Accelerated ray tracing for radiotherapy dose calculations on a GPU

M. de Greef,[a)] J. Crezee, J. C. van Eijk, R. Pool, and A. Bel
Department of Radiation Oncology, University of Amsterdam, Meibergdreef 9, 1105 AZ Amsterdam, The Netherlands

(Received 16 February 2009; revised 2 July 2009; accepted for publication 2 July 2009; published 12 August 2009)

Purpose: The graphical processing unit (GPU) on modern graphics cards offers the possibility of accelerating arithmetically intensive tasks. By splitting the work into a large number of independent jobs, order-of-magnitude speedups are reported. In this article, the possible speedup of PLATO's ray tracing algorithm for dose calculations using a GPU is investigated.
Methods: A GPU version of the ray tracing algorithm was implemented using NVIDIA's CUDA, which extends the standard C language with functionality to program graphics cards. The developed algorithm was compared based on the accuracy and speed to a multithreaded version of the PLATO ray tracing algorithm. This comparison was performed for three test geometries, a phantom and two radiotherapy planning CT datasets (a pelvic and a head-and-neck case). For each geometry, four different source positions were evaluated. In addition to this, for the head-and-neck case also a vertex field was evaluated.
Results: The GPU algorithm was proven to be more accurate than the PLATO algorithm by elimination of the look-up table for z indices that introduces discretization errors in the reference algorithm. Speedups for ray tracing were found to be in the range of 2.1–10.1, relative to the multithreaded PLATO algorithm running four threads. For dose calculations the speedup measured was in the range of 1.5–6.2. For the speedup of both the ray tracing and the dose calculation, a strong dependency on the tested geometry was found. This dependency is related to the fraction of air within the patient's bounding box resulting in idle threads.
Conclusions: With the use of a GPU, ray tracing for dose calculations can be performed accurately in considerably less time. Ray tracing was accelerated, on average, with a factor of 6 for the evaluated cases. Dose calculation for a single beam can typically be carried out in 0.6–0.9 s for clinically realistic datasets. These findings can be used in conventional planning to enable (nearly) real-time dose calculations. Also the importance for treatment optimization techniques is evident. [DOI: 10.1118/1.3190156]

Key words: radiotherapy, dose calculation, GPU, graphics card, treatment planning

I. INTRODUCTION

Ray tracing is an important and time consuming step in dose calculations for radiotherapy. For a 3D CT dataset, the relative electron density has to be line integrated from the source position to every volume element (voxel) in that dataset. Moreover, for radiotherapy treatment planning, this process must be repeated for each source position (gantry angle) to determine the total dose in each voxel.

The traditional way of designing a treatment plan consists of a manual determination of the optimal beam configuration iteratively. Every adjustment requires recalculation. Typically, calculating the contribution of a single beam to the dose distribution requires 15–60 s of the calculation time. If dose calculations could be performed real time, a major reduction in workload is achieved. Furthermore, optimization of intensity modulated radiotherapy (IMRT) plans with variable beam angles is a computationally intensive task. This applies especially to recent developments such as volumetric modulated arc therapy,[1] intensity modulated arc therapy,[2,3] and/or tomotherapy.[4,5] Speed improvement is also important in view of the growing amount of data acquired just prior or during treatment.[6] Image-guided radiotherapy can lead to adaptive procedures with possible replanning during the course of treatment.

Multicore processors have now become common. However, faster execution can only be achieved using multithreading, which requires rewriting of the algorithm. Moreover, the speedup is related to the number of cores, leading to a maximum speedup of four times in practice.

Another, rather expensive, option is using large CPU clusters. However, the associated overhead, which comes with network data transfer and thread scheduling, is large compared to the present execution time.

The ray tracing algorithm currently used in our department (as implemented in PLATO) is based on the algorithm proposed by Siddon.[7] This algorithm is widely used in various radiotherapy treatment planning systems. Since its publication, the Siddon algorithm has been optimized for dose calculations in PLATO.[8] However, there is still room for improvement for accurate (quasi) real-time user interaction and the advanced planning techniques mentioned earlier.

Since about 2000, the use of commodity graphics hardware for high-performance computing has become a subject of research for an increasing number of research groups in various fields. Designed for processing large amounts of data in parallel, the graphical processing unit (GPU) is used to speed up arithmetically intensive computational tasks.[9] This use of a the GPU outside its original context of computer graphics is referred to as general purpose computation on the GPU. The reported speedups are achieved by separating an arithmetically intensive task in a large number of lightweight tasks, threads, which operate on their own data. It is important that threads are not dependent on the (intermediate) results of other threads; this is referred to as data independence. Ray tracing for dose calculations is a good candidate for execution on the GPU. A large number of rays have to be traced and there is no dependency between these rays. The aim of this study is to investigate the possible acceleration of the presently used ray tracing algorithm for radiotherapy dose calculation using a GPU.

II. MATERIALS AND METHODS

II.A. General purpose computation on the GPU

Since about 2000, the field of general purpose computation on GPUs is growing. Modern graphics cards have become highly programable and are developing fast regarding their computational capacities.[10] Traditionally, toolkits for computer graphics such as OpenGL and DirectX were used to perform calculations outside of the computer graphics context. This approach is referred to as compute by drawing: A computational problem is translated into a graphical/visualization problem. This requires implementation of algorithms in a graphical programming language.

Apart from the compute-by-drawing strategy, there are a number of toolkits that allow programming on a more abstract level, closer to high-level programming languages. Examples of these languages are Brook,[11] RapidMind,[12] and CUDA (compute unified device architecture) by NVIDIA.[13] In this study the latter language was used. CUDA is an extension of standard C that is preprocessed by a compiler driver and compiled using a C compiler. CUDA uses a data-parallel programming model where a large number of parallel threads are organized in a two-dimensional grid. Within a grid, blocks are specified that serve as a unit in resource assignment (Fig. 1).

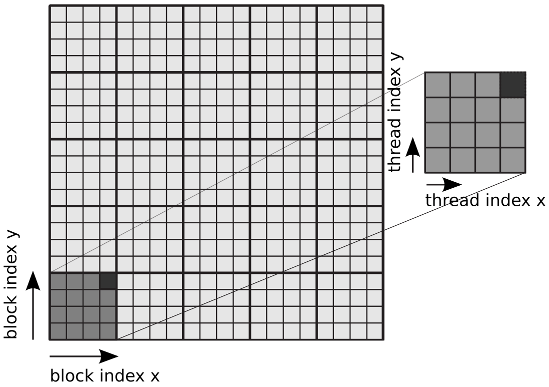


FIG. 1. The principle of a grid used to organize a number of parallel threads. A grid consists of a number of blocks that are identified by a block index. Within a block, the thread index identifies threads.

II.B. Siddon's algorithm for exact calculation of the radiological path

Since the GPU ray tracing algorithm for dose calculations will be based on the algorithm by Siddon,[7] the original algorithm is briefly summarized here. The algorithm is based on the representation of a 3D dataset as three sets of orthogonal planes. For a ray, traveling from the source to a point in the volume, the voxels through which the ray travels can be determined with the aid of these planes. The radiological path length (RPL) is found by summation of the length traveled by this ray in each voxel, multiplied by the relative electron density of the voxel. Figure 2 shows this procedure schematically. It should be noted that this method is different from methods that use numerical integration by sampling of the density along the ray trajectory. In contrast to the method used in this study, such a technique can show irregular behavior since the sampling distance has an impact on which voxels are taken into account and which ones are not.

The path of a ray is represented using the following parameterization:

$$x = x_1 + \alpha(x_2 - x_1), \tag{1}$$

$$y = y_1 + \alpha(y_2 - y_1), \tag{2}$$

$$z = z_1 + \alpha(z_2 - z_1), \tag{3}$$

where index 1 refers to the source position, index 2 to the center of the voxel the ray is traced to, and α is the distance traveled divided by the source-voxel distance. Now the dataset can be expressed as

$$X_i = X_0 + i\Delta X, \tag{4}$$

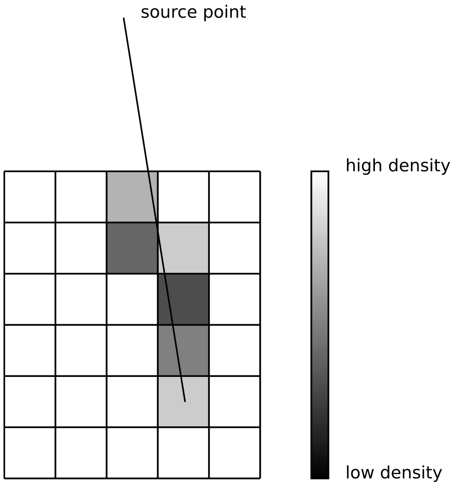


FIG. 2. Schematic 2D illustration of ray tracing as used in radiotherapy dose calculations. A ray is traced from the source to all the voxels in the dataset. The voxels a ray travels through on its way to a certain voxel in the dataset are collected so that the length of the fragments can be density weighted.

$$Y_j = Y_0 + j\Delta Y, \tag{5}$$

$$Z_k = \underline{z}_{\text{plane}}[k], \tag{6}$$

where the use of capital symbols refers to the fact that the coordinates indicate plane positions, ΔX and ΔY are the voxel dimensions in the x and y directions respectively, i, j, and k are plane indices, and X_0 and Y_0 are the x and y coordinates of the planes that are defined by i and j equal to zero, respectively. In contrast to the original algorithm, the dataset is allowed to be nonequidistant in the z direction. The reason for this is that CT scans for radiotherapy treatment planning may be acquired with varying slice thickness. The plane positions are stored in array $\underline{z}_{\text{plane}}$. From Eqs. (1)–(6), the value of α at an intersection of the ray with a plane can be easily calculated. For the intersection of the ray with planes, i, j, and k, it is found that

$$\alpha_x(i) = \frac{X_0 + i\Delta X - x_1}{x_2 - x_1}, \tag{7}$$

$$\alpha_y(j) = \frac{Y_0 + j\Delta Y - y_1}{y_2 - y_1}, \tag{8}$$

$$\alpha_z(k) = \frac{Z_k - z_1}{z_2 - z_1}, \tag{9}$$

for $\alpha_x(i)$ and $\alpha_y(j)$, this can be written in recursive form as

$$\alpha_x(i) = \alpha_x(i-1) + \frac{\Delta X}{x_2 - x_1} = \alpha_x(i-1) + \delta\alpha_x, \tag{10}$$

$$\alpha_y(j) = \alpha_y(j-1) + \frac{\Delta Y}{y_2 - y_1} = \alpha_y(j-1) + \delta\alpha_y. \tag{11}$$

Note that since the distance between two planes in the z direction is allowed to vary, such a recursive relation cannot be formulated for this direction. Using Eqs. (9)–(11), the values of α for all the intersections between the source point and the voxel the ray travels to can be calculated. This has to be done for every orthogonal plane direction and the results are stored in three different arrays

$$\underline{\alpha}_x = \{\alpha_x(i_{\min}), \ldots, \alpha_x(i_{\max})\}, \tag{12}$$

$$\underline{\alpha}_y = \{\alpha_y(j_{\min}), \ldots, \alpha_y(j_{\max})\}, \tag{13}$$

$$\underline{\alpha}_z = \{\alpha_z(k_{\min}), \ldots, \alpha_z(k_{\max})\}, \tag{14}$$

where subscripts "min" and "max" denote the first and last intersections of the ray with the orthogonal planes when traveling from the source to the target voxel.[7] When merging these three lists together with $\alpha_{\min}$ and $\alpha_{\max}$, this results in

$$\underline{\alpha} = \{\alpha_{\min}, \text{merge}(\underline{\alpha}_x, \underline{\alpha}_y, \underline{\alpha}_z), \alpha_{\max}\}, \tag{15}$$

where the merge operation as described by Siddon[7] creates an array with all the elements of α_x, α_y, and α_z in ascending order. Two consecutive elements from α are associated with entering and leaving a voxel. The length traveled in a voxel is therefore given by

$$l(m) = (\underline{\alpha}(m+1) - \underline{\alpha}(m))d_{12}, \tag{16}$$

where $l(m)$ is the length traveled in the mth voxel the ray passes when traveling from the source to the target voxel, d_{12} is the source-voxel distance. The voxel indices p and q can be calculated by

$$p(m) = \left\lfloor \frac{x_1 + \alpha_{\text{mid}}(x_2 - x_1) - X_0}{\Delta X} \right\rfloor, \tag{17}$$

$$q(m) = \left\lfloor \frac{y_1 + \alpha_{\text{mid}}(y_2 - y_1) - Y_0}{\Delta Y} \right\rfloor, \tag{18}$$

where $\lfloor \cdot \rfloor$ denotes the truncation operator and α_{mid} is given by

$$\alpha_{\text{mid}} = \frac{\underline{\alpha}(m) + \underline{\alpha}(m-1)}{2}. \tag{19}$$

The index in the z direction follows from a look-up table. The extent in the z direction in discretized into $N(=2048)$ intervals and for every interval the corresponding slice index is stored in an array $\underline{z}_{\text{index}}$. Using this approach, the slice index for a given z coordinate can be calculated by

$$r(m) = \underline{z}_{\text{index}}\left[\left\lfloor \frac{z_1 + \alpha_{\text{mid}}(z_2 - z_1) - Z_0}{(Z_{\max} - Z_0)/N} \right\rfloor\right], \tag{20}$$

where $Z_{\max}$ is the last "Z-plane". With this method the slice index can be retrieved fast. However, this comes at the cost of errors due to the discretization. The RPL is then eventually found by

$$\text{RPL} = d_{12} \sum_{m=1}^{N} \rho(p(m), q(m), r(m))(\underline{\alpha}(m+1) - \underline{\alpha}(m)), \tag{21}$$

where ρ is the relative electron density.

II.C. Method 1: Reference CPU algorithm

Siddon's algorithm[7] was modified in two respects for the purpose of optimization. This optimized algorithm is part of PLATO's dose calculation.[8] First of all, the volume that is ray traced can be limited to the volume inside the bounding box of the body contour because that is generally the volume of interest. As a side effect, the number of unnecessary loads from dynamic random access memory (DRAM) to cache memory is reduced.

The second optimization is based on reusing values of α. For all target voxels in a plane for which x is constant, the values of α_x for all "X planes" can be calculated once and reused for all voxels in this plane. The same holds for all voxels on a line in this particular plane for which y is constant. Here the values of α_y can be calculated once for all Y planes and they can be reused for all voxels in this line. In this way the number of intersections for which the value of α has to be calculated can be largely reduced. Timing of both strategies showed a reduction of 40%–60% in the calculation time.[8]

TABLE I. Specification of the hardware of systems I and II.

	System I	System II
CPU clock frequency (GHz)	2.66	2.66
Processor type	Intel® Xeon® quad-core	Intel®Q9400 quad-core
Graphics card type	NVIDIA®GeForce®8800 GTS	NVIDIA®GeForce®GTX 280
No. Processor cores[a]	96	240
Memory bandwith (Gbytes/s)	64	141.7
Graphical memory (Mbyte)	320	1024

[a]Reference 13.

The optimized CPU algorithm, using multithreading, will be referred to as the reference algorithm. This version is the current validated clinical standard as implemented (single threaded) in PLATO.

II.D. Method 2: GPU algorithm

Data parallelism was exploited by making every thread responsible for ray tracing to a single voxel in the region of interest. Rays can be traced completely independent and therefore this approach fits CUDA's programming model well.

Implementing Siddon's algorithm[7] on the graphics card required rewriting that part of the algorithm where $\underline{\alpha}_x$, $\underline{\alpha}_y$, and $\underline{\alpha}_z$ are computed and stored. For a CT dataset of size $512 \times 512 \times 100$, the following worst case estimation can be made for the dimensions of $\underline{\alpha}_x$, $\underline{\alpha}_y$, and $\underline{\alpha}_z$. If the source position is along the main diagonal of the CT volume and just outside the dataset, based on symmetry, the number of intersections is approximately half of the number of planes on average, i.e., 256, 256, and 50, respectively. Consequently, 2×562 floating-point numbers will be needed for every voxel, on average, to store the three lists and the merged result. This means that in total, approximately 100 Gbytes of memory is needed. Even though this is a very coarse estimation and largely overestimates the required amount of memory in practice, it illustrates that with this approach the available graphical memory will be insufficient. For this reason, the algorithm was rewritten by using a stepping approach. The algorithm in the Appendix describes the tracing procedure schematically.

After the determination of the value of α at the entrance of the volume (α_{min}), the potential next intersecting plane is calculated for three directions. These intersecting planes correspond to i_{min}, j_{min}, and k_{min}. Furthermore, the index of the entrance voxel in the linear array holding the density is determined.

After these initialization steps, the algorithm continues with a loop where the next intersecting plane is determined. This is achieved by computing the minimum of the value of α for the three potential next planes of intersection. This next intersecting plane now becomes the current plane of intersection and a new potential plane of intersection is calculated. This loop is continued until the value of α for the current intersecting plane is larger than or equal to unity. In order to access the the elements of $\underline{z}_{\text{plane}}$ fast, this array is stored in the constant memory of the graphics card.[13]

Voxel indices in the x and y directions are calculated from the value of α_{mid}. In the z direction the index is updated when leaving the current slice. Using this approach, the use of a look-up table in the z direction has become obsolete. This also eliminates the related discretization errors present in the reference algorithm.

II.E. Performance and accuracy analysis

The GPU algorithm was tested for accuracy and performance on two systems. Their specifications are summarized in Table I. Both systems have 32-bit Linux as their operating system (Fedora 8). All tests were done under CUDA version 1.1. The block size was chosen to be 16×16 threads for all evaluated cases. Based on hardware specifications, CPU timings are considered interchangeable for systems I and II.

A set of test geometries was defined. Table II gives a description of the used test geometries, and Table III describes the evaluated test cases. The different test cases were defined for resampled geometries with resampling factors (Δ_r) ranging from 1 to 4. The resampling factor is the reduction in resolution in the XY plane applied to both the X and Y directions. The resampling strategy, following the PLATO convention, is illustrated in Fig. 3.

TABLE II. Description of the different datasets that were used for performance and accuracy analysis. The bounding box is the smallest box that fits the patient. Original in-plane dimensions of the CT datasets are 512×512. Geometry A is a homogeneous water phantom. Geometries B and D are radiotherapy planning CT datasets of the pelvic region and geometry C is a planning CT of the head-and-neck region. As an illustration, Fig. 4 shows the central transversal slice for every dataset.

Geometry	No. of slices	Bounding box (−)	Spacing (mm)
A	31	452×452	$0.7^2 \times 10.0$
B	35	448×260	$0.8^2 \times 5.0$
C	103	499×292	$0.9^2 \times 3.0$
D	56	308×190	$0.9^2 \times 3.0$

TABLE III. Definition of five test cases. Test cases I–IV have been evaluated for test geometries A, B, and C. Test case V corresponds to irradiation with a vertex field and is therefore evaluated for test geometry C only. Figure 4 illustrates cases I–IV. Δ_r is the in-plane spacing relative to the original spacing.

Test case	Δ_r $(-)$	ϕ_{gantry} (deg)	ϕ_{table} (deg)
I	1	0	0
II	2	45	0
III	4	90	0
IV	4	270	0
V	1	90	90

II.E.1. Accuracy

Since the GPU ray tracing algorithm does not use a look-up table, the reference algorithm (described in Sec. II C) was extended with a more elaborate but exact algorithm to determine the corresponding slice index for a given z coordinate. This algorithm will be referred to as CPU hunt from here on[14] and is used as an accuracy benchmark for the GPU algorithm.

In addition, the GPU algorithm was compared to the reference algorithm to asses the effect of discretization errors resulting from the look-up table implementation.

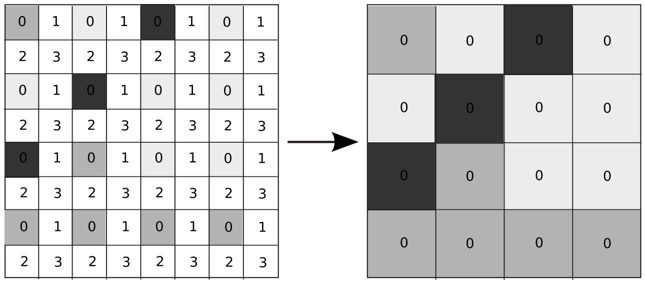


FIG. 3. Illustration of the resampling strategy, here for a resampling factor of 2. Four voxels, numbered from 0 to 3, are reduced to one voxel. The relative electron density is determined by the value of voxel 0. Resampling is applied in 2D, i.e., within the same CT slice.

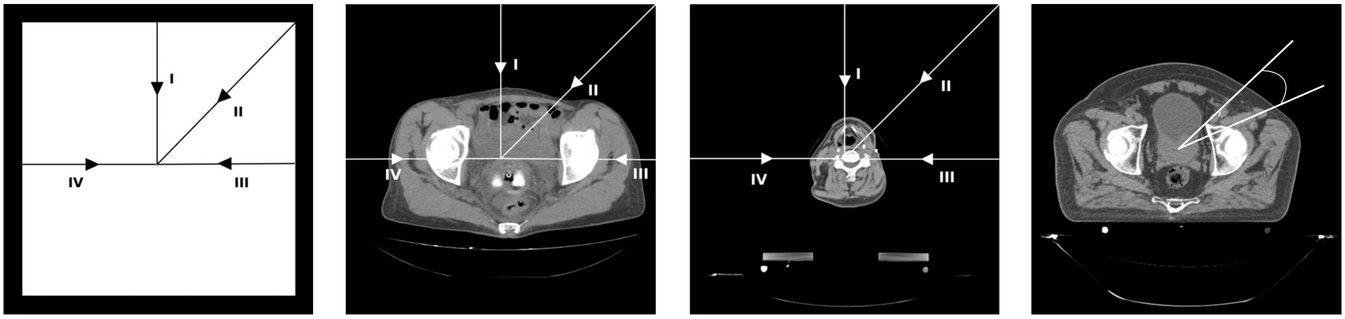
TABLE IV. Maximum absolute difference in RPL mm between the CPU-hunt algorithm and the GPU algorithm.

Test case	Test Geometry A	B	C
I	0.002	0.050	0.064
II	0.001	0.009	0.045
III	0.001	0.006	0.005
IV	0.000	0.004	0.007
V	–	–	0.011

II.E.2. Performance

Besides accuracy analysis, the performance of the GPU algorithm relative to the multithreaded CPU algorithm was investigated. Computation time was measured between the start of both ray tracing algorithms and the moment the results are stored in DRAM. For the GPU algorithm this means that the time needed to transfer the data from graphical memory to DRAM is included. Time required for initialization, e.g., transfer of data from DRAM to graphical memory (typically <20 ms) and driver initialization (typically $100-200$ ms) was excluded. The rationale for this choice is that for one planning session this is only done once while the actual ray tracing is performed numerous times.

As an additional performance benchmark, the gantry angle was rotated over 20° with 1° steps and for each source position the contribution to the dose distribution was calculated using both the reference and the GPU algorithm. Dose calculations were carried out for a 10×10 cm^2 field for two test geometries, C and D. Timings were performed using a resampling factor of both 1 and 2. In contrast to the GPU dose calculations, CPU dose calculations make a distinction between the beam volume extended with a margin (2 cm) and the remaining volume. Inside the first the specified resampling factor is used, outside a resampling factor of 8 is used. For geometry C, independent of the gantry angle, the field will "cover" the patients' head/neck. Here a starting angle of 0° was chosen. For geometry D, a pelvic case, a substantial part of the volume will be handled by the CPU



(a) (b) (c) (d)

FIG. 4. Transversal midplane for all test geometries. Roman symbols indicate test cases I–IV. For geometry D, the arc of source positions used for the dose calculation timings is indicated.

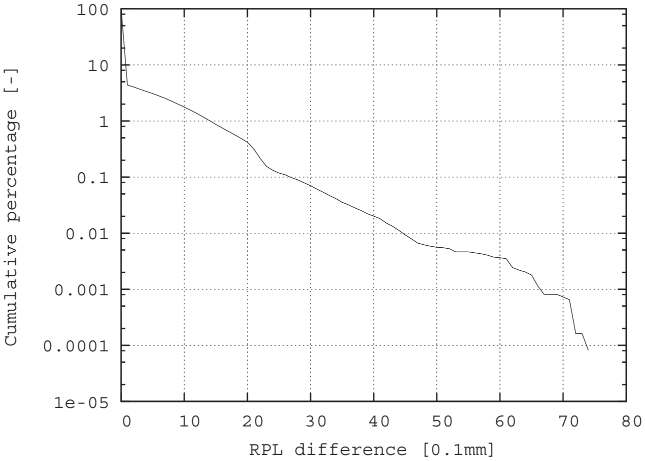


FIG. 5. Cumulative histogram of the absolute difference between the RPL calculated with the GPU and the CPU algorithm for test geometry C. The resampling factor was chosen to be 2.

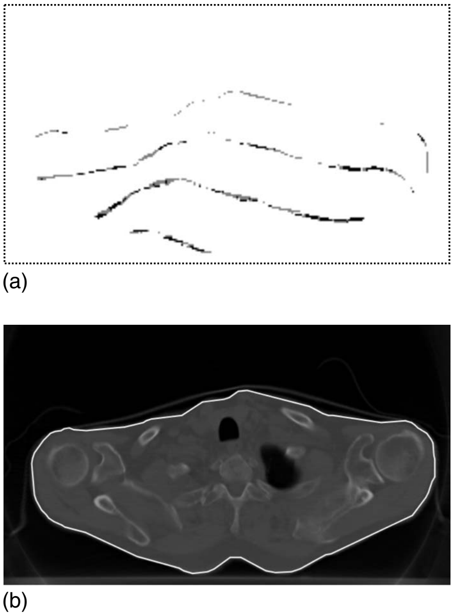
algorithm using a downsampled geometry when starting with a gantry angle of 0°. For this reason, the arc was started at 45°.

III. RESULTS

III.A. Accuracy analysis

III.A.1. GPU versus CPU hunt

Table IV shows the results of the comparison of the CPU-hunt algorithm with the presented GPU algorithm. For all evaluated test cases, the maximum difference found was 0.064 mm. The results given by the GPU algorithm for the two different graphics cards were compared and no differences were found. For this reason, the comparison between CPU-hunt and the GPU algorithm is reported for one system only.


(a)

(b)

FIG. 6. An example absolute difference map of the RPL for geometry C with the corresponding CT slice (31) for test case II. White, gray, and black correspond to 0–0.9, 0.9–1.8, and ⩾1.8 mm, respectively. Discretization errors at the body contour of the patient yield RPL differences projecting at the neighboring slices. Differences from multiple slices can project on the same slice as is clearly visible in this example. As can clearly be seen, there is a close resemblance between the body contour of this slice and the upper "contour" in the RPL difference map.

TABLE V. Timing of the reference (multithreaded CPU) and the GPU ray tracing algorithm on systems I and II for test geometry A. S denotes speedup, GPU execution time relative to CPU execution time.

		System I		System II	
Test case	CPU (s)	GPU (s)	S (−)	GPU (s)	S (−)
I	7.52	1.91	3.9	0.85	8.9
II	1.31	0.32	4.1	0.15	8.9
III	0.21	0.06	3.8	0.02	10.1
IV	0.18	0.05	3.6	0.02	8.4

III.A.2. GPU versus reference CPU algorithm

Figure 5 shows a cumulative histogram of the RPL difference between the GPU and the reference CPU algorithm for test geometry C using a resampling factor of 2. Due to discretization errors introduced by the look-up table, it was found that the slice index is incidentally calculated incorrectly by the CPU algorithm. For this particular example in 0.1% of the volume, a deviation larger than 2.7 mm was found (for geometries A and B a deviation larger than 4.4×10^{-4} and 1.5 mm, respectively, was found in 0.1% of the volume). As an illustration of the effect of discretization errors in the calculation of the slice index, Fig. 6 shows a slice of the absolute RPL difference map for geometry C, test case II.

III.B. Performance analysis

The timings of ray tracing for the different test cases are presented in Tables V–VII. Although, the achieved speedup is observed to be strongly dependent on the geometry tested for, the GPU algorithm is always faster. The timings for dose calculations when radiating while rotating the gantry over 20° in 1° steps are presented in Table VIII. The timings are

TABLE VI. Timing of the reference (multithreaded CPU) and the GPU ray tracing algorithm on systems I and II for test geometry B. S denotes speedup, GPU execution time relative to CPU execution time.

		System I		System II	
Test case	CPU (s)	GPU (s)	S (−)	GPU (s)	S (−)
I	2.22	1.08	2.1	0.47	4.7
II	0.44	0.13	3.3	0.06	7.1
III	0.1	0.03	3.4	0.01	8.1
IV	0.1	0.03	3.4	0.01	9.0

TABLE VII. Timing of the reference (multithreaded CPU) and the GPU ray tracing algorithm on systems I and II for test geometry C. S denotes speedup, GPU execution time relative to CPU execution time.

		System I		System II	
Test case	CPU (s)	GPU (s)	S (−)	GPU (s)	S (−)
I	5.98	3.88	1.5	1.82	3.3
II	1.12	0.84	1.3	0.41	2.7
III	0.18	0.12	1.5	0.06	2.8
IV	0.17	0.11	1.5	0.08	2.1
V	4.61	3.28	1.4	1.75	2.6

reported for resampling factors 1 and 2 for the multithreaded CPU algorithm and the GPU algorithm timed on both systems I and II. The tested geometries are C and D. Averaged over the 21 source positions, the times needed for a single dose calculation with a resampling factor of 2 were found to be 0.9 and 0.6 s, respectively.

IV. DISCUSSION AND CONCLUSIONS

The aim of this study was to investigate the possible acceleration of the presently used ray tracing algorithm for radiotherapy dose calculations using a GPU. Both accuracy and execution time of the GPU algorithm were compared to a CPU benchmark. It was shown that accurate ray tracing can be performed, on average, six times faster than with the multithreaded reference CPU algorithm.

The difference in radiological path length between the CPU-hunt and the GPU algorithm was found to be smaller than 0.1 mm, the precision with which the RPL is stored in practice. The present differences can be explained by differences in accumulation of round-off errors. In the GPU algorithm the values of α are calculated incrementally, whereas for the CPU algorithm for all the planes of intersection, the value of α is calculated independently using Eqs. (7)–(9).

The look-up table introduces discretization errors that can lead to a difference in RPL up to 7.5 mm (for test geometry C). At a large angle of incidence (relative to the z direction) and high density contrast, the GPU algorithm is therefore more accurate than the reference CPU algorithm.

These differences however are found in a marginal fraction of the irradiated volume: A deviation larger than 2.7 mm is found in only 0.1% of the volume (Fig. 5). Figure 6 shows that discretization errors at the location of the body contour are projected onto neighboring slices. Together with discretization errors at other high density contrast regions, this explains the found differences between the reference CPU and the GPU algorithm. As shown, in case of irradiation with a vertex field, the maximum error found was 0.5 mm. This is consistent with the explanation for the deviation in the coplanar cases since the angle of incidence is close to normal in this particular case. In clinical practice, the differences will even be less significant since the location of the differences found will be dependent on the beam angle. Deviations in the dose distribution will therefore be much smaller for a multiple beam plan than for a single beam.

The presented timings show that the GPU implementation can speed up the multithreaded CPU ray tracing algorithm considerably by up to a factor of 10. This speedup, however, is observed to be strongly dependent on the geometry. The speedup found for geometry C, the head-and-neck case, is relatively small. For this geometry, a relatively large fraction of voxels within the bounding box contains air and no tracing needs to be done to these voxels. Threads are started, however, for all voxels within the bounding box, ray tracing is only performed when the density is unequal to zero. When there are threads within a warp (a group of threads executed physically in parallel) that are tracing and threads that are idle, this effectively lowers the level of occupation of the GPU and therefore reduces performance. This could possibly be solved by compressing the grid by the removal of all threads that do not have to do any work.

The time required for a complete single beam dose calculation at a resampling factor of 2 was measured to be 0.6–0.9 s, depending on the geometry. The reported speedups for dose calculation are lower than the speedups reported for ray tracing only. The speedups measured for geometry C were 1.5 and 1.8 using resampling factors of 1 and 2, respectively. For ray tracing alone, speedups of 3.3 and 2.7 were measured. This can be explained by the fact the number of voxels for which ray tracing is carried out is larger for the GPU algorithm than that for the CPU algorithm since no distinction is made between being in- and outside the "beam volume." Furthermore, although more than 50% of the dose calculation time is made up by ray tracing, the remaining calculation time is not influenced by the GPU acceleration making the overall reduction relatively smaller.

It is inherently unknown which elements of the density array will be accessed during ray tracing. This results in a complex access pattern that is not consistent with the requirements for fast parallel accessing of data.[13] The ray tracing algorithm could possibly be optimized using texture memory. Texture memory is read-only memory that is optimized to acces grid data fast.[13,15]

Comparing the performance of the two different graphics cards, it is found that the GTX 280 card is on average 2.16

TABLE VIII. Timing of dose calculations for an arc of 21 source positions for resampling factors of 1 and 2 and two test geometries. S denotes speedup, GPU execution time relative to CPU execution time. Δ_r is the in-plane spacing relative to the original spacing. The starting angles were set to be 0° and 45°, respectively.

			System I		System II	
Geometry	Δ_r (−)	CPU (s)	GPU (s)	S (−)	GPU (s)	S (−)
C	1	97	90	1.1	64	1.5
	2	32	18	1.8	18	1.8
D	1	314	59	5.4	51	6.2
	2	55	15	3.7	11	4.8

times faster than the 8800 GTS card (minimum of 1.4, maximum of 2.66). The first benefits from a larger number of processor cores (240 versus 96) and a higher theoretical memory bandwidth (141.7 versus 64 Gbytes/s).

The developed GPU algorithm now enables dose calculations at a speed that will be experienced as real time for conventional forward planning based on clinically relevant datasets. In addition, there is no longer a dependency between execution time and field size. As mentioned in Sec. I this can lead to a major reduction in the workload of radiotherapy treatment planning. Moreover, the presented GPU algorithm can be used to accelerate more advanced treatment planning optimization techniques.[1–5]

ACKNOWLEDGMENTS

For this work, the source code of the PLATO planning system was used within the framework of the AMC–Nucletron (Veenendaal) treatment planning collaboration. Financial support was given by the Dutch Cancer Society (Grant No. UVA-2006-3484).

APPENDIX: GPU RAY TRACING ALGORITHM

```
$\alpha_{\text{current}}=\alpha_{\min}$ {Initialization}
$\alpha_{x,\text{next}}=\alpha_x(i_{\min})$ {$\alpha$ values of potential next intersecting planes}
$\alpha_{y,\text{next}}=\alpha_y(j_{\min})$
$\alpha_{z,\text{next}}=\alpha_z(k_{\min})$
Initialize $v$, $v_{\text{old}}$ {Linear Array indices}
while $\alpha_{\text{current}}<1.0$ do
  if $\alpha_{y,\text{next}}<\alpha_{x,\text{next}}$ && $\alpha_{y,\text{next}}<\alpha_{z,\text{next}}$ then
    $\alpha_{\text{mid}}=(\alpha_{\text{current}}+\alpha_{y,\text{next}})/2$ {Ray intersects $Y$-plane}
    $l=\alpha_{y,\text{next}}-\alpha_{\text{current}}$
    $\alpha_{\text{current}}=\alpha_{y,\text{next}}$
    $\alpha_{y,\text{next}}=\alpha_{y,\text{next}}+\delta\alpha_y$ {Potentially next $Y$-plane.}
    update $v$
  else if $\alpha_{x,\text{next}}<\alpha_{z,\text{next}}$ then
    $\alpha_{\text{mid}}=(\alpha_{\text{current}}+\alpha_{x,\text{next}})/2$ {Ray intersects $X$-plane}
    $l=\alpha_{x,\text{next}}-\alpha_{\text{current}}$
    $\alpha_{\text{current}}=\alpha_{x,\text{next}}$
    $\alpha_{x,\text{next}}=\alpha_{x,\text{next}}+\delta\alpha_x$ {Potentially next $Z$-plane.}
    update $v$
  else
    $\alpha_{\text{mid}}=(\alpha_{\text{current}}+\alpha_{z,\text{next}})/2$ {Ray intersects $Z$-plane}
    $l=\alpha_{x,\text{next}}-\alpha_{\text{current}}$
    $\alpha_{\text{current}}=\alpha_{z,\text{next}}$
    $\alpha_{z,\text{next}}=(z_{\text{plane}}[k]-z_1)/(z_2-z_1)$ {Potentially next $Z$-plane}
    update $v$
    update k
end if
if $\rho[v_{\text{old}}]\neq 0.0$ then
    RPL=RPL+$\rho[v_{\text{old}}]l$
  end if
    $v_{\text{old}}=v$
end while
  RPL=$d_{12}$RPL
```

[a] Electronic mail: m.degreef@amc.uva.nl

[1] K. Otto, “Volumetric modulated arc therapy: IMRT in a single gantry arc,” Med. Phys. **35**, 310–317 (2008).

[2] C. Yu, “Intensity-modulated arc therapy with dynamic multileaf collimation: An alternative to tomotherapy,” Phys. Med. Biol. **40**, 1435–1449 (1995).

[3] E. Wong, J. Chen, and J. Greenland, “Intensity-modulated arc therapy simplified,” Int. J. Radiat. Oncol., Biol., Phys. **53**, 222–235 (2002).

[4] T. Mackie, T. Holmes, S. Swerdloff, P. Reckwerdt, J. Deasy, J. Yang, B. Paliwal, and T. Kinsella, “Tomotherapy: A new concept for the delivery of dynamic conformal radiotherapy,” Med. Phys. **20**, 1709–1719 (1993).

[5] G. Grigorov, T. Kron, E. Wong, J. Chen, J. Sollazzo, and G. Rodrigues, “Optimization of helical tomotherapy treatment plans for prostate cancer,” Phys. Med. Biol. **48**, 1933–1943 (2003).

[6] D. Jaffray, J. Siewerdsen, J. Wong, and A. Martinez, “Flat-panel cone-beam computed tomography for image-guided radiation therapy,” Int. J. Radiat. Oncol., Biol., Phys. **53**, 1337–1349 (2002).

[7] R. Siddon, “Fast calculation of the exact radiological path for a three-dimensional CT array,” Med. Phys. **12**, 252–255 (1985).

[8] A. Bel, “Speed optimization of ray tracing for 3D dose calculation,” Radiother. Oncol. **61**, s38 (2001).

[9] J. Owens *et al.*, “A survey of general-purpose computation on graphics hardware,” Comput. Graph. Forum **26**, 80–113 (2007).

[10] *GPU Gems 2* edited by M. Pharr (Addison-Wesley Professional, Upper Saddle River, NJ, 2005), p. 451.

[11] Brook GPU, http://graphics.stanford.edu/projects/brookgpu/.

[12] RapidMind, http://www.rapidmind.net/.

[13] NVIDIA CUDA Compute Unified Device Architecture Programming Guide, NVIDIA Corporation, 1.1 edition (2007).

[14] W. Press, S. Teukolsky, W. Vetterling, and B. Flannery, *Numerical Recipes in C*, 2nd ed. (Cambridge University Press, Cambridge, 1992).

[15] NVIDIA CUDA Compute Unified Device Architecture Programming Guide, NVIDIA Corporation, 2.1 edition (2008).