Accelerating 3D Digital Differential Analyzer Ray Tracing Algorithm on the GPU Using CUDA

Yan Li[1], Hongyu Sun[1], Jingchao Yuan[1], Ting Wu[1], Yanshan Tian[1,2], Yanlin Li[3], Rui Zhou[1*], Kuan-Ching Li[4]
1. School of Information Science and Engineering, Lanzhou University, Lanzhou, China
2. Manthematic and Computer Science Department, Ningxia Normal University, Guyuan, China
3. Institute of Modern Physics, Chinese Academy of Sciences, Lanzhou, 730000, Gansu, China
4. Dept. of Computer Science and Information Engr., Providence University, Taichung, Taiwan
Email:zr@lzu.edu.cn

***Abstract*—We present a novel optimized 3DDDA ray tracing algorithm on GPUs with CUDA architecture. The grid data is built on the host and then copied to the device. The whole traversing procedure is done on the GPU device. So it just needs the data transformation between the host and the device once. The results show that the CUDA implementation parallel code using different block size on GPU achieves 2.9x to 5.9x faster speed compared with the CPU implementation. And the performance is significantly influenced by the size of the CUDA block. 3DDDA algorithm on CUDA architecture can really get the performance accelerated with a reasonable block size, but it is difficult to play high-end GPU performance. The main reason is that 3DDDA algorithm for memory access is discontinuous, so it is difficult to play high-frequency performance of RAM DDR5.**

Keywords*—*Ray Tracing; 3DDDA; CUDA; Parallel Computing

I. Introduction

Ray tracing algorithm is a popular technique for rendering images in computer graphics. It's widely used in CAD and graphics related fields due to its simple principle and able to generate all kinds of lifelike visual effect. It requires that each ray must calculate the intersection node with all objects. Then sort all the intersection nodes to determine the visible points. The efficiency of this simple processing is very low in a complex environment of the scene, hence it is needed to accelerate ray tracing algorithm. Acceleration of ray tracing technique is an important part of ray tracing algorithm, and the acceleration techniques mainly include the following aspects: improve the speed of intersection calculation, reduce the amount of calculation intersection, reduce the number of light rays, utilize the generalized ray and employ parallel algorithm to speed up[1][12]. The main representative method are space subdivision and ray coherence. Space subdivision based accelerating algorithms mainly includes BSP, Kd-tree, Octree and 3D-DDA.

Ray tracing is capable of simulating a variety of light reflection and refraction, generating high degree of photo-realism. In consequence, its computational cost is high. In graphics literature, many techniques were proposed to accelerate the computation, including specific data structures and more efficient algorithms. In particular, we are interested in exploring the parallelism of ray tracing based on GPU CUDA on the linux operating system in this paper. Current trend of processor design is turning toward the multi-core processors in CPU and many-core processors in GPU. Multi-core CPU has already become popular in personal computers with dual core or quad core, even eight-core. And the mainstream GPUs in PC nowadays even include hundreds of cores such as Nvidia Geforce GTX style GPU cards. How to fully utilize their high parallel computation capabilities to improve the efficiency of ray tracing becomes an important problem.

The remainder of the paper is organized as follows. Section II provides a description of the related works of the ray tracing technology that is mostly relevant to many computer graphics, physicals, medical imaging and other actual application fields. Introduces development history of the 3DDDA algorithm from serial code to parallel code, the CUDA architecture and the GPU memory system. Section III explains our 3DDDA parallel algorithm with CUDA on GPU. Section IV is the implementation of the 3DDDA_GPU algorithm and compares the throughput with CPU serial code. The concluding section is conclusion and future works.

II. Related works

In related works, we briefly review ray tracing technology and related algorithm,the CUDA architecture and GPU memory system for parallel computing and the application of the actual application scenarios.

A. Ray Tracing Technology

Ray tracing is a generation illumination of high-reality image technique based on rendering method. It can simulate the rays of light's reflection and refraction effect from the eye back through the image plane into the scene. Ray tracing also requires a great deal of calculation. The core idea of ray tracing is reverse tracking the light enters the eyes from the observer's position. The algorithm proposed by Turner Whitted in 1979[14][5]. In ray tracing, a ray of light is traced in a backwards direction. That is, we start from the eye or camera and trace the ray through a pixel in the image plane into the scene and determine what it hits. The pixel is later set to the color values returned by the ray. The figure.1 shows the ray tracing principle between eyes, objects and the light source. And there is a virtual image plane between the eyes and the objects.

In the sequence, we mainly describe various types of accelerating algorithm for ray tracing, and then focus on 3DDDA and the application prospects of three-dimensional DDA algorithm. With regard to the efficiency of ray tracing

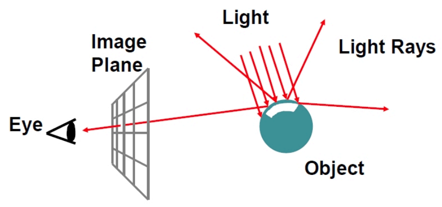

Fig. 1: Ray Tracing Principle.

algorithms, the main cause of algorithms' inefficiency is the blindness on ray intersection computing. Paying attention to the intersection computing between rays and sceneries which have no connection with these rays is meaningless. The attention, on computing between rays and sceneries after the first intersection node is useless either.

The scene space can be divided as a grid. Because of the coherence of space, tracked ray set off from the starting point, then cross the grid space sequentially[9], and finally reach the first intersection. This method is known as space partition. The correlation of divided space can be used to accelerate ray tracing process. Here we first introduce the three-dimensional digital differential analyzer(3DDDA) algorithm. In 1986, Fujimoto[4] put forward a ray tracing algorithm based on space uniform meshing technology. The algorithm divided the scene into a series of uniform three-dimensional grid, and then established auxiliary data structures spatially enumerated auxiliary data structure(SEADS)[6] to assist the calculation processing.

Upon determining the scene space partition resolution, every grid in SEADS will be located precisely by a triplet(i, j, k). Each grid contains a pointer point to its scene facet. Thus, when performing the ray tracing, the ray can simply sequentially compute intersection with scene facet in the grid where the ray passing through. In order to accelerate the ray tracing, Fujimoto extended the straight line rasterization digital differential analyzer algorithm to three dimensions, which was called three-dimensional grid across algorithm.Setting the ray's direction vector $V(V_x, V_y, V_z)$. First, we calculate the principal axis d of the traced ray. Here the d is caculated from equation 1:

$$|V_d| = max(|V_x|, |V_y|, |V_z|). \tag{1}$$

Set the two other coordinate directions of the traced ray are i and j. Then the three-dimensional digital differential analyzer grid crossing process can be decomposed into two two-dimensional DDA processes that we have introduced in the preceding chapters. Firstly, the algorithm projects the ray vertically onto the two coordinate planes intersected with the principal axis. Then perform the two-dimensional digital differential analyzer algorithm on two projection rays respectively. Based on the case of dense scenes case, the algorithm will be quite effective if the appropriate space partition resolution is selected. Currently, the algorithm has been extensively used in various kinds of commercial animation software.

B. 3DDDA Algorithm

The 3DDDA algorithm was first proposed by Fujimoto in 1986[4]. It is employed to ray traversing on the uniform grid[17]. We can calculate each sub-grids a ray goes through with 3DDDA algorithm. The figure.2 displays a schematic of a two-dimensional scenario.

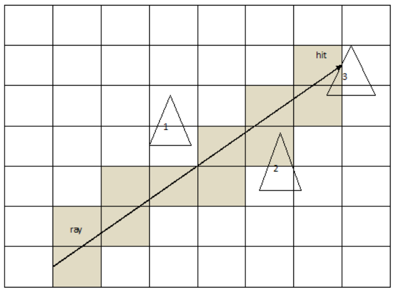

Fig. 2: Schematic of two-dimensional DDA traversal.

The gray cells in the figure.2 are the cells where the ray goes through. We just need to traverse these gray cells in the way it greatly reduces the number of intersection operations required in ray tracing procedure. And a DDA algorithm can calculate the cells a ray goes through very quickly. We can easily find the distribution of intersection points of lights and the grids are uneven, but in a single dimension it is uniform, as shown in Figure.3 to explain the distribution of intersection points in a single direction[13].

So each cell id can be calculated by simple increments in each direction because of the even distribution in each single direction. For example, as showed in figure.3(b), the ray is divided into 5 parts in the $y - direction$, and each part has a same length. The mark of ty_min is the t value of the first intersection point and the ty_max is the last one. The t value is the length from the start point of the ray to the intersection point in the ray tracing algorithm. Therefore, the coordinate of the intersection point can be calculated like this:$ray.origin + ray.direction * t$. With $ty\_minandty\_max$, we can get the increment of the y direction with equation 2:

$$dt_y = \frac{(ty\_max - ty\_min)}{n_y} \tag{2}$$

Here, n_y is the number of cells in the $y - direction$.

If the coordinate of the current cell in traversing is $< ix, iy, iz >$, and the t value for each direction is $< t_x, t_y, t_z >$, with increment $< dt_x, dt_y, dt_z >$. So, the next cell is the one which gets the min t value. For example, if $t_x + dt_x$ is the min one , then the next cell should be $< ix + step_x, iy, iz >$. The $step_x$ value is 1 when the direction of the ray is common with the positive $x - axis$, and it will be -1 when the direction of the ray is common with the negative $x - axis$. And these are same with $y - direction$ and $z - direction$.

If a ray hits an object in a cell, then there is no need to traverse the next cell for we just need the nearest intersection point. This is one of the end conditions. Another condition is

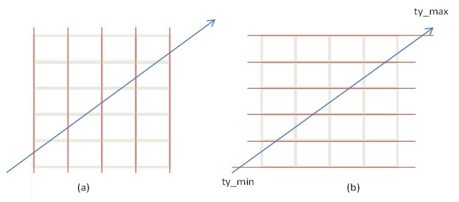

Fig. 3: Distribution of Intersection Points in a Single Direction.

that a ray hits nothing but just goes through the space. So, we need to set end condition as $< ix\_stop, iy\_stop, iz\_stop >$. The ik_stop value is also related with the ray direction. For example, $ix\_stop = nx$ when the ray direction is common with the positive $x - axis$, and $ix_stop = -1$ when the ray direction is common with the negative $x - axis$.

C. CUDA Parallel Archtecture

CUDA(Compute Unified Device Architecture)[11] is a general parallel computing architecture which is launched by NVIDIA. CUDA architecture enables GPU to parallel solve the problem of complicated calculations. It contains the CUDA instruction set architecture (ISA), and the inside of the GPU parallel computing engine. CUDA provides direct access to virtual instruction sets and parallel computing components of memory to the application developer. The figure.4 shows an example of CUDA processing flow.In the figure,there are four steps processing data in parallel methods.

1). Copy data from the host main memory to the GPU memory;

2). Instruction from CPU to awaken GPU to process the data;

3). The GPU process the data in parallel model by each CUDA core and store the results in GPU memory;

4). Carry the processing result from GPU memory to the host main memory.

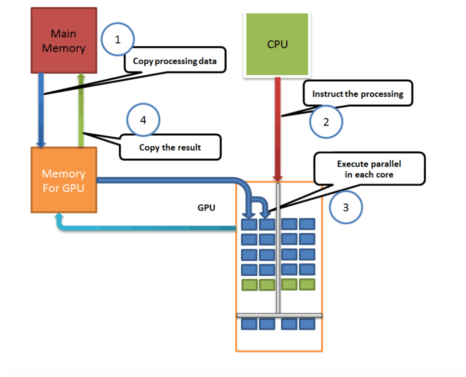

Fig. 4: Example of CUDA processing flow.

The GPU memory system contains four types of hierarchically organized memories: global, constant, texture and shared memories, as showed in Figure.4. Although the performance of on-chip memory is astonishing, its capacity is very limited. In the case of dose calculation, none of the onchip memory units is large enough to hold the dense data of majority object model data for real application cases. On the other hand, the global memory, which is the only choice for the main memory, suffers from large performance variations. The latency of the global memory is in the range of 400-800 clock cycles. For volume ray tracing in the actual application scenarios, the computation v.s. memory access ratio is normally too low to hide such long latency.The figure.5 shows the host-device memory system,and mainly in memories available to the GPU programing[16][8].

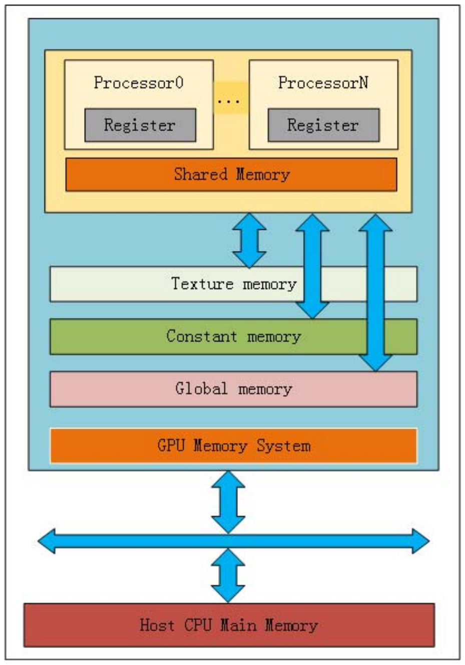

Fig. 5: Memories available to GPU programming.

To improve the performance of off-chip memory accesses, the memory controller of the GPU is designed to be able to read 32-bit, 64-bit, or 128-bit words from global memory into registers with just one single instruction if the memory accesses are contiguous and aligned. This grouping mechanism makes it critical to properly arrange the memory accesses for all GPU programs[7]. Otherwise, one may easily discover that a computation-intensive application on CPU becomes a memory-bounded application on GPU and lots of GPU hardware potential is wasted. Unfortunately, this takes place at the case for ray tracing[15].

III. 3DDDA_GPU Implementation with CUDA

The algorithm.1 shows a pseudo-code of the 3DDDA at a high level.

It is obvious that each iterator in the for-loop is independent, as the pseudo-code algorithm.1 shown. So each iterator can be parallel. Now, we need to implement the 3DDDA algorithm on the GPU(3DDDA_GPU) with CUDA architecture to use GPU's power.

Algorithm 1 3DDDA Algorithm

```
Require: global_bbox as the bounding box of the whole scene
  while each ray in ray_buffer do
    if the ray does not hit global_bbox then
      return false
    end if
    set< ix, iy, iz > start cell of the ray
    set loop=true
    set hit=false
    while loop==true do
      while each object in cell < ix, iy, iz> do
        try hit of the ray and the object
        if hit then
          hit=true
          compare and record the nearest hit point
        end if
      end while
      if hit then
        loop=false
      end if
      if tx< ty and tx < tz then
        ix=ix+step_x
        tx=tx+dtx
        if ix==ix_stop then
          loop=false
        end if
      else
        if ty< tz then
          iy=iy+step_y
          ty=ty+dty
          if ty==ty_stop then
            loop=false
          end if
        end if
      else
        iz=iz+step_z
        tz=tz+dtz
        if iz equal iz_stop then
          loop=false
        end if
      end if
    end while
    if hit then
      return ture
    else
      return false
    end if
  end while
```

First, we are required to store the grid data and objects data used by 3DDDA algorithm in the device memory. The grid data are mapped into a 3-dimentional array[2][3] . And each element as a cell is utilized to store 2 integers which indicate the begin index and the end index of a slice in an integer array. And the slice stores the indexes of all the objects in this cell. The specific relationship is shown in Figure.6.

Before the realization, we consider that any flow control instruction (if, switch, do, for, while) can significantly affect the instruction throughput by causing threads of the same warp to diverge. To reduce the impact caused by warp divergence, we should decrease the size of codes in the branch block. For instance, consider the code in Algorithm.2.

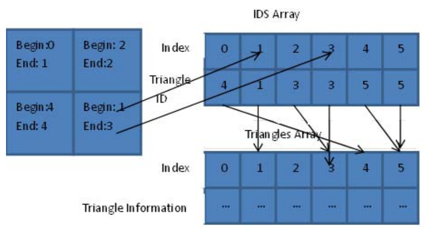

Fig. 6: Memories available to GPU programming.

Algorithm 2 A Branch Code Instance

```
Require: there have several logic judgement in this code
  if tx < ty and tx < tz then
    ix = ix + step_x
    tx = tx + dtx
  end if
  if ix==ix_stop then
    loop = false
  else
    if ty<tz then
      iy = iy + step_y
      ty = ty + dty
    end if
    if iy==iy_stop then
      loop = false
    else
      iz = iz + step_z
      tz = tz + dtz
    end if
  end if
  if iz==iz_stop then
    loop = false
  end if
```

Algorithm.3 presents an instance of the Algorithm.2 with the combined code ix, iy, iz into one array $i[3]$. Let $i[0] = ix, i[1] = iy, i[2] = iz$. And it noticeably puts forward a smaller code size. So the code should be cut like the instance optimized code in Algorithm.3.

And the same procedure has been carried out in other places of the code. Now we can use a CUDA thread to do a ray's traversing procedure. Launch a lot of threads in a kernel call, and we can calculate many rays in parallel.

IV. EXPERIMENT

We tested all the results and performance of our code in this paper with a platform which has a host CPU Intel Xeon X5650 (2.67GHz),8G main memory, a GPU Tesla C2050 with a compute capability 2.0, 448 multiprocessor cores with 1.15GHz core clock, 3GByte local memory with a bandwidth of 144GB/s, and CUDA version 5.5. Use including 69451 triangles from "The Stanford Models" file bun_zipper.ply[10] as our test data. The camera view point of the scene is set from maximum to minimum with a horizontal angle of 60 degrees,

Algorithm 3 An Optimized Branch Code

```
Require: this code have optimized the logic judegement
  if tx < ty and tx < tz then
    k=0
  else
    if ty<tz then
      k = 1
    else
      k=2
    end if
  end if
  i[k] += step[k]
  t[k] += dt[k]
  if i[k]==stop[k] then
    loop = false
  end if
```

and the resolution ratio is set to 1024*1024 pixel. And we only use an ambient environmental light without shadows and reflections. So we can concentrate on the computing in the 3DDDA algorithm.

The experimental results show that our GPU implementation can perform calculations in 0.44s with a 2.38M (rays/s) throughput. And our CPU implementation can complete the calculations in 1.29s with O3 level optimization, and a 0.81M (rays/s) throughput. Performance metric used the number of processed rays per second. Table.1 sets the block throughput under different size.

TABLE I: Throughput under different block size

Block size	GPU performance(rays/s)	CPU performance(rays/s)	Speed up
512	2.38M	0.81M	2.9
256	4.03M	0.81M	5.0
128	4.77M	0.81M	5.9
64	4.56M	0.81M	5.7
32	3.38M	0.81M	4.2

It is not difficult to find that the performance is best when the block size is 128. By analyzing the report from Nsight[1], we find that the SM register utilization rate is highest with the block size 128. According to CUDA documentation[2], we know that each CUDA block can only run on an SM in its life cycle. But an SM can concurrently execute multiple blocks. The concurrent block threads use limited registers on SM, thus restricting the number of concurrent threads warp. So, when the block size is not suitable, SM could only load one block, and the rest of the registers cannot accommodate the second block, thus leading to the waste of resources

Then we test the performance of 3DDDA GPU code on different GPUs.The best performance on Quadro 600 is 1.70M (rays/s). Although this value is lower than the worst performance parameters of Tesla C2050, considering the performance gap of TeslaC2050 and Quadro 600, we can find the program difficult to play high-end GPU power, for C2050 SM having 14 stream multiprocessors, which is 7 times than Quadro 600. After analysis using Nsight, the reason why Tesla C2050 has poor performance is the memory utilization is not high. In 3DDDA algorithm, a ray traversing from one grid to the next grid, we need access to the corresponding three-dimensional array of elements. Actually, the three-dimensional stored in a global memory with one dimension, can only guarantee the data stored contiguously on one dimension. Therefore, the data which CUDA threads read will not be continuously in memory with probability of 2/3. It will make DDR5 memory utilization rate not very high.

[1]Nsight:NVIDIA corporation develop tools

[2]http://docs.nvidia.com/cuda/cuda-c-programming-guide/#axzz3Ygsiw9kT

V. CONCLUSION AND FUTURE WORKS

3DDDA algorithm on CUDA architecture can really get the performance accelerated with a reasonable block size, but it is difficult to play high-end GPU performance. The main reason is that 3DDDA algorithm for memory access is saltatory, so it is difficult to play high-frequency performance of RAM DDR5. In contrast, 3DDDA algorithm is more suitable for low-end GPU. Therefore, for the large-scale 3DDDA algorithm, GPU cluster effect with the composition of the low-end GPU should be better.We can see from the figure.7 which shows that GPU parallel implementation of 3DDDA has a different performance with different GPU block size.

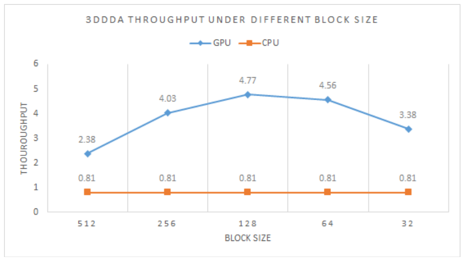

Fig. 7: 3DDDA Throughput Under Different Block Size.

The effect will not be very obvious if we run entire 3DDDA algorithm on GPU. Maybe we can run part of 3DDDA algorithm on GPU, and let GPU and CPU execute different parts. Meanwhile, use the stream of CUDA to hide the data transfered between host and device. In future work we will consider using share memory to speed up the grid data of global memory access and consider dividing the kernel function into sub-kernel in order to reduce pressure of the registers.

ACKNOWLEDGMENT

This work is supported in part by the "Strategic Priority Research Program" of the Chinese Academy of Sciences under Grant No.XDA01020304, National Natural Science Foundation of China under Grant No. 61402210 and 60973137, Program for New Century Excellent Talents in University under Grant No. NCET-12-0250, "Strategic Priority Research Program" of the Chinese Academy of Sciences with Grant No. XDA03030100, Gansu Sci.and Tech. Program under Grant No. 1104GKCA049, 1204GKCA061 and 1304GKCA018. The Fundamental Research Funds for the Central Universities under Grant No. lzujbky-2014-49, lzujbky-2013-k05, lzujbky-2013-43, lzujbky-2013-44 and lzujbky-2012-44, Gansu Telecom Cuiying Research Fund under Grant No. lzudxcy-2013-4, Google Research Awards and Google Faculty Award, China. Ningxia Normal University's science fund with Grant NO.ZZ201204, NGY2013106 and Ningxia Natural Science Foundation with Grant NO.NZ14278.

References

[1] F. Aguado, A. Formella, J. M. Hernando, and F. Isasi. Ray-tracing acceleration techniques. *Microwave and Optical Technology Letters*, 25(5):363–365, 2000.

[2] Didier Badouel and Thierry Priol. An efficient parallel ray tracing scheme for highly parallel architectures. In RichardL. Grimsdale and Arie Kaufman, editors, *Advances in Computer Graphics Hardware V*, EurographicSeminars, pages 93–106. Springer Berlin Heidelberg, 1992.

[3] S. Cioli, G. Ordeix, E. Ferna?ndez, M. Pedemonte, and P. Ezzatti. Improving the performance of a ray tracing algorithm using a gpu. In *Chilean Computer Science Society (SCCC), 2010 XXIX International Conference of the*, pages 11–20, Nov 2010.

[4] A. Fujimoto, T. Tanaka, and K. Iwata. Arts: Accelerated ray-tracing system. *Computer Graphics and Applications, IEEE*, 6(4):16–26, April 1986.

[5] Venkatraman Govindaraju, Peter Djeu, Karthikeyan Sankaralingam, Mary Vernon, and William R. Mark. Toward a multicore architecture for real-time ray-tracing. In *Proceedings of the 41st Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 41, pages 176–187, Washington, DC, USA, 2008. IEEE Computer Society.

[6] Michal Hapala, Tomáš Davidovič, Ingo Wald, Vlastimil Havran, and Philipp Slusallek. Efficient stack-less bvh traversal for ray tracing. In *Proceedings of the 27th Spring Conference on Computer Graphics*, SCCG '11, pages 7–12, New York, NY, USA, 2013. ACM.

[7] Pawan Harish and P.J. Narayanan. Accelerating large graph algorithms on the gpu using cuda. In Srinivas Aluru, Manish Parashar, Ramamurthy Badrinath, and ViktorK. Prasanna, editors, *High Performance Computing âĂŞ HiPC 2007*, volume 4873 of *Lecture Notes in Computer Science*, pages 197–208. Springer Berlin Heidelberg, 2007.

[8] S Heymann, K Muller, A Smolic, B Frohlich, and T Wiegand. Sift implementation and optimization for general-purpose gpu. In *Proceedings of the international conference in Central Europe on computer graphics, visualization and computer vision*, volume 144, 2007.

[9] Javor Kalojanov and Philipp Slusallek. A parallel algorithm for construction of uniform grids. In *Proceedings of the Conference on High Performance Graphics 2009*, pages 23–28. ACM, 2009.

[10] Venkat Krishnamurthy and Marc Levoy. Fitting smooth surfaces to dense polygon meshes. In *Proceedings of SIGGRAPH 96*, pages 313–324, 1996.

[11] D. Luebke. Cuda: Scalable parallel programming for high-performance scientific computing. In *Biomedical Imaging: From Nano to Macro, 2008. ISBI 2008. 5th IEEE International Symposium on*, pages 836–838, May 2008.

[12] Timothy J. Purcell, Ian Buck, William R. Mark, and Pat Hanrahan. Ray tracing on programmable graphics hardware. *ACM Trans. Graph.*, 21(3):703–712, July 2002.

[13] Ingo Wald, Thiago Ize, Andrew Kensler, Aaron Knoll, and Steven G Parker. Ray tracing animated scenes using coherent grid traversal. In *ACM Transactions on Graphics (TOG)*, volume 25, pages 485–493. ACM, 2006.

[14] Turner Whitted. An improved illumination model for shaded display. *Commun. ACM*, 23(6):343–349, June 1980.

[15] Bo Zhou, Cedric X. Yu, Danny Z. Chen, and X. Sharon Hu. Gpu-accelerated monte carlo convolution/superposition implementation for dose calculation. *Medical Physics*, 37(11):5593–5603, 2010.

[16] You Zhou and Ying Tan. Gpu-based parallel particle swarm optimization. In *Evolutionary Computation, 2009. CEC'09. IEEE Congress on*, pages 1493–1500. IEEE, 2009.

[17] Martin Zlatuška and Vlastimil Havran. Ray tracing on a gpu with cuda–comparative study of three algorithms. *Informe Técnico, Czech Technical University in Prague Faculty of Electrical Engineering*, 2009.