Loco-regional hyperthermia treatment planning: optimisation under uncertainty

de Greef, M.

Citation for published version (APA):
Chapter 7

Accelerated ray tracing for radiotherapy dose calculations on a GPU

This chapter was published as:

Purpose: The graphical processing unit (GPU) on modern graphics cards provides the possibility to accelerate arithmetically intensive tasks. By splitting the work in a large number of independent jobs, order-of-magnitude speeds-ups are reported. In this article, the possible speed-up 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 accuracy and speed to a multi-threaded version of the PLATO ray tracing algorithm. This comparison was performed for three test geometries, a phantom and two radiotherapy planning CT data-sets (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. Speed-ups for ray tracing were found to be in the range of 2.1 – 10.1, relative to the multi-threaded PLATO algorithm running four threads. For dose calculations the speed-up measured was in the range of 1.5 – 6.2. For the speed-up 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 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 data-sets. These findings can be used in conventional planning to enable (nearly) real-time dose calculations. Also the importance for treatment optimization techniques is evident.

7.1 Introduction

Ray-tracing is an important and time consuming step in dose calculations for radiotherapy. For a 3D CT data-set, the relative electron density has to be line-integrated from the source position to every volume element (voxel) in that data-set. 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 seconds of 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 (VMAT) (86), intensity modulated arc therapy (IMAT) (87; 88) and/or tomotherapy (89; 90). Speed improvement is also important in view of the growing amount of data acquired just prior or during treatment (91). Image-guided radiotherapy (IGRT) can lead to adaptive procedures with possible re-planning during the course of treatment.

Multi-core processors have now become common. However, faster execution can only be achieved using multi-threading which requires rewriting of the algorithm. Moreover, the speed-up is related to the number of cores leading to a maximum speed-up of four times in practice.

Another, rather expensive, option is using large CPU clusters. However, the associated overhead that 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 (92). 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 (93). 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 (94). This use of the GPU outside its original context of computer graphics is referred to as general purpose computation on the GPU (GPGPU). The reported speed-ups are achieved by separating an arithmetically intensive task in a large number of light-weight 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.

7.2 Materials and Methods

7.2.1 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 programmable and are developing fast regarding their computational capacities (95). 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 (96), RapidMind (97) and CUDA (Compute Unified Device Architecture) by NVIDIA (98). In this study the latter language was used. CUDA is an extension of standard C that is pre-processed 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 (figure 7.1).

7.2.2 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 (92), the original algorithm is briefly summarized here. The algorithm is based on the representation of a 3D data-set 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 travelled by this ray in each voxel, multiplied by the relative electron density of the voxel. Figure 7.2
Figure 7.1: 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.

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 behaviour 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:

\[
\begin{align*}
    x &= x_1 + \alpha(x_2 - x_1), \\
    y &= y_1 + \alpha(y_2 - y_1), \\
    z &= z_1 + \alpha(z_2 - z_1),
\end{align*}
\]

where index 1 refers to the source position, index 2 to the center of the voxel the ray is traced to and \( \alpha \) is the distance travelled divided by the source - voxel distance. Now the data set can be expressed as

\[
\begin{align*}
    X_i &= X_0 + i\Delta X, \\
    Y_j &= Y_0 + j\Delta Y, \\
    Z_k &= z_{\text{plane}}[k],
\end{align*}
\]

where the use of capital symbols refers to the fact that the coordinates indicate plane positions, \( \Delta X \) and \( \Delta Y \) are the voxel dimensions in \( x \)- and \( y \)-direction respectively, \( i \),
Figure 7.2: 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 data set. The voxels a ray travels through on its way to a certain voxel in the data-set are collected so that the length of the fragments can be density weighted.

\( j \) and \( k \) are plane indices and \( X_0 \) and \( Y_0 \) are the \( x \)- and \( y \)-coordinate of the planes that are defined by \( i \) and \( j \) equal to zero, respectively. In contrast to the original algorithm the data-set is allowed to be non-equidistant in \( z \)-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 \( z_{\text{plane}} \). From equation 7.1 – 7.3 and 7.4 – 7.6, the value of alpha at an intersection of the ray with a plane can be easily calculated. For the intersection of the ray with plane \( i \), \( j \) and \( k \) it is found that:

\[
\alpha_x(i) = \frac{X_0 + i \Delta X - x_1}{x_2 - x_1},
\]

\[
\alpha_y(j) = \frac{Y_0 + j \Delta Y - y_1}{y_2 - y_1},
\]

\[
\alpha_z(k) = \frac{Z_k - z_1}{z_2 - z_1},
\]
for $\alpha_x(i)$ and $\alpha_y(j)$ this can be written in recursive form as

$$\alpha_x(i) = \alpha_x(i - 1) + \frac{\Delta X}{x_2 - x_1} = \alpha_x(i - 1) + \delta \alpha_x, \quad (7.10)$$

$$\alpha_y(j) = \alpha_y(j - 1) + \frac{\Delta Y}{y_2 - y_1} = \alpha_y(j - 1) + \delta \alpha_y. \quad (7.11)$$

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 equation 7.9, 7.10 and 7.11, the values of $\alpha$ for all the intersections between 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:

$$\alpha_x = \{\alpha_x(i_{\text{min}}), \ldots , \alpha_x(i_{\text{max}})\}, \quad (7.12)$$

$$\alpha_y = \{\alpha_y(j_{\text{min}}), \ldots , \alpha_y(j_{\text{max}})\}, \quad (7.13)$$

$$\alpha_z = \{\alpha_z(k_{\text{min}}), \ldots , \alpha_z(k_{\text{max}})\}, \quad (7.14)$$

where subscripts ‘min’ and ‘max’ denote the first and last intersection of the ray with the orthogonal planes when traveling from source to the target voxel (92). When merging these three lists together with $\alpha_{\text{min}}$ and $\alpha_{\text{max}}$ this results in

$$\alpha = \{\alpha_{\text{min}}, \text{merge}(\alpha_x, \alpha_y, \alpha_z), \alpha_{\text{max}}\}, \quad (7.15)$$

where the merge operation as described by Siddon (92) creates an array with all the elements of $\alpha_x$, $\alpha_y$ and $\alpha_z$ in ascending order. Two consecutive elements from $\alpha$ are associated with entering and leaving a voxel. The length traveled in a voxel is therefore given by

$$l(m) = (\alpha(m + 1) - \alpha(m))d_{12}, \quad (7.16)$$

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

$$p(m) = \left\lceil \frac{x_1 + \alpha_{\text{mid}}(x_2 - x_1) - X_0}{\Delta X} \right\rceil, \quad (7.17)$$

$$q(m) = \left\lceil \frac{y_1 + \alpha_{\text{mid}}(y_2 - y_1) - Y_0}{\Delta Y} \right\rceil, \quad (7.18)$$
where $\lfloor \cdot \rfloor$ denotes the truncation operator and $\alpha_{\text{mid}}$ is given by

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

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

$$
r(m) = z_{\text{index}} \lfloor z_1 + \alpha_{\text{mid}}(z_2 - z_1) - Z_0 \right) / (Z_{\text{max}} - Z_0) / N \rfloor.
$$

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

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

where $\rho$ is the relative electron density.

### 7.2.3 Method 1: reference CPU algorithm

Siddon’s algorithm (92) was modified in two respects for the purpose of optimization. This optimized algorithm is part of PLATO’s dose calculation (93). First of all the volume that is ray-traced can be limited to the volume inside of 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 re-using values of $\alpha$. For all target voxels in a plane for which $x$ is constant, the values of $\alpha_x$ for all ’$X$-planes’ can be calculated once and re-used 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 $\alpha_y$ can be calculated once for all $Y$-planes and they can be re-used for all voxels in this line. In this way the number of intersections for which the value of $\alpha$ has to be calculated can be largely reduced. Timing of both strategies showed a reduction of 40 – 60% in calculation time (93).

The optimized CPU algorithm, using multi-threading, will be referred to as the reference algorithm. This version is the current validated clinical standard as imple-
mented (single-threaded) in PLATO.

7.2.4 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 (92) on the graphics card required rewriting that part of the algorithm where $\alpha_x, \alpha_y, \alpha_z$ are computed and stored. For a CT data-set of size $512 \times 512 \times 100$, the following worst case estimation can be made for the dimensions of $\alpha_x, \alpha_y$ and $\alpha_z$. If the source position is along the main diagonal of the CT volume and just outside of the data-set, based on symmetry, the number of intersections is approximately half the number of planes on average i.e. 256, 256 and 50, respectively. Consequently, $2 \times 562$ 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 GB 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. Algorithm 1 describes the tracing procedure schematically.

After determination of the value of $\alpha$ at the entrance of the volume ($\alpha_{\text{min}}$), the potential next intersecting plane is calculated for three directions. These intersecting planes correspond with $i_{\text{min}}, j_{\text{min}}$ and $k_{\text{min}}$. 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 $\alpha$ 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 the $\alpha$ for the current intersecting plane is larger than or equal to unity.

In order to access the the elements of $\hat{z}_{\text{plane}}$ fast, this array is stored in the constant memory of the graphics card (98).

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

The GPU algorithm was tested for accuracy and performance on two systems. Their specifications are summarized in table 7.1. Both systems have 32-bits Linux as their operating system (Fedora 8). All tests were done under CUDA version 1.1. The block size was chosen to be $16 \times 16$ threads for all evaluated cases. Based on hardware specifications, CPU timings are considered interchangeable for system I and II.

A set of test geometries was defined 7.4. Table 7.2 gives a description of the used test geometries, table 7.3 describes the evaluated test cases. The different test cases were defined for re-sampled geometries with re-sampling factors ($\Delta_r$) ranging from 1 to 4. The re-sampling factor is the reduction in resolution in the $XY$-plane applied to both the $X$- and $Y$-direction. The re-sampling strategy, following the PLATO convention, is illustrated in figure 7.3.

<table>
<thead>
<tr>
<th>System I</th>
<th>System II</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU clock frequency (GHz)</td>
<td>2.66</td>
</tr>
<tr>
<td>Processor type</td>
<td>Intel® Xeon® quad-core</td>
</tr>
<tr>
<td>Graphics card type</td>
<td>NVIDIA® GeForce®8800 GTS</td>
</tr>
<tr>
<td># Processor cores (98)</td>
<td>96</td>
</tr>
<tr>
<td>Memory bandwidth (GB/s)</td>
<td>64</td>
</tr>
<tr>
<td>Graphical Memory (MB)</td>
<td>320</td>
</tr>
</tbody>
</table>

Table 7.1: Specification of the hardware of system I and II.

<table>
<thead>
<tr>
<th>Geometry</th>
<th># Slices</th>
<th>Bounding box (mm)</th>
<th>Spacing (mm)</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>31</td>
<td>$452 \times 452$</td>
<td>$0.7^2 \times 10.0$</td>
</tr>
<tr>
<td>B</td>
<td>35</td>
<td>$448 \times 260$</td>
<td>$0.8^2 \times 5.0$</td>
</tr>
<tr>
<td>C</td>
<td>103</td>
<td>$499 \times 292$</td>
<td>$0.9^2 \times 3.0$</td>
</tr>
<tr>
<td>D</td>
<td>56</td>
<td>$308 \times 190$</td>
<td>$0.9^2 \times 3.0$</td>
</tr>
</tbody>
</table>

Table 7.2: Description of the different data-sets 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 data-sets are $512 \times 512$. Geometry A is a homogeneous water phantom. Geometries B and D are radiotherapy planning CT data-sets of the pelvic region and geometry C is a planning CT of the head-and-neck region. As an illustration, figure 7.4 shows the central transversal slice for every data-set.
Figure 7.3: Illustration of the re-sampling strategy, here for a re-sampling 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. Re-sampling is applied in 2D i.e. within the same CT slice.

<table>
<thead>
<tr>
<th>Test Case</th>
<th>$\Delta_r$ (-)</th>
<th>$\phi_{\text{gantry}}$ (deg)</th>
<th>$\phi_{\text{table}}$ (deg)</th>
</tr>
</thead>
<tbody>
<tr>
<td>I</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>II</td>
<td>2</td>
<td>45</td>
<td>0</td>
</tr>
<tr>
<td>III</td>
<td>4</td>
<td>90</td>
<td>0</td>
</tr>
<tr>
<td>IV</td>
<td>4</td>
<td>270</td>
<td>0</td>
</tr>
<tr>
<td>V</td>
<td>1</td>
<td>90</td>
<td>90</td>
</tr>
</tbody>
</table>

Table 7.3: Definition of five test cases. Test cases I–IV have been evaluated for test geometry A, B and C. Test case V corresponds to irradiation with a vertex field and is therefore evaluated for test geometry C only. Figure 7.4 illustrates case I – IV. $\Delta_r$ is the in-plane spacing relative to the original spacing.

Accuracy

Since the GPU ray tracing algorithm does not use a look-up table, the reference algorithm (described in section 7.2.3) 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 (99) and is used as an accuracy benchmark for the GPU algorithm.

In addition, the GPU algorithm was compared to the reference algorithm to assess the effect of discretization errors resulting from the look-up table implementation.
Figure 7.4: Transversal mid-plane for all test geometries. Roman symbols indicate test case I–IV. For geometry D the arc of source positions used for the dose calculation timings is indicated.

Performance

Besides accuracy analysis, the performance of the GPU algorithm relative to the multi-threaded 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 $10 \times 10$ cm$^2$ field for two test geometries, geometry C and D. Timings were performed using a re-sampling 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 re-sampling factor is used, outside a re-sampling 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 $0^\circ$ was chosen. For geometry D, a pelvic case, a substantial part of the volume will be handled by the CPU algorithm using a down-sampled geometry when starting with a gantry angle of $0^\circ$. For this reason the arc was started at $45^\circ$.

### 7.3 Results

#### 7.3.1 Accuracy analysis

**GPU versus CPU-hunt**

Table 7.4 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.

<table>
<thead>
<tr>
<th>Test Case</th>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>I</td>
<td>0.002</td>
<td>0.050</td>
<td>0.064</td>
</tr>
<tr>
<td>II</td>
<td>0.001</td>
<td>0.009</td>
<td>0.045</td>
</tr>
<tr>
<td>III</td>
<td>0.001</td>
<td>0.006</td>
<td>0.005</td>
</tr>
<tr>
<td>IV</td>
<td>0.000</td>
<td>0.004</td>
<td>0.007</td>
</tr>
<tr>
<td>V</td>
<td>-</td>
<td>-</td>
<td>0.011</td>
</tr>
</tbody>
</table>

Table 7.4: Maximum absolute difference in RPL mm between the CPU-hunt algorithm and the GPU algorithm.
GPU versus reference CPU algorithm

Figure 7.5 shows a cumulative histogram of the RPL difference between the GPU and the reference CPU algorithm for test geometry C using a re-sampling 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 geometry A and B a deviation larger than $4.4 \times 10^{-4}$ 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, figure 7.6 shows a slice of the absolute RPL difference map for geometry C, test case II.

\begin{figure}[h]
\centering
\includegraphics[width=0.5\textwidth]{figure7.5.png}
\caption{Cumulative histogram of the absolute difference between the RPL calculated with the GPU and the CPU algorithm for test geometry C. The re-sampling factor was chosen to be 2.}
\end{figure}

7.3.2 Performance analysis

The timings of ray tracing for the different test cases are presented in table 7.5 – 7.7. Although, the achieved speed-up 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 7.8. The timings are reported for re-sampling factors 1 and 2 for the multi-threaded CPU algorithm and the GPU algorithm timed on both system I and II.
<table>
<thead>
<tr>
<th>Test Case</th>
<th>System I</th>
<th>System II</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>CPU (s)</td>
<td>GPU (s)</td>
</tr>
<tr>
<td>I</td>
<td>7.52</td>
<td>1.91</td>
</tr>
<tr>
<td>II</td>
<td>1.31</td>
<td>0.32</td>
</tr>
<tr>
<td>III</td>
<td>0.21</td>
<td>0.06</td>
</tr>
<tr>
<td>IV</td>
<td>0.18</td>
<td>0.05</td>
</tr>
</tbody>
</table>

Table 7.5: Timing of the reference (multi-threaded CPU) and the GPU ray tracing algorithm on system I and II for test geometry A. S denotes speed-up, GPU execution time relative to CPU execution time.

<table>
<thead>
<tr>
<th>Test Case</th>
<th>System I</th>
<th>System II</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>CPU (s)</td>
<td>GPU (s)</td>
</tr>
<tr>
<td>I</td>
<td>2.22</td>
<td>1.08</td>
</tr>
<tr>
<td>II</td>
<td>0.44</td>
<td>0.13</td>
</tr>
<tr>
<td>III</td>
<td>0.1</td>
<td>0.03</td>
</tr>
<tr>
<td>IV</td>
<td>0.1</td>
<td>0.03</td>
</tr>
</tbody>
</table>

Table 7.6: Timing of the reference (multi-threaded CPU) and the GPU ray tracing algorithm on system I and II for test geometry B. S denotes speed-up, GPU execution time relative to CPU execution time.

<table>
<thead>
<tr>
<th>Test Case</th>
<th>System I</th>
<th>System II</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>CPU (s)</td>
<td>GPU (s)</td>
</tr>
<tr>
<td>I</td>
<td>5.98</td>
<td>3.88</td>
</tr>
<tr>
<td>II</td>
<td>1.12</td>
<td>0.84</td>
</tr>
<tr>
<td>III</td>
<td>0.18</td>
<td>0.12</td>
</tr>
<tr>
<td>IV</td>
<td>0.17</td>
<td>0.11</td>
</tr>
<tr>
<td>V</td>
<td>4.61</td>
<td>3.28</td>
</tr>
</tbody>
</table>

Table 7.7: Timing of the reference (multi-threaded CPU) and the GPU ray tracing algorithm on system I and II for test geometry C. S denotes speed-up, GPU execution time relative to CPU execution time.
Table 7.8: Timing of dose calculations for an arc of 21 source positions for re-sampling factor 1 and 2 and two test geometries. \( S \) denotes speed-up, GPU execution time relative to CPU execution time. \( \Delta r \) is the in-plane spacing relative to the original spacing. The starting angles were set to be 0 and 45°, respectively.

<table>
<thead>
<tr>
<th>Geometry</th>
<th>( \Delta r ) (-)</th>
<th>CPU (s)</th>
<th>GPU (s)</th>
<th>( S ) (-)</th>
<th>GPU (s)</th>
<th>( S ) (-)</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td>1</td>
<td>97</td>
<td>90</td>
<td>1.1</td>
<td>64</td>
<td>1.5</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>32</td>
<td>18</td>
<td>1.8</td>
<td>18</td>
<td>1.8</td>
</tr>
<tr>
<td>D</td>
<td>1</td>
<td>314</td>
<td>59</td>
<td>5.4</td>
<td>51</td>
<td>6.2</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>55</td>
<td>15</td>
<td>3.7</td>
<td>11</td>
<td>4.8</td>
</tr>
</tbody>
</table>

The tested geometries are C and D. Averaged over the 21 source positions, the time needed for a single dose calculation with a re-sampling factor of 2 was found to be 0.9 and 0.6 s, respectively.

## 7.4 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 6 times faster than with the multi-threaded 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 \( \alpha \) are calculated incrementally whereas for the CPU algorithm for all the planes of intersection, the value of \( \alpha \) is calculated independently using equations 7.7 – 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 vol-
volume: a deviation larger than 2.7 mm is found in only 0.1% of the volume (figure 7.5). Figure 7.6 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 co-planar 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 multi-threaded CPU ray tracing algorithm considerably, up to a factor of 10. This speed-up however is observed to be strongly dependent on the geometry. The speed-up 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 possible be solved by compressing the grid by removal of all threads that do not have to do any work.

The time required for a complete single beam dose calculation at a re-sampling factor of 2 was measured to be 0.6 – 0.9 seconds, depending on the geometry. The reported speed-ups for dose calculation are lower than the speed-ups reported for ray tracing only. The speed-up measured for geometry C was 1.5 and 1.8 using a re-sampling factor of 1 and 2, respectively. For ray tracing alone, speed-ups 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 for the CPU algorithm since no distinction is made between being in- and outside the ‘beam volume’. Furthermore, although more than 50 percent 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 (98). The ray tracing algorithm could possibly be optimized using texture memory. Texture memory is read-only memory that is optimized to access grid data fast (98; 100).

Comparing the performance of the two different graphics cards, it is found that the GTX 280 card is on average 2.16 times faster than the 8800 GTS card (minimum 1.4, maximum 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 GB/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 data-sets. In addition, there is no longer a dependency between execution time and field size. As mentioned in the introduction this can lead to a major reduction in the work load of radiotherapy treatment planning. Moreover, the presented GPU algorithm can be used to accelerate more advanced treatment planning optimization techniques (86; 87; 88; 89; 90).

Acknowledgements

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 UVA – 2006 – 3484).
Algorithm 1 Tracing

\[ \alpha_{\text{current}} = \alpha_{\text{min}} \{ \text{Initialization} \} \]
\[ \alpha_{x, \text{next}} = \alpha_{x}(i_{\text{min}}) \{ \alpha \text{ values of potential next intersecting planes} \} \]
\[ \alpha_{y, \text{next}} = \alpha_{y}(j_{\text{min}}) \]
\[ \alpha_{z, \text{next}} = \alpha_{z}(k_{\text{min}}) \]

Initialize \( v, v_{\text{old}} \) \{ Linear Array indices \}

\[ \text{while } \alpha_{\text{current}} < 1.0 \text{ do} \]
\[ \quad \text{if } \alpha_{y, \text{next}} < \alpha_{x, \text{next}} \&\& \alpha_{y, \text{next}} < \alpha_{z, \text{next}} \text{ then} \]
\[ \quad \quad \alpha_{\text{mid}} = (\alpha_{\text{current}} + \alpha_{y, \text{next}})/2 \{ \text{Ray intersects } Y \text{- plane} \} \]
\[ \quad \quad l = \alpha_{y, \text{next}} - \alpha_{\text{current}} \]
\[ \quad \quad \alpha_{\text{current}} = \alpha_{y, \text{next}} \]
\[ \quad \quad \alpha_{y, \text{next}} = \alpha_{y, \text{next}} + \delta \alpha_{y} \{ \text{Potentially next } Y\text{-plane.} \} \]
\[ \quad \quad \text{update } v \]
\[ \quad \quad \text{if } \rho[v_{\text{old}}] \neq 0.0 \text{ then} \]
\[ \quad \quad \quad \text{RPL} = \text{RPL} + \rho[v_{\text{old}}]l \]
\[ \quad \quad \text{end if} \]
\[ \quad \quad \text{update } k \]
\[ \quad \quad \text{end if} \]
\[ \quad \text{else if } \alpha_{x, \text{next}} < \alpha_{z, \text{next}} \text{ then} \]
\[ \quad \quad \alpha_{\text{mid}} = (\alpha_{\text{current}} + \alpha_{x, \text{next}})/2 \{ \text{Ray intersects } X \text{- plane} \} \]
\[ \quad \quad l = \alpha_{x, \text{next}} - \alpha_{\text{current}} \]
\[ \quad \quad \alpha_{\text{current}} = \alpha_{x, \text{next}} \]
\[ \quad \quad \alpha_{x, \text{next}} = \alpha_{x, \text{next}} + \delta \alpha_{x} \{ \text{Potentially next } Z\text{-plane.} \} \]
\[ \quad \quad \text{update } v \]
\[ \quad \quad \text{end if} \]
\[ \quad \text{else} \]
\[ \quad \quad \alpha_{\text{mid}} = (\alpha_{\text{current}} + \alpha_{z, \text{next}})/2 \{ \text{Ray intersects } Z \text{- plane} \} \]
\[ \quad \quad l = \alpha_{z, \text{next}} - \alpha_{\text{current}} \]
\[ \quad \quad \alpha_{\text{current}} = \alpha_{z, \text{next}} \]
\[ \quad \quad \alpha_{z, \text{next}} = (z_{\text{plane}}[k] - z_{1})/(z_{2} - z_{1}) \{ \text{Potentially next } Z\text{-plane.} \} \]
\[ \quad \quad \text{update } v \]
\[ \quad \quad \text{update } k \]
\[ \quad \text{end if} \]
\[ \quad \text{if } \rho[v_{\text{old}}] \neq 0.0 \text{ then} \]
\[ \quad \quad \text{RPL} = \text{RPL} + \rho[v_{\text{old}}]l \]
\[ \quad \text{end if} \]
\[ \quad v_{\text{old}} = v \]
\[ \text{end while} \]
\[ \text{RPL} = d_{12}\text{RPL} \]
Figure 7.6: An example absolute difference map of the RPL for geometry C with the corresponding CT slice (31) for test case II. White, gray, 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 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.