Efficient implementation of the 3D-DDA ray traversal algorithm on GPU and its application in radiation dose calculation

Article *in* Medical Physics · December 2012

DOI: 10.1118/1.4767755 · Source: PubMed

Kai Xiao

University of Notre Dame

10 PUBLICATIONS **39** CITATIONS

SEE PROFILE

Bo Zhou

272 PUBLICATIONS **4,723** CITATIONS

SEE PROFILE

Efficient Implementation of the 3D–DDA Ray Traversal Algorithm on GPU and Its Application in Radiation Dose Calculation

Kai Xiao, Danny Z. Chen, X. Sharon Hu
Department of Computer Science and Engineering
University of Notre Dame, Notre Dame, IN 46556
E–mail: {kxiao, dchen, shu}@nd.edu

Bo Zhou
Department of Radiation Oncology
University of Maryland School of Medicine, Baltimore, MD 21201
E–mail: bzhou@umd.edu

Abstract

Purpose: The three dimensional Digital Differential Analyzer (3D–DDA) algorithm is a widely used ray traversal method, which is also at the core of many convolution/superposition (C/S) dose calculation approaches. However, porting existing C/S dose calculation methods onto Graphics Processing Unit (GPU) has brought challenges to retaining the efficiency of this algorithm. In particular, straightforward implementation of the original 3D–DDA algorithm inflicts a lot of branch divergence which conflicts with the GPU programming model and leads to sub–optimal performance. In this paper, an efficient GPU implementation of the 3D–DDA algorithm is proposed, which effectively reduces such branch divergence and improves performance of the C/S dose calculation programs running on GPU.

Methods: The main idea of the proposed method is to convert a number of conditional statements in the original 3D–DDA algorithm into a set of simple operations (e.g., arithmetic, comparison and logic) which are better supported by the GPU architecture. To verify and demonstrate the performance improvement, this ray traversal method was integrated into a GPU–based Collapsed Cone Convolution/Superposition (CCCS) dose calculation program.

Results: The proposed method has been tested using a water phantom and various clinical cases on an NVIDIA GTX570 GPU. The CCCS dose calculation program based on the efficient 3D–DDA ray traversal implementation runs *1.42~2.67*X faster than the one based on the original 3D–DDA implementation, without losing any accuracy.

Conclusions: The results show that the proposed method can effectively reduce branch divergence in the original 3D–DDA ray traversal algorithm and improve the performance of the CCCS program running on GPU. Considering the wide utilization of the 3D–DDA algorithm, various applications can benefit from this implementation method.

Key Words: GPU, 3D–DDA ray traversal, branch divergence, dose calculation, CCCS.

I. INTRODUCTION

In modern radiotherapy treatment planning, the convolution/superposition (C/S) method can be viewed as a standard for dose calculation algorithms[1]. Recently, the Graphic Processing Unit (GPU) has become an effective platform for accelerating radiation dose calculation which is a computationally expensive process[2]. Many C/S dose calculation algorithms, including Collapsed Cone Convolution/Superposition[3] (CCCS) and Monte Carlo Convolution/Superposition[4] (MCCS) based methods, have been ported onto GPU and showed impressive performance improvement[5, 6, 7].

One vital module in the C/S dose calculation algorithms is ray traversal, which computes the traversing trajectories of energy particles through a given region and their exact radiological path lengths. In this module, the traversal of an individual particle is treated as one ray. From its source point (or the entrance point for a region of interest), the sequence of voxels a ray penetrates and the radiological length it traverses inside each voxel are computed, in order to calculate the locations of the voxels and establish the amount of radiation energy delivered. The total number of ray traversals performed by a C/S dose calculation algorithm (e.g., the Monte Carlo C/S method[8]) for a clinical case can be as many as millions, which accounts for a large proportion of the total computation time for the radiotherapy treatment planning process. Hence, a fast and accurate ray traversal method on GPU is a key to boost performance.

A widely used ray traversal algorithm for C/S dose calculation is the 3D–DDA algorithm[9]. As an efficient voxel space traversal method, the 3D–DDA algorithm has been adopted by a number of clinical dose calculation software packages, such as the one in the Panther system[10] (Prowess, Chico, CA). In C/S dose calculation, the 3D–DDA algorithm iterates through the voxels along the ray's traversal path. As the ray passes each voxel, the voxel index and the radiological length are computed for determining the Total Energy Released per unit Mass (TERMA) as well as depositing energy.

A number of different versions of the 3D–DDA algorithm have been proposed. For example, Stolte et al.[11] adopted additional logical (e.g., masks and negative notations) and fixed–point arithmetic operations to improve the traversal accuracy and speed. Stephenson et al.[12] used an iterative technique based on what they called "runs" (a run is a set of contiguous voxels with the same coordinate value in one direction) in order to improve the efficiency of ray traversal for long paths. Fox et al.[13] proposed a method of transforming the major axis of the voxel grid to align with the ray direction, which reduces the number of iterations during ray traversal. However, the previous development mainly focused on methods for decreasing the number of iterations or instructions. A critical issue which considerably affects performance on GPU is that the 3D–DDA algorithm contains a group of nested conditional statements in its inner–loop and this has received little attention previously since its impact on performance in CPU environment is not significant.

As a method of computing the exact radiological paths, Siddon's algorithm[14, 15] can also be used in C/S dose calculation. For every ray, instead of performing the stepping logic iteratively as 3D–DDA, Siddon's algorithm pre–computes an ordered "distance" array for each axis, where an individual element in each array represents the distance from the ray's source point to every boundary plane along the corresponding axis. The arrays for the three axes are then merged into a "reference" array whose elements are sorted in the increasing order of the distances between the source point and boundary planes. Therefore, the radiological path information of a ray can be directly read from its reference array during dose calculation. However, unless a number of rays have the same stepping sequence and share the same reference array, storing a reference array for each ray requires a huge amount of memory space, especially when the number of rays and grid size are large. With the limited memory capacity, most GPUs do not have sufficient memory for implementing this algorithm. As illustrated in the work of de Greef et al. [16], Siddon's algorithm needs to be rewritten by using a stepping approach in order to port the C/S dose calculation based on it onto GPU. Such "rewriting" results in a structure of nested conditional statements which is quite similar to the one in the 3D–DDA method, as shown in the appendix of de Greef et al.'s paper[16].

To improve the GPU based C/S dose calculation performance, several groups have investigated a few methods for optimizing memory access patterns generated during ray traversal[17, 18]. They explored methods for modifying and scheduling ray distribution and data alignment in order to reduce the number of off–chip memory transactions by coalescing, and for utilizing specific GPU features such as cache hierarchy to reduce the latency of memory transactions. However, the performance of ray traversal modules was not considered explicitly in these studies.

Our investigation shows that the execution of the original 3D–DDA ray traversal method on GPU is not very efficient, because it contains a large number of conditional statements, such as "if–else", which cause branch divergence and degrade performance[19, 20]. Note that in the Single–Instruction–Multiple–Data (SIMD) architecture of NVIDIA GPUs, 32 consecutive threads in the same block (i.e. ThreadIndex[0, .. ,31]) form a "warp" and share the same instruction dispatching unit. Ideally, threads in a warp achieve the best performance when they execute the same instruction flow simultaneously. When a warp of threads encounters a conditional statement and takes different execution branches (e.g. true/false from an "if" statement), they issue separate instructions following the different branches in the code. The instruction unit assigned to such a warp then sequentially issues and dispatches corresponding instructions for each subset of divergent threads, which turns parallel execution of threads in a warp into serial execution. Such a situation is called *branch divergence*. To alleviate the overhead of branch divergence, NVIDIA provides a branch predication mechanism which schedules execution of every instruction controlled by a conditional structure with a per–thread condition code (referred to as predicate). Only instructions with true predicates are actually executed and those with false predicates cannot write results, evaluate addresses or read operands. Nonetheless, such a mechanism cannot be applied to branches containing more than 4 to 7 instructions or nested conditions since the execution cost can be too high in these scenarios[21]. Our observation reveals that the heavily–used conditional statements in the 3D–DDA algorithm are nested and often have large numbers of instructions which can seriously deteriorate its performance on GPU.

In this paper, we present an alternative method to implement the 3D–DDA algorithm for reducing branch divergence. The proposed method, referred to as 3D–DDA–nc, replaces a number of conditional statements in the original 3D–DDA algorithm by a set of arithmetic, comparison and logical operations. Since this new set of operations contains no branches, they utilize the GPU's underlying SIMD feature more effectively and hence improve the ray traversal efficiency. We have integrated the 3D–DDA–nc method into a GPU–based CCCS dose calculation program and tested it on various clinical cases. The results show that the CCCS program based on our 3D–DDA implementation is *1.42~2.67*X faster than that based on the original version. Our improved 3D–DDA implementation can also be applied to other applications involving ray traversal (e.g., graphics ray tracing) where branch divergence causes significant performance degradation on GPU.

II. METHODS

Ray traversal is a procedure of computing the propagation path of a ray in a given region. A ray is uniquely defined by two vectors S and Dir, where S is the source point and Dir is the ray's direction. A point on the ray is represented by $S + T \times Dir$, where the variable T ($T \geq 0$) is the distance from the source to that point. Below, we first briefly review the original 3D–DDA ray traversal algorithm and then present our 3D–DDA–nc method.

Algorithm 1 summarizes the essential operations in the original 3D–DDA algorithm. The first part (Lines 1–14) performs all necessary initialization while the second part (Lines 15–33) conducts the actual ray traversal iteratively voxel by voxel. A vector, $Step$, is introduced such that each element in $Step$ represents the positive or negative index change when the ray crosses a voxel and is initialized to *–1* or *1* (Line 5). When the ray is parallel to any grid axis, the corresponding element of $Step$ is set to *0*. Vector $DeltaT$ represents the distance along the ray when it traverses from one boundary plane to the next parallel boundary plane (Lines 12–14). The elements of vector T are used to store the path length along the ray from S to the next boundary to be evaluated (initialized in Line 7). Therefore, given voxel $CurrentV$, the 3D–DDA algorithm first finds the boundary plane on which the ray exits the voxel $CurrentV$ by identifying the minimum element in T (Lines 16, 17, 25). Then, the minimum element of T is increased by $DeltaT$ (Lines 18, 21, 29). Finally, the corresponding voxel index is incremented by $Step$ to update $CurrentV$ to the next voxel (Lines 19, 22, 30). An example of applying the 3D–DDA algorithm to traverse a ray in the 2–D space is illustrated in Figure 1. Note that when the ray is parallel to any grid axis, there is no voxel index change along that direction. In this case, the corresponding element in T is set to a pre–defined large enough value, indicating that the ray does not intersect any boundary plane along this direction (Lines 8–9).

As shown in Algorithm 1, the procedure of finding the minimum element of T involves a group of nested “if–else” statements (Lines 16–32). Considering the fact that one thread handles a ray at one time in most GPU based ray traversal applications, directly implementing the original 3D–DDA algorithm leads to the possibility that the threads in a warp take different branches and result in branch divergence. The frequency and seriousness of such divergence depend on a number of factors, including the distribution of rays (e.g., their source points and directions) and characteristics of the traversing region (e.g., the region size and voxel locations). For example, a warp of threads handling the traversal of rays whose source points and directions differ significantly would most likely introduce branch divergence and deteriorate the GPU execution performance.

We propose a different implementation for the original 3D–DDA algorithm in order to eliminate the conditional statements used for computing the minimum element of T. Instead of checking and comparing the elements of T by using conditional statements, our implementation, referred to as 3D–DDA–nc, maintains a vector *VoxelIncr* through a set of comparison and logical operations on T. Each element in *VoxelIncr* serves as a flag, indicating whether there is an index change along the corresponding grid axis when the ray exits the voxel *CurrentV*. Specifically, *VoxelIncr.x* is defined as follows:

$$VoxelIncr.x = (T.x \le T.y)\,\&\&(T.x \le T.z) \tag{1}$$

VoxelIncr.y and *VoxelIncr.z* are defined in a similar manner. Note that Equation (1) does not require any conditional statement and can be evaluated by comparisons (e.g., “less–or–equal”) and logical operations (e.g., “and”).

The observation below forms the basis for the 3D–DDA–nc method.

Observation 1: *An element in VoxelIncr has a value “1” if and only if the corresponding element in T is the minimum.*

The correctness of Observation 1 follows immediately from Algorithm 1. For instance, if $T.x <= T.y$ and $T.x <= T.z$, then by Algorithm 1, we know that the ray must exit *CurrentV* from its *x* boundary and *VoxelIncr.x* is set to *1*.

Since a ray may have negative directions, to support both increment and decrement of a voxel index, the *Step* vector as used in Algorithm 1 is also adopted in our new implementation. Hence, $CurrentV.x$ can be updated during each step of ray traversal as follows:

$$CurrentV.x += VoxelIncr.x \times Step.x \tag{2}$$

$CurrentV.y$ and $CurrentV.z$ can be computed in the similar manner. Furthermore, $T.x$ can be calculated from $VoxelIncr.x$ and $DeltaT.x$ as follows:

$$T.x += VoxelIncr.x \times DeltaT.x \tag{3}$$

Similarly, $T.y$ and $T.z$ can be computed.

We summarize the 3D–DDA–nc method in Algorithm 2. Note that the initialization part (Lines 1–14) of Algorithm 2 is the same as that of Algorithm 1 while the iterative part in Algorithm 2 uses arithmetic, comparison and logical operations instead of conditional statements. The correctness of Algorithm 2 is easy to verify. As an example, consider a ray leaving from a voxel's negative *x* boundary face. According to 3D–DDA–nc, *Step.x* is set to *–1*. Since $T.x$ is the minimum element in T, vector *VoxelIncr* is set to *[1, 0, 0]* in Lines 17–19. Then, vector *CurrentV* is changed by *[–1, 0, 0]*, which means that the next voxel's index is one less in the *x* direction than that of the voxel from which the ray is leaving.

For the situations that a ray exits a voxel from one of its edges or corners, some elements of T are equal. In such a case, multiple elements of *VoxelIncr* are updated to *1*, and our 3D–DDA–nc method still computes the next voxel correctly. For example, suppose a ray exits from a corner of a voxel at which the voxel's negative *x*, positive *y*, and negative *z* faces join. Then vector *VoxelIncr* is *[1, 1, 1]* and *Step* is *[–1, 1, –1]*, yielding the change of the vector *CurrentV* as *[–1, 1, –1]*. Note that for this particular case, 3D–DDA–nc computes the next voxel with only one iteration (Lines 17–25 in Algorithm 2). In comparison, for the original 3D–DDA shown in Algorithm 1, three iterations are used. The fact that 3D–DDA–nc naturally supports the special cases of rays passing through edges or corners makes ray traversal based on our method more efficient.

With our new implementation, the conditional statements in the original 3D–DDA method (Lines 16–32 of Algorithm 1) are replaced by the non–conditional operations in Lines 17–25 of Algorithm 2. This replacement requires 21 additional instructions (6 comparisons, 3 logical operations, 3 integer–to–floating–point conversions, 3 floating–point multiple–and–adds, 3 integer multiplications and 3 integer additions) and 3 additional registers. This additional overhead represents a small portion of the total numbers of instructions and registers used in most real–world applications. For example, in the CCCS application considered in this paper, thousands of instructions are used and 46 registers are needed. The performance improvements gained by reducing branch divergence outweigh the additional overhead, as to be shown by the experimental results. This conclusion applies to any scenarios where branch divergence causes significant performance degradation due to serialization of costly operations such as memory reads/writes.

III. RESULTS

To evaluate the effectiveness of our 3D–DDA–nc algorithm, we implemented two versions of the GPU–based CCCS program according to Ahnesjö's paper[3], one following the original 3D–DDA algorithm and the other adopting our 3D–DDA–nc method. The original 3D–DDA based CCCS program has been used in the work of Zhou et al.[18], and its C–version variant has been used in a commercial product of Prowess Panther TPS. Both implementations used 384 kernel ray directions and were tested on NVIDIA GTX570 (Fermi architecture, 480 cores, 1.6GHz core frequency). All programs were developed under the NVIDIA CUDA v4.0 environment and the performance data were collected by NVIDIA Compute Profiler v4.0.

To use the 3D–DDA algorithm in CCCS, the geometric path length traversed by a ray inside each voxel needs to be determined in order to calculate the radiological path length. In the original 3D–DDA algorithm, this is done by storing in variable *minT* the minimum element of *T* selected by the conditional structure. The difference between the *minT* values of the current and previous iterations is the geometric path length traversed by the ray inside the current voxel. In 3D–DDA–nc, determining the minimum elements can be done by the CUDA's built–in instruction *fmin(x,y)*, which returns the minimum of *x* and *y*. The actual code is listed below. When applied to CCCS, these lines of code are inserted between Line 19 and Line 20 of Algorithm 2.

```
minTcurrent = fmin(T.x, fmin(T.y, T.z) );
geoLength = minTcurrent – minT;
minT = minTcurrent;
```

Performance comparisons were conducted on a water phantom and six clinical cases as shown in Table 1. It is worth to mention that, the CCCS program used in this test includes an important implementation detail to avoid performing ray traversal and dose calculation operations when the rays have no energy to release. With this implementation, the ray traversal starts at the field boundary instead of the image boundary, and terminates when the ray exits the image or completely releases its energy. Since the execution time of CCCS dose calculation is proportional to the number of beams involved, we tested only one beam aiming at the center of the phantom for each case. Figure 2 summarizes the execution times of the two implementations, showing that a speedup of ***1.42~2.67*X** is achieved. We further evaluated the accuracy of the dose results computed by both implementations. The results show that our modification to the ray traversal method has no impact on the accuracy of the calculated dose results, which is expected since 3D–DDA–nc does not affect the rays' traversal paths but only changes the method of computing them.

Although the execution times for the test cases that we presented are short, it is important to point out that a typical treatment plan commonly involves multiple beams (e.g., 5–9 beams for an IMRT plan, and up to several hundred beams for a rotational delivery plan). Thus, the reduction in execution time by half or more offers considerable improvement in real treatment planning scenarios, especially when no accuracy comprise occurs.

In order to identify the key contributor to the observed performance improvement, we also conducted profiling on both implementations to collect internal performance data for further analysis. These data are shown in Table 2, which include the total number of branches, the number of divergent branches, the total number of executed instructions, and the memory throughputs.

- Columns 3 and 4 of Table 2 show the numbers of branches and divergent branches in the execution of each test case, respectively. As shown in the table, our 3D–DDA–nc method can effectively **reduce the total branches and divergent branches** of the CCCS program on GPU (**by *4.2~8.6*X**).
- Column 5 compares the total number of executed instructions. The total number of instructions executed in CCCS based on 3D–DDA–nc is ***3.7~5.1*X less** than that

based on the original 3D–DDA. Since branch divergence forces the GPU to execute some instructions sequentially, the decrease in branch divergence by 3D–DDA–nc helps processor cores to share the instruction flow and execute in parallel, which leads to the reduction in the number of executed instructions.

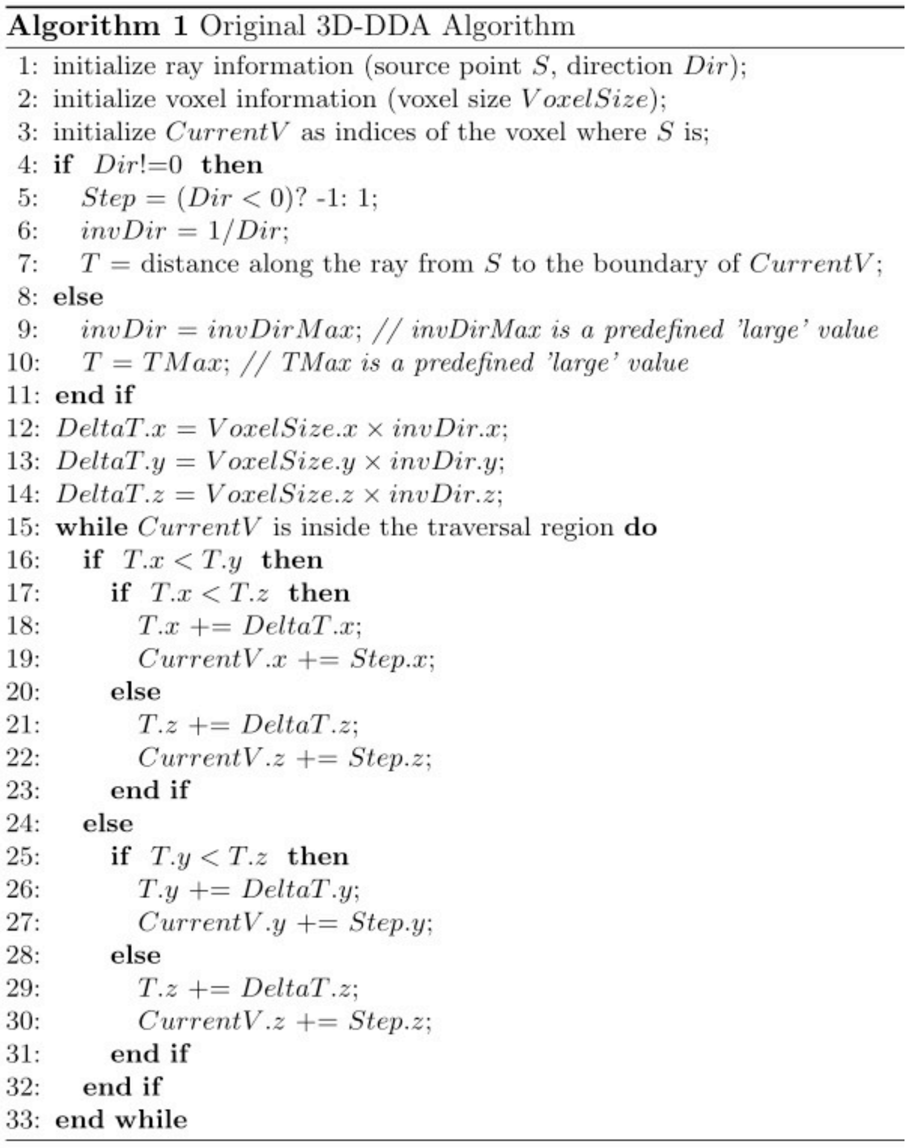
- Columns 6–8 of Table 2 summarize the memory read/write/overall throughputs for our test cases. The CCCS algorithm is well known to be memory bounded for GPU implementations[18]. Therefore, any performance improvement on CCCS must be accompanied by off–chip (DRAM) memory throughput improvement. As shown in the table, the 3D–DDA–nc method **improves the read throughput by *2.1~4.2*X** because the enhanced parallel execution from our method provides more opportunities for the GPU's memory controller to explore memory locality. However, the write throughput is reduced (as shown in Column 7) due to the increase in simultaneous write operations to the same addresses for dose deposition.

In our implementation, atomic writes are used to maintain coherence for simultaneous writes to the same memory location. With the improved parallel execution by 3D–DDA–nc, more write requests are generated simultaneously, and thus more atomic operations are required. This conflicting fact limits the level of speedup ultimately achieved. However, since atomic operations are quite efficient in the latest GPU architecture and the number of write requests is much less than that of read requests, our method still results in an overall memory throughput improvement. As shown in Column 8, the **overall memory throughput** of the CCCS execution with the 3D–DDA–nc method is ***1.4~2.7*X faster** than that with the original method. Note that, for those CCCS implementations in which atomic writes are avoided (such as that by Chen et al.[17]), our method for reducing branch divergence could lead to even bigger performance improvement.

IV. CONCLUSIONS

We present a simple yet effective method for implementing the 3D–DDA ray traversal algorithm on GPU by replacing a number of conditional statements with arithmetic, comparison and logical operations. Our method reduces the number of divergent branches during execution, hence improving the performance of 3D–DDA on GPU. The experimental results for several clinical cases demonstrate that on a state–of–the–art GPU platform, a CCCS program based on the improved 3D–DDA algorithm can attain around ***2*X** performance speedup from our modification without losing any dose accuracy on the tested clinical cases.

The proposed implementation can be readily applied to other 3D–DDA based applications, including graphics ray tracing and 3D animations. The actual achievable performance improvement for a specific application depends on various factors, such as the operations to be executed for every traversal step and the types of instructions following the conditional statements. In general, for applications where expensive instructions (e.g., memory accesses) follow the conditional statements of the 3D–DDA, our implementation can provide non–trivial performance benefits. For example, in graphics ray tracing, each step of traversing a voxel requires to access the object information that the voxel contains. Reducing branch divergence in this application will improve memory read throughput similar to that in the CCCS application (see Column 6 of Table 2). In addition, for many GPU applications containing massive conditional statements (e.g., image reconstruction, viewing transformation and volume visualization), the concept of replacing branches described in this work should also be helpful in reducing branch divergence and its negative impact on performance.



Algorithm 1 Original 3D-DDA Algorithm

```
 1: initialize ray information (source point $S$, direction $Dir$);
 2: initialize voxel information (voxel size $VoxelSize$);
 3: initialize $CurrentV$ as indices of the voxel where $S$ is;
 4: if $Dir$!=0 then
 5:   $Step = (Dir < 0)$? -1: 1;
 6:   $invDir = 1/Dir$;
 7:   $T$ = distance along the ray from $S$ to the boundary of $CurrentV$;
 8: else
 9:   $invDir = invDirMax$; // invDirMax is a predefined 'large' value
10:   $T = TMax$; // TMax is a predefined 'large' value
11: end if
12: $DeltaT.x = VoxelSize.x \times invDir.x$;
13: $DeltaT.y = VoxelSize.y \times invDir.y$;
14: $DeltaT.z = VoxelSize.z \times invDir.z$;
15: while $CurrentV$ is inside the traversal region do
16:   if $T.x < T.y$ then
17:     if $T.x < T.z$ then
18:       $T.x$ += $DeltaT.x$;
19:       $CurrentV.x$ += $Step.x$;
20:     else
21:       $T.z$ += $DeltaT.z$;
22:       $CurrentV.z$ += $Step.z$;
23:     end if
24:   else
25:     if $T.y < T.z$ then
26:       $T.y$ += $DeltaT.y$;
27:       $CurrentV.y$ += $Step.y$;
28:     else
29:       $T.z$ += $DeltaT.z$;
30:       $CurrentV.z$ += $Step.z$;
31:     end if
32:   end if
33: end while
```

Algorithm 1. The original 3D–DDA Algorithm[9].

Algorithm 2 The 3D-DDA-nc method

```
1: initialize ray information (source point $S$, direction $Dir$);
2: initialize voxel information ( voxel size $VoxelSize$);
3: initialize $CurrentV$ as indices of the voxel where $S$ is;
4: if $Dir$!=0 then
5:     $Step = (Dir < 0)$? -1: 1;
6:     $invDir = 1/Dir$;
7:     $T$ = distance along the ray from $S$ to the boundary of $CurrentV$;
8: else
9:     $invDir = invDirMax$; // invDirMax is a predefined 'large' value
10:    $T = TMax$; // TMax is a predefined 'large' value
11: end if
12: $DeltaT.x = VoxelSize.x \times invDir.x$;
13: $DeltaT.y = VoxelSize.y \times invDir.y$;
14: $DeltaT.z = VoxelSize.z \times invDir.z$;
15: $VoxelIncr = 0$;
16: while $CurrentV$ is inside the traversal region do
17:     $VoxelIncr.x = (T.x <= T.y)$ && $(T.x <= T.z)$ ;
18:     $VoxelIncr.y = (T.y <= T.x)$ && $(T.y <= T.z)$ ;
19:     $VoxelIncr.z = (T.z <= T.x)$ && $(T.z <= T.y)$ ;
20:     $T.x$ += $VoxelIncr.x \times DeltaT.x$;
21:     $T.y$ += $VoxelIncr.y \times DeltaT.y$;
22:     $T.z$ += $VoxelIncr.z \times DeltaT.z$;
23:     $CurrentV.x$ += $VoxelIncr.x \times Step.x$;
24:     $CurrentV.y$ += $VoxelIncr.y \times Step.y$;
25:     $CurrentV.z$ += $VoxelIncr.z \times Step.z$;
26: end while
```

Algorithm 2. The 3D–DDA–nc method.

Test Case	Image Size *	Field Size *	Voxel Size (mm) **
Water Phantom	200×200×200	50× 50	1.0×1.0×1.0
Breast A	512×512×168	125×125	0.5×0.5×3.0
Breast B	512×512× 63	110×110	1.0×1.0×1.0
Lung A	512×512×112	125×125	1.0×1.0×3.0
Lung B	512×512×111	110×110	1.2×1.2×3.0
Head&neck A	512×512×144	125×125	1.2×1.2×3.0
Head&neck B	384×384× 58	90× 90	0.6×0.6×3.0

* Number of voxels in each dimension

** Geometric length of a voxel in each dimension

Table 1. Configuration of the test cases.

CT Image	Method	Branch	Divergence	Instruction	Memory Throughput (GB/s)		
					Read	Write	Overall
Water phantom	Original	4.78E+08	*9.07E+05*	4.13E+09	*13.05*	10.07	23.12
	3D–DDA–nc	1.81E+08	**2.23E+05**	1.73E+09	**22.74**	6.78	28.52
BreastA	Original	8.62E+09	*1.02E+07*	5.87E+10	*6.93*	4.09	11.02
	3D–DDA–nc	9.91E+08	**1.72E+06**	1.14E+10	**28.31**	2.57	30.89
BreastB	Original	4.07E+09	*2.85E+06*	3.29E+10	*26.57*	10.95	37.53
	3D–DDA–nc	6.83E+08	**5.59E+05**	7.50E+09	**72.78**	4.55	77.34
LungA	Original	6.74E+09	*6.64E+06*	4.60E+10	*13.13*	6.71	19.85
	3D–DDA–nc	9.52E+08	**1.35E+06**	1.08E+10	**42.67**	3.25	45.92
LungB	Original	6.03E+09	*6.09E+06*	4.26E+10	*14.05*	7.15	21.20
	3D–DDA–nc	8.61E+08	**1.21E+06**	9.73E+09	**44.11**	3.22	47.34
Head&neck A	Original	6.62E+09	*6.80E+06*	3.20E+10	*15.04*	7.29	22.43
	3D–DDA–nc	9.68E+08	**1.57E+06**	8.79E+09	**32.34**	4.15	36.49
Head&neck B	Original	3.45E+09	*4.71E+06*	2.20E+10	*7.52*	4.74	12.26
	3D–DDA–nc	5.12E+08	**1.17E+06**	5.90E+09	**15.49**	1.66	17.15

Table 2. Detailed execution data comparison between CCCS implementations based on the original 3D–DDA and 3D–DDA–nc.

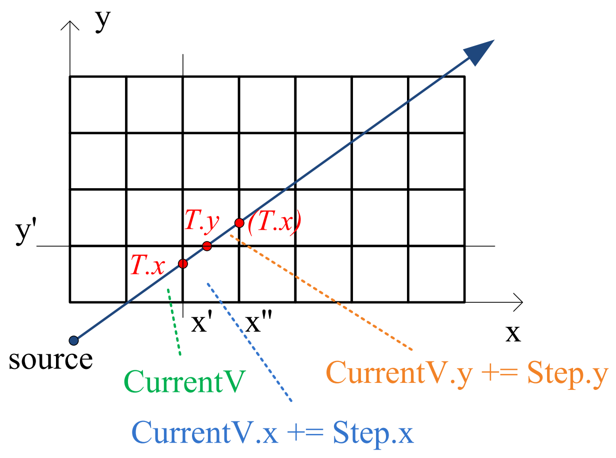


Figure 1. A 3D–DDA ray traversal example in the 2–D space. The 3D–DDA algorithm checks the two planes *x*' and *y*' to determine the exit point of the ray from voxel *CurrentV*, by computing the distances from the source to *x*' and to *y*' as *T.x* and *T.y* and finding the minimum as the exit point. Starting from *CurrentV*, in the first step, the exit point is on the boundary of *x*'. So *CurrentV.x* is increased and *T.x* is updated onto the next boundary of *x*''; in the second step, the exit point is on y', and thus *CurrentV.y* is increased.

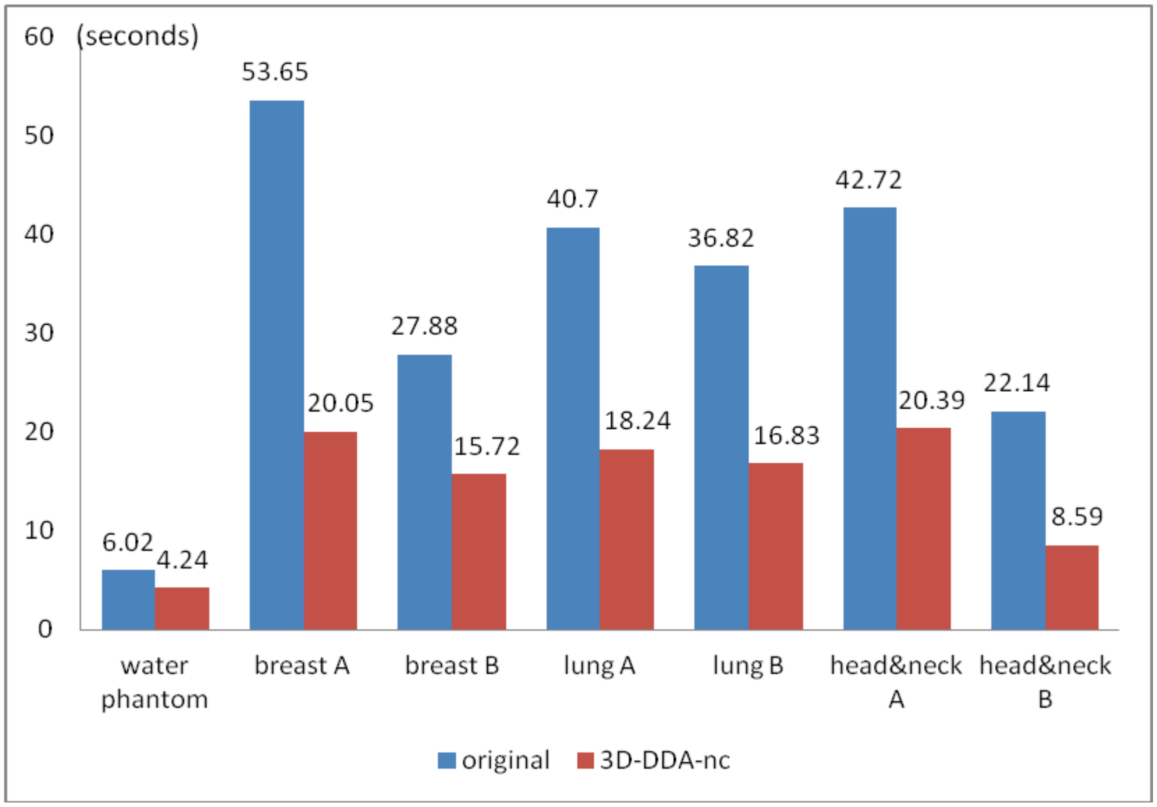


Figure 2. Total execution time comparison between the CCCS implementations based on the original 3D–DDA and 3D–DDA–nc for the water phantom and six clinical test cases. The speedup factors are 1.42– 2.67. In terms of the per processed voxel execution time, the original 3D–DDA has a range of 5.43 ns – 9.05 ns, and the 3D–DDA–nc has 3.36 ns – 4.23 ns for the clinical test cases. The variance in the speedup factors and per–voxel execution time is mainly due to the memory layout of test images and additional atomic writes introduced by the increased parallel executing threads in 3D–DDA–nc.

ACKNOWLEDGEMENT

The research of D.Z. Chen was supported in part by NSF under Grant CCF-0916606.

REFERENCES

[1] W. Lu, G. Olivera, M. Chen, P. Reckwerdt, and T. Mackie, "Accurate convolution/superposition for multi–resolution dose calculation using cumulative tabulated kernels," *Phys. Med. Biol.*, 50(4), 2005.

[2] S. Hissoiny, B. Ozell, H. Bouchard, and P. Despres, "GPUMCD: A new GPU–oriented Monte Carlo dose calculation platform", *Med. Phys.,* 38(2), 2011.

[3] A. Ahnesjö, "Collapsed cone convolution of radiant energy for photon dose calculation in heterogeneous media," *Med. Phys.*, 16(4), 1989.

[4] S. Naqvi, M. Earl, and D. Shepard, “Convolution/superposition using the Monte Carlo method,” *Phys. Med. Biol*., 48(14), 2003.

[5] R. Jacques, R. Taylor, J. Wong, and T. McNutt, “Towards real–time radiation therapy: GPU accelerated superposition/convolution,” *Computer Methods and Programs in Biomedicine,* 98(3), 2010.

[6] S. Hissoiny, B. Ozell and P. Despres, “A convolution–superposition dose calculation engine for GPUs,” *Med. Phys*., 37(3), 2010.

[7] R. Jacques, J. Wong, T. McNutt, and R. Taylor, “Read–time dose computation: GPU–accelerated source modeling and superposition/convolution”, *Med. Phys*., 38(1), 2011.

[8] B. Zhou, X.S. Hu, D.Z. Chen, and C.X. Yu, “GPU–accelerated Monte Carlo convolution/ superposition implementation for dose calculation,” *Med. Phys*., 37(11), 2010 .

[9] J. Amanatides and A. Woo, “A fast voxel traversal algorithm for ray tracing,” *Eurographics*, 3(10), 1987.

[10] T. Knoos, E. Wieslander, L. Cozzi, C. Brink, A. Fogliate, D. Albers, H. Nystrom, and S. Lassen, “Comparison of dose calculation algorithms for treatment planning in external photon beam therapy for clinical situations,” *Phys. Med. Biol*., 51(22), 2006.

[11] N. Stolte, and R. Caubet, “Discrete ray–tracing of huge voxel spaces”, *Computer Graphics Forum*, 14(3), 1995.

[12] P. Stephenson, and B. Litow, “Making the DDA run: Two–dimensional ray traversal using runs and runs of runs”, *Processings of the 24th Australasian Computer Science Conference*, 177–183, 2001.

[13] C.Fox, H. Romeijn, and J. Dempsey, “Fast voxel and polygon ray–tracing algorithms in intensity modulated radiation therapy treatment planning”, *Med, Phys*., 33(5), 2006.

[14] R. Siddon, “Fast calculation of the exact radiological path for a three–dimensional CT array,” *Med. Phys*, 12(3), 1985.

[15] F. Jacobs, E. Sunderman, B. De Sutter, M. Chrisiaens, and I. Lemahieu, “A fast algorithm to calculate the exact radiological path through a pixel or voxel space”, *Journal of Computing and Information Technology*, 6(1), 1998.

[16] M. de Greef, J. Crezee, J. C. van Eijk, R. Pool, and A. Bel, “Accelerated ray tracing for radiotherapy dose calculations on a GPU,” *Med. Phys*., 36(9), 2009.

[17] Q. Chen, M. Chen, and W. Lu, “Ultrafast convolution/superposition using tabulated and exponential kernel,” *Med. Phys*., 38(3), 2011.

[18] B. Zhou, X.S. Hu, and D.Z. Chen, “Memory–efficient volume ray tracing on GPU for radiotherapy,” *IEEE 9th Symposium on Application Specific Processors*, 5(6), 2011.

[19] T. Aila and S. Laine, “Understanding the efficiency of ray traversal on GPUs,” *Proceedings of the Conference on High Performance Graphics*, 2009.

[20] T.D. Han and T.S. Abdelrahman, “Reducing branch divergence in GPU programs,” *Proceedings of the Fourth Workshop on General Purpose Processing on Graphics Processing Units*, 2011.

[21] NVIDIA Corp., “NVIDIA CUDA Compute Unified Device Architecture,” *Programming Guide*, 2011.