPU-MCML and GPU-MCML. Further investigation revealed that $\sim$15 s was required to generate the simulation output file for 10^7 voxels, while less than 1 second was required for the 10^5 voxels configuration. The main reason was that the size of the output file also increased from 1.25 MB to 125 MB. This overhead was very significant given the relatively short GPU kernel execution time; neglecting this overhead, the speedup would have been $\sim$320x. However, the performance still degraded slightly compared to another configuration (dr=dz=0.01, nr=500, and nz=200) with the same grid coverage (namely a cylindrical grid with a radius of 5 cm and depth of 2 cm). This is likely due to the decreased proportion of voxels that can be stored in the shared memory, leading to more `atomicAdd()` operations for writing to global memory.

4.6 Validation

The validation procedure for GPU-MCML is similar to that presented in Section 3.5.1.

4.6.1 Test Cases

Three test cases were used to validate GPU-MCML, including the skin model from Table 3.3 (dr=0.01 cm, dz=0.002 cm, nr=256, and nz=256), the homogeneous slab used in Section 4.5.4 (dr=dz=0.01 cm, nr=500, and nz=200), and a ten-layered geom-

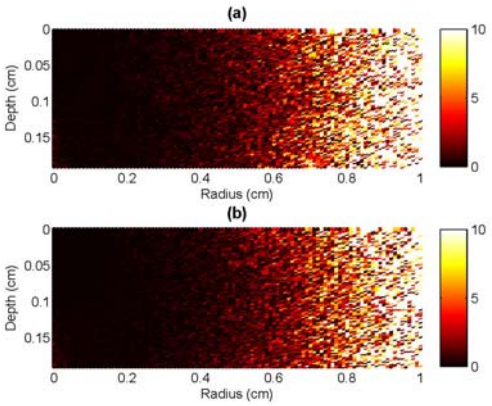


Figure 4.7: Distribution of relative error for the skin model using 100 million photon packets: (a) GPU-MCML vs. CPU-MCML, (b) CPU-MCML vs. CPU-MCML. All versions adopted the Tausworthe random number generator [79]. Colour bar represents percent error from 0% to 10%.

etry alternating between 2 materials, each layer with a thickness of 0.1 cm (material 1: μ_a=0.1 cm^{-1}, μ_s=90 cm^{-1}, g=0.9, and n=1.5; material 2: μ_a=0.2 cm^{-1}, μ_s=50 cm^{-1}, g=0.5, and n=1.2; dr=dz=0.01 cm, nr=500, and nz=200). The last two test cases were adopted from Alerstam et al. [86] in their validation of CUDAMCML.

4.6.2 Error Distribution

Figure 4.7 shows two very similar distributions, verifying that the difference observed between the $A[r][z]$ arrays from GPU-MCML and CPU-MCML was within the statistical uncertainty between two runs of CPU-MCML for the case of the skin model.

Figure 4.8 shows the distribution of error for the other two test cases - the homogenous slab and ten-layered geometry. The patterns appear different, as expected, due to the different input parameters used. The reference maps identified these differences as the statistical uncertainty between runs, rather than the errors added by the GPU-MCML implementation. In particular, the conversion to single-precision floating point arithmetic, as shown by these plots, resulted in a negligible increase in error.

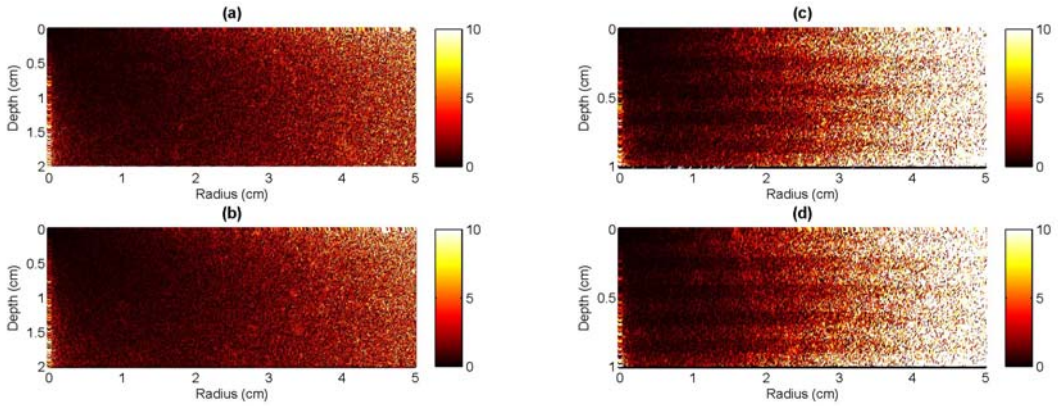


Figure 4.8: Distribution of relative error for a homogeneous slab [left panels (a) and (b)] and ten-layered geometry [right panels (c) and (d)] using 10 million photon packets: (a, c) GPU-MCML vs. CPU-MCML, (b, d) CPU-MCML vs. CPU-MCML. All versions adopted the Tausworthe random number generator [79]. Colour bar represents percent error from 0% to 10%.

4.6.3 Light Dose Contours

GPU-MCML vs. CPU-MCML

To show the accuracy of GPU-MCML within the context of PDT treatment planning, the skin model was used as the simulation model. The absorption probability density was first converted to fluence for the impulse response and the isofluence contour lines generated by CPU-MCML and GPU-MCML were compared. Note that finite-sized beams are modelled in the next section to separate any error potentially introduced by the convolution operation [53].

Figure 4.9 shows that the isofluence lines produced by GPU-MCML and CPU-MCML matched very well. A minor shift in the position of the isofluence lines was only noticeable for very low fluence levels. Notice that the isofluence line located at a radius of ~ 0.7 cm corresponds to the transition region in Fig. 4.7 where the statistical uncertainty between runs starts to increase appreciably due to the low photon count. If this isofluence line is of importance (i.e., if it is near the threshold fluence level for activating the photosensitizers), more photon packets can be launched to achieve the desired accuracy for PDT treatment

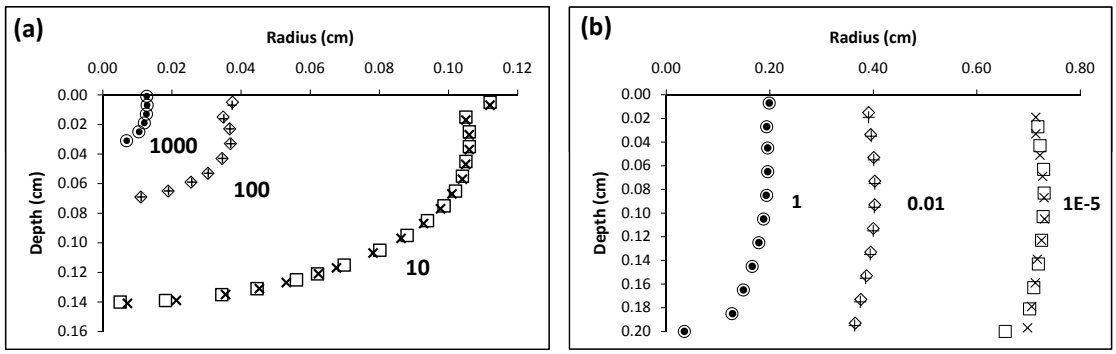


Figure 4.9: Isofluence lines for the impulse response generated by GPU-MCML and CPU-MCML using the skin model (10^8 photon packets): ○, ◊, and □, results from CPU-MCML; and •, +, and ×, results from GPU-MCML. (a) Isofluence lines for fluence levels at 1000, 100, and 10 cm^{-2}, as indicated on the figure and (b) isofluence lines for fluence levels at 1, 0.01, and 0.00001 cm^{-2}.

planning.

Considerations for Treatment Planning

To employ MC-based light dosimetry in PDT treatment planning, a number of additional factors must be considered. A key consideration in treatment planning is the number of photon packets required as this directly affects the geometric uncertainty of the simulated isofluence contours. This is particularly important as the zone of necrosis is affected by the location of the threshold fluence level according to the threshold light dose model presented in Chapter 1. The number of photon packets also has a direct impact on the overall treatment planning time. Therefore, this section considers a hypothetical scenario to illustrate the impact of the number of photon packets on both accuracy and time.

Suppose a 1 cm diameter flat, circular light beam at 633 nm is used for surface illumination on a skin tumour in PDT. Further suppose the treatment planning problem is constrained to the determination of an appropriate incident fluence [J/cm^2] for complete coverage of a tumour with a diameter of 0.8 cm and a depth of 0.1 cm (assuming a safety margin is included), without compromising the surrounding tissue. For topical

ALA-based PDT of primary non-melanoma skin tumours such as basal cell carcinoma, the total incident light dose delivered generally ranges from 60 - 250 J/cm^2 [87]. To illustrate this problem, Fig. 4.10 (a) shows the fluence distribution resulting from an incident fluence of 100 J/cm^2. Assuming a threshold fluence dose of 30 J/cm^2 [88], the zone of necrosis extends up to 0.5 cm radially and 0.15 cm below the skin surface, as shown by the isofluence contours in Fig. 4.10 (b). This is not ideal as the tumour is only 0.1 cm in depth and 0.8 cm in diameter. To optimize the PDT effect based on the threshold model, the light dose distribution can be repeatedly adjusted to better match the shape of the tumour. In this case, the parameter to be optimized is the incident fluence, which requires iterative computation of the light dose distribution.

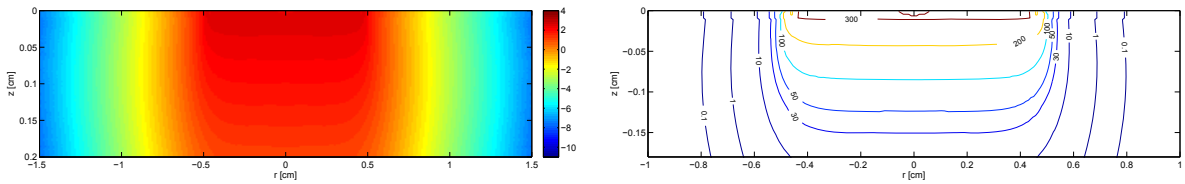
The optimization process becomes much more complicated for IPDT, which generally involves more sophisticated source geometries, increasing the degrees of freedom in the search space. An interesting development is the emergence of tailored diffuser, which can emit light along the optical fibre with a desired emission profile [89]. However, given the extra degrees of freedom introduced by tailored diffusers, treatment planning can become an even more daunting task without accelerating the light dosimetry procedure.

An alternative way to further decrease the computation time for MC-based light dosimetry (in addition to hardware acceleration schemes) is to optimize the number of photon packets launched based on the clinically acceptable level of uncertainty discussed in Chapter 1. As shown in Fig. 4.11(a)-(c), the statistical noise in the isofluence contours rapidly decreases as the number of photon packets is increased from 10^3 to 10^5. The difference between the contours produced at 10^5 [Fig. 4.11(c)] and 10^8 [Fig. 4.11(d)] photon packets is only noticeable at the lowest fluence level plotted (namely 0.1 J/cm^2) despite the fact that the simulation required almost 1000 times longer to complete. For this simple case, even 10^3 photon packets might suffice to determine the approximate location for the threshold light dose contour (assumed previously to be 30 J/cm^2). Compared to the threshold contour produced at 10^8 photon packets, the threshold contour generated using

only 10^3 photon packets is within +/- 0.3 - 0.5 mm in geometric uncertainty. However, the number of photon packets required clinically in the general 3-D case is anticipated to be orders of magnitude greater due to the following reasons.

First, absorption can no longer be scored in a radially symmetric grid based on the cylindrical coordinate system in the 3-D case with multiple sources. This assumption made by the MCML algorithm and the accelerated GPU-MCML greatly decreases the number of photon packets required. Second, the resolution of the grid dictates the photon statistics in the voxels. A finer grid requires a larger number of photons due to the lower photon count per voxel. Finally, if multiple sources were implanted in a heterogeneous 3-D tissue geometry, the light dose contours could be more irregular in shape. The tighter dose constraints placed around critical structures would require higher accuracy in the computation of these contours. In many cases, the threshold light dose for sensitive critical structures could be significantly lower. This is demonstrated in recent clinical trials in IPDT, in which clinical experience showed that the threshold light dose for the rectum is much lower than that of the prostate [15]. Therefore, for clinical dosimetry, it would be wise to increase the number of photon packets to more accurately simulate the light dose contours for even sub-threshold regions within the clinical target volume. These low-fluence regions have low photon counts and require more computation time to generate better statistics. An important observation here is that since the computation time is directly proportional to the number of photon packets launched, increasing the number of photon packets by a factor of ten also increases the simulation time by the same factor. As a result, MC-based light dosimetry for the 3-D case is expected to be even more time-consuming, which highlights the utility of the GPU-based approach for dosimetry.

A rigorous analysis of the trade-off between accuracy and simulation time for interstitial PDT treatment planning will be the subject of future work, as the capability to model such scenarios (involving complex 3-D tissue geometry and multiple implanted



(a) Intensity map **(b)** Isofluence contours

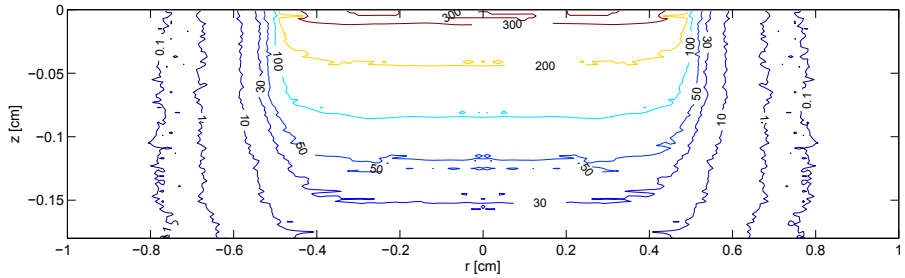
Figure 4.10: Fluence distribution [J/cm^2] in the skin model due to a 100 J/cm^2 flat, circular beam with a radius of 0.5 cm (generated with 10^8 photon packets): (a) an intensity map showing the fluence value at each voxel and (b) isofluence contours showing the location of selected fluence levels.

diffusers, possibly with a tailored emission profile) does not exist in the MCML code.

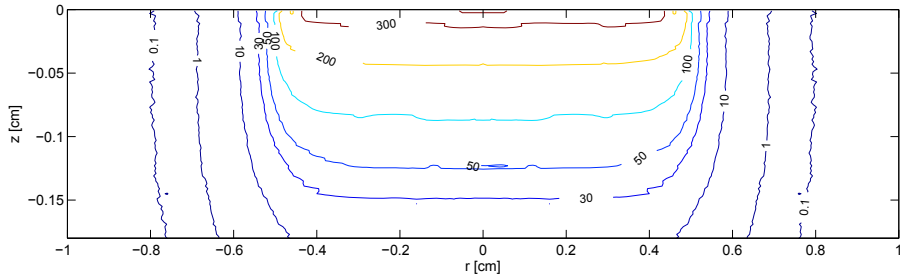
4.7 Summary

Using a skin model as the simulation input, the GPU-accelerated MCML implementation achieved a 277-fold speedup on a single NVIDIA GTX 280 graphics card with 30 multiprocessors, compared to a 3 GHz Intel Xeon processor. By scaling to a Quad-GPU system with 120 multiprocessors, a 1052-fold speedup was obtained. Performance can be further improved with the use of a GPU-based computing cluster. In fact, the use of NVIDIA GPUs for supercomputing has been made possible by the commercial development of the NVIDIA Tesla^{TM} Computing Server systems, such as the NVIDIA Tesla S1070 system [90] with 960 scalar processors, which can be stacked in a cluster configuration (with each server node priced at ∼20,000 USD [91] at the time of writing).

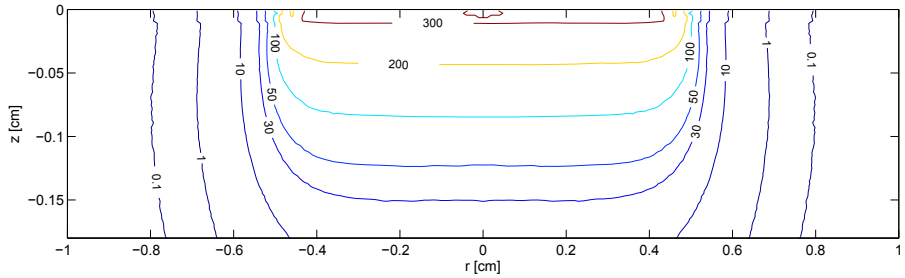
In this chapter, high accuracy was also demonstrated by the close correspondence between the isofluence lines generated by GPU-MCML and CPU-MCML. Finally, the development process illustrates the subtle nature of the underlying NVIDIA GPU architecture, necessitating a different approach to programming to achieve high performance. A number of unique challenges still remain before MC-based light dosimetry can be routinely used in PDT treatment planning, as discussed in the next chapter.



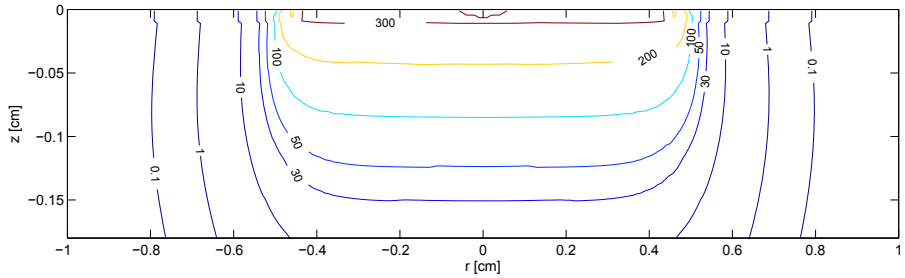
(a) 10^3 photon packets



(b) 10^4 photon packets



(c) 10^5 photon packets



(d) 10^8 photon packets

Figure 4.11: Isofluence contours [J/cm^2] for varying number of photon packets in the skin model. (A 100 J/cm^2 flat, circular beam with a radius of 0.5 cm is located at the origin.)

Chapter 5

Conclusions

5.1 Summary of Contributions

This thesis focused on the exploration of hardware-based approaches to accelerate a Monte Carlo simulation for modelling light propagation in biological tissue - specifically for the purpose of enabling MC-based light dosimetry in PDT treatment planning. Using the widely cited MCML code as a gold standard, two such approaches were attempted.

The first approach, as described in Chapter 3, involved designing custom computer hardware on an FPGA that was tailored to the MCML computation. The key advantages of the FPGA-based approach include the flexibility of customizing the hardware to efficiently execute the computation as well as the portability and low power consumption of a custom hardware solution compared to a CPU or GPU-based solution. In this work, such flexibility allowed the MCML computation to be broken down into many highly customized pipeline stages and enabled the use of numerous hardware-based optimizations, such as resource sharing across modules. The low power consumption also means that an expensive cooling system is not required for heat dissipation, unlike in conventional CPU-based supercomputing. However, the flexibility offered by an FPGA translates to the longer development time for the first working hardware prototype. Each subsequent

revision further required hours of system-wide design re-verification and re-compilation in the CAD software before it could be tested on a real FPGA board. Despite these challenges, a fully validated hardware design was successfully created that achieved a 28-fold speedup (using a skin model for simulation) on a modern DE-3 board with a Stratix III FPGA compared to a 3-GHz Intel Xeon processor. This design was also replicated on 4 FPGA devices using the TM-4 platform, showing a linear improvement in performance as a function of the number of FPGA devices used.

In Chapter 4, a CUDA program was written and optimized to accelerate the MCML computation using multiple NVIDIA GPUs. This approach required considerably less development effort to obtain the first working prototype with the use of CUDA - the C-like programming language extension made for general-purpose computing on NVIDIA GPUs. However, without careful optimizations, the initial CUDA program only achieved a 4-fold speedup (using the same skin model) on a NVIDIA 8800 GTX graphics card compared to the same Xeon processor mentioned previously. A considerable amount of time was dedicated to optimizing the CUDA program by tailoring the implementation to the unique features of the NVIDIA GPU architecture. This often required re-thinking the overall parallelization strategy and the clear identification of major performance bottlenecks. In the MCML code, the accumulation of absorption was one such bottleneck due to atomic accesses to the global memory. A solution based on caching the high fluence regions in shared memory was implemented in this work, which led to a significant reduction in simulation time. After numerous other GPU-specific optimizations, the final implementation showed an approximately 270-fold speedup on a NVIDIA GTX 280 GPU with 30 multiprocessors (or a total of 240 scalar processors). By scaling to 4 GPUs with a total of 120 multiprocessors (or 960 scalar processors), a speedup of over 1000-fold was obtained. To put this 1000-fold speedup in context, a treatment plan that requires a week of simulation time can be completed in approximately 10 minutes with this acceleration factor.

Although the GPU-based solution seems more promising at this stage, it is not entirely clear that this massive speedup can be maintained for the more sophisticated 3-D case due to the limitations imposed by the current NVIDIA GPU architecture. On the contrary, the FPGA-based approach enables the flexible customization of the hardware architecture to suit the computations, despite the greater development efforts required.

5.2 Future Work

For future work, a number of interesting research directions remain to be pursued to enable the use of MC-based light dosimetry models for PDT treatment planning. Several avenues for future research are outlined below.

5.2.1 Extension to 3-D and Support for Multiple Sources

The MCML code, which was the basis of this work, can only model light propagation from a single beam in a multi-layered tissue geometry. For realistic treatment planning, the extension to a 3-D model along with the support for multiple sources would be necessary to simulate more complex tissue and source geometries.

The main implications for the FPGA-based implementation are listed below:

1. **Need for external memory:** For 3-D cases, the requirements for storage will increase appreciably, due to the presence of a much larger absorption array and the need to store the tissue optical properties for potentially thousands of voxels in the worst case. As an example, for a 3-D array with 256 x 256 x 256 64-bit elements, a total of ∼1024 MBits of memory would be required (compared to 4 Mbits for a 2-D array with 256 x 256 64-bit elements). Therefore, the on-chip memory space will no longer be sufficient, and off-chip, external memory must be used. Note that the TM-4 platform contains 8 GB of external memory, so storage space is abundant.

2. **Increase in pipeline depth:** The use of external memory, instead of on-chip memory, also translates to a higher latency for memory accesses. Additional pipeline stages will be required in modules such as the Fluence Update Core to access the absorption array. Extra stages will also be required with the expansion of the model to 3-D, particularly in the Reflect/Transmit Core. However, due to the use of a pipelined hardware architecture, the performance will likely not be severely impacted by the increased pipeline depth given the stages are properly balanced to maintain the clock speed.

The extension to 3-D has different implications for the GPU-based implementation, as follows:

1. **Increase in register usage and code divergence:** A number of key simulation steps, particularly reflection at tissue interfaces, need to be modified to propagate the photon in a 3-D voxel-based tissue geometry. However, code expansion increases the register usage, which may degrade the performance due to the decrease in the maximum number of threads that can be created. Also, the divergence of the code will likely increase due to the increased number of interfaces for reflection, which may further degrade the performance.

2. **Increase in demand for shared memory:** The increase in the size of the absorption array, together with more photon sources, will require a modified approach to capture the high fluence region effectively in the small, but fast shared memory. This is an important strategy used in this thesis to avoid frequent atomic instructions that access the bigger, but much slower global memory.

3. **Increase in global memory usage/access:** The fast, constant memory used in this work will be insufficient to store the read-only tissue optical properties for potentially thousands of voxels, in the worst-case scenario. The need to store

and read these values will increase the frequency of global memory accesses, likely resulting in a further reduction in speedup.

5.2.2 Sources of Uncertainties

Apart from the geometric uncertainty of the simulated isofluence contours discussed in Section 4.6.3, a number of other sources of uncertainties must be accounted for in PDT treatment planning, several of which are listed below.

1. **Intra- and inter-patient variations in tissue optical properties:** Accurate light dosimetry depends on the accurate determination of tissue optical properties. The intra- and inter-patient variations in these values can directly affect the robustness of a treatment plan. To account for these variations, real-time light dosimetry for each patient has been proposed [21], but such in-vivo measurements were shown to have an uncertainty of at least +/-10% [92, 93]. Interestingly, the Monte Carlo method can also be used to estimate the absorption and scattering coefficients from reflectance data through iterative fitting procedures. The techniques presented in this work to accelerate MC simulations may also lead to the more accurate determination of tissue optical properties in the future.

2. **Diffuser placement:** The execution of the treatment plan is affected by the uncertainty in the placement of the light diffusing fibres in interstitial PDT applications. This uncertainty is especially critical for more complex anatomical regions such as those in the head and neck. Tailored diffusers, which can emit light with a customized emission profile, are being developed to shape the light dose in more complex cases [94]. However, the flexibility of tailoring the emission profile also makes the inverse problem in treatment planning more time-consuming to solve. This further highlights the utility of accelerating MC-based light dosimetry in PDT.

3. **Heterogeneity in the threshold light dose:** In a recent prostate IPDT clinical

trial that adopted the PDT threshold model for treatment planning, Davidson et al. reported the significant heterogeneity in the threshold light dose observed between patients [15]. This highlights the need for a more integrated approach in PDT treatment planning that takes into account other important parameters in PDT dosimetry, such as the dynamic nature of tissue oxygenation level, the effect of photobleaching on the photosensitizer concentration, and changes in the optical properties of the treated regions (as discussed in Chapter 1).

To account for the above sources of uncertainties, real-time online dosimetry may be a promising future direction for PDT treatment planning. For example, online dosimetry will allow in-vivo measurements of the changing tissue optical properties and tissue oxygenation level, which can be used to update the treatment plan in real-time. The acceleration of light dosimetry further makes real-time treatment planning a potential possibility.

5.2.3 PDT Treatment Planning using FPGA or GPU Clusters

With the anticipated increase in simulation time for the 3-D case, the computing infrastructure can be expanded as necessary for PDT treatment planning in complex scenarios. The potential for scalability was demonstrated in both Chapter 3 and Chapter 4, which showed the linear improvement in speedup achieved by the multi-FPGA and multi-GPU solutions for the MCML code.

The dramatic reduction in treatment planning time potentially achieved by an FPGA cluster or a GPU cluster may enable real-time treatment planning based on the most recent images of the treated volume, taking into account the changing tissue optical properties as the treatment progresses. Currently, pretreatment models assume constant values for tissue optical properties and ignore the dynamic nature of tissues, which could directly affect treatment outcomes in interstitial PDT [95]. The significant performance gain provided by the hardware approach, as demonstrated in this thesis, may enable

sophisticated MC-based models to be employed for PDT treatment planning in heterogeneous, spatially complex tissues in the future.

Appendix A

Source Code for the Hardware

The hardware description (or Verilog code) for the Fluence Update Core in Fig. 3.4 is given below for illustrative purposes. The original C code is also provided for comparison.

```
1  void Drop(InputStruct* In_Ptr, PhotonStruct* Photon_Ptr, OutStruct* Out_Ptr) {
2    double dwa;                  // Absorbed weight (Delta W in Section 2.2.4)
3    double x = Photon_Ptr->x;    // Current x-coordinate of the photon packet
4    double y = Photon_Ptr->y;    // Current y-coordinate of the photon packet
5    double izd, ird;             // Array indices to A[r][z] - temporary
6    short  iz, ir;               // Array indices to A[r][z] - final
7    short  layer = Photon_Ptr->layer; // Current layer of the photon packet
8    double mua, mus;             // Absorption, scattering coefficients (current layer)
9
10   izd = Photon_Ptr->z/In_Ptr->dz;          // compute array index for z dimension
11   if(izd>In_Ptr->nz-1) iz=In_Ptr->nz-1;    // if (outside the grid), absorb at the edge
12   else iz = izd;                           // otherwise, use the computed index izd
13   ird = sqrt(x*x+y*y)/In_Ptr->dr;          // compute array index for r dimension
14   if(ird>In_Ptr->nr-1) ir=In_Ptr->nr-1;    // if (outside the grid), absorb at the edge
15   else ir = ird;                           // otherwise, use the computed index ird
16
17   mua = In_Ptr->layerspecs[layer].mua;     // retrieve absorption coefficient
18   mus = In_Ptr->layerspecs[layer].mus;     // retrieve scattering coefficient
19   dwa = Photon_Ptr->w * mua/(mua+mus);     // Compute Delta W using Eq. 2.9
20   Photon_Ptr->w -= dwa;                    // decrease photon weight by Delta W
21   Out_Ptr->A_rz[ir][iz]+= dwa;             // add Delta W to absorption array at [ir,iz]
22 }
```

Listing A.1: Original Fluence Update implementation in the MCML code [40]

```
1  ////////////////////////////////////////////////////////////////////
2  ////  Absorber Hardware Module:                                 ////
3  ////  Pipelined Implementation of Fluence Update Computation    ////
4  ////////////////////////////////////////////////////////////////////
5
6  module Absorber (
7    //INPUTS
8    clock, reset, enable,
9    //From hopper
10   weight_hop, hit_hop, dead_hop,
11   //From Shared Registers
12   x_pipe, y_pipe, z_pipe, layer_pipe,
13   //From System Register File (5 layers)
14   muaFraction1, muaFraction2, muaFraction3, muaFraction4, muaFraction5,
15   //I/O to on-chip mem -- check interface
16   data, rdaddress, wraddress, wren, q,
17   //OUTPUT
18   weight_absorber
19 );
20
21 ////////////////////////////////////////////////////////////////////////////
22 //PARAMETERS
23 ////////////////////////////////////////////////////////////////////////////
24 parameter NR=256;
25 parameter NZ=256;
26 parameter NR_EXP=8;                //meaning NR=2^NR_exp or 2^8=256
27 parameter RGRID_SCALE_EXP=21;      //2^21 = RGRID_SCALE
28 parameter ZGRID_SCALE_EXP=21;      //2^21 = ZGRID_SCALE
29 parameter BIT_WIDTH=32;
30 parameter BIT_WIDTH_2=64;
31 parameter WORD_WIDTH=64;
32 parameter ADDR_WIDTH=16;           //256x256=2^8*2^8=2^16
33 parameter LAYER_WIDTH=3;
34 parameter PIPE_DEPTH = 37;
35
36 ////////////////////////////////////////////////////////////////////////////
37 //INPUTS
38 ////////////////////////////////////////////////////////////////////////////
39 input clock;
40 input reset;
```

```
41  input enable;
42
43  //From hopper
44  input [BIT_WIDTH-1:0] weight_hop;
45  input hit_hop;
46  input dead_hop;
47
48  //From Shared Reg
49  input signed [BIT_WIDTH-1:0] x_pipe;
50  input signed [BIT_WIDTH-1:0] y_pipe;
51  input [BIT_WIDTH-1:0] z_pipe;
52  input [LAYER_WIDTH-1:0] layer_pipe;
53
54  //From System Reg File
55  input [BIT_WIDTH-1:0] muaFraction1, muaFraction2, muaFraction3, muaFraction4,
        muaFraction5;
56
57  ////////////////////////////////////////////////////////////////////////////
58  //OUTPUTS
59  ////////////////////////////////////////////////////////////////////////////
60  output [BIT_WIDTH-1:0] weight_absorber;
61
62  ////////////////////////////////////////////////////////////////////////////
63  //I/O to on-chip mem -- check interface
64  ////////////////////////////////////////////////////////////////////////////
65  output [WORD_WIDTH-1:0] data;
66  output [ADDR_WIDTH-1:0] rdaddress, wraddress;
67  output wren;      reg wren;
68  input [WORD_WIDTH-1:0] q;
69
70  ////////////////////////////////////////////////////////////////////////////
71  //Local AND Registered Value Variables
72  ////////////////////////////////////////////////////////////////////////////
73  //STAGE 1 - Initialization and no computation
74
75  //STAGE 2
76  reg [BIT_WIDTH_2-1:0] x2_temp, y2_temp;    //From mult
77  reg [BIT_WIDTH_2-1:0] x2_P, y2_P;          //Registered Value
78
79  //STAGE 3
80  reg [BIT_WIDTH_2-1:0] r2_temp, r2_P;
```

```
81  wire [BIT_WIDTH_2-1:0] r2_P_wire;
82
83  //STAGE 4
84  reg [BIT_WIDTH-1:0]    fractionScaled;
85  reg [BIT_WIDTH-1:0]    weight_P4;
86  reg [BIT_WIDTH-1:0]    r_P;
87  wire [BIT_WIDTH-1:0]     r_P_wire;
88  reg [BIT_WIDTH_2-1:0] product64bit;
89  reg [BIT_WIDTH-1:0] dwa_temp;
90
91  //STAGE 14
92  reg [BIT_WIDTH-1:0]    ir_temp;
93  reg [BIT_WIDTH-1:0]    iz_temp;
94
95  //STAGE 15
96  reg [BIT_WIDTH-1:0]    ir_P;
97  reg [BIT_WIDTH-1:0]    iz_P;
98  reg [BIT_WIDTH-1:0]    ir_scaled;
99  reg [ADDR_WIDTH-1:0] rADDR_temp;
100 reg [ADDR_WIDTH-1:0] rADDR_16;
101
102 //STAGE 16
103 reg [WORD_WIDTH-1:0] oldAbs_MEM;
104 reg [WORD_WIDTH-1:0] oldAbs_P;
105 reg [ADDR_WIDTH-1:0] rADDR_17;
106
107 //STAGE 17
108 reg [BIT_WIDTH-1:0] weight_P;
109 reg [BIT_WIDTH-1:0] dwa_P;
110 reg [BIT_WIDTH-1:0] newWeight;
111 reg [WORD_WIDTH-1:0] newAbs_P;
112 reg [WORD_WIDTH-1:0] newAbs_temp;
113 reg [ADDR_WIDTH-1:0] wADDR;
114
115 ////////////////////////////////////////////////////////////////////////////
116 //PIPELINE weight, hit, dead
117 ////////////////////////////////////////////////////////////////////////////
118 //WIRES FOR CONNECTING REGISTERS
119 wire  [BIT_WIDTH-1:0]      weight  [PIPE_DEPTH:0];
120 wire  hit  [PIPE_DEPTH:0];
121 wire  dead  [PIPE_DEPTH:0];
```

```
122
123 //ASSIGNMENTS FROM INPUTS TO PIPE
124 assign weight[0] = weight_hop;
125 assign hit[0] = hit_hop;
126 assign dead[0] = dead_hop;
127
128 //ASSIGNMENTS FROM PIPE TO OUTPUT
129 assign weight_absorber =weight[PIPE_DEPTH];
130
131 //GENERATE PIPELINE
132 genvar i;
133 generate
134   for(i=PIPE_DEPTH; i>0; i=i-1) begin: weightHitDeadPipe
135     case(i)
136     //REGISTER 17 on diagram!!
137     18:
138     begin
139     PhotonBlock2 photon(
140       //Inputs
141       .clock(clock),
142       .reset(reset),
143       .enable(enable),
144
145       .i_x(newWeight),
146       .i_y(hit[17]),
147       .i_z(dead[17]),
148
149       //Outputs
150       .o_x(weight[18]),
151       .o_y(hit[18]),
152       .o_z(dead[18])
153     );
154     end
155     default:
156     begin
157     PhotonBlock2 photon(
158       //Inputs
159       .clock(clock),
160       .reset(reset),
161       .enable(enable),
162
```

```
163         .i_x(weight[i-1]),
164         .i_y(hit[i-1]),
165         .i_z(dead[i-1]),
166
167         //Outputs
168         .o_x(weight[i]),
169         .o_y(hit[i]),
170         .o_z(dead[i])
171       );
172       end
173       endcase
174     end
175 endgenerate
176
177 ////////////////////////////////////////////////////////////////////////////
178 //PIPELINE ir, iz, dwa
179 ////////////////////////////////////////////////////////////////////////////
180 //WIRES FOR CONNECTING REGISTERS
181 wire   [BIT_WIDTH-1:0]       ir   [PIPE_DEPTH:0];
182 wire   [BIT_WIDTH-1:0]       iz   [PIPE_DEPTH:0];
183 wire   [BIT_WIDTH-1:0]       dwa  [PIPE_DEPTH:0];
184
185 //ASSIGNMENTS FROM INPUTS TO PIPE
186 assign ir[0] = 0;
187 assign iz[0] = 0;
188 assign dwa[0] = 0;
189
190 //GENERATE PIPELINE
191 generate
192   for(i=PIPE_DEPTH; i>0; i=i-1) begin: IrIzDwaPipe
193     case(i)
194     //NOTE: STAGE 14 --> REGISTER 14 on diagram !!    ir, iz
195     15:
196     begin
197     PhotonBlock1 photon(
198       //Inputs
199       .clock(clock),
200       .reset(reset),
201       .enable(enable),
202
203       .i_x(ir_temp),
```

```
204         .i_y(iz_temp),
205         .i_z(dwa[14]),
206
207         //Outputs
208         .o_x(ir[15]),
209         .o_y(iz[15]),
210         .o_z(dwa[15])
211       );
212       end
213
214       //NOTE: STAGE 4 --> REGISTER 4 on diagram !!    dwa
215       5:
216       begin
217       PhotonBlock1 photon(
218         //Inputs
219         .clock(clock),
220         .reset(reset),
221         .enable(enable),
222
223         .i_x(ir[4]),
224         .i_y(iz[4]),
225         .i_z(dwa_temp),
226
227         //Outputs
228         .o_x(ir[5]),
229         .o_y(iz[5]),
230         .o_z(dwa[5])
231       );
232       end
233
234       default:
235       begin
236       PhotonBlock1 photon(
237         //Inputs
238         .clock(clock),
239         .reset(reset),
240         .enable(enable),
241
242         .i_x(ir[i-1]),
243         .i_y(iz[i-1]),
244         .i_z(dwa[i-1]),
```

```
245
246         //Outputs
247         .o_x(ir[i]),
248         .o_y(iz[i]),
249         .o_z(dwa[i])
250       );
251       end
252       endcase
253     end
254 endgenerate
255
256 ////////////////////////////////////////////////////////////////////////////
257 //STAGE BY STAGE PIPELINE DESIGN
258 ////////////////////////////////////////////////////////////////////////////
259
260 ///////////////STAGE 2 - square of x and y//////////////////////////
261 always @(*) begin
262   if (reset) begin
263     x2_temp=0;
264     y2_temp=0;
265   end
266   else  begin
267      x2_temp=x_pipe*x_pipe;
268      y2_temp=y_pipe*y_pipe;
269   end
270 end
271
272 ///////////////STAGE 3 - square of r//////////////////////////
273 always @(*) begin
274   if (reset)
275     r2_temp=0;
276   else
277     r2_temp=x2_P+y2_P;
278 end
279
280 ///////////////STAGE 4 - Find r and dwa//////////////////////////
281 //Create MUX
282 always@(*)
283    case(layer_pipe)
284        1: fractionScaled=muaFraction1;
285        2: fractionScaled=muaFraction2;
```

```
286         3: fractionScaled=muaFraction3;
287         4: fractionScaled=muaFraction4;
288         5: fractionScaled=muaFraction5;
289         default: fractionScaled=0; //Sys Reset case
290     endcase
291
292
293 always @(*) begin
294     if (reset) begin
295         weight_P4=0;
296       r_P=0;
297         product64bit=0;
298         dwa_temp=0;
299     end
300     else begin
301         weight_P4=weight[4];
302       r_P=r_P_wire;  //Connect to sqrt block
303         product64bit=weight_P4*fractionScaled;
304
305         //Checking corner cases
306         if (dead[4]==1)        //Dead photon
307             dwa_temp=weight_P4;//drop all its weight
308         else if (hit[4]==1)    //Hit Boundary
309             dwa_temp=0;        //Don't add to absorption array
310         else
311             dwa_temp=product64bit[63:32];
312     end
313 end
314
315 assign r2_P_wire=r2_P;
316
317 Sqrt_64b  squareRoot (
318         .clk(clock),
319         .radical(r2_P_wire),
320         .q(r_P_wire),
321         .remainder()
322         );
323
324 ///////////////STAGE 14 - Find ir and iz//////////////////////////
325 always @(*) begin
326     if (reset) begin
```

```
327       ir_temp=0;
328       iz_temp=0;
329     end
330     else begin
331       ir_temp=r_P>>RGRID_SCALE_EXP;
332       iz_temp=z_pipe>>ZGRID_SCALE_EXP;
333
334       //Checking corner cases!!!
335       if (dead[14]==1) begin
336             ir_temp=NR-1;
337             iz_temp=NZ-1;
338       end
339       else if (hit[14]==1) begin
340          ir_temp=0;
341          iz_temp=0;
342       end
343
344         if (iz_temp>=NZ)
345             iz_temp=NZ-1;
346
347         if (ir_temp>=NR)
348             ir_temp=NR-1;
349
350     end
351 end
352
353 ///////////////STAGE 15 - Compute MEM address//////////////////////
354 always @(*) begin
355     if (reset) begin
356         ir_P=0;
357         iz_P=0;
358         ir_scaled=0;
359          rADDR_temp=0;
360      end
361     else begin
362         ir_P=ir[15];
363         iz_P=iz[15];
364         ir_scaled=ir_P<<NR_EXP;
365          rADDR_temp=ir_scaled+iz_P;
366      end
367 end
```

```
368
369 ///////////////STAGE 16 - MEM read//////////////////////////
370 always @(*) begin
371   if (reset)
372     oldAbs_MEM=0;
373   else begin
374     //Check Corner cases (RAW hazards)
375      if (ir[16]==ir[17] && iz[16]==iz[17])
376         oldAbs_MEM=newAbs_temp;
377      else if (ir[16]==ir[18] && iz[16]==iz[18])
378         oldAbs_MEM=newAbs_P;          //RAW hazard
379      else
380         oldAbs_MEM=q;    //Connect to REAL dual-port MEM
381   end
382
383 end
384
385 ///////////////STAGE 17 - Update Weight//////////////////////////
386 always @(*) begin
387   if(reset) begin
388      dwa_P=0;
389     weight_P=0;
390     newWeight = 0;
391     newAbs_temp =0;
392    end
393   else begin
394      dwa_P=dwa[17];
395      weight_P=weight[17];
396     newWeight=weight_P-dwa_P;
397     newAbs_temp=oldAbs_P+dwa_P;    //Check bit width casting (64-bit<--64-bit+32-bit)
398    end
399 end
400
401 ////////////////////////////////////////////////////////////////////////////////
402 //STAGE BY STAGE - EXTRA REGISTERS
403 ////////////////////////////////////////////////////////////////////////////////
404 always @ (posedge clock)
405 begin
406   if (reset) begin
407     //Stage 2
408     x2_P<=0;
```

```
409       y2_P <=0;
410
411       //Stage 3
412       r2_P <=0;
413
414       //Stage 15
415        rADDR_16<=0;
416
417       //Stage 16
418       oldAbs_P <=0;
419       rADDR_17<=0;
420
421       //Stage 17
422       newAbs_P <=0;
423      wADDR <=0;
424     end
425
426     else if (enable) begin
427       //Stage 2
428       x2_P<=x2_temp;      //From comb logic above
429       y2_P<=y2_temp;
430
431       //Stage 3
432       r2_P<=r2_temp;
433
434       //Stage 15
435        rADDR_16<=rADDR_temp;
436
437        //Stage 16
438       oldAbs_P<=oldAbs_MEM;
439       rADDR_17<=rADDR_16;
440
441        //Stage 17
442        newAbs_P<=newAbs_temp;
443        wADDR <=rADDR_17;
444     end
445 end
446
447 ////////////////////////////////////////////////////////////////////////////
448 //INTERFACE to on-chip MEM
449 ////////////////////////////////////////////////////////////////////////////
```

```
450 always @ (posedge clock)
451 begin
452   if (reset)
453       wren <=0;
454   else
455       wren<=1;          //Memory enabled every cycle after global enable
456 end
457
458 assign rdaddress=rADDR_temp;
459 assign wraddress=rADDR_17;
460 assign data=newAbs_temp;
461
462 endmodule
463
464
465 ////////////////////////////////////////////////////////////////////
466 ////  INTERNAL                                                  ////
467 ////  Pipeline Module - Type 1 (for 32-bit, 32-bit, 32-bit case)  ////
468 ////  Implementation of Internal Pipeline Module                ////
469 ////////////////////////////////////////////////////////////////////
470 //Photons that make up the register pipeline
471 module PhotonBlock1(
472   //Inputs
473   clock,
474   reset,
475   enable,
476
477   i_x,
478   i_y,
479   i_z,
480
481   //Outputs
482   o_x,
483   o_y,
484   o_z
485   );
486
487 ////////////////////////////////////////////////////////////////////////////
488 //PARAMETERS
489 ////////////////////////////////////////////////////////////////////////////
490 parameter BIT_WIDTH=32;
```

```
491 input clock;
492 input reset;
493 input enable;
494 input [BIT_WIDTH-1:0] i_x;
495 input [BIT_WIDTH-1:0] i_y;
496 input [BIT_WIDTH-1:0] i_z;
497
498 output  [BIT_WIDTH-1:0] o_x;
499 output  [BIT_WIDTH-1:0] o_y;
500 output  [BIT_WIDTH-1:0] o_z;
501
502 wire  clock;
503 wire  reset;
504 wire  enable;
505 wire  [BIT_WIDTH-1:0] i_x;
506 wire  [BIT_WIDTH-1:0] i_y;
507 wire  [BIT_WIDTH-1:0] i_z;
508
509 reg [BIT_WIDTH-1:0] o_x;
510 reg [BIT_WIDTH-1:0] o_y;
511 reg [BIT_WIDTH-1:0] o_z;
512
513 always @ (posedge clock)
514   if(reset) begin
515     o_x<= {BIT_WIDTH{1'b0}} ;
516     o_y<= {BIT_WIDTH{1'b0}};
517     o_z<= {BIT_WIDTH{1'b0}};
518   end else if(enable) begin
519     o_x<= i_x;
520     o_y<= i_y;
521     o_z<= i_z;
522   end
523 endmodule
524
525
526 ////////////////////////////////////////////////////////////////////
527 ////  INTERNAL                                                  ////
528 ////  Pipeline Module - Type 2 (for 32-bit, 1-bit, 1-bit case)  ////
529 ////  Implementation of Internal Pipeline Module                ////
530 ////////////////////////////////////////////////////////////////////
531
```

```
532 //Photons that make up the register pipeline
533 module PhotonBlock2(
534   //Inputs
535   clock,
536   reset,
537   enable,
538
539   i_x,
540   i_y,
541   i_z,
542
543   //Outputs
544   o_x,
545   o_y,
546   o_z
547   );
548
549 ////////////////////////////////////////////////////////////////////////////////
550 //PARAMETERS
551 ////////////////////////////////////////////////////////////////////////////////
552 parameter BIT_WIDTH=32;
553
554 input clock;
555 input reset;
556 input enable;
557 input [BIT_WIDTH-1:0] i_x;
558 input i_y;
559 input i_z;
560
561 output  [BIT_WIDTH-1:0] o_x;
562 output  o_y;
563 output  o_z;
564
565 wire  clock;
566 wire  reset;
567 wire  enable;
568 wire  [BIT_WIDTH-1:0] i_x;
569 wire  i_y;
570 wire  i_z;
571
572 reg [BIT_WIDTH-1:0] o_x;
```

```
573 reg o_y;
574 reg o_z;
575
576 always @ (posedge clock)
577   if(reset) begin
578     o_x<= {BIT_WIDTH{1'b0}} ;
579     o_y<= 1'b0;
580     o_z<= 1'b0;
581   end else if(enable) begin
582     o_x<= i_x;
583     o_y<= i_y;
584     o_z<= i_z;
585   end
586 endmodule
```

Listing A.2: Absorber.v (Hardware Description for Fluence Update Core)

Appendix B

Source Code for the CUDA program

This appendix only shows the most interesting parts of the source code, which are the GPU kernel code that performs the key part of the simulation (`MCMLcuda_kernel.h` and `MCMLcuda_kernel.cu`) and the host software (`MCMLcuda.cu`) that performs miscellaneous tasks. These tasks include initializing the program, launching the kernel on the GPU, and retrieving the simulation output from the GPU for display on the host computer. Implementation details for other less interesting tasks (e.g., parsing the simulation input) are omitted.

```
1  #ifndef _MCMLCUDA_KERNEL_CUH_
2  #define _MCMLCUDA_KERNEL_CUH_
3
4  #include "MCML/src/MCML.H"
5
6  ////////////////////////////////
7  // Overflow handler usage counter
8  // For debugging only
9  #define COUNT_OVERFLOW
10
11 ////////////////////////////////
12 // precomputing layer spec
13 #define PRECOMPUTE
14
15 ////////////////////////////////
16 //GPU Kernel specific
```

```
17 ////////////////////////////
18 #define NTHREAD 256
19 #define NBLOCK 30
20
21 ////////////////////////////
22 //Shared mem
23 ////////////////////////////
24 //NOTE: Make sure MAX_IR * MAX_IZ is less than 4096 (max - 16kB shared)
25 //and divisible by NTHREAD for proper initialization to 0.
26 #define MAX_IR 28
27 #define MAX_IZ 128
28
29 #define MAX_OVERFLOW 4000000000 //MAX_UINT32 - some buffer room
30
31 ////////////////////////////
32 //Multi-GPU
33 ////////////////////////////
34 #define MAX_GPU_COUNT 4
35
36 ////////////////////////////
37 //MCML constants
38 ////////////////////////////
39 #define WEIGHT_SCALE 12000000
40
41 #define WTH 1E-4F
42 #define PI_const 3.1415926f
43
44 #define COSNINETYDEG 1.0E-6F
45 #define COSZERO (1.0F - 1.0E-6F)
46 #define CHANCE 0.1F
47 #define INVCHANCE 10.0F
48
49 ////////////////////////////////////////////////////////////////////////////
50 // CPU-side data structures (multi-GPU)
51 ////////////////////////////////////////////////////////////////////////////
52 typedef struct {
53   int deviceid;
54   double ** A_rz; // 2D absorption
55   unsigned int num_photons; // specified in input file and divided by GPU_N
56   unsigned int s1_base, s2_base, s3_base; //distinct seeds for each GPU
57 } GPUinout;
```

```
58
59 ////////////////////////////////////////////////////////////////////////////
60 // Internal GPU-side data structures
61 ////////////////////////////////////////////////////////////////////////////
62 /************
63 * Simplified version of input specifications that GPU kernel needs
64 * Note that GPU takes in InputStructGPU and LayerStruct separately
65 */
66 typedef struct {
67   unsigned int num_photons; /* to be traced. */
68   float dz; /* z grid separation.[cm] */
69   float dr; /* r grid separation.[cm] */
70   unsigned int nz; /* array range 0..nz-1. */
71   unsigned int nr; /* array range 0..nr-1. */
72
73   unsigned int num_layers; /* number of layers. */
74 } InputStructGPU;
75
76 typedef struct {
77   float z0, z1; /* z coordinates of a layer. [cm] */
78   float n; /* refractive index of a layer. */
79   float mua; /* absorption coefficient. [1/cm] */
80   float mus; /* scattering coefficient. [1/cm] */
81   float g; /* anisotropy. */
82
83   float cos_crit0, cos_crit1;
84 } LayerStructGPU;
85
86 static __device__ __constant__ InputStructGPU d_In_Ptr;
87
88 // NOTE: Fixed at 20 for now (2 ambient layers--> so usable 18 layers)
89 #define MAXLAYER 20
90
91 // layer spec array
92 #ifdef PRECOMPUTE
93 typedef struct
94 {
95   float z0, z1;        /* z coordinates of a layer. [cm] */
96   float n;             /* refractive index of a layer. */
97   float g;             /* anisotropy. */
98
```

```
99    float cos_crit0, cos_crit1;
100
101   int is_glass;         /* non-zero if mua == mus == 0.0F, zero otherwise */
102   float muas;           /* mua + mus */
103   float rmuas;          /* 1/(mua+mus) */
104   float mua_muas;       /* mua/(mua+mus) */
105 } LayerStructGPU_precomp;
106
107 static __device__ __constant__ LayerStructGPU_precomp d_layerspecs[MAXLAYER];
108 #else
109 static __device__ __constant__ LayerStructGPU d_layerspecs[MAXLAYER];
110 #endif
111
112 #endif // _MCMLCUDA_KERNEL_CUH_
```

Listing B.1: MCMLcuda_kernel.h (Header File for GPU program)

```
1  /****************************************************
2   *  CUDA GPU version of MCML
3   *  Kernel code for Monte Carlo simulation of photon
4   *  propagation in multi-layered turbid media.
5   *  Using shared memory for high fluence region (near photon beam)
6   ****/
7
8  #ifndef _MCMLCUDA_KERNEL_H_
9  #define _MCMLCUDA_KERNEL_H_
10
11 #include <stdio.h>
12 #include "MCMLcuda_kernel.h"
13
14 //****************************************************
15 // >>>>>>>>>> Launch()
16 // Initialize the photon.
17 __device__ void Launch(float *w, float WINIT, unsigned int *dead,
18     unsigned int *layer, float *s, float *sleft, float *x, float *y, float *z,
19     float *ux, float *uy, float *uz) {
20   *w = WINIT;
21   *dead = 0;
22   *layer = 1;
23   *s = 0.0f;
24   *sleft = 0.0f;
25
```

```
26    *x = 0.0f;
27    *y = 0.0f;
28    *z = 0.0f;
29    *ux = 0.0f;
30    *uy = 0.0f;
31    *uz = 1.0f;
32  }
33
34  //*********************************************************
35  // >>>>>>>>> Hop()
36  // Move the photon s away in the current layer of medium.
37  __device__ void Hop(float s, float ux, float uy, float uz, float *x, float *y,
38      float *z) {
39    *x += s * ux;
40    *y += s * uy;
41    *z += s * uz;
42  }
43
44  /*********************************************************
45   *  >>>>>>>>> StepSizeInTissue()
46   *  Pick a step size for a photon packet when it is in tissue.
47   *  If the member sleft is zero, make a new step size
48   *  with: -log(rnd)/(mua+mus).
49   *  Otherwise, pick up the leftover in sleft.
50   ****/
51  __device__ void ComputeStepSize(unsigned int layer, float rand0, float *s,
52      float *sleft) {
53
54  #ifdef PRECOMPUTE
55    float rmuas=d_layerspecs[layer].rmuas;
56
57    if (*sleft == 0.0f) { /* make a new step. */
58      *s = (-__logf(rand0)) * rmuas;
59    } else { /* take the leftover. */
60      *s = (*sleft) * rmuas;
61      *sleft = 0.0f;
62    }
63  #else
64    float muas=d_layerspecs[layer].mua+ d_layerspecs[layer].mus;
65
66    if (*sleft == 0.0f) { /* make a new step. */
```

```
67      *s = __fdividef((-__logf(rand0)), muas);
68    } else { /* take the leftover. */
69      *s = __fdividef((*sleft), muas);
70      *sleft = 0.0f;
71    }
72  #endif
73
74  }
75
76  /***********************************************************
77   *  >>>>>>>>> HitBoundary()
78   *   Check if the step will hit the boundary.
79   *   Return 1 if hit boundary.
80   *   Return 0 otherwise.
81   *
82   *   If the projected step hits the boundary, the members
83   *   s and sleft of Photon_Ptr are updated.
84   ****/
85  __device__ int HitBoundary(unsigned int layer, float z0, float z1, float z, float uz,
86      float *s, float *sleft) {
87
88    float dl_b; /* step size to boundary. */
89
90    /* Distance to the boundary. */
91    if (uz > 0.0f)
92      dl_b = __fdividef((z1 - z), uz); /* dl_b>0. */
93    else if (uz < 0.0f)
94      dl_b = __fdividef((z0 - z), uz); /* dl_b>0. */
95
96    if (uz != 0.0f && (*s) > dl_b) {
97      /* not horizontal & crossing. */
98
99  #ifdef PRECOMPUTE
100   float muas=d_layerspecs[layer].muas;
101 #else
102   float muas=d_layerspecs[layer].mua+ d_layerspecs[layer].mus;
103 #endif
104
105     *sleft = ((*s) - dl_b) * (muas);
106     *s = dl_b;
107     return 1;
```

```
108   } else
109     return 0;
110 }
111
112 /*****************************************************
113  *  >>>>>>>>> UltraFast version()
114  *  >>>>>>>>> Reduced divergence
115 */
116
117 __device__ void FastReflectTransmit(float rand3, float *ux, float *uy,
118     float *uz, unsigned int *layer, unsigned int* dead) {
119
120     /* Collect all info that depend on the sign of "uz". */
121     float cos_crit;
122     int new_photon_layer;
123     if (*uz > 0.0F) {
124         cos_crit = d_layerspecs[(*layer)].cos_crit1;
125         new_photon_layer = (*layer)+1;
126     } else {
127         cos_crit = d_layerspecs[(*layer)].cos_crit0;
128         new_photon_layer = (*layer)-1;
129     }
130
131     // cosine of the incident angle (0 to 90 deg)
132     float ca1 = fabsf(*uz);
133
134     // The default move is to reflect.
135     *uz = -(*uz);
136
137     // Moving this check down to "RFresnel = 0.0F" slows down the
138     // application, possibly because every thread is forced to do
139     // too much.
140     if (ca1 > cos_crit)
141     {
142         /*  Compute the Fresnel reflectance. */
143
144         // incident and transmit refractive index
145         float ni = d_layerspecs[(*layer)].n;
146         float nt = d_layerspecs[new_photon_layer].n;
147         float ni_nt = __fdividef(ni, nt);   // reused later
148
```

```
149         float sa1 = sqrtf(1.0F-ca1*ca1);
150         float sa2 = fminf(ni_nt * sa1, 1.0F);
151         if (ca1 > COSZERO) sa2 = sa1;
152         float uz1 = sqrtf(1.0F-sa2*sa2);     // uz1 = ca2
153
154         float ca1ca2 = ca1 * uz1;
155         float sa1sa2 = sa1 * sa2;
156         float sa1ca2 = sa1 * uz1;
157         float ca1sa2 = ca1 * sa2;
158
159         float cam = ca1ca2 + sa1sa2; /* c- = cc + ss. */
160         float sap = sa1ca2 + ca1sa2; /* s+ = sc + cs. */
161         float sam = sa1ca2 - ca1sa2; /* s- = sc - cs. */
162
163         float rFresnel = __fdividef(sam, sap*cam);
164         rFresnel *= rFresnel;
165         rFresnel *= (ca1ca2*ca1ca2 + sa1sa2*sa1sa2);
166
167         // Hope "uz1" is very close to "ca1".
168         if (ca1 > COSZERO) rFresnel = 0.0F;
169         // In this case, we do not care if "uz1" is exactly 0.
170         if (ca1 < COSNINETYDEG || sa2 == 1.0F) rFresnel = 1.0F;
171
172         if (rFresnel < rand3)
173         {
174             // The move is to transmit.
175             *layer = new_photon_layer;
176             *dead = (*layer == 0 || *layer > d_In_Ptr.num_layers);
177
178             // Let's do these even if the photon is dead.
179             *ux *= ni_nt;
180             *uy *= ni_nt;
181             // Is this faster?
182             *uz = -copysignf(uz1, *uz);
183         }
184     }
185
186 }
187
188 /*****************************************************
189  *  >>>>>>>>> Spin()
```

```
190  *  Choose a new direction for photon propagation by
191  *  sampling the polar deflection angle theta and the
192  *  azimuthal angle psi.
193  *
194  *  Note:
195  *    theta: 0 - pi so sin(theta) is always positive
196  *    feel free to use sqrtf() for cos(theta).
197  *
198  *    psi:   0 - 2pi
199  *    for 0-pi  sin(psi) is +
200  *    for pi-2pi sin(psi) is -
201  ****/
202 __device__ __inline__ void Spin(float *g, float *ux, float *uy, float *uz,
203     float *rand5, float *rand7) {
204   //>>>>>>>>> Spin()
205   float cost, sint; /* cosine and sine of the polar deflection angle theta. */
206   float cosp, sinp; /* cosine and sine of the azimuthal angle psi. */
207   float SIGN;
208   float temp; //deep within the function (if ... else)
209   float last_ux = *ux;
210   float last_uy = *uy;
211   float last_uz = *uz;
212
213   /*************************************************
214    * >>>>>>> SpinTheta
215    *  Choose (sample) a new theta angle for photon propagation
216    *  according to the anisotropy.
217    *
218    *  If anisotropy g is 0, then
219    *    cos(theta) = 2*rand-1.
220    *  otherwise
221    *    sample according to the Henyey-Greenstein function.
222    *
223    *  Returns the cosine of the polar deflection angle theta.
224    ****/
225
226     cost = 2.0F * (*rand5) - 1.0F;
227     temp = __fdividef((1.0f - (*g) * (*g)), 1.0F + (*g)*cost);
228     if ((*g) != 0.0F) {
229         cost = __fdividef(1.0F + (*g) * (*g) - temp*temp, 2.0F * (*g));
230     }
```

```
231     cost = fmaxf(cost, -1.0F);
232     cost = fminf(cost, 1.0F);
233
234   sint = sqrtf(1.0f - cost * cost);
235
236   __sincosf(2.0f * PI_const * (*rand7), &cosp, &sinp);
237
238   if (fabsf(last_uz) > COSZERO) { /* normal incident. */
239     *ux = sint * cosp;
240     *uy = sint * sinp;
241     SIGN = ((last_uz) >= 0.0f ? 1.0f : -1.0f);
242     *uz = cost * SIGN;
243
244   } else { /* regular incident. */
245     temp = rsqrtf(1.0f - last_uz * last_uz);
246     *ux = sint * (last_ux * last_uz * cosp - last_uy * sinp) * temp
247         + last_ux * cost;
248     *uy = sint * (last_uy * last_uz * cosp + last_ux * sinp) * temp
249         + last_uy * cost;
250     *uz = __fdividef(-sint * cosp, temp) + last_uz * cost;
251   }
252 }
253
254 //*********************************************************
255 //Generate a random number between 0 and 1.
256 __device__ __inline__ void Rand(unsigned int * s1, unsigned int * s2,
257     unsigned int * s3, float *rand0, float *rand3,
258     float *rand5, float *rand7, float *rand8) {
259   unsigned int b;
260
261   //rand0
262   b = (((*s1 << 13) ^ *s1) >> 19);
263   *s1 = (((*s1 & 4294967294) << 12) ^ b);
264   b = (((*s2 << 2) ^ *s2) >> 25);
265   *s2 = (((*s2 & 4294967288) << 4) ^ b);
266   b = (((*s3 << 3) ^ *s3) >> 11);
267   *s3 = (((*s3 & 4294967280) << 17) ^ b);
268   *rand0 = (float) ((*s1 ^ *s2 ^ *s3) * 2.3283064365e-10f);
269
270   //rand3
271   b = (((*s1 << 13) ^ *s1) >> 19);
```

```
272   *s1 = (((*s1 & 4294967294) << 12) ^ b);
273   b = (((*s2 << 2) ^ *s2) >> 25);
274   *s2 = (((*s2 & 4294967288) << 4) ^ b);
275   b = (((*s3 << 3) ^ *s3) >> 11);
276   *s3 = (((*s3 & 4294967280) << 17) ^ b);
277   *rand3 = (float) ((*s1 ^ *s2 ^ *s3) * 2.3283064365e-10f);
278
279   //rand5
280   b = (((*s1 << 13) ^ *s1) >> 19);
281   *s1 = (((*s1 & 4294967294) << 12) ^ b);
282   b = (((*s2 << 2) ^ *s2) >> 25);
283   *s2 = (((*s2 & 4294967288) << 4) ^ b);
284   b = (((*s3 << 3) ^ *s3) >> 11);
285   *s3 = (((*s3 & 4294967280) << 17) ^ b);
286   *rand5 = (float) ((*s1 ^ *s2 ^ *s3) * 2.3283064365e-10f);
287
288
289   //rand7
290   b = (((*s1 << 13) ^ *s1) >> 19);
291   *s1 = (((*s1 & 4294967294) << 12) ^ b);
292   b = (((*s2 << 2) ^ *s2) >> 25);
293   *s2 = (((*s2 & 4294967288) << 4) ^ b);
294   b = (((*s3 << 3) ^ *s3) >> 11);
295   *s3 = (((*s3 & 4294967280) << 17) ^ b);
296   *rand7 = (float) ((*s1 ^ *s2 ^ *s3) * 2.3283064365e-10f);
297
298   //rand8
299   b = (((*s1 << 13) ^ *s1) >> 19);
300   *s1 = (((*s1 & 4294967294) << 12) ^ b);
301   b = (((*s2 << 2) ^ *s2) >> 25);
302   *s2 = (((*s2 & 4294967288) << 4) ^ b);
303   b = (((*s3 << 3) ^ *s3) >> 11);
304   *s3 = (((*s3 & 4294967280) << 17) ^ b);
305   *rand8 = (float) ((*s1 ^ *s2 ^ *s3) * 2.3283064365e-10f);
306 }
307
308 /// ////////////////////////////////////////////////////////////////////////
309 //! MCML GPU Core
310 ////////////////////////////////////////////////////////////////////////////
311 static __global__ void
```

```
312 MCMLKernel(unsigned int AVE_ITER_PERPHOTON, unsigned long long int * A, unsigned int
        total_photon, float WINIT,
313         unsigned int s1_base, unsigned int s2_base, unsigned int s3_base, unsigned int *
                total_simulated
314 #ifdef COUNT_OVERFLOW
315         , unsigned int *overflow_count
316 #endif
317         )
318 {
319 #ifdef COUNT_OVERFLOW
320   unsigned int photon_overflow_count=0;
321 #endif
322
323 #ifdef REG_ONLY
324   unsigned int sum=0;
325 #endif
326
327   unsigned int photon_count=0;
328   unsigned int last_addr=0, last_val=0;
329   unsigned int last_ir=0, last_iz=0; //further optimization
330   __shared__ unsigned int A_shared [MAX_IR*MAX_IZ];
331
332   // overflow handling: store a 0/1 flag for overflow in the elements
333   // each thread is responsible for
334   __shared__ unsigned int tmp_data[NTHREAD];
335
336   //Reset to zero just in case (for sorting later)
337   for (int i=0; i<MAX_IR*MAX_IZ/NTHREAD; i++)
338     A_shared[threadIdx.x+i*NTHREAD]=0;
339
340   //------------------------------------------------------
341   //! Registers
342   //>>>>>>>>>> Photon Structure per thread
343   float photon_x, photon_y ,photon_z; /* Cartesian coordinates.[cm] */
344   float photon_ux, photon_uy, photon_uz;/* directional cosines of a photon. */
345   //unsigned int photon_w; /* weight. */
346   float photon_w;
347
348   unsigned int photon_dead; /* 1 if photon is terminated. */
349   unsigned int photon_layer; /* index to layer where the photon packet resides. */
350   float photon_s; /* current step size. [cm]. */
```

```
351   float photon_sleft; /* step size left. dimensionless [-]. */
352   unsigned int photon_hit;
353
354   // Random number array per thread
355   float rand0, rand3, rand5, rand7, rand8;
356
357   //------------------------------------------------------
358   //! Set different rand seed for each thread based on id
359   unsigned int s1, s2, s3;
360   s1=s1_base + blockIdx.x*NTHREAD*1000000 + threadIdx.x*1000000;
361   s2=s2_base + blockIdx.x*NTHREAD*1000000 + threadIdx.x*1000000;
362   s3=s3_base + blockIdx.x*NTHREAD*1000000 + threadIdx.x*1000000;
363
364   const float INITWEIGHT=WINIT;
365
366   //------------------------------------------------------
367   //! Simulate photon group (total_photon/NTHREAD) per thread
368   unsigned int NITER=total_photon/(NTHREAD*NBLOCK) * AVE_ITER_PERPHOTON;
369
370   Launch (&photon_w, INITWEIGHT, &photon_dead, &photon_layer,
371     &photon_s, &photon_sleft,
372     &photon_x, &photon_y, &photon_z,
373     &photon_ux, &photon_ux, &photon_uz);
374
375   for (int iIndex = 0; iIndex < NITER; ++iIndex) {
376     tmp_data[threadIdx.x] = 0;  //reset overflow flags
377
378     Rand(&s1, &s2, &s3,
379       &rand0, &rand3,
380       &rand5, &rand7, &rand8);
381
382     //>>>>>>>>>> StepSizeInTissue()
383     ComputeStepSize ( photon_layer, rand0,
384       &photon_s, &photon_sleft);
385
386     //>>>>>>>>>> HitBoundary()
387     photon_hit = HitBoundary (photon_layer, d_layerspecs[photon_layer].z0, d_layerspecs[
          photon_layer].z1, photon_z, photon_uz,
388       &photon_s, &photon_sleft);
389
390     Hop ( photon_s, photon_ux, photon_uy, photon_uz,
```

```
391       &photon_x, &photon_y, &photon_z);
392
393     if(photon_hit) {
394       FastReflectTransmit (rand3, &photon_ux, &photon_uy, &photon_uz, &photon_layer, &
              photon_dead);
395     }
396     else {
397
398       //>>>>>>>>> Drop()
399       unsigned int hit_edge=0;
400
401       unsigned int iz =__fdividef(photon_z , d_In_Ptr.dz);
402       if (iz>=d_In_Ptr.nz) {
403         hit_edge=1;
404         iz=d_In_Ptr.nz-1;
405       }
406
407       unsigned int ir =__fdividef( sqrtf(photon_x * photon_x + photon_y * photon_y) ,
              d_In_Ptr.dr);
408       if (ir>=d_In_Ptr.nr){
409         hit_edge=1;
410         ir=d_In_Ptr.nr-1;
411       }
412
413 #ifdef PRECOMPUTE
414       float dwa = photon_w * d_layerspecs[photon_layer].mua_muas;
415 #else
416       float mua=d_layerspecs[photon_layer].mua;
417       float mus=d_layerspecs[photon_layer].mus;
418       float dwa = photon_w * mua/(mua+mus);
419 #endif
420
421       photon_w -= dwa;
422
423 #ifdef REG_ONLY
424       sum+=dwaINT;
425 #else
426
427       //only record if photon is not at the edge!!
428       //this will be ignored anyways.
429       if (!hit_edge) {
```

```
430           unsigned int curr_addr = (ir * d_In_Ptr.nz + iz);
431           unsigned int curr_val = (unsigned int)(dwa * WEIGHT_SCALE);
432
433           if (curr_addr==last_addr) //coalesce last entry (minor improvements with more
                  history)
434             last_val+=curr_val;
435           else {
436             if (last_ir<MAX_IR && last_iz<MAX_IZ) { //fits inside the shared mem
437               unsigned shared_addr = (last_ir * MAX_IZ + last_iz);
438               //need new addressing scheme for distinct shared mem layout (MAX_IR by
                      MAX_IZ)
439               unsigned oldval=atomicAdd((unsigned int *) &A_shared[shared_addr], (unsigned
                      int) last_val);
440
441               if (oldval>=MAX_OVERFLOW) { // Detects overflow
442                 tmp_data[shared_addr % NTHREAD] = 1;
443 #ifdef COUNT_OVERFLOW
444                 photon_overflow_count++;
445 #endif
446               }
447             }
448             else { //must be written to global mem
449               atomicAdd((unsigned long long int *) &A[last_addr], (unsigned long long int)
                      last_val);
450             }
451
452             //Promote current value as last entry
453             last_ir=ir; last_iz=iz; //new optimization to capture more high light dose
                  elements
454             last_addr=curr_addr;
455             last_val=curr_val;
456           }
457 #endif
458         } //end if (!hit_edge)
459         //>>>>>>>>>> Drop()
460
461         Spin( &d_layerspecs[photon_layer].g, &photon_ux, &photon_uy, &photon_uz,
462           &rand5, &rand7);
463       }
464
465       /*******************************************************
```

```
466       *  >>>>>>>>> Roulette()
467       * The photon weight is small, and the photon packet tries
468       *  to survive a roulette.
469       ****/
470       if( photon_w < WITH && !photon_dead)
471       {
472         if(photon_w == 0.0f)
473           photon_dead = 1;
474         else if(rand8 < CHANCE) /* survived the roulette.*/
475           photon_w *= INVCHANCE;
476         else
477           photon_dead = 1;
478       }
479
480       if (photon_dead) {
481         ++photon_count;
482         Launch (&photon_w, INITWEIGHT, &photon_dead, &photon_layer,
483           &photon_s, &photon_sleft,
484           &photon_x, &photon_y, &photon_z,
485           &photon_ux, &photon_uy, &photon_uz);
486
487       }
488
489 #ifndef REG_ONLY
490       ////////////////////////////////////////////////////
491       //Necessary to handle overflow in 32-bit shared buffer A[r][z]
492       __syncthreads();
493
494       // Enter a phase of handling VIP range overflow.
495       if (tmp_data[threadIdx.x])
496       {
497         // Flush all elements I am responsible for to the global memory.
498         for (unsigned int i = threadIdx.x; i < MAX_IR*MAX_IZ; i += NTHREAD)
499         {
500           unsigned int ir = i / MAX_IZ;
501           unsigned int iz = i - ir * MAX_IZ;
502           unsigned glob_addr = ir * d_In_Ptr.nz + iz;
503
504           atomicAdd((unsigned long long int *) &A[glob_addr], (unsigned long long int) (((
                 unsigned int*)A_shared)[i]) );
505           ((unsigned int*)A_shared)[i]=0;
```

```
506
507         }
508       }
509
510       __syncthreads();
511 #endif
512       ////////////////////////////////////////////////////
513
514     } // end of big for loop
515
516 #ifdef REG_ONLY
517     A[0]+=sum;
518 #endif
519
520     atomicAdd(total_simulated, photon_count);
521
522 #ifdef COUNT_OVERFLOW
523     atomicAdd(overflow_count, (unsigned int)photon_overflow_count);
524 #endif
525
526 #ifndef REG_ONLY
527     ////////////////////////////////////////////////////////////////
528     //Flush content from N-multiprocessor shared mem into global memory
529     __syncthreads();
530     if (threadIdx.x==0) { //nominate the first thread in each block to do the job
531       for (int ir =0; ir<MAX_IR; ir++ ) {
532         for (int iz =0; iz<MAX_IZ; iz++ ) {
533           unsigned int glob_addr = (ir * d_In_Ptr.nz + iz);
534           unsigned int shared_addr = (ir * MAX_IZ + iz);
535
536           atomicAdd((unsigned long long int *) &A[glob_addr], (unsigned long long int)
                A_shared[shared_addr]);
537         }
538       }
539     }
540 #endif
541
542 }
543
544 #endif // #ifndef _MCMLCUDA_KERNEL_H_
```

Listing B.2: MCMLcuda_kernel.cu (Kernel code or GPU program)

```
1  /*****************************************************
2  *  CUDA GPU version of MCML
3  *  Main program for Monte Carlo simulation of photon
4  * propagation in multi-layered turbid media.
5  ****/
6
7  //!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
8  // NOTE: Set #define WINXP 1 when Visual Studio is used to compile
9  //!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
10 //#define WINXP 1
11
12 // These includes are ONLY needed in Visual Studio 2005
13 // They are not required in Linux
14 #ifdef WINXP
15 #include "MCMLsrc/MCMLNR.C"
16 #include "MCMLsrc/MCMLIO.C"
17 #include "MCMLsrc/MCMLGO.C"
18 #endif
19
20 // includes, system
21 #include <stdlib.h>
22 #include <stdio.h>
23 #include <string.h>
24 #include <math.h>
25
26 // includes, project
27 #include <cutil.h>
28 #include <cuda_runtime.h>
29
30 #ifdef WINXP
31 #include "multithreading_winxp/multithreading.h"
32 #else
33 #include <multithreading.h>
34 #endif
35
36 // includes, kernels
37 #include "MCMLcuda_kernel.cu"
38 #include "MCMLcuda_kernel.h"
```

```
39
40 #include "MCML_src/MCML.H"
41
42 ////////////////////////////////////////////////////////////////////////////
43 // GLOBAL VARIABLES
44 unsigned int AVE_ITER_PERPHOTON; //to be determined by test run
45
46 unsigned int total_nphotons [MAX_GPU_COUNT];
47
48 // Multi-GPU
49 GPUinout GPUinfo[MAX_GPU_COUNT];
50 int GPU_N;
51
52 // Input (avoid passing)
53 InputStruct in_parm;
54
55 // Output
56 float Rsp;  /* specular reflectance. [-] */
57 double ** finalA_rz; /* 2D probability density in turbid */
58 unsigned int NITER;
59 double total_simulated=0.0;
60
61 //Simulation total time
62 unsigned int timer = 0;
63
64 ////////////////////////////////////////////////////////////////////////////
65 // From MCMLIO.C and MCMLGO.C
66 FILE *GetFile(char *);
67 short ReadNumRuns(FILE* );
68 void ReadParm(FILE* , InputStruct * );
69 void CheckParm(FILE* , InputStruct * );
70 float Rspecular(LayerStruct * );
71
72 ////////////////////////////////////////////////////////////////////////////
73 //! Report time and write results.
74 ////////////////////////////////////////////////////////////////////////////
75 void ReportResult()
76 {
77   FILE *file;
78
79   double dz = (double)in_parm.dz;
```

```
80    double dr = (double)in_parm.dr;
81    //short nl = in_parm.num_layers;
82    short iz,ir;
83    //short il;
84    double scale1;
85
86    /* Scale A_rz. */
87    scale1 = 2.0*PI*dr*dr*dz*(total_simulated)*WEIGHT_SCALE;
88    /* volume is 2*pi*(ir+0.5)*dr*dr*dz.*/
89    /* ir+0.5 to be added. */
90    for(iz=0; iz<in_parm.nz; iz++)
91      for(ir=0; ir<in_parm.nr; ir++)
92        finalA_rz[ir][iz] /= (ir+0.5)*scale1;
93
94    file = fopen(in_parm.out_fname, "w");
95    if(file == NULL) nrerror("Cannot open file to write.\n");
96
97    for(ir=0;ir<in_parm.nr;ir++)
98      for(iz=0;iz<in_parm.nz;iz++)  {
99        fprintf(file, "%12.4E ", finalA_rz[ir][iz]);
100       if( (ir*in_parm.nz + iz + 1)%5 == 0) fprintf(file, "\n");
101     }
102
103     fprintf(file, "\n");
104
105     fclose(file);
106
107 }
108
109 ////////////////////////////////////////////////////////////////////////////
110 //! Get the file name of the input data file from the
111 //! argument to the command line.
112 ////////////////////////////////////////////////////////////////////////////
113 void GetFnameFromArgv(int argc,
114             char * argv[],
115             char * input_filename)
116 {
117   if(argc>=2) {      /* filename in command line */
118     strcpy(input_filename, argv[1]);
119   }
120   else
```

```
121     input_filename[0] = '\0';
122 }
123
124 static CUT_THREADPROC runGPUi(GPUinout * GPUi) {
125   ////////////////////////////
126   //Multi-GPU
127   ////////////////////////////
128   CUDA_SAFE_CALL (cudaSetDevice(GPUi->deviceid));
129   printf ("========GPU_%d__running========\n", GPUi->deviceid);
130
131   ////////////////////////////////
132   //Set up Constants in constant mem
133   InputStructGPU h_In_Ptr;
134
135   h_In_Ptr.num_photons=in_parm.num_photons;
136
137   h_In_Ptr.dz=in_parm.dz;
138   h_In_Ptr.dr=in_parm.dr;
139   h_In_Ptr.nz=in_parm.nz;
140   h_In_Ptr.nr=in_parm.nr;
141
142   h_In_Ptr.num_layers=in_parm.num_layers;
143
144 #ifdef PRECOMPUTE
145   /* Cache some precomputed result for each layer spec. */
146   LayerStructGPU_precomp layerspecs_p[MAXLAYER];
147
148   for (int i = 0; i < in_parm.num_layers+2; ++i) {
149     layerspecs_p[i].z0 = in_parm.layerspecs[i].z0;
150     layerspecs_p[i].z1 = in_parm.layerspecs[i].z1;
151     layerspecs_p[i].n = in_parm.layerspecs[i].n;
152     layerspecs_p[i].g = in_parm.layerspecs[i].g;
153     layerspecs_p[i].cos_crit0 = in_parm.layerspecs[i].cos_crit0;
154     layerspecs_p[i].cos_crit1 = in_parm.layerspecs[i].cos_crit1;
155
156     float muas = in_parm.layerspecs[i].mua + in_parm.layerspecs[i].mus;
157     layerspecs_p[i].muas = muas;
158     layerspecs_p[i].is_glass = (muas == 0.0F) ? 1 : 0;
159     if (layerspecs_p[i].is_glass == 0) {
160       layerspecs_p[i].rmuas = 1.0F/muas;
161       layerspecs_p[i].mua_muas = in_parm.layerspecs[i].mua / muas;
```

```
162       }
163     }
164
165     CUDA_SAFE_CALL( cudaMemcpyToSymbol(d_layerspecs, layerspecs_p,
166       sizeof(LayerStructGPU_precomp)*MAXLAYER ) );
167 #else
168     //Copy in Layer geometry
169     CUDA_SAFE_CALL( cudaMemcpyToSymbol(d_layerspecs, in_parm.layerspecs,
170       sizeof(LayerStruct)*MAXLAYER ) );
171 #endif
172
173     //Copy in Input specifications (most importantly num_layers, dr, dz, Wth)
174     CUDA_SAFE_CALL( cudaMemcpyToSymbol(d_In_Ptr, &h_In_Ptr, sizeof(InputStructGPU) ) );
175
176     unsigned int mem_size2=sizeof(unsigned long long int)*in_parm.nr*in_parm.nz;
177
178     ////////////////////////////////////
179     //allocate 2D absorption array on host memory
180     unsigned long long int* h_A = (unsigned long long int*) malloc(mem_size2);
181     //Temp array for mem copy
182     for (int i=0; i<in_parm.nr*in_parm.nz; i++)
183       h_A[i]=0;
184
185     unsigned long long int * h_A_rz=(unsigned long long int *)malloc (mem_size2);
186
187     ////////////////////////////////////
188     // allocate device memory for result
189     unsigned long long int *d_A;
190     CUDA_SAFE_CALL( cudaMalloc( (void**) &d_A, mem_size2));
191
192     ////////////////////////////////////
193     // Init device mem to 0 by copying
194     CUDA_SAFE_CALL( cudaMemcpy( d_A, h_A, mem_size2,
195       cudaMemcpyHostToDevice) );
196
197     /* Allocate a global variable that stores the total # of photons
198     * processed.
199     */
200     unsigned int *g_total_nphotons;
201     cudaMalloc((void**) &g_total_nphotons, sizeof(unsigned int));
202     cudaMemset(g_total_nphotons, 0, sizeof(unsigned int));
```

```
203
204 #ifdef COUNT_OVERFLOW
205   // Count the number of times the VIP ranges in smem overflow.
206   unsigned int *g_overflow_count;
207   cudaMalloc((void**)&g_overflow_count, sizeof(unsigned int));
208   cudaMemset(g_overflow_count, 0, sizeof(unsigned int));
209 #endif
210
211   ////////////////////////////////
212   // setup execution parameters
213   dim3  grid( NBLOCK, 1, 1);
214   dim3  threads( NTHREAD, 1, 1);
215
216   ////////////////////////////////
217   // execute the kernel
218
219   //Simulation MCMLGPU kernel exec time
220   unsigned int execTimer = 0;
221   CUT_SAFE_CALL( cutCreateTimer( &execTimer));
222   CUT_SAFE_CALL( cutStartTimer(execTimer));
223
224   float WINIT=(1.0F - Rsp);  //*WEIGHT_SCALE;
225
226   MCMLKernel<<< grid, threads>>>( AVE_ITER_PERPHOTON, d_A, GPUi->num_photons, WINIT,
227     GPUi->s1_base, GPUi->s2_base, GPUi->s3_base, g_total_nphotons
228 #ifdef COUNT_OVERFLOW
229     , g_overflow_count
230 #endif
231     );
232
233   CUT_CHECK_ERROR("!!!!!ERROR!!!!! ---> Kernel execution failed");
234   cudaThreadSynchronize();
235
236   CUT_SAFE_CALL( cutStopTimer(execTimer));
237
238   /* Copy back the photon count. */
239   cudaMemcpy(&total_nphotons[GPUi->deviceid], g_total_nphotons, sizeof(unsigned int),
240     cudaMemcpyDeviceToHost);
241
242 #ifdef COUNT_OVERFLOW
243   // Copy back the overflow count.
```

```
244   unsigned int overflow_count = 0;
245   cudaMemcpy(&overflow_count, g_overflow_count, sizeof(unsigned int),
246     cudaMemcpyDeviceToHost);
247   printf("!!!!!!!!!!! VIP overflow count = %u\n", overflow_count);
248 #endif
249
250   printf( "\n\n>>>>>>GPU Kernel Exec time: %f (ms)\n", cutGetTimerValue( execTimer));
251   printf ("||||||||||Num of photons actually simulated per GPU = %u \n", total_nphotons[
        GPUi->deviceid]);
252   printf ("||||||||||GPU exec time per million photons= %lf sec\n", cutGetTimerValue(
        execTimer)/(double)total_nphotons[GPUi->deviceid] * 1000000 /1000);
253
254
255   // copy result from device to host
256   CUDA_SAFE_CALL( cudaMemcpy( h_A_rz, d_A, mem_size2,
257     cudaMemcpyDeviceToHost) );
258
259   ////////////////////////////
260   //Multi-GPU
261   ////////////////////////////
262   for (int ir = 0; ir < in_parm.nr; ir++)
263     for (int iz = 0; iz < in_parm.nz; iz++)
264       GPUi->A_rz[ir][iz] = (double) h_A_rz[ir * in_parm.nz + iz];
265
266   ////////////////////////////////
267   // cleanup memory
268   free(h_A);
269   CUDA_SAFE_CALL(cudaFree(d_A));
270   CUT_SAFE_CALL( cutDeleteTimer( execTimer));
271
272   CUT_THREADEND;
273 }
274
275 void sum() {
276   for (int i = 0; i < GPU_N; i++)
277     for (int ir = 0; ir < in_parm.nr; ir++)
278       for (int iz = 0; iz < in_parm.nz; iz++)
279         finalA_rz[ir][iz] += GPUinfo[i].A_rz[ir][iz];
280 }
281
282 ////////////////////////////////////////////////////////////////////////////
```

```
283 //! Set up and Launch GPU Kernel for MCML simulation
284 ////////////////////////////////////////////////////////////////////////////
285 void
286 runMCMLmultiGPU()
287 {
288   ////////////////////////////
289   //Multi-GPU
290   ////////////////////////////
291   CUTThread tid [MAX_GPU_COUNT];
292
293   //Calculate how many photons each GPU needs to launch
294   NITER=in_parm.num_photons/(NTHREAD*NBLOCK);
295   total_simulated=NITER*NTHREAD*NBLOCK;
296
297   for (int i=0; i<GPU_N; i++) {
298     GPUinfo[i].A_rz = AllocDoubleMatrix(0,in_parm.nr-1,0,in_parm.nz-1);
299     GPUinfo[i].deviceid=i;
300     GPUinfo[i].num_photons=(unsigned int)total_simulated/GPU_N;
301     GPUinfo[i].s1_base=1113244+i;
302     GPUinfo[i].s2_base=4712433+i;
303     GPUinfo[i].s3_base=7437331+i;
304
305     tid[i]=cutStartThread((CUT_THREADROUTINE) runGPUi, &GPUinfo[i]);
306   }
307
308
309   printf ("========== waiting for sim threads to complete ========\n");
310   cutWaitForThreads(tid, GPU_N);
311
312   total_simulated=0;
313   for (int i=0; i<GPU_N; i++)
314     total_simulated+=(double)total_nphotons[i];
315
316   sum();
317 }
318
319 ////////////////////////////////////////////////////////////////////////////
320 //! Small test run to estimate the number of iterations required per photon
321 ////////////////////////////////////////////////////////////////////////////
322 unsigned int
323 runtestMCMLmultiGPU()
```

```
324 {
325   ////////////////////////////
326   //Multi-GPU
327   ////////////////////////////
328   CUTThread tid [MAX_GPU_COUNT];
329
330   //Calculate how many photons each GPU needs to launch
331   NITER=300;
332   total_simulated=NITER*NTHREAD*NBLOCK;
333   printf (">>>>>>>>>>>>NITER = %u \n", NITER);
334   for (int i=0; i<GPU_N; i++) {
335     GPUinfo[i].A_rz = AllocDoubleMatrix(0,in_parm.nr-1,0,in_parm.nz-1);
336     GPUinfo[i].deviceid=i;
337     GPUinfo[i].num_photons=(unsigned int)total_simulated/GPU_N;
338     GPUinfo[i].s1_base=1113244+i;
339     GPUinfo[i].s2_base=4712433+i;
340     GPUinfo[i].s3_base=7437331+i;
341
342     tid[i]=cutStartThread((CUT_THREADROUTINE) runGPUi, &GPUinfo[i]);
343   }
344
345
346   printf ("========= waiting for sim threads to complete========\n");
347   cutWaitForThreads(tid, GPU_N);
348
349   total_simulated=0;
350   for (int i=0; i<GPU_N; i++)
351     total_simulated+=(double)total_nphotons[i];
352   printf ("total_simulated = %lf \n", total_simulated);
353
354   return (unsigned int) AVE_ITER_PERPHOTON*NITER*NTHREAD*NBLOCK/(unsigned int)
        total_simulated;
355
356 }
357
358 ////////////////////////////////////////////////////////////////////////////
359 //! Execute Monte Carlo simulation for one independent run.
360 ////////////////////////////////////////////////////////////////////////////
361 void DoOneRun(short NumRuns)
362 {
363   Rsp = Rspecular(in_parm.layerspecs);
```

```
364
365   AVE_ITER_PERPHOTON=200; //initialization
366   printf ("\n\n\n---------- RUNNING TEST RUN ----------\n\n\n");
367   AVE_ITER_PERPHOTON=runtestMCMLmultiGPU();
368
369   for (int i=0; i<GPU_N; i++) {
370     FreeDoubleMatrix(GPUinfo[i].A_rz, 0, in_parm.nr-1, 0,in_parm.nz-1);
371   }
372
373   printf ("Estimated AVE_ITER_PERPHOTON = %u", AVE_ITER_PERPHOTON);
374
375   printf ("\n\n\n---------- RUNNING FULL RUN ----------\n");
376   printf ("---------- Input File: %u photons ----------\n\n\n", in_parm.num_photons);
377
378   printf ("========Multi-GPU host threads starting========\n");
379   runMCMLmultiGPU();
380   printf ("========Multi-GPU host threads finished GPU simulation========\n");
381
382   ReportResult();
383   free(in_parm.layerspecs);
384 }
385
386 ////////////////////////////////////////////////////////////////////////////
387 //! Program main
388 ////////////////////////////////////////////////////////////////////////////
389 int
390 main( int argc, char** argv)
391 {
392   ///////////////////////////////////
393   //Start timer
394   CUT_SAFE_CALL( cutCreateTimer( &timer));
395   CUT_SAFE_CALL( cutStartTimer( timer));
396
397   ///////////////////////////////////
398   //MCML front-end (parsing file input)
399   char input_filename[STRLEN];
400   FILE *input_file_ptr;
401   short num_runs; /* number of independent runs. */
402
403   CUDA_SAFE_CALL (cudaGetDeviceCount(&GPU_N));
404
```

```
405   printf ("========Number of GPUs available: %d \n", GPU_N);
406
407   if (argc==3) { //Last argument for GPU_N
408     if (atoi(argv[2])<=GPU_N)
409       GPU_N=atoi(argv[2]);
410     else
411       printf ("NOTE: You specified more than the max # of gpus available.  \nDefaulting
              to max available=%u \n", GPU_N);
412     //else use the max number of GPU_N available
413   }
414   else {
415     printf ("NOTE: You did not specify the number of gpus to be used.  \nDefaulting to 1
            GPU \n");
416     GPU_N=1;
417   }
418
419   printf ("========Number of GPUs used: %d \n\n", GPU_N);
420
421   printf ("\\\\\\\\GPU_MCML////////\n");
422   printf ("\\\\\\\\%d blocks////////\n", NBLOCK);
423   printf ("\\\\\\\\%d threads////////\n", NTHREAD);
424
425   GetFnameFromArgv(argc, argv, input_filename);
426   input_file_ptr = GetFile(input_filename);
427   CheckParm(input_file_ptr, &in_parm);
428   num_runs = ReadNumRuns(input_file_ptr);
429
430
431   while(num_runs--) {
432     ReadParm(input_file_ptr, &in_parm);
433
434     //Error check - limited to MAXLAYER by const memory (8kB) size
435     if (in_parm.num_layers+2>MAXLAYER) {
436       printf ("ERROR: Total number of layers + 2 ambient layers >"
437         "MAXLAYER (%u layers specified in MCMLcuda_kernel.h) \n", MAXLAYER);
438       exit(0);
439     }
440
441     finalA_rz=AllocDoubleMatrix(0,in_parm.nr-1,0,in_parm.nz-1);
442
443     DoOneRun(num_runs);
```

```
444     }
445
446     ////////////////////////////////////
447     //Halt device and clean up
448     FreeDoubleMatrix(finalA_rz, 0, in_parm.nr-1, 0,in_parm.nz-1);
449     for (int i=0; i<GPU_N; i++) {
450       FreeDoubleMatrix(GPUinfo[i].A_rz, 0, in_parm.nr-1, 0,in_parm.nz-1);
451     }
452     fclose(input_file_ptr);
453
454
455     ////////////////////////////////////
456     // Stop timer and report time/results
457     CUT_SAFE_CALL( cutStopTimer( timer));
458     printf ("\n\n>>>>>>TOTAL number of photons simulated: %u \n", (unsigned int)
            total_simulated);
459     printf( ">>>>>>TOTAL Processing time: %f (ms)\n", cutGetTimerValue( timer));
460     CUT_SAFE_CALL( cutDeleteTimer( timer));
461
462     CUT_EXIT(argc, argv);
463
464     return(0);
465 }
```

Listing B.3: MCMLcuda.cu (Host CPU code)

Bibliography

[1] S. Brown, E. Brown, and I. Walker, "The present and future role of photodynamic therapy in cancer treatment," *Lancet Oncology* **5**(8), pp. 497–508, 2004.

[2] T. Dougherty, "Photodynamic Therapy," *Photochemistry and Photobiology* **58**(6), pp. 895–900, 1993.

[3] T. Dougherty, "An update on photodynamic therapy applications," *Journal of Clinical Laser Medicine & Surgery* **20**(1), pp. 3–7, 2002.

[4] B. Henderson and T. Dougherty, "How does photodynamic therapy work?," *Photochemistry and Photobiology* **55**(1), pp. 145–157, 1992.

[5] P. Access and R. Access, "Verteporfin therapy for subfoveal choroidal neovascularization in age-related macular degeneration: four-year results of an open-label extension of 2 randomized clinical trials: TAP Report No. 7," 2005.

[6] M. Wilson, "Lethal photosensitisation of oral bacteria and its potential application in the photodynamic therapy of oral infections," *Photochemical & Photobiological Sciences* **3**(5), pp. 412–418, 2004.

[7] B. Funke, A. Jungel, S. Schastak, K. Wiedemeyer, F. Emmrich, and U. Sack, "Transdermal photodynamic therapy–a treatment option for rheumatic destruction of small joints?," *Lasers in surgery and medicine* **38**(9), p. 866, 2006.

[8] S. Fien and A. Oseroff, "Photodynamic therapy for non-melanoma skin cancer.," *Journal of the National Comprehensive Cancer Network: JNCCN* **5**(5), p. 531, 2007.

[9] U. Nseyo, J. DeHaven, T. Dougherty, W. Potter, D. Merrill, S. Lundahl, and D. Lamm, "Photodynamic therapy (PDT) in the treatment of patients with resistant superficial bladder cancer: a long term experience," *Journal of Clinical Laser Medicine & Surgery* **16**(1), pp. 61–68, 1998.

[10] J. Usuda, H. Kato, T. Okunaka, K. Furukawa, H. Tsutsui, K. Yamada, Y. Suga, H. Honda, Y. Nagatsuka, T. Ohira, *et al.*, "Photodynamic Therapy (PDT) for Lung Cancers," *Journal of Thoracic Oncology* **1**(5), p. 489, 2006.

[11] B. Overholt, K. Wang, J. Burdick, C. Lightdale, M. Kimmey, H. Nava, M. Sivak, N. Nishioka, H. Barr, N. Marcon, *et al.*, "Five-year efficacy and safety of photodynamic therapy with Photofrin in Barrett's high-grade dysplasia," *Gastrointestinal Endoscopy* **66**(3), pp. 460–468, 2007.

[12] S. Stylli and A. Kaye, "Photodynamic therapy of cerebral glioma–a review Part II–clinical studies," *Journal of Clinical Neuroscience* **13**(7), pp. 709–717, 2006.

[13] J. Trachtenberg, R. Weersink, S. Davidson, M. Haider, A. Bogaards, M. Gertner, A. Evans, A. Scherz, J. Savard, J. Chin, *et al.*, "Vascular-targeted photodynamic therapy (padoporfin, WST09) for recurrent prostate cancer after failure of external beam radiotherapy: a study of escalating light doses," *BJU International* **102**(5), pp. 556–562, 2008.

[14] M. Biel, "Advances in photodynamic therapy for the treatment of head and neck cancers," *Lasers in Surgery and Medicine* **38**(5), 2006.

[15] S. Davidson, R. Weersink, M. Haider, M. Gertner, A. Bogaards, D. Giewercer, A. Scherz, M. Sherar, M. Elhilali, J. Chin, *et al.*, "Treatment planning and dose anal-

ysis for interstitial photodynamic therapy of prostate cancer," *Physics in Medicine and Biology* **54**(8), pp. 2293–2313, 2009.

[16] C. Gomer, M. Luna, A. Ferrario, S. Wong, A. Fisher, and N. Rucker, "Cellular targets and molecular responses associated with photodynamic therapy," *Journal of Clinical Laser Medicine & Surgery* **14**(5), pp. 315–321, 1996.

[17] B. Wilson, M. Patterson, and L. Lilge, "Implicit and explicit dosimetry in photodynamic therapy: a new paradigm," *Lasers in Medical Science* **12**(3), pp. 182–199, 1997.

[18] F. Hetzel, S. Brahmavar, Q. Chen, S. Jacques, M. Patterson, B. Wilson, and T. Zhu, "Photodynamic therapy dosimetry," *AAPM report* **88**, 2005.

[19] M. Dereski, M. Chopp, J. Garcia, and F. Hetzel, "Depth measurements and histopathological characterization of photodynamic therapy generated normal brain necrosis as a function of incident optical energy dose," *Photochemistry and photobiology* **54**(1), pp. 109–112, 1991.

[20] T. Farrell, B. Wilson, M. Patterson, and M. Olivo, "Comparison of the in vivo photodynamic threshold dose for photofrin, mono-and tetrasulfonated aluminum phthalocyanine using a rat liver model," *Photochemistry and photobiology* **68**(3), pp. 394–399, 1998.

[21] A. Johansson, J. Axelsson, S. Andersson-Engels, and J. Swartling, "Realtime light dosimetry software tools for interstitial photodynamic therapy of the human prostate," *Medical Physics* **34**, p. 4309, 2007.

[22] T. Mang, "Dosimetric concepts for PDT," *Photodiagnosis and Photodynamic Therapy* **5**(3), pp. 217–223, 2008.

[23] M. Patterson, B. Wilson, and D. Wyman, "The propagation of optical radiation in tissue I. Models of radiation transport and their application," *Lasers in Medical Science* **6**(2), pp. 155–168, 1991.

[24] S. Jacques and B. Pogue, "Tutorial on diffuse light transport," *Journal of Biomedical Optics* **13**, p. 041302, 2008.

[25] S. Arridge, M. Schweiger, M. Hiraoka, and D. Delpy, "A finite element approach for modeling photon transport in tissue," *Medical Physics* **20**, p. 299, 1993.

[26] E. Heath, J. Seuntjens, and D. Sheikh-Bagheri, "Dosimetric evaluation of the clinical implementation of the first commercial IMRT Monte Carlo treatment planning system at 6 MV," *Medical Physics* **31**, p. 2771, 2004.

[27] C. Ma, E. Mok, A. Kapur, T. Pawlicki, D. Findley, S. Brain, K. Forster, and A. Boyer, "Clinical implementation of a Monte Carlo treatment planning system," *Medical Physics* **26**, p. 2133, 1999.

[28] B. Wilson and G. Adam, "A Monte Carlo model for the absorption and flux distributions of light in tissue," *Medical Physics* **10**, p. 824, 1983.

[29] L. Grossweiner, L. Jones, J. Grossweiner, and B. Rogers, *The science of phototherapy: an introduction*, p.143-145, Kluwer Academic Pub, 2005.

[30] B. Wilson and M. Patterson, "The physics, biophysics and technology of photodynamic therapy," *Physics in Medicine and Biology* **53**(9), p. 61, 2008.

[31] I. Chetty, B. Curran, J. Cygler, J. DeMarco, G. Ezzell, B. Faddegon, I. Kawrakow, P. Keall, H. Liu, C. Ma, *et al.*, "Report of the AAPM Task Group No. 105: Issues associated with clinical implementation of Monte Carlo-based photon and electron external beam treatment planning," *Medical Physics* **34**, p. 4818, 2007.

[32] A. Rendon, *Biological and Physical Strategies to Improve the Therapeutic Index of Photodynamic Therapy*. PhD thesis, University of Toronto, 2008.

[33] "ASAP - Getting Started Guide," *Breault Research Organization* , 2009. `http://www.breault.com/resources/kbasePDF/broman0108_getstart.pdf`.

[34] J. Feyh, R. Gutmann, A. Leunig, L. Jager, M. Reiser, R. Saxton, D. Castro, and E. Kastenbauer, "MRI-guided laser interstitial thermal therapy (LITT) of head and neck tumors: progress with a new method," *Journal of Clinical Laser Medicine & Surgery* **14**(6), pp. 361–366, 1996.

[35] S. Arridge and M. Schweiger, "Image reconstruction in optical tomography," *Philosophical Transactions: Biological Sciences* **352**(1354), pp. 717–726, 1997.

[36] A. Colasanti, G. Guida, A. Kisslinger, R. Liuzzi, M. Quarto, P. Riccio, R. G., and F. Villani, "Multiple Processor Version of a Monte Carlo Code for Photon Transport in Turbid Media," *Computer Physics Communications* **132**, pp. 84–93, 2000.

[37] S. Coyle, K. Thomas, T. Naughton, M. Charles, and W. Toms, "Distributed Monte Carlo Simulation of Light Transportation in Tissue," in *Parallel and Distributed Processing Symposium, 2006. Proceedings. 20th IEEE International Symposium on*, p. 4, 2006.

[38] V. Betz, J. Rose, and A. Marquardt, *Architecture and CAD for deep-submicron FPGAs*, Kluwer Academic Publishers Norwell, MA, USA, 1999.

[39] J. Rose, A. El Gamal, and A. Sangiovanni-Vincentelli, "Architecture of field-programmable gate arrays," *Proceedings of the IEEE* **81**(7), pp. 1013–1029, 1993.

[40] L. Wang, S. Jacques, and L. Zheng, "MCML - Monte Carlo modeling of light transport in multi-layered tissues," *Computer Methods and Programs in Biomedicine* **47**(2), pp. 131–146, 1995.

[41] C. Ma, E. Mok, A. Kapur, T. Pawlicki, D. Findley, S. Brain, K. Forster, and A. Boyer, "Clinical implementation of a Monte Carlo treatment planning system," *Medical physics* **26**, p. 2133, 1999.

[42] A. Pang, A. Smith, P. Nuin, and E. Tillier, "SIMPROT: using an empirically determined indel distribution in simulations of protein evolution," *BMC bioinformatics* **6**(1), p. 236, 2005.

[43] N. Metropolis and S. Ulam, "The monte carlo method," *Journal of the American Statistical Association* , pp. 335–341, 1949.

[44] H. Goldstine and A. Goldstine, "The electronic numerical integrator and computer (ENIAC)," *IEEE Annals of the History of Computing* , pp. 10–16, 1996.

[45] N. Metropolis, "The beginning of the Monte Carlo method," *From Cardinals to Chaos: Reflections on the Life and Legacy of Stanislaw Ulam* , p. 125, 1989.

[46] H. Kahn and A. Marshall, "Methods of reducing sample size in Monte Carlo computations," *Journal of the Operations Research Society of America* , pp. 263–278, 1953.

[47] G. Palmer and N. Ramanujam, "Monte Carlo-based inverse model for calculating tissue optical properties. Part I: Theory and validation on synthetic phantoms," *Applied optics* **45**(5), pp. 1062–1071, 2006.

[48] Q. Liu, C. Zhu, and N. Ramanujam, "Experimental validation of Monte Carlo modeling of fluorescence in tissues in the UV-visible spectrum," *Journal of Biomedical Optics* **8**, pp. 223–236, 2003.

[49] D. Hidovic-Rowe and E. Claridge, "Modelling and validation of spectral reflectance for the colon," *Physics in Medicine and Biology* **50**(6), pp. 1071–1094, 2005.

[50] J. Reuss and D. Siker, "The pulse in reflectance pulse oximetry: modeling and experimental studies," *Journal of clinical monitoring and computing* **18**(4), pp. 289–299, 2004.

[51] D. Boas, J. Culver, J. Stott, and A. Dunn, "Three dimensional Monte Carlo code for photon migration through complex heterogeneous media including the adult human head.," *Optics express* **10**(3), pp. 159–170, 2002.

[52] V. Tuchin, "Light scattering study of tissues," *Physics-Uspekhi* **40**(5), pp. 495–515, 1997.

[53] L. Wang, S. Jacques, and L. Zheng, "CONV - convolution for responses to a finite diameter photon beam incident on multi-layered tissues," *Computer methods and programs in biomedicine* **54**(3), pp. 141–150, 1997.

[54] S. Prahl, M. Keijzer, S. Jacques, and A. Welch, "A Monte Carlo model of light propagation in tissue," *Dosimetry of Laser Radiation in Medicine and Biology* **155**, pp. 102–111.

[55] L. Henyey and J. Greenstein, "Diffuse radiation in the galaxy," in *Annales d'Astrophysique*, **3**, 1940.

[56] M. Born, E. Wolf, and A. Bhatia, *Principles of optics*, Pergamon Press, New York, USA, 1980.

[57] D. Côté and I. Vitkin, "Robust concentration determination of optically active molecules in turbid media with validated three-dimensional polarization sensitive monte carlo calculations," *Opt. Express* **13**(1), pp. 148–163, 2005.

[58] T. Harris and H. Kahn, "Estimation of particle transmission by random sampling," *Natl. Bur. Stand. Appl. Math. Ser* **12**, pp. 27–30, 1951.

[59] J. Hendricks and T. Booth, "MCNP variance reduction overview," *Lect. Notes Phys* **240**, pp. 83–92, 1985.

[60] J. Briesmeister, "MCNP TM–A General Monte Carlo N-Particle Transport Code," *Version 4A, Los Alamos National Laboratory Manual LA-12625-M, Los Alamos National Laboratory, Los Alamos, New Mexico, USA* , 1993.

[61] W. Lo, K. Redmond, J. Luu, P. Chow, J. Rose, and L. Lilge, "Hardware acceleration of a Monte Carlo simulation for photodynamic treatment planning," *Journal of Biomedical Optics* **14**, p. 014019, 2009.

[62] J. Luu, K. Redmond, W. Lo, P. Chow, L. Lilge, and J. Rose, "FPGA-based Monte Carlo Computation of Light Absorption for Photodynamic Cancer Therapy," in *Proceedings of the 7th IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM), Napa Valley, CA*, 2009 (to appear).

[63] I. Kuon, R. Tessier, and J. Rose, "FPGA Architecture: Survey and Challenges," *Foundations and Trends® in Electronic Design Automation* **2**(2), pp. 135–253, 2007.

[64] S. Brown and J. Rose, "FPGA and CPLD architectures: A tutorial," *IEEE Design & Test of Computers* **13**(2), pp. 42–57, 1996.

[65] M. Gokhale, J. Frigo, C. Ahrens, J. Tripp, and R. Minnich, "Monte Carlo Radiative Heat Transfer Simulation on a Reconfigurable Computer," *Lecture notes in computer science* , pp. 95–104, 2004.

[66] K. Whitton, X. Hu, Y. Cedric, and D. Chen, "An FPGA Solution for Radiation Dose Calculation," in *Field-Programmable Custom Computing Machines, 2006. Proceedings. 14th Annual IEEE Symposium on*, pp. 227–236, 2006.

[67] "Handel-C Language Reference Manual," *Agility* . `http://www.agilityds.com/literature/HandelC_Language_Reference_Manual.pdf`.

[68] "ModelSim," *Mentor Graphics* , 2009. http://www.model.com/.

[69] V. Fanti, R. Marzeddu, C. Pili, P. Randaccio, and J. Spiga, "Monte Carlo Computations for Radiotherapy with the use of Dedicated Processors," in *Nuclear Science Symposium Conference Record, 2006. IEEE*, **4**, 2006.

[70] A. S. Pasciak and J. R. Ford, "High-speed Evaluation of Track-structure Monte Carlo Electron Transport Simulations," *Physics in Medicine and Biology* **19**(53), pp. 5539–5553, 2008.

[71] V. Betz and J. Rose, "VPR: A new packing, placement and routing tool for FPGA research," *Lecture Notes in Computer Science* **1304**, pp. 213–222, 1997.

[72] T. Grtker, S. Liao, G. Martin, and S. Swan, *System Design with SystemC*, Springer, 2002.

[73] S. Brown and Z. Vranesic, *Fundamentals of digital logic with Verilog design*, McGraw-Hill Science/Engineering/Math, 2002.

[74] "Introduction to the Quartus II Software," *Altera Corporation* . http://www.altera.com/literature/manual/intro_to_quartus2.pdf.

[75] G. De Micheli, *Synthesis and optimization of digital circuits*, McGraw-Hill Higher Education, 1994.

[76] J. Fender, J. Rose, and D. Galloway, "The transmogrifier-4: an FPGA-based hardware development system with multi-gigabyte memory capacity and high host and memory bandwidth," in *Field-Programmable Technology, 2005. Proceedings. 2005 IEEE International Conference on*, pp. 301–302, 2005.

[77] Terasic Technologies, "Altera DE3 Development and Education Board user handbook," Nov 2008.

[78] W. Ligon III, S. McMillan, G. Monn, K. Schoonover, F. Stivers, and K. Underwood, "A re-evaluation of the practicality of floating-point operations onFPGAs," in *IEEE Symposium on FPGAs for Custom Computing Machines, 1998. Proceedings*, pp. 206–215, 1998.

[79] P. L'Ecuyer, "Maximally Equidistributed Combined Tausworthe Generators," *Mathematics Of Computation* **65**(213), pp. 203–213, 1996.

[80] J. Reuss, O. Inc, and W. Germantown, "Multilayer modeling of reflectance pulse oximetry," *IEEE Transactions on Biomedical Engineering* **52**(2), pp. 153–159, 2005.

[81] R. Stallman *et al.*, *Using GCC: the GNU compiler collection reference manual*, GNU Press, 2003.

[82] "Dual-Core Intel Xeon Processor 5160 Data Sheet," *Intel Corporation* . `http://ark.intel.com/cpu.aspx?groupId=27219`.

[83] "Quartus II 8.1: PowerPlay Power Analysis," *Altera Corporation* . `http://www.altera.com/literature/hb/qts/qts_qii53013.pdf`.

[84] "GPU-accelerated Monte Carlo simulation for photodynamic therapy treatment planning," in *European Conferences on Biomedical Optics (ECBO), Munich, Germany*, SPIE, 2009 (to appear).

[85] "CUDA Programming Guide 2.3," *NVIDIA Corporation* , 2009. `http://developer.download.nvidia.com/compute/cuda/2_3/toolkit/docs/NVIDIA_CUDA_Programming_Guide_2.3.pdf`.

[86] E. Alerstam, T. Svensson, and S. Andersson-Engels, "Parallel computing with graphics processing units for high-speed Monte Carlo simulation of photon migration," *Journal of Biomedical Optics* **13**, p. 060504, 2008.

[87] Q. Peng, K. Berg, J. Moan, M. Kongshaug, and J. Nesland, "5-Aminolevulinic acid-based photodynamic therapy: principles and experimental research," *Photochemistry and photobiology* **65**(2), pp. 235–251, 1997.

[88] G. Lin, M. Tsoukas, M. Lee, S. Gonzalez, C. Vibhagool, R. Anderson, and N. Kollias, "Skin necrosis due to photodynamic action of benzoporphyrin depends on circulating rather than tissue drug levels: implications for control of photodynamic therapy," *Photochemistry and photobiology* **68**(4), pp. 575–583, 1998.

[89] A. Rendon, J. Beck, and L. Lilge, "Treatment planning using tailored and standard cylindrical light diffusers for photodynamic therapy of the prostate," *Physics in Medicine and Biology* **53**(4), p. 1131, 2008.

[90] "Tesla S1070 GPU Computing System," *NVIDIA Corporation* , 2009. `http://www.nvidia.com/docs/IO/43395/SP-04154-001_v02.pdf`.

[91] "Tesla S1070 1U GPU Computing Server," *AMAX* , 2009. `http://www.amax.com/CS_TeslaS10701UGPU.asp`.

[92] A. Kienle, L. Lilge, M. Patterson, R. Hibst, R. Steiner, B. Wilson, *et al.*, "Spatially resolved absolute diffuse reflectance measurements for noninvasive determination of the optical scattering and absorption coefficients of biological tissue," *Applied Optics* **35**(13), pp. 2304–2314, 1996.

[93] M. Nichols, E. Hull, and T. Foster, "Design and testing of a white-light, steady-state diffuse reflectance spectrometer for determination of optical properties of highly scattering systems," *Applied optics* **36**(1), pp. 93–104, 1997.

[94] A. Rendon and L. Lilge, "Towards conformal light delivery using tailored cylindrical diffusers: attainable light dose distributions," *Physics in Medicine and Biology* **51**(23), pp. 5967–5976, 2006.

[95] A. Johansson, N. Bendsoe, K. Svanberg, S. Svanberg, and S. Andersson-Engels, "Influence of treatment-induced changes in tissue absorption on treatment volume during interstitial photodynamic therapy," *Medical Laser Application* **21**(4), pp. 261–270, 2006.