Hardware Acceleration of Biophotonic Simulations

by

Tanner Young-Schultz

A thesis submitted in conformity with the requirements
for the degree of Master of Applied Science
Graduate Department of Electrical and Computer Engineering
University of Toronto

© Copyright 2020 by Tanner Young-Schultz

Abstract

Hardware Acceleration of Biophotonic Simulations

Tanner Young-Schultz
Master of Applied Science
Graduate Department of Electrical and Computer Engineering
University of Toronto
2020

The simulation of light propagation through tissue is important for medical applications like diffuse optical tomography (DOT), bioluminescence imaging (BLI) and photodynamic therapy (PDT). These applications involve solving an inverse problem, which works backwards from a light distribution to the parameters that caused it. These inverse problems have no general closed-form solution and therefore are approximated using iterative techniques. Increasing the accuracy of the approximation requires performing many light propagation simulations which is time-consuming and computationally intensive.

We describe algorithmic techniques to improve the performance, accuracy and usability of the fastest software simulator for forward light propagation, FullMonteSW. Additionally, we explore two acceleration methods using a GPU and an FPGA. Our results show that the GPU and FPGA accelerator improve the performance by 4-13x and 4x, respectively, over the software baseline. We give insight for improving the performance and usability of the GPU- and FPGA-accelerated simulators for various medical applications.

Acknowledgements

To my lab mates, I am extremely grateful for the fun times and support. Over the last two years, I have enjoyed everything from the laughs to the heated (friendly) arguments.

To my supervisors, Vaughan Betz and Stephen Brown. You supported me greatly both personally and professionally for the past two years and helped me grow as a student, professional, and a person. Thank you for everything.

Lastly, to my wonderful parents. Your unconditional love and support has helped me succeed. Through good and bad times, you have always been there for me and for that I thank you.

Contents

1 Introduction **1**

1.1 Motivation . . . 1

1.2 Contributions . . . 2

1.3 Organization of Thesis . . . 2

2 Background **4**

2.1 Tissue Optics . . . 4

2.2 Inverse Problems . . . 5

2.3 Medical Applications of Light . . . 5

2.3.1 Diffuse Optical Tomography (DOT) . . . 5

2.3.2 Bioluminescence Imaging (BLI) . . . 6

2.3.3 Photodynamic Therapy (PDT) . . . 6

2.4 Simulation Models for Light Propagation . . . 8

2.4.1 Geometry Description . . . 8

2.4.2 Light Source Descriptions . . . 10

2.4.3 Output Data . . . 11

2.4.4 Numerical Solutions . . . 12

2.5 Computing Platforms . . . 15

2.5.1 Central Processing Unit (CPU) . . . 15

2.5.2 Graphics Processing Unit (GPU) . . . 16

2.5.3 Field Programmable Gate Array (FPGA) . . . 18

2.6 Previous Implementations . . . 20

3 Software Optimizations **25**

3.1 Representing Medical Light Sources in Simulation . . . 25

3.1.1 Cylindrical Diffuser Light Source . . . 25

3.1.2 Mesh Region Light Source . . . 27

3.2 Improving the Performance of Tetrahedral Mesh Queries . . . 29

3.2.1 Point to Containing Tetrahedron Lookup using RTrees . . . 30

4 Hardware Acceleration **33**

4.1 Motivation . . . 33

4.2 Design Choices . . . 34

4.3 Validation of Output . . . 36

4.4 Packet Launch . . . 36
4.5 Hardware Accelerator Integration . . . 37
4.6 GPU Implementation: FullMonteCUDA . . . 38
4.6.1 Design Overview . . . 38
4.6.2 Results . . . 42
4.7 FPGA Implementation: FullMonteFPGACL . . . 45
4.7.1 Design Overview . . . 45
4.7.2 Results . . . 51
4.7.3 Development Productivity . . . 54

5 Conclusions and Future Work **57**
5.1 Conclusions . . . 57
5.2 Future Work . . . 58
5.2.1 FPGA . . . 58
5.2.2 GPU . . . 60

Bibliography **62**

List of Tables

2.1 NVIDIA GPU Memories [49] 16
2.2 A subset of existing MC light propagation simulators 21

4.1 The different compute architectures used in this work 35
4.2 Models used for the validation and benchmarking of accelerated simulators 37
4.3 Performance increase for each FullMonteCUDA optimization over FullMonteSW using the NVIDIA Titan Xp GPU 40
4.4 Normalized L1-norm values for the models from Table 4.2 using 10^8 packets for two differently seeded CPU simulations (row 1) and a GPU and CPU simulation (row 2) . . . 43
4.5 Performance results for CUDAMCML, MCtet and FullMonteCUDA using the MCML layered models from [70] 43
4.6 Performance results for TIM-OS, MMCM, FullMonteSW and FullMonteCUDA 44
4.7 Performance comparison of FullMonteCUDA against FullMonteSW using 10^8 packets . . . 44
4.8 Performance comparison of FullMonteCUDA against FullMonteSW using 10^6 packets . . . 44
4.9 Normalized L1-norm values across the benchmark models using 10^6 packets for two differently seeded CPU simulations and an FPGA and CPU simulation 51
4.10 Stratix 10 resources for a single FullMonteFPGACL pipeline 52
4.11 Runtimes for various benchmarks on a CPU, GPU and a single and duplicated FPGA pipeline 53
4.12 Estimating the time for FullMonteSW hardware acceleration implementations to be prototyped and to outperform the software by a single developer 55

5.1 FPGA precisions for various data determined by software emulation [18] 58

List of Figures

2.1 High-Resolution Diffuse Optical Tomography (HR-DOT) setup [28] 6
2.2 From left to right: Reconstructed BLI image, CT scan, PET scan and dissection photograph of mouse with a tumor (reproduced from [44]) 7
2.3 2D slice of a curved surface (left) modelled using voxels (middle) and tetrahedrons (right) - the arrows represent the estimated surface normal $\hat{\boldsymbol{n}}$ 9
2.4 Layered geometry with normal-incident pencil beam light source p 9
2.5 Example light sources with emission profiles [2] 10
2.6 The refined hop-drop-spin algorithm used in this work 13
2.7 A high-level depiction of the GPU streaming multiprocessor (left) and scalar processor (right) 17
2.8 The general flow of a CUDA program. Squares states are run on the CPU and the purple oval is run on the GPU. 18

3.1 Planar theta angle distribution options for surface light sources 26
3.2 Generating a random vector ($\hat{\boldsymbol{d}}$) in a hemisphere centered on the z-axis (left) and rotating this vector to be centered on a plane's (F) normal (middle and right). 26
3.3 The side (left) and top (right) view of a virtual cylinder source with surface normal emission (top) and Lambertian emission (bottom) 28
3.4 Illustration of a mesh region light source with the mesh on the left, the isolated surface light source in the middle and the isolated volume light source on the right 29
3.5 A mesh region light source with the regions (left) and the emission profile using a surface normal distribution (right) 30
3.6 Plane-to-point distance calculation for a point inside (left) and outside (right) of a triangle. 31
3.7 2D object partition (left) and corresponding RTree (right) 31

4.1 Integration of FullMonteCUDA and FullMonteFPGACL into existing software project. . . 38
4.2 The host code of FullMonteCUDA 39
4.3 NVVP *PC Sampling* data for the final GPU implementation using the Titan Xp NVIDIA GPU [67] 40
4.4 The number of execution and memory dependency stalls reported by NVVP before the constant material cache, before the accumulation cache and the final implementation with both the constant cache and the 1-entry accumulation cache [67] 42
4.5 Output tetrahedral fluence plots of the *cube_5med* model (Table 4.2) for FullMonteSW (a) and FullMonteCUDA (b) using 10^8 packets. 43

4.6 The core *hop-drop-spin* algorithm for FullMonteFPGACL 45
4.7 Summary of the FullMonteFPGACL system. Large arrows represent CPU-BOARD and FPGA-BOARD memory transfers, dotted arrows represent FPGA-CPU signals and circles represent OpenCL kernels with the arrows between them representing unidirectional channels. 46
4.8 Using global memory (top) and pipes/channels (bottom) for inter-kernel communication . 47
4.9 Output tetrahedral fluence plots of the *cube_5med* model (Table 4.2) for FullMonteSW (a) and FullMonteFPGACL (b) using 10^8 packets. 52
4.10 The power draw of Intel CPUs and FPGAs (Stratix family) across process generations [3]. 54

5.1 Proposed memory structure for duplicating FullMonteCL pipelines [68] 60

Chapter 1

Introduction

1.1 Motivation

The use of light in medical applications has increased in prominence due to its safety for patients, generally low cost and relatively simple monitoring options. Light can be directed using external light sources or fibre optic probes that are inserted into the patient. In medicine, light has therapeutic, diagnostic and imaging use cases, some of which will be discussed in this thesis to provide context and motivation for the work.

Light is used in diffuse optical tomography (DOT) and bioluminescence imaging (BLI) to diagnose and image diseased tissues. In DOT [13], external light of a particular wavelength is used to illuminate the tissue in question. The light travels through the tissue and is partially transmitted, scattered and reflected. The light detected at the exterior surface can be used to infer an image of the underlying tissue. Many light simulations with different tissue optical properties are performed to determine the underlying tissue structure that gives the best match for the detected light. In BLI [47], cells are selectively transfected with a virus that alters the genes of the cell and causes it to emit light. The light emitted at the exterior surface can be quantified, but many simulations are required to determine the size and location of the collection of light-emitting cells based on the detected light distribution. This method is currently being used in a laboratory setting to track the size and location of cancerous tissues in pre-clinical treatment studies [47].

Photodynamic therapy is a targeted and minimally-invasive treatment that can selectively kill diseased cells, such as cancer cells and bacterial infections. The patient is given a non-harmful photosensitizer (PS) either topically, orally or by injection that accumulates preferentially in highly proliferating tissues, like tumours. Both the light and the inactive PS are safe for the patient on their own, but when photons interact with the PS, the PS activates and causes surrounding oxygen to become reactive. Reactive oxygen damages the PS containing cells and, after sufficient damage is accumulated, cause tissue apoptosis or necrosis. The overall goal of PDT is to cause enough damage to a specific tissue, typically a tumour or bacterial infection, to destroy it while minimizing the damage to the surrounding healthy tissue [63].

Given information about the light sources and the geometry and optical properties of the tissues, determining the propagation of light through the tissues is called a *forward* problem. DOT, BLI and PDT treatment planning all require solving what is called an *inverse* problem. That is, given a measured

or desired distribution of light, what parameters would create it. As discussed later, none of these inverse problems have known closed-form solutions and therefore must be approximated using iterative methods. The accuracy of the approximation is proportional to the number of forward light propagation simulations that can be completed in an acceptable time. The need for inverse solvers to run many forward light propagation simulations motivates the development of an accurate, fast and flexible light propagation simulator.

As will be discussed later in Section 2, there are various techniques for simulating the propagation of light through tissue, each with different complexity-accuracy tradeoffs. For our simulator, we choose to use a Monte Carlo light propagation technique with a tetrahedral mesh model. This allows our simulator to be highly accurate at the expense of computational complexity. The need for a fast light propagation simulator motivates the user of hardware acceleration.

1.2 Contributions

In this work, we enhance the fastest published tetrahedral-mesh Monte Carlo software simulator, FullMonteSW [21]. Based on input from medical professionals, we develop two new light source models that allow the simulator to be more usable and accurate for BLI and PDT. We also improve the performance of the simulator using various advanced algorithmic and data structure optimizations. We implement FullMonteSW on an NVIDIA GPU to achieve a 4-13x speedup and produce the fastest GPU simulator of its kind. We also create the first complete and verified FPGA-accelerated implementation of a tetrahedral-mesh Monte Carlo biophotonic simulator, and achieve up to a 4x speed improvement and 11x energy-efficiency improvement over the highly optimized FullMonteSW, while using only a fraction of the FPGA resources. The principle contributions of this thesis can be summarized as follows:

- The implementation of two new lights sources to allow FullMonte to more accurately represent real clinical scenarios in BLI and PDT treatment planning
- A performance and usability improvement of the best-in-class software simulator via algorithm and data structure enhancements
- The creation of the fastest and most flexible GPU-based tetrahedral-mesh Monte Carlo biophotonic simulator
- The creation of the first complete and verified FPGA-based tetrahedral-mesh Monte Carlo biophotonic simulator

1.3 Organization of Thesis

The remainder of this thesis is structured as follows. Chapter 2 provides a more thorough review of the relevant medical applications, the physics of how light interacts with materials (particularly with bodily tissues), a brief introduction to the compute resources we used in this work and the current state of the art in relevant simulators. Chapter 3 discusses our implementation of the two new light source models and their benefit to the accuracy and usability of the simulator for medical applications like BLI and PDT. Chapter 3 also discusses a significant performance improvement we made to the FullMonteSW simulator using a spatial sorting data structure. Chapter 4 is broken into two major sections that

describe how the FullMonteSW algorithm is accelerated using a GPU and FPGA. Finally, Chapter 5 concludes the work and provides suggestions for future work to extend and improve the performance of both the GPU- and FPGA-accelerated simulators.

Chapter 2

Background

This chapter provides relevant information for the research presented in the remainder of this thesis. We begin with a discussion on tissue optics and give a high-level description of how light interacts with tissues. Next, we define biophotonic *inverse* problems, why they are computationally difficult to solve and how to approach approximating their solutions using many iterations of a *forward* problem, which is solved by the simulators developed in our work. Next, we discuss a few diagnostic, imaging and therapeutic applications of light. We introduce the two main models for solving light propagation: finite element with diffusion theory approximation and Monte Carlo simulation. We justify our decision to use a Monte Carlo based model and provide a more formal definition of the forward Monte Carlo light propagation problem solved by our simulator. We discuss how the 3D geometry is discretized for simulation and how we model actual medical light emitters in simulation. This chapter ends with a brief introduction to the different computing technologies used in our work and a summary of the relevant state-of-the-art light propagation simulators.

2.1 Tissue Optics

This section provides a brief introduction to the behavior of light in tissues or tissue-like media within the medical *optical window* (roughly between 630nm-1060nm). Jacques [35] gives a more detailed discussion of tissue optics and experimental values for different tissue types at various wavelengths.

Tissues have several attributes that determine how light propagates through them. The *absorption coefficient* (μ_a) of a material measures the fraction of light (of a particular wavelength) that is absorbed per unit length travelled. Tissues are also *turbid*, meaning they scatter light. This is represented by a *scattering coefficient* (μ_s) and an *anisotropy factor* (g). The scattering coefficient of a material represents the average number of times a photon will scatter per unit length travelled, while the anisotropy factor represents the typical amount of forward direction retained after a scattering event. Materials with a g value of 0 are isotropic, meaning they scatter light equally in all directions. Additionally, when light is travelling through heterogeneous media, photons may cross an interface between two materials (tissues with different refractive indices, n) and the physics involving internal reflection, Fresnel reflection and refraction are considered.

The *attenuation coefficient* (or the *transport Mean Free Path*) (μ_t) of a material is defined as the probability of a scattering **or** absorption event per unit length travelled. Since the probability of scat-

tering and absorption are both exponentially distributed independent variables, we can compute the attenuation coefficient by summing them ($\mu_t = \mu_s + \mu_a$). In addition, the *albedo* (α) of a material is the probability that an event (absorption or scattering) is a scattering event ($\alpha = \frac{\mu_s}{\mu_s+\mu_a}$). A material with a high albedo has a much larger scattering coefficient than absorption coefficient ($\mu_s >> \mu_a$) and therefore photons passing through it scatter many times before they are absorbed. In the optical window we consider, scattering typically occurs 10-100x more often than absorption.

2.2 Inverse Problems

In the context of light propagation simulations, there are two main problems one wishes to solve. The first is called the *forward* problem. Here, the parameters of the simulation are given and the simulator determines the distribution of light that these parameters produce. Examples of these parameters are: the geometry description; the optical properties of the regions in the geometry and the number, type, position, orientation and intensity of the light sources. The second, and arguably more interesting problem, is the *inverse* problem. Here, the distribution of light (or a desired distribution) is given and we wish to find the parameters that caused (or would cause) this result. For example, a desired light distribution may be given along with the geometry description and optical properties and the inverse problem would determine the light source parameters that could produce this distribution (this specific technique will be explained further in Section 2.3.3). For the general case in complex geometries, these inverse problems do not have closed-form analytical solutions. Therefore, it is necessary to approximate the solution to the inverse problem using many iterations of the forward problem with different light source parameters. The many iterations required to solve the inverse problem places emphasis on the need for a fast and accurate solution to the forward problem in order to sufficiently explore the large solution space.

2.3 Medical Applications of Light

2.3.1 Diffuse Optical Tomography (DOT)

Diffuse optical tomography (DOT) is a medical imaging technique that uses mathematical tomographic techniques to reconstruct 3D images of tissues by measuring the light interaction between pairs of sources and detectors [13]. DOT is often used as a complement to functional MRI (fMRI) by providing similar information but using different mechanisms with a different quality-cost tradeoff (lower quality and lower cost compared to fMRI). DOT shows significant potential for continuous monitoring of brain oxygenation, for example with premature infants in neonatal intensive care and stroke victims [39]. In these scenarios, MRI is not suitable due to its cost, size and patient comfort concerns, resulting in an advantage for light-based techniques. A DOT setup uses several sources of light operating at multiple wavelengths with detectors placed around the area of interest, as shown in Fig 2.1. The light is emitted at various wavelengths from the sources, propagates through the tissue and is received at the corresponding detectors. The major difficulty with this imaging technique is that the light can be scattered many times before arriving at a detector. Determining the structure of the underlying tissue that caused the light scattering is an inverse problem with no closed-form solution. Therefore, many forward simulations with different tissue parameters (size, location and optical properties) are required to find the best match to

the experimental results. The quality of DOT is strongly correlated with the accuracy and quantity of the forward light propagation simulations. The feasibility of this technique depends on the computational ability of the underlying simulator and therefore motivates the development of a fast and accurate light propagation simulator.

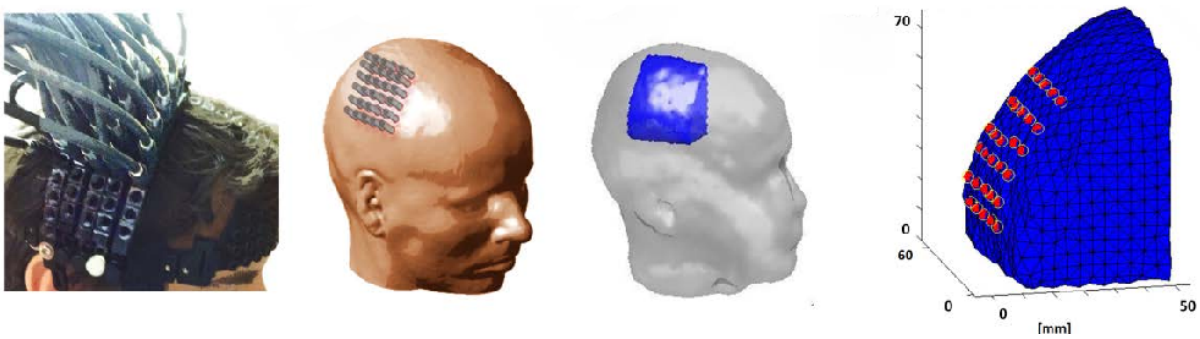

Figure 2.1: High-Resolution Diffuse Optical Tomography (HR-DOT) setup [28]

2.3.2 Bioluminescence Imaging (BLI)

Bioluminescence Imaging (BLI) [47] is another interesting application of light for medical imaging. BLI is currently being used in a laboratory setting to track the size and location of cancerous tissues in pre-clinical small animal treatment studies. An example of this is shown in Fig 2.2. In BLI, a cell type presenting a specific disease is transfected with a virus that changes its genes and causes it to create a protein that produces light without the application of an external energy source. This allows the cells to be monitored using a low-light camera. When these cells duplicate, they retain the light emitting gene. Most current BLI work is qualitative by tracking the progression and spread of the diseased cells. A more modern technique known as Quantitative BLI (qBLI) [42] reconstructs a geometric model of the volume of interest using knowledge of the underlying anatomical structure and optical properties. This information provides constraints for an inverse solver that tries to minimize the difference between the simulated and experimental light pattern by running many light propagation simulations. A fast forward light propagation simulator allows more simulations to be run and thereby increases the quality of the reconstructed structure.

2.3.3 Photodynamic Therapy (PDT)

Photodynamic therapy is a light-based therapy used to kill diseased tissues, such as cancer cells and bacterial infections. In PDT, the patient receives a photosensitizer (PS), either topically, orally or by injection. The PS accumulates preferentially in tissues with rapidly dividing cells, like tumors, and absorbs light of a specific wavelength. Physicians administer the light using either external (e.g. light irradiating the skin) or internal (e.g. needle-sized fiber optic probe) sources. Both the inactive PS and light are harmless on their own, but when enough photons interact with the PS it activates and causes the surrounding oxygen to become reactive. The reactive oxygen species damage the cells and, given sufficient damage, cause tissue death by either apoptosis or necrosis. The goal of PDT is to inflict enough damage to kill the target tissue while minimizing the damage to healthy tissue [63].

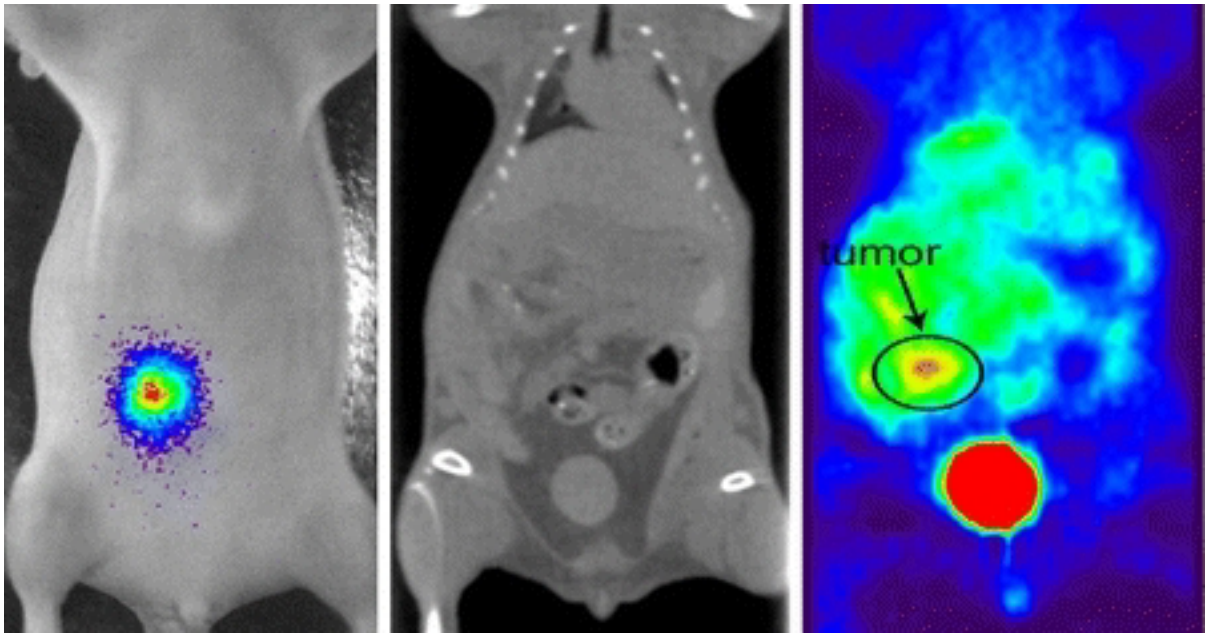

Figure 2.2: From left to right: Reconstructed BLI image, CT scan, PET scan and dissection photograph of mouse with a tumor (reproduced from [44])

Interstitial PDT (IPDT) is the use of PDT within the body using fiber optic probes that are inserted into the patient using needles. IPDT complicates treatment planning since it has many more degrees of freedom; however it has the potential to treat more vital organs inside the body. There are many factors which affect the outcome of PDT including the light distribution, PS concentration and tissue oxygenation. The most controllable factors are the light source parameters (i.e. number, type, position, orientation and power).

PDT has two main uses for a light propagation simulator. The first is for treatment planning verification, where the light source parameters and PS concentrations are known and the simulator is used to determine if a given plan is *good* (i.e. there is sufficient target tissue coverage and acceptable damage to the healthy tissue). These simulations are *online*, meaning the patient has already been administered the PS and the fiber optic probes have been injected. In this case, it is important to have a fast simulator which can determine the safety and effectiveness of the imminent treatment quickly and therefore reduce the time the patient spends in the operating room. The device used to perform the simulations will need to be portable as it may need to move between operating rooms. This type of simulator requires a low power design that can be performed on-site. The second use of the simulator is for PDT treatment plan development, which uses many *offline* simulations. In this case, instead of determining the quality of an existing treatment plan, the simulator is used to solve the inverse problem (discussed previously in Section 2.2) of creating a treatment plan with the goal of reducing the damage to healthy tissue. As already discussed, the most controllable factors for PDT treatment planning are the light source parameters. Hence, PDT treatment planning involves determining the type, quantity, position and power of the light sources that minimizes healthy tissue damage while still eliminating the target tissue. This inverse problem has no closed-form solution and requires iterative techniques to approximate the optimal solution. The performance bottleneck of existing PDT treatment planning tools is the amount of time spent running forward light propagation simulations [65, 66]. Developing a

fast simulator for the forward problem could allow inverse solvers to finish in minutes rather than hours or improve the treatment quality by running more simulations in an allocated time.

2.4 Simulation Models for Light Propagation

The *forward problem* solved by light propagation simulators requires the following problem description:

1. A description of the 3D geometry that divides it into one or more regions, where each region is generally a tissue type
2. A set of material properties for each of the different regions in the geometry
3. A list of light sources
4. The type of output data to be collected

The following sections will discuss the problem description in more detail and the section will conclude with a discussion on models for solving the light propagation given the description.

2.4.1 Geometry Description

There are a number of ways to describe a geometry when modelling light propagation, each with its own accuracy-complexity tradeoffs. In general, geometries consist of a set of elements. No matter how the geometry is described, each element has a boundary with a defined surface normal ($\hat{\boldsymbol{n}}$), a set of material optical properties (μ_s, μ_a, g and n) and a set of neighbouring elements. The geometric element description must support the following functions:

- Determining whether a point is within the element
- Determine the point on the boundary with another element where a ray intersects
- Calculating the volume of the element
- Finding the elements directly adjacent to the current element

In general, sets of elements are used to discretize the continuous regions of the 3D geometry. For example in Fig 2.3, the 2D region is discretized using pixels (squares) in the middle and triangles on the right. The analogous shapes in 3D are voxels (cubes) and tetrahedrons, respectively. The following sections will discuss different geometric representations for 3D volumes and their respective accuracy-complexity tradeoffs. This discussion will also justify the choice of a tetrahedral geometry for our work as it enables the most general light propagation simulator.

Infinite

The simplest geometry description is an infinite homogenous medium. Here, the 3D geometry is modelled as a single infinite homogeneous element. This model has no external boundaries (infinite) and no material boundaries (homogeneous). These simplifications allow models using this geometry to have an exact analytical solution using diffusion theory (discussed later in this section). The simplicity of this geometry results in high computational performance, but at the cost of flexibility and practicality since few real-life medical cases can be reduced to this geometry.

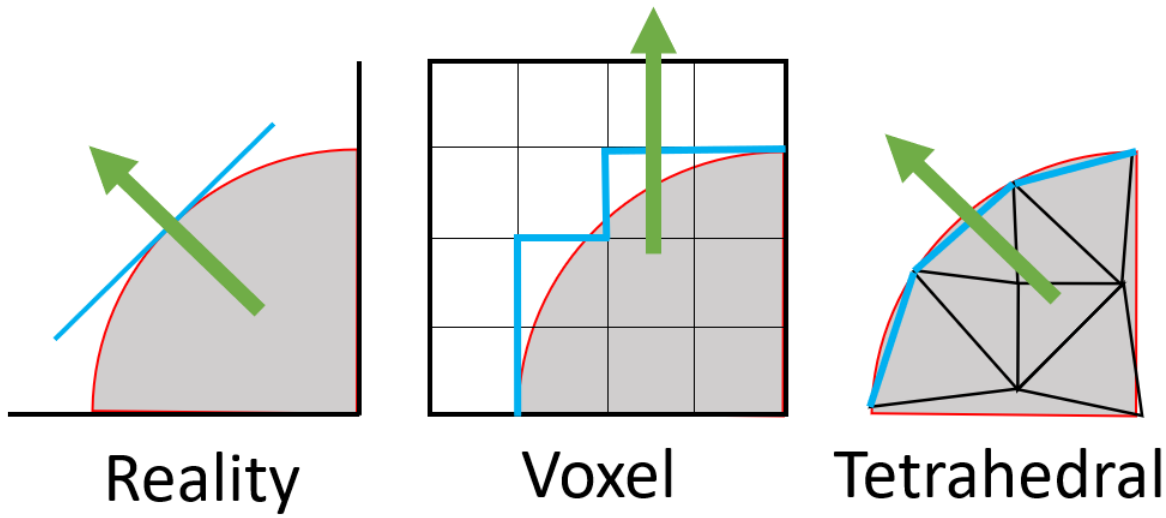

Figure 2.3: 2D slice of a curved surface (left) modelled using voxels (middle) and tetrahedrons (right) - the arrows represent the estimated surface normal $\hat{\boldsymbol{n}}$

Planar

The first widely used simulators model the geometry with planes (or layers) of infinite width and height but finite depth (called semi-infinite), as shown in Fig 2.4. Describing this geometry requires only the depth and material properties of each layer. Detecting boundary intersections is slightly more difficult than the infinite case, since now the photon can reflect or refract when moving between layers with different material properties. However, the calculation can be simplified since the normal of every layer is parallel to the z-axis. Planar geometries are practical for skin related diseases since the geometry maps well to the problem. However, for general clinical models with many regions and curved surfaces, the planar geometry is far too restrictive.

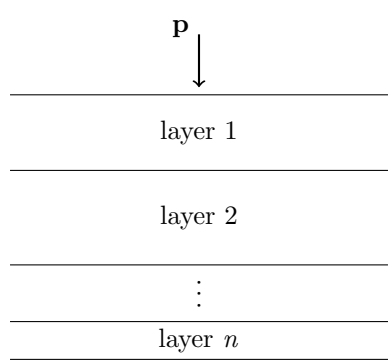

Figure 2.4: Layered geometry with normal-incident pencil beam light source p

Voxelized

More complex 3D geometries can be discretized by using many voxel (cube) elements. While this approach can model a much wider variety of geometries than the infinite and planar elements described earlier, it cannot accurately represent curved surfaces and can result in very inaccurate surface normals,

as shown in Fig 2.3. Moreover, the voxel model's inaccuracy in representing surface normals does not improve even as the number of voxels is increased (and hence the size of each voxel decreases). Binzoni et al. [11] provide a more detailed examination of the shortcomings of voxels in representing the normals of curved surfaces. As discussed later in this section, in heterogeneous media the accuracy of the surface normal is crucial in simulating the reflection and refraction of photons that cross material boundaries.

Tetrahedral

3D geometries can also be discretized using tetrahedrons (triangular based prisms) instead of voxels, at the cost of additional complexity. Compared to voxels, tetrahedrons provide a more accurate representation of the 3D geometry, especially at curved surfaces and material boundaries, as shown in Fig 2.3. The surface normals are much more closely approximated, and, unlike with voxels, the accuracy of surface normals increases as the tetrahedron size shrinks. This allows for the choice of more accuracy when needed, at the cost of more tetrahedra. However, this increased accuracy comes at the cost of increased complexity and computation; tetrahedrons lack the regularity of cubic voxels and therefore have more complex ray intersection and point containment calculations. A more detailed discussion of tetrahedral meshing can be found in *Computational Geometry* written by O'Rourke [52]. The work in this thesis is intended to be highly accurate for general medical applications and therefore we decided to use a tetrahedral mesh to represent the 3D geometry.

2.4.2 Light Source Descriptions

There are various types of real-life medical light sources with different shapes and emission profiles that need to be modelled in the light propagation simulation. A subset of these light sources are listed below and illustrated in Fig 2.5.

- Isotropic point
- Cylindrical diffuser
- Fiber cone

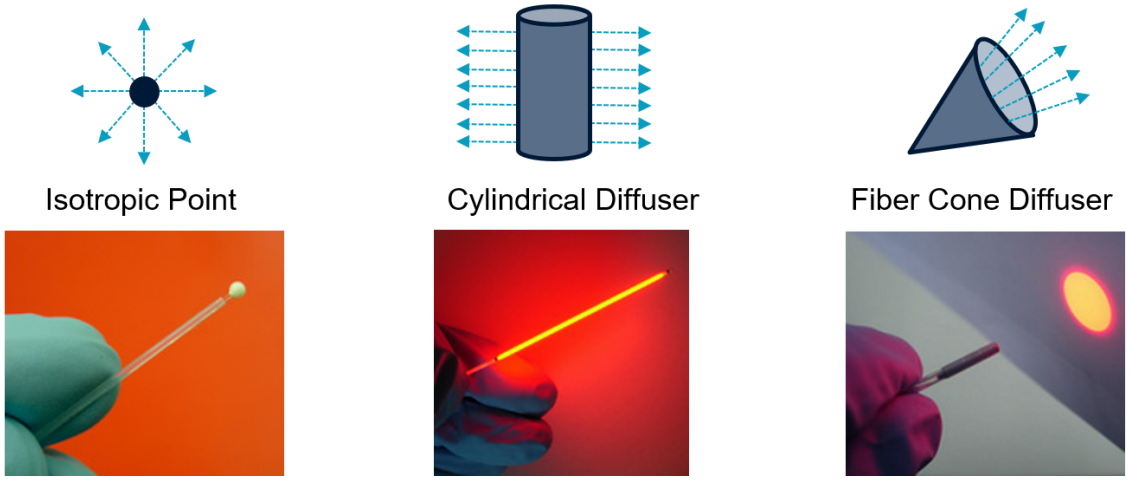

Figure 2.5: Example light sources with emission profiles [2]

Previous simulators may not support all the source types due to restrictions from design simplifications or a development choice not to include them. Diffusion theory solutions (discussed in Section 2.4.4) only support isotropic sources since the diffusion approximation is not compatible with any concept of directed light. It is also possible to create more complex profiles by using a sum of simpler point sources.

2.4.3 Output Data

When solving light propagation in turbid media, the *Radiative Transfer Equation* (RTE) for a specific wavelength is the conservation relation that must be solved [53]. The RTE is a complex partial differential equation, shown in Equation 2.1, that sets the conditions for a function, $L(x, \hat{\Omega})$, to describe the radiance at a point x in direction $\hat{\Omega}$. This equation is made up of components that account for light emittance ($s(x, \hat{\Omega}, t)$), scattering ($\mu_s(x, \hat{\Omega}' \rightarrow \hat{\Omega})$) and absorption. Biophotonic simulations are often concerned with the fluence Φ, which is the integral of radiance at a specific point x, as shown in Equation 2.2.

$$\frac{1}{v}\frac{\partial}{\partial t}L(x, \hat{\Omega}, t) + \hat{\Omega} \cdot \nabla L(x, \hat{\Omega}, t) + \mu_t(x)L(x, \hat{\Omega}, t) = s(x, \hat{\Omega}, t) + \int_{\Omega} L(x, \hat{\Omega}', t) d\mu_s(x, \hat{\Omega}' \rightarrow \hat{\Omega}) d\hat{\Omega}' \quad (2.1)$$

$$\Phi_x = \iint_{\Omega} L(x, \hat{\Omega}) d\hat{\Omega} dt \quad (2.2)$$

As discussed in Section 2.4.1, we discretize the problem geometry into finite sized elements R_i with homogeneous optical properties. The average fluence in a region of finite volume V_{R_i} can be calculated using Equation 2.3. For generality, in this section we will use the term *element* to refer to the 3D shape (e.g. planar slab, voxel, tetrahedron) used to discretize the geometry. Later, we discuss our choice of tetrahedral elements for our simulator.

$$\Phi_{R_i} = \frac{1}{V_{R_i}} \int_{R_i} \Phi_x dV \quad (2.3)$$

Our simulator tracks the photon absorption within each volume, which is proportional to energy E_{R_i}. Since each mesh element has homogeneous optical properties, Equations 2.2 and 2.3 allow us to derive an equation for the average fluence Φ_{R_i} in a finite sized element, shown in Equation 2.4.

$$\Phi_{R_i} = \frac{E_{R_i}}{V_{R_i}\mu_a} [Jcm^{-2}] \quad (2.4)$$

This allows us to calculate the average fluence for each mesh element using the tracked photon absorption energy in the element E_{R_i}, the element volume V_{R_i} and the absorption coefficient assigned to the element μ_a. To track the absorbed energy during the simulation, we maintain one entry in an array for each element. Every time an absorption event occurs, the simulator reads from the array, adds the absorbed value to the read value and then writes it back into the same array entry; we call this a *read-accumulate-write* operation.

Other applications, like BLI and DOT, are interested in *surface fluence emittance*; the fluence escaping specific regions within the 3D geometry, or the 3D geometry itself. For a discrete element S_i that has E_{S_i} photon energy passing through it and area A_{S_i}, the average surface fluence Φ_{S_i} can be calculated using Equation 2.5. The shape of the surface S_i depends on the element chosen to discretize the 3D geometry. As visualized in Fig 2.3, if tetrahedrons were used, S_i would be a triangle; if voxels were used,

S_i would be a rectangle. Tracking these values during the simulation requires an array with an entry per element face (bounded above by $4 * |elements|$, since one face is shared between two elements, unless it is on the surface of the geometry). Post-processing techniques can be used to determine which faces are on the surface between two regions in the geometry or which faces are on the surface of the geometry.

$$\Phi_{S_i} = \frac{E_{S_i}}{A_{S_i}} [Jcm^{-2}] \tag{2.5}$$

2.4.4 Numerical Solutions

Since the RTE is a complicated partial differential equation, it only has analytical solutions for extremely simple homogenous cases and approximations. Analytical methods break down at the interfaces between materials, the external boundaries of a model and at the sources and sinks of light [40, 41]. For these reasons, two numerical solutions are often employed to approximate the resulting light distribution: the Finite Element Method (FEM) or the Monte Carlo method (MC). Both methods discretize the problem and numerically approximate the RTE, but in different ways.

Finite Element Method (FEM)

Finite Element Method (FEM) solutions for diffuse light propagation model the light by assuming it is diffusing across a concentration gradient, similar to heat transfer or chemical diffusion. Jacques and Pogue [36] provide a more in-depth discussion on FEM solutions for diffuse light propagation. The restrictions placed on the problem definition for accurate FEM solutions are summarized in the list below:

1. All materials must have a high albedo ($\alpha \approx 1 \implies \mu_s >> \mu_a$)
2. There are no non-scattering voids in the geometry ($\mu_s > 0$, for all regions)
3. All light sources are isotropic
4. Results are typically invalid within a few *mean free paths* of a light source
5. All materials have a uniform refractive index

These restrictions reduce the problem to solving a large sparse matrix, for which there are many fast and accurate techniques. However, these restrictions can make the solution unusable for practical applications. For example in PDT, it is typical to use diffusing cylinder and cut-end fiber light sources; these are inherently non-isotropic and therefore not supported by FEM solutions due to restriction 3. Additionally, light sources in PDT are usually placed very close to or within the diseased tissue and therefore the accuracy of the simulation near these light sources is crucial; this is not supported in FEM solutions due to restriction 4. Finally, in many practical medical applications, the complex anatomy will have many material interfaces with vastly different refractive indices and possibly cavities of air where light is not scattered (e.g. the oral cavity, the lungs and the bladder); this breaks restrictions 2 and 5. As discussed in the next section, the Monte Carlo method does not introduce any of the aforementioned restrictions. In this work, we wish to develop an accurate and general light propagation simulator and therefore choose to use the Monte Carlo method for approximating the RTE.

Monte Carlo Method

Monte Carlo (MC) methods for simulating light propagation do so by tracking individual photons using random numbers sampled from the statistical distribution of interaction event probabilities, such that the expected behaviour of the photon is physically and statistically accurate. Millions of these photons are simulated and after a sufficient number of individual simulations, the output converges to a statistically accurate result. Since the path of a single photon has no effect on the path of any other photon, the photons are mutually exclusive and therefore the MC algorithm is highly parallelizable.

In this work, we use the popular *hop-drop-spin* method first proposed by Wilson and Adam [62]. Prahl et al. [55] refined the algorithm in 1989 by adding roulette and anisotropic scattering (discussed later in this section). More recently, Wang et al. [61] based their work, MCML, on the hop-drop-spin method for layered geometries. The following sections give a brief overview of the refined hop-drop-spin algorithm used in this work, which is visualized in Fig 2.6. A more detailed description can be found in the original MCML paper [61].

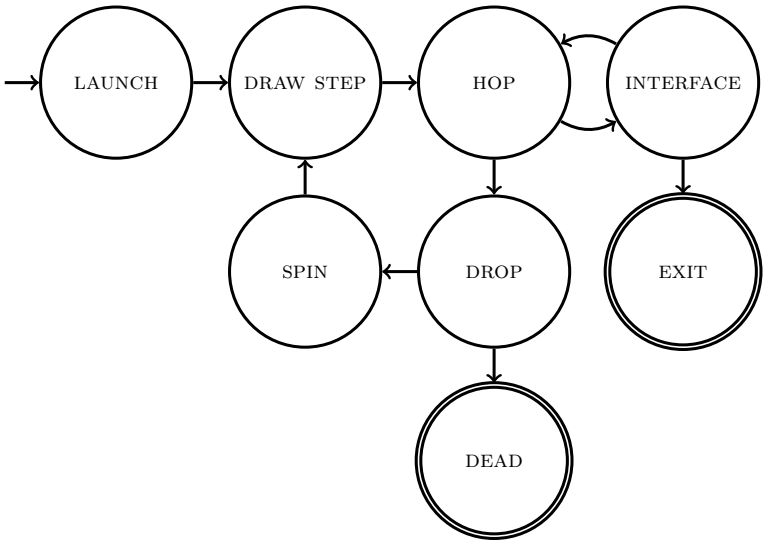

Figure 2.6: The refined hop-drop-spin algorithm used in this work

Launch The LAUNCH stage initializes the photon packet with an initial position $\boldsymbol{p}$, direction $\hat{\boldsymbol{d}}$ and weight w. Tracking the path and weight of photon *packets* as they are gradually absorbed is more accurate and computationally efficient than simulating individual photons [34], which would be completely absorbed by the tissue at their first absorption event.

Draw Step As discussed in Section 2.1, the attenuation coefficient (μ_t) defines the average number of interactions (scattering and absorption) per unit length travelled. In the DRAW STEP stage, a random step length s is generated using Equation 2.6 where μ_t is the attenuation coefficient of the material in which the packet currently resides, and r is a uniformly distributed random number in the range $[0, 1)$ (i.e. $U_{0,1}$). Taking the $-log$ of a uniformly distributed number gives an exponentially distributed random number. For us, this represents the exponentially decaying probability of travelling a distance s

in Equation 2.6. Once the step length has been calculated, the packet then proceeds to the HOP stage.

$$s = -\frac{log(r)}{\mu_t}, r \in U_{0,1} \tag{2.6}$$

Hop Given a step length s, the HOP stage moves the packet along the current direction vector $\hat{\boldsymbol{d}}$ by the step length s to the position $\boldsymbol{p}' = \boldsymbol{p} + s\hat{\boldsymbol{d}}$. If the packet remains in the same tetrahedron after moving from $\boldsymbol{p}$ to $\boldsymbol{p}'$, then the HOP stage is finished and the packet moves to the DROP stage. However, if the packet crosses a tetrahedral boundary, it is moved to the INTERFACE stage to handle potential boundary condition logic.

Interface The INTERFACE stage handles boundary conditions when a packet moves from one tetrahedron T to an adjacent tetrahedron T' during the HOP stage. If the packet exits the mesh (T' is outside the mesh), then computation for this packet ceases. If the user chose to track surface emittance, then we accumulate the current packet weight into the variable tracking emittance for that tetrahedral face. If the packet does not exit the mesh, then the intersection point on the shared tetrahedral face and incident angle θ_i are determined using ray tracing calculations. If the refractive indices (n and n') of the tetrahedrons differ, then the packet crosses a material boundary and the INTERFACE stage performs reflection and refraction calculations.

Snell's law states that if $sin\theta_i > \frac{n}{n'}$, then light incurs total internal reflection (TIR). Otherwise, light is partially refracted and reflected (Fresnel reflection). To model the partial reflection and refraction in simulation, we randomly choose one or the other with appropriate probability. Equation 2.7 is the average of two Fresnel reflection coefficients and the result R is the average probability of Fresnel reflection. We model the probability of a photon incurring Fresnel reflection as a Bernoulli random variable of R. That is, we generate a uniformly distributed random number $x \in U_{0,1}$, and if $R > x$, the packet undergoes Fresnel reflection, otherwise the packet's direction angle is refracted using Equation 2.8.

$$R = \frac{R_s + R_p}{2} = \frac{1}{2}\left[\left|\frac{n_i cos\theta_i - n_t cos\theta_t}{n_i cos\theta_i + n_t cos\theta_t}\right|^2 + \left|\frac{n_i cos\theta_t - n_t cos\theta_i}{n_i cos\theta_t + n_t cos\theta_i}\right|^2\right] \tag{2.7}$$

$$\theta_t = \frac{n}{n'} sin\theta_i \tag{2.8}$$

In the case of both reflection and refraction, the packet's direction vector $\hat{\boldsymbol{d}}$ is updated based on the resulting angle θ_t. Next, the step length s from the HOP stage is reduced by the distance the packet travelled to reach the boundary of the two tetrahedrons, giving a remaining step length of s'. In addition, if the attenuation coefficients of the tetrahedrons differ, then the step length is updated using Equation 2.9 where μ_t and μ_t' are the attenuation coefficients of the previous and current tetrahedrons, respectively. Finally, the packet goes back to the HOP stage with an updated step length s'' and potentially a new direction vector.

$$s'' = -\frac{\mu_t}{\mu_t'} s' \tag{2.9}$$

Drop Once the entire step from the DRAW STEP stage is complete, the DROP stage deposits some of the packet's energy into the current tetrahedron to model photon absorption. As discussed in Section 2.1, the albedo α is the probability that an event is a scattering event (as opposed to an absorption

event). Therefore, $1 - \alpha$ is the probability that an event is an absorption event. To correctly model the behaviour of photons using a photon packet, a packet with weight (i.e. energy) w deposits a fraction of its weight $(1 - \alpha)w$ into the current tetrahedron and continues to the next stage with a new weight of $w' = \alpha w$. The deposited weight is accumulated into the array entry tracking the absorbed energy for the tetrahedron in which the packet currently resides. The energy accumulated in each tetrahedron is used at the end of the simulation to calculate the fluence using Equation 2.4.

Once the fraction of weight has been removed from the packet, the packet enters the *roulette* sub-stage. If the packet's new weight w' is greater than a threshold (w_{min}), then the roulette sub-stage does nothing. If the weight is below w_{min}, then the packet is given a 1-in-p chance of surviving with an increased weight of pw' (due to energy conservation). If the packet loses roulette, it is marked as DEAD and its execution ceases. As a packet propagates and loses more weight due to repeated absorption events, its contribution to the overall energy accumulation declines rapidly. Thus, the purpose of the roulette phase is to avoid wasting computation on increasingly insignificant packets [21]. If the packet's weight was above the roulette threshold or if it survived roulette, it moves to the SPIN stage.

Spin As discussed in Section 2.1, biological tissues are generally *turbid*, which means they scatter light. This is accounted for in the SPIN stage. In this stage, the packet's direction vector $\hat{\boldsymbol{d}}$ is changed by an angle θ which is calculated from a random sample of the Henyey-Greenstein scattering function (Equation 2.10) where the variable g is the anisotropy factor of the material in which the packet currently resides. Once the packet has been spun, it moves back to the DRAW STEP stage where the entire process repeats.

$$cos\theta = \frac{1}{2g}\left[1 + g^2 - \left(\frac{1 - g^2}{1 - gq}\right)^2\right], q \in U_{-1,1} \tag{2.10}$$

2.5 Computing Platforms

Due to the slowing of Moore's law and the end of Dennard scaling, developers can no longer rely on an automatic performance increase with the release of new computing technology. Large scale designs require large cooling and power considerations, which influences developers to consider different computing methods to achieve high performance while minimizing power and space. This work focuses on three implementations of the same FullMonte algorithm, each using a different computing platform: Central Processing Units (CPUs), Graphics Processing Units (GPUs) and Field Programmable Gate Arrays (FPGAs). Each platform presents significantly different strengths and weakness, with different abstractions to the programmer. The following sections will give a brief overview of these computing platforms.

2.5.1 Central Processing Unit (CPU)

CPUs consist of fixed computing hardware that exposes a predefined *instruction set* that allows a developer to implement desired functions. The fundamental paradigm that CPUs use is for a central data processing unit to move data in from storage, execute some operations on the data and write the result back to storage. In general, significantly more area and power is used moving data compared to the actual computation [29]. Therefore, the high-level goal of the developer creating an application for

a CPU is to minimize data transfer.

Modern CPUs contain multiple cores, i.e. multiple copies of the central data processing unit mentioned before. This allows mutually exclusive computations to be performed in parallel by utilizing the multiple CPU cores. In addition, the cores in modern CPUs often support *simultaneous multithreading* (or *hyperthreading*). This techniques allows a single CPU core to execute two computations simultaneously by leveraging idle execution units and reducing the overhead involved with context switching [58]. Modern CPUs also support *vector* instructions, which are also called *single instruction multiple data* (SIMD) operations. As the SIMD name suggests, these instructions perform a single operation to a set (vector) of data in parallel using special central data processing units. Each of the multiple cores in modern CPUs support both hyperthreading and vector instructions. This causes a multiplicative increase in performance when using multiple hyperthreaded cores for processing. In Section 2.6, we briefly discuss the performance effect that multithreading, hyperthreading and vector instructions have on our CPU baseline simulator, FullMonteSW.

2.5.2 Graphics Processing Unit (GPU)

GPU Architecture

A GPU consists of an array of *streaming multiprocessors* (SMs) with a set of *scalar processors* (SPs) laid out like the left of Fig 2.7. The number of SMs and SPs depends on the specific GPU device. For example, the NVIDIA Quadro P5000 GPU has 16 SMs that each have 128 SPs [22]. The memory hierarchy of GPUs is also illustrated in Fig 2.7. The largest memory in the GPU is global memory which is used to transfer data between the CPU and GPU. Global memory is the only memory with enough capacity to store the tetrahedral-meshes used in our work but the access times are slow, as shown in Table 2.1. However, the GPU contains smaller and faster memories that can be used to improve the latency and throughput of accesses. The L2 cache of the entire GPU provides device managed caches for global memory. In addition, each SM contains memory that the developer can use to create custom caches including constant cache, shared memory and registers, as indicated in the right of Fig 2.7. The constant cache can only be read by the SPs, so it is often used for storing computational constants, such as the material properties in our case. Shared memory and registers can be read from and written to by all SPs in an SM, and can be used for explicitly managed caches. In our work, we use both of these to cache absorption events locally to avoid excessive accesses to the much slower global memory. Table 2.1 summarizes the relative access latency and capacity of these GPU memories [49].

Table 2.1: NVIDIA GPU Memories [49]

Memory	Cycle latency (relative)	Size
Registers	1	63-255 per thread
Local	8-21x	512kB per thread
Shared	8x	48-96kB per block
Constant	8-34x	64kB
Texture	8-34x	12-64kB
Global	34x	64MB-32GB

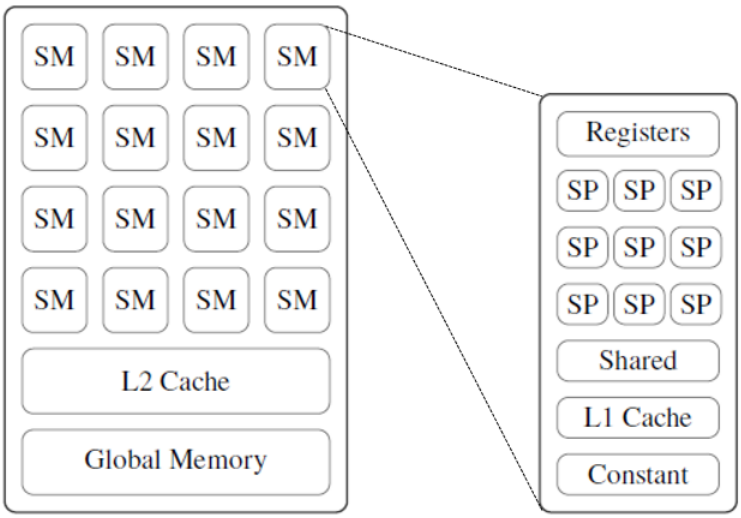

Figure 2.7: A high-level depiction of the GPU streaming multiprocessor (left) and scalar processor (right)

Compute Unified Device Architecture (CUDA)

The *Compute Unified Device Architecture* (CUDA) is NVIDIA's development interface that provides a high-level view of the GPU architecture making it easy for developers to program. CUDA is a C-style programming interface consisting of host code that runs on the CPU and device *kernels* that run on the GPU [49]. The general flow for the CUDA host program is illustrated in Fig 2.8. The host code loads the data, copies it to the GPU, launches the kernel on the GPU, waits for the GPU to finish, copies the output data back from the GPU and finishes execution. The CUDA kernel code represents the execution of a single thread of computation. When launching the GPU kernel, the host code is responsible for specifying the number of threads to use, and these threads are run in groups called blocks. To define the number of threads, the user specifies the number of threads per block and the total number of blocks (*total threads* = *threads_per_block* * *total_blocks*). Each block is assigned to a single SM and, once assigned, all of the threads in the block run exclusively on that SM until completion. When a block is assigned to an SM its threads are automatically divided into groups of 32 called *warps* [49]. The threads in a warp execute in lock step on multiple SPs in one SM, where each SP is executing the same instruction on different data.

GPU programming using the NVIDIA CUDA development platform

Efficiently accelerating an algorithm using a GPU requires knowledge of the underlying hardware and development platform. For more information, the reader is directed towards the NVIDIA *Programming Guide* [49] and *Best Practices Guide* [48] which discuss extensively the GPU architecture, CUDA development platform and best methods for programming NVIDIA GPUs. Previous works [4, 5, 43, 70] discuss these concepts in the context of MC light propagation simulations.

Three significant challenges arise when implementing our algorithm on a GPU. First, the tetrahedral meshes used to represent realistic models require a large amount of memory (∼gigabytes). Modern GPUs can fit these large meshes into global memory but, as discussed in the previous sections, the access times are slow. Moreover, due to the random behaviour of MC simulations, the tetrahedrons are accessed

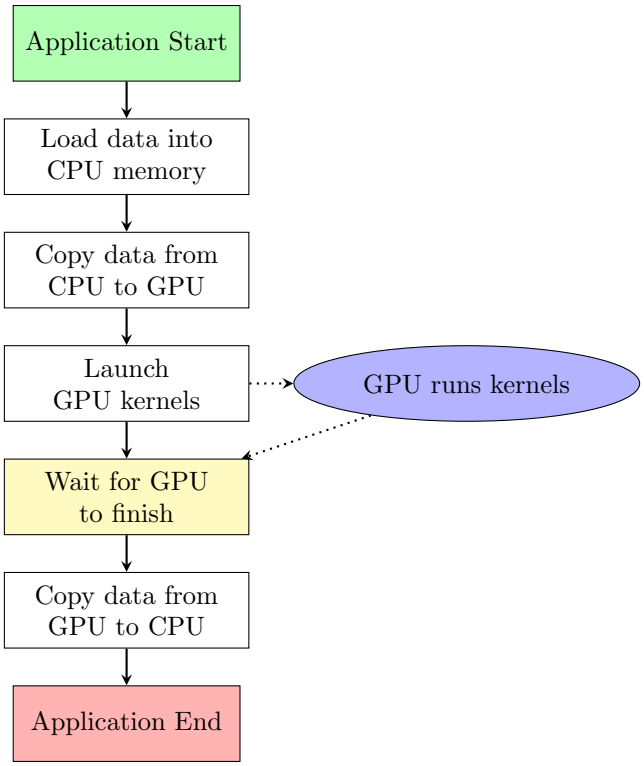

Figure 2.8: The general flow of a CUDA program. Squares states are run on the CPU and the purple oval is run on the GPU.

irregularly, making them difficult to cache effectively. Thus, we expect a GPU implementation of the algorithm to be *memory bound*, meaning that kernels will often stall waiting for memory accesses. Second, MC simulations require many blocks of code that may or may not execute depending on the state of the simulation. This *conditional* code does not map well to the GPU and can cause significant performance degradation due to the lock step execution of threads in a warp; this is called *thread divergence* [15]. Lastly, when many photon packets are being simulated simultaneously in multiple threads, it is possible that multiple packets can drop weight into the same tetrahedron at the same time. The developer must account for this by ensuring that simultaneous absorptions into a tetrahedron are handled accurately and efficiently.

2.5.3 Field Programmable Gate Array (FPGA)

FPGA Architecture and Design

Both CPUs and GPUs execute a set of sequential instructions to perform a specific task. This is a simple model for developers as it is similar to algorithmic thinking: "First do A, then do B, then do C, done". FPGAs take a different approach by performing computations *spatially*. At a high level, an FPGA is an array of programmable logic and specific processing elements including memory blocks, arithmetic blocks (multiplication and addition) and registers all connected by programmable routing that moves inputs and outputs to and from the different blocks. FPGAs are *reprogrammable* by loading a *bitstream* onto the device which specifies what functions the elements implement and how the inputs/outputs of these elements are wired together. This allows memory and computation elements to be intermixed

and therefore be stored *closer* together. As discussed in the previous CPU section, this can reduce the energy expended to move data between computation units. Applications that present *pipeline parallelism* are typically good candidates for FPGA acceleration. Exploiting pipeline parallelism involves chaining together large sets of dependent computations into a pipeline for processing. The maximum operating frequency ($Fmax$) of a circuit is the inverse of the longest combinational logic delay between two registers. When many pieces of data are being pushed through a pipeline, the initiation interval (II) is the number of cycles the pipeline must wait before starting a new computation. The performance of a pipeline is dependant on the circuit's Fmax and the II. Given input data of size N and a pipeline with a latency of L, the number of cycles (C) to perform the entire computation is given by Equation 2.11 and the computation time (t) is given by Equation 2.12. As shown in Equation 2.11, minimizing the II is crucial to the performance of the pipeline for large values of N (i.e. large problem sizes).

$$C = L + II * (N - 1) \tag{2.11}$$

$$t = C * \frac{1}{Fmax} \tag{2.12}$$

FPGA Programming

Typically, FPGA designs are described using a *Register Transfer Level* (RTL) *Hardware Description Language* (HDL) like Verilog or VHDL. HDL code is a low-level description of the circuit to be implemented on the FPGA and requires the developer to manually manage which functions are performed in each clock cycle. Writing custom RTL code gives the hardware developer extreme design flexibility and careful optimizations can result in a design that is highly efficient in performance, area and power. However, RTL descriptions are extremely verbose and require a cycle-by-cycle description of the algorithm. Historically, this has made FPGA designs error-prone, time consuming and expensive to develop, difficult to debug and tedious to extend to new devices, architectures and applications. In addition, developing a complete system that interacts with a host CPU for data input and output is difficult as it usually involves creating or using low-level programming interfaces and drivers to transfer data between the CPU and FPGA.

High Level Synthesis (HLS) provides a generalized abstraction to the hardware by allowing developers to use sequential languages, like C/C++, or explicitly-parallel languages, like OpenCL. HLS languages (e.g. Intel HLS and Vivado HLS) relieve the burden of describing the cycle-by-cycle description of an algorithm by writing instead allowing the developer to write sequential code, which the HLS tool then automatically schedules to determine which operations occur in each clock cycle. Unfortunately, this reduces the flexibility of the design since the user has less control over the instantiated circuit, especially when using the FPGA's hard blocks (e.g. RAMs and DSPs), which can significantly affect the performance and area of the design. Most HLS tools work by compiling sequential functions into RTL IP cores. When using HLS, the user must often manually connect these IP cores using a system integrator. Similar to RTL designs, HLS designers must determine how to transfer data between the CPU and FPGA when creating a full system, which usually includes low-level driver code that is time-consuming to create and use.

Both Intel and Xilinx provide an OpenCL SDK for developing FPGA applications using the OpenCL parallel programming language. OpenCL is very similar to CUDA, which was discussed in the previous

GPU section. The thread structure and host/kernel interactions are nearly identical. When programming FPGAs, OpenCL shares many of the same advantages and disadvantages as C/C++ HLS; it sacrifices flexibility for ease-of-development. Unlike C/C++ HLS, the OpenCL standard contains methods for interfacing with the device (in our case, an FPGA), which includes transferring data between the CPU and FPGA and querying device kernels while they run. The manufacturer of the board creates a *board support package* (BSP) that is used by the OpenCL compiler to abstract the host-accelerator interactions [32]. This makes developing a full FPGA-accelerated system generally much simpler and faster than both RTL and C/C++ HLS, as the CPU-FPGA communication is specified at a higher and standardized level. In Section 4.1, we justify why we use OpenCL HLS for our application over C/C++ HLS and traditional HDL. In Section 4.7.2 we discuss the strengths and weakness of OpenCL that we encountered in our work.

2.6 Previous Implementations

Table 2.2 summarizes some of the published MC simulators using CPUs, GPUs and FPGAs. The following sections give a brief description of each simulator and provide some strengths and weaknesses compared to the FullMonte approach. In Chapter 4, we compare our GPU- and FPGA-accelerated simulators to a subset of the simulators in this list. It is important to note that, in most cases, hardware accelerated simulators benchmark their performance against a CPU baseline. We have discovered that these baseline CPU implementations exhibit various levels of optimization, which can make comparing relative performance results for hardware accelerated versions difficult and confusing. We have found that multithreaded implementations show linear performance scaling with the number of CPU cores as well as significant improvement using *simultaneous multithreading* (described in Section 2.5). Therefore, multithreaded implementations using modern CPUs with six cores are inherently ∼9x faster than single-threaded implementations [58]. We also found that hand-coded vector instructions (described in Section 2.5) provided an additional speedup of ∼8x [58]. In summary, this shows that **unoptimized single-threaded implementations can be ∼72x slower than optimized multithreaded implementations on modern CPUs**. This fact should be kept in mind when reading the next sections that describe previous MC simulator implementations and their performance numbers.

MCML

The MCML algorithm, originally developed by Wang et al. [61], remains a widely-used MC simulator for turbid media. It uses a planar (i.e. layered) geometry with a normally-incident pencil beam light source. The main drawbacks of this approach are the limited number of light sources and the restrictions of the planar geometry that is not capable of representing the 3D curved surfaces of general biological tissues. CUDAMC was an initial attempt at accelerating the MCML algorithm using a GPU [5]. It uses a single, semi-infinite, planar slab that does not absorb photons. The reported 1000x speedup over the single-threaded CPU code likely reflects the absence of absorption events and limiting the geometry to a single planar slab. CUDAMCML [5] is a more complete implementation of the MCML algorithm which achieves a 100x speedup over the original single-threaded MCML code. The 10x difference relative to CUDAMC is likely attributable to the increased complexity of multiple absorbing layers. More recent work by Alerstam and Lo [4] and Lo et al. [43] called GPU-MCML achieved a 600x speedup over the original MCML code. This incremental improvement on CUDAMCML was achieved by caching

Table 2.2: A subset of existing MC light propagation simulators

Implementation	Geometry	Acceleration Type
MCML	Planar	
CUDAMC	Semi-infinite planar	GPU
CUDAMCML	Planar	GPU
GPU-MCML	Planar	GPU
FBM	Planar	FPGA
tMCimg	Voxel	
MCX	Voxel	GPU
Dosie	Voxel	GPU
MCxyz	Voxel	GPU
Hung et al.	Voxel	FPGA
Afsharnejad et al.	Voxel	FPGA
TIM-OS	Tetrahedral	Vector (automatic)
MMCM	Tetrahedral	Vector (manual)
MOSE	Tetrahedral	GPU
Powell and Leung	Tetrahedral	GPU
MCtet	Tetrahedral	GPU
Cassidy et al.	Tetrahedral	FPGA
FullMonteSW	Tetrahedral	Vector (manual)
FullMonteCUDA	Tetrahedral	GPU
FullMonteFPGACL	Tetrahedral	FPGA

absorption around the directed beam light source and by using a modern GPU architecture. In the same work as GPU-MCML, Lo et al. also created FBM [43] an implementation of MCML on an Altera Stratix III FPGA. They constrained the model to 10 layers and a limited number of accumulation elements. With these constraints, they achieved and advantage of 45x in speed and 700x in energy-efficiency over the single-threaded, unoptimized CPU code. However, CUDAMC, CUDAMCML, GPU-MCML and FBM are all still restricted by the original MCML limitations.

tMCimg and MCX

tMCimg [14] was one of the first open-source, voxelized MC simulators. tMCimg was developed specifically to model the human head and brain for DOT. tMCimg is single-threaded because it was developed at a time when multi-core machines were scarce. MCX [26] is a GPU-accelerated version of tMCimg and therefore has the same geometrical and use-case limitations. MCX reports a 75-300x speedup over the single-threaded tMCimg software implementation depending on the simulation options. Yu et al. [69] extended and improved the implementation of MCX using OpenCL which, compared to the CUDA implementation, allowed them to more easily target a heterogeneous computing platform. Their optimizations allowed them to achieve up to a 56% improvement on AMD GPUs, 20% on Intel CPUs/GPUs and 10% on NVIDIA GPUs [69].

Dosie

Beeson et al. developed Dosie [8], which calculates light transport and photokinetics for PDT in mouse models. Dosie uses a voxel-based geometry and internal or external light sources. The authors do not compare their work against other simulators, but report an absolute runtime of 21 seconds for 2×10^6 packets in a cube model with 10^6 voxels. The developers of Dosie acknowledge that tetrahedral models can fit curved surfaces better than voxel models [11].

MCxyz

MCxyz [33] is an open-source, single-threaded, voxel-based MC simulator. It uses the same *hop-drop-spin* method as FullMonte but only supports emission from a single isotropic point source. MCmatlab [45] extended the MCxyz algorithm to include a finite-element heat diffusion and Arrhenius-based thermal tissue damage simulator with a MATLAB interface. MCmatlab is roughly 17x faster than the single-threaded unoptimized baseline MCxyz [45]. Dupont et al. [24] extended MCxyz to specifically simulate the cylindrical diffusers used in IPDT. They accelerated MCxyz using an NVIDIA GPU but only provide performance results for a simple homogeneous cube model. They report a 745x improvement over the unoptimized, single-threaded MCxyz code for this model [24].

Hung et al.

Hung et al. [30] described the MCML algorithm using OpenCL and targeted both an NVIDIA GTS 450 GPU and Intel Stratix V FPGA. In this work, they used the same OpenCL code to target both the GPU and FPGA. As discussed later in Chapter 4, we found it necessary to write completely different OpenCL code for the FPGA than a GPU to achieve reasonable performance. The authors reported that the GPU and FPGA provided a speedup of 64x and 21x, respectively, over the unoptimized and single-threaded MCML code for layered models.

Afsharnejad et al.

Afsharnejad et al. [1] implemented a voxel-based simulator on a Xilinx Kintex Ultrascale FPGA using the Vivado HLS design suite. They report a 3x speedup over FullMonteSW (which uses tetrahedral elements) for the TIM-OS *cube_5med* model and a 3.7x improvement in energy-efficiency. The simulations are limited to isotropic point sources and 8000 voxel elements. However, as discussed in Section 2.4.1, voxels have simpler intersection calculations but cannot accurately represent the curved surfaces in realistic medical models.

TIM-OS

Before FullMonteSW, TIM-OS [59] was the fastest tetrahedral-mesh MC simulator. TIM-OS is highly optimized software that uses automatic compiler vectorization (i.e. the compiler tries to replace conventional instructions with vector instructions whenever possible). TIM-OS greatly exceeds the performance of MCML on simple layered models, while also supporting more complex tetrahedral models [20]. TIM-OS does not support tracking fluence through surfaces which makes it unsuitable for BLI and DOT.

MMCM

MMCM is a widely used, multithreaded, tetrahedral-mesh, MC simulator written in C [25]. It can use meshing shapes other than tetrahedrons, but the authors do not present any significant benefit of that feature. Fang and Kaeli [27] extended MMCM to use manually-coded vector instructions (SSE) with support for multiple ray tracing algorithms. With these optimizations, they report an overall speedup of up to 26% over the baseline (non-SSE) implementation.

MOSE

The Mouse Optical Simulation Environment (MOSE) [38] was developed for optical imaging techniques, such as fluorescence molecular tomography and bioluminescence tomography. Ren et al. [56] created gpu-MOSE, a GPU-accelerated version of the software targeting an NVIDIA GPU. Both MOSE and gpu-MOSE were validated against MCML and gpu-MOSE achieved a speedup of up to 10x over the single-threaded MOSE code.

Powell and Leung

Powell and Leung [54] developed a GPU-accelerated MC simulator to model the acousto-optic effect in heterogeneous turbid media for imaging and optical property measurement. The simulator was validated against MCML for various models and benchmarked against MMCM. Their GPU-accelerated simulator achieved a 2x speedup over the multithreaded MMCM code.

MCtet

MCtet [70] is a GPU-accelerated, tetrahedral-mesh MC simulator. It does not claim to support fluence tracking through surfaces making it unsuitable for BLI and DOT. It was validated against three benchmark models: two MCML layered models and a TIM-OS cube model similar to *cube_5med* [59]. However, MCtet's performance was only benchmarked using the two MCML layered models; the authors do not provide performance comparisons against TIM-OS for any of the three models.

Cassidy et al.

Cassidy et al. [19] implemented FullMonteSW on an Altera Stratix V FPGA using the Bluespec SystemVerilog (BSV) language. The simulator only supports homogeneous meshes which are limited to 48k tetrahedrons. The authors do not provide raw performance numbers as the combined hardware-software system was not functionally complete. The performance estimates were based on simulations and hardware reports. They estimate a 4x speed improvement and 67x energy-efficiency improvement over the optimized and multithreaded FullMonteSW.

FullMonteSW

FullMonteSW [21] is the baseline we use for the validation and performance benchmarking of our hardware accelerated implementations. FullMonteSW is the fastest and most full featured tetrahedral-mesh MC biophotonic simulator written in software. It uses multithreading, hyperthreading and hand-coded vector instructions to achieve high performance. FullMonteSW has been validated and benchmarked

against state-of-the-art simulators TIM-OS and MMCM, achieving a speedup over them of 1.5x and 2.42x, respectively [20]. Chapter 4 discusses how we accelerated FullMonteSW using a GPU and FPGA.

Chapter 3

Software Optimizations

This section discusses additions and optimizations made to the existing software simulator, FullMonteSW, that improve its performance, accuracy and usability. The first section discusses two new light sources that make the simulator more accurate at representing the real light sources used in PDT and BLI. The second section discusses an optimization technique that improves the performance of the simulator and common post-processing queries made by applications that use the simulator to perform more complicated tasks, like PDT treatment planning.

3.1 Representing Medical Light Sources in Simulation

As discussed in Section 2.3.3, PDT treatment planning often uses complex light sources, like cut-end fibers and cylindrical diffusers. In BLI, the source of light is a collection of light-emitting cells, which translates to a region of elements in the 3D mesh. Representing these real light sources in simulation requires non-trivial mathematics and intelligent programming to maintain high performance. The following sections discuss how we implement these light sources and their respective use-cases.

3.1.1 Cylindrical Diffuser Light Source

In IPDT, cylindrical diffusers are commonly used to produce light. The cladding of the fiber optic probe is stripped away such that light emits from the surface of the cylindrical probe but not out of the ends, as shown in Fig 2.5. This process is not perfect, which causes light to emit with a higher probability normal to the surface and the probability shrinks as the angle becomes more parallel to the surface. This is represented by a Lambertian distribution, as shown in Fig 3.1. To increase the accuracy of PDT treatment planning, it is important to accurately represent these cylindrical diffusers in simulation. To achieve this, we developed code for a cylinder light source that emits light similarly to the cylindrical diffusers. The cylinder light source is defined by the endpoints, radius, power and *theta distribution*. The theta distribution is the range of angles, with respect to the cylinder surface, that an emitted photon can have. The user has three options, as depicted in Fig 3.1: normal to the surface, a uniform distribution on the unit hemisphere or a Lambertian distribution. Fig 3.2 illustrates how we randomly generate a vector in a hemisphere centered on an arbitrary plane (a 2D representation is shown, but the concepts are the same for 3D). Algorithm 1 describes how the cylinder source randomly emits photon packets. Fig 3.3 illustrates the emission profile of the cylinder source using a surface normal emission

(top) and Lambertian emission (bottom). We created these images by placing a cylindrical light source in a homogenous cube with high absorption and low scattering; the darker red represents a higher fluence.

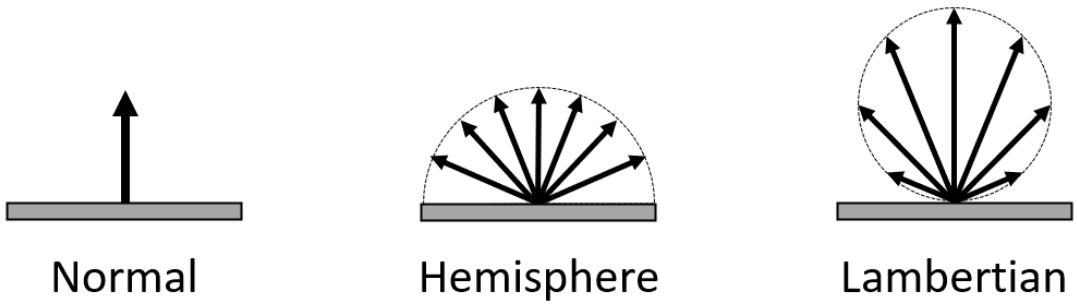

Figure 3.1: Planar theta angle distribution options for surface light sources

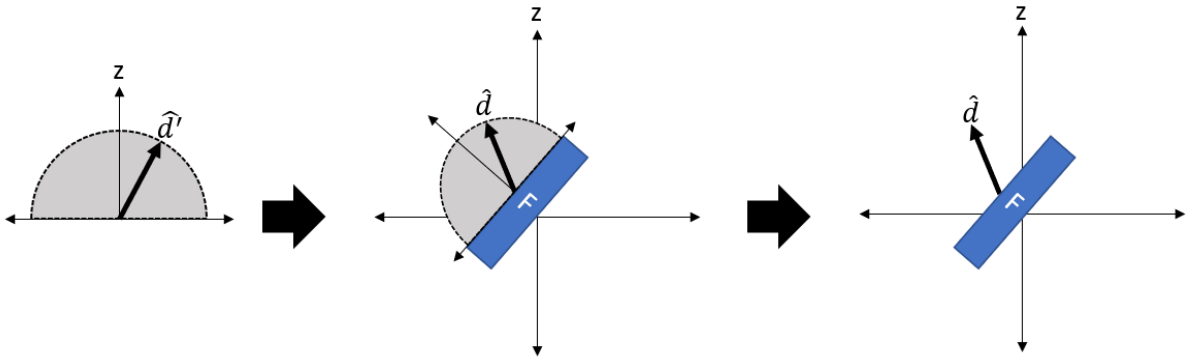

Figure 3.2: Generating a random vector ($\hat{\boldsymbol{d}}$) in a hemisphere centered on the z-axis (left) and rotating this vector to be centered on a plane's (F) normal (middle and right).

Algorithm 1: Emitting Light from Virtual Cylindrical Diffuser

Data: The cylinder information

Result: An initial position $\boldsymbol{p}$ and direction $\hat{\boldsymbol{d}}$ for the packet

$\boldsymbol{p} \leftarrow$ randomly choose a point on surface of the cylinder (excluding the circular ends)

$\hat{\boldsymbol{c}} \leftarrow$ the normal vector at point $\boldsymbol{p}$ on the cylinder surface

if *theta distribution is NORMAL* **then**

  $\hat{\boldsymbol{d}} \leftarrow \hat{\boldsymbol{c}}$

else

  `// this section is illustrated in Fig 3.2`

  `// generate` $\hat{\boldsymbol{d}}'$ `assuming the plane normal is oriented on the positive z-axis`

  $\phi \leftarrow U_{0,2\pi}$

  if *theta distribution is HEMISPHERE* **then**

    $\theta \leftarrow \arccos U_{-1,1}$

  else if *theta distribution is LAMBERTIAN* **then**

    $\theta \leftarrow \arccos \sqrt{U_{0,1}}$

  `// convert to cartesian coordinates. Absolute value of z-component gives a hemisphere rather than a sphere`

  $\hat{\boldsymbol{d}}' \leftarrow (sin\theta sin\phi, sin\theta cos\phi, |cos\theta|)$

  `// Rotate the hemisphere (and the random point) by centering the z-axis along the cylinder surface normal`

  $\hat{\boldsymbol{d}} \leftarrow$ rotate $\hat{\boldsymbol{d}}'$ vector to align with $\hat{\boldsymbol{c}}$

3.1.2 Mesh Region Light Source

As discussed in Section 2.3.2, BLI involves transfecting a group of diseased cells, like a tumor, with a virus that alters their genes and causes them to emit light. Since the transfected cells are the sources of light and also part of the mesh (a region or regions in the mesh), supporting BLI requires creating an arbitrary light source from volumetric regions of the mesh. In IPDT, light sources are implanted into the patient and therefore, in reality, have their own region in the mesh with optical properties that can affect the light propagation. The sources in Section 2.4.2 passively emit light in the simulation; they are not part of the geometry and therefore do not affect the propagation of the light once it is emitted. Therefore, to explore the effect of light emitters with their own region and optical properties, it is necessary to include them as regions in the mesh and emit light from the volume or surface of that region.

A 2D illustration of the mesh region light source is shown in Fig 3.4. The user selects the region of the mesh to emit light from and we perform pre-processing steps to determine the tetrahedrons that belong to that region and which have faces on the surface of the region. For example, in Fig 3.4 four faces of the triangles are on the surface of the square shaded region. The user can choose whether to emit light from only the surface of the region (Fig 3.4, middle) or from within the region (Fig 3.4, right). If the user chooses to emit from the surface of the region, they can choose one of the three theta angle distributions discussed in the previous section, as shown in Figure 3.1. Algorithm 2 describes how we emit light isotropically from the volume of a region in the mesh, while Algorithm 3 describes the more complicated method of emitting light from the surface of a region using different theta angle

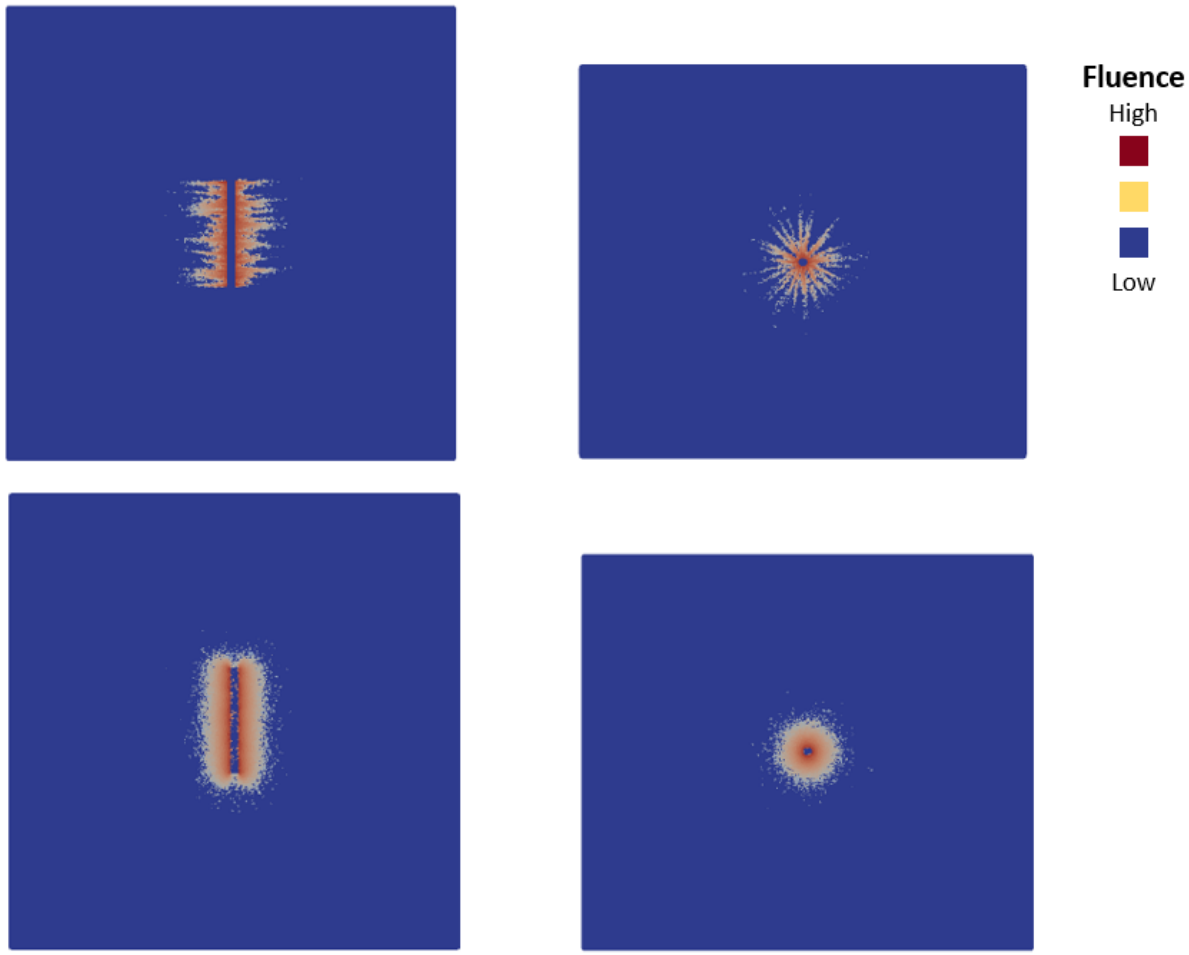

Figure 3.3: The side (left) and top (right) view of a virtual cylinder source with surface normal emission (top) and Lambertian emission (bottom)

distributions. Fig 3.5 shows an example of the mesh region light source. The left image shows the two regions in the mesh (grey cylinder within a red square) and the right image shows the resulting fluence distribution using a method similar to that in the previous section (high absorption in the surrounding square).

Algorithm 2: Emitting from Mesh Region Volume

Data: List of tetrahedrons in region R

Result: An initial position $\boldsymbol{p}$ and direction $\hat{\boldsymbol{d}}$ for the packet

$T \leftarrow$ randomly choose a tetrahedron in the region R distributed by volume

$\boldsymbol{p} \leftarrow$ generate a random point inside the volume of T

$\hat{\boldsymbol{d}} \leftarrow$ generate a direction vector with isotropic distribution

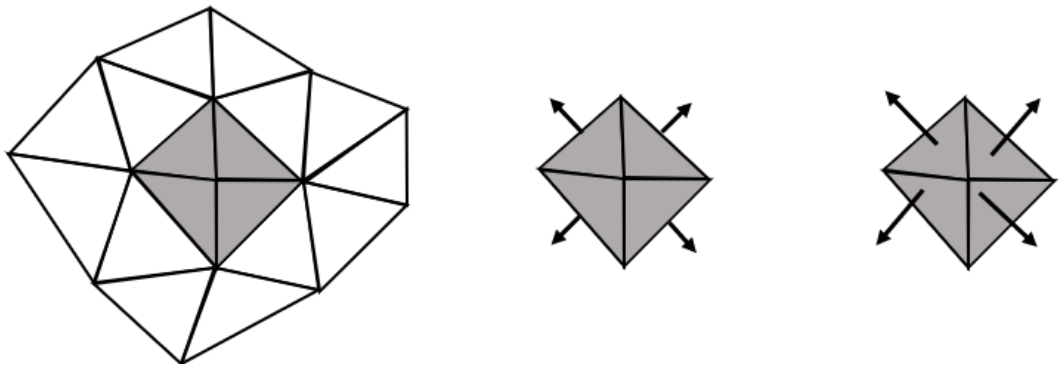

Figure 3.4: Illustration of a mesh region light source with the mesh on the left, the isolated surface light source in the middle and the isolated volume light source on the right

Algorithm 3: Emitting from Mesh Region Surface

Data: List of tetrahedrons in region R

Result: An initial position $\boldsymbol{p}$ and direction $\hat{\boldsymbol{d}}$ for the packet

$F \leftarrow$ randomly choose a tetrahedral face on the region R distributed by surface area

$\boldsymbol{p} \leftarrow$ generate a random point on F

if *theta distribution is NORMAL* **then**

  $\hat{\boldsymbol{d}} \leftarrow$ the normal vector of F

else

  `// this section is illustrated in Fig 3.2`

  `// generate` $\hat{\boldsymbol{d}}'$ `assuming the plane normal is oriented on the positive z-axis`

  $\phi \leftarrow U_{0,2\pi}$

  if *theta distribution is HEMISPHERE* **then**

    $\theta \leftarrow \arccos U_{-1,1}$

  else if *theta distribution is LAMBERTIAN* **then**

    $\theta \leftarrow \arccos \sqrt{U_{0,1}}$

  `// convert to cartesian coordinates. Absolute value of z-component gives a hemisphere rather than a sphere`

  $\hat{\boldsymbol{d}}' \leftarrow (sin\theta sin\phi, sin\theta cos\phi, |cos\theta|)$

  `// Rotate the hemisphere (and the random point) by centering the z-axis along the cylinder surface normal`

  $\hat{\boldsymbol{d}} \leftarrow$ rotate $\hat{\boldsymbol{d}}'$ vector to align with the normal vector of F

3.2 Improving the Performance of Tetrahedral Mesh Queries

A common operation in both the light propagation simulation itself and post-processing queries for applications like PDT treatment planning involves determining which tetrahedron element in the mesh contains a given point $\boldsymbol{p}$. For example, finding the tetrahedron containing the random point generated on or in the 3D volume of the cylindrical diffuser described in Section 3.1 is necessary to launch a photon packet in simulation. These operations can be expensive, especially for real medical applications with large meshes. In this work, we optimized this query for the general case by using an RTree data

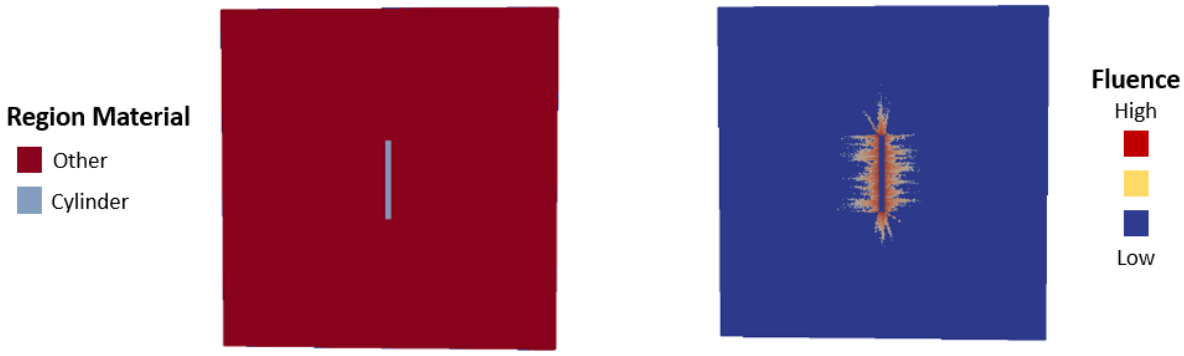

Figure 3.5: A mesh region light source with the regions (left) and the emission profile using a surface normal distribution (right)

structure. The following section provides a brief discussion on how this data structure works and how it is used in the context of our work to improve the performance of the simulator.

3.2.1 Point to Containing Tetrahedron Lookup using RTrees

The goal of the query is simple: given a point $\boldsymbol{p}$ inside the boundaries of the mesh, find the tetrahedron that contains it. Given a tetrahedron T, the normal vector of each of its faces $\hat{\boldsymbol{n}}_i$ (which by convention, are directed towards the inside of the tetrahedron) and a constant point $\boldsymbol{c}_i$ anywhere on each face, we can determine if $\boldsymbol{p}$ is within T using Algorithm 4. The height h_i, is the distance from 2D plane made by the ith tetrahedral face to the point $\boldsymbol{p}$. Since each face normal vector $\hat{\boldsymbol{n}}_i$ points towards the inside of the tetrahedron, if the distance (h_i) from the face to the point $\boldsymbol{p}$ is positive for all faces, we know the point is within the tetrahedron. Contrarily, if the distance is negative for any face, then the point is outside of the tetrahedron. An example of both cases is illustrated in Fig 3.6 for a single face of a triangle.

Algorithm 4: Check if point is inside tetrahedron

Data: Tetrahedron T defined by face normals and constants $(\hat{\boldsymbol{n}}_i, \boldsymbol{c}_i)$ and query point $\boldsymbol{p}$
Result: Whether $\boldsymbol{p}$ is within the tetrahedron T
for $i \leftarrow 0$ **to** 3 **do**
  $h_i \leftarrow \hat{\boldsymbol{n}}_i \cdot (\boldsymbol{p} - \boldsymbol{c}_i)$
return true if $h_i \geq 0, \forall i \in [0, 3]$, otherwise false

The trivial solution for finding the tetrahedron containing a point is to check every tetrahedron in the mesh, performing Algorithm 4 for each. This query can be performed millions of times during a simulation (e.g. every time a new photon packet is launched) and also when implementing more complicated applications like light source placement in PDT treatment planning. In this work, we significantly improved the performance of this query by using an *RTree* datastructure.

The RTree datastructure hierarchically partitions a set of objects to reduce search time. In our case, the *space* is $\mathbb{R}^3$, over which we wish to partition the tetrahedrons. This changes the time complexity of the query from linear ($O(n)$) to logarithmic ($O(logn)$), where n is the number of tetrahedra in the mesh; that is, we only consider a logarithmic number of tetrahedrons, rather than checking every tetrahedron. An example RTree structure for $\mathbb{R}^2$ is shown in Fig 3.7. The left figure shows the spatial view of the partitioning, while the right figure shows an instance of the RTree datastructure. If we wish to find

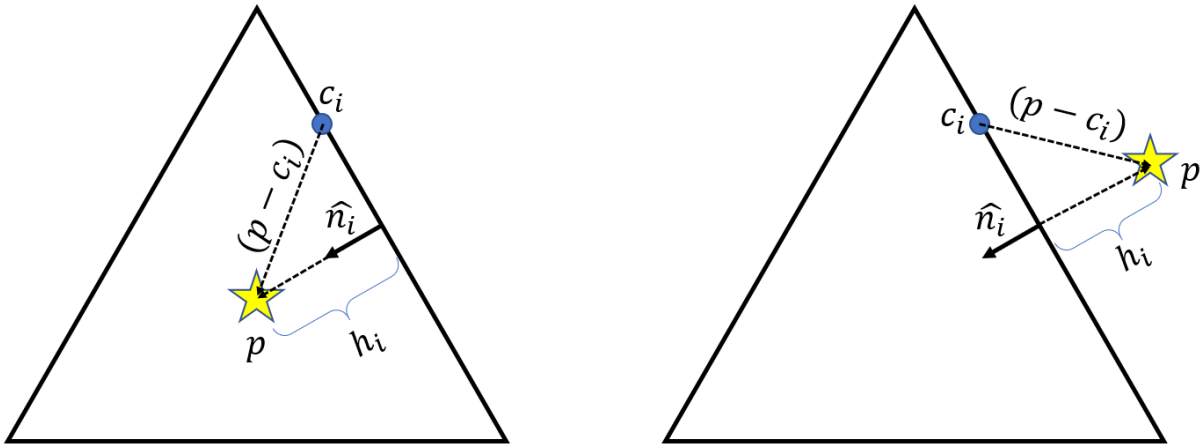

Figure 3.6: Plane-to-point distance calculation for a point inside (left) and outside (right) of a triangle.

the object containing some point, we move down the tree on the right of Fig 3.7 and check if the point is contained in each of the partitions until we find the object (or a set of possible objects) that *could* contain the point (L in Algorithm 5). For example, if the query point is the star in Fig 3.7, the RTree would return the list $L = \{C, E\}$, since the point is contained in both rectangles. In our application, L contains all of the tetrahedra whose bounding boxes contain the query point. However, being contained in the bounding box of a tetrahedron does not mean the point is in the tetrahedron itself. To find the tetrahedron that contains the query point, we do a linear search over the relatively small (∼10-100 tetrahedra) list L and determine which tetrahedron contains the point using Algorithm 4. Algorithm 5 summarizes how we use the RTree to find the tetrahedron containing point $\boldsymbol{p}$.

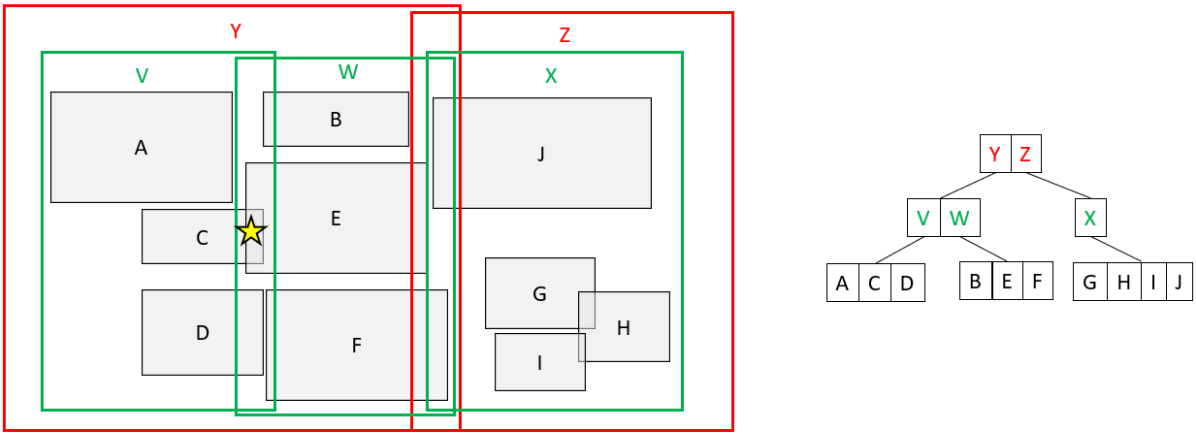

Figure 3.7: 2D object partition (left) and corresponding RTree (right)

Algorithm 5: RTree Query

Data: Set of tetrahedrons T, RTree R and query point $\boldsymbol{p}$

Result: The tetrahedron containing $\boldsymbol{p}$

$L \leftarrow$ query R for point $\boldsymbol{p}$

$t \leftarrow$ linearly search L for tetrahedron containing $\boldsymbol{p}$ (uses Algorithm 4)

return t

Our RTree approach results in checking significantly less tetrahedrons when performing the query. To verify the RTree's accuracy and measure its performance, we randomly generate a set of 200 points in a mesh with ≈1.1M tetrahedrons, which is a typical number from our benchmark meshes. For each random point, we perform both a linear and RTree lookup and ensure that both queries return the same tetrahedron (the linear query has already been rigorously verified). To benchmark performance, we measure the latency of each query. In total, the 200 linear and RTree queries take 18427ms and 115ms, respectively. The average linear query latency is 92ms while the average RTree query latency is 0.4ms. In summary, the RTree query achieves over a 230x speedup over the linear query. Due to Amdahl's law [6], the RTree's effect on the overall performance of a simulation depends on the total time spent launching packets (the only place where the point-to-tetrahedron lookup is used), which can vary across models and simulation parameters. To quantify the RTree's effect on the simulators performance, we use a realistic model with 21 million tetrahedra and a single cylinder light source. We found that the RTree lookup resulted in a 10x speedup in total simulation time over the naive linear lookup.

Chapter 4

Hardware Acceleration

In this chapter we discuss two methods for accelerating the FullMonteSW simulator using a GPU and FPGA: FullMonteCUDA and FullMonteFPGACL, respectively. We begin this chapter with some motivation for exploring different acceleration methods. We then provide a high-level discussion on the design choices for hardware acceleration, in particular, how to integrate the accelerated simulator into the existing project. Next, we discuss how we validate the accuracy of our hardware accelerated simulators using the software simulator as a baseline. The final sections describe the details of our accelerated implementations and compare the performance against other state of the art simulators.

4.1 Motivation

Improving the performance of our software simulator further can improve the quality of automated PDT treatment planning [66] or the quality of the images created by BLI and DOT by running more simulations in an allocated amount of time. As discussed briefly in Section 2.6 and in [20, 21], FullMonteSW utilizes multithreading, hyperthreading, optimized datastructures and hand-coded vector instructions. These optimizations make FullMonteSW the fastest and most full-featured MC biophotonic light propagation simulator written in software [20]. Additional software optimizations show diminishing performance improvements, which influences the investigation of other acceleration techniques. The following sections discuss the motivation for using a GPU and FPGA for hardware acceleration.

Why a GPU?

As discussed in Section 2.5, a typical CPU contains a small number (∼1-12) of fast and flexible computing cores that operate independently. In comparison, GPUs contain many (∼700-4000) simpler computing cores with subsets that work in lock-step. The cost of the simplicity and massive parallelism in a GPU is that a single core operates slower than a single CPU core. However, for applications, like ours, that have significant *data parallelism* (i.e. perform the same operation on many pieces of data), the GPU can improve throughput. These factors influence us to accelerate the algorithm on a GPU for two main reasons. First, we hypothesize that the development difficulty will be much lower than other acceleration techniques, like an FPGA. We also hypothesize that creating an initial prototype will be relatively simple since the CPU and GPU code will be very similar in structure. We believe that most of the programming effort will be abstracting the acceleration properly to make it transparent to the user and making GPU

specific optimizations to improve performance. Second, as discussed in Section 2.4.4, the Monte Carlo algorithm is highly parallel since the path of a packet has no effect on the path of any other packet. Therefore, packets may be launched completely in parallel, even on different devices (i.e. a cluster of GPUs), so long as the energy accumulation arrays are summed together once the simulations finish. This characteristic of the algorithm maps naturally the GPU architecture: each thread is assigned a single packet to propagate through the mesh until it loses roulette or exits the mesh.

Why an FPGA?

As discussed in Section 2.5.3, the FPGA uses a different computation technique than CPUs and GPUs. The spatial computation performed by FPGAs can be more difficult for the developer to describe than the computation for CPUs and GPUs. Along with the relatively difficult programming paradigms for FPGAs, we believe that implementing the algorithm on an FPGA will be a more difficult task than the GPU. However, the algorithm has various attributes that make FPGAs an attractive choice for acceleration. As discussed in the previous section, there is ideal data parallelism in the Monte Carlo algorithm and packets may be launched completely in parallel. In addition, the packets exhibit significant pipeline parallelism, since the stages in the computation are mutually exclusive between packets. In Fig 2.6 each circle could be a pipeline stage connected by logic that moves packets from one stage to another. For example, one packet could be in the DRAW STEP stage, another in the HOP stage and another in the DROP stage and these operations could be performed and the packets moved to the next stage in parallel since they are mutually exclusive. In this simple example, we are computing the propagation of three packets at the same time. As will be discussed in Section 4.7, we plan to construct a pipeline like this but at a much finer level with significantly more pipeline stages to extract even more parallelism. This technique will allow for a deep pipeline with no stalls and a high operating frequency that will increase packet throughput and therefore the simulation performance. In addition, given sufficient FPGA resources, we can duplicate this pipeline to fill the FPGA and, due to the various degrees of parallelism, simulate even more packets in parallel.

Another decision for the FPGA design is the programming language we use to describe the algorithm. We considered using Verilog HDL, C++ HLS and OpenCL HLS to describe the algorithm. As discussed previously in Section 2.5.3, each choice has its own productivity-performance tradeoffs. Based on experiences and difficulties with implementing the algorithm in lower-level RTL languages, we will use the Intel FPGA SDK for OpenCL (version 18.1). The challenge with using OpenCL is describing the algorithm in a way that the compiler (*AOC* in the case of the Intel OpenCL SDK for FPGAs) is able to generate a functioning pipeline without sacrificing substantial performance and area. Section 4.7 will discuss the strengths and weakness we found with using the FPGA SDK for OpenCL and how it could be enhanced to better support applications like ours.

Table 4.1 summarizes the compute architectures used in this work. We tried to the best of our abilities to fairly compare our implementations against one another, and against other simulators, by accounting for the differences in the hardware (e.g. process generation, price, power, etc).

4.2 Design Choices

Before developing the hardware accelerated simulators, we set some design goals which are summarized in the following list:

Table 4.1: The different compute architectures used in this work

Hardware	Brand/Name	Year	Process Node	Details
CPU	Intel Core i7-6850	2016	14nm	**Clock Rate:** 3.8 GHz **Cores:** 6 (12 threads) **RAM:** 32GB **Vector instructions?** AVX2 **TDP:** 140W
GPU	NVIDIA Quadro P5000	2016	16nm	**Clock Rate:** 1.607GHz **SMs:** 20 **SPs/SM:** 128 **Board Memory:** 16GB **TDP:** 180W
GPU	NVIDIA Titan Xp	2017	16nm	**Clock Rate:** 1.405GHz **SMs:** 30 **SPs/SM:** 64 **Board Memory:** 12GB **TDP:** 250W
FPGA	Terasic DE10-Pro (Stratix 10)	2019	14nm	**LEs:** 2800K **DSPs:** 5760 **RAMs:** 229 Mbits **DDR (on board):** 32GB

1. Give statistically correct results across various benchmarks
2. Make the acceleration transparent to the user
3. Gather the same output data as the software simulator
4. Support **all** light sources supported by the software
5. Achieve a 4x performance improvement or better

These goals require important decisions to be made about the design of the simulator and careful planning before implementation. The next section will discuss our method of validating the accuracy of the hardware accelerated simulation results. The software simulator is comprised of complex and efficient C/C++ code that is inherently difficult to use, modify and extend. The simulator is used predominately by medical researchers who may lack the knowledge or time to use complex C/C++ APIs. Therefore, to simplify the usage of the simulator without limiting its flexibility, we create a TCL scripting interface [58, 20]. The hardware accelerated simulators should be supported by this scripting interface and the hardware acceleration should be made transparent to the user by ensuring inputs from and outputs to the user are the same (or as similar and simple as possible) as the software simulator. This requires additional pre- and post-processing steps to convert the input data to target the specific acceleration device and gather the output data from the device and process it accordingly. A major limitation of existing hardware accelerated simulators [1, 19] is the lack of support for complex light sources. As discussed in Section 2.3, accurately representing the complex light sources in simulation is vital to the accuracy of DOT, BLI and PDT. Therefore, our goal is for the hardware accelerated simulators to support the same set of light sources as the software. Lastly, we wish to achieve at least a 4x speed improvement over the already highly optimized software simulator, thereby matching the estimated acceleration of

the RTL implementation [19]. This would exceed the hypothetical performance improvement estimated in [19] and the performance achieved by [1] on voxel geometries.

4.3 Validation of Output

FullMonteSW uses the TinyMT random number generator (RNG) [57] which is *pseudorandom*; that is, given the same simulation parameters (mesh, material optical properties and number of packets to simulate), multiple runs of the simulation will produce identical output. However, even when these simulation parameters are kept constant, it is still possible for differences in the outputs to arise. For example, a different initial RNG seed, number of threads or underlying compute architectures cause discrepancies between simulations with the same parameters. This means that the output from our accelerated simulators may be different than the output of the software baseline, even though the simulation parameters are identical. Therefore, we must develop a method to accurately compare the output from our accelerated simulators against the output from the software baseline.

The *normalized L1-norm* ($|\hat{x}|_1$) of two simulations, A and B, can be computed using Equation 4.1, where $\Phi_A(i)$ and $\Phi_B(i)$ are the fluences in tetrahedron i from simulations A and B, respectively. To compare our accelerated simulators against the software baseline, we first perform two software simulations with different RNG seeds and compute the normalized L1-norm. Since the only parameter changing is the RNG seed, this gives us a measure of the random noise for the specific simulation. Next, we perform the same simulation using one of our accelerated simulators and compute the normalized L1-norm, where simulation A is one of the software simulations, and B is that from the accelerated simulator. We then compare the two normalized L1-norm values; if these numbers are similar, we can conclude that the difference between the software simulation and accelerated simulation is comparable to random noise. If the difference between our accelerated simulations and the baseline is comparable to random noise, then our accelerated simulators are producing statistically correct output since the software simulator has already been validated [20]. Tables 4.4 and 4.9 summarize the normalized L1-norm values of the various benchmarks from Table 4.2 to validate the accuracy of our accelerated simulators.

$$|\hat{x}|_1 = \frac{\sum_i |\Phi_A(i) - \Phi_B(i)|}{\sum_i |\Phi_A(i)|} \tag{4.1}$$

4.4 Packet Launch

Our goal for the hardware accelerated simulators is to support every light source supported by the software simulator. This can be achieved one of two ways: by implementing the complex packet launching code in the hardware accelerators or by using the already existing software code, generating the launched packets in the CPU and transferring the launched packet data to the hardware accelerator. As shown in Fig 2.6, a packet is launched once during the simulation (the LAUNCH stage) and undergoes thousands of scattering and absorption events before either dying in *roulette* or exiting the mesh. Thus, the acceleration of the LAUNCH stage is not crucial to the performance of the simulator, due to Amdahl's law [6]. Therefore, for the hardware accelerated versions, we compute the launching of packets in software before starting the simulation and have the LAUNCH stage in the hardware accelerated simulators read the packet from memory. This allows the hardware accelerated versions to instantly support all the light

Table 4.2: Models used for the validation and benchmarking of accelerated simulators

Model	Tetrahedrons	Light Sources	Materials ($\mu_s[mm^{-1}]$, $\mu_a[mm^{-1}]$, g, n)[2]
HeadNeck[1]	1088680	Isotropic Point, Pencil Beam, Fiber Cone	tongue (83.3, 0.95, 0.93, 1.37) tumour (9.35, 0.12, 0.92, 1.39) larynx (15, 0.55, 0.9, 1.36) teeth (60, 0.99, 0.95, 1.48) bone (100, 0.3, 0.9, 1.56) tissues (10, 1.49, 0.9, 1.35) fat (30, 0.2, 0.78, 1.32)
HeadNeckTumour[1]	8944	Isotropic Point	air(0.0, 0.0, 0.0, 1.0) tumour (9.35, 0.13, 0.92, 1.39)
Bladder[1]	1706958	Isotropic Point, Pencil Beam, Volume, Ball, Line, Fibre Cone	air (0, 0, 0, 1.37) urine (0.1, 0.01, 0.9 1.37) surround (100, 0.5, 0.9, 1.39)
cube_5med [59]	48000	Isotropic Point	mat1 (20, 0.05, 0.9, 1.3) mat2 (10, 0.1, 0.7, 1.1) mat3 (20, 0.2, 0.8, 1.2) mat4 (10, 0.1, 0.9, 1.4) mat5 (20, 0.2, 0.9, 1.5)
FourLayer [59]	9600	Pencil Beam	layer1 (10, 0.05, 0.9, 1.3) layer2 (30, 0.1, 0.95, 1.5) layer3 (10, 0.05, 0.9, 1.3) layer4 (30, 0.1, 0.95, 1.5)

[1] Material optical properties extracted from literature [7, 9, 10, 17, 23, 51, 60, 64]
[2] Material properties: scattering coefficient (μ_a), absorption coefficient (μ_s), anistropy (g) and refractive index (n)

sources already implemented in software and reduces the development time and effort required to add new light sources or modify existing ones. Sections 4.6 and 4.7 will describe the specifics of how this was done for the GPU and FPGA accelerated versions, respectively.

4.5 Hardware Accelerator Integration

To achieve our goal of transparent acceleration, we integrate the hardware accelerators into the existing software project using the method shown in Fig 4.1. The user can either allow the simulator runtime to choose which hardware to use, or request for the simulation to be run on a CPU, GPU or FPGA using a set of TCL scripting commands. This method of integration results in maximal code reuse. As shown in Fig 4.1, the code to process input, start the simulation and produce output is shared across each implementation. The accelerators have a small amount of pre- and post-processing code to convert to and from a format suitable for the specific device. As discussed in the previous section, this also allows each simulator to use the same code for launching packets (part of the generic pre-processing step in Fig 4.1).

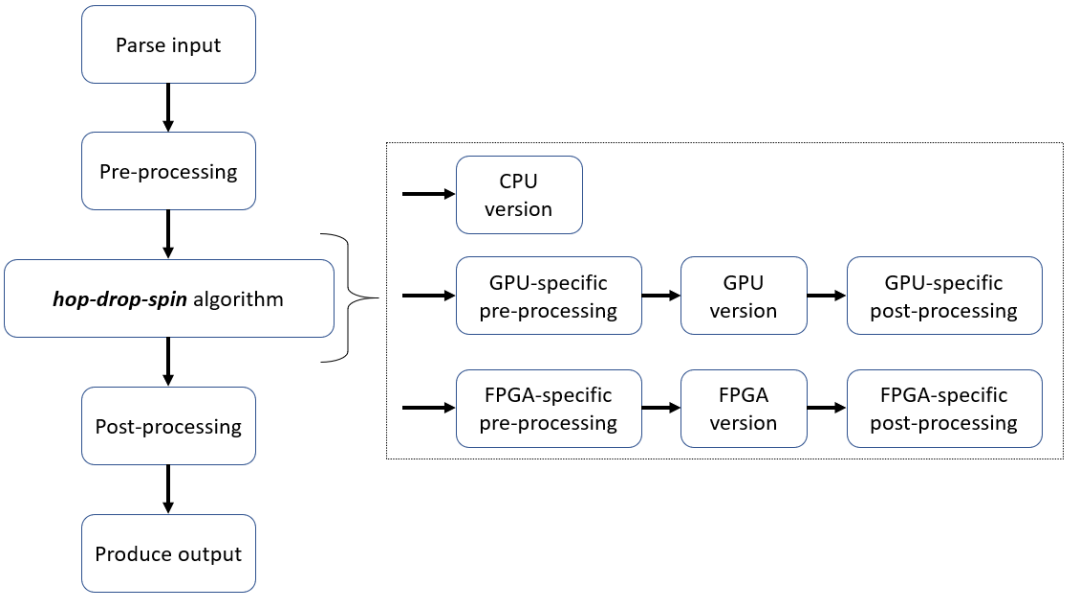

Figure 4.1: Integration of FullMonteCUDA and FullMonteFPGACL into existing software project.

4.6 GPU Implementation: FullMonteCUDA

This section discusses how we implement the FullMonteSW algorithm using an NVIDIA GPU. Specifically, we used both an NVIDIA Quadro P5000 and NVIDIA Titan Xp GPU in this work. For the software baseline and GPU host code we used an Intel Core i7-6850 3.8GHz CPU with 6 physical cores (12 virtual cores with hyperthreading), vector instructions enabled and 32GB of RAM. This section begins with a discussion of how the FullMonteSW code was translated to a GPU kernel and various optimizations we made to better tailor the algorithm to the GPU architecture. We then discuss the accuracy and performance results of our GPU accelerated simulator, FullMonteCUDA. We conclude this section with an analysis of the GPU code to identify the performance bottlenecks.

4.6.1 Design Overview

Since FullMonteSW is a multithreaded application, it is not complex to convert it to GPU code. The software code has a function which performs the simulation for a single packet. To implement the GPU kernel code, we first copy this function (and all sub-functions) nearly line-for-line to CUDA code. This conversion is simple since CUDA is a C++ based parallel programming language. We also create the CUDA host code that coordinates the GPU accelerator. The software uses a small number of threads ($N \approx [1-24]$) to perform the computation of many packets ($P \approx [10^6, 10^7]$). Thus, each software thread is assigned $\lceil P/N \rceil$ packets to simulate. In comparison, each thread in the GPU kernel simulates one packet. However, the GPU has limits on the maximum number of threads per simulation [49]. Therefore, launching all photon packets may require multiple invocations of the kernel based on the maximum number of threads and global memory size of the GPU. To improve the usability, the GPU accelerator automatically determines the number of packets to launch and makes multiple asynchronous kernel invocations until all of the packets have been launched. After all of the GPU kernels have been launched, the host can either wait for them to finish or continue performing other computation. Once

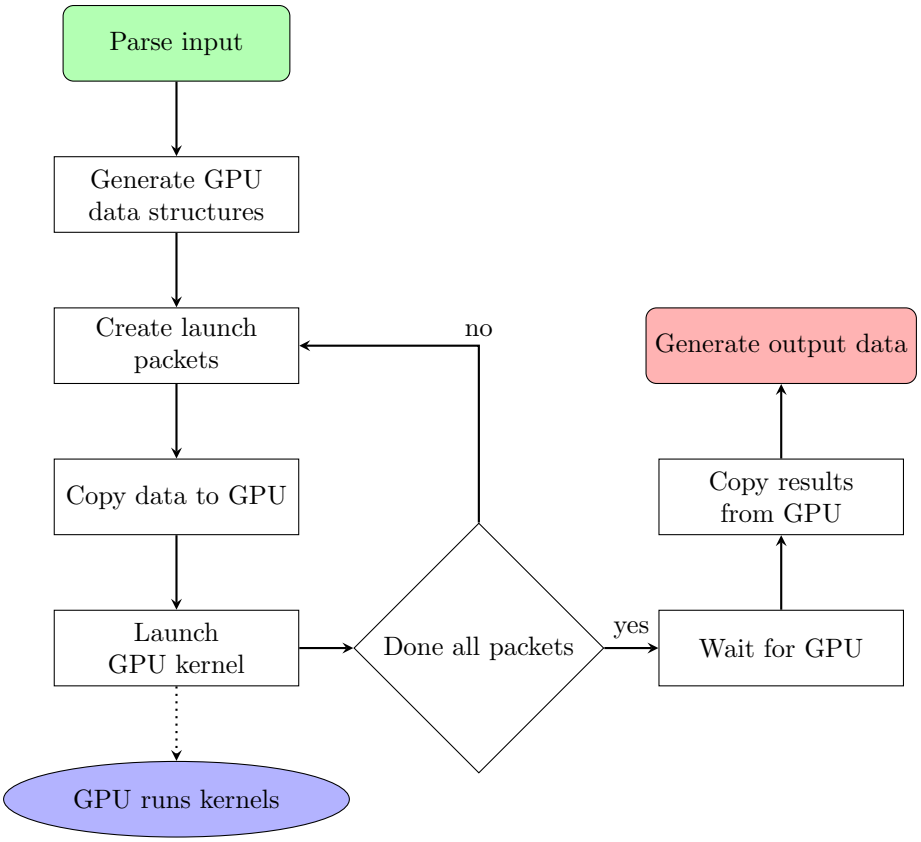

Figure 4.2: The host code of FullMonteCUDA

all of the kernels have finished executing in the GPU, the CPU host code copies the output data from the GPU memory into its own memory and generates the output files using the same methods as the software simulator.

The host logic for a single simulation is shown in Fig 4.2. The integration of FullMonteCUDA into FullMonteSW was crucial for various reasons. As discussed earlier in this chapter, the integration allowed the GPU accelerator to automatically support all of the light sources implemented in the software. More generally, since the input parsing, output producing and light source code is shared with the software simulator, any modifications, additions and optimizations to this code are instantly usable by both the CPU and GPU simulators. This drastically improves the usability, code readability and development productivity of the GPU accelerator.

Optimizations

In the ideal scenario all GPU threads are constantly performing computations and therefore the device is running at maximum capacity. However, this ideal case is almost never attained as threads are sometimes stalled waiting for data to be ready. For FullMonteCUDA, the most significant reasons for stalls are *memory dependencies* and *execution dependencies*. A memory dependency occurs when accessing memory, causing the GPU to stall until the request is complete. The length of such a stall can vary depending on where the memory resides, as shown in Table 2.1. An execution dependency occurs when an instruction depends on the result of a previous instruction, causing the GPU to stall until the

first instruction is finished.

We use the *NVIDIA Visual Profiler* (NVVP) to identify bottlenecks in the code and more accurately measure the impact of optimizations. Fig 4.3 shows profiling results for the final implementation of the algorithm on the *HeadNeck* mesh from Table 4.2 using 10^6 packets. The chart illustrates the distribution of reasons for kernel stalls and helps pinpoint latency bottlenecks [50]. It shows that the largest number of stalls in the fully optimized implementation are due to memory dependencies. This is typical for applications like ours that have large datasets and unpredictable memory access patterns. In Fig 4.3, the *other* kernel stalls can be caused by issues like instruction fetching, instruction issuing, memory throttling and more. For more information see NVIDIA's *NVVP User Guide* [50].

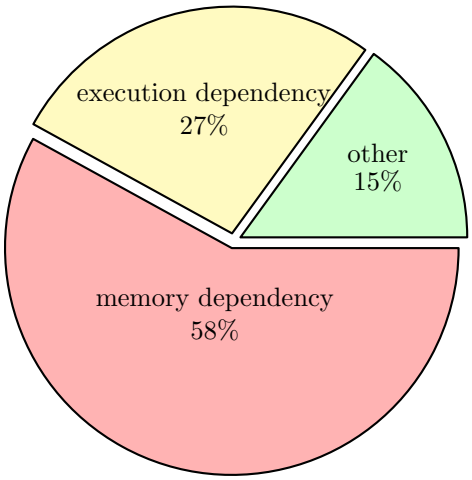

Figure 4.3: NVVP *PC Sampling* data for the final GPU implementation using the Titan Xp NVIDIA GPU [67]

The naive implementation of the GPU kernel described in the previous section, which was nearly identical to the software code, achieved a speedup of 2x over the software but performed sub-optimally on the GPU. We implement a series of optimizations which better tailor the algorithm to the GPU architecture. Table 4.3 summarizes the performance results for some of these optimizations and the following sections discuss them in more detail.

Table 4.3: Performance increase for each FullMonteCUDA optimization over FullMonteSW using the NVIDIA Titan Xp GPU

Optimization	Incremental Speedup
Naive	2x
CUDA vector datatypes and math operations	2.5x
Materials constant cache	1.6x
Thread local accumulation cache	1.3x
Total	**10.4x**

Launching packets in the host FullMonteSW supports a wide range of light sources which require many conditional statements. As discussed previously, we chose to compute packet launches in the CPU host and send the data to the GPU kernel before starting the simulation. For the GPU specifically, this

decision was made for two reasons. First, it allows the same complicated light emitter code to be used whether GPU-acceleration is being used or not. This reduces the development time and effort required to add new sources or modify existing ones. Second, the code that launches packets is highly divergent, which does not map well to GPUs [15]. This would result in increased *thread divergence* and decreased performance.

Vector datatypes and math operations CUDA provides native vector datatypes (e.g. `float2`, `float4`, `uint4`, etc.) [49] which are similar to the vector datatypes used in the software simulator. These are C-style `structs` with specific memory alignment requirements. The alignment and support in the GPU for these special datatypes make them particularly efficient when accessing the entire vector at once. Therefore, we group logical sets of data, for example positions, direction vectors and material optical properties, into vector datatypes to improve memory performance.

As discussed in Section 2.4.4, the hop-drop-spin algorithm performs many complicated calculations that rely heavily on the use of floating-point mathematical operations (e.g. `recip`, `cos`, `sin`, `dot`, `cross`). Therefore, we investigated the use of variants and approximations to these functions and various CUDA compiler flags to improve their performance. When modifying the functions and flags, we monitor the change in performance and ensured the accuracy of the result using the method discussed earlier in Section 4.3. We determine that it is safe to use the `use_fast_math` flag to turn on all CUDA fast-math approximations without sacrificing accuracy. Together, the use of vector datatypes and fast math approximations results in a $\sim$2.5x speed improvement, as shown in Table 4.3.

Constant materials caching Typical medical applications have a small number of materials in the model (≈ 5) and these properties do not change over the course of the simulation. Therefore, we store them in the constant memory of the GPU (Table 2.1) to improve their read latency. A single set of material properties requires 72 bytes of storage, which we pad to 128 bytes to align them to a 128 byte boundary (the cache line size of the GPU) to further improve memory performance. Therefore, the maximum number of materials supported by FullMonteCUDA is bounded by the size of the GPU's constant cache. All of the currently supported NVIDIA GPUs have a 64kB constant cache which can store up to 500 materials. This is sufficient for most practical medical applications. Table 4.3 shows that the storage of material properties in the constant cache improves the performance by almost 2x. This performance improvement is due to a significant reduction in the number of memory and execution dependency stalls, as shown in Fig 4.4.

Local accumulation buffers During the DROP stage of the propagation algorithm (Fig 2.6), the packet drops some of its weight into the tetrahedron it currently resides in. A read-accumulate-write operation is performed for that tetrahedron entry in the accumulation array to model photon absorption. To preserve data consistency, this accumulation uses an atomic operation to read the current energy in the tetrahedron, add to it and then write it back. However, atomic operations can be computationally expensive as they will stall other threads trying to update the same tetrahedron, which stalls the entire warp the thread belongs to. In the DRAW STEP stage, the generated step length depends on the attenuation coefficient of the material the packet currently resides in, as shown in Equation 2.6. Depending on the *granularity* of the mesh (i.e. the size of the tetrahedrons) relative to the attenuation coefficient (and therefore the range of step lengths generated by Equation 2.6), packets may take several steps within a single tetrahedron before moving to the next. We use this information to create a custom cache for

each thread to store accumulations locally and avoid excessive atomic *read-accumulate-write* operations to global memory. We implement this by using the local memory of the GPU to store an array of tetrahedron IDs and the currently accumulated energy for that tetrahedron, for each thread. When a packet drops energy into a tetrahedron, the local accumulation array is scanned for the current tetrahedron using the ID. If the value is currently cached in local memory, then the accumulation happens locally, otherwise the *least recently used* entry in the cache is written back to global memory and the current tetrahedron replaces it in local memory. We tested caches of size 1-32 entries and found that all configurations improved performance but a single-entry is optimal. This is because the time to linearly scan the cache and perform LRU eviction logic outweighed the benefit of a slightly higher hit rate. We also tried implementing a directed mapped cache with up to 32 entries. For our application, the single entry cache performed better than the directed mapped cache, for similar reasons as the LRU cache. The single entry cache results in an ∼30% speed improvement, as shown in Table 4.3. This is caused by a significant reduction in the number of memory dependency stalls, as shown in Fig 4.4.

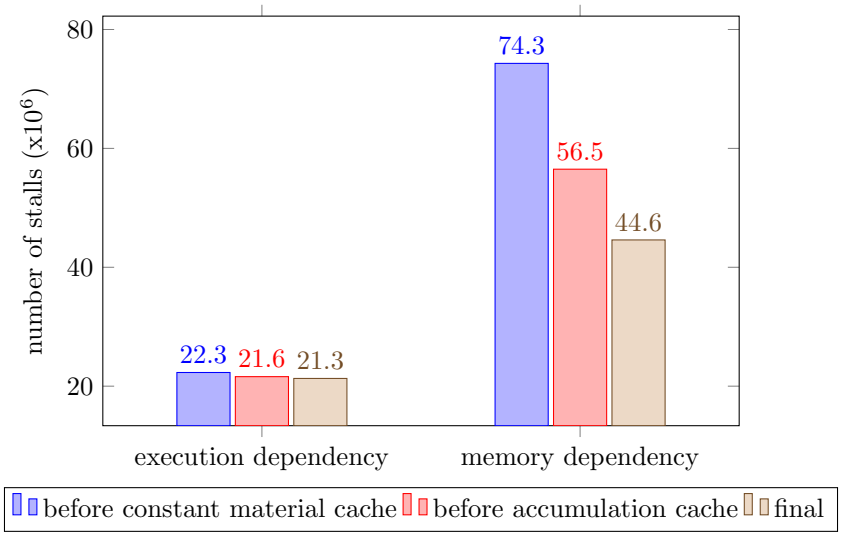

Figure 4.4: The number of execution and memory dependency stalls reported by NVVP before the constant material cache, before the accumulation cache and the final implementation with both the constant cache and the 1-entry accumulation cache [67]

4.6.2 Results

To validate the output from our FullMonteCUDA, we use the method described earlier in Section 4.3. Table 4.4 compares the normalized L1-norm values between two differently seeded software simulations and between a software and GPU-accelerated simulation. This table validates the output of our GPU-accelerated simulation by showing that the difference in output between FullMonteSW and FullMonteCUDA is comparable to random noise. Figure 4.5 shows the fluence plots for the *cube_5med* model using 10^8 packets with FullMonteSW (left) and FullMonteCUDA (right).

We benchmarked FullMonteCUDA against another GPU-accelerated tetrahedral-mesh light propagation simulator, MCtet. To do so, we used the same layered MCML models described in [70], as these are the only models for which MCtet's authors provide performance data. The results, summarized in Table 4.5, show that FullMonteCUDA achieves a speedup of 11x over MCtet. Even though FullMonte-

Table 4.4: Normalized L1-norm values for the models from Table 4.2 using 10^8 packets for two differently seeded CPU simulations (row 1) and a GPU and CPU simulation (row 2)

	HeadNeck	Bladder	cube_5med	FourLayer
FullMonteSW-FullMonteSW	0.0027	0.0322	0.0028	0.0012
FullMonteCUDA-FullMonteSW	0.0026	0.0342	0.0031	0.0020

(a) (b)

Figure 4.5: Output tetrahedral fluence plots of the *cube_5med* model (Table 4.2) for FullMonteSW (a) and FullMonteCUDA (b) using 10^8 packets.

CUDA was designed for far more complex geometries than these layered models, it still outperforms CUDAMCML which is tailored specifically for layered geometries.

Table 4.5: Performance results for CUDAMCML, MCtet and FullMonteCUDA using the MCML layered models from [70]

Model	CUDAMCML (s)	MCtet (s)	FullMonteCUDA (s)	
			Quadro P5000	Titan Xp
1-layer	27.1	103.4	23.7	16.8
3-layer	83.8	433.8	75.8	40.9

We also benchmarked FullMonteCUDA against state-of-the-art software simulators using more complex tetrahedral models, for which it was designed. We extended the performance analysis from [20] using the same compiler, compiler settings and models to compare FullMonteCUDA against TIM-OS, MMCM and FullMonteSW. We validated the output for 10^6 packets using the method described in Section 4.3 and found that the results from FullMonteCUDA agree with those from the other simulators. The results, summarized in Table 4.6, show that FullMonteCUDA achieves a 12x speedup over MMCM, up to 19x over TIM-OS and up to 11x over FullMonteSW. As discussed in [20], we were unable to reproduce the MMCM results for the *Colin27* mesh due to a reported bug in MMCM for the combination of simulation options necessary to accurately compare MMCM and FullMonteSW/FullMonteCUDA. Since FullMonteSW is 2.5x faster than MMCM on small meshes, we conservatively estimate that this will scale linearly to larger meshes. As discussed in the next paragraph, we measure FullMonteCUDA to be ∼10x faster than FullMonteSW on large meshes. Thus, we expect FullMonteCUDA to be ∼25x faster than MMCM on large meshes.

To further benchmark FullMonteCUDA against FullMonteSW, we use the complex geometry models

Table 4.6: Performance results for TIM-OS, MMCM, FullMonteSW and FullMonteCUDA

Model	TIM-OS (s)	MMCM (s)	FullMonteSW (s)	FullMonteCUDA (s)	
				Quadro P5000	Titan Xp
Colin27 [25]	34.1	—	19.8	2.7	1.8
Digimouse [59]	4.7	9.4	3.8	0.9	0.8

in Table 4.2 with packet counts ranging from 10^6 to 10^8. The results, summarized in Tables 4.7 and 4.8, show that FullMonteCUDA achieves a performance improvement between 4-13x across the various benchmark models. For each case, we verify the accuracy of the output using the method described in Section 4.3.

As shown in Tables 4.7 and 4.8, FullMonteCUDA performs better when simulating more packets. As illustrated in Fig 4.2, FullMonteCUDA has the additional overhead of transferring data between the CPU and GPU. To quantify this, we use the *HeadNeck* model from Table 4.2 and measure the overhead time and runtime change when launching more packets. We define the overhead time as the time required to calculate all the launch packets on the CPU and transfer the launch packet and tetrahedral mesh data from the CPU memory to the GPU memory. We find that the total runtime for 10^6 packets is 1.2 seconds. We measure the overhead time to be 0.1 seconds - roughly 8% of the total runtime. When the number of packets is increased 100x to 10^8, the runtime jumps to 31.8 seconds while the overhead time increases to only 0.15 seconds - roughly 0.5% of the total runtime. For inverse solvers, this overhead can be amortized across the thousands of simulations since the mesh, which represents nearly all of the memory being transferred, typically remains constant across the many forward simulation iterations and therefore only needs to be transferred to the GPU memory once.

Our results indicate that the GPU is capable of handling the large and complex tetrahedral meshes required for general clinical models. Moreover, as Table 4.2 and the performance results indicate, the GPU achieves equivalent or greater speedups for the larger mesh sizes. This shows that FullMonteCUDA is able to scale up to large and realistic clinical models and highlights the value of our GPU memory optimizations from Section 4.6.1.

Table 4.7: Performance comparison of FullMonteCUDA against FullMonteSW using 10^8 packets

Model	FullMonteSW (s)	FullMonteCUDA (s)		Speedup	
		Quadro P5000	Titan Xp	Quadro P5000	Titan Xp
HeadNeck	412.4	66.4	31.8	6x	13x
Bladder	1838.3	357.8	215.8	5x	9x
cube_5med	486.5	121.6	69.1	4x	7x
FourLayer	187.9	46.3	24.7	4x	8x

Table 4.8: Performance comparison of FullMonteCUDA against FullMonteSW using 10^6 packets

Model	FullMonteSW (s)	FullMonteCUDA (s)		Speedup	
		Quadro P5000	Titan Xp	Quadro P5000	Titan Xp
HeadNeck	5.1	1.5	1.2	3x	4x
Bladder	18.3	10.0	3.7	2x	5x
cube_5med	5.0	1.3	0.9	4x	6x
FourLayer	2.0	0.5	0.4	4x	5x

4.7 FPGA Implementation: FullMonteFPGACL

This section describes how we implement the FullMonteSW algorithm on a Terasic DE10-Pro board with an Intel Stratix 10 FPGA (1SG280LU2F50E1VG) and 32GB of external board memory. For the software baseline and FPGA OpenCL host code we use an Intel Core i7-6850 3.8GHz CPU with 6 physical cores (12 virtual cores with hyperthreading), AVX2 enabled and 32GB of RAM. This section begins with an in depth discussion on how the FullMonteSW code is translated into FPGA OpenCL kernels and various optimizations to better tailor the algorithm to the FPGA architecture. We discuss the FPGA resource utilization and the accuracy, performance and energy-efficiency of our final FPGA design. We conclude the section with a discussion on the development productivity from using OpenCL for FPGAs compared to other FPGA programming languages.

4.7.1 Design Overview

As we did for the GPU, we begin by performing a simple conversion from C++ CPU code to OpenCL code. Again, this conversion is rather simple due to the similarities between C++ and OpenCL. However, unlike the GPU, the software code does not map well to the FPGA and results in terrible performance on the FPGA, with an II of 512 and an Fmax of 120MHz. Thus, we take a different approach to the problem that breaks away from the typical OpenCL/CUDA programming paradigm, in which we re-coded the hop-drop-spin algorithm to guide the compiler to re-create the coarse hardware pipeline structure that we know is achievable. The pipeline in Fig 4.6 maps directly to the high-level hardware blocks of the FullMonteFPGACL pipeline and is a slightly modified version of the pipeline depicted in Fig 2.6. This section will describe how we implement the pipeline in Fig 4.6 on the Terasic DE10-Pro board. For the random number generation, we use a version of the TinyMT RNG from Saito and Matsumoto [57] which allows us to generate sixty-four 32-bit random numbers every clock cycle.

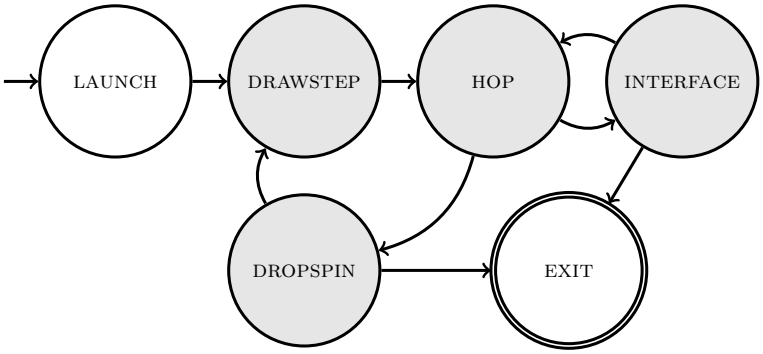

Figure 4.6: The core *hop-drop-spin* algorithm for FullMonteFPGACL

We implement the FullMonteSW algorithm by using OpenCL for FPGAs to describe a pipeline with the same structure as Fig 4.6. At a high level, we create blocks of sequential logic to perform each of the stages (the circles in Fig 4.6) and connect them with FIFO buffers to transfer data between the stages (the arrows in Fig 4.6). Within each stage, we implement complex logic to perform the light propagation calculations described in Section 2.4.4. We then create the CPU host code to parse inputs, generate outputs, transfer data to and from the FPGA board and coordinate the start and end of the simulation. Similarly to FullMonteCUDA, the integration of FullMonteFPGACL into the FullMonteSW

project greatly simplifies this step as the complicated input parsing and output generation is already implemented and we simply write code to convert to and from a format suitable for the FPGA. The entire FullMonteFPGACL system (CPU host and FPGA kernels) is illustrated in Fig 4.7 where the labelled arrows represent the following:

1. The CPU transferring simulation data to the FPGA.
2. The CPU signalling each of the kernels in the FPGA to start.
3. The launch kernel reading the pre-launched packets from the global memory of the board (this is described later in this section).
4. The dropspin kernel writing the on-chip tetrahedral energy accumulation arrays to global memory for the CPU to read.
5. The exit kernel on the FPGA signalling to the CPU when all of the packets have exited (i.e. the simulation is finished).
6. The CPU reading the output tetrahedral energy accumulation arrays from the FPGA to produce output.

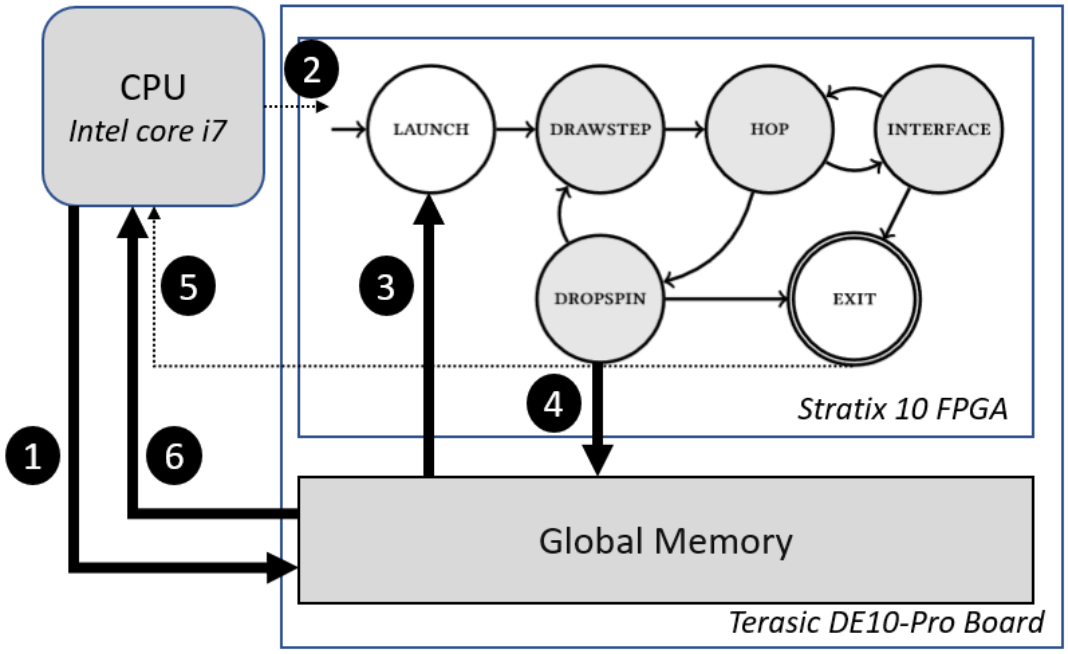

Figure 4.7: Summary of the FullMonteFPGACL system. Large arrows represent CPU-BOARD and FPGA-BOARD memory transfers, dotted arrows represent FPGA-CPU signals and circles represent OpenCL kernels with the arrows between them representing unidirectional channels.

The code to support many light sources is complex, and new sources are regularly added based on medical requirements and device advancements. As discussed in Section 4.4, we compute packet launches in the CPU host and send the data to the FPGA kernel before starting the simulation. As an emitted photon packet undergoes many interactions with the tissue before being terminated, most of

the computation is in the other stages of the pipeline in Fig 4.6, rather than in the LAUNCH stage. We therefore create an array of pre-launched packets in the CPU that is passed to the FPGA before the simulation starts. The OpenCL standard makes this easy by allowing us to pass the array of pre-launched packets as an argument to the LAUNCH kernel (arrow 1 in Fig 4.7). Launching packets from the LAUNCH kernel is done by reading the data from global memory (arrow 3 in Fig 4.7) and moving the packet to the DRAW STEP kernel. Keeping the complex conditional code required to implement the various sources on the CPU avoids the significant area usage and potentially reduced performance that would have been required to implement photon launching completely on the FPGA. Keeping the launching logic in the CPU also allows our FPGA design to instantly support the same light sources as the software - avoiding a major limitation of other high-performance FPGA implementations [1, 19].

Both Intel and Xilinx support OpenCL pipes, which are a method that allows data to be transferred efficiently from one kernel to another. Pipes bypass the slow global memory traditionally used for inter-kernel communication by using FIFO queues implemented in the on-chip RAMs of the FPGA, as illustrated in Fig 4.8. The major limitation of pipes is that they do not support `struct` datatypes. This makes moving complex pieces of data (like the definition of a packet in our case) difficult. To address this, we use the Intel FPGA SDK for OpenCL channels extension [32]. Channels are specific to the Intel FPGA implementation of the OpenCL standard (i.e. they are not part of the official OpenCL 2.0 standard [46]). They are similar to pipes and support the `struct` datatype, which makes moving more complex pieces of data between kernels easier. For our design, each stage in Figure 4.6 is a separate OpenCL kernel and the arrows represent a unidirectional channel that moves data from one kernel to the next. The reading and writing of channels is illustrated in Code 4.1.

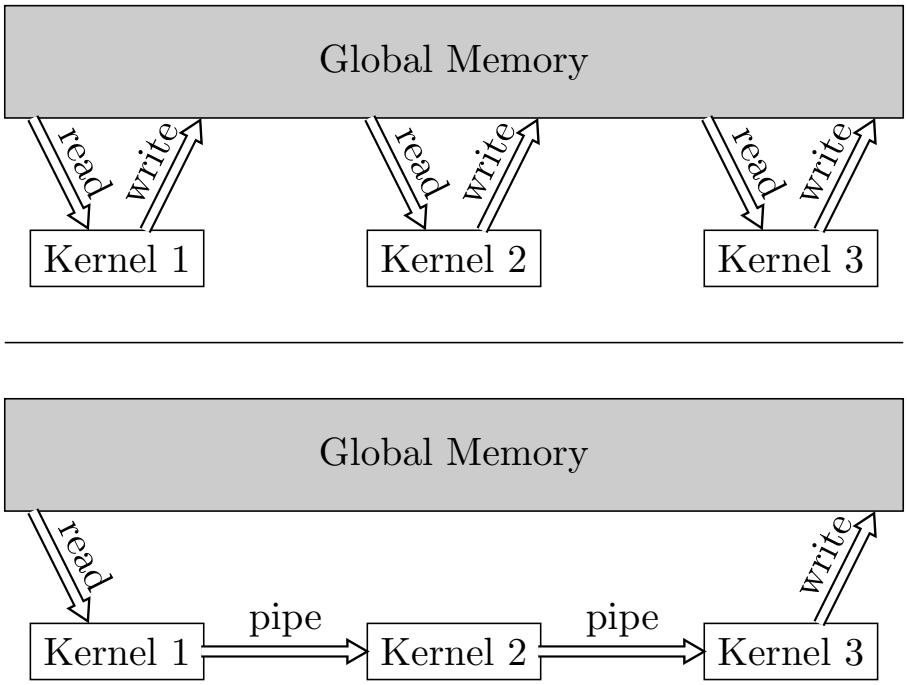

Figure 4.8: Using global memory (top) and pipes/channels (bottom) for inter-kernel communication

As discussed in Section 2.4.4, the FullMonte algorithm contains many trigonometric, vector, matrix and random number distribution functions. FullMonteSW [20] uses 32-bit floating point precision for

these calculations, while previous FPGA versions [1, 19] use fixed-point values of various size and precision. In this work, use 32-bit floating-point precision in the OpenCL kernels. This allows portions of complicated sequential logic to be translated from CPU C++ code to FPGA OpenCL code and therefore greatly improves development productivity and code readability. As discussed later in Section 4.7.2, we achieved good performance and area results using 32-bit floating-point representations. In Section 5.2.1, we discuss the potential benefits and future work of using fixed-point representations in OpenCL for FPGAs.

Aside from the LAUNCH kernel, where the number of loop iterations is known, the code in the shaded kernels of Fig 4.6 is structured similarly to Code 4.1. Fig 4.7 shows that the number of times a packet goes through the circular pipeline path consisting of the DRAWSTEP, HOP, INTERFACE and DROPSPIN kernels is dependent on random factors (the RNG, the material optical properties, the light source parameters, etc). This is represented by the infinite loop in the kernel skeleton from Code 4.1. OpenCL code that is structured in this way creates a block of hardware on the FPGA that reads from an input channel in a non-blocking fashion, performs the logic of the kernel if the data read from the channel is valid, writes the data for the next kernel into the output channel and repeats indefinitely.

```
__kernel kernel_name(/** kernel args **/) {
    /** initialization **/
    while(true) {
        in_data = read_channel_nb_intel(IN_CHANNEL, &valid);
        if(valid) {
            /** kernel logic **/
            write_channel_intel(OUT_CHANNEL, out_data);
        }
    }
}
```

Code 4.1: Skeleton code for the shaded kernels in Figure 4.7

When designing FullMonteFPGACL, we focused on creating a pipeline that produces valid results with good performance. Moreover, we wanted a design that could be easily debugged and extended to new devices or different algorithms. The *only* limitation placed on the model being simulated by FullMonteFPGACL is a maximum of 65k tetrahedral elements in order to fit the entire mesh in on-chip memory. One restriction we found in OpenCL for FPGAs is that separate kernels cannot share local memory. Both the DRAWSTEP and INTERFACE kernel from Fig 4.6 require access to the same read-only tetrahedral mesh data. The ideal scenario would store one copy of the mesh in on-chip memory with two read ports, one for each kernel. However, this is not possible due to the restrictions of the OpenCL language. To move forward, we decided to simply duplicate the mesh memory in both kernels. In Section 5.2.1, we discuss a method to create a custom on-chip caching structure which could allow the mesh to be stored in the large global memory of the board without decimating the performance of the simulator.

In our initial attempt to create the pipeline in Fig 4.6, the compiler generates a circuit with an Fmax of 300MHz and is able to schedule our design with an II of 1 for all kernels except for the DROPSPIN kernel, which had an II of 29. While an II of 29 is not optimal, this is a substantial improvement over the naive implementation that had an II of 600. We find that the double precision (i.e. `double`) read-accumulate-write operation from Code 4.2, which models absorption in the DROPSPIN stage, is the

culprit for the high II. Through software profiling (discussed further in Section 5.2.1), we determine that using a 64-bit floating-point to track tetrahedral absorption can be replaced by a 64-bit fixed-point number (using a `long` datatype) without overflow or loss of significant precision. This optimization lowers the II of the DROPSPIN kernel from 29 to 2.

The remaining II of 2 is a result of the AOC detecting a potential dependency in the *tetraEnergy* array. The compiler breaks apart and pipelines the read-accumulate-write operation into two stages: read and accumulate-write. Thus, if consecutive iterations of the loop in Code 4.2 are to the *same* tetrahedron, then the entire read-accumulate-write operation (the read and then the accumulate-write) of the first iteration must be completed in its entirety before the next iteration reads from the same address, or else accumulations could be lost. Since the AOC cannot determine at compile time (without an intervention from the developer) if consecutive loop iterations will write to the same memory address (i.e. the same tetrahedron), it conservatively assigns an II of 2 to the DROPSPIN kernel and thus guarantees that read-accumulate-write operations from consecutive loop iterations will not overlap.

```
local double tetraEnergy[MAX_ONCHIP_TETRAS];

while(true) {
    /** ... **/
    tetraEnergy[packet.tetra_idx] += packetWeightLoss;
    /** ... **/
}
```

Code 4.2: Initial DROPSPIN kernel snippet

To remove this dependency and achieve an II of 1, we create two instances of the accumulation array and alternate read-accumulate-write operations on consecutive loop iterations. We use the compiler pragma (*ivdep*) to tell the AOC how many consecutive loop iterations are guaranteed not to access the same memory locations. Since we alternate the writes to two memories, we know that directly consecutive loop iterations will write to different memories and the previously described dependency has been removed. This optimization, illustrated in Code 4.3, achieves an II of 1 but results in a lower Fmax of 150MHz.

```
local long tetraEnergy[MAX_ONCHIP_TETRAS][2];

#pragma ivdep safelen(2)
while(true) {
    /** ... **/
    tetraEnergy[packet.tetra_idx][cache_idx & 0x1] += DOUBLE2FIXED(
        packetWeightLoss);
    cache_idx++;
    /** ... **/
}
```

Code 4.3: DROPSPIN kernel snippet after removing the initial memory dependency

We now wish to increase the circuits Fmax, without causing the II of any kernels to increase. Using the Quartus Timing Analyzer, we know that the critical path is from the output register of the *tetraEnergy* memory, through a 64-bit adder and to the input register of the same memory. This shows that, with

our previous change, the AOC is scheduling the read-accumulate-write into a single cycle which results in a longer critical path and therefore a lower Fmax. To address this, we perform the same optimization described previously, by again duplicating the accumulation arrays (from two to four), increase the loop safe length from two to four using the *ivdep* pragma and manually insert pipeline registers between the read, accumulate and write operations using the built-in AOC function *__fpga_reg()* [32]. This technique, illustrated in Code 4.4, allows us to achieve an II of 1 and an Fmax of 312 MHz. This optimization sacrifices an increase in FPGA resources for performance by using 4x more on-chip memory for the accumulation arrays. However, a single accumulation array uses ∼8% of the Stratix 10 FPGA BRAMs which, in our opinion, justifies our tradeoff decision.

```
local long tetraEnergy[MAX_ONCHIP_TETRAS][4];

#pragma ivdep safelen(4)
while(true) {
    /** ... **/
    long currVal = __fpga_reg(tetraEnergy[packet.tetra_idx][cache_idx&0x3]);
    long newVal  = __fpga_reg(currVal + DOUBLE2FIXED(packetWeightLoss));
    tetraEnergy[packet.tetra_idx][cache_idx&0x3] = newVal;
    cache_idx++;
    /** ... **/
}
```

Code 4.4: Final DROPSPIN kernel snippet

With many FPGA resources remaining, as shown in Table 4.10, we explore the potential for pipeline duplication. This would allow us to take advantage of both the pipeline and thread level parallelism inherent in the algorithm using the FPGA. To duplicate the pipeline, we use code like that summarized in Code 4.5 to instantiate multiple copies of the original kernel from Code 4.1. This includes the duplication of channels used to pass data between the now duplicated kernels. We do this by using an array of channels, one per pipeline instance, and access them by the pipeline index p. In the LAUNCH kernel, each instance of the pipeline processes a fraction of the total photon packets requested. If $Npkts$ is the total number of packets requested, then each pipeline processes $N = Npkts/NUM_PIPELINES$ packets with an offset of $p * N$ into global memory. To simplify the pipeline duplication, we also duplicate the on-chip tetrahedral geometry memory and energy accumulation arrays for each pipeline. Instantiating $PIPELINE_N$ copies reduces the maximize size of the memory (and therefore the maximum number of tetrahedrons that can be stored on chip) by a factor of $NUM_PIPELINES$. Therefore, we only instantiate two instances of the pipeline. This allows us to store 65k tetrahedral elements on-chip and use some of the existing models from Table 4.2 for validation and benchmarking. Each pipeline is responsible for half of the total packets requested and stores the energy accumulations in its own local arrays. Therefore, after the simulation is finished but before being written back to global memory (arrow 4 in Figure 4.7), the accumulation arrays for each pipeline instance are summed together locally and then written back to global memory. The reduction of the accumulation arrays could be done on the CPU instead. However, doing the reduction on the FPGA requires a small number of FPGA resources (three 64-bit adders and a few pipeline registers) but reduces the amount of output data to be transferred from the FPGA to the CPU by 4x.

The two pipeline design achieves an II of 1 and an Fmax of 285 MHz. The lower Fmax compared

to the single pipeline is likely attributable to the increased FPGA routing complexity of the larger two pipeline design. Section 5.2.1 discusses a more sophisticated method for scaling up the design with even more pipeline instances and more complex on-chip and off-chip memory.

```
__kernel kernel_name(/** kernel args **/) {
    /** initialization **/
    while(true) {
        #pragma unroll PIPELINE_N
        for(int p=0; p<PIPELINE_N; p++) {
            in_data = read_channel_nb_intel(IN_CHANNEL[p], &valid);
            if(valid) {
                /** main kernel logic **/
                write_channel_intel(OUT_CHANNEL[p], out_data);
            }
        }
    }
}
```

Code 4.5: Skeleton code for duplicating kernels

4.7.2 Results

Validation

Similar to the GPU-accelerated simulator, we validate FullMonteFPGACL using the method described earlier in Section 4.3. Table 4.9 summarizes the normalized L1-norm values and proves the validity of our FPGA-accelerated simulator. In addition, Fig 4.9 qualitatively shows the accuracy of our FPGA-accelerated simulator.

Table 4.9: Normalized L1-norm values across the benchmark models using 10^6 packets for two differently seeded CPU simulations and an FPGA and CPU simulation

	cube_5med	FourLayer	Tumor
FullMonteSW-FullMonteSW	0.0322	0.0012	0.0027
FullMonteFPGACL-FullMonteSW	0.0342	0.0020	0.0026

FPGA Utilization

The hardware resources required for each kernel and the total design are summarized in Table 4.10. The *kernel total* row shows the total resource utilization for the partition of the design dedicated to the OpenCL kernels. The *design total* row shows the resource summary reported by the Quartus compilation report. The extra resources make up the global interconnect for interacting with the DDR memory controller, global memory caches for DDR accesses and OpenCL board interface logic. The RAM utilization is high due to the on-chip caching of tetrahedrons. However, as mentioned in the footnote of Table 4.10, when the tetrahedron data is *not* cached on-chip, only 10% of the total FPGA RAMs are used. Reducing memory usage, removing the mesh element constraint and duplicating the pipeline to scale up performance are discussed further in Section 5.2.1.

(a) (b)

Figure 4.9: Output tetrahedral fluence plots of the *cube_5med* model (Table 4.2) for FullMonteSW (a) and FullMonteFPGACL (b) using 10^8 packets.

Table 4.10: Stratix 10 resources for a single FullMonteFPGACL pipeline

Kernel	ALM	FF	RAM	DSP
RNG	27187	59805	0	2
LAUNCH	3958.3	8008	0	0
DRAWSTEP	21100	43903	2173	11
HOP	8573.3	25221	86	47
INTERFACE	27827.2	66669	2373	66
DROPSPIN	20347.8	54563	996	59
EXIT	178.1	380	0	0
kernel total	109171.7	258549	5628	185
% of available	12%	7%	50%*	3%
design total	185208	406112	5866	185
% of available	20%	11%	52%*	3%

* 10% when compiled without the on-chip tetrahedron cache

Performance and Energy Efficiency

To benchmark the performance of FullMonteFPGACL, we use the three models from Table 4.2 with less than 64k tetrahedrons (cube_5med, FourLayer and HeadNeckTumour) so that all tetrahedrons can fit on-chip. We simulate these models with packet counts ranging from 10^6-10^7 (the typical range of packets for most applications) using FullMonteSW and FullMonteFPGACL with a single and double pipeline instance. FullMonteSW is configured to use 12 threads with *AVX2* instructions enabled. The performance results are summarized in Table 4.11 and show that FullMonteFPGACL achieves a speedup of 2.1-3.9x over FullMonteSW. As with the GPU, the performance improvement depends on the mesh complexity, with the highest speedups occurring on the most complex and realistic mesh, the HeadNeck-Tumour. FullMonteFPGACL exceeds the performance of the simulator created by Afsharnejad et al. [1] while being able to support a wider range of models and applications due to its integration into the FullMonteSW project, usage of tetrahedral mesh elements (as opposed to voxels) and support of all the

light sources supported by FullMonteSW (as opposed to only isotropic point sources). FullMonteFPGACL is 1-3x *slower* than FullMonteCUDA. However later in Section 5.2.1, we suggest future research suggestions for reducing FPGA resource utilization, removing the mesh element constraint and scaling up the FPGA design by filling the chip with up to 8 instances of the pipeline. With 8 instances of the pipeline, we believe the FPGA should achieve a performance improvement of 8-16x and 1-4x over FullMonteSW and FullMonteCUDA, respectively.

To investigate the energy-efficiency of FullMonteFPGACL, we compare the energy-efficiency (performance per Watt) against FullMonteSW (CPU) and FullMonteCUDA (GPU). We estimate the power of a single FPGA pipeline to be 30W using the Quartus PowerPlay Power Analysis post-synthesis vectorless power estimation with a 12.5% input toggle rate. Using the same method, we estimate a 45W power usage for two instances of the FPGA pipeline. We attempted to use the OpenCL simulator to generate input vectors to enable more accurate FPGA power estimations more accurate. However, after substantial effort we were still unable to make the OpenCL simulator work for the OpenCL compiler (version 18.1) on the Ubuntu 16.04 operating system. The Intel Core i7 CPU has a thermal design power (TDP) of 140W and the NVIDIA Titan Xp GPU has a TDP of 250W. Both the GPU- and FPGA-accelerated simulators require a CPU host to perform the pre- and post-processing, including the computation of the launch packets discussed earlier in Section 4.4. This will add to the power of both the GPU- and FPGA-accelerated simulators; however, since most of the simulation computation is placed on the GPU- or FPGA-accelerator, the power overhead will be negligible, especially for simple light sources. To compare energy-efficiency, we conservatively assume 85% of the CPU and GPU TDP (119W and 213W, respectively). To compute the energy-efficiency, we use the performance numbers from Table 4.11 and normalize by the power to compute the simulator's *performance per Watt*. A single FullMonteFPGACL pipeline is up to 10x and 4x more energy-efficient than the CPU and GPU, respectively. The duplicated pipeline version achieves an 11x and 5x energy-efficiency improvement compared to the CPU and GPU, respectively. If the single pipeline is duplicated to fill the FPGA, we expect to achieve an energy-efficiency improvement of up to 17x and 7x over the CPU and GPU versions, respectively. The improvement is due to amortizing the device static power overhead over more productive pipeline units. The duplication of the pipeline is discussed further in Section 5.2.1.

Table 4.11: Runtimes for various benchmarks on a CPU, GPU and a single and duplicated FPGA pipeline

Packets	Model	CPU (s)	GPU (s)	FPGA (s)	
				1 pipeline	2 pipelines
10^6	cube_5med	5.0	1.3	3.5	2.3
	FourLayer	2.0	0.4	1.3	0.8
	Tumor	1.7	0.4	0.7	0.5
10^7	cube_5med	48.7	12.3	35.1	22.9
	FourLayer	18.8	4.7	13.2	8.6
	Tumor	17.9	3.7	7.2	4.6

The authors in [19] report an energy-efficiency of 67x over their baseline CPU (14W total). Our lower energy-efficiency improvement over a CPU baseline can be explained by several factors. First, our OpenCL design uses 32-bit floating-point precision which results in more area usage than the custom-width fixed-point precision used in [19]. Second, the problem size is bigger since we allow for more tetrahedrons to be stored in the FPGA RAM blocks. Lastly, as shown in Fig 4.10, FPGA power has

increased with newer process generations, especially from 28 to 14nm. In addition, the power gap between CPUs and FPGAs has been shrinking with newer process generations [3].

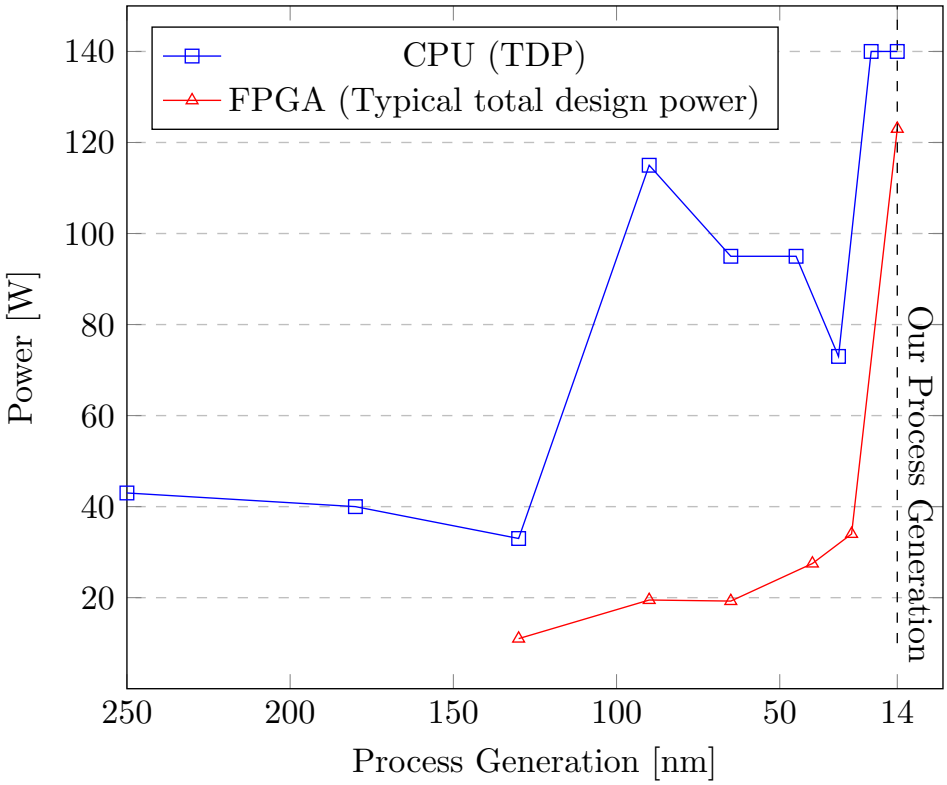

Figure 4.10: The power draw of Intel CPUs and FPGAs (Stratix family) across process generations [3].

4.7.3 Development Productivity

In this work we use OpenCL for FPGAs over C++ HLS and HDL languages for various reasons, with the most important being development productivity. As discussed in Section 2.5.3, some of the advantages of using OpenCL over lower-level languages are: improvements in development productivity, code maintenance, code extendibility and the ability to quickly create a full FPGA-accelerated system. This section will, to the best of our ability, quantify the development effort and time across various hardware acceleration implementations of FullMonteSW.

Table 4.12 provides a rough estimate of the *single developer* time required for the different hardware acceleration implementations to both reach the prototype stage, as well as achieve a measurable performance improvement over the baseline software implementation. In addition, Table 4.12 shows the *size* of the designs in terms of number of files and lines of code, giving a measure of the readability and extensibility of the design.

Earlier in this work we discussed FullMonteCUDA, which accelerated FullMonteSW using an NVIDIA GPU and the CUDA SDK. Table 4.12 shows that the prototype took a single developer roughly a week to create. This prototype achieved a 2x speedup over FullMonteSW and the final version achieved between a 4-13x speedup depending on the model being simulated. The major factors in the fast and effective development of the accurate and efficient simulator are:

Table 4.12: Estimating the time for FullMonteSW hardware acceleration implementations to be prototyped and to outperform the software by a single developer

Language	Source Code (files, lines)	Time to (single developer):		Performance Increase over SW
		Prototype	Performance	
CUDA	(6, 1140)	1 week	1 week	4-13x
Bluespec	(82, 17685)	6 months[1]	>3 years[1]	—[1]
Vivado HLS	(35, 45888)	*unknown*	∼4 months[3]	3x[2,4]
OpenCL for FPGA	(11, 1400)	2 weeks	3 months	4x

[1] The pipeline was incomplete and not fully validated
[2] Supports only a subset of the features and light sources supported by FullMonteSW
[3] It took four developers 2 months
[4] Sacrifices accuracy by using voxels and not tetrahedrons

- The comprehensive and easy-to-follow NVIDIA documentation and example projects
- The wide spread usage of the the CUDA SDK, providing a plethora of online support
- The short compile times compared to FPGAs
- The similarity of CUDA to the CPU multithreading paradigm used in FullMonteSW

Cassidy et al. [19] targeted an Intel Stratix V FPGA using Bluespec SystemVerilog (BSV). The intent of using BSV was to abstract from pure SystemVerilog RTL and improve development productivity without substantially sacrificing area and performance. However, Table 4.12 shows that this was not the case. The simulator only supports homogeneous models and, while the highly optimized pipeline stages can be verified in simulation individually, when the entire pipeline is connected and tested the results are inaccurate in both simulation and hardware. In addition, due to the complexity of the FPGA design, only a subset of the features from FullMonteSW are supported.

Afsharnejad et al. [1] accelerated a modified version of FullMonteSW for voxel meshes using Vivado HLS targeting a Xilinx FPGA. As shown in Table 4.12, the simplified design took four developers two months to create and achieved a 3x speed improvement over FullMonteSW on one benchmark model. The longer design time and increased design complexity (in terms of number of files and lines of code) compared to FullMonteFPGACL is likely attributable to the use of C++ based HLS (Vivado HLS), instead of OpenCL. As discussed in Section 2.5.3, using C++ HLS requires the developer, in addition to creating a complex simulator, to compile the C++ code into blocks of FPGA logic, connect them using a system integrator and find a way to transfer data to the external memory of the FPGA board. These tasks can be error prone, tedious and time consuming, which we believe accounts for the increase in development time and design complexity compared to our design.

In this work we created FullMonteFPGACL, which is an FPGA-accelerated version of FullMonteSW using OpenCL for FPGAs. As shown in Table 4.12, it took a single developer around 2 weeks to create a functioning prototype - similar to the time required for the GPU. This included creating the CPU-FPGA system and verifying its functionality in emulation. Roughly another 2 months were required to investigate the inefficiencies of the prototype and test various techniques that led to the optimizations discussed in Section 4.7.1 and achieve a 4x performance improvement over FullMonteSW. The main development productivity bottleneck was the long (∼8 hour) compile times for the FPGA, of which the AOC front-end took roughly an hour and Quartus the remaining seven. The OpenCL emulator was useful for verifying the functionality of small pieces of code. However, the validation of the entire pipeline requires running in the range of 10^6-10^7 packets, which takes up to 8 hours in the emulator. Therefore,

to validate the entire pipeline, we found it more productive to compile the entire design for the FPGA (∼8 hours) and run it on the actual hardware (∼seconds), as it performed a more rigorous validation of the pipeline and benchmarked the performance in a similar amount of time as using the emulator. To debug performance bottlenecks, we *attempted* to use the new OpenCL Simulator (a wrapper around ModelSim) developed by Intel. However, after many weeks of debugging the simulator we found that it was not in a usable state at the time of this work. The OpenCL simulator would have been extremely useful for us to find performance bottlenecks and create input vectors to increase the accuracy of the power estimations from Section 4.7.2.

Using OpenCL significantly reduced the amount of written code (both for the FPGA and to communicate between the CPU and FPGA) compared to HDL languages and C++ HLS, as shown by the data in Table 4.12. In addition, we found that the use of OpenCL drastically decreased the effort and time involved in creating a complete heterogeneous CPU-FPGA system, by raising the abstraction level of the CPU-FPGA communication. The OpenCL compiler was capable of optimizing and pipelining the complex sequential code within each kernel. This made converting CPU code to target an FPGA less cumbersome by allowing us, in many cases, to directly copy blocks of sequential C++ CPU code to OpenCL for the FPGA.

The use of OpenCL was beneficial for development productivity; however it does not remove the necessity for the developer to *think* like a hardware designer. As discussed in Section 4.7.1, simply copying all the CPU code to OpenCL code for the FPGA (as we did for FullMonteCUDA) results in performance over 500x *worse* than the CPU. We found it most effective for the developer to begin by considering the hardware circuit they wish to create, then exploring alternative and sometimes counter-intuitive methods to describe that circuit in OpenCL. We believe that OpenCL for FPGAs is **not** a method for software developers to program FPGAs, but rather a method for increasing the productivity of a hardware developer. This is made evident by the performance debugging we undertook and optimizations we made to achieve the high performance for the FPGA. Specifically, some of these tasks included: the investigation of hardware area and timing reports, identifying issues in the generated HDL and creating a high-level solution in the OpenCL code. We found the most difficult parts of the design were: (1) structuring the channel logic to communicate accurately and efficiently between kernels (2) correctly unrolling OpenCL loops to create the duplicated pipelines (3) optimizing the performance for 64-bit read-accumulate-write operations to the FPGA RAMs, as discussed in Section 4.7.1 and (4) dealing with OpenCL's inability to share local memory across kernels, as discussed in Section 4.7.1.

Chapter 5

Conclusions and Future Work

This chapter summarizes the contributions of this thesis and gives suggestions for future research to extend this work.

5.1 Conclusions

The main contributions of this thesis advance the field of hardware accelerated biophotonic simulators by creating the fastest GPU-accelerated simulator and first complete and verified FPGA-accelerated simulator of their kind. These accelerators are fully integrated into the full-featured FullMonteSW simulator, so they can be used with all the features of that software, while accelerating the most CPU-intensive calculations. In addition, this thesis further enhanced FullMonteSW by making other improvements to its performance, accuracy and usability.

Our addition of the cylindrical diffuser and mesh region light sources makes the simulator more accurate in representing the light propagation in PDT treatment and BLI, respectively. In the future, the addition of new light sources and processing techniques could extend the applicability of the simulator to other applications in the medical field and even to applications outside of biophotonics where objects need to be located in turbid media. Examples of potential applications include autonomous vehicles navigating through light scattering fog or aquatic navigation when diving close to the seabed [12, 37]. The RTree data structure that we implemented in FullMonteSW improved the point to containing tetrahedron query speed by over 230x. This query can happen millions of times in a single forward simulation and even more when solving inverse problems. This query is not part of the key hop-drop-spin loop that we hardware accelerate, and therefore its software performance can be vital to the performance of the simulator and to inverse solvers which use it.

To this date, the GPU-accelerated simulator that we created, FullMonteCUDA, is the fastest tetrahedral-mesh Monte Carlo biophotonic simulator. FullMonteCUDA is completely integrated with the FullMonteSW code, which allows the simulation acceleration to be transparent to the user. FullMonteCUDA provides a 4-13x speed improvement over the fastest software simulator for this problem (FullMonteSW), which can enable inverse solvers for complex medical applications, like DOT, BLI and PDT, to be more accurate and feasible.

We also explored FPGAs as a means for hardware acceleration and developed FullMonteFPGACL: the first complete and verified FPGA-accelerated simulator of its kind. FullMonteFPGACL achieved

up to a 4x speed improvement over FullMonteSW and an 11x improvement in energy-efficiency. Similar to FullMonteCUDA, FullMonteFPGACL is completely integrated with the FullMonteSW code, which allows the acceleration to be transparent to the user. While FullMonteFPGACL trails FullMonteCUDA in performance, we only use a fraction (∼20%) of the FPGA area for the two pipelines. In the next section, we discuss an outline for future work that could remove its current constraint of storing at most 65k tetrahedra and scale up the performance of the FPGA design to achieve a 16x speed improvement and 17x energy efficiency improvement over FullMonteSW, which would exceed the performance of FullMonteCUDA.

5.2 Future Work

In this section we will discuss methods for improving the performance and usability of the FPGA and GPU accelerators from this work.

5.2.1 FPGA

This section will discuss methods for reducing the FPGA resource utilization, removing the mesh element size constraint and scaling up the design to increase performance.

Fixed-point Representation

Our design uses 32-bit precision integers (i.e. `int` and `uint`) for data not requiring decimal point precision (e.g. tetrahedron and material IDs) and 32-bit precision floating point numbers (i.e. `float`) for all data entries that require decimal point precision (e.g. positions, vectors, material properties, lengths, etc), with the exception of the 64-bit fixed-point energy accumulation that was discussed in Section 4.7.1. We performed software profiling (details of which can be found in [18]) to determine the range and precision needs for different variables and the results are summarized in Table 5.1. Future work could include using the *arbitrary integer precision extension* of the Intel FPGA SDK for OpenCL [32] which allows the OpenCL design to easily use custom width and precision fixed-point integers. This could result in up to a 2x reduction in RAM, DSP and ALM utilization for the OpenCL kernels. This reduction in resources would allow for more throughput per device by instantiating even more pipelines on the device, as discussed later in this section.

Table 5.1: FPGA precisions for various data determined by software emulation [18]

Data	Total Bits	Integer bits	Fractional bits	Range	Precision
3D Unit Vector	18	1	17	[-1, 1]	8×10^{-6}
3D Position	18	4	14	[-8cm, 8cm]	6×10^{-5}cm
Step length	18	6	12	[0, 63]	1.2×10^{-4}
Packet weight	36	1	35	[0, 1]	3×10^{-11}
Tetrahedral absorption	64	29	35	[0, 2×10^{8}]	3×10^{-11}
Tetrahedral ID	25	25	—	[0, 3×10^{7}]	—
Material ID	4	4	—	[0, 15]	—

Custom Memory Controller

Our design limits the size of the mesh to 65k tetrahedral elements. However, practical medical models can contain more than 1 million tetrahedrons. The storage of data is broken down into two components: read-only tetrahedral geometry data and read-write energy accumulation array data. We profiled the memory access pattern and found that there is little temporal reuse of data and that a 4-8 entry LRU cache for both the geometry and accumulation arrays could provide a hit-rate near 60%. Larger LRU caches show poor trade-offs in terms of area and hit-rate increase. As hypothesized, the memory profiling results across various benchmark models showed highly irregular access patterns. This access pattern does not fit the typical cache architecture and eviction policies found in most CPUs and GPUs. However, we found that, across all benchmarks, the access pattern has a Zipf-like distribution [16]. This means that a high percentage of accesses are to a relatively small portion of the tetrahedrons.

In future work, the geometry and energy accumulation array data could be moved to the high-capacity external memory of the board and the controller implemented using the FPGA RAMs. For both the geometry and energy accumulation data, we would create a 4-8 entry LRU L1 cache backed by a static L2 cache. To populate the L2 cache, we would simulate a small subset of packets pre-emptively, sort the tetrahedrons based on the Zipf distribution (i.e. by access rate) and store the highest accessed tetrahedrons in the static L2 cache. The overhead of running a subset of packets pre-emptively in this manner would decrease the performance of a single simulation. However, this time could be amortized when running many simulations with the same mesh, which is the typical case when solving PDT inverse problems [66].

Multiple Pipeline Instances

In Section 4.7.1, we discussed how we instantiated two instances of the pipeline in the FPGA. This section will discuss a more sophisticated approach that can utilize the memory controller discussed in the previous section and decrease the FPGA RAM utilization when scaling the design up even further.

Based on the resource utilization from Table 4.10, the limiting factor for duplicating the pipeline is RAM (50%). However, without the on-chip caching of tetrahedrons, the RAM usage of the kernels drops to $\sim$10% per pipeline and the limiting factor becomes ALMs (12%). Thus, we believe that 8 copies of the pipeline can fit on-chip, if the RAMs are managed properly.

A naive approach would be to duplicate the on-chip memory for each instance of the pipeline, as we did for the current design. However, as the number of pipelines increases, the duplicated memory will severely limit the amount of tetrahedrons that can be stored on chip. In this trivial approach, assuming that the memory controller from the previous section is implemented, the L1 and L2 caches for both the geometry and energy accumulation data would be duplicated. This would decrease the maximum size of the static L2 caches for the memory controllers and therefore reduce the overall hit-rate. However, by utilizing the two ports and multi-pumping support of the FPGA RAMs [31, 32], we can share a single memory across multiple pipelines, thereby reducing the RAM utilization per pipeline and increasing the size of the memories.

The energy accumulation arrays are read from and written to, meaning that the RAMs used to implement their cache must have one port for reading and the other for writing. In addition, the current Intel FPGA OpenCL documents [31, 32] only mention multi-pumping support for memory reads. Therefore, the on-chip memory controller cache for the energy accumulation arrays must be duplicated

for each pipeline instance.

The tetrahedral geometry data is read-only, meaning that multiple pipeline instances can share the two read ports of the RAMs. In addition, each read port can be double or triple-pumped, allowing a single cache to supply 2, 4 or 6 instances of the pipeline if configured using no multi-pumping, double-pumping or triple-pumping, respectively. We hypothesize that we could fit 8 pipelines in the FPGA and therefore the most logical configuration for this would use two copies of the static L2 cache and double-pump the two read ports, as depicted in Figure 5.1. This configuration would allow the static L2 geometry caches to store ∼4x more tetrahedrons than the naive configuration.

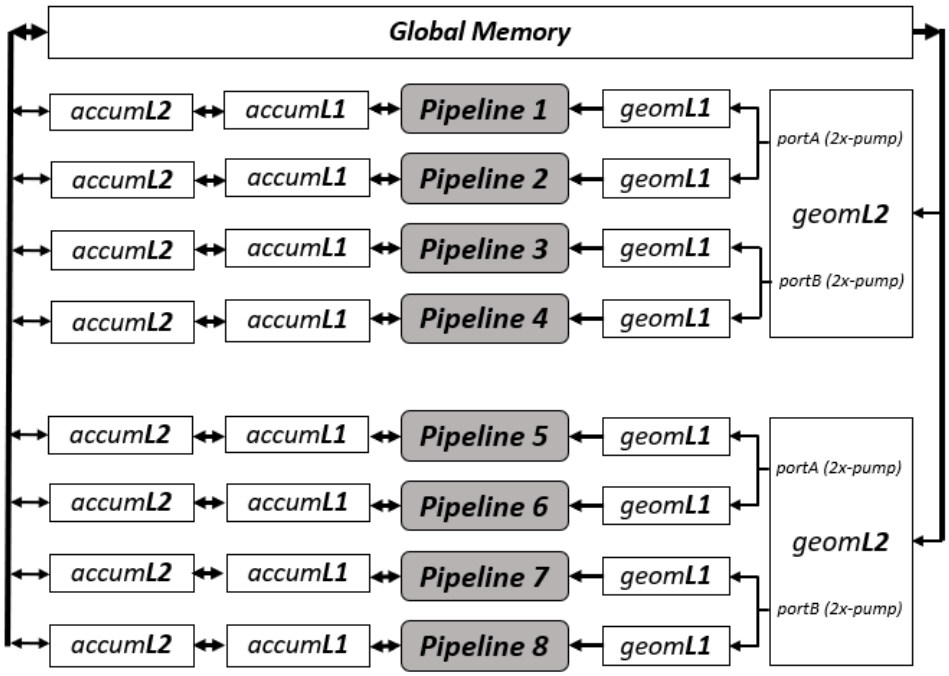

Figure 5.1: Proposed memory structure for duplicating FullMonteCL pipelines [68]

To estimate the performance of this design, we scale up the single-pipeline performance by a factor of 8, which is similar to the performance scaling we experienced with 2 pipelines (Section 4.7.2) as well as the use of more CPU cores [58]. To estimate the energy-efficiency of this design, we multiply the core dynamic power (excluding I/O) by a factor of 8. This estimate is conservative since the global board control logic and the shared on-chip RAM caches (and therefore their dynamic power) will not scale linearly with the number of pipeline instances. This leads to an estimated 16x performance and 17x energy-efficiency improvement over FullMonteSW and a 1-4x performance and 7x energy-efficiency improvement over FullMonteCUDA.

5.2.2 GPU

After creating the high performance GPU design, we used the *NVIDIA Visual Profiler* (NVVP) to identify bottlenecks in the code. Fig 4.3 shows the profiling results for our design using the *HeadNeck* mesh from Table 4.2 using 10^6 packets and a single isotropic point source. The chart in Fig 4.3 shows the distribution of stall reasons for our kernel and lets us pinpoint the performance bottlenecks [50]. The

chart shows that the largest number of stalls in our optimized design are due to memory dependencies, which is typical for applications like ours with large datasets and unpredictable memory access patterns.

As we hypothesized, the tetrahedron intersection calculation was a major bottleneck consisting of mostly memory dependencies. All memory dependency bottlenecks occur when accessing tetrahedral data. Future work could investigate more advanced tetrahedron caching schemes in shared or constant memory. For example, in the previous section we discussed that the tetrahedron access pattern has a Zipf access pattern. Therefore, implementing a Zipf cache in shared or constant memory could reduce this global memory bottleneck for tetrahedrons. Prefetching tetrahedrons may also decrease the global memory bottleneck and improve the performance. When a packet resides in a tetrahedron, we know it will move to one of the four adjacent tetrahedrons on the next step. Therefore, every time a packet moves to a new tetrahedron, the kernel could prefetch the data of the four adjacent tetrahedrons while other calculations are being performed. If the packet moves to the next tetrahedron, the request for the tetrahedron memory will already be in-flight (or finished) and the global memory access latency may be reduced.

Bibliography

[1] Yasmin Afsharnejad, Abdul-Amir Yassine, Omar Ragheb, Paul Chow, and Vaughn Betz. Hls-based fpga acceleration of light propagation simulation in turbid media. In *Proceedings of the 9th International Symposium on Highly-Efficient Accelerators and Reconfigurable Technologies*, page 11. ACM, 2018.

[2] TeamCo Agency. Medlight. `http://www.medlight.com/`. Accessed: 2019-12-23.

[3] I. Ahmed, L. L. Shen, and V. Betz. Becoming more tolerant: Designing fpgas for variable supply voltage. In *2019 29th International Conference on Field Programmable Logic and Applications (FPL)*, pages 1–8, 2019.

[4] Erik Alerstam, William Lo, Tianyi David Han, Jonathan Rose, Stefan Andersson-Engels, and Lothar Lilge. Next-generation acceleration and code optimization for light transport in turbid media using gpus. *Biomedical Optics Express*, 1:658–75, 09 2010.

[5] Erik Alerstam, Tomas Svensson, and Stefan Andersson-Engels. Parallel computing with graphics processing units for high-speed monte carlo simulation of photon migration. *Journal of biomedical optics*, 13(6):060504, 2008.

[6] Gene M Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the April 18-20, 1967, spring joint computer conference*, pages 483–485. ACM, 1967.

[7] AN Bashkatov, EA Genina, VI Kochubey, and VV Tuchin. Optical properties of the subcutaneous adipose tissue in the spectral range 400–2500 nm. *Optics and spectroscopy*, 99(5):836–842, 2005.

[8] Karl Beeson, Evgueni Parilov, Mary Potasek, Michele Kim, and Timothy Zhu. Validation of combined monte carlo and photokinetic simulations for the outcome correlation analysis of benzoporphyrin derivative-mediated photodynamic therapy on mice. *Journal of Biomedical Optics*, 24:1, 03 2019.

[9] Frederic Bevilacqua, Dominique Piguet, Pierre Marquet, Jeffrey D. Gross, Bruce J. Tromberg, and Christian Depeursinge. In vivo local determination of tissue optical properties: applications to human brain. *Applied optics*, 38(22):4939–4950, 1999.

[10] Jonas Binding, Juliette Ben Arous, Jean-François Léger, Sylvain Gigan, Claude Boccara, and Laurent Bourdieu. Brain refractive index measured in vivo with high-na defocus-corrected full-field oct and consequences for two-photon microscopy. *Optics express*, 19(6):4833–4847, 2011.

[11] Tiziano Binzoni, Terence Leung, Remo Giust, Daniel Rüfenacht, and Amir Gandjbakhche. Light transport in tissue by 3d monte carlo: Influence of boundary voxelization. *Computer methods and programs in biomedicine*, 89:14–23, 02 2008.

[12] Wolfram Blattner and Michael B. Wells. Monte carlo studies of light transport through natural atmospheres. Technical report, Radiation Research Associates Inc, 1973.

[13] D. A. Boas, D. H. Brooks, E. L. Miller, C. A. DiMarzio, M. Kilmer, R. J. Gaudette, and Quan Zhang. Imaging the body with diffuse optical tomography. *IEEE Signal Processing Magazine*, 18(6):57–75, Nov 2001.

[14] David A. Boas, Joseph P. Culver, Jonathan J. Stott, and Andrew K. Dunn. Three dimensional monte carlo code for photon migration through complex heterogeneous media including the adult human head. *Opt. Express*, 10(3):159–170, Feb 2002.

[15] V. Boyer, D. El Baz, and M.A. Salazar-Aguilar. Chapter 10 - gpu computing applied to linear and mixed-integer programming. In Hamid Sarbazi-Azad, editor, *Advances in GPU Research and Practice*, Emerging Trends in Computer Science and Applied Computing, pages 247 – 271. Morgan Kaufmann, Boston, 2017.

[16] Lee Breslau, Pei Cao, Li Fan, Graham Phillips, Scott Shenker, et al. Web caching and Zipf-like distributions: Evidence and implications. In *Ieee Infocom*, volume 1, pages 126–134. INSTITUTE OF ELECTRICAL ENGINEERS INC (IEEE), 1999.

[17] H Buiteveld, J H. Hakvoort, and Marcel Donze. Optical properties of pure water. *Proc. SPIE, Ocean Optics XII*, 2258:174–183, 01 1994.

[18] Jeffery Cassidy. *FullMonte: Fast Biophotonic Simulations*. PhD thesis, University of Toronto, 2014.

[19] Jeffery Cassidy, Lothar Lilge, and Vaughn Betz. Fast, power-efficient biophotonic simulations for cancer treatment using fpgas. In *2014 IEEE 22nd Annual International Symposium on Field-Programmable Custom Computing Machines*, pages 133–140, May 2014.

[20] Jeffery Cassidy, Ali Nouri, Vaughn Betz, and Lothar Lilge. High-performance, robustly verified Monte Carlo simulation with FullMonte. *Journal of Biomedical Optics*, 23:1 – 11, 2018.

[21] Jeffrey Cassidy, Lothar Lilge, and Vaughn Betz. Fullmonte: A framework for high-performance monte carlo simulation of light through turbid media with complex geometry. *Proc. SPIE*, 8592, 02 2013.

[22] NVIDIA Corporation. Nvidia tesla p100, 2016.

[23] Vinh Nguyen Du Le, John Provias, Naresh Murty, Michael S Patterson, Zhaojun Nie, Joseph E Hayward, Thomas J Farrell, William McMillan, Wenbin Zhang, and Qiyin Fang. Dual-modality optical biopsy of glioblastomas multiforme with diffuse reflectance and fluorescence: ex vivo retrieval of optical properties. *Journal of biomedical optics*, 22(2):027002, 2017.

[24] Clément Dupont, Gregory Baert, Serge Mordon, and Maximilien Vermandel. Parallelized monte-carlo using graphics processing units to model cylindrical diffusers used in photodynamic therapy: From implementation to validation. *Photodiagnosis and Photodynamic Therapy*, 2019.

[25] Qianqian Fang. Mesh-based monte carlo method using fast ray-tracing in plücker coordinates. *Biomed. Opt. Express*, 1(1):165–175, Aug 2010.

[26] Qianqian Fang and David A. Boas. Monte carlo simulation of photon migration in 3d turbid media accelerated by graphics processing units. *Opt. Express*, 17(22):20178–20190, Oct 2009.

[27] Qianqian Fang and David R. Kaeli. Accelerating mesh-based monte carlo method on modern cpu architectures. *Biomed. Opt. Express*, 3(12):3223–3230, Dec 2012.

[28] Christina Habermehl, Christoph H. Schmitz, and Jens Steinbrink. Contrast enhanced high-resolution diffuse optical tomography of the human brain using icg. *Opt. Express*, 19(19):18636–18644, Sep 2011.

[29] Mark Horowitz. 1.1 computing's energy problem (and what we can do about it). In *2014 IEEE International Solid-State Circuits Conference Digest of Technical Papers (ISSCC)*, pages 10–14. IEEE, 2014.

[30] Shih-Hao Hung, Min-Yu Tsai, Bo-Yi Huang, and Chia-Heng Tu. A platform-oblivious approach for heterogeneous computing: A case study with monte carlo-based simulation for medical applications. In *Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, FPGA '16, pages 42–47, New York, NY, USA, 2016. ACM.

[31] Intel Corporation. Intel FPGA SDK for OpenCL best practices guide, 01 2019.

[32] Intel Corporation. Intel FPGA SDK for OpenCL programming guide, 01 2019.

[33] Steven Jacques. Mcxyz, 2013.

[34] Steven Jacques and Lihong Wang. *Monte Carlo Modeling of Light Transport in Tissues*, pages 73–100. 01 1995.

[35] Steven L Jacques. Optical properties of biological tissues: a review. *Physics in Medicine & Biology*, 58(11):R37, 2013.

[36] Steven L Jacques and Brian W Pogue. Tutorial on diffuse light transport. *Journal of biomedical optics*, 13(4):041302, 2008.

[37] Robert A. Leathers, Trijntje V. Downes, Curtis O. Davis, and Curtis D. Mobley. Monte carlo radiative transfer simulations for ocean optics: a practical guide. Technical report, Naval Research Lab Washington Dc Applied Optics Branch, 2004.

[38] Hui Li, Jie Tian, Fuping Zhu, Wenxiang Cong, Lihong V Wang, Eric A. Hoffman, and Ge Wang. A mouse optical simulation environment (mose) to investigate bioluminescent phenomena in the living mouse with the monte carlo method. *Academic Radiology*, 11(9):1029–1038, 2004.

[39] Steve M Liao, Nicholas M Gregg, Brian R White, Benjamin W Zeff, Katelin A Bjerkaas, Terrie E Inder, and Joseph P Culver. Neonatal hemodynamic response to visual cortex activity: high-density near-infrared spectroscopy study. *Journal of biomedical optics*, 15(2):026010, 2010.

[40] André Liemert, Dominik Reitzle, and Alwin Kienle. Analytical solutions of the radiative transport equation for turbid and fluorescent layered media. *Scientific reports*, 7(1):3819, June 2017.

[41] André Liemert and Alwin Kienle. Explicit solutions of the radiative transport equation in the p3 approximation. *Medical physics*, 41(11):111916, 2014.

[42] Junting Liu, Yabin Wang, Xiaochao Qu, Xiangsi Li, Xiaopeng Ma, Runqiang Han, Zhenhua Hu, Xueli Chen, Dongdong Sun, Rongqing Zhang, et al. In vivo quantitative bioluminescence tomography using heterogeneous and homogeneous mouse models. *Optics express*, 18(12):13102–13113, 2010.

[43] William Lo, Keith Redmond, Jason Luu, Paul Chow, Jonathan Rose, and Lothar Lilge. Hardware acceleration of a monte carlo simulation for photodynamic treatment planning. *Journal of Biomedical Optics*, 14:014019, 01 2009.

[44] Yujie Lu, Hidevaldo B Machado, Qinan Bao, David Stout, Harvey Herschman, and Arion F Chatzioannou. In vivo mouse bioluminescence tomography with radionuclide-based imaging validation. *Molecular Imaging and Biology*, 13(1):53–58, 2011.

[45] Dominik Marti, Rikke N Aasbjerg, Peter E Andersen, and Anders K Hansen. MCmatlab: an open-source, user-friendly, MATLAB-integrated three-dimensional monte carlo light transport solver with heat diffusion and tissue damage. *Journal of biomedical optics*, 23(12):121622, 2018.

[46] Aaftab Munshi. The opencl specification. In *2009 IEEE Hot Chips 21 Symposium (HCS)*, pages 1–314. IEEE, 2009.

[47] Vasilis Ntziachristos, Jorge Ripoll, Lihong V. Wang, and Ralph Weissleder. Looking and listening to light: the evolution of whole-body photonic imaging. *Nature Biotechnology*, 23(3):313–320, 2005.

[48] NVIDIA Corporation. NVIDIA CUDA C best practices guide, 05 2019.

[49] NVIDIA Corporation. NVIDIA CUDA C programming guide, 05 2019.

[50] NVIDIA Corporation. NVIDIA profiler user's guide, 05 2019.

[51] Eiji Okada, Michael Firbank, Martin Schweiger, Simon R. Arridge, Mark Cope, and David T. Delpy. Theoretical and experimental investigation of near-infrared light propagation in a model of the adult head. *Applied optics*, 36(1):21–31, 1997.

[52] Joseph o'Rourke. *Computational geometry in C*. Cambridge university press, 1998.

[53] Michael S Patterson, Brian C Wilson, and Douglas R Wyman. The propagation of optical radiation in tissue i. models of radiation transport and their application. *Lasers in Medical Science*, 6(2):155–168, 1991.

[54] Samuel Powell and Terence S. Leung. Highly parallel monte-carlo simulations of the acousto-optic effect in heterogeneous turbid media. *Journal of biomedical optics*, 17(4):045002, 2012.

[55] Scott A Prahl. A monte carlo model of light propagation in tissue. In *Dosimetry of laser radiation in medicine and biology*, volume 10305, page 1030509. International Society for Optics and Photonics, 1989.

[56] Nunu Ren, Jimin Liang, Xiaochao Qu, Jianfeng Li, Bingjia Lu, and Jie Tian. Gpu-based monte carlo simulation for light propagation in complex heterogeneous tissues. *Optics express*, 18(7):6811–6823, 2010.

[57] Mutsuo Saito and Makoto Matsumoto. SIMD-oriented fast mersenne twister: a 128-bit pseudorandom number generator. In *Monte Carlo and Quasi-Monte Carlo Methods*, pages 607–622, 2006.

[58] Fynn Schwiegelshohn, Tanner Young-Schultz, Yasmin Afsharnejad, Daniel Molenhuis, Lothar Lilge, and Vaughn Betz. Fullmonte: fast monte-carlo light simulator. In *Medical Laser Applications and Laser-Tissue Interactions IX*, volume 11079, page 1107910. International Society for Optics and Photonics, 2019.

[59] Haiou Shen and Ge Wang. A tetrahedron-based inhomogeneous monte carlo optical simulator. *Physics in Medicine & Biology*, 55(4):947, 2010.

[60] Hugo J. van Staveren, Marleen Keijzer, Tijmen Keesmaat, Harald Jansen, Wim J. Kirkel, Johan F. Beek, and Willem M. Star. Integrating sphere effect in whole-bladder-wall photodynamic therapy: Iii. fluence multiplication, optical penetration and light distribution with an eccentric source for human bladder optical properties. *Physics in Medicine & Biology*, 41(4):579, 1996.

[61] Lihong Wang, Steven L. Jacques, and Liqiong Zheng. Mcml — monte carlo modeling of light transport in multi-layered tissues. *Computer Methods and Programs in Biomedicine*, 47(2):131 – 146, 1995.

[62] Brian C. Wilson and Gerhard Adam. A monte carlo model for the absorption and flux distributions of light in tissue. *Medical Physics*, 10(6):824–830, 1983.

[63] Brian C. Wilson and Michael S. Patterson. The physics, biophysics and technology of photodynamic therapy. *Physics in Medicine and Biology*, 53(9):R61–109, May 2008.

[64] AN Yaroslavsky, PC Schulze, IV Yaroslavsky, R Schober, F Ulrich, and HJ Schwarzmaier. Optical properties of selected native and coagulated human brain tissues in vitro in the visible and near infrared spectral range. *Physics in Medicine & Biology*, 47(12):2059, 2002.

[65] Abdul-Amir Yassine, William Kingsford, Yiwen Xu, Jeffrey Cassidy, Lothar Lilge, and Vaughn Betz. Automatic interstitial photodynamic therapy planning via convex optimization. *Biomedical Optics Express*, 9:898, 02 2018.

[66] Abdul-Amir Yassine, Lothar Lilge, and Vaughn Betz. Tolerating uncertainty: photodynamic therapy planning with optical property variation. *Proc. SPIE*, 10860, 02 2019.

[67] Tanner Young-Schultz, Stephen Brown, Lothar Lilge, and Vaughn Betz. FullMonteCUDA: a fast, flexible, and accurate gpu-accelerated monte carlo simulator for light propagation in turbid media. *Biomed. Opt. Express*, 10(9):4711–4726, Sep 2019.

[68] Tanner Young-Schultz, Lothar Lilge, Stephen Brown, and Vaughn Betz. Using opencl to enable software-like development of an fpga-accelerated biophotonic cancer treatment simulator. In *The 2020 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pages 86–96, 2020.

[69] Leiming Yu, Fanny Nina-Paravecino, David R Kaeli, and Qianqian Fang. Scalable and massively parallel monte carlo photon transport simulations for heterogeneous computing platforms. *Journal of biomedical optics*, 23(1):010504, 2018.

[70] Christian Zoller, Ansgar Hohmann, Florian Foschum, Simeon Geiger, Martin Geiger, Thomas Peter Ertl, and Alwin Kienle. Parallelized monte carlo software to efficiently simulate the light propagation in arbitrarily shaped objects and aligned scattering media. *Journal of Biomedical Optics*, 23:1, 06 2018.