Computer Methods and Programs in Biomedicine 150 (2017) 97–105

Contents lists available at ScienceDirect

Computer Methods and Programs in Biomedicine

journal homepage: www.elsevier.com/locate/cmpb

Massively parallel simulator of optical coherence tomography of inhomogeneous turbid media

Siavash Malektaji[a], Ivan T. Lima Jr.[b], Mauricio R. Escobar I.[a], Sherif S. Sherif[a,*]

[a] University of Manitoba, Department of Electrical and Computer Engineering, 75A Chancellor's Circle, Winnipeg, Manitoba R3T 5V6, Canada
[b] North Dakota State University, Department of Electrical and Computer Engineering, 1411 Centennial Boulevard, Fargo, ND 58108-6050, USA

A R T I C L E I N F O

Article history:
Received 31 March 2016
Revised 31 July 2017
Accepted 7 August 2017

Keywords:
Optical coherence tomography
Light propagation in tissue
Parallel processing

A B S T R A C T

Background and objective: An accurate and practical simulator for Optical Coherence Tomography (OCT) could be an important tool to study the underlying physical phenomena in OCT such as multiple light scattering. Recently, many researchers have investigated simulation of OCT of turbid media, e.g., tissue, using Monte Carlo methods. The main drawback of these earlier simulators is the long computational time required to produce accurate results. We developed a massively parallel simulator of OCT of inhomogeneous turbid media that obtains both Class I diffusive reflectivity, due to ballistic and quasi-ballistic scattered photons, and Class II diffusive reflectivity due to multiply scattered photons.
Methods: This Monte Carlo-based simulator is implemented on graphic processing units (GPUs), using the Compute Unified Device Architecture (CUDA) platform and programming model, to exploit the parallel nature of propagation of photons in tissue. It models an arbitrary shaped sample medium as a tetrahedron-based mesh and uses an advanced *importance sampling* scheme.
Results: This new simulator speeds up simulations of OCT of inhomogeneous turbid media by about two orders of magnitude. To demonstrate this result, we have compared the computation times of our new parallel simulator and its serial counterpart using two samples of inhomogeneous turbid media. We have shown that our parallel implementation reduced simulation time of OCT of the first sample medium from 407 min to 92 min by using a single GPU card, to 12 min by using 8 GPU cards and to 7 min by using 16 GPU cards. For the second sample medium, the OCT simulation time was reduced from 209 h to 35.6 h by using a single GPU card, and to 4.65 h by using 8 GPU cards, and to only 2 h by using 16 GPU cards. Therefore our new parallel simulator is considerably more practical to use than its central processing unit (CPU)-based counterpart.
Conclusions: Our new parallel OCT simulator could be a practical tool to study the different physical phenomena underlying OCT, or to design OCT systems with improved performance.

1. Introduction

An increasing number of biomedical applications greatly benefit from optical coherence tomography (OCT) imaging [1–8]. This high-resolution, non-invasive technique is ideal for, but not limited to, imaging soft tissues. Further studies of the underlying physical phenomena and design of novel OCT systems could be carried analytically or through experiments. However, a practical computer simulator of such systems could enable and/or facilitate such efforts. In this paper, we describe a massively parallel simulator of OCT, (which we call OCT-MPS) of inhomogeneous turbid media using a graphics processing unit (GPU) that is more than one order of magnitude faster than its central processing unit (CPU)-based counterpart [9].

Due to its accuracy and applicability to arbitrary media, Wilson and Adam introduced the Monte Carlo (MC) method to solve the Radiative Transport Equation (RTE) that has become a widely accepted approach to model photon migration in tissue [10]. Later Wang et al. [11], developed a well-known MC simulation of light transport in multilayered turbid media (MCML). A broad review of methods to simulate light transport in turbid media can be found in Zhu and Liu [12].

The first MC simulator of OCT imaging was introduced by Smithies et al. in 1998, but it was limited to OCT signals from single layered media [13]. In 1999, Yao and Wang developed an MCML-based simulator of OCT of multilayered media that used an importance sampling method to reduce the variance of simulated OCT signals from a narrow tissue slice [14]. Lima et al.

* Corresponding author.
E-mail address: Sherif.Sherif@umanitoba.ca (S.S. Sherif).

http://dx.doi.org/10.1016/j.cmpb.2017.08.001
0169-2607/© 2017 Published by Elsevier Ireland Ltd.

improved this simulator by introducing an advanced *importance sampling* scheme that decreased the computation time of OCT signals from multilayered tissue by two orders of magnitude [15,16]. In 2007, Kirillin et al. developed a simulator of OCT of non-planar multilayered media [17]. In this approach, the boundaries of layers inside the media were modeled as mathematical functions, e.g., sinusoidal functions. This simulator has been used to simulate OCT imaging of human enamel in which its boundaries were modeled as non-parallel planes [7]. Dolganova et al. used this simulator by Kirillin et al. to simulate OCT images of skin with dysplastic nevus [18]. Moreover, Shlivko et al. used this simulator to analyze the structure and optical parameters of layers of skin with thick and thin epidermis [19]. Kirillin et al. also used this simulator to analyze the Speckle statistics in OCT signals. Using the Monte Carlo simulation and experimental results, they showed that Speckle could be modeled with Gamma distribution [20]. Periyasamy and Pramanik developed a Monte Carlo simulator for OCT of multilayered media with embedded objects (such as a sphere, cylinder, ellipsoid, and cuboid) [21] and used importance sampling to reduce the computation time of simulations [22]. In his Ph.D. thesis, Sinan Zhao has developed a Monte Carlo based TD-OCT and FD-OCT simulator [23]. To model media with complex geometry, he subdivided media into cuboidal voxels. He used importance sampling and parallelization to reduce the computation time of his simulator. The disadvantage of using cubical voxel is its inaccuracy in the estimation of the specular (i.e., Fresnel) reflection from tilted surfaces [23].

Recently an MC-based simulator of OCT of inhomogeneous turbid media with arbitrary spatial distributions was introduced by Malektaji et al. [9]. In this OCT simulator, an arbitrary object was represented as a tetrahedron mesh that could model an arbitrary shape with any desired accuracy. An advantage of using tetrahedrons as the building blocks of an arbitrary object is that it minimizes the computation cost of OCT simulation compared to other possible building blocks [9]. This tetrahedron mesh could be obtained using mesh generator applications such as NETGEN [24]. Even though Malektaji et al. used an advanced *importance sampling* scheme to reduce computations, simulation of a *B-scan* of a sphere inside a slab, using 10^7 photon packets, required approximately 360 h on a typical central processing unit (CPU) -based desktop computer [9].

In this paper, we describe a massively parallel implementation of the CPU-based simulator described in [9]; this parallel implementation resulted in a reduction of simulation time by more than one order of magnitude. Therefore, it would be considerably more practical to use our new parallel implementation, i.e., OCT-MPS, compared to its CPU-based counterpart. Our massively parallel implementation runs on graphics processing units (GPUs), using the Compute Unified Device Architecture (CUDA) platform and programming model by NVIDIA [25]. In the following sections, we give an overview of the core concepts of parallel programming with CUDA before explaining in detail the parallel implementation of our simulator. Also, we describe the simulation of OCT imaging of inhomogeneous objects, our GPU memory utilization and our generation of parallel random numbers. We also provide portability guidelines for running our simulator on different CUDA GPUs. Finally, we present simulation results of OCT of nonplanar inhomogeneous media with arbitrary shapes and discuss the reduction in computation time.

2. Simulation of OCT signals from of an inhomogeneous object

Following the procedure to simulate OCT signals from an object consisting of arbitrary shaped regions, in [9], each region is defined with its optical parameters; scattering coefficient μ_s, absorption coefficient μ_a, refractive index n, and an anisotropy factor g. Simulation of light propagation is modeled with a number of photon packets undergoing a random walk inside this object. During such random walks, the photon packets experience absorption and scattering events, where one photon packet splits into two packets traveling in different directions. To simulate OCT signals, a large number of photon packets are launched with the same initial position representing a thin beam incident perpendicular to the surface of the medium. The photon packets are traced inside the medium according to the rules described in references [11,14]. The flowcharts of the process of tracing photon packets inside the medium are shown in Fig. 1.

A collecting fiber, with a specific acceptance angle, $\theta_{\max}$, and radius, $d_{\max}$, is located at the top of the medium to detect backscattered photons. The photons are collected by the fiber probe if their angle and position are within the acceptance angle of the probe. Three types of photons are collected from a depth z by the fiber: (1) ballistic photons that are single scattered, (2) quasi-ballistic photons that are multiply scattered within the coherence length of the optical source, and (3) multiply scattered photons beyond the coherence length of the optical source. Ballistic and quasi-ballistic photons contribute to Class I diffusive reflectivity. The multiply scattered photons beyond the coherence length contribute to Class II diffusive reflectivity. It has been shown that Class II diffusive reflectivity is the main limiting factor in imaging depth of OCT [26,27].

3. Implementation of our OCT massively parallel simulator

The implementation of our OCT massively parallel simulator (OCT-MPS) addresses the high computational load of processing a typically large number of samples required for a suitable accuracy. In our simulator, a large number of samples, i.e., photon packets, are launched simultaneously and traced independently. Thus, we should expect a considerable reduction in computation time due to parallel implementation. In this Section, we describe the CUDA programming environment, the design and implementation of OCT-MPS, including photon packet tracing, random number generation, and memory utilization. We also discuss its portability across different GPUs.

3.1. CUDA programming environment

Compute Unified Device Architecture (CUDA) is a platform and programming model developed by NVIDIA for graphics processing units (GPUs). Its Application Programming Interface (API) offers extensions for many industry standard programming languages, like the C language, with CUDA's accelerated libraries and compiler directives [28]. The CUDA environment allows one to simultaneously develop code intended for both central processing unit (CPU) and GPU.

The basic unit of execution in a CUDA program is a *thread*, which runs independently and concurrently with a big number of other similar threads. CUDA uses an execution model known as Single Instruction, Multiple Thread (SIMT), which allows independence between threads. SIMT allows each thread to have different execution path, separate registers, and possibly execution divergence that would result in different values in instruction address counters [29]. The most common execution divergence occurs with control flow statements (e.g., *if-then-else, switch*), where threads that are not meeting a logical condition are stopped until all other threads fulfilling this condition finish executing this condition [30,31]. To minimize performance penalties, CUDA caches data from stopped threads for fast access. However, avoiding execution divergence and minimizing execution times of all logical conditions are preferable but not always simple to implement [32,33].

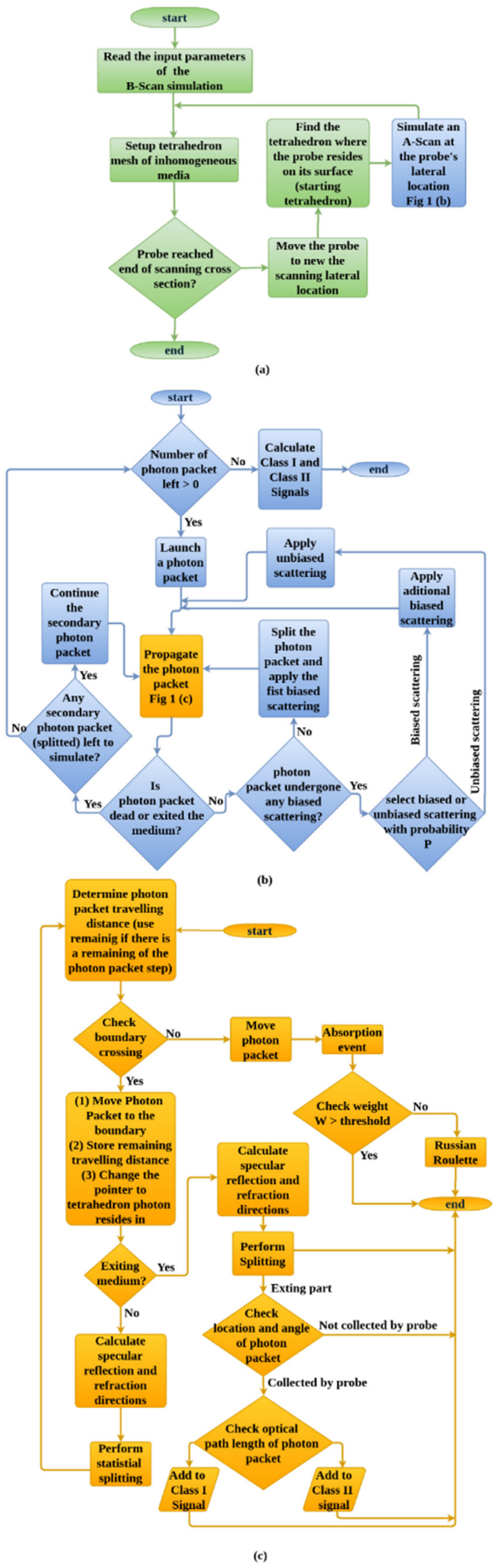
start
Read the input parameters of the B-Scan simulation
Setup tetrahedron mesh of inhomogeneous media
Find the tetrahedron where the probe resides on its surface (starting tetrahedron)
Simulate an A-Scan at the probe's lateral location Fig 1 (b)
Probe reached end of scanning cross section?
Move the probe to new the scanning lateral location
end
(a)
start
Number of photon packet left > 0
No
Calculate Class I and Class II Signals
end
Yes
Launch a photon packet
Apply unbiased scattering
Apply aditional biased scattering
Continue the secondary photon packet
Propagate the photon packet Fig 1 (c)
Split the photon packet and apply the fist biased scattering
Yes
Any secondary photon packet (splitted) left to simulate?
No
Biased scattering
Unbiased scattering
No
Is photon packet dead or exited the medium?
Yes
No
photon packet undergone any biased scattering?
Yes
select biased or unbiased scattering with probability P
(b)
Determine photon packet travelling distance (use remainig if there is a remaining of the photon packet step)
start
Check boundary crossing
No
Move photon packet
Absorption event
Yes
Check weight W > threshold
No
Russian Roulette
Yes
end
(1) Move Photon Packet to the boundary (2) Store remaining travelling distance (3) Change the pointer to tetrahedron photon resides in
Calculate specular reflection and refraction directions
Perform Splitting
Exiting medium?
Yes
Exting part
No
Check location and angle of photon packet
Not collected by probe
Collected by probe
Calculate specular reflection and refraction directions
Check optical path length of photon packet
Perform statistial splitting
Add to Class I Signal
Add to Class II signal
(c)

Fig. 1. Flowcharts describing the tracing of a photon packet inside an inhomogeneous object. The flowchart of photon packet tracing can be divided into three flowcharts: (a) Initializing the simulation by setting up the tetrahedron mesh data structure representing the medium (b) Applying Importance Sampling (c) Tracing photon packets inside the medium.

In the CUDA environment, the GPU hardware is identified as *device* and the CPU hardware is known as the *host*. The code executed on a GPU is called a *kernel,* where it consists of a typically large number of identical *threads*. Whenever the *host* code encounters a *kernel* function, the later will be executed by the GPU's processors in an asynchronous manner. Once this *kernel* is launched, control is immediately relinquished to the *host* and further *host* code will be executed by the CPU alongside the GPU execution. Any function called and executed by the GPU is known as *device function*. Data needed by a *kernel* has to be moved back and forth from the *host's* memory to the *device's* memory. We note that CPU and GPU memory spaces are completely separate and have no direct access to each other.

A GPU has different memory types that are configured in a hierarchical structure. The *global memory* is available to all threads, but it has slow access, i.e., about 400 to 600 clock cycles. The *register* is a relatively small memory space assigned to each *thread* with very fast, i.e., one clock cycle, and exclusive access. *Constant memory* is available to all threads and is read-only. *Local memory* is only available for groups of 32 threads, called a *warp,* and is where all variables actively used by this *warp* are allocated. The compiler sets this *local memory* space according to the *kernel*'s requirement at execution time. *Constant* and *local* memories are part of the *global memory's* physical space but are cached to speed up their access times.

A *kernel* function, which consists of a typically large number of identical *threads*, is executed with a given *kernel configuration* that sets up the GPU's memory space and its computational resources needed during execution. The *kernel configuration* also defines a *grid* of *thread blocks* that are identified by a number of *thread blocks.* Every *thread block* could include more than one *warp.*

To avoid race conditions [34], where more than one thread needs to write to the same memory location, CUDA offers special operators called *atomic* operators. These *atomic* operators have the same precision as their counterpart CPU based operators.

Compute capability is NVIDIA's classification of CUDA capable GPU hardware. It is an incremental number to group different hardware with similar computational capabilities, i.e., types of arithmetic operators and their precision. *Compute capability* is typically backward compatible and does not refer to the hardware's computational speed.

3.2. Design of our OCT massively parallel simulator

Our OCT-MPS, which simulates OCT of inhomogeneous turbid media, is implemented in NVIDIA's CUDA environment using extensions of the C language. Our GPU configuration and code design are similar to the GPU-MCML simulator [35]. Computational intensive functions, i.e., photon packet tracing, are implemented in a kernel function, while less computational intensive ones, i.e., I/O data handling, are executed in the host. Three configuration text files are read at program initialization: *object.txt* which stores the optical parameters of the object, *mesh.txt* which stores coordinates of the tetrahedron mesh elements and *sim_param.txt* which stores user defined simulator parameters, e.g., the coherence length of the optical source, collection angle of fiber probe, etc. Fig. 2 shows a flowchart of the OCT-MPS algorithm.

3.3. Tracing photon packets and recording OCT signals

As shown in Fig. 2, each thread traces the complete path of a photon packet from its launch into the object until the end of its life, or when it exits the object. When a photon packet dies, if more photon packets need to be traced, another one will be launched using the same thread. Therefore, the number of photon

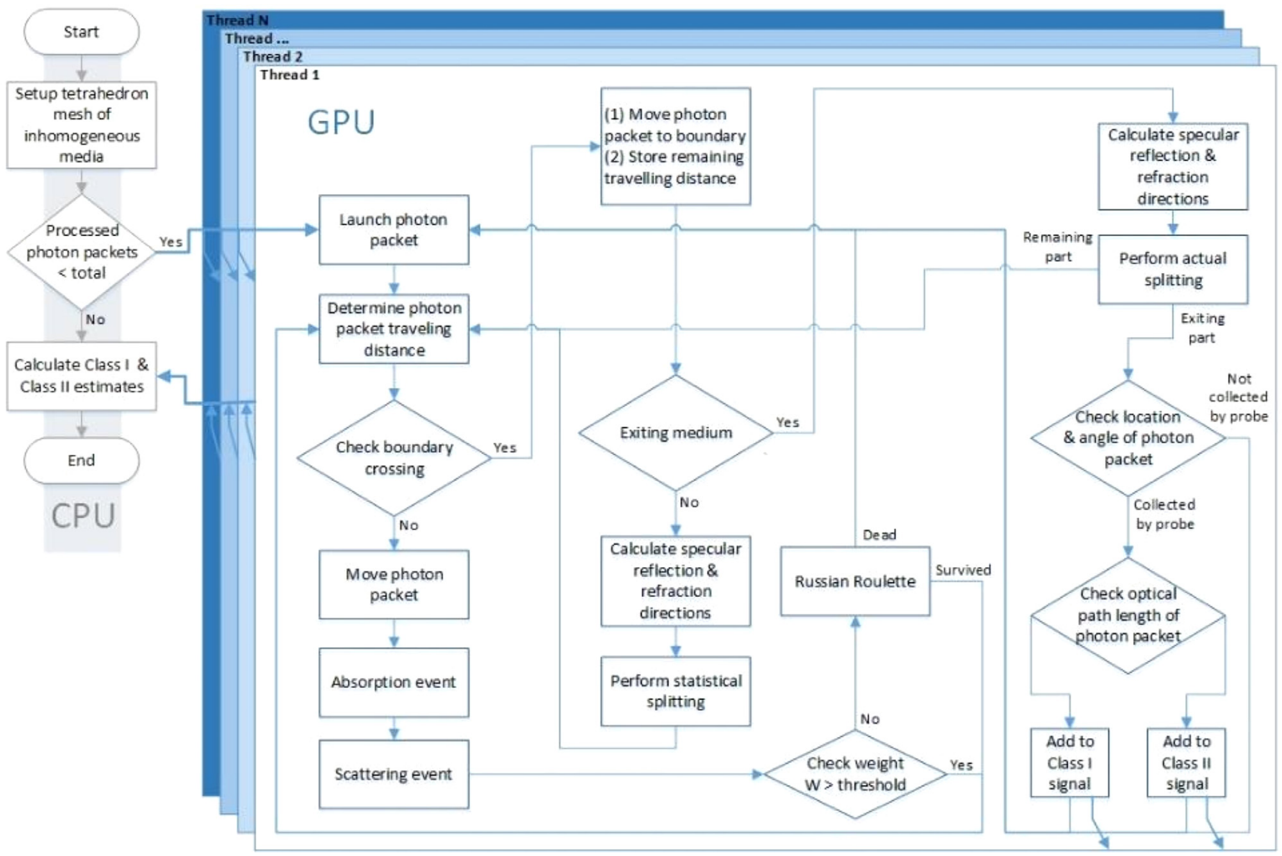
Start
Setup tetrahedron mesh of inhomogeneous media
Processed photon packets < total
Yes
No
Calculate Class I & Class II estimates
End
CPU
Thread N
Thread ...
Thread 2
Thread 1
GPU
Launch photon packet
Determine photon packet traveling distance
Check boundary crossing
Yes
No
Move photon packet
Absorption event
Scattering event
(1) Move photon packet to boundary
(2) Store remaining travelling distance
Exiting medium
Yes
No
Calculate specular reflection & refraction directions
Perform statistical splitting
Russian Roulette
Dead
Survived
Check weight W > threshold
No
Yes
Calculate specular reflection & refraction directions
Remaining part
Perform actual splitting
Exiting part
Check location & angle of photon packet
Not collected by probe
Collected by probe
Check optical path length of photon packet
Add to Class I signal
Add to Class II signal

Fig. 2. Flowchart of the OCT-MPS algorithm. A CPU starts the simulation by setting up the tetrahedron mesh data structure, and by launching a CPU thread per GPU card. The GPU cards simulate the photon packet tracing inside the medium. The CPU threads then gather the simulation results from the GPU cards and calculate both Class I and Class II signals.

packets traced per thread depends on the total number of photons to be simulated and the maximum number of simultaneous threads that could run on the GPU. This number of simultaneous threads depends on the GPU's *compute capability* and is set as an initial configuration of the kernel.

All functions that trace a photon packet in a thread are implemented as device functions and are called by the CPU-launched kernel: *OCTMPSKernel*. The names of these functions and the names of their variables are self-descriptive, where objects that are common between the CPU and the GPU use the "CamelCase" naming convention [36], and variables used in the GPU only have lowercase names.

1. *OCTMPSKernel*: is the main function in our simulator and controls the flow of the photon packet life. It initializes a data structure, *tstates*, to store parameters of the traced photon packet, e.g., its weight, location, and direction, in addition to parameters of the tetrahedron where it is located. It also initializes an identical data structure to store parameters of a split photon packet that could arise from a potentially biased scattering. Also, it creates the following variables (1) *is_active*: a flag to indicate if a thread is active, which occurs when the number of packets left to be simulated is more than the maximum simultaneous number of threads, (2) seed vector for random number generators. This function also uses the structure *DeviceMem* to keep track of simulation results, generated random numbers, and the number of traced photon packets. A Russian roulette routine is also implemented in this function for photon packets with very low weights. It determines whether to terminate or increase the weight, of such photon packets. Also if the number of photon packets left to simulate is less than the total number of available threads, this function sets the *is_active* flags corresponding to these idle threads to false. This would prevent these idle threads from utilizing GPU resources.
2. *RestoreThreadState*: a device function that restores the state of each thread, e.g., optical parameters of the photon packet being traced and its random number streams.
3. *ComputeStepSize*: determines the random traveling distance of a photon packet. It also updates the photon packet parameter's structure, *tstates*.
4. *Hop*: moves the photon packet according to the traveling distance computed previously.
5. *HitBoundary*: checks if a photon packet crossed the boundary of the tetrahedron where it is located. The result of this function determines one of two logical branches in the code corresponding to a photon packet crossing, or not crossing, its tetrahedron's boundary.
6. *FastReflectTransmit*: if a photon packet crosses its tetrahedron boundary, this function checks if the neighboring tetrahedron belongs to a different region inside the medium. If true, it calculates the reflectance and transmittance at this boundary. It is the most computationally expensive device function. It calculates the number of photons to be reflected towards the probe and the number of photons to be transmitted to the adjacent tetrahedron. This splitting of the photon packet involves updating its weight and directional cosines. This function performs a read from the GPU's global

memory, in addition to atomic operations to record Class I and Class II signals.

7. *Drop*: if a photon packet did not cross its tetrahedron boundary, this function updates its weight.
8. *SpinBias*: updates the scattering direction of a photon packet, by introducing a bias that increased the probability of the photon packet is scattered towards the optical detector [15]. It is the second most computational expensive device function.
9. CopyPhotonStruct: stores optical parameters of the back-scattered (biased scattering towards the optical detector) portion of an original photon packet that was split at a boundary. These stored optical parameters allow simulation of this back-scattered portion after tracing the transmitted portion of the original packet.
10. *Spin*: updates the scattering angle of a photon packet without applying any bias.
11. *LaunchPhoton*: launches a new photon packet and resets its structure to the default values.
12. *SaveThreadState*: saves parameters related to the active thread, e.g., random number streams, and the parameters of currently traced photon packet, e.g., weight, location and direction, to the *tstates* structure.

Given that some photon packet traces will finish their simulation before the others, a mechanism is implemented to minimize thread stalls due to long-lived traces. Each thread keeps track of the total number of steps that a photon packet underwent since its incidence into the medium. It then compares this number to a user defined constant. When the maximum allowed number of steps, MAX_NUM_STEPS, is reached, *OCTMPSKernel* saves the states of these long-lived active threads to CPU memory. Then the program launches a new kernel with the maximum number of possible threads, including the saved long-lived threads, to ensure higher utilization of the GPU's hardware at any given time.

After the required number of photon packets are simulated, results are copied to CPU memory. The CPU then evaluates both Class I and Class II *A-scans* and saves them to a file. The above process of simulating an *A-scan* is repeated at different locations until the Class I and Class II *B-scans* are obtained.

3.4. OCT-MPS memory utilization

A kernel, *InitThreadState,* allocates and initializes all data structures to be used by *OCTMPSKernel*, using *cudaMalloc* and *cudaMemset* CUDA commands. The coordinates of individual tetrahedrons that comprise the tetrahedron mesh are stored as a linked list in *global memory*. Access to this linked list results in a considerable implementation delay. Individual tetrahedrons are identified by an ID, stored in the structure *tstates* located in *local memory,* to track the tetrahedron where a traced photon packet is located. The optical parameters of every distinct region of the inhomogeneous medium are stored in *constant memory*. To reduce the number of slow accesses to *global memory*, the optical parameters of a region are cached in *tstates,* which is the structure associated with the currently traced photon packet.

3.5. Random number generator

The random number generation in a parallel environment is not a trivial matter. We need to avoid any type of inter-thread and intra-thread correlations, the generator has to have a long period and low computational cost. The random number generator used in our simulator is the Multiply-With-Carry (MWC) due to its light computation weight and hardware resources requirement [37]. OCT-MPS uses an improved implementation of MWC by Alerstam et al. [35] that reduces possible correlations between the threads. The implementation by Alerstam et al. uses one stream of random numbers for all functions involved in a trace. OCT-MPS creates an independent stream of random numbers for the spinning functions, another for the reflection and refraction function, and a third one for the rest of functions where random numbers are needed.

The seeds for the different random number streams previously described can be obtained from the system's clock or set manually assuming that the three are different. Each thread in the device will have a unique, safe prime number as the seed. A safe prime number is a prime number with a form, $2p+1$, where p is also a prime number. A list of safe prime numbers was created previously and is available as a text file with our implementation. Our OCT-MPS implementation of the MWC uses nine registers per thread.

3.6. Portability across GPUs

OCT-MPS is ready to work with many hardware configurations, with support for a wide range of CUDA *compute capability* (versions 1.3 to 3.5), as well as with single and multiple GPUs. The different pre-configured parameters guarantee the maximum utilization of the available GPU resources with the specific data while overlapping as much as possible the latency produced by data transfers. These parameters can be easily modified or replaced in future releases on future CUDA platforms.

From *compute capability* 1.3 onwards, CUDA supports double precision floating-point numbers that we use in our implementation. Due to the high computational cost of using double precision addition, CUDA has not implemented an optimized operator for atomic addition of double precision numbers in global memory. To overcome this problem, an atomic addition function was implemented using the available atomic operator for compare-and-swap method (*atomicCAS*) as described in reference [28].

Some optimized operators are only available in higher *compute capability* versions. The OCT-MPS macros are defined to use optimized operators whenever possible, but more computationally expensive operators could be used whenever greater precision is required.

4. Results

4.1. Simulation Case 1

As the first simulation example, we present a simulated OCT image of an object consisting of a slab containing two spheres and a tri-axial ellipsoid. The slab is extended for 3 mm in each lateral dimension (x axis and y axis) and 1 mm thick in the axial direction (z axis). A tri-axial ellipsoid is placed with its center at position (0.0, 0.0, 0.02) mm. The three vectors representing the axes of this ellipsoid are given by $v_1 = [0.006\sin(60o),\ 0,\ -0.006\cos(60o)]^T$ mm, $v_2 = [0,\ 0.1,\ 0]^T$ mm, and $v_3 = [0.017\sin(30°),\ 0,\ 0.017\cos(30°)]^T$ mm. Two spheres with radius of 0.005 mm are placed at positions (0.008, 0.0, 0.01) mm and (−0.008, 0.0, 0.03) mm. An abstract view of this object can be seen in Fig. 3(a). The tetrahedron mesh of this object, generated with NETGEN [24], is shown in Fig. 3(b). This tetrahedron mesh has a total of 5248 vertices and 29,697 tetrahedrons. The optical parameters of this object are shown in Table 1.

The simulated optical fiber probe used has radius 1 μm, acceptance angle 5°, and an optical source with coherence length $l_c = 15\ \mu$m. The biasing coefficient of the implemented importance sampling is $a = 0.925$ and additional bias probability $p = 0.5$, as in [9].

Fig. 4 shows the obtained image generated by OCT-MPS for Class I reflectivity-based *B-Scan* OCT of the inhomogeneous media described above. This simulation was obtained using the

Table 1
Optical parameters of the object.

Medium	Absorption coefficient μ_a (cm^{-1})	Scattering coefficient μ_s (cm^{-1})	Anisotropy factor G	Refractive index N
Slab	1.5	60	0.9	1
Spheres	3	120	0.9	1
Ellipsoid	3	120	0.9	1

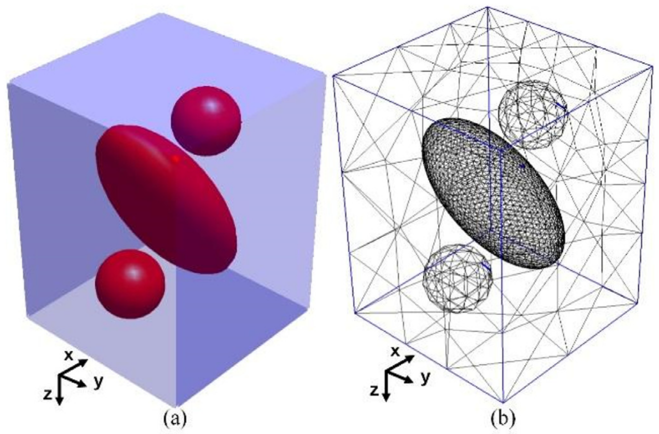

Fig. 3. Abstract view and tetrahedron mesh of the object used in *Simulation Case 1*. Inhomogeneous medium: Ellipsoid and two spheres inside a slab. (a) Spatial structure. (b) Tetrahedron mesh.

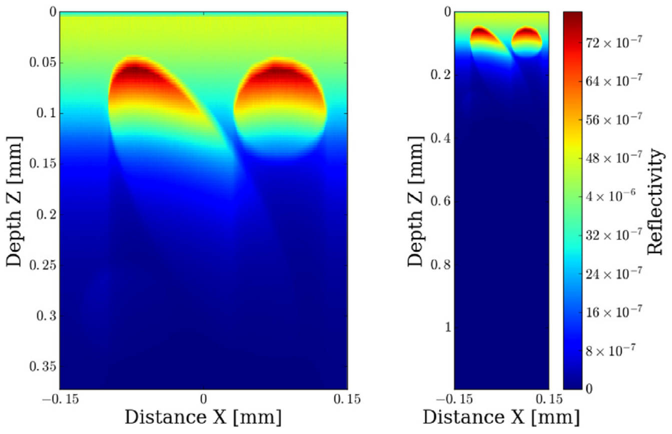

Fig. 4. Class I reflectivity-based *B-scan* OCT images. Class I *B-scan* OCT image (right), along with its enlarged version (left), generated by OCT-MPS.

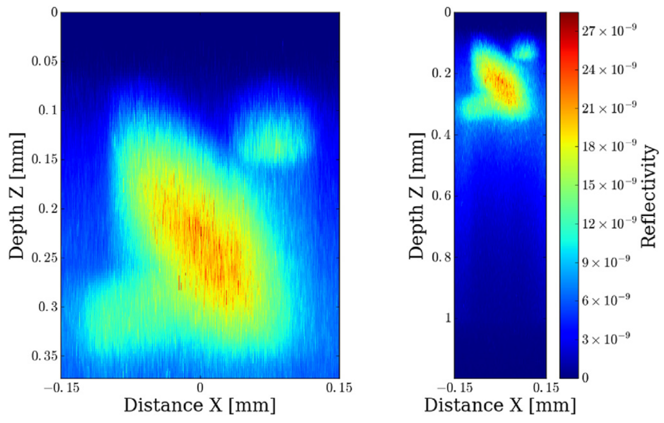

Fig. 5. Class II reflectivity-based *B-scan* OCT images. Class II *B-scan* OCT image (right), along with its enlarged version (left), generated by OCT-MPS.

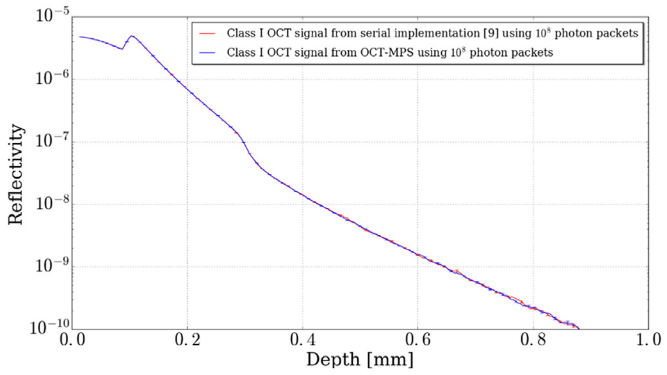

Fig. 6. Comparison of the Class I signal of a sample A-Scan obtained by serial simulator and OCT-MPS. *A-scans* representing Class I reflectivity-based OCT signals from the above simulation obtained by OCT-MPS (blue) and by the serial implementation (red) [9] at the lateral position x = 0.0 mm of the object (A-scan) using 10^8 photon packets. (For interpretation of the references to colour in this figure legend, the reader is referred to the web version of this article.)

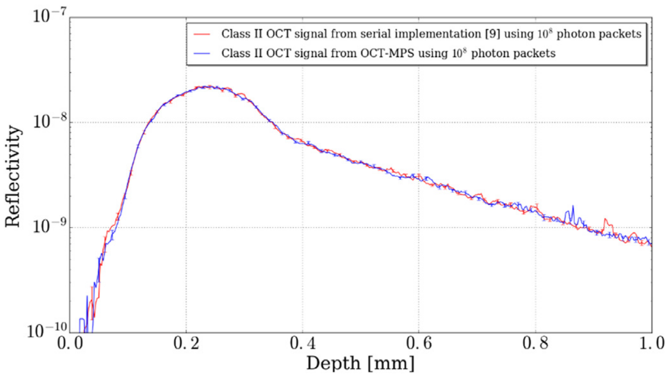

Fig. 7. Comparison of the Class II signal of a sample A-Scan obtained by serial simulator and OCT-MPS. *A-scans* representing Class II reflectivity-based OCT signals from the above simulation obtained by OCT-MPS (blue) and from the serial implementation (red) [9] at the lateral position x = 0.0 mm of the object (A-scan) using 10^8 photon packets. (For interpretation of the references to colour in this figure legend, the reader is referred to the web version of this article.)

lateral simulated resolution of 500 equidistant *A-scans* along the *x*-axis from $x = -0.15$ mm to $x = 0.15$ mm.

Similarly, Fig. 5 shows a simulated *B-scan* of Class II OCT image that was generated by OCT-MPS.

Both Figs. 4 and 5 were obtained using the Tesla K40c with compute capability 3.5 in the Kepler architecture. The K40c card has 2880 scalar processor (15 streaming multiprocessors with 192 CUDA cores each) running at 745 MHz, and total global memory of 12 GB (DDR5).

Figs. 6 and 7 show an A-Scan of Class I reflectivity and Class II reflectivity, respectively, which are obtained at an arbitrary chosen lateral position x = 0.0 mm of the object using 10^8 photon packets, by OCT-MPS and its serial implementation [9]. Both Figs. 6 and 7 show that OCT-MPS produced results that are in excellent agreement with those produced by its serial implementation [9]. The serial code was executed on a computer with an Intel® Xeon® CPU E5-2660 v2 with clock speed at 2.20 GHz and 64 GB of RAM running on a 64-bit Ubuntu 15.10 operating system.

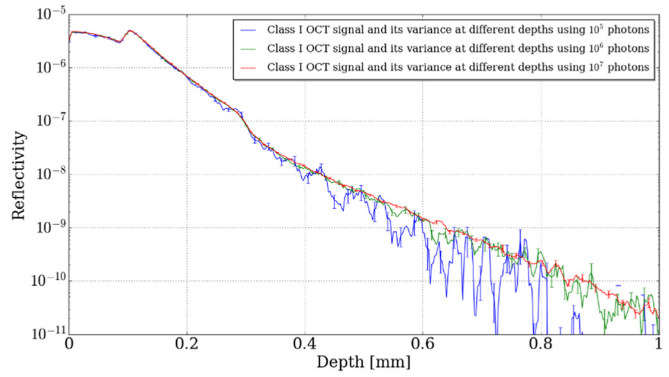

Fig. 8. Comparison of Class I OCT signals obtained using different numbers of photon packets. Class I OCT signals and their confidence intervals using 10^7 (red), 10^6 (green), and 10^5 (blue) photon packets. (For interpretation of the references to colour in this figure legend, the reader is referred to the web version of this article.)

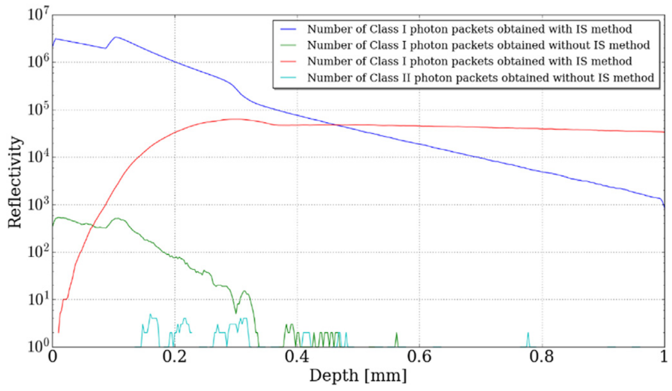

Fig. 9. Comparison of collected number of Class I and Class II photon packets. Number of photon packets collected by the probe that contributes to the Class I and Class II OCT signals at different depths with and without using Importance Sampling (IS) method.

The accuracy of Monte Carlo simulations is generally quantified by the variance of the numerical results. Fig. 8 shows Class I reflectivity (*A-scan*) of the object and the corresponding confidence interval (CI) at an arbitrary lateral position using 10^5, 10^6, and 10^7 photon packets. The CI of the Monte Carlo estimates of Class I and Class II, $\mathrm{CI}_{1,2}(z)$, are defined as

$$\mathrm{CI}_{1,2} \equiv [R_{1,2}(z) - \sigma_{1,2}(z),\ R_{1,2}(z) + \sigma_{1,2}(z)].$$

We note from Fig. 8 that, as expected, the accuracy of this OCT simulator (serial and parallel implementations) decreases with the imaging depth [14]. Moreover, we observed that at least 10^7 photon packets are needed to simulate the Class I signal up to a depth of 1 mm.

Fig. 9 shows the number of photon packets collected by the probe that contributes to Class I and Class II OCT signal at different depths by using Importance Sampling (IS) method and without using it. As could be seen in Fig. 9, by using the importance sampling method the number of photon packets collected by the probe that contributes to Class I and Class II signals is increased by more than four orders of magnitudes which result in high efficiency of the simulator.

4.2. Simulation Case 2

As the second simulation example, we present a simulated OCT image of an object consisting of a slab containing a tri-axial ellipsoid. The slab is extended for 3 mm in each lateral dimension (x axis and y axis) and 0.4 mm thick in the axial direction (z axis). A tri-axial ellipsoid is placed with its center at position (0.0, 0.0, 0.2) mm. The three vectors representing the axes of this ellipsoid are given by $v_1 = [0.2, 0, 0]^T$ mm, $v_2 = [0, 0.1, 0]^T$ mm, and $v_3 = [0, 0, 0.1]^T$ mm. An abstract view of this object can be seen in Fig. 10(a). The tetrahedron mesh of this object, generated with NETGEN [24], is shown in Fig. 10(b). This tetrahedron mesh has a total of 1275 vertices and 6673 tetrahedrons. The optical parameters of this object are shown in Table 2.

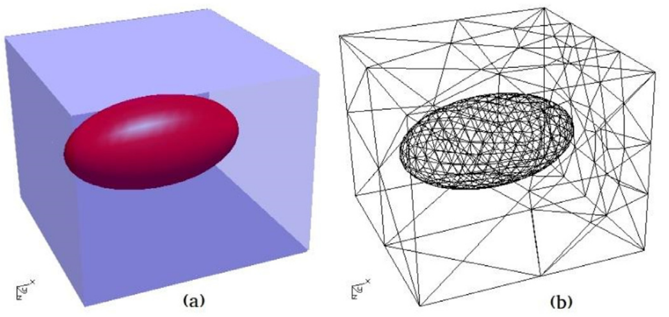

Fig. 10. Abstract view and the tetrahedron mesh of object used in *Simulation Case 2*. Inhomogeneous media: Horizontal ellipsoid spheres inside a slab. (a) Spatial structure. (b) Tetrahedron mesh.

The simulated optical fiber probe and the importance sampling parameters that were used are same parameters as in the first example. Fig 11(a) and (b) present the obtained image generated by OCT-MPS for Class I reflectivity-based *B-Scan* OCT of the inhomogeneous media described above. This simulation was obtained using the lateral simulated resolution of 100 equidistant *A-scans* along the x-axis from $x = -0.25$ mm to $x = 0.25$ mm.

5. Discussion

To compare the computation times of the OCT-MPS and the serial implementation, we ran both of the above simulation examples on different platforms. We ran the serial code on a computer with an Intel® Xeon® CPU E5-2660 v2 with clock speed at 2.20 GHz and 64 GB of RAM running on a 64-bit Ubuntu 15.10 operating system. We also ran the OCT-MPS on (I) a computer with an NVIDIA Tesla K40c card with compute capability 3.5 in the Kepler architecture. The K40c card has 2880 scalar processor (15 streaming multiprocessors with 192 CUDA cores each) running at 745 MHz, and total global memory of 12 GB (DDR5). (II) Amazon Web Services (AWS) p2.8xlarge instance with 8 NVIDIA Tesla K80 Cards with compute capability of 3.7 in Kepler architecture. The K80 card has two Tesla GK210 GPUs. Each Tesla GK210 GPU has 2496 processors running at 560 MHz and total global memory of 12 GB (DDR5). (III) AWS p2.16xlarge instance with 16 NVIDIA Tesla K80 cards.

Figs. 12 and 13 show the comparison of total B-Scan simulation time of the above-mentioned simulation examples on these platforms. We observed an improvement in the processing speed from the serial implementation to the MPS implementation of the code of close to two orders of magnitude.

As it can be seen in Figs. 12 and 13, the speed up gain is almost linearly related to the number of GPU cards used. This is because GPU cards are running independently in OCT-MPS with no communication among them. Each GPU card simulates the same number of photons (equal to the total number of photons divided by the total number of cards) independently of the other GPU card.

Figs. 14 and 15 show comparison of the computation time per A-scan for the first and second example respectively obtained on mentioned platforms. As shown in Figs. 14 and 15 the computation time per *A-scan* depends on the optical parameters of the media. For example, in regions with higher scattering coefficient μ_s the

Table 2
Optical parameters of the object.

Medium	Absorption coefficient μ_a (cm^{-1})	Scattering coefficient μ_s (cm^{-1})	Anisotropy factor g	Refractive index N
Slab	1.5	60	0.9	1
Ellipsoid	3	120	0.9	1

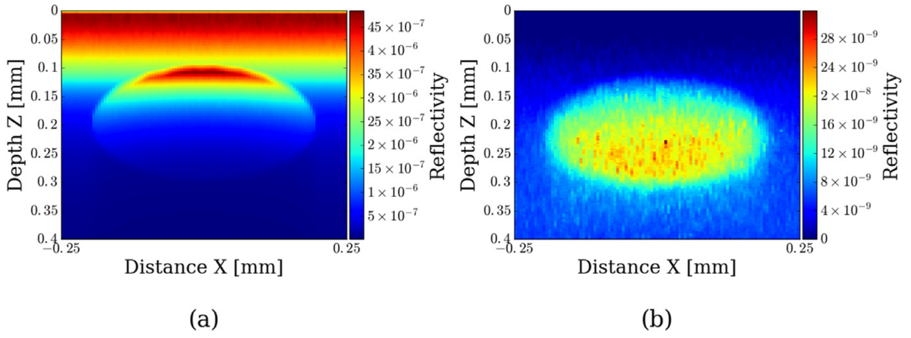

Fig. 11. Class I reflectivity-based *B-scan* OCT images. (a) Class I *B-scan* OCT image generated by OCT-MPS (b) Class II *B-scan* OCT image generated by OCT-MPS.

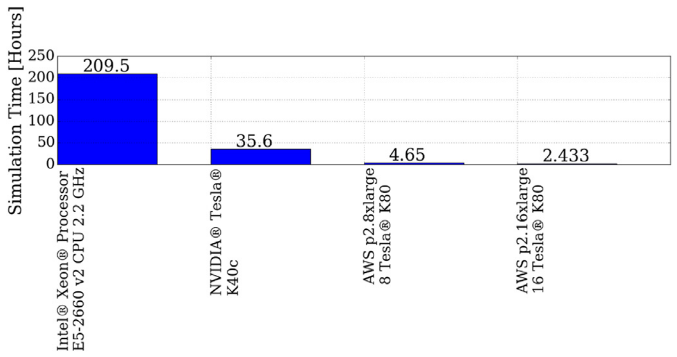

Fig. 12. Comparison of simulation time of the OCT B-Scan images of the object from *Simulation Case 1*. The computation times of simulating B-Scan image of the object in *Simulation Case 1* using serial simulator, OCT-MPS on NVIDIA Tesla K40c card, OCT-MPS on Amazon AWS p2.8xlarge instance and OCT-MPS on Amazon AWS p2.16xlarge instance are compared. As shown the simulation times decrease almost linearly with the number of GPU cards used.

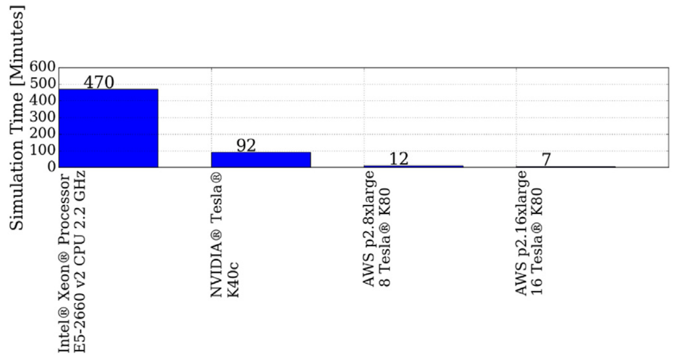

Fig. 13. Comparison of simulation time of the OCT B-Scan images of the object from *Simulation Case 2*. The computation times of simulating B-Scan image of the object in *Simulation Case 2* using serial simulator, OCT-MPS on NVIDIA Tesla K40c GPU card, OCT-MPS on Amazon AWS p2.8xlarge instance and OCT-MPS on Amazon AWS p2.16xlarge instance are compared. As shown the simulation times decrease almost linearly with the number of GPU cards used.

simulation will perform an increased number of scattering calculations which will result in longer processing time.

In *Simulation Case 1*, an A-scan simulation of regions with the lowest scattering coefficient takes about 13 s in the parallel implementation on the AWS p2.16xlarge instance. The same *A-scan* takes 20 min in the serial implementation.

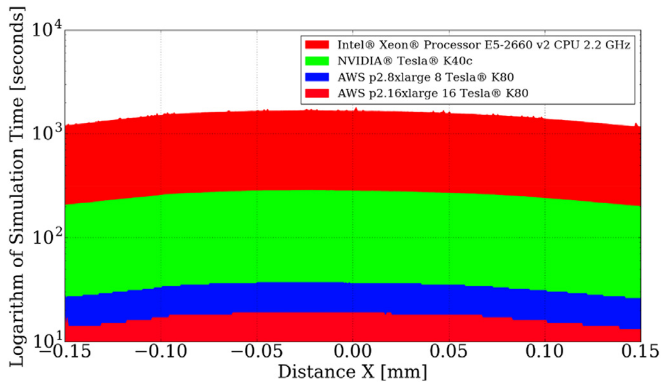

Fig. 14. Comparison of computation time per A-Scan of the first simulation obtained by different platforms. The computation times per A-Scans of the object in *Simulation Case 1* at different lateral positions obtained using serial simulator, OCT-MPS on NVIDIA Tesla K40c GPU card, OCT-MPS on Amazon AWS p2.8xlarge instance, and OCT-MPS on Amazon AWS p2.16xlarge instance are represented by Light Red, Green, Blue and Dark Red curves, respectively. (For interpretation of the references to colour in this figure legend, the reader is referred to the web version of this article.)

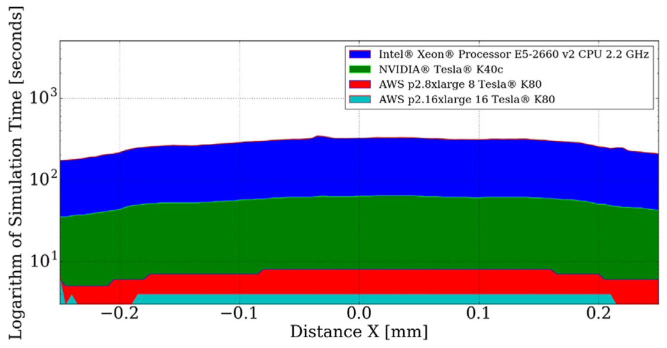

Fig. 15. Comparison of computation time per A-Scan of the second simulation obtained by different platforms. The computation times per A-Scans of the object in *Simulation Case 2* at different lateral positions obtained using serial simulator, OCT-MPS on NVIDIA Tesla K40c GPU card, OCT-MPS on Amazon AWS p2.8xlarge instance, and OCT-MPS on Amazon AWS p2.16xlarge instance is represented by Dark Blue, Green, Red, and Light Blue curves, respectively. (For interpretation of the references to colour in this figure legend, the reader is referred to the web version of this article.)

The longest computation time per *A-scan* performed by OCT-MPS on the AWS p2.16xlarge instance takes 19 s while in the serial implementation simulation of the same A-Scan takes about 27 min. The average computation time per *A-scan* is 17 s on OCT-MPS on the AWS p2.16xlarge instance.

In *Simulation Case 2*, the simulation an A-scan from the region with lowest scattering coefficient takes about 3 s in the parallel implementation on the AWS p2.16xlarge instance. The same A-scan takes about 3.6 min on serial implementation.

The computation time gain depends on the complexity of the media to simulate. The A-scan computation times of both examples show that average speed up factor of the A-Scan simulations time obtained by parallel code to the serial implementation increases in the regions with lower scattering coefficients. However, comparing the computation gain of the first and second example shows that when the A-scan simulation time of the medium is very low, due to, e.g., the shallow depth of the slab, the computation time gain decreases (from 86 to 67). This is because at such low simulation time, the time spent on the data transfer between the CPU memory and the GPU memory accounts for a larger percentage of the processing time in the MPS implementation of the code.

5. Conclusions

We developed a massively parallel simulator of OCT of inhomogeneous turbid media that we named OCT-MPS. This simulator was implemented on GPUs using NVIDIA's CUDA architecture and extensions of the C language. We exploited the intrinsic parallelism of the latest implementation of MC methods to trace thousands of photon packets simultaneously. We also implemented an importance sampling technique that we developed to reduce the simulation time further. We used a tetrahedron-based mesh to simulate arbitrary shaped objects, and we implemented a computationally efficient parallel pseudo-random number generator. We showed that an acceleration of up to two orders of magnitude could be achieved compared to the available serial OCT simulator for arbitrary shaped objects. The accuracy of our simulation results was verified against previously validated serial simulations.

OCT-MPS assumes an infinitely thin optical beam. As future work, we plan to implement an OCT simulator with a beam of finite width, in addition to the inclusion of light polarization, to simulate physical OCT systems more accurately. The current implementation of the OCT-MPS simulates time domain OCT (TD-OCT), but it could be extended to swept source OCT (SS-OCT) simulation. SS-OCT simulators, however, would require optical coefficients of the object at different wavelengths.

Acknowledgments

We would like to thank Dr. Ian Jeffrey, Electrical and Computer Engineering Department, University of Manitoba, for allowing us access to a CUDA capable GPU card in his research lab.

References

[1] W. Drexler, J.G. Fujimoto, Optical Coherence Tomography, Springer, 2008.
[2] W. Drexler, U. Morgner, R.K. Ghanta, F.X. Kärtner, J.S. Schuman, J.G. Fujimoto, Ultrahigh-resolution ophthalmic optical coherence tomography, Nat. Med. 7 (4) (2001) 502–507.
[3] M.C. Pierce, J. Strasswimmer, B.H. Park, B. Cense, J.F. de Boer, Advances in optical coherence tomography imaging for dermatology, J. Invest. Dermatol. 123 (3) (2004) 458–463.
[4] P.F. Escobar, J.L. Belinson, A. White, N.M. Shakhova, F.I. Feldchtein, M.V. Kareta, N.D. Gladkovas, Diagnostic efficacy of optical coherence tomography in the management of preinvasive and invasive cancer of uterine cervix and vulva, Int. J. Gynecol. Cancer 14 (3) (2004) 470–474.
[5] D.P. Popescu, C. Flueraru, Y. Mao, S. Chang, J. Disano, S.S. Sherif, M.G. Sowa, Optical coherence tomography: fundamental principles, instrumental designs and biomedical applications, Biophys. Rev. 3 (3) (2011) 155–169.
[6] A.M. Pagnozzi, R.W. Kirk, B.F. Kennedy, D.D. Sampson, R.A. McLaughlin, Automated quantification of lung structures from optical coherence tomography, Biomed. Opt. Exp. 4 (11) (2013) 2683–2695.
[7] B. Shi, Z. Meng, L. Wang, T. Liu, Monte Carlo modeling of human tooth optical coherence tomography imaging, J. Opt. 15 (7) (2013) 075304.
[8] F.A. South, E.J. Chaney, M. Marjanovic, S.G. Adie, S.A. Boppart, Differentiation of *ex vivo* human breast tissue using polarization-sensitive optical coherence tomography, Biomed. Opt. (2014) OSA BT3A-71.
[9] S. Malektaji, I.T. Lima Jr., S.S. Sherif, Monte Carlo simulation of optical coherence tomography for turbid media with arbitrary spatial distributions, J. Biomed. Opt. 19 (4) (2014) 046001.
[10] B. Wilson, G. Adam, A Monte Carlo model for the absorption and flux distributions of light in tissue, Med. Phys. 10 (6) (1983) 824–830.
[11] L. Wang, S.L. Jacques, L. Zheng, MCML – Monte Carlo modeling of light transport in multi-layered tissues, Comput. Meth. Programs Biomed. 47 (2) (1995) 131–146.
[12] C. Zhu, Q. Liu, Review of Monte Carlo modeling of light transport in tissues, J. Biomed. Opt. 18 (5) (2013) 050902.
[13] D.J. Smithies, T. Lindmo, Z. Chen, J.S. Nelson, T.E. Milner, Signal attenuation and localization in optical coherence tomography studied by Monte Carlo simulation, Phys. Med. Biol. 43 (10) (1998) 3025–3044.
[14] G. Yao, L. Wang, Monte Carlo simulation of an optical coherence tomography signal in homogeneous turbid media, Phys. Med. Biol. 44 (9) (1999) 2307.
[15] I.T. Lima, A. Kalra, S.S. Sherif, Improved importance sampling for Monte Carlo simulation of time-domain optical coherence tomography, Biomed. Opt. Express 2 (5) (2011) 1069–1081.
[16] I.T. Lima, A. Kalra, H.E. Hernandez-Figueroa, S.S. Sherif, Fast calculation of multipath diffusive reflectance in optical coherence tomography, Biomed. Opt. Express 3 (4) (2012) 692–700.
[17] M.Y. Kirillin, E. Alarousu, T. Fabritius, R.A. Myllylä, A.V. Priezzhev, Visualization of paper structure by optical coherence tomography: Monte Carlo simulations and experimental study, J. Opt. Soc. Rapid 2 (2007) 07031.
[18] I.N. Dolganova, A.S. Neganova, K.G. Kudrin, K.I. Zaytsev, I.V. Reshetov, Monte Carlo simulation of optical coherence tomography signal of the skin nevus, J. Phys. Conf. Ser. 673 (1) (2016) 012014 IOP Publishing.
[19] I.L. Shlivko, M.Y. Kirillin, E.V. Donchenko, D.O. Ellinsky, O.E. Garanina, M.S. Neznakhina, P.D. Agrba, V.A. Kamensky, Identification of layers in optical coherence tomography of skin: comparative analysis of experimental and Monte Carlo simulated images, *Skin Res. Technol.* 21 (4) (2015) 419–425.
[20] M.Y. Kirillin, G. Farhat, E.A. Sergeeva, M.C. Kolios, A. Vitkin, Speckle statistics in OCT images: Monte Carlo simulations and experimental studies, Opt. Lett. 39 (12) (2014) 3472–3475.
[21] V. Periyasamy, M. Pramanik, Monte Carlo simulation of light transport in turbid medium with embedded object—spherical, cylindrical, ellipsoidal, or cuboidal objects embedded within multilayered tissues, J. Biomed. Opt. 19 (4) (2014) 045003-045003.
[22] V. Periyasamy, M. Pramanik, Importance sampling-based Monte Carlo simulation of time-domain optical coherence tomography with embedded objects, Appl. Opt. 55 (11) (2016) 2921–2929.
[23] S. Zhao, Advanced Monte Carlo Simulation and Machine Learning for Frequency Domain Optical Coherence Tomography, California Institute of Technology, 2016.
[24] J. Schöberl, NETGEN and advancing front 2D/3D-mesh generator based on abstract rules, Comput. Visual. Sci. 1 (1) (1997) 41–52.
[25] T.R. Halfhill, Parallel processing with CUDA, Microprocess. Rep. 1 (28) (2008) 08-01.
[26] M.J. Yadlowsky, J.M. Schmitt, R.F. Bonner, Multiple scattering in optical coherence microscopy, Appl. Opt. 34 (25) (1995) 5699–5707.
[27] M.Y. Kirillin, A.V. Priezzhev, R.A. Myllylä, Role of multiple scattering in formation of OCT skin images, Quantum Electron. 38 (6) (2008) 570–575.
[28] NVIDIA Corporation, "CUDA Programming Guide 6.5," (2014).
[29] J. Cheng, M. Grossman, T. McKercher, Professional CUDA C Programming, John Wiley & Sons, 2014.
[30] D. Kirk, W. Hwu, Programming Massively Parallel Processors, Morgan Kaufmann, 2010.
[31] J. Sanders, E. Kandrot, CUDA by Example, Addison-Wesley, 2011.
[32] S. Cook, CUDA Programming, Morgan Kaufmann, 2013.
[33] N. Wilt, The CUDA Handbook: A comprehensive Guide to GPU Programming, Pearson Education, 2013.
[34] M.J. Quinn, Parallel Computing: Theory and Practice, McGraw-Hill, 1994.
[35] E. Alerstam, W.C.Y. Lo, T.D. Han, J. Rose, S. Andersson-Engels, L. Lilge, Next-generation acceleration and code optimization for light transport in turbid media using GPUs, Biomed. Opt. Express 1 (2) (2010) 658–675.
[36] T. Misfeldt, G. Bumgardner, A. Gray, The Elements of C++ Style, Cambridge University, 2004.
[37] G. Marsaglia, Random number generators, J. Mod. Appl. Stat. Meth. 2 (1) (2003) 2–13.