A GPU Implementation of a Shooting and Bouncing Ray Tracing Method for Radio Wave Propagation

Dan Shi, Xiaohe Tang, Chu Wang, Ming Zhao, and Yougang Gao

School of Electronic Engineering
Beijing University of Posts and Telecommunications, Beijing, 100876, China
shidan@buptemc.com

Abstract ─ Shooting and bouncing ray tracing method (SBR) is widely adopted in radio wave propagation simulations. Compared with the center-ray tube model, the lateral-ray tube model is more accurate but more time consuming. As a result, we use graphics processing unit (GPU) to accelerate the lateral-ray tube model. In this paper, we proposed a GPU-Based shooting and bouncing lateral-ray tube tracing method that is applied to predicting the radio wave propagation. The numerical experiment demonstrates that the GPU-based SBR can significantly improve the computational efficiency of lateral-ray tube model about 16 times faster, while providing the same accuracy as the CPU-based SBR. The most efficient mode of transferring the data of triangle faces is also discussed.

Index Terms ─ Compute unified device architecture (CUDA), graphics processing unit (GPU), radio wave propagation, ray tracing, shooting and bouncing ray (SBR).

I. INTRODUCTION

In the past few decades, electromagnetic environment (EME) simulation technology has been growing in its popularity, for it is significant both for military use and for civil use. As a result, various computational electromagnetic methods have been applied in this field. Among all kinds of computational methods, the shooting and bouncing ray (SBR) [1, 2] tracing method is a high frequency asymptotic one for calculating the radio wave propagation through environments with regions of reflecting surfaces, diffracting edges and so on [3]. At present, there are several models proposed, which have their own characteristics. Tube creating can be categorized into two different schemes using center-ray tubes (a ray is shot from the center of the patch wavefront) or lateral-ray tubes (rays are shot from vertices of the patch wavefront), depending on the number of rays chosen to build a tube. The ray cone is a kind of center-ray tube. When the rays transmitted are treated as ray cones, overlap and double counting are unavoidable because of the spherical wavefront during the propagation process [4-7]. But regular polygons such as triangles, squares and hexagons can completely cover an area without leaving gaps or existing overlaps. Using lateral-ray tube tracing methods can get a more accurate result than using center-ray tube tracing methods. However, the cost of tracing lateral-ray tubes is much higher than tracing center-ray tubes [8]. Therefore, we propose to use the graphics processing unit (GPU) to accelerate the shooting and bouncing lateral-ray tube method.

It is obvious that ray tracing is well suitable for parallel processing due to the independence of rays [9]. Carr et al. first implemented the ray-triangle intersection on the GPU in 2002 [10]. Tao used center-ray tube model to trace the valid tubes in the radar cross section (RCS) prediction on the GPU in 2010 [11]. In this paper, abandoning the inaccurate center-ray tube model, we use the lateral-ray tube model and fully implement the shooting and bouncing lateral-ray tube tracing method on the GPU.

This paper is organized as follows. Section II discusses the method of GPU-Based shooting and bouncing lateral-ray tube tracing. In Section III, modeling and implementation details is introduced. In Section IV, the results and discussion are given. Last section is the conclusion.

II. GPU-BASED SBR

GPU is a specialized device that has many cores working together. Typically, every 32 threads compose a warp which is the basic executing unit of the GPU, and the 32 threads execute the same instruction on different data simultaneously [12]. This effectively reduces the memory access delay by 32 times.

In software, a typical compute unified device architecture (CUDA) program consists of two parts. One part is the CPU codes that control the process of the whole program, and the other part is the GPU part that does the parallel work [13]. A function that executes on the GPU is typically called a "kernel" [14].

The procedure of the GPU-Based shooting and bouncing lateral-ray tube tracing method is divided into three steps. They are, generating original ray tubes,

Submitted On: November 27, 2016
Accepted On: June 9, 2017

reflecting calculation, and diffracting calculation. Among all the steps, reflecting calculation and diffracting calculation executes their kernels separately. The details of these steps are discussed in Section II-A through Section II-C.

A. Generating lateral-ray tubes from a transmitter

The transmitter is modeled as a point source, and for purpose of considering all possible angles of departure of rays, a regular icosahedron is inscribed inside the unit sphere. To achieve better resolution, each face of the icosahedron is tessellated into N equal segments where N is the tessellation. Rays are launched through icosahedron vertices and at the intersection points of tessellated triangle faces. Figure 1 shows an example with N = 32, which is used in our model. This method of launching the source rays provides wavefronts that completely subdivide the surface of the unit sphere with nearly equal shape and area [4]. An original ray tube is composed of three adjacent rays as Fig. 1 illustrates.

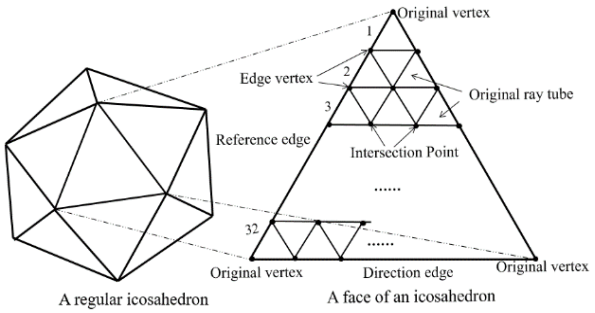

Fig. 1. A regular icosahedron and tessellation of icosahedron face.

B. Intersection tests and reflecting calculation

The CUDA program traces all original ray tubes synchronously. Parallelism is introduced by running main threads scheduling child thread that accomplishes the calculation, and each child thread shows up as a separate process.

The heart of the matter is to distribute the computation to over ten thousand of individual, controllable, and analogous threads. Since every thread is supposed to perform almost the same task, the distribution appears particularly significant, which signifies that we should ensure every distributed computation process resembles each other so that a universal kernel function (consistent input and output, same calculation formula, etc.) can be the template for every child thread. Therefore, we assign a CUDA thread to a single ray tube. A thread merely traces one single ray tube, which ensures the independence and the similarity of different CUDA threads.

The most time-consuming part is the intersection tests of ray tubes as follows:

1) Calculating the reflection point

Any point on the ray can be represented as $\vec{O} + t\vec{r}$ (where $\vec{O}$ represents the original point of the ray, $\vec{r}$ represents the direction vector of the ray, t represents the distance coefficient, if $t > 0$, then it represents the point is in the positive direction) as Fig. 2 shows, and any point inside a triangle face can be represented as $u\overrightarrow{AB} + v\overrightarrow{AC} + \vec{A}$ (where u and v represent the distance coefficient of $\overrightarrow{AB}$ and $\overrightarrow{AC}$, if $0 < u < 1$, $0 < v < 1$, $0 < u + v < 1$, then it represents the point is inside the triangle ABC as Fig. 3 shows:

$$\vec{O} + t\vec{r} = u(\vec{B} - \vec{A}) + v(\vec{C} - \vec{A}) + \vec{A}, \quad (1)$$

$$t\vec{r} - u(\vec{B} - \vec{A}) - v(\vec{C} - \vec{A}) = \vec{A} - \vec{O}. \quad (2)$$

Let $\boldsymbol{\alpha_1} = \vec{r}$, $\boldsymbol{\alpha_2} = \vec{B} - \vec{A}$, $\boldsymbol{\alpha_3} = \vec{C} - \vec{A}$, $\boldsymbol{\beta} = \vec{A} - \vec{O}$, then,

$$\boldsymbol{\alpha_1} t - \boldsymbol{\alpha_2} u - \boldsymbol{\alpha_3} v = \boldsymbol{\beta}.$$

Let $d = |\boldsymbol{\alpha_1}\ \boldsymbol{\alpha_2}\ \boldsymbol{\alpha_3}|$, if $d \neq 0$, on the basis of Cramer's Rule [15]:

$$t = \frac{|\boldsymbol{\beta}\ \ \boldsymbol{\alpha_2}\ \ \boldsymbol{\alpha_3}|}{d}, u = \frac{|\boldsymbol{\alpha_1}\ \ \boldsymbol{\beta}\ \ \boldsymbol{\alpha_3}|}{d}, v = \frac{|\boldsymbol{\alpha_1}\ \ \boldsymbol{\alpha_2}\ \ \boldsymbol{\beta}|}{d}.$$

$$\text{If } 0 < u < 1,\ \ 0 < v < 1 \quad 0 < u + v < 1. \quad (3)$$

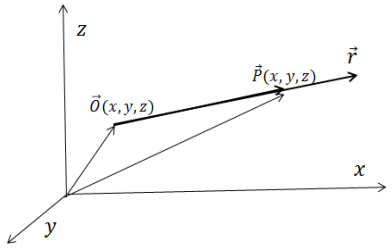

Fig. 2. Point $\vec{P}$ on a ray $\vec{r}$.

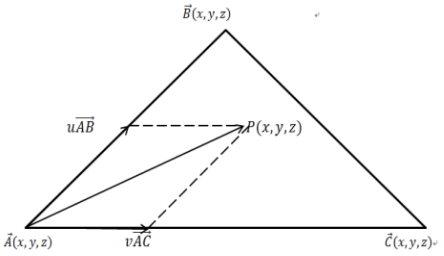

Fig. 3. Point $\vec{P}$ inside a triangle ABC.

Then it represents the intersection point is in the positive direction of the ray and inside the triangle face as well, where t represents the distance between the original point and the reflection point. Loop the computation with all faces, then compare t, intersection point with minimum t value is the reflection point and go to step 2). However, if the result does not meet (3), it represents that the ray has not intersected with the buildings or terrains, and step 2) is supposed to be skipped.

2) Calculating the reflection vector

As is shown in Fig. 4, $\vec{\imath}$ is the normalized incident vector, $\vec{r}$ is the normalized reflection vector, and $\vec{n}$ is the normal vector of a triangle face. Angle of incidence equals to the angle of reflection, therefore quadrangle MONQ is a rhombus, so $\mathrm{OQ} = 2\mathrm{OP}$. On basis of step 1), the coordinate of O is deterministic.

Solve the equation to calculate $\vec{r}$:

$$\vec{r} = \vec{\imath} - 2\overrightarrow{PO} = \vec{\imath} - 2(\vec{\imath} \cdot \vec{n})\vec{n}, \tag{4}$$

where $\vec{n} = \frac{\overrightarrow{AB} \times \overrightarrow{AC}}{|\overrightarrow{AB} \times \overrightarrow{AC}|}$, $\vec{A}$, $\vec{B}$, $\vec{C}$ are the vertices of a triangle face.

3) Case analysis

If neither of a ray of the ray tube has an intersection point (case 0), this ray tube is discarded. If some or all three rays have intersection points (case 1, 2, 3, 4 and 5), it is necessary to consider all kinds of reflection and diffraction cases based on the coordinate of intersection points:

0. three rays do not intersect a building;
1. three rays intersect the same face of a building;
2. one rays intersect a face of a building while the other two do not;
3. two rays intersect the same face of a building while the other one does not;
4. two rays intersect two adjacent face of a building while the other one does not;
5. two rays intersect the same face of a building while the other one intersects an adjacent face of a building.

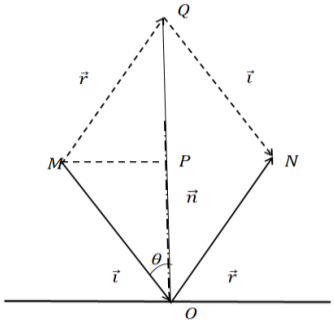

Fig. 4. A ray $\vec{\imath}$ intersects with a plane.

We mainly consider six cases above. It is obvious that the reflection exists in all cases except case 0 while diffraction does not exist in case 0 and 1. Figure 5 shows five cases of ray tubes intersecting the building faces. As for case 2, 3, 4, 5, each thread will calculate the coordinates of diffraction edges.

GPU specializes in tedious repetitive numerical calculation and is weak in dealing with complicated logic structure; hence it is reasonable to run highly intensive computational task on the GPU like solving equations in step 1) and 2). Parallel numerical calculation indicates that when the amount of incident ray tubes is large and the formulas are complex, the acceleration effect is particularly obvious compared with CPU serial programs.

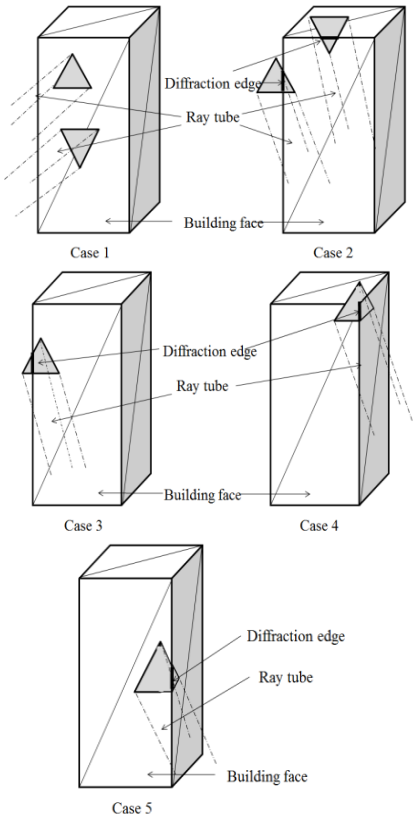

Fig. 5. Five cases that ray tubes intersect with building faces.

C. Generating diffracting rays

Once the wavefront of a ray tube illuminates an edge of two adjacent faces and the two adjacent faces make up a wedge, this ray tube will generate diffraction rays. The following paragraphs will show the procedure.

A single GPU thread represents an incident ray tube to be diffracted. Based on the incident ray tube and the diffraction edge, we can get an original point of the incident ray tube and two intersections of the wavefront of the incident ray tube with diffraction edge. Then, two virtual incident rays are created. Each of them generates a group of diffraction rays. We need to specify the count N of the generated diffraction rays. Suppose the dihedral angle of the two adjacent faces, which can make up a wedge, is θ. The cross section of the circular cone, which is a sector, can be divided into N-1 parts with the angle of $\frac{360-\theta}{\mathrm{N}-1}$. As a result, the N-1 parts are able to construct N-1 ray tubes of which the wavefront is a quadrangle, as is shown in Fig. 6. Considering the consecutive thread ID, we can get three arrays, A[i], B[i] and R[i]. Among

the three arrays, A[i] and B[i] contain the diffraction rays to be generated, and the other one R[i] contains the diffraction ray tubes which consist of the diffraction rays in the first two arrays. As is shown in Fig. 7, we use i to represent the ray tube number. The relationship between diffraction rays and diffraction ray tubes is also shown. Besides, the ray tube number i is in a loop from (idx * N) to (idx+1) * N, in which idx means thread ID.

Different with original ray tubes and reflection ray tubes, the diffraction ray tubes are quadrangle ray tubes, which means each lateral-ray tube consist of four rays.

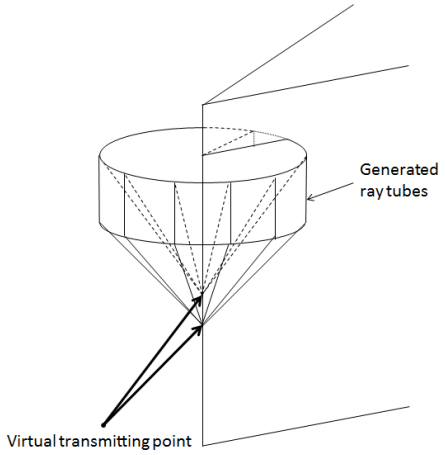

Fig. 6. Generating ray tubes from diffracting edge.

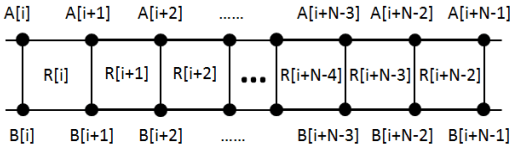

Fig. 7. Data structure of diffraction rays and ray tubes.

III. MODELING AND IMPLEMENTATION

To verify the efficiency of the proposed GPU-based shooting and bouncing lateral-ray tube tracing method, a CPU-based version is also implemented for comparison. A model is established, and several numerical examples are tested. The GPU hardware used in this research is Quadro K620 of Compute Capability 5.0, with 2 GB of memory. The CPU hardware used in this research is Intel(R) Xeon(R) CPU E5-2603 v3 @ 1.60 GHz with 6 cores. Our implementation runs atop Windows 7 with the CUDA Toolkit 7.5. As all future NVIDIA GPUs will support CUDA, the proposed GPU-based SBR is scalable across future generations.

CUDA provides a simple and general C/C++ language interface to the programmers and the programming on GPU does not have much difference from using application programming interfaces.

The GPU-based shooting and bouncing lateral-ray tube tracing method is applied to a sample environment. There are 4 buildings set on the terrain. All the buildings are with the same height, which is 100 m above the terrain. The entire model is made up of 19650 triangle faces. The material parameter of the buildings is $\varepsilon_r = 15$, $\sigma = 0.015$. The material parameter of the ground is $\varepsilon_r = 25$, $\sigma = 0.02$. Considering the architecture of the transmitter is a regular icosahedron, we can estimate the complexity according to the subdivision coefficient. In this sample, we set the subdivision coefficient to 32. As a result, 20480 original triangle ray tubes are generated from the transmitter. There are 2 groups of receivers. One is made up of receivers ranging from (0, 0, 50) to (0, 160, 50) with a 20 - m step. The other group is made up of receivers ranging from (0, 0, 20) to (0, 0, 100) with a 10 - m step. The modeling is shown in Fig. 8.

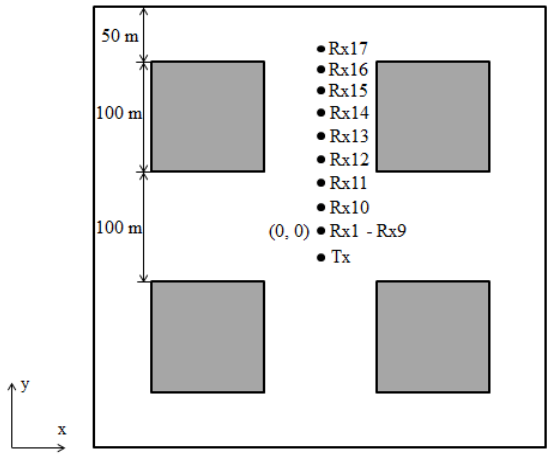

Fig. 8. A sample model of propagation environment.

The data of the terrain and the buildings are imported from the electronic map. All the data necessary in the experiment including the terrain and buildings are saved in the global memory. In CUDA programming, the number of blocks and threads per block is specified by the programmer, and each thread has a unique thread ID and block ID to identify the unique data assigned to each thread. As a result, each lateral-ray tube can be specified through thread ID and block ID. In our experiment, considering the 20480 original triangle ray tubes, the maximum block size and the thread count in per block in our implementation is 32 × 640. In addition, because of the limited device memory, we cannot transfer all the triangle faces data into the GPU. We resolve this problem by transferring the data in batches.

IV. RESULTS AND DISCUSSION

Last part we introduced the modeling, knowing that there are 19650 triangle faces and 20480 original triangle ray tubes. Figure 9 and Fig. 10 show the E field vs. receiver distance and E field vs. receiver height. Both

are compared with the commercial electromagnetic simulation software Wireless Insite.

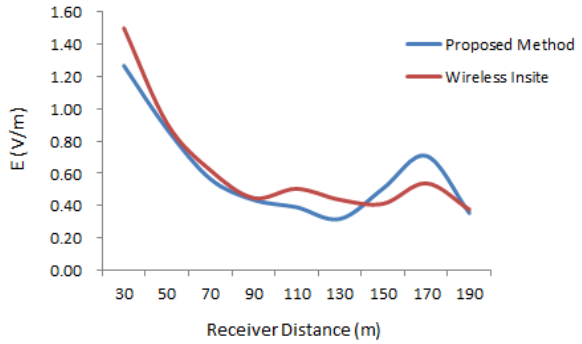

Fig. 9. Comparison of E field vs. receiver distance with proposed method and Wireless Insite simulation.

In Fig. 9, the heights of the transmitter and the receivers are all 50 m. Since the distance between the transmitter and the receivers varies from 30 m to 190 m with a 20 - m step, the E field decreases in general. However, under the influence of the building reflection, the E field decreases slowly even grows a little when the distance becomes longer and longer.

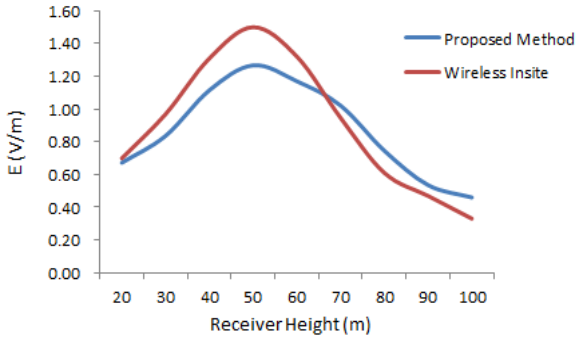

Fig. 10. Comparison of E field vs. receiver height with proposed method and Wireless Insite simulation.

In Fig. 10, the height of the transmitter is 50 m. The horizontal distance between the transmitter and all the receivers is 30 m. Since the height of the receivers varies from 20 m to 100 m with a 10 - m step, the distance between the transmitter and the receivers decreases at first and then increases with the receiver height. As a result, the electric field increases at first for the receiver becomes closer to the transmitter. However, for the receivers above 50 m, the electric field decreases with the increase of receiver height.

From the Table 1, we can get the information that the execution speed on the GPU is more than 16 times higher than CPU.

Table 1: Comparisons between execution time for CPU and GPU

Type	Time
Executing on CPU	140122 ms
Executing on GPU	8706 ms

Below we will put emphasis on analyzing the factors which affect the executing time.

The executing time on the CPU is as follow:

$$t_{CPU} = \frac{N}{f_{CPU}} \times \frac{1}{C_{CPU}} \times \frac{1}{\eta}, \quad (5)$$

where N represents the data scale inputted, f_{CPU} represents the CPU frequency and C_{CPU} represents the CPU's capability of calculation, η represents the efficiency of the algorithm.

We pay more attention to the factors which affect the executing time on the GPU. The formula is shown as follow:

$$t_{GPU} = t_{kernel} + t_{memcpy} = t_n \times \left\lfloor \frac{N}{n} \right\rfloor + t_{N \bmod n} + t_m \times \lceil \frac{N}{n} \rceil, \quad (6)$$

$$t_n = \frac{n}{f_{GPU}} \times \frac{1}{C_{GPU}} \times \frac{1}{\eta} + n \times a, \quad (7)$$

$$t_0 = 0, \quad (8)$$

where t_{GPU} represents the total execution time on the GPU, t_{kernel} represents the execution time cost in the kernel functions, t_{memcpy} represents the time spent on copying data from the GPU to the CPU, N represents the data scale inputted, n represents the data scale inputted per time, $\lfloor \frac{N}{n} \rfloor$ represents rounding down to $\frac{N}{n}$, $\lceil \frac{N}{n} \rceil$ represents rounding up to $\frac{N}{n}$, t_n represents the computation time with n triangle faces transferred, t_m represents the time spent on copying the data once, f_{GPU} represents the GPU frequency, C_{GPU} represents the GPU's capability of calculation, η represents the efficiency of the algorithm, and a is a constant which affects the speed of copying memory changing with different GPUs.

Because of the limited GPU memory, we cannot copy all the data from the CPU to the GPU. As a result, we should divide the data into several parts. Then we copy each part from the CPU to the GPU. We do not copy the second part of the data until the first part of the data has been calculated. So do the rest parts of the data. This is the reason why $\frac{N}{n}$ is in the formula. The GPU frequency influences the memory clock rate so the frequency is higher the more time is saved. Additionally, the capability of the GPU is stronger, the more time is saved.

In the formula (6), $\frac{N}{n}$ is decided by programmers. In our experiment, N depends on the count of triangle faces and the count of original ray tubes. We divide all the 19626 triangle faces into several groups. Meanwhile, we test the time of the intersection test which is the most

time-consuming part of the whole experiment. As is shown in Fig. 11.

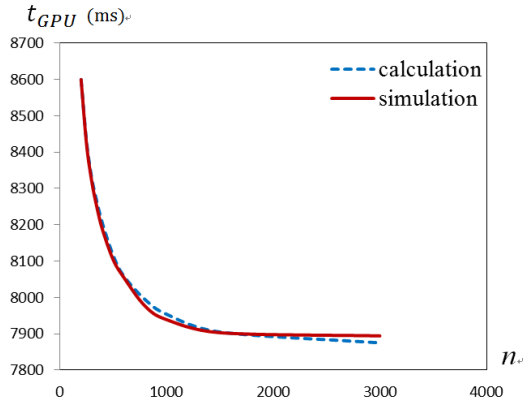

Fig. 11. Computation time on the GPU predicted and measured.

In the formula (6), as the n grows, $t_n \times \lfloor \frac{N}{n} \rfloor + t_{N \bmod n}$, which is t_{kernel}, does not change. t_m changes slightly, too. So the formula (6) mainly depends on $\lfloor \frac{N}{n} \rfloor$. Therefore, we should try our best to get the biggest n to enhance the efficiency. We treat $n = 200$ as a basic unit. Then we predict the calculation time by the formula (6), as is shown in Fig. 11. For our GPU, the biggest n is up to 4500. If $n > 4500$, there will not be enough space to save the data. So in the example of the comparisons of the CPU and the GPU, the n of the GPU is chosen as 4500.

For the efficiency of algorithm, we can use the shared memory to store the triangle face information instead of global memory to save time. Proper distribution way of blocks and threads also reduces the total execution time. In addition, improvement of access mode can increase the operation efficiency, too.

V. CONCLUSION

This paper mainly introduced a GPU-Based shooting and bouncing lateral-ray tube tracing method that is applied to predicting the radio wave propagation. This method can be applied in electrically large scenes which is time-consuming. Then we discussed the most efficient mode of transferring the data of triangle faces, which is a necessary part in the shooting and bouncing ray tracing algorithm. The results proved that the method can greatly reduce the computation time. Moreover, this proposed method can be implemented on the future GPU devices which support the CUDA programming.

REFERENCES

[1] H. Ling, R. C. Chow, and S. W. Lee, "Shooting and bouncing rays: Calculating the RCS of an arbitrarily shaped cavity," *IEEE Trans. Antennas Propag.*, vol. 37, no. 2, pp. 194-205, 1989.

[2] J. Baldauf, S. W. Lee, L. Lin, S. K. Jeng, S. M. Scarborough, and C. L. Yu, "High frequency scattering from trihedral corner reflectors and other benchmark targets: SBR vs. experiments," *IEEE Trans. Antennas Propag.*, vol. 39, no. 9, pp. 1345-1351, 1991.

[3] F. Weinmann, "UTD shooting-and-bouncing extension to a PO/PTD ray tracing algorithm," *Applied Computational Electromagnetics Society Journal*, vol. 24, no. 3, pp. 281-293, June 2009.

[4] S. Y. Seidel and T. S. Rappaport, "Site-specific propagation prediction for wireless in-building personal communication system design," *IEEE Trans. Veh. Technol.*, vol. 43, pp. 879-891, Nov. 1994.

[5] S. Chen and S. Jeng, "An SBR/image approach for radio wave propagation in indoor environments with metallic furniture," *IEEE Trans. Antennas Propagat.*, vol. 45, pp. 98-106, Jan. 1997.

[6] C. Yang, B. Wu, and C. Ko, "A ray-tracing method for modeling indoor wave propagation and penetration," *IEEE Trans. Antennas Propagat.*, vol. 46, pp. 907-919, June 1998.

[7] H. Suzuki and A. S. Mohan, "Ray tube tracing method for predicting indoor channel characteristic map," *Electron. Lett.*, vol. 33, no. 17, pp. 1495-1496, 1997.

[8] C. Saeidi, F. Hodjatkashani, and A. Fard, "New tube-based shooting and bouncing ray tracing method," *Proc. IEEE ATC*, Hai Phong, Vietnam, pp. 269-273, Oct. 12-14, 2009.

[9] A. Capozzoli, O. Kilic, C. Curcio, and A. Liseno, "The success of GPU computing in applied electromagnetics," *Applied Computational Electromagnetics Society Express Journal*, vol. 1, no. 4, pp. 113-116, Apr. 2016.

[10] N. A. Carr, J. D. Hall, and J. C. Hart, "The ray engine," in *Proc. Graphics Hardware'02*, pp. 37-46, Sep. 2002.

[11] Y. Tao, H. Lin, and H. Bao, "GPU-based shooting and bouncing ray method for fast RCS prediction," *IEEE Trans. Antennas Propagat.*, vol. 58, no. 2, pp. 494-502, Feb. 2010.

[12] H. Meng and J. Jin, "Acceleration of the dual-field domain decomposition algorithm using MPI–CUDA on large-scale computing systems," *IEEE Trans. Antennas Propagat.*, vol. 62, no. 9, pp. 4706-4715, Sep. 2014.

[13] K. Wang and Z. Shen, "A GPU-based parallel genetic algorithm for generating daily activity plans," *IEEE Trans. on Intelligent Transportation Systems*, vol. 13, no. 3, pp. 1474-1480, Sep. 2012.

[14] Z. Shen, K. Wang, and F.-H. Zhu, "Agent-based traffic simulation and traffic signal timing optimization with GPU," in *Proc. 14th Int. IEEE Conf.*

Intell. Transp. Syst., pp. 145-150, 2011.

[15] G. Cramer, "Introduction à l'Analyse des lignes Courbes algébriques," (in French). *Geneva: Europeana*., pp. 656-659, 1750. Retrieved 2012-05-18.

Dan Shi received her Ph.D. degree in Electronic Engineering from Beijing University of Posts and Telecommunications in Beijing, China in 2008. Now she is an Associate Professor in Beijing University of Posts and Telecommunications. Her research interests include electromagnetic compatibility, electromagnetic environment, and electromagnetic computation. She is Chair of IEEE EMC Beijing chapter, Vice Chair of URSI E Commission in China, and General Secretary of EMC section of China Institute of Electronics.

Xiaohe Tang received his Bachelor degree in Telecommunication Engineering from University of Shanghai for Science and Technology in Shanghai, China in 2015. Now he is studying for Master degree in Beijing University of Posts and Telecommunications in Beijing, China. His research interests include electromagnetic compatibility and electromagnetic environment.

Chu Wang received his Bachelor degree in Electronic and Information Engineering from Xidian University in Xi'an, China in 2015. Now he is studying for Master degree in Beijing University of Posts and Telecommunications in Beijing, China. His research interests include electromagnetic compatibility and electromagnetic environment.

Ming Zhao received his Bachelor degree in Computer Science and Technology Department from Beijing University of Posts and Telecommunications in Beijing, China in 2015. Now he is studying for Master degree in Beijing University of Posts and Telecommunications in Beijing, China. His research interests include electromagnetic compatibility and electromagnetic environment.

Yougang Gao received his B.S. degree in Electrical Engineering from National Wuhan University, China in 1950. He was a Visiting Scholar in Moscow Technical University of Communication and Information in Russia from 1957 to 1959. He is now a Professor and Ph.D. Supervisor in Beijing University of Posts and Telecommunications, China. He has been an Academician of International Information Academy of UN since 1994. He was once the Chairman of IEEE Beijing EMC Chapter and Chairman of China National E-Commission for URSI. He became an EMP Fellow of US Summa Foundation since 2010.