Accelerating Monte Carlo Simulations with an NVIDIA® Graphics Processor

Paul Martinsen,[1,*] Johannes Blaschke,[2] Rainer Künnemeyer,[3] Robert Jordan[1]

[1] The Plant and Food Research Institute of New Zealand, Hamilton, New Zealand

[2] Philipps-Universität Marburg, Hessen, Germany

[3] The University of Waikato, Hamilton, New Zealand

* Corresponding author: pmartinsen@hortresearch.co.nz, +64-7-959-4469

Abstract

Modern graphics cards, commonly used in desktop computers, have evolved beyond a simple interface between processor and display to incorporate sophisticated calculation engines that can be applied to general purpose computing. The Monte Carlo algorithm for modelling photon transport in turbid media has been implemented on an NVIDIA® 8800GT graphics card using the CUDA toolkit. The Monte Carlo method relies on following the trajectory of millions of photons through the sample, often taking hours or days to complete. The graphics-processor implementation, processing roughly 110 million scattering events per second, was found to run more than 70 times faster than a similar, single-threaded implementation on a 2.67 GHz desktop computer.

PACS Codes: 87.10.Rt, 33.80.Gj

Keywords: Monte Carlo photon transport, Scattering media, General purpose graphics card computing, parallel computing

Program Summary

Manuscript title:	Accelerating Monte Carlo Simulations with an NVIDIA Graphics Processor
Authors:	Paul Martinsen, Johannes Blaschke, Rainer Künnemeyer, Bob Jordan
Program title:	Phoogle-C/ Phoogle-G
Licensing provisions:	The random number library used has a LGPL license.
Programming language:	C++

Computer(s) for which the program has been designed:	Designed for Intel PC's. Phoogle-G requires a NVIDIA graphics card with support for CUDA 1.1
Operating system(s) for which the program has been designed:	Windows XP
RAM required to execute with typical data:	1GB
Has the code been vectorised or parallelized?:	Phoogle-G is written for SIMD architectures.
Number of processors used:	1
Supplementary material:	n/a
Keywords:	Monte Carlo photon transport, Scattering media, General purpose graphics card computing, parallel computing
PACS:	87.10.Rt, 33.80.Gj
CPC Library Classification:	21.1 *Radiation Physics*
External routines/libraries used:	Charles Karney Random number library Microsoft Foundation Class library NVIDA CUDA library [1]
CPC Program Library subprograms used:	n/a

Nature of problem

The Monte Carlo technique is an effective algorithm for exploring the propagation of light in turbid media. However, accurate results require tracing the path of many photons within the media. The independence of photons naturally lends the Monte Carlo technique to implementation on parallel architectures. Generally, parallel computing can be expensive, but recent advances in consumer grade graphics cards have opened the possibility of high-performance desktop parallel-computing.

Solution method

In this pair of programmes we have implemented the Monte Carlo algorithm described by Prahl et al. [2] for photon transport in infinite scattering media to compare the performance of two readily accessible architectures: a standard desktop PC and a consumer grade graphics card from NVIDIA.

Restrictions

The graphics card implementation uses single precision floating point numbers for all calculations. Only photon transport from an isotropic point-source is supported. The graphics-card version has no user interface. The simulation parameters must be set in the source code. The desktop version has a simple user interface; however some properties can only be accessed through an ActiveX client (such as Matlab).

Running time

Runtime can range from minutes to months depending on the number of photons simulated and the optical properties of the medium.

References

1. http://www.nvidia.com/object/cuda_home.html

2. S. Prahl, M. Keijzer, Sl Jacques, A. Welch, SPIE Institute Series 5 (1989) 102

Introduction

Monte Carlo simulation is commonly used for modelling photon transport in turbid media [1–, 2, 3, 4]. The gold standard for photon modelling, the Monte Carlo technique is often used for assessing the performance of models and analytic solutions to the radiation transport equation [5–, 6, 7], the accuracy of experimental results [8], estimating power density in laser treatment and in solving the inverse problem to estimate optical properties from experimental measurements [9–, 10, 11]. Its strength lies in simplicity. Individual photons are tracked as they propagate: scattered, absorbed, reflected and refracted by the medium using simple physical laws that permit ready modelling of sophisticated geometries. Its Achilles heel lies here too. Typically, millions of photons must be traced at each wavelength to obtain precise results requiring hours or days of computation time.

The Monte Carlo algorithm is well suited for parallel calculation, tracing photons simultaneously, and many implementations have been studied [12–, 13, 14, 15, 16, 17, 18]. Two approaches are commonly employed: parallel computing and distributed computing.

In a parallel computing environment, the processing hardware contains tens to thousands of processing units that often share resources such as memory. In a distributed computing environment, the program runs simultaneously on multiple computers communicating over a shared network. The latter often employs unused desktop machines that sit idle overnight. Until recently, parallel computing hardware has been characteristic of supercomputers and has been less readily accessible than networks of desktop machines, primarily due to capital cost. However, increasing demand for high-quality graphics from the entertainment industry has imbued desktop graphics co-processors with raw computation performance rivalling low-end supercomputers. For example, NVIDIA's® (California, USA) latest graphics processor (June, 2008) the GTX280 claims a performance of nearly 10^9 floating-point operations per second. While graphics processors are still several orders of magnitude below the top-500 supercomputers [19], their price (typically less than $US1000) offers very attractive performance per dollar. Graphics processors have been applied to speed up many algorithms from N-body simulations and microscope image registration to visualisation of white matter connectivity and solution of the time-independent Schrödinger equation with performance increases of up to 100 fold [20–, 21, 22 23, 24].

To benchmark the performance that might be realised for Monte Carlo simulation on graphics processor engines, we have implemented the Monte Carlo algorithm for photon transport from an isotropic point-source in an infinite, homogenous, turbid medium using i) a desktop processor and ii) an NVIDIA 8800GT graphics processor. Relative performance of the two implementations is compared. Though the application presented involves tracing rays in a scattering environment, we expect this approach could also be applied to ray tracing for geometric optics.

Method

Our implementation of the Monte Carlo algorithm is based on the work by Prahl et al. [25, 26]. Briefly, the algorithm keeps track of a photons position, heading and probability of surviving sequential scattering and absorption events, updating these as the photon propagates through the medium (Figure 1). The photon is launched, from the origin, with a survival probability of 1 that decreases at each scatter/absorption event until reaching a

predetermined threshold (0.001 here), whereupon tracking typically finishes and a new photon is launched[†]. Just before the photon is killed off, a 'roulette' step gives it a chance at a boost of "life". During roulette, which ensures energy conservation, there is a small chance (0.1 here) that that survival-probability is increased by a factor of 10 and tracing continues until the probability drops below the threshold again. Whenever the photon's position coincides with a linear array detector aligned with the z-axis, the power deposited (through absorption) into the medium is recorded in a running tally, E_T (Watts), at the appropriate position in the detector array. The power deposited during the i^{th} interaction is given by:

$$E_i = E_s \cdot p_i \cdot \frac{\mu_a}{\mu_s + \mu_a}.$$

Here, p_i is the probability of the photon surviving i interactions, μ_s is the scattering coefficient (1/m), μ_a is the absorption coefficient (1/m) and E_s is the source intensity (Watts). After tracing all photons, the fluence rate, ϕ (W/m^2), for an element of the detector array can be calculated:

$$\phi = \frac{E_T}{N \cdot \mu_a \cdot v},$$

where N is the number of photons simulated (258,048 here) and v is the volume of the detector element (8 mm^3, here). Next we outline the implementation of this algorithm before covering the details that distinguish the graphics processor implementation in more detail.

[†] This is a variance reduction technique that can also be described in terms of packets of photons (see Prahl *et al.* [25], for example), a fraction (the "weight") of which are absorbed as the light propagates. Numerically equal to weight, casting light propagation in terms of survival probability avoids the difficult concept of fractional photons.

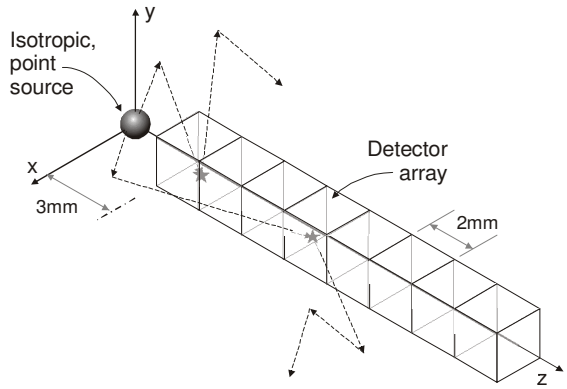

Figure 1. This schematic diagram illustrates the simulation geometry. The energy absorbed is tallied for absorption events occurring within the detector array (★'s).

Implementation Outline

We implemented and tested the Monte Carlo algorithm first in Matlab™ (The Mathworks, Massachusetts, USA) then translated it into two versions: one for the computer's core-processor and one for an NVIDIA 8800GT graphics processor installed on a PCI-Express × 16 bus. We have made the source code for both applications available in the Computer Physics Communications Programme Library [27].

The core-processor version was implemented in C++ using Visual Studio 2005 (Microsoft Corporation). The photon-tracing engine was implemented in a single thread separate to the user interface thread, and C++ language features with a significant overhead (such as virtual function calls) were avoided. The complier optimization options were set to favour speed and streaming SIMD extensions 2 were enabled. The simulation is initialised with the medium's optical properties and traces a single photon at a time until the requisite number of photons has been simulated. The detector array contains 100 detectors aligned along the z-axis, each a cube with 2 mm long sides with the centre of the first detector 3 mm from the origin.

The graphics processor version was implemented in C using the CUDA toolkit, version 1.1 (NVIDIA, California, USA). CUDA includes tools for building applications or libraries that execute on NVIDIA's graphics cards. CUDA provides support for common mathematical operations including hyperbolic, trigonometric and logarithmic functions; more complex operations such as matrix manipulation and Fourier transforms have also been implemented. After being initialised, batches of 21,504 photons are traced until the requisite number of photons has been simulated. Each batch is arranged in a grid of 84 ×

256-thread blocks. Sets of thread-blocks are assigned to multiprocessors for execution by the graphics card; the division of batches into blocks of threads facilitates this. Conceptually, the graphics processor traces 21,504 photons simultaneously. In practice, the graphics processor assigns blocks for execution as each multiprocessor becomes available. Scheduling more blocks than the graphics card can process simultaneously helps keep the graphics card busy and dilutes the overhead of interactions with the core-processor and operating system.

The detector array is in the same position in the graphics- and core-implementations (Figure 1). However, the graphics version uses only the first 60 elements—this is not a limitation of the graphics-card hardware, rather we found few photons were detected beyond this range. In our implementation, a separate detector-array tally (that is 21,504 tallies) is maintained for each photon traced simultaneously. This uses about 5 MB (about 80 KB per detector element) of the 512 MB available on the graphics card. So a 100 or even 200 element linear-detector would be quite practical with the current approach, though a 2-D or 3-D array detector would require a more sophisticated method to tally the deposited energy.

Both versions of the photon engine ran on a 2.67 GHz Intel Core-2 Duo E6750 machine with 2 GB of RAM running the Windows XP operating system. The core-processor version was executed on the main system processor. The graphics-processor version was executed on the system graphics card, which also served as the system display card, through the standard NVIDIA display adapter driver, version 6.14.11.6921. The 8800GT graphics card is clocked at 1.5 GHz, has 512 MB of onboard high-speed memory and 14 single-instruction, multiple-data (SIMD) multiprocessors. Each multiprocessor contains eight streaming processors each of which contains four processing units for a total of 32 SIMD data streams per multiprocessor [28]. Double precision (64-bit) floating-point arithmetic was used for most of the simulations on the core-processor implementation while single-precision (32-bit) arithmetic was used for the graphics processor implementation. Newer graphics processors (such as the NVIDIA GTX 200 series) support double precision arithmetic in hardware but were not available when this work was done. Double precision arithmetic can be implemented in software on the graphics processor at a significant cost in performance (anecdotally, an order of magnitude though this has not been tested). We also tested the performance of the core-processor implementation using single-precision

arithmetic. The two implementations of the photon-tracing engine are very similar, however, there are differences in four areas: energy recording, precision, random number generation and roulette. These are addressed in detail in the next section.

Graphics Processor Implementation Detail

In the core-processor implementation, a single detector array is sufficient to tally the absorbed energy as photons are traced. In the parallel implementation, this approach could lead to race-conditions: errors in the tally could occur if several absorption events for different photons occurred simultaneously within the same detector. To avoid this, a separate detector array buffer is maintained in global memory on the graphics card for each photon traced within a batch. With 21,504 photons traced in each batch and 60 detectors this requires about 5 MB of memory, about 80 KB per detector element. A more sophisticated approach could reduce memory requirements below 1 MB, however with plenty of memory available (512 MB on the 8800GT card) the simpler approach was preferred. The tally accumulates with each batch. After all photons have been traced the separate tallies are totalled to produce the final result in two steps. First, blocks of 256 tallies are totalled with the sum from each row stored in a leading diagonal fashion in the buffer. Finally, the total for each block is computed from the leading diagonals and stored in the detector output array. This approach was selected for two reasons. Firstly, storing the partial tallies in leading diagonals allows the graphics processor to coalesce memory access improving performance [28]. Secondly, using partial sums helps offset the single-point precision of the graphics-processor.

Random number generators for both our graphics- and core-processor implementations are based on the well-known Mersenne Twister 19937 algorithm, first published by Matsumoto and Nishimura in 1998 [29]. The core-processor implementation uses the Karney Mersenne Twister library [30], an interface to the SIMD fast Mersenne Twister library [31]. This algorithm takes advantage of parallel features of modern CPUs such as multistage pipelining and SIMD instructions. The graphics processor implementation uses an adaptation of the Mills multithreaded implementation of the Mersenne Twister algorithm [32]. Mills implementation generates a 64-bit random number simultaneously for up to 623 threads. We use the lower half of this to generate 32-bit random numbers.

The core-processor pseudo random number generator is seeded by a single integer and maintains a 624 × 32-bit state from which pseudo random numbers are computed sequentially. In the graphics-processor implementation a separate stream of random numbers is required for each photon traced in a batch. Ideally, a 21,504 dimension custom random number generator would be created to provide these streams following, for example, Matsumoto and Nishimura [33]. However, this is time consuming. So, for this performance comparison, Matsumoto and Nishimura's 623-dimension generator is used and a separate state maintained for each thread-block. For 100 blocks per grid, this requires about 250 KB of global memory. The generator is keyed to a thread identifier, which provides highly independent random number streams within each thread block. A separate random number generator, seeded by a single integer, is used to generate seeds to initialise these states.

The standard Monte Carlo roulette scheme to conserve energy undermines parallel execution of the algorithm by providing a 10%, for example, chance of each photon surviving the first roulette test [25]. Parallel processing on a single-instruction, multiple-data would leave 90% of the processors idle in this arrangement. To avoid this in the graphics-processor implementation, we apply roulette at the thread-block level. That is, instead of each photon having a 0.1 chance of passing roulette, each block of 256 photons has a 0.1 chance of passing roulette. If the *block* passes, tracing of *all* photons in the block continues until the next roulette test. As the same proportion of photons pass roulette in both the core- and graphics-processor implementations, this does not affect the simulation result provided a large number of blocks are executed; energy remains conserved.

As mentioned above, execution of the photon-tracing code is broken into a grid consisting of blocks of threads. The configuration is typically selected to suit the algorithm and maximise performance within the hardware constraints. The Monte Carlo algorithm provides one constraint: the random number generation algorithm relies on between 227 and 312 threads per block (though this could be eliminated by selecting a different generator). Hardware provides the second constraint. The photon-tracing engine requires 28 registers. This limits the maximum number of threads to 256 per block [28]. Finally, each grid execution must complete within five seconds, otherwise the operating system will terminate the call. This limitation is to help prevent errant drivers from locking up the system (with an infinite loop, for example), but does not apply if a display is not attached

to the graphics card. With 256 threads per block, this limitation sets an upper limit of about 84 blocks per grid so that tracing can complete within five seconds for the range of optical properties of interest. A multiple of 14 was selected for the block-size to allow even distribution of blocks over the 14 multiprocessors in the 8800GT card.

Monte Carlo simulations have been run on both implementations for a range of optical properties by varying the absorption (0.005 to 0.07 cm^{-1} in five equally spaced steps) and reduced scattering coefficients (3.8 to 9.8 cm^{-1} in five equally spaced steps). Anisotropy was held constant at 0.578. The values selected are typical of 1% Intralipid, an optical phantom, and biological tissue at near-infrared wavelengths [34]. Performance was measured by the number of photons traced per second and the number of scattering events per second in each case. Performance was also measured as a function of blocks per grid for the graphics processor at a single absorption of 0.064 cm^{-1} and reduced scattering of 9.25 cm^{-1} to judge the impact on runtime configuration.

Results and Discussion

We have not heavily optimised either implementation favouring instead a straightforward implementation of the Monte Carlo algorithm for comparison of the relative performance of core- and graphics-processor platforms. Consequently higher performance is likely to be possible from both algorithms with careful optimization, such as careful code tuning and employing the techniques of Zolek et al. [35].

The tracing speed was estimated for different grid sizes to determine the optimal number of blocks to include within each execution batch (Figure 2). We found the total simulation time consistently exceeded five seconds, the limit imposed by the operating system, when absorption was low (0.006 cm^{-1}) and more than 100 blocks were included making this a practical upper limit. A saw-tooth relationship was observed. The magnitude of the saw-tooth drops from about 10% to less than 3% as the number of thread-blocks in each grid increases. Saw-tooth peaks generally coincided with integral multiples of 14. As the graphics processor contains 14 multiprocessors and each block may only execute on one multiprocessor, performance degradation is probably caused by idle multiprocessors. Based on these results, 84 blocks were selected for each batch to maximise performance and reduce the risk of incomplete simulations.

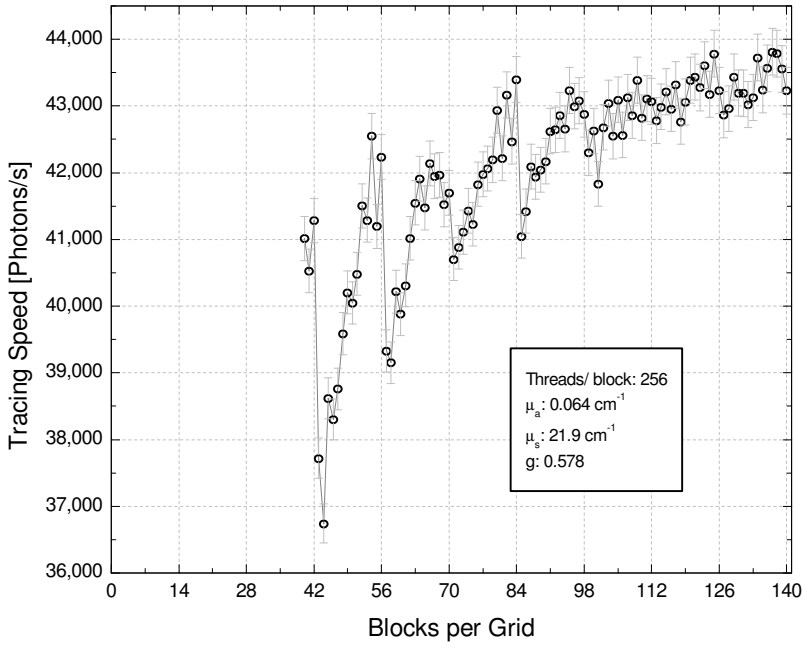

Figure 2. Graphics-processor simulation performance is plotted as a function of the number of blocks within each grid. Note: the 8800GT card used for these simulations has 14 multiprocessors. Data points are connected for clarity; fractional blocks are not possible.

Figure 3 shows the fluence rate calculated from a simulation of 258,048 photons using the core- and graphics-processors implementations for low (0.0064 cm^{-1}) and high (0.064 cm^{-1}) absorption coefficient. The mean of fifty replicate simulations, along with a 95% confidence interval, is plotted for the graphics-processor. A single-replicate is plotted for the core-processor, as the simulation speed is much slower. The reduced scattering coefficient (9.259 cm^{-1}) and anisotropy (0.578) were the same in each case. The result from the core-processor falls largely within the 95% confidence interval estimated on the graphics-processor indicating both simulations are producing the same result within the experimental error.

As expected, fluence initially drops very quickly as the light is scattered close to the source, then follows an exponential decay in the far-field as absorption dominates. At higher absorption, the light is more quickly attenuated; by 5 cm, there is a drop of nearly two orders of magnitude in the fluence rate. Scattering increases the path travelled by the photons, increasing the loss to absorption more quickly than the physical distance would suggest.

Precise Monte Carlo simulations require observation of a large number of photons. At the higher absorption, most of the photons are absorbed nearer the source, so few reach

beyond 8 cm. This is apparent in Figure 3: particularly at the higher absorption (lower trace). As the fluence rate decreases, the size of the 95% confidence-interval for the fluence-rate increases with increasing distance from the source indicating greater uncertainty or, equivalently, more noise. Replication on the graphics-processor has allowed us to estimate these uncertainties in the simulation data reasonably quickly. However the mean also represents the fluence-rate estimated by tracing nearly 13 million photons. The higher speed of the graphics processor enables such large simulations and, because noise is related to the number of photons detected, can provide more accurate results in a shorter time.

The simulation time on the core-processor was 72 minutes (low absorption) and 7.2 minutes (high absorption). On the graphics-processor the simulation took 60.8 ± 0.5 s (low absorption) and 6.09 ± 0.05 s (high absorption); each execution batch contained 84 blocks with 256 threads in each block and was repeated 12 times for a total of 258,048 photons. A 0.8% standard error in the simulation time on the graphic-processor was observed, which we attribute to the statistical nature of the simulation. The timing error was not estimated for the core-processor implementation because each simulation was very time consuming.

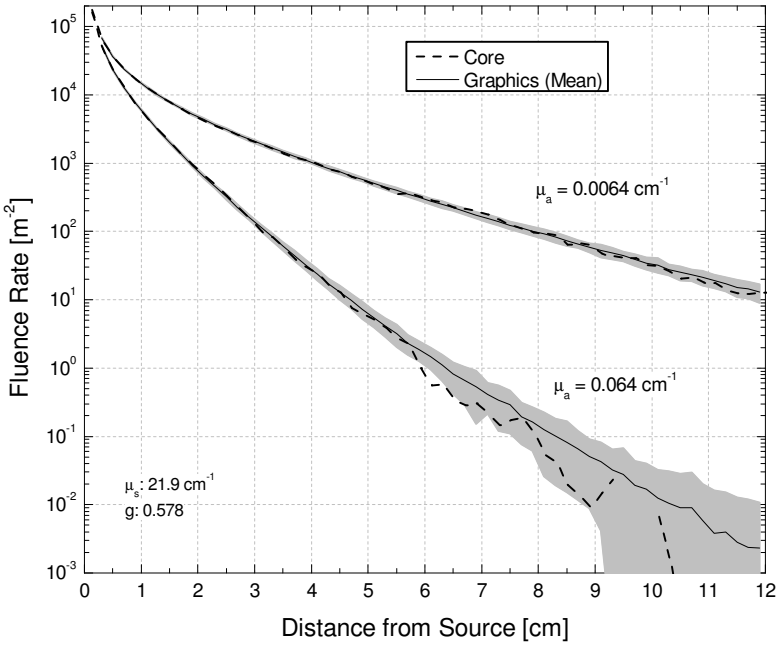

Figure 3. The fluence rate calculated by the core- and graphics-processor implementations is compared for high and low absorption. A mean and 95% confidence interval (grey fill) of 50 repeats is shown for the graphics-processor implementation; a single simulation result is shown for the core-processor implementation.

The simulations were repeated at a range of optical properties on both the graphics and core-processors. The core-processor simulations were repeated with single (32-bit) and double-precision (64 bit) arithmetic. The performance, in photons traced per second, is plotted in Figure 4 for the graphics and double-precision core-processor simulations. The difference in performance between single and double-precision arithmetic was less than 3%, though we have not investigated if this difference is statistically significant. We found a predominantly linear relationship between the ratio of absorption and scattering, and the tracing speed. The Monte Carlo algorithm implemented on the NVIDIA 8800GT graphics processor was consistently more than 70 times faster than the same algorithm implemented, as a single thread, on a 2.67 GHz Intel Duo processor.

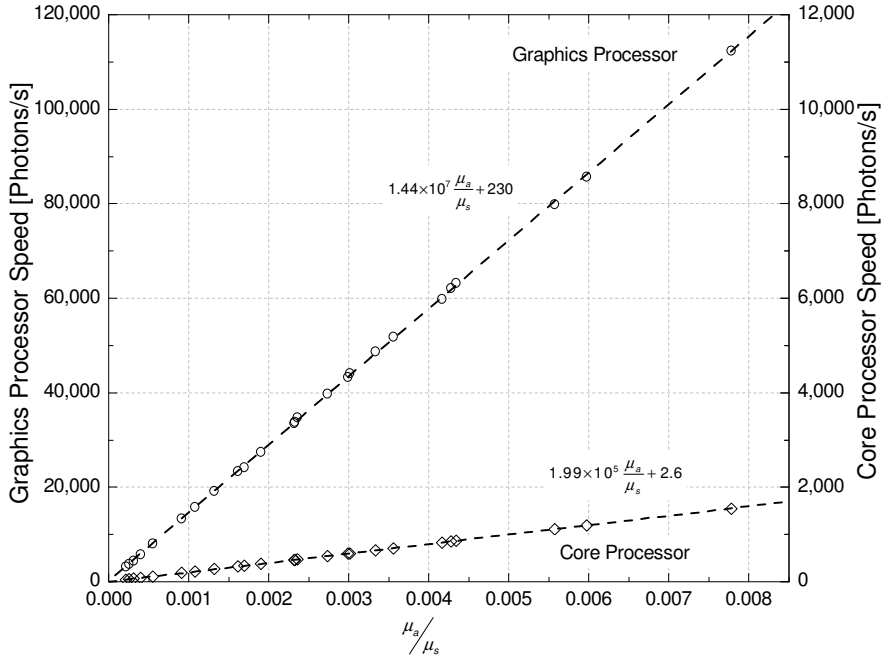

Figure 4. The simulation speed on a graphics- (left scale) and core-processor (right scale) for a range of optical properties, typical of 1% Intralipid between 700 and 1000 nm, is plotted.

The driving factor behind the simulation time is the albedo: the photon's probability of surviving a chain of scattering-absorption events is multiplied by the albedo for each event until the probability drops below the roulette threshold. To a first approximation (neglecting roulette), the number of scattering events per photon is given by:

$$N = \frac{\log(T_1)}{\log(\alpha)}$$

Here, T_1, the threshold for invoking roulette is 0.001 and α is the medium albedo, $\mu_s/(\mu_a + \mu_s)$. For a roulette probability of 0.1, this underestimates the number of scattering events by just over 10%. However, it allows a rough estimate of the rate at which scattering events are processed by the two implementations: about 1.5 million and 110 million scattering events per second on the core- and graphics-processor implementations, respectively. In passing, we note that even the core-processor version was about 80 times faster than our original implementation in Matlab. It is likely the performance of the Matlab version could be increased using the Matlab compiler, however this toolbox was not available to us.

Since these measurements were made, NVIDIA have released two models in the next generation of graphics processors: the GTX280 and the GTX260. The top end GTX280 has

30 multiprocessors: more than double that of the 8800GT. We expect the performance of the graphics processor implementation presented here would double if twice as many multiprocessors were available for the simulation. Additional optimization of the algorithm, such as trading precision for speed in trigonometric functions [35], or implementation to better suit the parallel execution environment may also boost performance. Early investigations into moving the pseudo random number generator state from global to faster shared memory suggest this would double tracing speed. We hope to investigate this further in the future. It appears an additional order of magnitude increase in speed is not unrealistic with current hardware nor is this an immovable limit: significant advances in graphics processor performance occur at least annually.

This work has focused on the relatively trivial example of an isotropic source in an infinite, homogeneous medium to test the suitability of graphics cards to this problem. For practical problems, the algorithm must be extended to support anisotropic sources, such as optical fibres, the complex geometries of human, animal or plant tissues and heterogeneous media. The first is trivial: the program could be easily extended to arbitrarily complex light sources. Moving to complex geometries and materials will be more difficult, however, because the single-instruction, multiple-data architecture of the graphics processor makes a large contribution to performance. In the context of our Monte Carlo simulation, this means that the same step in the algorithm is applied to every photon simultaneously. In other words, the photons are distinguished only by their data, not by their stage in the algorithm. The challenge in implementing more complex geometries will be maintaining the parallel structure of the algorithm. Though not trivial, this is not unrealistic. Graphics cards were created to implement complex, interactive 3-D worlds for computer games so appear well suited to the task and is the subject of continuing research.

Conclusion

The Monte Carlo algorithm for simulating photon transport in turbid media has been implemented on a standard desktop computer and modern graphics processor to assess relative performance. The parallel graphics processor implementation was found to trace photons more than 70 times faster than the single-threaded desktop computer implementation, processing roughly 110 million scattering events per second. We believe

this result suggests graphics cards offer a significant performance and cost advantage over distributed computing clusters for modelling light transport in scattering media or complex optical systems.

Acknowledgements

This work was supported with funding from the New Zealand Foundation for Research, Science and Technology (C06x0402).

1 D. Rogers, Phys. Med. Biol. 51 (2006) R287–R301.

2 D. Fraser, R. Jordan, R. Künnemeyer, V. McGlone, Postharvest Biol. Tec. 27 (2003) 185

3 S. Flock, B. Wilson, M. Patterson, IEEE T. Bio-med. Eng. 36 (1989) 1169

4 G. Palmer, C. Zhu, T. Breslin, F. Xu, K. Gilchrist, N. Ramanujam, Appl. Opt. 45 (2006) 1072

5 H. Xu, T. Farrell, M. Patterson, J. Biomed. Opt. 11 041104, (2006) 1

6 F. Martelli, M. Bassani, L. Alianelli, L. Zangheri, G. Zaccanti, Phys. Med. Biol. 45 (2000) 1359

7 D. Bates, J. Porter, J. Quant. Spectrosc. Radiat. Transf. 109 (2008) 1802

8 M. Nichols, E. Hull, T. Foster, Appl. Opt. 36 (1997) 93

9 G. Palmer, N. Ramanujam, Appl. Opt. 45 (2006) 1062

10 Z. Ma, H. Zhao, Y. Tanikawa, Feng Gao, Proc. SPIE 6434, (2007) 64342C

11 F. Bevilacqua, A. Dunn, J. You, J. Tromberg, V. Venugopalan, Opt. Lett. 26 (17), (2001) 1335

12 J. Wood, H. Al-Bahadili, Ann. Nucl. Energy 17 (1990) 465

13 D. Kirkby, D. Delpy, Phys. Med. Biol. 42 (1997) 1203

14 A. Colasanti, G. Guida, A. Kisslinger, R. Liuzzi, M. Quarto, P. Riccio, G. Roberti, F. Villani, Comput. Phys. Commun. 132 (2000) 84

15 Y. Dewaraja, M. Ljungberg, A. Majumdar, A. Bose, K. Koral, Comput. Meth. Prog. Bio. 67 (2002) 115

16 J. Giersch, A. Weidemann, G. Anton, Nucl. Instrum. Meth. A. 509 (2003) 151–156.

17 M. Thomason, R. Longton, J. Gregor, G. Smith, R. Hutson, Comput. Meth. Prog. Bio. 75 (2004) 251

18 T. Binzoni, T. Leung, R. Giust, D. Rüfenacht, A. Gandjbakhche, Comput. Meth. Prog. Bio. 89 (2008) 14

19 Top 500 supercomputer sites project, http://www.top500.org/tags/june2008

20 A. Ruiz, M. Ujaldon, L. Cooper, K. Huang, Journal of Signal Processing Systems (2008) in press

21 J. Molemaker, J. Cohen, S. Patel, J. Noh, in: ACM SIGGRAPH Symposium on Computer Animation, ed. M. Gross, D. James, (2008)

22 A. Anderson, W. Goddard, P. Schröder, Comput. Phys. Comm. 177 (2007) 298

23 L. Nyland, M. Harris, J. Prins, in: *GPU Gems 3,* ed. H. Nguyen, (Addison-Wesley, 2007) 677

24 W. Jeong, P. Fletcher, R. Tao, R. Whitaker, IEEE T. Vis. Comput. Gr. 13 (2007) 1480–1487.

25 S. Prahl, M. Keijzer, Sl Jacques, A. Welch, SPIE Institute Series 5 (1989) 102

26 S. Jacque, L. Wang, in: Optical-Thermal Response of Laser-Irradiated Tissue, ed. A. Welch, M. van Gemert (Plenum Press, New York, 1995)

27 http://cpc.cs.qub.ac.uk/

28 NVIDIA Corporation, http://www.nvidia.com/object/cuda_develop.html

29 M. Matsumoto, T. Nishimura, ACM Trans. on Modelling and Computer Simulation, 8 (1998) 3

30 C. Karney, `http://charles.karney.info/random/`

31 M. Saito, M. Matsumoto, in: Monte Carlo and Quasi-Monte Carlo Methods 2006, ed. A. Keller, S. Heinrich, H. Niederreiter (Springer, 2008) 607

32 E. Mills, `http://forums.nvidia.com/index.php?showtopic=31159&st=20&p=217039&#entry217039`

33 M. Matsumoto, T. Nishimura, in: Monte Carlo and Quasi-Monte Carlo Methods 1998, ed. H. Niederreiter, J. Spanier (Springer, 2000) p 56

34 V. A. McGlone, P. Martinsen, R. Künnemeyer, B. Jordan, B. Cletus, Phys. Med. Biol. 52 (2007) 2367

35 N. Zolek, A. Liebert, R. Maniewski, Comput. Meth. Prog. Bio. 84 (2006) 50