Medical Physics
Current Issue | Authors | Submissions | Advertise | Search

Med Phys. Nov 2010; 37(11): 5593–5603. PMCID: PMC2967714
Published online Oct 6, 2010. doi: 10.1118/1.3490083

GPU-accelerated Monte Carlo convolution/superposition implementation for dose calculation

Bo Zhou[a] and Cedric X. Yu[b]

Department of Radiation Oncology, University of Maryland School of Medicine, Baltimore, Maryland 21201

Danny Z. Chen[c] and X. Sharon Hu[d]

Department of Computer Science and Engineering, University of Notre Dame, Notre Dame, Indiana 46556

a) Electronic addresses: bzhou@umd.edu and allen.bo.zhou@gmail.com
b) Electronic mail: cyu002@umaryland.edu
c) Electronic mail: dchen@nd.edu
d) Electronic mail: shu@nd.edu

Received December 19, 2009; Revised July 6, 2010; Accepted August 5, 2010.

This article has been corrected. See Med Phys. 2011 March 01; 38(3): 1732.

This article has been cited by other articles in PMC.

Abstract

Go to:

Purpose: Dose calculation is a key component in radiation treatment planning systems. Its performance and accuracy are crucial to the quality of treatment plans as emerging advanced radiation therapy technologies are exerting ever tighter constraints on dose calculation. A common practice is to choose either a deterministic method such as the convolution/superposition (CS) method for speed or a Monte Carlo (MC) method for accuracy. The goal of this work is to boost the performance of a hybrid Monte Carlo convolution/superposition (MCCS) method by devising a graphics processing unit (GPU) implementation so as to make the method practical for day-to-day usage.

Methods: Although the MCCS algorithm combines the merits of MC fluence generation and CS fluence transport, it is still not fast enough to be used as a day-to-day planning tool. To alleviate the speed issue of MC algorithms, the authors adopted MCCS as their target method and implemented a GPU-based version. In order to fully utilize the GPU computing power, the MCCS algorithm is modified to match the GPU hardware architecture. The performance of the authors' GPU-based implementation on an Nvidia GTX260 card is compared to a multithreaded software implementation on a quad-core system.

Results: A speedup in the range of 6.7–11.4× is observed for the clinical cases used. The less than 2% statistical fluctuation also indicates that the accuracy of the authors' GPU-based implementation is in good agreement with the results from the quad-core CPU implementation.

Conclusions: This work shows that GPU is a feasible and cost-efficient solution compared to other alternatives such as using cluster machines or field-programmable gate arrays for satisfying the increasing demands on computation speed and accuracy of dose calculation. But there are also inherent limitations of using GPU for accelerating MC-type applications, which are also analyzed in detail in this article.

Keywords: GPU acceleration, Monte Carlo, dose calculation

INTRODUCTION

Go to:

To meet the demands of high accuracy and high efficiency of radiation therapy (RT), newly emerging RT techniques employ a substantially larger number of radiation beams in the treatment plans[1, 2, 3] or apply online replanning strategies.[4, 5, 6] The use of a large number of beams and online replanning present a daunting challenge to dose calculation even for the state-of-the-art quad-core computing systems.

Current dose calculation methods are mainly classified into two categories: The deterministic methods and Monte Carlo (MC) methods. Deterministic methods, such as the convolution/superposition (CS) algorithms, use the concept of energy deposition kernel, which was first introduced by several independent investigators[7, 8, 9] and extended by Papanikolaou et al.[10] to polyenergetic spectra. Mackie et al.[11] and Boyer[12] later pointed out that separating a point kernel into primary dose and scatter dose kernels leads to better accuracy. Ahnesjö [13] proposed the collapsed-cone convolution/superposition (CCCS) method to deposit the energy directly onto the cone axis instead of the space covered by the cone. Reckwerdt and Mackie[14] reduced the computation time of the CCCS method by introducing a grid with parallel transport lines. In PINNACLE3,[15] the speed of computation is further improved by adaptively varying the resolution of the dose computation grid.

Since the CS model is derived from experimental approximation, a number of variations, such as polyenergetic beam sources, depth hardening, and kernel tilting, need to be considered to improve the accuracy.[10, 16] CS algorithms are normally viewed to be fast but less accurate than MC methods and the calculation time is typically proportional to the number of beams used. On the other hand, it has been widely accepted that MC methods provide the best accuracy for radiotherapy dose calculation. However, the long execution time limits the feasibility of MC methods in clinical applications. For example, a conventional IMRT (seven fields) plan can take 5–10 min to compute. For arc therapy planning computation on Varian's and Philips's systems, it still takes 10–20 min on eight-core CPUs for both the initial pencil beam calculation and final dose calculation. Thus, MC simulation is used mostly as a plan verification tool or for quality assurance.

Nonetheless, recent results by Tang et al.[17] indicated that the calculation time of a MC method is theoretically independent of the number of beams used due to the random sampling nature of MC. Thus, when the calculation involves a large number of beams, the MC method can be advantageous in computational efficiency over deterministic CCCS methods for dose calculation since there is no added cost in computation.

In order to build a MC-based day-to-day planning tool, the execution time of the MC methods still needs to be shortened substantially. Toward this goal, we use the massively parallel graphics processing unit (GPU) to accelerate the dose computation because GPU has been proved to be an effective approach in the domain of high performance scientific computation.[18] Using GPU as a hardware accelerator has attracted a lot of attention recently; several research groups have studied GPU implementation for dose calculation. For example, Hissoiny et al.[19] implemented a convolution/superposition algorithm on a GPU and a speedup of 10–20× over single-threaded CPU implementation was reported on an Nvidia 8800GTX card. Jacques et al.[20] demonstrated an impressive speedup of 34–98× based on a multiresolution approach which introduces less than 1% accuracy loss. Greef et al.[21] investigated the possibility of using GPU to accelerate the raytracing procedure which is a key component in many dose calculation algorithms.[22] The reported speedup is also based on a single-threaded CPU implementation and is in the range of 2.1–10.1×. Although the above works gained very good speedup, they all focus on deterministic dose calculation methods. As mentioned by Hissoiny et al., MC-based dose calculation may encounter many difficulties in the process of being ported onto GPU, but this issue was not discussed further and analyzed. We are aware of one effort which implemented an EGS4-based MC algorithm on a GPU and achieved a speedup of 10–30× over a single-threaded CPU implementation.[22] Unfortunately, that work was reported in a short paper and many details are not revealed. A recent effort using a GPU only managed to achieve a less than 5.5× speedup as compared to their software implementation.[23]

In this paper, we present a GPU-based implementation for the Monte Carlo convolution/superposition (MCCS) algorithm. The MCCS algorithm combines the strengths of the deterministic methods and the MC methods. Its speed advantage over traditional MC methods and accuracy advantage over traditional CS algorithms make itself a suitable candidate for the purpose of constructing a MC-based routine planning tool, although the MCCS algorithm still has certain accuracy limitations on handling heterogeneous cases, such as those often occurring in the lung and bone cases. Due to the conflicts between the random nature of the MC methods and the GPU architecture, implementing the MCCS algorithm efficiently is a highly nontrivial task and only small improvements could be obtained if the conflicts were not carefully resolved. In our work, we have analyzed the memory access patterns methodically and introduced a number of modifications to the original MCCS algorithm in order to maximally utilize the limited memory bandwidth available on the GPU. Our final implementation achieves a speedup in the range of 6.7–11.4× over a multithreaded implementation on a quad-core system (25–40× over a single-threaded implementation[24]) and offers solid agreement in terms of accuracy. This implementation leads to the completion time of a typical clinical case to be within 3 min.

The rest of this paper is organized as follows. In Sec. 2, we briefly review the MCCS algorithm (software) and the GPU architecture (hardware). We then present our CPU-based and GPU-based implementations in Sec. 3 The clinical cases are used to evaluate the accuracy and performance of our GPU implementation and are discussed in Sec. 4 Section 5 concludes the paper.

BACKGROUNDS

Go to:

Our work is to implement the MCCS algorithm on a GPU platform. The characteristics of both algorithm and platform play a crucial role in determining the final speedup result. In this section, we briefly introduce these two components and focus especially on those characteristics related to our design decisions. Readers are recommended to refer to the original paper for more details on the MCCS algorithm and the white paper[25] from Nvidia Corp. for more information on the GPU architecture.

Algorithm background

The MCCS algorithm calculates dose by combining the stochastic nature of the Monte Carlo simulation with deterministic kernel superposition. The Monte Carlo method is used to transport particles from a modeled primary and scatter source through the accelerator head to points in the patient where the first interaction happens. The location and the type of interaction, as well as the amount of energy transfer, are randomly determined as with the classic Monte Carlo method. Rather than continuing to follow the secondary particles to figure out their energy transfers (by secondary photons) and depositions (by recoiled electrons) as with the classic Monte Carlo methods, the transferred energy from the first event is spread around the point of interaction using the precalculated dose-spread-arrays just as in convolution/superposition algorithms. Computationally, it consists of four steps as described below (see Fig. 1).

(1) Pick a particle from the sources: Particles are randomly picked from the focal and extrafocal sources, modeled with the dual source model.[26, 27] For each particle, its energy is randomly sampled from the modeled energy spectrum of the beam and its direction is randomly sampled from a continuous angular distribution modeled to produce the measured radiation fluence profile. The extrafocal source is characterized as a planar source with a radial intensity distribution $f(r')$, which is derived from the measured in-air output factors. The probability of a photon being emitted from point (r',φ') is governed by the following independent possibilities:

$$P(r') = f(r')r'dr', \tag{1}$$

$$P(\varphi') = \frac{d\varphi'}{2\pi}. \quad (2)$$

The energy of this photon is determined by the Compton scattering of the focal photon with energy $h\nu$ and the inner product of the unit vectors e_1 and e_2, where the angular variation of the spectrum is modeled according to the suggestion by Starkschall et al.[28]

(2) Transport the particle to patient surface: The selected particle is transported through the head of the linear accelerator structures, such as the multileaf collimator (MLC) and wedge, to the patient surface using raytracing. Therefore, the effects of the collimating structures and beam modifiers, such as tongue and groove effect, beam hardening, and the effects of MLC leaf heights, are explicitly taken into consideration.

(3) Determine the energy transfer event: From the patient surface, the distance which the particle traverses before the first interaction (along the particle trajectory) is randomly sampled to determine the site of energy transfer. The radiological path length between the surface and the site of interaction is determined by following the possibility:

$$\lambda = \int \rho(\Lambda) d\Lambda = -\frac{1}{\mu} \log_e(\xi), \quad (3)$$

where ξ is a random number between 0 and 1, μ is the attenuation coefficient, and Λ is the geometrical path length.

(4) Energy spread to volumes around the interaction point: Rather than depositing the energy transfer to the medium from a single primary particle at the interaction point, as in the classic Monte Carlo simulation, the MCCS algorithm considers the first interaction by a large number of particles of the same energy and direction and is referred to as particle splitting in the MCCS algorithm context. The dose spread in a homogeneous water phantom can then be represented by a dose-spread-array precalculated with Monte Carlo simulation from monoenergetic photons. From each interaction site, multiple deposition directions are randomly sampled according to an even angular distribution. Energy is deposited to each voxel traversed along these directions in a collapsed-cone fashion. Namely, the energy deposited into a Cartesian voxel located at (r,θ) from the kernel origin by a particle with photon energy $h\nu$ is

$$\Delta E = \int\int\int_V \rho G(h\nu, \theta, r) r^2 \sin\theta dr d\theta d\varphi, \quad (4)$$

where $G(h\nu,\theta,r)$ is the expected dose at (r,θ) resulting from a single primary photon interaction in an infinite water medium, θ,φ are angles from the polar axes, and ρ is the density.

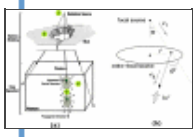

Figure 1

(a) The MCCS algorithm. The algorithm includes two sequential process, where machine modeling takes the geometry variations into account through a raytracing process and the dose deposition adopts the CS approach. (b) Head scatter modeling of the MCCS ...

The combination of MC random sampling of individual particles and the point kernel superposition allows more precise transport of primary particles as compared to the CS method but without the lengthy secondary particle transport as in Monte Carlo simulation. Unlike the traditional CS methods in which the total energy released per unit mass (TERMA) is precomputed for each voxel before the dose deposition, the MCCS method

replaces the TERMA by random sampling of photon energy, direction, and interaction points. The random weighted sampling of the particle energy makes spectral corrections unnecessary. (It is worth mentioning that the randomness also makes the MCCS algorithm a forward dose calculation method only and cannot perform calculation from the deposition point of view.) Moreover, the MCCS's ability to model all aperture-specific effects makes it capable of modeling sharp fluence changes due to geometry variation, such as those due to the MLC tongue and groove effect and MLC leaf heights. In addition, by sampling beam angles and field segments according to their relative weights, the MCCS method is proved to be able to efficiently calculate a multisegment/multibeam IMRT plan.[17] However, other than above advantages, these random sampling procedures also present some unique challenges to GPU acceleration of the MCCS algorithm.

GPU architecture and Compute Unified Device Architecture programming model

Physically, one GPU may contain more than hundreds of lightweight processors which are able to accomplish computation jobs simultaneously. To achieve better silicon area utilization, a total of M×N processors are organized in a hierarchical manner, in which each set of M cores forms a multiprocessor, leading to N multiprocessors in total as shown in Fig. 2.[25]

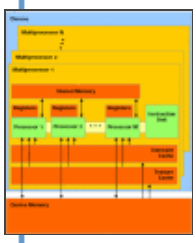

Figure 2
GPU hardware architecture (Ref. [25]).

Each core in a multiprocessor has its own register file and arithmetic logic unit which allows it to accomplish a specific computation job (referred to as a thread in the CUDA context). All cores in one multiprocessor share one shared memory unit which offers high bandwidth and low memory access latency. However, the amount of available shared memory is highly limited. Meanwhile, the M cores in a multiprocessor only achieve the maximum performance if they all follow the same instruction sequence. If execution divergence occurs due to conditional expressions such as "if" or "switch" clauses, some cores must become inactive. This architecture is often referred to as the single instruction multithread (SIMT) architecture. In the worst case, if M cores are assigned M different execution paths, then it leads to a completely sequential execution. With the added synchronization cost, it is even worse than feeding all M execution sequences into only one core. The resource constraints play an important role in the process of deploying an algorithm into GPU and violating such constraints could eventually lead to a design failure.

At a higher level, there are multiple other types of memories including global, constant, and texture memories which are provided to the multiprocessors. Constant and texture memories reside on the chip and are cached. They provide relatively high bandwidth, but their sizes are limited and can only be used as read-only memory in most scenarios. The most plentiful memory, global memory, is off-chip and not cached. In order to effectively utilize the off-chip memory bandwidth, the GPU memory controller is designed to buffer, reorder, and then coalesce large numbers of memory requests.[29, 30] For example, it is able to perform a 128-byte memory access through one transaction as long as the addresses lie in the same segment. Depending on the access patterns, the access latency for global memory could range from one clock cycle to several hundreds of clock cycles. Thus, an inefficient memory management strategy may eventually offset the possible speedup due to the computation power of massively parallel processors. Programmers can improve the performance by feeding the memory controller with a sequence with better data locality.

Corresponding to the underlying hardware, the CUDA program model takes advantage of the thread grids which contain a large number of threads. These threads are scheduled to run on multiprocessors in groups of 32 threads each (called a warp) by a hardware thread scheduler, which is transparent to the programmer. Only by overlapping computation and memory accesses can the GPU hide the long latency from accessing

the global memory.

Challenges in mapping the MC algorithm onto the GPU architecture

Without investigating the GPU architecture, one might come to a quick conclusion that a MC method is a perfect target application for GPU acceleration since GPU possesses enormous computation power and allows many threads to run in parallel; a MC algorithm, on the other hand, is computationally intensive and the particles in the MC algorithm propagate and interact independently.

However, this is only one side of the story. In fact, some GPU architecture features are in direct conflict with the stochastic nature of the MC algorithm.

First of all, the stochastic nature of the MC algorithm makes the position, trajectory, life-time, and many other aspects of a particle not predictable. This means that the execution path (for example, when to annihilate and where to exit the phantom) and the memory location to be accessed (for example, while performing the raytracing and fetching parameters) are determined in a random manner and lack of data locality. Yet, the performance of a contemporary processor (especially GPU) heavily depends on data locality. How to enhance data access locality to improve the performance has been an active research area in computer science for many years. Cache, instruction predication, and out-of-order execution are some successful examples in practice. However, the random feature of any MC method destroys the assumptions of these approaches and makes them inefficient. As pointed out in the CUDA programming guide, a common practice of using GPU is to use it as a coprocessor: (1) Transfer the data from the global memory to the shared memory; (2) process the data; and (3) transfer the result back to the global memory. However, for MC methods, it is not uncommon that only a few words are usable when a block of data is transferred from the off-chip global memory into the on-chip shared memory.

Second, although millions of particles travel and behave without interfering with one another, they do share the same phantom in the 3D space. Due to the large number of voxels that a phantom may have, information such as density has to be stored in the off-chip global memory. Also, if particles are spreading and traveling in a truly random manner, the possibility of accessing each voxel is equal and the average memory access latency could become very high.

Last but not least, the essence of GPU is to hide the memory access behind the computation and thus, the ratio between the computation and memory access is crucial to the final performance result. In many Monte Carlo models for the dose calculation problem, the particles' behavior is intended to be simple and does not involve heavy work load. Therefore, the MC methods are of a memory intensive nature (instead of computation intensive), which further underscores the needs for intelligent arrangement of memory accesses.

IMPLEMENTATIONS

Go to:

In this section, we first describe our GPU-based implementation in detail. We then briefly discuss a multithreaded CPU implementation which will be used as the basis for evaluating the GPU implementation.

GPU implementation

In order to accelerate an application, it is crucial to identify the performance bottleneck first. Prior to the actual implementation of the GPU code, we conducted a profiling of the MCCS code and the result indicated that the MCCS program is relatively memory intensive. For a batch with 67 200 histories, the total memory accesses (including all kernels, density, and geometry information such as MLC positions etc., but without local variables) are 1.32×10^9 on average (see Table 3) and the running time on a 2.4 GHz CPU (single-threaded mode) is 31 s on average. Even if we assume the ideal situation where every memory access can be accomplished in one clock cycle (800 MHz), the ratio between computation and memory access is still low.

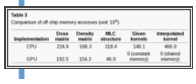

Table 3

Comparison of off-chip memory accesses (unit: 10^6).

Our focus of the GPU implementation, therefore, is on organizing judiciously the memory accesses of the MCCS algorithm to increase both the temporal and spatial data locality with respect to the GPU structure. To achieve the largest possible data locality, we introduced a number of modifications to the original MCCS algorithm. The modified pseudocode of the GPU-based MCCS method is shown in Algorithms 1 and 2. Below, we discussed these modifications in detail.

Stage-based processing Unlike the CPU implementation in which each parallel thread tracks one photon from generation to annihilation, we calculate dose in two stages explicitly: Machine modeling and dose deposition (as shown in Fig. 1). Once the particle reaches the phantom, it "stops" and all information related to it, including the coordinates of interaction points, trajectory direction, and kernel, etc., is extracted and saved into a buffer in the global memory. Once the number of particles in the buffer reaches a preset threshold, the GPU switches to the dose deposition stage and these stored particles are resumed to deliver radiation energy.

The reason for us to take this two-stage approach is that the GPU-based system is normally low on cache and lacks virtual memory mechanism. If each thread in GPU is responsible for the whole history of one photon, then the GPU would not be able to hold a sufficiently large number of threads to hide the memory access latency because the required register number and on-chip memory size for each thread would have to increase correspondingly in order to keep track of the status of each particle. Thus, a tradeoff must be made to attain a balance between the design simplicity and system performance.

Double buffer for interpolation Interpolation is used heavily in the MCCS algorithm to improve speed and accuracy. The most frequently invoked one is to deposit dose in each voxel penetrated by the kernel direction. Thus, the speed of this operation is critical to the overall performance. In our practical situations, the cell size of the kernels is larger than that of the phantom. This means that dose deposition into any voxel involves only two adjacent cells in the kernel if linear interpolation is used. To utilize this feature, a double buffer is created in the shared memory region and the next possible data in the kernel is always prefetched (see line 10 of Algorithm 2).

Data duplication Raytracing is a procedure for determining the radiological length in the phantom along the particle's trajectory direction. As indicated in Fig. 1, it is one of the fundamental procedures in the process of dose calculation and its speed has a huge impact on the overall performance. Our raytracing program is based on the 3D DDA algorithm proposed by Amanatides and Woo[31] which has been shown to be very efficient for voxel transversal in uniform grids. Regardless of acceleration approaches, the raytracing function of a particle starting at (x_0,y_0,z_0) with trajectory direction (α,β,γ) can be modeled by the equations below

$$\begin{cases} x_{m+1} = x_m + \alpha^* t_m \\ y_{m+1} = y_m + \beta^* t_m \\ z_{m+1} = z_m + \gamma^* t_m \\ t_m = \min\left(\dfrac{(m+1)^*\Delta x - x_m}{|\alpha|}, \dfrac{(m+1)^*\Delta y - y_m}{|\beta|}, \dfrac{(m+1)^*\Delta z - z_m}{|\gamma|} \right) \\ \Lambda_m = \sqrt{(x_{m+1}-x_m)^2 - (y_{m+1}-y_m)^2 - (z_{m+1}-z_m)^2} \end{cases}, \quad (5)$$

where (x_m,y_m,z_m) and $(x_{m+1},y_{m+1},z_{m+1})$ denote the entering/exiting points of voxel V.

The above equations reveal the fact that the process of determining the geometrical path length Λ_m captures a simple linear relationship and is very suitable for GPU implementation since they involve only pure computation. However, the purpose of raytracing is normally to derive the radiological length $\lambda_m = \sum_{m=0}^{M} \rho(i_m, j_m, k_m)^* \Lambda_m$, where $\rho(i_m,j_m,k_m)$ is density for the voxel and is continuously read from the

memory. Although any two voxels along the raytracing direction are neighbors in the phantom, they are normally not adjacent to each other in the memory (unless the raytracing direction happens to be the direction of converting a 3D matrix into a 1D matrix) because all memories are logically 1D tables. This makes the coalescing of global memory accesses hard. (As also observed by Hissoiny et al.,[19] the GPU implementation of convolution/superposition is normally memory-bounded and we believe this is the underlying reason.) One may also notice that the raytracing operation as shown in the above equations is not computationally intensive and thus leads to a low computation vs memory access ratio. Given that our application is memory bandwidth constrained (instead of CPU execution constrained), any improvement on the memory bandwidth utilization would benefit the overall performance.

In this case, our strategy for improving the data locality is to reorganize the memory layout to match the raytracing memory access pattern according to the signs of the raytracing direction (α,β,γ). Since the vector (sign(α),sign(β),sign(γ)) divides the 3D space into eight quadrants, we make eight corresponding copies (ρ_0,ρ_1,...,ρ_7) of the density volume $\rho(i,j,k)$, which are represented as 16-bit floating point numbers. For a given ray direction falling in one of the eight quadrants, a voxel and its seven neighboring voxels are organized as one group and stored continuously in the memory as shown in Fig. 3. In this way, it is guaranteed that we can always find the neighbors of a voxel within a distance less than 8. Since the density volume is represented as 16-bit floating numbers and the GPU memory controller supports 128-byte native global memory transaction, the proposed memory layout allows at least two memory reads to fall into one global memory read transaction. In theory, this could at least double the memory read efficiency for the raytracing process if no memory controller is involved. However, the actual speedup improved by data duplication is measured to be only around 15%, which is similar to the improvement by Stone et al.[32] This means that the memory controller works quite efficiently because the 15% improvement also includes the performance benefit from cache due to better data locality. In the meantime, this performance improvement is at the cost of using eight times of memory space. To balance the performance and memory requirement, we opt to apply this data duplication strategy only for the raytracing before the particle is split.

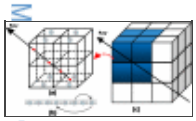

Figure 3

Memory layout optimization for the raytracing procedure. Central voxel (voxel 6) and its seven neighbors from a 3D grid (c) are extracted and saved as one group according to the beam direction (a). For the given direction, the ray penetrates voxels 6, **...**

Photon grouping By default, the photons are generated in an arbitrary order and the associated kernel is interpolated from two “close” members in a given set of kernels. The total size of these given kernels is large and cannot be stored entirely in any of the on-chip memories. Given how frequently the interpolation is invoked and the latency of the off-chip memory, leaving kernels in the global memory space is a high price to pay in terms of performance.

However, it is possible to put one or two kernels in the constant memory of the GPU. Hence, we group together the photons using the same kernel and perform dose deposition group by group. The disadvantage of this grouping strategy is that it is an additional process and extra memory space is needed to keep the photons that have not been processed. Nonetheless, this overhead is acceptable since it eliminates a lot of off-chip memory accesses and substantially improves the overall performance.

Behavior modification and cache usage To improve data locality, additional modifications to the MCCS algorithm are also made. For example, instead of generating a photon and assigning a beam number according to the MU weight, our implementation first determines how many photons are needed for each beam in order to create the desired dose distribution and then generates the required number of photons. Furthermore, we use the GPU texture cache as a general-purpose cache to store the density matrix. However, the dose matrix is not stored in the texture cache since the texture cache acts like a read-only cache and

cannot adequately support random memory writes.

Random number generator As an underlying engine, the random number generator (RNG) plays a crucial role in MC algorithms. Besides its effects on the distribution, performance, and period, a highly desired feature in our implementation is the minimum register usage, since the total number of registers used by each thread eventually determines how many threads can be run in parallel. Instead of adopting the RNG in the publicly available Mersenne Twister implementation from the Nvidia's Software Development Kit, a parallel version of a 48-bit linear congruent random number generator was implemented, which is based on the Rand48( ) function from the standard Unix implementation. As a result, only two extra registers are needed by each thread. Thus, each thread of our GPU implementation can have its own RNG with a different seed. This distributed RNG is also helpful in reducing the synchronization cost as there is no centralized RNG or random number buffer for each thread to obtain a usable random number.

Execution divergence control To reduce the negative impact of execution divergence, several techniques are applied to make all threads follow a uniform execution path as much as possible. For example, a conceptual virtual phantom with zero density is created surrounding the actual phantom. The dose deposition process for one photon is terminated once all rays derived from it enter the virtual phantom. Also, in the process of coding, based on the same principle, mathematical operations are more preferred than logical operations provided that there is no additional cost.

With the above modifications to the MCCS algorithm, we implemented the new MCCS algorithm in the CUDA environment. Speedup in the range of 6.7–11.4× over software implementation on the quad-core system can be achieved as shown in Sec. 3B.

Multithreaded CPU implementation

To help evaluate the performance of the GPU implementation, instead of using the original single-threaded implementation, we implemented a multithreaded version of the MCCS algorithm. This provides a more fair comparison as many desktop computers are equipped with multicore (such as dual-core or quad-core) processors nowadays and can benefit from multithreading. Our CPU implementation faithfully follows the steps shown in Sec. 2. Each thread keeps track of the whole life span of one particle. To confirm that our multithreaded implementation (on a quad-core system) does not waste the computation power, a running time comparison with the single-threaded version was done and a speedup between 3.72–3.86× was observed. If the program initialization and finalization time was taken into consideration, the speedup was in the range of 3.1–3.4× since they add overhead that cannot be accelerated through the multithreading process. This ratio is fairly good when the synchronization overhead among threads is taken into consideration. For example, a frequently encountered scenario is to avoid simultaneous writing to one location by multiple threads. In this case, an atomic operation such as mutex or compare-and-swap is needed.

RESULTS

Go to:

In this section, we used a specific phantom setup to evaluate the GPU implementation and compared it to the multithreaded implementation running on a PC. Two lung cases were also applied to test the performance of our approach.

Evaluation setup

Our CPU implementations are all pure C programs compiled in Visual Studio 2008 and run on a quad-core system with the Windows Vista 64-bit operating system. The single-threaded version has been used as a quality verification tool at the University of Maryland for multiple years. The GPU implementation is developed on the same PC with a GTX260-based graphics card (216 cores). (All GTX200 series GPUs adopt the same architecture and Nvidia documents state that the architecture will remain unchanged for the next several generations to support CUDA. The speedup presented here can be scaled according to the memory

bandwidth of other GPUs since the application is memory-bounded on GPU.) For our execution configuration, all threads are partitioned into 256 thread blocks, each containing 256 threads. Thus, one CUDA kernel of Nvidia has a total of 65 536 threads. Since the application performance heavily depends on the hardware specification, we summarize the important features of our platforms in Table 1.

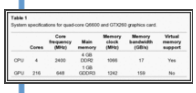

Table 1
System specifications for quad-core Q6600 and GTX260 graphics card.

To verify the accuracy and performance, we used a phantom setup with seven beams to thoroughly evaluate our design by collecting run time profiling information. In addition, we applied our implementation to two clinical lung cases to perform further evaluation. To show the relationship between performance and the number of histories, different numbers of batches were used (one batch contains 672 000 photon histories). The setups of these cases are summarized in Table 2.

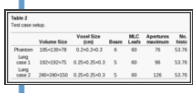

Table 2
Test case setup.

Performance evaluation

As discussed in Sec. 3, one of our goals is to reduce the number of off-chip memory accesses since this is crucial for the performance gain. In order to verify that the memory system is indeed the performance bottleneck, we intentionally adjusted the computation time of core computation tasks by inserting delay instructions. These delay functions are simply some meaningless codes and are designed not to be optimized away by the complier. In the meantime, the memory accesses remain unchanged. We observed that the measured running time did not increase if the delay is small. This confirms our conclusion that the target application is memory-bounded.

To see whether our implementation indeed helps improve the memory efficiency, before we measured the running time, we profiled the total number of memory accesses for different types of information and accumulated them. The results of five major components are shown in Table 3, where all kernel accesses (46.3% of the total memory accesses in CPU implementation) have been converted into on-chip memory operations by applying the techniques discussed in Sec. 3, and they in turn led to performance improvement.

In this paper, all computation times are measured between the end of the initialization (including system initialization and data transfer to/from the main memory) and the beginning of saving dose results into a file because these operations are unavoidable overhead and execute only once. To evaluate the effects of the number of histories on the speedup, the GPU execution time for the phantom case is measured from 10 to 80 batches and the average result for one batch varies from 792 to 1204 ms. Our measurement shows that a batch of histories takes 8.91 s (on average) to run on the quad-core CPU. Hence, the GPU implementation demonstrates a speedup of 6.7–11.4× over the CPU-based implementation (see Fig. 4). When compared to a single-threaded implementation, the speedup is in the range between 25.7 and 39.1×. As the number of batches becomes smaller, the speedup decreases because the data transfer time dominates the total execution time, although our implementation performs the computation asynchronously with the data transferring. Increasing the number of batches beyond a certain point leads to gradual decrease of the speedup. For the performance degradation for a large number of batches, we have not identified the critical reason yet since many factors may contribute to it, for example, synchronization cost, communication overhead, etc. As shown in Table 4, the execution time on GPU for the two lung cases (53.76×10^6 particles) was 81 and 174 s, respectively. The measured time on PC for the same cases was 810 and 1635 s. This indicates a speedup of 10 and 9.4×.

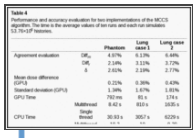

Table 4

Performance and accuracy evaluation for two implementations of the MCCS algorithm. The time is the average values of ten runs and each run simulates 53.76×10^6 histories.

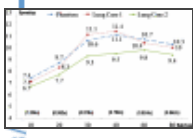

Figure 4

Sample dose distribution of lung cases. The left side is the 98th slice along the Y direction (XZ plane) from lung case 1. The right side is also on the XZ plane, but is the 125th slice (240 in total) from lung case 2.

Accuracy evaluation

In this section, two questions are addressed: (i) How accurate the MCCS algorithm is and (ii) how well the two implementations agree with each other.

The uncertainty of dose volume from the GPU implementation is evaluated through the mean dose difference and the standard deviation, which are derived based on the results collected from ten runs. As shown in Table 4, the standard deviations for dose at the isocenter are between 1.34% and 1.81%, which are acceptable statistical fluctuations for clinical usage. The mean dose difference is measured to be 0.21%–0.43%.

We also used several statistical measurements to evaluate the closeness of the resulting dose data. Let $D_{\mathrm{CPU}}(x,y,z)$ and $D_{\mathrm{GPU}}(x,y,z)$ represent the dose results at voxel (x,y,z) from the two implementations, respectively. We calculate the following values in the region with more than 20% of the maximum dose (since only high dose regions are of interest to clinical applications):

- Maximum voxel difference

$$\mathrm{Diff}_m = \max\left(\frac{|D_{\mathrm{CPU}}(x,y,z) - D_{\mathrm{GPU}}(x,y,z)|}{D_{\mathrm{CPU}}(x,y,z)}\right). \tag{6}$$

  This value is used to measure the voxel-to-voxel difference.
- Maximum relative voxel difference

$$\mathrm{Diff}_r = \frac{\max(|D_{\mathrm{CPU}}(x,y,z) - D_{\mathrm{GPU}}(x,y,z)|)}{\max(D_{\mathrm{CPU}}(x,y,z))}. \tag{7}$$

  This value shows the closeness between the dose results from the two implementations relative to the maximum dose in the CPU implementation.
- Average voxel difference

$$\delta = \frac{\sum|D_{\mathrm{CPU}}(x,y,z) - D_{\mathrm{GPU}}(x,y,z) \,/\, D_{\mathrm{CPU}}(x,y,z)|)}{N}, \tag{8}$$

  where N is the total number of voxels in the volume of interest. This value reflects the overall agreement between the two implementations.

The analysis results for the three test cases are summarized in Table 4 and each of them uses the maximum number of histories as shown in Table 2 to produce a stable result.

To further investigate the accuracy issue, we also implemented one version of the GPU code which contains the same RNG as the CPU code and simulated it with only one thread active. The results agree with each other except that some rounding errors are found. Thus, the disagreement between the CPU results and GPU results is largely due to the fact that the two approaches adopt different RNGs. The effect of variant RNGs on MCCS algorithm itself was also studied by Chen[33] and a 3.28% $Diff_r$ was reported. To illustrate the agreement more intuitively, we show the overlapped isodose curves of the center transversal slice of the phantom case in Fig. 5. The dose result for the center slice of lung case 2 is also shown in Fig. 6.

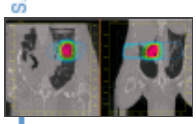

Figure 5

Speedup of GPU implementation over a multithreaded CPU implementation on a quad-core PC with 2.4 GHz working frequency. The speedup shall be scaled accordingly when compared to single-threaded implementation. The time in brackets is the average execution ...

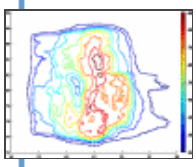

Figure 6

Overlapped isodose curves for central transversal slice from phantom test case. The X and Y axes represent the voxel indexes near the radiated volume.

CONCLUSION

Go to:

In this paper, we present a massively parallel version of a MCCS algorithm for dose calculation which is implemented on a GPU-based system. The resulting speedup over a multithreaded CPU implementation is between 6.7 and 11.4×, which makes our approach possible to produce a precise dose result based on the MCCS algorithm within 3 min. This indicates that GPU-based implementation of MC methods can be feasible and cost-efficient for satisfying the increasing demands for either computation speed or accuracy by advanced radiation therapy technologies. Given the important role of dose calculation in the treatment planning process, such improvements can eventually boost the treatment quality. Implementing a MC-based method on GPU presents nontrivial challenges since it requires careful tune-up on the algorithm side to match the SIMT programming model and architectural constraints of GPU. Our initial version of the GPU code was actually discouraging, yet the performance has been improved substantially after the techniques in Sec. 3 were applied.

Even though the speedup of our GPU-based approach for the MCCS method is quite substantial, it is still not as impressive as some other types of applications.[9] The fact that other related work on GPU-based acceleration for Monte Carlo algorithms has reported similar speedups as ours confirmed our observation on the inherent limitations of GPU-based implementations for Monte Carlo based algorithms.[12] The random nature of these algorithms makes it difficult to achieve data locality, which causes many advanced performance-improving features of GPU to become less efficient or even invalid. To obtain even better speedup, we plan to investigate other approaches to accelerate the Monte Carlo type of algorithms, for example, the field-programmable gate array based design which, similar to a dedicated integrated circuit chip, provides much more flexibility in terms of memory structures, but at a lower cost than such circuit chips.

Algorithm 1 GPU-based MCCS Algorithm (Before Entering Phantom)

1: initialize memories and RNGs,
2: i=0,

$$N_{req} = \frac{Total\ \ Number\ \ of\ \ Particles}{Number\ \ of\ \ Threads}$$

3: while $i<N_{req}$ do
4: randomly generate a particle trajectory direction *dir*, $i=i+1$
5: if $dir \in allowedrange$ then
6: while not entering phantom do
7: raytracing one step further
8: if still in the boundary then
9: attenuate and modify other parameters
10: else
11: terminate this particle and goto line 3
12: end if
13: end while
14: randomly assign an energy level to this particle
15: grouping this particle according to its information
16: save the information into allocated buffer
17: else
18: terminate this particle and goto line 3
19: end if
20: end while

Algorithm 2 GPU-based MCCS Algorithm (After Entering Phantom)

1: retrieve one particle from the buffer
2: move the corresponding kernel into shared memory if necessary
3: generate particle-specific kernel through interpolation
4: while inside the phantom do
5: split the particle along the *dir*
6: if position is in the phantom then
7: for i=1 TO 6 STEP 1 **do**
8: while particle hi phantom do
9: raytrace one step further
10: pre-fetch next kernel data
11: perform linear interpolation on dose
12: accumulate dose result
13: end while
14: end for
15: else
16: terminate this particle and goto line 1
17: end if

18: end while

ACKNOWLEDGMENTS

Go to:

The research of Danny Chen was supported in part by NIH Grant No. R01-CA117997-01A2 and NSF Grant Nos. CCF-0515203 and CCF-0916606.

References

Go to:

1. Wang C., Luan S., Tang G., Chen D. Z., Earl M., and Yu C. X., "Arc-modulated radiation therapy (AMRT): A single-arc form of intensity-modulated arc therapy," Phys. Med. Biol. 53, 6291–6303 (2008).10.1088/0031-9155/53/22/002 [PubMed] [Cross Ref]
2. Shepard D. M., Cao D., Afghan M. K. N., and Earl M. A., "An arc-sequencing algorithm for intensity modulated arc therapy," Med. Phys. 34, 464–470 (2007).10.1118/1.2409239 [PubMed] [Cross Ref]
3. Cameron C., "Sweeping-window arc therapy: An implementation of rotational IMRT with automatic beam-weight calculation," Phys. Med. Biol. 50, 4317–4336 (2005).10.1088/0031-9155/50/18/006 [PubMed] [Cross Ref]
4. Mohan R., Zhang X., Wang H., Kang Y., Wang X., Liu H., Ang K., Kuban D., and Dong L., "Use of deformed intensity distributions for online modification of image-guided IMRT to account for interfractional anatomic changes," Int. J. Radiat. Oncol., Biol., Phys. 61(4), 1258–1266 (2005).10.1016/j.ijrobp.2004.11.033 [PubMed] [Cross Ref]
5. Feng Y., Castro-Pareja C., Shekhar R., and Yu C., "Direct aperture deformation—An inter-fraction image guidance strategy," Med. Phys. 33(12), 4490–4498 (2006).10.1118/1.2374675 [PubMed] [Cross Ref]
6. Ahunbay E., Chen G., Peng C., Yu C., Narayanan S., and Li X. A., "An on-line adaptive planning strategy for inter-fraction image guidance," Int. J. Radiat. Oncol., Biol., Phys. 69(3), S23–S23 (2007).10.1016/j.ijrobp.2007.07.040 [Cross Ref]
7. Boyer A. L. and Mok E. C., "Photon beam modeling using Fourier transform techniques," in Proceedings of the Eighth International Conference on the Use of Computers in Radiation Therapy, Toronto, Canada, 1984, pp. 14–16.
8. Mohan R., Chui C., and Lidofsky L., "Differential pencil beam dose computation model for photons," Med. Phys. 13(1), 64–73 (1986).10.1118/1.595924 [PubMed] [Cross Ref]
9. Mackie T. R. and Scrimger J. W., "Computing radiation dose for high energy x-rays using a convolution method," in Proceedings of the Eighth International Conference on the Use of Computers in Radiation Therapy, Toronto, Canada, 1984, pp. 36–40.
10. Papanikolaou N., Mackie T. R., Meger-Wells C., Gehring M., and Reckwerdt P., "Investigation of the convolution method for polyenergetic spectra," Med. Phys. 20, 1327–1336 (1993).10.1118/1.597154 [PubMed] [Cross Ref]
11. Mackie T. R., Bielajew A. F., Rogers D. W. O., and Battista J. J., "Generation of photon energy deposition kernels using the EGS Monte Carlo code," Phys. Med. Biol. 33, 1–20 (1988).10.1088/0031-9155/33/1/001 [PubMed] [Cross Ref]
12. Boyer A. L., "Relationship between attenuation coefficients and dose-spread kernels," Radiat. Res. 113, 235–242 (1988).10.2307/3577199 [PubMed] [Cross Ref]
13. Ahnesjö A., "Collapsed cone convolution of radiant energy for photon dose calculation in heterogeneous media," Med. Phys. 16, 577–592 (1989).10.1118/1.596360 [PubMed] [Cross Ref]
14. Reckwerdt P. J. and Mackie T. R., "Superposition/convolution speed improvements using run-length raytracing," Med. Phys. 19, 784 (1992).
15. McNutt T., "The ADAC Pinnacle3 collapsed cone convolution superposition dose model," See http://www.medical.philips.com.
16. Mackie T. R., Scrimger J. W., and Battista J. J., "A convolution method of calculating dose for 15-MV x-rays," Med. Phys. 12, 188–196 (1985).10.1118/1.595774 [PubMed] [Cross Ref]

17. Tang G., Earl M. A., Luan S., Wang C., Yu C. X., and Naqvi S. A., "Stochastic versus deterministic kernel-based superposition approaches for dose calculation of intensity modulated arcs," Phys. Med. Biol. 53(17), 4733–4746 (2008).10.1088/0031-9155/53/17/018 [PubMed] [Cross Ref]
18. Owens J. D., Luebke D., Govindaraju N., Harris M., Krüger J., Lefohn A. E., and Purcell T. J., "A survey of general-purpose computation on graphics hardware," Comput. Graph. Forum 26, 80–113 (2007).10.1111/j.1467-8659.2007.01012.x [Cross Ref]
19. Hissoiny S., Ozell B., and Després P., "Fast convolution-superposition dose calculation on graphics hardware," Med. Phys. 36, 1998–2005 (2009).10.1118/1.3120286 [PubMed] [Cross Ref]
20. Jacques R., Taylor R., Wong J., and McNutt T., "Towards real-time radiation therapy: GPU accelerated superposition/convolution," in Proceedings of the High-Performance Medical Image Computing and Computer Aided Intervention Workshop, 2008). [PubMed]
21. de Greef M., Crezee J., van Eijk J. C., Pool R., and Bel A., "Accelerated ray tracing for radiotherapy dose calculations on a GPU," Med. Phys. 36(9), 4095–4192 (2009).10.1118/1.3190156 [PubMed] [Cross Ref]
22. Jahnke L., Fleckenstein J., Clausen S., Hesser J., and Wenz F., "Speeding up of GEANT4 based Monte Carlo Simulations for radio therapy treatments on SIMD machines," in Proceedings of the American Society for Therapeutic Radiology and Oncology (ASTRO) 50th Annual Meeting, Boston, MA, 21–25 September 2008.
23. Jia X., Gu X., Sempau J., Choi D., Majumdar A., and Jiang S. B., "Development of a GPU-based Monte Carlo dose calculation code for coupled electron-photon transport," Phys. Med. Biol. 55(11), 3077–3089 (2010).10.1088/0031-9155/55/11/006 [PubMed] [Cross Ref]
24. Zhou B., Hu X. S., Chen D. Z., and Yu C. X., "A GPU-based implementation of Monte Carlo superposition for dose calculation," in Proceedings of the 51st AAPM Annual Meeting, Los Angles, CA, July 2009.
25. Nvidia Corp., NVIDIA CUDA Compute Unified Device Architecture Programming Guide, 2008.
26. Jaffray D. A., Battista J. J., and Munro P., "X-ray sources of medical linear accelerators: Focal and extra-focal radiation," Med. Phys. 20, 1417–1427 (1993).10.1118/1.597106 [PubMed] [Cross Ref]
27. Liu H., Mackie T. R., and McCullough E. C., "A dual source photon beam model used in convolution/superposition dose calculations for clinical megavoltage x-ray beams," Med. Phys. 24(12), 1960–1974 (1997).10.1118/1.598110 [PubMed] [Cross Ref]
28. Starkschall G., Steadham R. E., Popple R. A., Ahmad S., and Rosen I. I., "Beam-commissioning methodology for a three-dimensional convolution/superposition photon dose algorithm," J. Appl. Clin. Med. Phys. 1, 8–27 (2000).10.1120/1.308246 [PubMed] [Cross Ref]
29. Yuan G. L., "GPU compute memory systems," M.S. thesis of Applied Science, University of British Columbia, Canada, 2009.
30. Ryoo S., Rodrigues C. I., Baghsorkhi S. S., Stone S. S., Kirk D. B., and Hwu W. W., "Optimization principles and application performance evaluation of a multithreaded GPU using CUDA," in Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, Salt Lake City, UT, 2008, pp. 73–82.
31. Amanatides J. and Woo A., "A fast voxel traversal algorithm for ray tracing," Proceedings of Eurographics, 1987, pp. 3–10.
32. Stone S. S., Haldar J. P., Tsao S. C., Hwu W. W., Liang Z., and Sutton B. P., "Accelerating advanced MRI reconstructions on GPUs," in Proceedings of the 2008 Conference on Computing Frontiers, New York, NY, 2008, pp. 261–272.
33. Chen Y., "Using multi-FPGA platform to accelerate Monte Carlo superposition dose calculation," M.S. thesis, Department of Computer Science and Engineering, University of Notre Dame, USA, 2009.