Phys. Med. Biol. **56** (2011) 7145–7162 doi:10.1088/0031-9155/56/22/010

A GPU implementation of EGSnrc's Monte Carlo photon transport for imaging applications

Jonas Lippuner[1,2] **and Idris A Elbakri**[1,3,4]

[1] Department of Physics and Astronomy, University of Manitoba, Winnipeg, Manitoba, R3T 2N2, Canada
[2] Department of Mathematics, University of Manitoba, Winnipeg, Manitoba, R3T 2N2, Canada
[3] Department of Radiology, University of Manitoba, Winnipeg, Manitoba, R3T 2N2, Canada
[4] CancerCare Manitoba, Department of Medical Physics, 675 McDermot Ave, Winnipeg, Manitoba, R3E 0V9, Canada

E-mail: jonas@lippuner.ca

Received 25 May 2011, in final form 25 August 2011
Published 25 October 2011
Online at stacks.iop.org/PMB/56/7145

Abstract

EGSnrc is a well-known Monte Carlo simulation package for coupled electron–photon transport that is widely used in medical physics application. This paper proposes a parallel implementation of the photon transport mechanism of EGSnrc for graphics processing units (GPUs) using NVIDIA's Compute Unified Device Architecture (CUDA). The implementation is specifically designed for imaging applications in the diagnostic energy range and does not model electrons. No approximations or simplifications of the original EGSnrc code were made other than using single floating-point precision instead of double precision and a different random number generator. To avoid performance penalties due to the random nature of the Monte Carlo method, the simulation was divided into smaller steps that could easily be performed in a parallel fashion suitable for GPUs. Speedups of 20 to 40 times for 64^3 to 256^3 voxels were observed while the accuracy of the simulation was preserved. A detailed analysis of the differences between the CUDA simulation and the original EGSnrc was conducted. The two simulations were found to produce equivalent results for scattered photons and an overall systematic deviation of less than 0.08% was observed for primary photons.

(Some figures in this article are in colour only in the electronic version)

1. Introduction

Monte Carlo simulations are widely used to simulate the effects of ionizing radiation in medical physics applications, such as radiotherapy treatments and x-ray imaging (e.g. Raeside (1976), Andreo (1991), Verhaegen and Seuntjens (2003), Rogers (2006)). Monte Carlo simulations can produce highly accurate results, but often require long computation times. A well-known,

multi-purpose Monte Carlo simulation package is EGSnrc (Kawrakow *et al* 2010), which is designed for coupled transport of electrons and photons through an arbitrary geometry in the energy range of 1 keV to 10 GeV.

A relatively new approach to accelerating computationally intensive simulations is the use of graphics processing units (GPUs) instead of central processing units (CPUs). Over the last few years, GPUs have evolved from specialized microprocessors for graphics rendering to programmable, highly parallel, multi-core processors that can be used for general-purpose computations (Blythe 2008). Modern GPUs consist of hundreds of cores capable of executing thousands of instances of a single program simultaneously with different input data. NVIDIA developed the Compute Unified Device Architecture (CUDA) (NVIDIA 2010*b*), which is a high-level GPU programming interface for C. With CUDA, one can write programs that execute on NVIDIA GPUs and take full advantage of the parallel architecture.

Parallel programming with GPUs, specifically using CUDA, has been successfully used to accelerate codes and simulations in various fields of science by up to three orders of magnitude (e.g. Shiraki *et al* (2009), Januszewskia and Kostur (2010)). Various Monte Carlo simulations have seen significant speedups of one, two or three orders of magnitude when implemented with CUDA (e.g. Preis *et al* (2009), Gulati and Khatri (2009), Alerstam *et al* (2008)). GPU computing has also become popular in medical physics (Pratx and Xing 2011). For example, GPU implementations of a ray tracing algorithm (Després *et al* 2008) and radiotherapy treatment planning (Gu *et al* 2009, Men *et al* 2009, Lo *et al* 2009) achieved significant speedups.

Some work has been carried out in accelerating Monte Carlo simulations for coupled electron–photon transport. Jia *et al* (2010) implemented the dose planning method (DPM) (Sempau *et al* 2000) with CUDA and achieved a speedup of 4.5 to 5.5 times. Hissoiny *et al* (2011) developed a new GPU code for coupled electron–photon transport with emphasis on dose calculation. They achieved a speedup of more than 900 times over EGSnrc and good agreement between their simulation results and EGSnrc's. Badal and Badano (2009) implemented the photon interaction models of PENELOPE 2006 (Salvat *et al* 2006) and achieved a speedup of up to 27 times for x-ray simulations.

In this paper, we present an implementation of the Monte Carlo photon transport mechanism of EGSnrc for CUDA. The main goal of this work is to make the simulation faster by implementing it for GPUs without changing the original method or introducing any approximations. The only exceptions to this are that we use a different random number generator (RNG) that is more suited for GPUs, and that we use single floating-point precision instead of double precision, because current GPUs are optimized for single precision. The code presented in this paper is specifically designed for imaging applications and only simulates photons propagating through a voxelized volume. Dose scoring and electrons are not implemented.

Note that there are many other techniques to improve the efficiency of EGSnrc and reduce the simulation time. For example, Kawrakow and Fippel (2000) investigated numerous variance reduction techniques and the Woodcock tracking algorithm (Woodcock *et al* 1965), also known as δ-scattering, is capable of producing a significant speedup for imaging oriented simulations such as this. However, such methods are not considered in this paper since our goal is to achieve a speedup solely by using CUDA. Furthermore, any modification to the EGSnrc code that would make it run faster can also be implemented in our CUDA version and would thus make that run faster too.

This paper is organized as follows: section 2 provides some technical background about the CUDA architecture and the functionality of EGSnrc. Section 3 gives a detailed description of our implementation with CUDA and the experiments that were performed to measure the

Table 1. Important terms related to CUDA.

Kernel	Function that is executed on the GPU
Thread	One instance of a kernel executing on the GPU
Warp	Group of 32 threads executing concurrently on a multiprocessor
Block	Larger group of threads that can communicate via shared memory
Global memory	Largest but slowest memory on the GPU that all threads can access
Registers	Fast but limited private memory of one thread (32 bits per register)

speedup and accuracy of our code. The results of the experiments are presented in section 4 and discussed in section 5.

2. Background

2.1. The CUDA architecture

Table 1 shows some of the most important terms related to CUDA used in this paper. When programming with CUDA, the main program is executed on the CPU and only certain functions, called *kernels*, are executed on the GPU. When a kernel is executed, thousands of *threads* are created on the GPU and each thread executes one instance of the kernel simultaneously with other threads. Threads are grouped into *blocks* and all threads within the same block can communicate with each other through shared memory (NVIDIA 2010*b*, chapter 2). Furthermore, all threads can access the same *global memory* space, which is the largest memory space (several gigabytes on modern cards), but also the slowest. Each thread has a set of private *registers* which are used to store intermediate results. One register holds 32 bits, e.g. one float. Accessing registers is fast, but the number of available registers is limited (NVIDIA 2010*a*, section 3.2).

The GPU itself consists of several multiprocessors and multiple thread blocks can be active at the same time on a single multiprocessor. Each multiprocessor always executes groups of 32 threads, called a *warp*, in parallel. The multiprocessors switch between different active warps to hide the memory latencies associated with reading and writing data from and to memory. While at any given time only instructions for 32 threads are actually executed in parallel on each multiprocessor, there are hundreds of active warps per multiprocessor that are running simultaneously (NVIDIA 2010*b*, chapter 4).

The instructions for one warp are executed in a single instruction multiple data (SIMD) fashion. Ideally, all threads in the warp perform the same calculation on different data. Full efficiency is achieved when all threads share the same execution path. If the execution paths of some threads diverge because of data-dependent conditions, the different branches will be executed serially, which can drastically decrease the performance. It is therefore imperative that the threads in a warp diverge as little as possible. This only applies to the threads in the same warp, because different warps execute independently in any case (NVIDIA 2010*a*, chapter 6).

2.2. Functionality of EGSnrc

EGSnrc uses random inputs following empirical or theoretical distributions to simulate the propagation of one particle (photon, electron or positron) through the simulation geometry. This is called a *history*, and many histories are simulated to obtain the final simulation result. A history is started by determining the initial properties of the particle and then calling the

subroutine SHOWER. During the simulation, secondary particles may be created. The *stack* is the list of all particles active in the history and contains all properties of the particles, e.g. energy, position, direction, etc. SHOWER transports all particles on the stack through the simulation geometry until all are discarded because they leave the simulation geometry or their energy falls below a cutoff. SHOWER calls the subroutines PHOTON and ELECTR, which handle photon and electron/positron transport, respectively. These two subroutines call other subroutines to simulate different interactions. They also call HOWFAR and HOWNEAR, which are supplied by the user to specify the simulation geometry; this is called the *user code*. The subroutine AUSGAB is called before and after different events and can be used in the user code for scoring various quantities (Kawrakow *et al* 2010, section 3.2).

The EGSnrc core and many of its most widely used user codes are written in Mortran, which is a macro language that generates Fortran code. The EGSnrc C++ class library (Kawrakow *et al* 2009) provides a C++ interface for EGSnrc and allows one to write user codes in C++ without having to write a single line of Mortran or Fortran. However, such C++ user code still uses the Mortran/Fortran core of EGSnrc that implements the actual Monte Carlo simulation and the physics model.

3. Methods

3.1. Implementation with CUDA

In this work, we concentrate on the photon transport model. Our implementation is specific to imaging a voxelized volume onto an image plane; hence we restrict ourselves to the transport of photons only and completely ignore electrons.

3.1.1. Parallelization of the simulation. Parallelizing the simulation is not as simple as running multiple histories in parallel since the individual histories diverge very quickly. If each thread runs one history, the warps will probably diverge so strongly that the CUDA implementation may be slower than the original EGSnrc. To reduce the divergence, we divide the simulation of one history into the following steps.

- *Create new photon:* create a new photon and start a history; this is equivalent to SHOWER.
- *Transport photon one step:* transport a photon one step through the geometry, determine if an interaction takes place and, if so, which one. This is equivalent to PHOTO and includes HOWFAR for the geometry specifications. HOWNEAR is not needed, because it is only used for electron transport. Since only voxelized volumes are supported, the HOWFAR implementation of the EGS_XYZGeometry class from the EGSnrc C++ class library is used.
- *Interactions:* Rayleigh, Compton, photoelectric and pair production interactions are modeled. The Compton interaction only changes the energy and direction of the photon and does not create an electron. The photoelectric and pair production interactions simply destroy the photon without creating other particles.
- *Propagate to image plane:* propagate a photon that has left the simulation geometry to the image plane and score its energy in the appropriate pixel.

The CUDA code is written entirely in C. The code pieces of the individual simulation steps were extracted from the original EGSnrc Mortran code and rewritten in C. No changes to the logic of the EGSnrc code were made. The only difference between the original code and ours was that the latter used single floating-point precision instead of double precision and that in

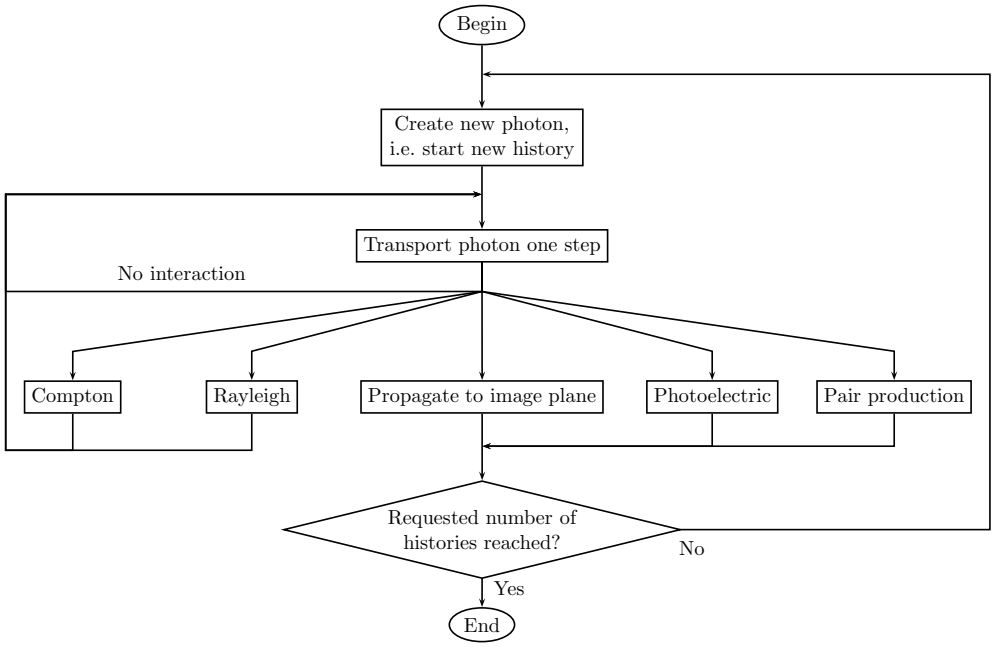


Figure 1. Flowchart showing how a history is simulated, what the possible simulation steps are and how they relate to one other.

some cases the flow was changed to make it more suitable for GPUs. Our code still performed the same calculations in the same manner, so the underlying physics model was not changed.

Note that we did not implement a step equivalent to `AUSGAB`. Instead, we integrated the functionality of `AUSGAB` into the other parts, e.g. counting scattering events in the functions that model Rayleigh and Compton interactions, or using a specific function for scoring a particular quantity, like the propagation step.

The calculations for the individual simulation steps are mostly independent of the current location, direction and energy of the photon. So, there is almost no divergence between histories in the same simulation step. But the order of the steps in a particular history is completely different from the other histories. Hence, if two threads run different histories, they still diverge, but the divergence is easier to handle. The flowchart in figure 1 shows how a history is simulated, what the possible simulation steps are and how they relate to one other. A history is complete when the photon leaves the simulation geometry and is propagated to the image plane, or when a photoelectric or pair production interaction occurs which simply destroys the photon. A new history is immediately started if the requested number of histories has not been reached.

3.1.2. Simulation overview. The main simulation program executes on the CPU and first initializes the media data, simulation geometry and other simulation parameters. The data required for the simulation are then copied to the GPU and the simulation of the histories is performed on the GPU by repeatedly executing the simulation kernel followed by the summing kernel until the requested number of histories is reached.

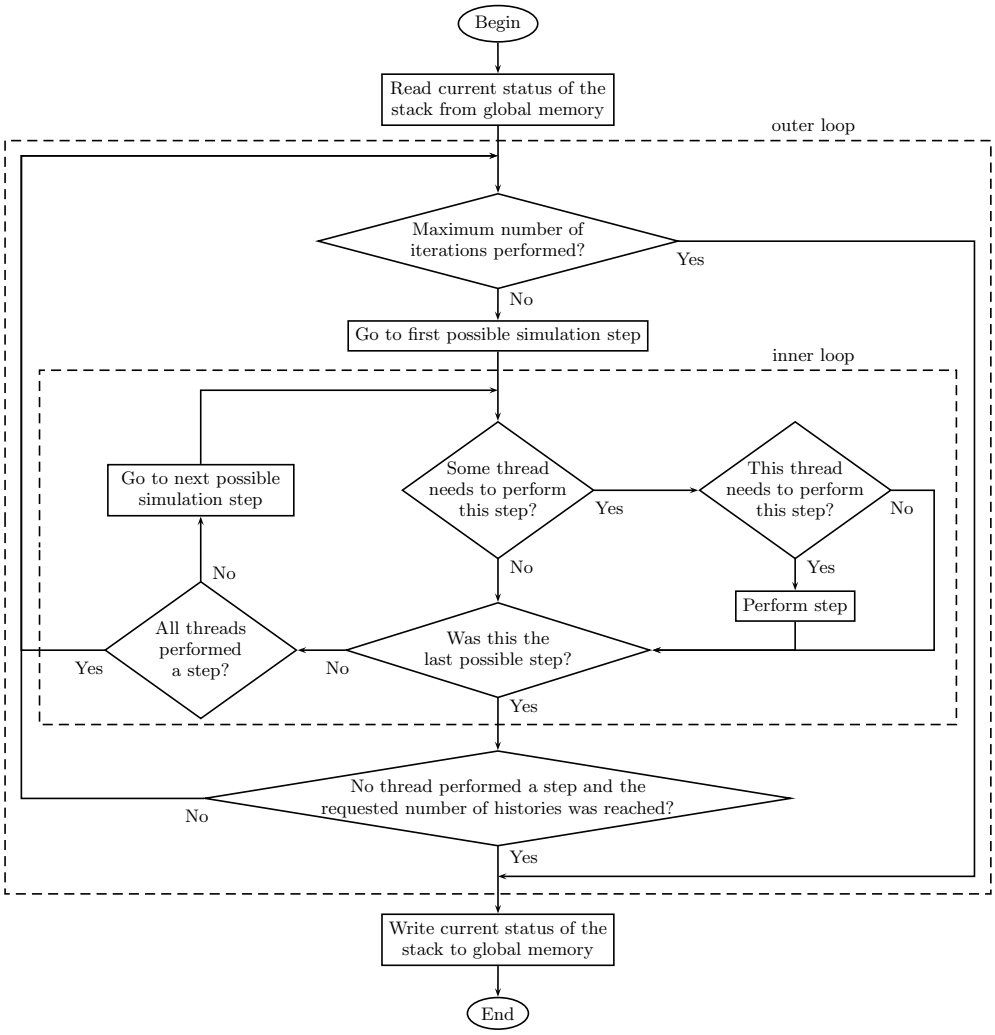


Figure 2. Flowchart of the simulation kernel executing on the GPU.

On the GPU, hundreds of histories are calculated in parallel. Each thread runs one history until it is finished and then starts a new one if more histories need to be simulated. In the simulation kernel, each thread performs thousands of simulation steps, thus simulating multiple histories. Once the simulation kernel is finished, the summing kernel is executed, which sums up the results of the simulation kernel. The simulation kernel is then executed again, followed by the summing kernel, until the simulation is finished. The following sections describe the two kernels in more detail.

3.1.3. Simulation kernel. Each thread has a private stack that contains exactly one photon. Figure 2 shows a flowchart of the simulation kernel executing on the GPU. When the kernel is started, each thread reads the current status of its stack from global memory and then keeps the stack in registers. Then the threads in the same warp step together through two nested loops.

In each iteration of the outer loop, each thread performs at least one simulation step. To ensure that no two threads in the same warp perform different simulation steps at the same time, the threads go together through the inner loop. Each iteration of the inner loop corresponds to one possible simulation step, i.e. transport step, Compton interaction, Rayleigh interaction, etc. Those threads that need to perform the current simulation step perform it while the others wait. Once each thread has performed at least one step or the last possible simulation step is reached, the inner loop is finished and the next iteration of the outer loop is started.

The outer loop is complete if a determined number of iterations have been performed, or no thread performed a simulation step and the requested number of histories has been reached; hence, no active particles are left and no new particles will be created. Once the outer loop is exited, the threads write the current status of their stack back to global memory and then the summing kernel is executed.

This scheme may seem very complicated and one might think that a simple `switch` statement would suffice since the GPU would automatically serialize the different branches of the `switch` statement in one warp. However, this is problematic because the current RNG (see section 3.1.5) requires all threads in the warp for calculating random numbers. With a `switch` statement, the threads that are not performing the current simulation step would be unavailable for calculating random numbers since they would be in a different section of the code. But with our looping approach, these inactive threads are still going through the code of the current step with the other threads and are thus available for calculating random numbers when needed. The inactive threads are disabled in the sense that they do not perform any memory accesses or calculations (except for calculating random numbers).

A simple measure for the degree of the divergence in the simulation is the average number of different simulation steps performed in the inner loop. If d different simulation steps were performed in one iteration of the outer loop, then each thread performed at least one simulation step, but in total, d different simulation steps were performed. So $1/d$ is a lower bound for the fraction of threads that were active on average during this iteration of the outer loop. It is only a lower bound because it is possible that some threads performed more than one simulation step. The order in which different steps are performed was chosen such that the most frequent steps are performed first, namely transport step, new particle, propagation to image plane, Compton interaction, photoelectric effect, Rayleigh interaction and pair production. So if in one history a transport step is performed and it is determined that a Compton interaction will occur next for this photon, then the Compton interaction is also performed when the inner loop reaches that simulation step.

3.1.4. Summing kernel. The scored quantity in our simulation is the energy of photons arriving at a pixelated image plane. Each element of the scoring array corresponds to one pixel of the image plane and holds the total amount of energy reaching that pixel. Once a photon leaves the simulation geometry, it is propagated to the image plane in a straight line determined by its exit location and direction, and its energy is added to the total energy already deposited in the pixel where it hits the plane.

Summing up a large number of values using floating-point arithmetic is prone to compounded rounding errors. One technique to reduce these errors is to first sum up fewer values and then add the subtotals together (Linz 1970). In order to do this, each thread block has a temporary scoring array that is initialized to zero every time the simulation kernel is launched. During the execution of the simulation kernel, the threads accumulate their results (in single precision) in the temporary scoring array of their block. After the simulation kernel is finished, the summing kernel adds up the temporary scoring arrays and accumulates those

subtotals in one total scoring array; this addition is done in double precision to further reduce rounding errors.

The temporary and total scoring arrays are stored in global memory. During the accumulation of individual energies into the temporary scoring arrays in the simulation kernel, it could happen that two threads need to add the energy of a photon into the same pixel. To avoid data hazards, the threads add the energies of their photons to the temporary scoring array using an atomic function, which ensures that the read, modify and write operations are completed without interference from any other threads.

3.1.5. Random number generator. Currently, the only RNG implemented in the EGSnrc C++ class library, which was used for the EGSnrc simulations, is RANMAR. It calculates random numbers serially and is thus not well suited for GPUs. On the one hand, if each thread used one instance of RANMAR with a different initial seed, too much shared memory would be required to store the states. On the other hand, if the threads of one warp shared one instance of RANMAR, then only one thread would be used to calculate all the random numbers, because this is done in a serial way, and the remaining 31 threads would be idle. Since many random numbers are used, this would result in a serious performance penalty.

For this reason, our code uses a different RNG, namely a variant of the Mersenne Twister (Matsumoto and Nishimura 1998) specifically designed for GPUs called Mersenne Twister for Graphics Processors (MTGP) (Saito 2010). The Mersenne Twister is a very widely used RNG that passes the diehard tests (Matsumoto and Nishimura 1998) and almost all of the more stringent TestU01 tests (L'Ecuyer and Simard 2007). The Mersenne prime exponent 3217 is used because it produces a large period ($2^{3217} - 1 \approx 10^{968}$) while only requiring 404 bytes to store its current state.

The main advantage of MTGP is that the random numbers are calculated in parallel. Each warp has its own Mersenne Twister and all the threads in the warp are used to calculate the random numbers in parallel. There are 101 random numbers in one status array and 32 threads in a warp. Hence, each thread can use three random numbers and then the status array has to be updated; the remaining five random numbers are not used. Furthermore, since all threads are required to update the status array, when some threads in the warp use a random number, all other threads also have to advance their counter, but without actually using that random number.

The author of MTGP also developed a program to generate various parameter sets for the MTGP, which was used to generate different parameter sets for each warp. All warps get the same initial seed, but since they all have different parameters, they will produce highly independent sequences (Matsumoto and Nishimura 2000).

Since the RNG in our CUDA implementation is different from the one used in EGSnrc, our simulation will not produce identical results. However, the results should be statistically equivalent to the EGSnrc results, i.e. the difference between the CUDA and EGSnrc simulations should be less than the combined statistical uncertainty. Even if we used RANMAR in our CUDA implementation, we would still not get the same results because our CUDA code is structured very differently. In EGSnrc one history is simulated after another, while in CUDA many histories are simulated in parallel. Therefore, even if we had the same sequence of random numbers, these numbers would be consumed in a very different way so that each CUDA history would effectively use a different sequence of random numbers than the EGSnrc histories.

3.2. Computational experiments

Two computational experiments comparing our proposed CUDA implementation to EGSnrc were conducted. The first experiment measured the speedup of the CUDA implementation

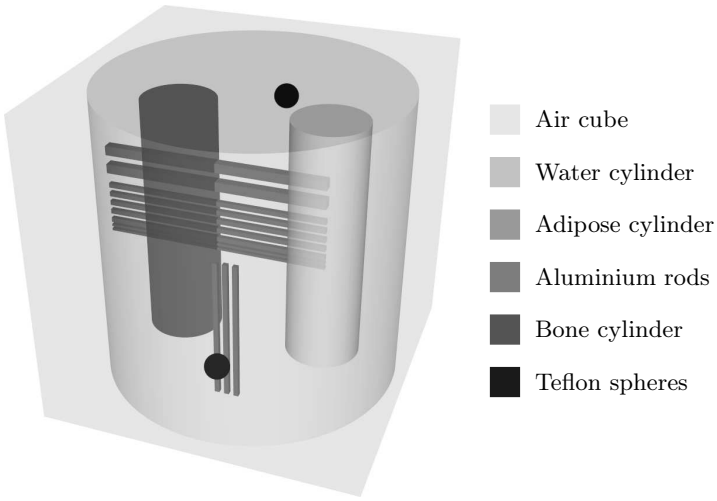


Figure 3. Frontal view of the unvoxelized phantom. Different voxelizations of this phantom were used in the CUDA and EGSnrc simulations.

versus EGSnrc for various numbers of voxels, and the second experiment measured the accuracy of the CUDA results compared to the EGSnrc results.

3.2.1. Simulation setup and hardware. The phantom used in both experiments was a water cylinder with 12 cm diameter and height embedded in an air cube with a side length of 12.8 cm. Two smaller cylinders (3 cm diameter, 10 cm high) consisting of bone and adipose tissue, as well as two teflon spheres with 1 cm diameter and various rectangular aluminum rods with thicknesses ranging from 1 mm to 4 mm, were placed inside the water cylinder. Different materials were used to test the agreement between the CUDA code and EGSnrc for weakly and strongly attenuating media. The phantom was voxelized to cubic voxels with 64, 128, 192 and 256 voxels in each dimension for the first experiment and 128 for the second. Figure 3 shows the unvoxelized phantom.

The x-ray source was a point source located 30 cm from the center of the phantom along its central axis. The energy spectrum of the source was a simulated 80 kVp spectrum of a tungsten target with 0.8 mm beryllium, 4 mm aluminum and 250 mm air filters. The point source was collimated onto the front surface of the phantom. The image plane was perpendicular to the central axis of the phantom and located 30 cm behind its center. The image plane consisted of 512×512 square pixels of side length 1 mm.

During the simulation, the number of Compton and Rayleigh scatter events that a photon underwent was tracked and when the photon was propagated to the image plane, it was assigned to one of the following four categories: primary (never scattered), Compton (Compton scattered once), Rayleigh (Rayleigh scattered once) and multiple scatter (scattered more than once). Each simulation produced five different images of the total energy fluence in each pixel of the image plane normalized to the number of histories performed. Four images corresponded to the four different photon categories and the last image was the sum of all categories, referred to as the total.

The EGSnrc simulations were performed with the user code Epp (Lippuner *et al* 2011), which was specifically designed for x-ray imaging and scatter analysis. Epp is based on the EGSnrc C++ class library and thus uses RANMAR. Electron transport in the underlying

EGSnrc code was completely disabled. The simulation kernel of the CUDA simulation was run in single precision, but the initialization, i.e. calculating media data, and the summing kernel were run in double precision. The EGSnrc code was run in single and double precision.

The EGSnrc simulations were compiled with g++ and gfortran under SUSE Linux 11 and run on the same machine, which had 16 GB RAM and two Intel Xeon Quad-Core X5460 CPUs with a clock rate of 3.16 GHz and 12 MB L2 Cache. However, only a single CPU thread was used for the simulations. The CUDA implementation was compiled with Microsoft Visual Studio 2008 under Microsoft Windows Vista using the CUDA Toolkit 3.1 and was run on the same machine on an EVGA GeForce GTX 470 SuperClocked video card. The GPU had 14 multiprocessors with a total of 448 CUDA cores and 1280 MB of global memory. The clock rates of the core, shader and memory (effective) were 625, 1250 and 3402 MHz, respectively. The simulation kernel was launched with two blocks per multiprocessor, 64 registers per thread, 512 threads per block and 32 768 iterations of the outer loop per kernel. Due to the large number of registers used, only one block could be active on a multiprocessor at any given time.

3.2.2. First experiment: speedup. Four simulations with different numbers of voxels N_{vox} were run. Each simulation was split up into $N_{\text{batch}} = 10$ batches using different random number seeds and each batch simulated 10^8 histories. The total CPU/GPU time T was the time the CPU or GPU spent in calculating histories. Note that the time required for the initialization has no influence on the speedup because it is not included in the CPU/GPU time. It makes sense to exclude the initialization time because it does not depend on the number of histories simulated. For comparison, however, the initialization time was measured separately.

The efficiency ε of a Monte Carlo simulation is (e.g. Bielajew and Rogers (1993))

$$\varepsilon = \frac{1}{s^2 T}, \tag{1}$$

where s is the total statistical uncertainty and T is the total CPU/GPU time. From batch b, we get the total energy fluence Φ_{bij} in the pixel (i, j). The statistical uncertainty $\Delta\Phi_{ij}$ in the pixel (i, j) is then given by

$$\Delta\Phi_{ij} = \sqrt{\frac{\langle \Phi_b^2 \rangle_{ij} - \langle \Phi_b \rangle_{ij}^2}{N_{\text{batch}} - 1}}. \tag{2}$$

The total statistical uncertainty s is the average of the uncertainties in all pixels:

$$s = \langle \Delta\Phi_{ij} \rangle. \tag{3}$$

Finally, the speedup of the CUDA simulation over EGSnrc is the ratio of the efficiencies:

$$\text{speedup} = \frac{\varepsilon^{\text{CUDA}}}{\varepsilon^{\text{EGS}}}. \tag{4}$$

For the CUDA simulations, the average number $\langle d \rangle$ of different simulation steps performed in one iteration of the outer loop was measured to quantify the degree of the divergence. $1/\langle d \rangle$ gives a lower bound for the fraction of threads that were active on average during the whole simulation.

3.2.3. Second experiment: accuracy. To investigate whether the CUDA results were statistically equivalent to the EGSnrc results and to quantify possible systematic deviations, we repeated the simulation with the 128^3 voxel phantom. The five different images (primary, Compton, Rayleigh, multiple scatter and total) were compared separately. To see whether the results of two simulations are statistically equivalent, a paired Student's t-test was used

to compare CUDA and EGSnrc in single and double precision. Additionally, a more detailed analysis was carried out to investigate the differences between CUDA and EGSnrc in double precision. To put the results into perspective, the same analysis was performed on two EGSnrc runs in double precision with different random number seeds. All simulations were run with $N_{\text{batch}} = 20$ and 10^8 histories per batch.

The test statistic t of the paired Student's t-test is given by

$$t = \frac{\langle D_{ij} \rangle}{\text{std}(D_{ij})/\sqrt{n}} \tag{5}$$

where

$$D_{ij} = \langle \Phi_b \rangle_{ij}^{\text{CUDA}} - \langle \Phi_b \rangle_{ij}^{\text{EGS}} \tag{6}$$

is the difference of the fluencies in pixel (i, j), $\text{std}(D_{ij})$ is the standard deviation of those differences and n is the number of pixels used in the analysis. From t, the two-sided p-value was calculated because the sign of the mean difference did not matter. The p-value is the probability that a test statistic as extreme or more extreme than the one measured is observed, assuming that the results of the two simulations are statistically equivalent. If we use the significance level 5%, we reject the null hypothesis that the results of two simulations are equivalent if the p-value is less than 0.05.

For the detailed analysis, we adopted the comparison method described by Kawrakow and Fippel (2000, section 2.5.1). Ideally, the normalized differences given by

$$x_{ij} = \frac{\langle D_{ij} \rangle}{\sqrt{\left(\Delta\Phi_{ij}^{\text{CUDA}}\right)^2 + \left(\Delta\Phi_{ij}^{\text{EGS}}\right)^2}} \tag{7}$$

should be normally distributed. The Kawrakow–Fippel method assumes that there are two distinct systematic deviations. That is, a fraction α_1 of all pixels has a systematic deviation of Δ_1 standard deviations, a fraction α_2 of all pixels has a systematic deviation of Δ_2 standard deviations and the remaining fraction $1 - \alpha_1 - \alpha_2$ of all pixels has no systematic deviation. The four parameters α_1, α_2, Δ_1 and Δ_2 then produce a fit for the measured distribution of the x_{ij}'s. We calculated the Cramér–von-Mises criterion ω^2 to quantify the total difference between the measured distribution and the fit. An advantage of ω^2 was that the data did not need to be binned and its value was not influenced by a bin size. In our case, ω^2 was only a relative goodness-of-fit measure, with a smaller value indicating a better fit.

To avoid large errors caused by poor photon statistics arriving at the image plane, only pixels which satisfied the following criteria were considered in the analysis.

(i) The fluence $\langle \Phi_b \rangle_{ij}$ was nonzero in the CUDA and EGSnrc images and so the statistical uncertainty $\Delta\Phi_{ij}$ was also nonzero in both images.

(ii) The statistical uncertainty was less than 50% of the fluence, i.e. $\Delta\Phi_{ij}/\langle \Phi_b \rangle_{ij} < 0.5$, in the CUDA and EGSnrc images.

4. Results

All quantities were rounded to 4 significant digits. In this and the following section, C stands for CUDA and Ed/Es stands for EGSnrc in double/single precision.

4.1. First experiment: speedup

Table 2 shows the total CPU/GPU time of all batches, the average initialization time of one batch and the average number of histories per second of C, Ed and Es for different numbers of voxels. Note in particular that the CPU/GPU time is the total of all batches and the initialization

Table 2. The total CPU/GPU time of all batches, the average initialization time of one batch and the average number of histories of one batch of CUDA (C), EGSnrc in double (Ed) and single (Es) precision for different numbers of voxels.

	$T_{\mathrm{CPU/GPU}}$ (s)			$\langle T_{\mathrm{init}} \rangle$ (s)		$\langle H/T_{\mathrm{CPU/GPU}} \rangle$ (s^{-1})		
$N_{\mathrm{vox}}^{1/3}$	C	Ed	Es	C	Ed/Es	C	Ed	Es
64	290.7	6 195	5 814	0.6408	0.1980	3 641 000	161 400	172 000
128	468.2	13 410	12 380	3.102	0.7800	2 253 000	74 600	80 770
192	612.1	22 660	21 960	9.186	2.406	1 657 000	44 140	45 540
256	780.0	31 600	30 890	20.65	5.625	1 313 000	31 640	32 370

Table 3. The speedups between CUDA (C), EGSnrc in double (Ed) and single (Es) precision, the average number $\langle d \rangle$ of different simulation steps performed in one iteration of the outer loop and the lower bound $1/\langle d \rangle$ for the fraction of threads that were active on average in CUDA for different numbers of voxels.

$N_{\mathrm{vox}}^{1/3}$	C versus Ed	C versus Es	Es versus Ed	$\langle d \rangle$	$1/\langle d \rangle$ (%)
64	22.58	21.16	1.067	2.240	44.64
128	30.18	27.84	1.084	1.788	55.93
192	37.66	36.39	1.035	1.579	63.35
256	41.54	40.57	1.024	1.459	68.56

Table 4. The two-sided p-values of the paired Student's t-test between CUDA (C), EGSnrc in double (Ed) and single (Es) precision for the different images.

Image	C versus Ed	C versus Es	Ed versus Ed	Es versus Ed
Primary	1.086×10^{-27}	5.408×10^{-28}	0.4237	0.9033
Compton	0.1136	0.4130	0.8737	0.4511
Rayleigh	0.2065	0.9898	0.1714	0.2057
Multiple	0.8259	0.9286	0.4232	0.7600
Total	2.604×10^{-23}	8.295×10^{-25}	0.3949	0.7012

time is the average for one batch, i.e. the amount of time that one simulation needs for the initialization, independent of the number of histories. The initialization times of Ed and Es were the same.

Table 3 gives the speedups between C, Ed and Es as defined in (4), the average number of different simulation steps performed in one iteration of the outer loop and the corresponding fraction of threads that were active on average in CUDA.

4.2. Second experiment: accuracy

Table 4 shows the two-sided p-values of the paired Student's t-test between C, Ed and Es for the different images. Table 5(a) shows the detailed comparison between C and Ed for the five images. $\langle x_{ij} \rangle$ and Δx_{ij} are the mean and standard deviation of the normalized differences, x_{ij}, respectively; α_1, Δ_1, α_2 and Δ_2 are the parameters obtained from the Kawrakow–Fippel analysis and ω^2 is the Cramér–von-Mises criterion. Table 5(b) shows the same data for the comparison between the two Ed runs.

Table 5. Detailed comparison of (a) C and Ed and (b) the two Ed runs. For the five images, the mean and standard deviation of the measured normalized differences, the four parameters obtained from the Kawrakow–Fippel analysis and the Cramér–von-Mises criterion ω^2 are given.

Image	$\langle x_{ij} \rangle$ (10^{-3})	Δx_{ij}	α_1 (%)	Δ_1	α_2 (%)	Δ_2	ω^2 (10^{-6})
(a) CUDA versus double precision EGSnrc							
Primary	−36.52	1.027	3.964	−1.025	0.1252	3.282	3.780
Compton	4.774	1.028	0.8715	1.944	0.6004	−2.026	0.8087
Rayleigh	7.985	1.022	0.8549	1.825	0.3400	−2.240	0.1956
Multiple	1.175	1.028	0.7524	1.991	0.7327	−1.884	0.8643
Total	−12.23	1.028	1.332	−1.591	0.3490	2.568	1.111
(b) Double precision EGSnrc versus double precision EGSnrc							
Primary	−3.487	1.026	0.8260	−1.877	0.6076	1.978	0.8417
Compton	−0.03935	1.029	0.8984	1.784	0.8587	−1.870	0.1628
Rayleigh	−3.194	1.025	0.9628	−1.643	0.6636	1.902	0.1636
Multiple	2.367	1.025	0.7780	1.832	0.5939	−2.001	0.7523
Total	0.4329	1.027	0.8455	1.828	0.8331	−1.804	0.3553

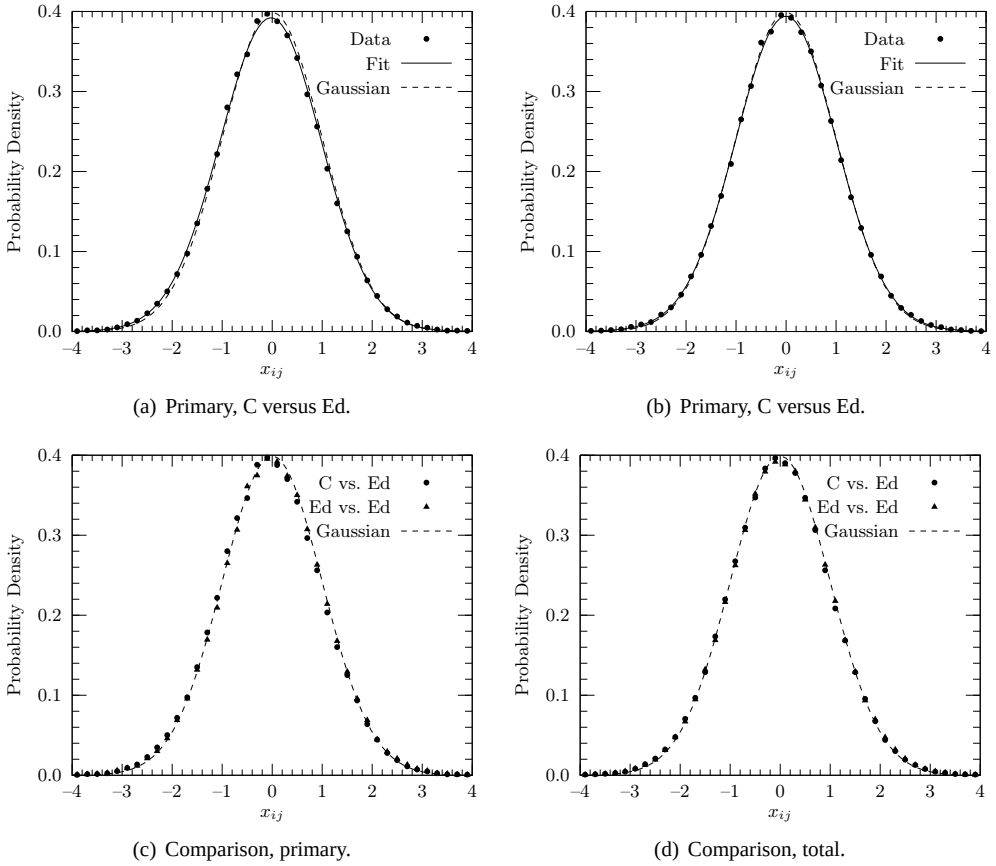
Figures 4(a) and (b) show the measured distribution of the x_{ij}'s and the fit obtained from the analysis of the primary images of C and Ed along with the Gaussian. The other images had similar or better agreement between the data and the fit, and between the data and the Gaussian. Figures 4(c) and (d) compare the measured differences between C and Ed, and between the two Ed runs in the primary and total images. The other images presented a very similar picture with good agreement between C versus Ed and Ed versus Ed and also between those and the Gaussian.

Finally, figure 5(a) is the total CUDA result image, shown on a log scale to enhance the visibility of structures inside the phantom. The dark area around the bright square is the shadow of the collimator and contains only scattered photons. The total result image of Ed (not shown) appeared identical except for statistical variations. Figure 5(b) shows the x_{ij}'s of the total image for C versus Ed. All other difference images, including those of Ed versus Ed, appeared virtually identical without any discernible structure or patterns. For increased contrast, the range $[-4, 4]$ was chosen. Less than 0.04% of the pixels fell outside of this range and they are shown as ± 4.

5. Discussion

5.1. First experiment: speedup

CUDA had longer initialization times than EGSnrc, but they were still comparable. The speedups of 20 to 40 times achieved by our CUDA implementation over Ed and Es are comparable to the speedups obtained by Badal and Badano (2009). Running EGSnrc in single precision resulted only in a marginal speedup. We can see clearly that, as the number of voxels increased, the average number of different simulation steps performed in one iteration of the outer loop decreased and thus more threads were active on average, resulting in a higher speedup. This happened because, as the number of voxels increased, their size decreased and a transport step did not step across a voxel boundary, so the fraction of transport steps increased, thus decreasing the divergence.



(a) Primary, C versus Ed.

(b) Primary, C versus Ed.

(c) Comparison, primary.

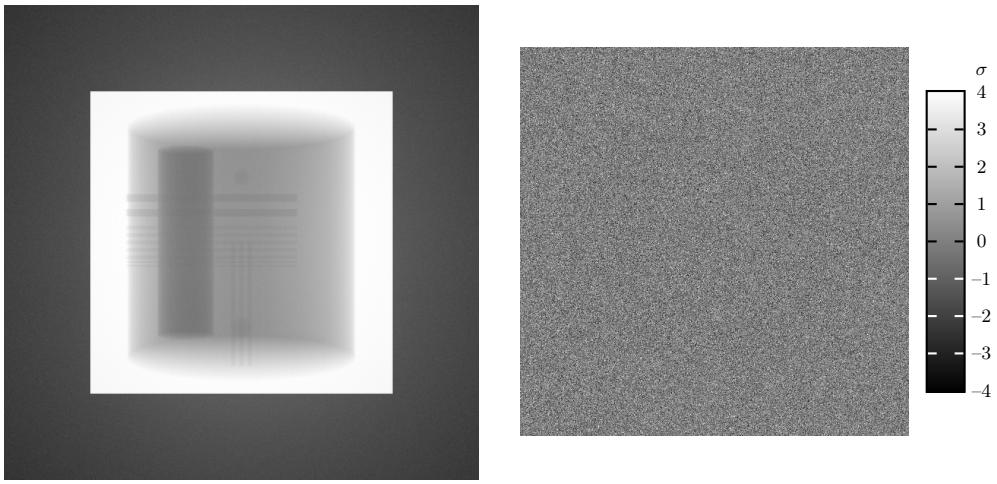
(d) Comparison, total.

Figure 4. Measured distribution of the differences in the primary image (a) between CUDA (C) and EGSnrc in double precision (Ed) and (b) between the two Ed runs. Comparison of the measured distributions of the differences between C and Ed and the two Ed runs in (c) the primary image and (d) the total image.

It is important to note that the speedup was measured with respect to a single CPU thread. The computer that ran the EGSnrc simulation was part of a Linux cluster consisting of many nodes, each of which had one or two CPUs with up to four cores. Typically, an EGSnrc simulation would be run in parallel by simply dividing the entire workload among several CPUs and/or CPU cores. Each CPU or CPU core would then run an independent simulation with different random number seeds and the results of all simulations would be combined at the end. This kind of task parallelism has been exploited for a long time (e.g. Kirkby and Delpy (1997)), but it is different from the data parallelism achieved in our CUDA implementation by processing instructions for many histories in parallel. To show the benefit of our approach, the simulation time of a single GPU using data parallelism is compared to the simulation time of a single CPU thread not using any parallel processing.

5.2. Second Experiment: accuracy

Table 4 shows that the *p*-values for Ed versus Ed and Es versus Ed are all much larger than 0.05; thus there is no evidence to suggest that the two Ed runs and Es and Ed were not equivalent.



(a) Total CUDA image in log scale.

(b) Total x_{ij}, C versus Ed.

Figure 5. (a) The total image of the CUDA simulation in a log scale. (b) The normalized differences x_{ij} in units of standard deviation between CUDA (C) and EGSnrc in double precision (Ed) in the total image.

This means that running EGSnrc in single precision introduces no measurable errors. The same goes for the scatter images (Compton, Rayleigh and multiple scatter) of C versus Ed and C versus Es. Further, the *p*-values of the scatter images of C versus Es are higher than for C versus Ed, indicating that our CUDA simulation is a bit closer to Es than Ed, which is to be expected. However, in the primary and total images, we clearly reject the null hypothesis that C and Ed or Es are equivalent.

Table 5 shows that the means of the x_{ij}'s were mostly of the order 10^{-3} or smaller while only two were of order 10^{-2}. The means of the differences between the two Ed runs tended to be smaller than those between C and Ed. The standard deviations of the x_{ij}'s were about the same in all cases and close to 1. Most α's were less than or close to 1%, indicating that only very few pixels had systematic deviations, which were mostly less than or about 2 standard deviations. Most values of ω^2 were about the same. Generally, the fit seemed to be better for the scatter images and worse for the primary and total images. The values of ω^2 tended to be higher for C versus Ed, indicating that the fit was slightly worse for those differences.

A closer look at the analysis parameters for Ed versus Ed (table 5(b)) reveals that $\alpha_1 \approx \alpha_2$ and $\Delta_1 \approx -\Delta_2$ for all five images. This means that the differences were about symmetrically distributed around 0, but their spread was slightly higher than that of the normal distribution. This can also be seen from the fact that the means were very close to 0 and the standard deviations were slightly larger than 1. It was found that the standard deviations, and thus the spread, decreased with increasing number of batches. We can thus conclude that the two Ed runs did produce statistically equivalent results for all images and the slightly higher spread of the differences arose from a small underestimation of the statistical uncertainty $\Delta\Phi_{ij}$ due to the relatively small sample size ($N_{\text{batch}} = 20$).

The differences between C and Ed (table 5(a)) in the scatter images presented a very similar picture. The differences were approximately symmetric and on the same order as for Ed versus Ed, as was the goodness of the fit. The primary image from C versus Ed showed the largest deviation from a standard normal distribution, which was also reflected in the total

image. Almost 4% of the pixels had a negative systematic deviation of about 1 standard deviations, while almost no pixels had a positive deviation. This means that CUDA tended to underestimate the energy fluence compared to Ed in the primary image, which also led to the relatively large negative mean difference. Since we are seeing *p*-values that are also very small for C versus Es in the primary and total images, we cannot attribute these differences to the fact that CUDA used single precision. So these systematic differences must be due to the different RNG that was used in CUDA. Further investigation would be necessary to determine which of the two generators produced the 'correct' results. But in any case, even though there are measurable systematic differences, these differences are very small. The average combined statistical uncertainty in the primary image was only about 1.7% of the average energy fluence, so about 4% of the pixels had a systematic difference of less than 2% ($1.1 \times 1.7\% = 1.87\%$), which gives an overall systematic difference of less than 0.08% ($4\% \times 2\% = 0.08\%$).

The fit in figure 4(a) seems to be quite good even though this one had the largest value for ω^2. Furthermore, despite the largest systematic deviation for this image, the measured distribution was still close to the Gaussian. As seen in the analysis, the fit for Ed versus Ed in the primary image (figure 4(b)) is closer to the Gaussian and it fits the data also quite well. Figure 4(c) shows that there are more negative differences between C and Ed than between Ed and Ed, again indicating that C slightly underestimated Ed in the primary image. For the total image (figure 4(d)), the agreement is better.

The spatial distribution of the differences between C and Ed in the total image (figure 5(b)) appears completely random. There are no visible structures in spite of the fact that there are sharp discontinuities in the result image (figure 5(a)). The largest systematic difference was found in the primary image which corresponds to the bright area around the cylinder, but this systematic difference is not visible in the difference image at all, thus implying that it is much smaller than the statistical uncertainty.

6. Conclusion

Our proposed CUDA implementation of EGSnrc achieves a speedup of 20 to 40 times over the conventional CPU implementation of EGSnrc in single or double precision for phantoms consisting of 64^3 to 256^3 voxels. This speedup is similar to what others (e.g. Badal and Badano (2009)) have found for x-ray Monte Carlo simulations. Multiple GPUs can be used to split up the workload and thus further reduce the simulation time. CUDA also has the advantage that it scales very well to newer hardware. Our implementation is easily portable to newer and faster video cards. Only a few simple parameters have to be adjusted to optimize our code for a different GPU and take full advantage of its capabilities.

We have also found that running EGSnrc in single precision only produces a very small speedup compared to double precision and there are no measurable differences. In the scatter images (Compton, Rayleigh and multiple scatter), our CUDA simulation is equivalent to EGSnrc in double precision. In the primary and total images, we found a measurable overall systematic difference of less than 0.08%, which cannot be attributed to the fact that CUDA used single floating-point precision.

The idea of dividing the simulation of a history into steps and performing the same steps in parallel in different threads to avoid warp divergence can be extended to the transport of electrons. In that case, an additional mechanism is needed to have multiple particles on the stack. While this will be more complex, a speedup over EGSnrc can still be expected. Even without electron tracking, dose scoring can be accommodated by estimating the dose from the photon energy deposited in each voxel. Such a method may produce acceptable results in the diagnostic energy range if the voxels are not too small.

Acknowledgements

This work was supported by funding from the CancerCare Manitoba Foundation. The authors thank Dr Harry Ingleby for helpful discussions and feedback and Dr H C Wolfart for valuable editorial comments.

References

Alerstam E, Svensson T and Andersson-Engels S 2008 Parallel computing with graphics processing units for high-speed Monte Carlo simulation of photon migration *J. Biomed. Opt.* **13** 060504

Andreo P 1991 Monte Carlo techniques in medical radiation physics *Phys. Med. Biol.* **36** 861–920

Badal A and Badano A 2009 Accelerating Monte Carlo simulations of photon transport in a voxelized geometry using a massively parallel graphics processing unit *Med. Phys.* **36** 4878–80

Bielajew A F and Rogers D W O 1993 Variance Reduction Techniques *NRC Report PIRS-0396* Institute for National Measurement Standards, National Research Council, Ottawa, Canada
http://rcwww.kek.jp/research/egs/docs/pdf/nrc-pirs0396.pdf

Blythe D 2008 Rise of the Graphics Processor *Proc. IEEE* **96** 761–78

Després P, Rinkel J, Hasegawa B H and Prevrhal S 2008 Stream processors: a new platform for Monte Carlo calculations *J. Phys.: Conf. Ser.* **102** 012007

Gu X, Choi D, Men C, Pan H, Majumdar A and Jiang S B 2009 GPU-based ultra-fast dose calculation using a finite size pencil beam model *Phys. Med. Biol.* **54** 6287–97

Gulati K and Khatri S P 2009 Accelerating statistical static timing analysis using graphics processing units *Asia and South Pacific Design Automation Conf., ASP-DAC 2009* pp 260–5

Hissoiny S, Ozell B, Bouchard H and Després P 2011 GPUMCD: A new GPU-oriented Monte Carlo dose calculation platform *Med. Phys.* **38** 754–64

Januszewskia M and Kostur M 2010 Accelerating numerical solution of stochastic differential equations with CUDA *Comput. Phys. Commun.* **181** 183–8

Jia X, Gu X, Sempau J, Choi D, Majumdar A and Jiang S B 2010 Development of a GPU-based Monte Carlo dose calculation code for coupled electron-photon transport *Phys. Med. Biol.* **55** 3077–86

Kawrakow I and Fippel M 2000 Investigation of variance reduction techniques for Monte Carlo photon dose calculation using XVMC *Phys. Med. Biol.* **45** 2163–83

Kawrakow I, Mainegra-Hing E, Rogers D, Tessier F and Walters B 2010 The EGSnrc Code System: Monte Carlo Simulation of Electron and Photon Transport *NRC Report PIRS-701* Ionizing Radiation Standards, National Research Council, Ottawa, Canada
http://irs.inms.nrc.ca/software/egsnrc-V4-2.3.1/documentation/pirs701/

Kawrakow I, Mainegra-Hing E, Tessier F and Walters B 2009 The EGSnrc C++ class library *NRC Report PIRS-898 (rev A)* Ionizing Radiation Standards, National Research Council, Ottawa, Canada
http://irs.inms.nrc.ca/software/egsnrc-V4-2.3.1/documentation/pirs898/

Kirkby D R and Delpy D T 1997 Parallel operation of Monte Carlo simulations on a diverse network of computers *Phys. Med. Biol.* **42** 1203–8

L'Ecuyer P and Simard R 2007 TestU01: A C library for empirical testing of random number generators *ACM Trans. Math. Softw.* **33** Article 22 / 1–40

Linz P 1970 Accurate floating-point summation *Commun. ACM* **13** 361–2

Lippuner J, Elbakri I A, Cui C and Ingleby H R 2011 Epp: A C++ EGSnrc user code for x-ray imaging and scattering simulations *Med. Phys.* **38** 1705–8

Lo W C Y, Han T D, Rose J and Lilge L 2009 GPU-accelerated Monte Carlo simulation for photodynamic therapy treatment planning *Proc. SPIE 7373* SPIE p 737313

Matsumoto M and Nishimura T 1998 Mersenne Twister: A 623-dimensionally equidistributed uniform pseudorandom number generator *ACM Trans. Modeling Comput. Simul.* **8** 3–30

Matsumoto M and Nishimura T 2000 Dynamic creation of pseudorandom number generators *Monte Carlo and Quasi-Monte Carlo Methods 1998* Springer Berlin pp 56–69
http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/DC/dc.html

Men C, Gu X, Choi D, Majumdar A, Zheng Z, Mueller K and Jiang S B 2009 GPU-based ultrafast IMRT plan optimization *Phys. Med. Biol.* **54** 6565–73

NVIDIA 2010*a NVIDIA CUDA C Best Practices Guide Version 3.1* NVIDIA Corporation
http://developer.download.nvidia.com/compute/cuda/3_1/toolkit/docs/NVIDIA_CUDA_C_BestPracticesGuide_3.1.pdf

NVIDIA 2010*b NVIDIA CUDA C Programming Guide Version 3.1.1* NVIDIA Corporation
http://developer.download.nvidia.com/compute/cuda/3_1/toolkit/docs/NVIDIA_CUDA_C_ProgrammingGuide_3.1.pdf

Pratx G and Xing L 2011 GPU computing in medical physics: A review *Med. Phys.* **38** 2685–97
Preis T, Virnau P, Paul W and Schneider J J 2009 GPU accelerated Monte Carlo simulation of the 2D and 3D Ising model *J. Comput. Phys.* **228** 4468–77
Raeside D E 1976 Monte Carlo principles and applications *Phys. Med. Biol.* **21** 181–97
Rogers D W O 2006 Fifty years of Monte Carlo simulations for medical physics *Phys. Med. Biol.* **51** R287–301
Saito M 2010 A Variant of Mersenne Twister Suitable for Graphic Processors arXiv:1005.4973v2 [cs.MS]
Salvat F, Fernández-Varea J M and Sempau J 2006 *PENELOPE-2006: A Code System for Monte Carlo Simulation of Electron and Photon Transport* OECD Nuclear Energy Agency Issy-les-Moulineaux, France `http://www.oecd-nea.org/science/pubs/2006/nea6222-penelope.pdf`
Sempau J, Wilderman S J and Bielajew A F 2000 DPM, a fast, accurate Monte Carlo code optimized for photon and electron radiotherapy treatment planning dose calculations *Phys. Med. Biol.* **45** 2263–91
Shiraki A, Takada N, Niwa M, Ichihashi Y, Shimobaba T, Masuda N and Ito T 2009 Simplified electroholographic color reconstruction system using graphics processing unit and liquid crystal display projector *Opt. Express* **17** 16038–45
Verhaegen F and Seuntjens J 2003 Monte carlo modelling of external radiotherapy photon beams *Phys. Med. Biol.* **48** R107–64
Woodcock E, Murphy T, Hemmings P and Longworth S 1965 Techniques used in the gem code for monte carlo neutronics calculations in reactors and other systems of complex geometry *Proc. Conf. on Applications of Computing Methods to Reactor Problems (Argonne National Laboratories Report ANL-7050)* p 557