GPU-accelerated Monte Carlo simulation for photodynamic therapy treatment planning

William Chun Yip Lo[a], Tianyi David Han[b], Jonathan Rose[b], and Lothar Lilge[a]

[a] University of Toronto, Department of Medical Biophysics, 610 University Avenue, Toronto, ON, Canada M5G 2M9;

[b] University of Toronto, The Edward S. Rogers Sr. Department of Electrical and Computer Engineering, 10 King's College Road, Toronto, ON, Canada M5S 3G4

ABSTRACT

Recent improvements in the computing power and programmability of graphics processing units (GPUs) have enabled the possibility of using GPUs for the acceleration of scientific applications, including time-consuming simulations in biomedical optics. This paper describes the acceleration of a standard code for the Monte Carlo (MC) simulation of photons on GPUs. A faster means for performing MC simulations would enable the use of MC-based models for light dose computation in iterative optimization problems such as PDT treatment planning. We describe the computation and how it is mapped onto the many parallel computational units now available on the NVIDIA GTX 200 series GPUs. For a 5 layer skin model simulation, a speedup of 277x was achieved on a single GTX280 GPU over the code executed on an Intel Xeon 5160 processor using 1 CPU core. This approach can be scaled by employing multiple GPUs in a single computer - a 1052x speedup was obtained using 4 GPUs for the same simulation.

Keywords: graphics processing units, Monte Carlo simulation, treatment planning, photodynamic therapy, MCML

1. INTRODUCTION

Photodynamic therapy (PDT) is an emerging treatment modality in oncology and other fields.[1] Improvements in PDT efficacy, especially for interstitial applications, require faster computational tools to enable efficient treatment planning, particularly when tissue heterogeneity impedes the use of diffusion theory. To maximize PDT efficacy, accurate models of light propagation that accounts for target volume geometry and its heterogeneity should be employed. For computing light dose (fluence) distribution, the Monte Carlo (MC) method is advantageous due to its flexibility and accuracy. Unfortunately, MC-based models are very time-consuming. The acceleration of MC-based light dose computation would enable its use in iterative optimization problems such as PDT treatment planning.

This paper presents the acceleration of a widely accepted MC code, called Monte Carlo for Multi-Layered media (MCML),[3] on NVIDIA graphics processing units (GPUs) which are designed for highly parallel computing. The highly parallelizable nature of MC simulations in general makes this class of simulations a good candidate for parallelization on GPUs. GPU-accelerated scientific computing is becoming increasingly popular with the release of an easier-to-use programming model and environment from NVIDIA (called CUDA, short for compute unified device architecture).[2] CUDA provides a C-like programming interface for NVIDIA GPUs and it suits general-purpose applications much better than traditional GPU programming languages such as OpenGL. However, performance optimization of a CUDA program requires careful consideration of the GPU architecture to exploit the full potential of a GPU. In this paper, we report on the exciting performance achieved, and more importantly, share the experience of experimenting with different parallelization and optimization schemes, which reveals the unique challenges in CUDA programming and the subtlety of the NVIDIA GPU architecture.

Further author information: (Send correspondence to L.L.)
W.C.Y.L.: E-mail: wlo@uhnres.utoronto.ca, Telephone: 416-946-4501 x5767
T.D.H.: E-mail: han@eecg.toronto.edu, Telephone: 416-978-5274
J.R.: E-mail: jayar@eecg.toronto.edu, Telephone: 416 978-6992
L.L.: E-mail: llilge@uhnres.utoronto.ca, Telephone: 416-946-4501 x5743

Therapeutic Laser Applications and Laser-Tissue Interactions IV, edited by Ronald Sroka, Lothar D. Lilge,
Proc. of SPIE-OSA Biomedical Optics, SPIE Vol. 7373, 737313 · © 2009 SPIE-OSA
CCC code: 1605-7422/09/$18 · doi: 10.1117/12.831944

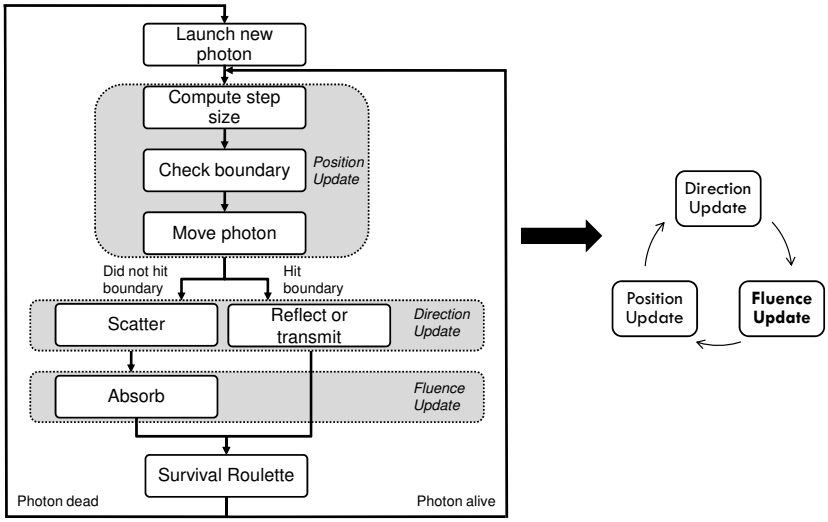

Figure 1. Left: Flow-chart of MCML simulation. Right: Simplified version to represent MCML in subsequent sections.

2. BACKGROUND

2.1 MCML

MCML provides an MC model of steady-state light transport in multi-layered media. With modifications, it can form the basis for light dose computation in PDT treatment planning. MCML assumes infinitely wide layers and models the propagation of photon packets from an infinitely narrow light beam perpendicular to the surface. In MCML, absorption is recorded in a 2-D absorption array called A[r][z], representing the photon absorption probability density [cm^{-3}] as a function of radius r and depth z. This can be converted into photon fluence [measured in cm^{-2} for the impulse response from the source beam]. Millions of photon packets are required to generate a high quality, low-noise fluence distribution useful for treatment planning. Note that extended sources can be modelled by convolving the photon distribution for the impulse response. In MCML, the simulation of each photon packet involves a similar sequence of computational steps and is independent of other photon packets. Therefore, multiple photon packets can be processed in parallel if multiple processors are available.

Figure 1 shows a flow chart of the key steps in an MCML simulation: position update, direction update, and fluence update. First, a new photon packet is launched from the surface perpendicularly into the multi-layered geometry and is assigned an initial weight. The position update step moves the photon packet to its next interaction site by a step size obtained from sampling a probability distribution based on the photon packet's mean free path between interaction events. The new position is generated after checking for interaction with nearby boundaries (which are interfaces between layers). If the photon hits a boundary, it is either reflected or transmitted. This is determined by computing the internal reflectance using Fresnel's formulas and generating a random number between 0 and 1. If the internal reflectance is greater than the random number, the photon packet is internally reflected. Otherwise, the photon packet is transmitted into the next layer and its new direction is calculated using Snell's law. If the photon does not hit a boundary, part of the photon's weight is absorbed at the current position based on the absorption coefficient and the new direction is calculated using the Henyey-Greenstein function[4] to model scattering. Finally, a survival roulette uses a random number to determine if the tracking of a photon should be terminated when its weight reaches a predefined threshold. If the photon survives the roulette, the photon's weight is increased to maintain energy conservation and the key steps are repeated. If the photon is terminated, a new photon is launched. Further details on MCML can be found in the paper by Wang et al.[3]

2.2 Related Work

One common approach to reduce the computation time of MC-based photon migration simulations involves distributing the processing of photon packets across multiple processors. Since each group of photon packets

can be simulated independently on each processor and the overhead of summing the partial results generated per processor is small in large simulations, this approach reduces the simulation time almost linearly. That is, given N processors, the multi-processor implementation achieves approximately N-fold *speedup* (single-processor execution time/multi-processor execution time).[6] Despite the relative simplicity of this traditional software-based parallelization scheme, the cost of acquiring and maintaining a dedicated, high-performance computer cluster for iterative optimization problems, such as PDT treatment planning, can be substantial.

An alternative approach involves the use of a hardware-based acceleration scheme. Previously, we implemented a custom digital hardware design on field programmable gate arrays (FPGAs), based on MCML, to simulate light absorption in multi-layered tissue.[7] The custom hardware outperformed the MCML software running on a 3-GHz Intel Xeon processor by approximately 80 times on a multi-FPGA platform called TM-4. A skin model (same as the one in this work) was used as the simulation problem. Instead of creating custom hardware *de novo*, this work explores the use of commodity graphics processing hardware or GPUs from NVIDIA.

In terms of previous attempts to use GPUs for MC simulations, Alerstam et al. reported ∼1000x speedup on the NVIDIA GeForce 8800GT graphics card, compared to an Intel Pentium 4 processor, for the simulation of time-resolved photon migration in a simple, semi-infinite geometry.[8] The same group has also recently released a CUDA-based implementation of MCML called CUDAMCML (beta version), reporting an order of magnitude lower speedup when absorption is recorded in a multi-layered geometry. The performance worsens by another order of magnitude (to a speedup of ∼10x) with fewer voxels (absorption array elements).

This work proposes a different approach to handle the inefficiency in the scoring of absorption and addresses the question of how various optimizations can dramatically affect the performance of MC-based simulations for photon migration, based on our own experimentation with accelerating MCML on GPUs. Since the MC method is widely applied in computational physics and most MC simulations share a set of common features, the development process is described to assist other investigators. The final, optimized implementation was also tested on a multi-GPU system to show the possibility of using a cluster of GPUs for PDT treatment planning.

2.3 CUDA-based GPU Programming

This section reviews the fundamentals of CUDA programming on NVIDIA GPUs. Only the technical terms necessary for understanding the subsequent sections are introduced. For a full description on NVIDIA GPU and CUDA, readers may consult the CUDA programming guide.[2]

2.3.1 Graphics processing units

The proliferating gaming industry has driven the rapid evolution of graphics processing units (GPUs). Since a GPU is specialized for the parallel processing of high frame rate, high definition graphics, it is well-suited for computationally intensive, highly parallel applications. The widening gap between CPU and GPU in terms of peak floating point computational power and memory bandwidth makes the GPU an attractive platform for scientific computing.[2]

The underlying hardware architecture of a NVIDIA GPU is illustrated in Fig. 2.[2] The remainder of this section gives a brief overview of the architecture using the NVIDIA GeForce GTX280 GPU as an example. First, the GTX280 GPU contains 30 *multiprocessors*, each of which contains 8 scalar processors (SPs). Each multiprocessor uses a mode of instruction execution called *single-instruction, multiple-thread* (SIMT). (A *thread* is a unit of work or execution; for example, a thread can process one pixel in an image-processing application.) For the purpose of this discussion, SIMT means that all SPs within a multiprocessor always execute the same instruction, but on potentially different data. The main implication of the SIMT architecture is that a GPU with 240 SPs cannot be viewed as 240 independent processors (rather, it should be considered as 30 independent processors that can compute 8 similar computations at a time).

Second, the memory hierarchy of the GPU must be understood by the programmer, as there is a significant difference in the time required to access different types of memory. The GPU contains a large off-chip memory called *global memory* or *device memory*. It is large (typically ∼1 GB) but relatively slow (∼600 clock cycles of access latency). A GPU also contains fast on-chip memory, including *registers* with typically single clock cycle of access latency, *shared memory* at close to register speed, and a low-latency cache for *constant memory* and

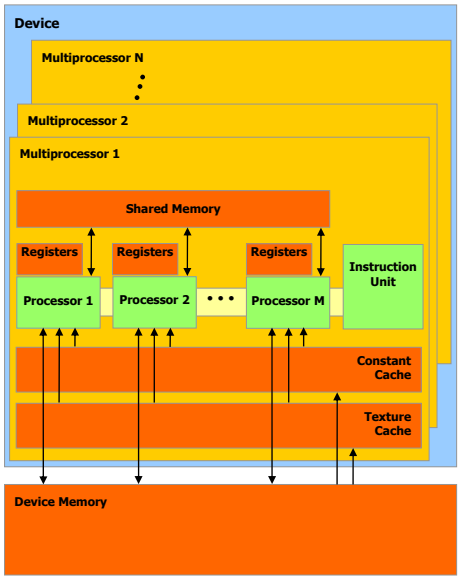

Figure 2. Hardware Architecture of a NVIDIA GPU (N=30 and M=8 for GTX 280)[2]

texture memory (which reside on the device memory), as shown in Fig. 2. There are 16,384 32-bit registers per multiprocessor (for newer graphics cards with compute capability 1.2[2]) that are distributed across the SPs to store private temporary variables, a 16 kB shared memory per multiprocessor useful for communication between the SPs, and an 8 kB constant cache per multiprocessor that store exclusively read-only data. Finally, there is a region in device memory called *local memory* reserved for large data structures, including arrays, which cannot be placed inside registers. Note that local memory is somewhat a misnomer since it is as slow as global memory.

2.3.2 Programming with CUDA

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

In CUDA, the programmer writes GPU code in the form of *kernels*, which are similar to regular C functions. Multiple copies are executed in parallel by the GPU *threads*. Therefore, the programmer must first divide the application into parallel units of execution, which are assigned to threads. These threads are in turn organized into *thread blocks*. Next, the programmer specifies a kernel configuration - the number of thread blocks and number of threads/block. This is an important decision as each thread block is executed on a single multiprocessor and the threads inside are mapped onto the individual SPs.

Note that the GPU is usually referred to as the *device* while the CPU side is considered the *host*. Communication between the host and the device occurs via the global memory. CUDA requires the programmer to explicitly manage the storage of data on the GPU and the data transfer between the device memory (GPU) and the host memory (CPU). The programmer must be aware of the size and access restrictions of each type of memory in order to properly write and launch a GPU kernel. The scope of a variable may differ depending on the specific type of memory used. For example, constant memory and global memory are accessible to all threads, while shared memory is only shared by threads within a block. By default, variables declared without specifying the type of memory are stored in registers, which are private to each thread and cannot be accessed by other threads.

In many applications, the threads cannot execute independently and must communicate or synchronize through some shared variables. This often requires the use of *atomic instructions* to coordinate access to a

shared variable (stored in global or shared memory). Atomic instructions guarantee data consistency by allowing only one thread to update (read and then modify in one instruction that cannot be divided) the shared variable at any time, which stalls other threads in the process. Sometimes, expensive atomic instructions can be avoided by replicating the shared data structure in each thread. However, if the shared data structure is large, the size of the memory will limit the number of threads that can be launched.

2.3.3 CUDA-based acceleration techniques

For the highest performance, programmers must write CUDA code with the GPU architecture in mind. Two important objectives include (1) maximizing parallelism and instruction throughput (2) efficient memory usage.

To maximize parallelism, the CPU code should be divided into data-parallel portions that require minimum synchronization. *Branch divergence*, i.e. threads within a *warp* (a group of 32 consecutive threads mapped to a multiprocessor) taking different execution paths, should be avoided whenever possible. To further maximize instruction throughput, the faster intrinsic math functions designed for the GPU can be used. Double-precision floating point operations should be avoided as they are much more expensive than single-precision operations.

Efficient memory usage aims to reduce the amount of time required for fetching data from the device memory (off-chip and slow). There are mainly three ways to achieve this goal:

1. Reduce the number of accesses to the device memory by caching frequently accessed data in fast, on-chip memories, such as registers and shared memory.
2. *Coalesce* or group accesses to the global memory so that the device memory bandwidth is used more efficiently. Here, the requirement of coalescing is that the threads in a half-warp (16 threads) must access a contiguous memory segment.
3. Increase the level of parallelism (i.e. the number of active threads) to hide memory latency.

Unfortunately, optimizations in these three categories often compete with one another for hardware resources. For example, caching data in registers increases register usage per thread and can limit the number of threads that can be launched. One of the key challenges in optimizing a CUDA program is to find a scheme of assigning resources that achieves the best performance.

3. GPU-ACCELERATED MCML

In this section, the implementation details of the GPU-accelerated MCML are presented, showing how our approach leverages both the parallelism and avoids memory bottlenecks. The key development stages are also described to summarize the thought process and challenges encountered before arriving at the final solution.

3.1 Implementation Details

Fig. 3 shows an overview of the parallelization scheme used to accelerate MCML on one of the four NVIDIA GPUs in our system. Compared to the serial execution of MCML on a single-core CPU where only one photon packet is simulated at a time, the GPU-accelerated version of MCML can simulate many photon packets in parallel. To scale up to four GPUs, the same configuration is replicated four times, except that each instance must initialize a different seed and declare a separate absorption array per GPU. In the current implementation, 30 thread blocks, each containing 256 threads, are created within each GPU. Each thread block is mapped onto one of the 30 available multiprocessors (per GPU) and the 256 threads interleave its execution on the 8 scalar processors within each multiprocessor (see Fig. 2).

Each thread executes the main iteration of MCML independently of each other, except for one step in fluence update, which is a key performance bottleneck. This step is the accumulation of absorption in the common A[r][z] array in global memory, which is performed using the `atomicAdd()` instruction as multiple threads may simultaneously access the same array element. (Note that although A[r][z] could be replicated per thread to completely avoid atomic instructions, the maximum number of threads would be limited by the size of the device memory, as discussed later.) Since this atomic instruction only supports integer operations, the elements of

A[r][z] need to be integerized using a scaling factor (empirically determined to be 12 million) to maintain the precision of the computation. To avoid handling computational overflow (reaching beyond the maximum value due to the use of fixed point, integer representation) in global memory, each element in the array is declared as a 64-bit integer (instead of 32-bit integer). However, using `atomicAdd()` to access global memory is particularly slow, both because global memory access is a few orders of magnitude slower than shared memory and because atomicity prevents parallel execution of this portion of the code. This worsens with increasing number of threads due to the higher probability of simultaneous access to an element (also known as contention).

To reduce contention and access latency to A[r][z], two memory optimizations, caching in registers and shared memory, are used. The first optimization is based on the observation that consecutive absorption events can happen at nearby (or even the same) locations in A[r][z]. In each thread, consecutive writes to the same location of the A[r][z] are accumulated in a register until a different location is computed. The second optimization is based on the high access rate of A[r][z] elements near the photon source (or the origin in MCML). This is a main source of contention. Therefore, for each thread block, this portion of A[r][z] is buffered in the shared memory, which is updated atomically by threads within the block. These per-block buffers are flushed to A[r][z] in the global memory as necessary. To buffer as many elements near the photon source as possible, the size of each element was reduced to 32-bit. This reduction causes a far greater risk of overflow (which occurs when the accumulated value exceeds $\sim 2^{32}$). To prevent overflow, the old value is always checked before adding. If overflow is imminent, the value is flushed to global memory, which uses 64-bit representation. Note that 64-bit `atomicAdd()` to shared memory is also currently not supported. To further avoid atomic accesses, photons beyond the detection grid do not accumulate their weights at the perimeter of the grid, unlike in MCML. Note that elements at the perimeter give invalid values in the original MCML.[3] This optimization does not change the correctness of the simulation, and ensures that performance is not degraded if the number of voxels in the detection grid is decreased, which forces photons to be absorbed at the boundary elements (significantly increasing contention and access latency to these elements in A[r][z]).

Another major problem with the original MCML code for GPU-based implementation is its abundance of branches (e.g., `if` statements). Also, some branches depend on a random number to decide the execution path. Two optimizations were implemented to tackle this problem. The first one was proposed by Alerstam et al.,[8] which removes the divergence caused by significant variation in the number of simulation steps for each photon before termination. In their scheme, each thread simulates a fixed number of steps and a new photon packet is launched immediately after the previous one dies. The second optimization for divergence reduction targets the `Reflect` function inside direction update, which involves a conditional branch with nearly identical paths except using different variables. This branch is replaced with a much smaller one that decides which variables to use, followed by the common piece of computation. This optimization almost halves the number of instructions executed (in this step) by each thread and reduces divergence.

This implementation also includes a number of other optimizations, such as using GPU-specific math functions (__sincosf, __logf), reducing local memory usage by expanding arrays into individual elements, and storing read-only tissue layer specifications in constant memory.

3.2 Development Process

3.2.1 Single-precision floating point conversion (1 person-week)

The original MCML uses double-precision floating point data representation. Although double-precision operations are supported on our newer NVIDIA graphics cards, they are not as efficient as single-precision floating point operations. Also, our first graphics card (NVIDIA GeForce 8800GTX) does not support double-precision arithmetic. Therefore, 1 person-week was spent on converting all double-precision operations into the corresponding single-precision operations and validating the modifications made to the C code.

3.2.2 Creation of GPU program skeleton (2 person-weeks)

A simple program skeleton was created containing a stub kernel to write at known locations in the absorption array to ensure that the data management code, kernel configuration, and the memory addressing scheme were implemented correctly. The different programming syntax for various types of memories made this step less trivial than originally expected. Also, kernel execution errors, such as segmentation faults, are not trivial to debug on

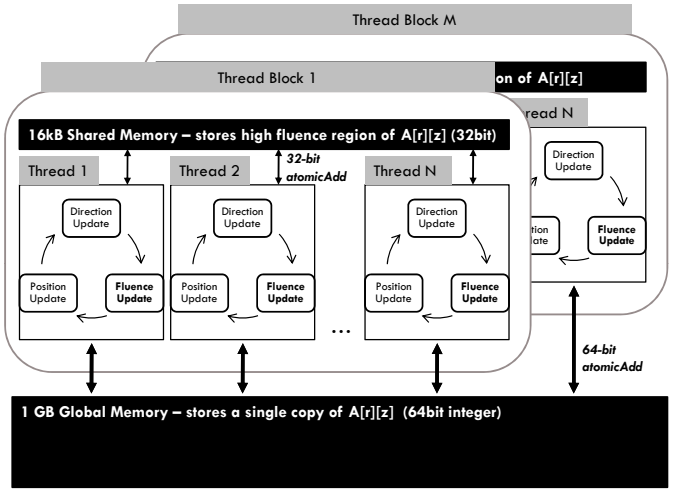

Figure 3. Parallelization Scheme of GPU-accelerated MCML (M=30 and N=256 for each GPU)

GPUs. This intermediate step was useful for understanding unexpected program behaviour and ensuring the correct syntax was used, thus significantly saving debugging time later.

3.2.3 Development of the first MCML kernel (1 person-month)

Most of this time was spent on converting the MCML functions into CUDA functions. This initial version used a very basic parallelization scheme, in which a private copy of A[r][z] was created per thread in global memory. The unique ID assigned to each thread was used to construct a distinct seed for random number generation per thread. These modifications ensured that each thread could execute entirely in parallel. The final result was obtained by summing the partial results stored in the private A[r][z] copies by launching another GPU kernel.

3.2.4 Optimizations (3 person-months)

A significant portion of the time was spent on experimenting with various parallelization schemes and optimization approaches as the identification of performance bottlenecks was not straight-forward. One of the failed schemes, however, led to the better characterization of the performance bottlenecks. This failed scheme involved creating multiple kernels, each executing only one key step in the simulation (e.g., position update). As register contents disappear after a kernel ends, the photon data stored in registers was flushed to global memory between kernel calls. Although this implementation was slow due to excessive global memory access, it proved useful for pinpointing performance bottlenecks as the CUDA profiler[10] does not provide the break-down of execution time within a kernel. In terms of more fine-grain optimizations, these included reducing local memory usage, divergence, and instruction count. The details of these optimizations and their effect on performance are discussed next.

4. PERFORMANCE

For performance comparison, a five layer skin model with the tissue optical properties (absorption coefficient μ_a, scattering coefficient μ_s, anisotropy factor g, and refractive index n) from Table 1 was selected as the simulation input. The resolution of the scoring grid for A[r][z] was set to dr=0.01 cm (radially) and dz=0.002 cm (in the z direction), while the number of voxels was set to 256×256.

4.1 GPU and CPU Platforms

The execution time of the GPU-accelerated MCML (named here $GPU - MCML$) was first measured on a single-GPU system with one NVIDIA GTX280 graphics card (30 multiprocessors). The same code was then tested on a Quad-GPU system consisting of two NVIDIA GTX280 graphics cards and one NVIDIA GTX295

Table 1. Optical properties of the five-layer skin tissue at 633nm[11]

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

Table 2. Performance comparison between GPU-MCML and CPU-MCML using the skin model (10^8 photons)

Platform (Configuration)	Number of scalar processors	Simulation Time (s)	Speedup
Intel 3-GHz Xeon 5160 processor	–	6102	1
NVIDIA GTX295 (using only 1 GPU)	240	22.7	269
NVIDIA GTX295 (2 GPUs)	480	12.5	488
NVIDIA GTX295 + one GTX280 (3 GPUs)	720	7.7	792
NVIDIA GTX295 + two GTX280 (4 GPUs)	960	5.8	1052

graphics card (with 2 GPUs or 60 multiprocessors). The final $GPU-MCML$ was compiled using the latest CUDA Toolkit (version 2.2) and was tested in both Ubuntu 8.04 and Windows XP (Visual Studio 2005). The number of GPUs used can be specified at run-time and the simulation is split equally among these GPUs.

For baseline performance comparison, a 3-GHz Intel Xeon dual-core processor (Xeon 5160 processor) was selected due to its high performance. The original, CPU-based MCML (named here $CPU-MCML$) was compiled with the highest optimization level (gcc -O3 flag) and its execution time was measured on one of the two available cores on the Xeon processor. All execution times included the main simulation and all pre-/post-processing operations.

4.2 Speedup

Table 2 shows the execution time of GPU-MCML as the number of GPUs and the associated number of scalar processors was increased. The performance of the solution was roughly proportional to the number of GPUs used. Using all 4 GPUs or equivalently 960 scalar processors, the simulation time for 100 million photon packets in the skin model was reduced from approximately 1.7 h to 5.8 s. This time also included the overhead of summing the partial results from all 4 GPUs and the creation of the simulation output file, which took less than 1 second. With 4 GPUs, the overall speedup achieved was 1052x.

4.3 Effect of Optimizations

The initial, and in retrospect somewhat simplistic, MCML kernel described in section 3.2.3 yielded a very low performance. This section highlights the key optimizations applied and some insights gained through failures.

Although photon packets can be processed in parallel without synchronization using a private copy of A[r][z] per thread, this initial approach only provided ∼4x speedup on a graphics card with 128 scalar processors. 1024 threads (32 blocks x 32 threads/block) were launched. Inspection of compiler intermediate files revealed that each thread required 292 bytes of local memory, mostly for storing the arrays used by the random number generator. Since local memory is as slow as global memory, a series of optimizations were applied to reduce local memory usage, as shown in Table 3. First, a more efficient random number generator called three-component Tausworthe generator[12] (period length $\approx 2^{88}$) was adopted as it does not use arrays. This optimization led to a two-fold speedup. Second, another array storing nine random numbers was expanded into individual elements, allowing the compiler to allocate them in registers. This optimization resulted in another 1.3x speedup. The last optimization involved fine-tuning the kernel configuration and doubling the number of threads to 2048 (16 blocks x 128 threads/block), resulting in an additional 1.7x speedup. At this point, the total speedup was only 18x on the 8800 GTX card. The initial approach suffered from two problems. First, the size of the device memory limited the total number of threads that could be launched, leading to GPU resource under-utilization. Second, the memory access patterns across each half-warp (16 consecutive threads) could rarely be coalesced.

The migration to the new NVIDIA GTX280 graphics card was a key milestone since this card (unlike the old 8800 GTX) supports 64-bit atomic instructions. Before making any changes, a baseline measurement was made

Table 3. Effect of local memory usage on the execution time of GPU-MCML using the skin model for 10^8 photons.

Optimization	Local Memory Usage (byte)	Simulation Time[a] (s)	Speedup
1. Original, unoptimized version	292	1460	4
2. Use a Tausworthe random number generator[12]	68	759	8
3. Expand arrays into individual elements	28	578	11
4. Double the number of threads	28	338	18

[a] Measured on the NVIDIA 8800GTX graphics card

Table 4. Summary of optimizations (simulation time measured using the skin model for 10^8 photons).

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

[a] Measured on the NVIDIA GTX280 graphics card

[b] This is the version after optimization 4 from Table 3.

using the same code (private copies of A[r][z]) on the new card, as shown in Table 4. A rather disappointing speedup was obtained. Next, using the 64-bit integer `atomicAdd()` operation, a single copy of A[r][z] (64-bit integer per element) was created in global memory to be accessed by all threads (7680 threads in total). The device memory size no longer restricted the parallelism and more threads could be launched, leading to a 3-fold performance improvement. Unfortunately, the next optimization, involving the use of the GPU intrinsic math functions where possible (such as integer division, trigonometric, and logarithmic functions), showed only marginal improvement, contrary to our original hypothesis. This highlights one key issue with optimization, which is the identification of the major performance bottleneck. Using the multi-kernel approach described in section 3.2.4, the direction update and fluence update steps were shown to be the key performance bottlenecks by the CUDA profiler.

As discussed in section 3.1, one particularly crucial bottleneck is the use of atomic instructions to access global memory. To solve this problem, the first approach we attempted involved storing the most recent fluence update history in fast, on-chip memory after observing the locality of memory access for these updates. Each thread uses a register to buffer the most recent write to the same location in A[r][z]. This scheme increased the speedup to ∼54x. We then expanded this scheme to buffer writes to multiple locations of A[r][z] (in each thread) by using the shared memory, which further improves the performance to ∼90x. The final approach proposed in this paper, which involved buffering the high fluence regions of A[r][z] in shared memory, led to significantly better performance (∼133x) due to the great reduction of the number of expensive, atomic accesses to global memory. To maximize the number of high-fluence voxels that can stored in the small shared memory, the previous scheme of buffering recent writes to multiple locations of A[r][z] was not implemented simultaneously (only the last entry is saved in a register, with noticeable performance difference). Once the fluence update step was optimized, the reduction of divergence in the `Reflect` function (section 3.1), the reduction of instruction count (e.g., by removing common subexpressions and removing unnecessary random number generation in two mutually exclusive branches), and the optimization of the overflow handler (by flushing a group of elements once overflow is detected in a single element to avoid frequent flushing) led to an additional ∼2x performance.

4.4 Effect of Grid Geometry

To test the effect of grid resolution and the number of voxels on execution time, a thick, homogeneous slab (μ_a=0.1 cm^{-1}, μ_s=90 cm^{-1}, n=1.4, g=0.9, thickness=100 cm) was used. These optical properties are based on the test case used by Alerstam et al.[8] in their validation of CUDAMCML. As shown in Table 5, the grid resolution (dr and dz) only changed the simulation time of GPU-MCML slightly . The difference was within one second. The most significant difference in simulation time was observed when the number of voxels in the radial direction (nr) and in the z direction (nz) was increased from 500 × 200 (10^5 voxels) to 5000 × 2000 (10^7 voxels).

Table 5. Effect of grid geometry on simulation time for 10^7 photons in a thick homogeneous slab using GTX280

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

Interesting, this configuration increased the simulation time of both CPU-MCML and GPU-MCML. Further investigation revealed that ∼15 s was required to generate the simulation output file (The output file generation time was less than 1 second for the 500 × 200 voxel configuration). The size of the output file also increased from 1.25 MB to 125 MB. This overhead was very significant given the relatively short GPU kernel execution time; neglecting this overhead, the speedup would have been ∼320x. However, the performance still degraded slightly compared to another configuration (dr=dz=0.01, nr=500, and nz=200) with the same grid coverage (a cylindrical grid with a radius of 5 cm and depth of 2 cm). This is likely due to the decreased proportion of voxels that can be stored in the shared memory buffer, leading to more `atomicAdd()` operations for writing to global memory.

5. VALIDATION

5.1 Test Cases

Three test cases were used to validate GPU-MCML, including the skin model from Table 1 (dr=0.01 cm, dz=0.002 cm, nr=256, and nz=256), the homogeneous slab used in section 4.4 (dr=dz=0.01 cm, nr=500, and nz=200), and a ten layer geometry alternating between 2 materials, each layer with a thickness of 0.1 cm (material 1: μ_a=0.1 cm^{-1}, μ_s=90 cm^{-1}, g=0.9, and n=1.5; material 2: μ_a=0.2 cm^{-1}, μ_s=50 cm^{-1}, g=0.5, and n=1.2; dr=dz=0.01 cm, nr=500, and nz=200). The last two test cases were adopted from Alerstam et al.

5.2 Error Distribution

Since MC simulations are non-deterministic, the simulation results (A[r][z]) produced by GPU-MCML were validated against those from CPU-MCML with the statistical uncertainty in mind. First, the difference (percent error) between the A[r][z] arrays generated by GPU-MCML and CPU-MCML was plotted as a function of radius r and depth z. This was compared against the statistical difference between two runs of CPU-MCML with the same number of photon packets (reference map), but different random number seeds. If the GPU-MCML implementation added significant errors, there should be noticeable differences between the two maps. This was repeated for all test cases.

Figure 4 shows two very similar distributions, verifying that the difference observed between the A[r][z] arrays from GPU-MCML and CPU-MCML was within the statistical uncertainty between two runs of CPU-MCML for the case of the skin model. Figure 5 shows the distribution of error for the other two test cases, the homogenous slab and ten layer geometry. The patterns appear different, as expected, due to the different input parameters used. The reference maps identified these differences as the statistical uncertainty between runs, rather than the errors added by the GPU-MCML implementation. In particular, the conversion to single-precision floating point arithmetic, as shown by these plots, resulted in a negligible increase in error.

5.3 Light Dose Contours

Next, to show the accuracy of GPU-MCML within the context of PDT treatment planning, the skin model was used as the simulation model. A[r][z] was first converted to fluence for the impulse response and the isofluence contour lines generated by CPU-MCML and GPU-MCML were compared.

Figure 6 shows that the isofluence lines produced by GPU-MCML and CPU-MCML matched very well. A minor shift in the position of the isofluence lines was only noticeable for very low fluence levels, such as those at 0.00001 cm^{-2} (8 orders of magnitude smaller than the fluence near the centre - 1000 cm^{-2}). Notice that this

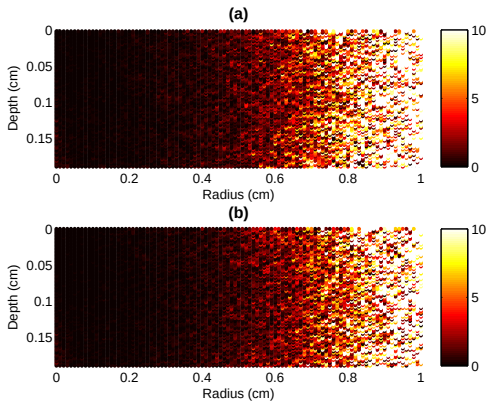

Figure 4. Distribution of relative error for the skin model using 100 million photon packets: (a) GPU-MCML vs. CPU-MCML, (b) CPU-MCML vs. CPU-MCML. All versions adopted the Tausworthe random number generator.[12] Colour bar represents percent error from 0% to 10%.

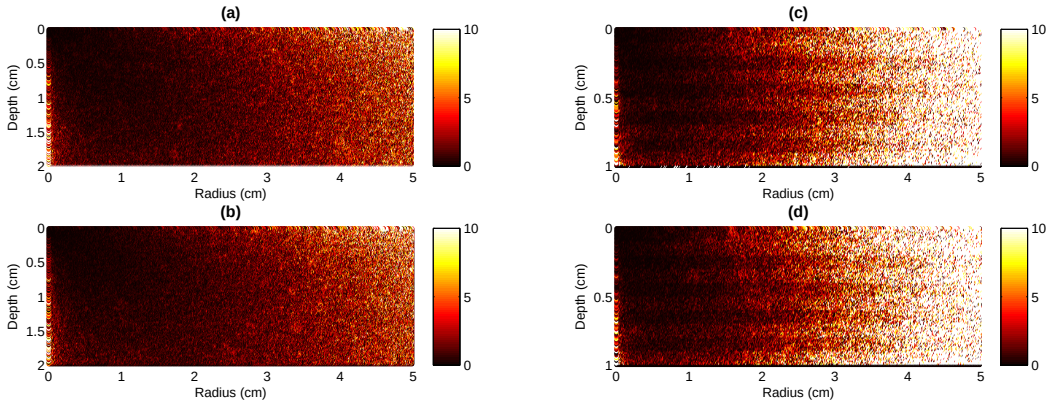

Figure 5. Distribution of relative error for a homogeneous slab [left panels (a) and (b)] and ten layer geometry [right panels (c) and (d)] using 10 million photon packets: (a, c) GPU-MCML vs. CPU-MCML, (b, d) CPU-MCML vs. CPU-MCML. All versions adopted the Tausworthe random number generator.[12] Colour bar represents percent error from 0% to 10%.

isofluence line, which is located at a radius of ~ 0.7 cm, corresponds to the transition region in Fig. 4 where the statistical uncertainty between runs starts to increase appreciably due to the low photon count. If this isofluence line is of importance (i.e., if it is near the threshold fluence level for activating the photosensitizers), more photon packets can be launched to achieve the desired accuracy for PDT treatment planning.

6. CONCLUSION

Using a skin model as simulation input, the GPU-accelerated MCML achieved a 277-fold performance on a NVIDIA GTX 280 graphics card with 30 multiprocessors and 1052-fold performance on a Quad-GPU system with 120 multiprocessors compared to a 3 GHz Intel Xeon processor. High accuracy was maintained as indicated by the close correspondence between the isofluence lines generated by GPU-MCML and CPU-MCML. The development process presented also illustrate the subtle nature of the underlying NVIDIA GPU architecture, necessitating a different approach to programming to achieve high performance.

For future work, the extension to a 3-D model along with support for multiple sources would be necessary to simulate more realistic tissue geometry and emission from multiple, implanted optical fibres. The implications include the greater demand on global memory due to the much larger absorption array required for the 3-D case. The increase in the number of absorption grid elements together with more photon sources will require a different

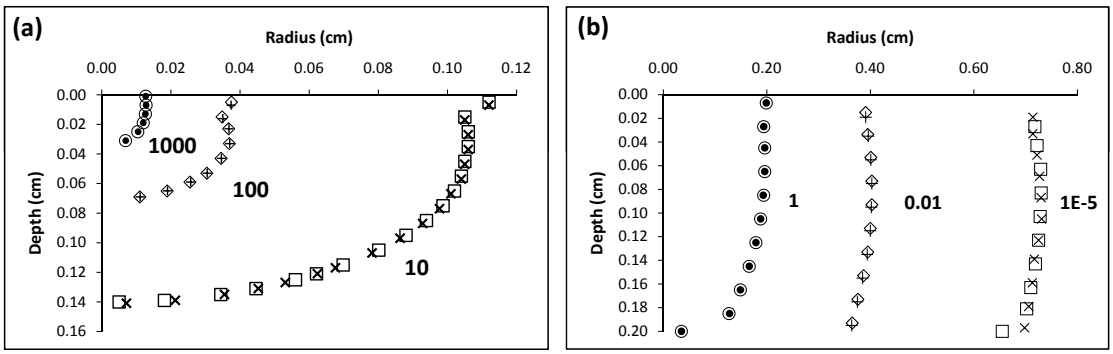

Figure 6. Isofluence lines for the impulse response generated by GPU-MCML and CPU-MCML using the skin model: ○, ◊, and □, results from CPU-MCML; and •, +, and ×, results from GPU-MCML. (a) Isofluence lines for fluence levels at 1000, 100, and 10 cm^{-2}, as indicated on the figure and (b) isofluence lines for fluence levels at 1, 0.01, and 0.00001 cm^{-2}.

approach to capture the high fluence region in shared memory to avoid frequent atomic instructions to global memory. Also, the constant memory will no longer be sufficient to store the tissue optical properties (at least not on a per-voxel basis). Finally, a number of key steps in MCML, including the checking of boundaries, need to be modified to propagate the photon in a 3-D voxel-based tissue geometry. Despite some potential challenges, GPU-based computing, possibly with thousands of scalar processors using a GPU cluster, still presents as an interesting option for enabling the use of MC-based models for PDT treatment planning in heterogeneous, spatially complex tissue geometry in the near future.

ACKNOWLEDGMENTS

The authors acknowledge the financial support from CIHR Grant No. 68951, NSERC Discovery Grant No. 171074, NSERC CGS-M and PGS-M scholarships. Erik Alerstam's insights and advice are greatly appreciated.

REFERENCES

[1] Dougherty, T., "Photodynamic Therapy," *Photochemistry and Photobiology* **58**(6), 895–900 (1993).
[2] Nvidia, C., "Compute Unified Device Architecture (CUDA) Programming Guide 2.1," *NVIDIA Corporation* (2009).
[3] Wang, L., Jacques, S., and Zheng, L., "MCML - Monte Carlo modeling of light transport in multi-layered tissues," *Computer Methods and Programs in Biomedicine* **47**(2), 131–146 (1995).
[4] Henyey, L. and Greenstein, J., "Diffuse radiation in the galaxy," in [*Annales d'Astrophysique*], **3** (1940).
[5] Colasanti, A., Guida, G., Kisslinger, A., Liuzzi, R., Quarto, M., Riccio, P., G., R., and Villani, F., "Multiple Processor Version of a Monte Carlo Code for Photon Transport in Turbid Media," *Computer Physics Communications* **132**, 84–93 (2000).
[6] Coyle, S., Thomas, K., Naughton, T., Charles, M., and Tom, W., "Distributed Monte Carlo Simulation of Light Transportation in Tissue," in [*Proceedings of 20th IEEE International Symposium on Parallel and Distributed Processing*], 4 (2006).
[7] Lo, W., Redmond, K., Luu, J., Chow, P., Rose, J., and Lilge, L., "Hardware acceleration of a Monte Carlo simulation for photodynamic treatment planning," *Journal of Biomedical Optics* **14**, 014019 (2009).
[8] Alerstam, E., Svensson, T., and Andersson-Engels, S., "Parallel computing with graphics processing units for high-speed Monte Carlo simulation of photon migration," *Journal of Biomedical Optics* **13**, 060504 (2008).
[9] NVIDIA, C., "Compute Unified Device Architecture–Reference Manual," (2008).
[10] NVIDIA, C., "CUDA Profiler 1.1," (2008).
[11] Tuchin, V., "Light scattering study of tissues," *Physics-Uspekhi* **40**(5), 495–515 (1997).
[12] L'Ecuyer, P., "Maximally Equidistributed Combined Tausworthe Generators," *Mathematics Of Computation* **65**(213), 203–213 (1996).