Performance Evaluation of CUDA Based Ray Tracer

Submitted by

Nitin Garg	19BCE0239
Aadarsh N Prasad	19BCE2168
Suyash	19BCE2297
Ananya Anand	19BCE2332

Course Code: CSE 4001

Course Title: A Parallel and Distributed Computing

Under the guidance of

Dr. S. Anto

Associate Professor, SCOPE,

VIT , Vellore.

SCHOOL OF COMPUTER SCIENCE AND ENGINEERING

April, 2022

INDEX

1. INTRODUCTION

One of the most important objectives of computer graphics is the simulation of light and reflections. Over the years, the illuminance of objects has been improved vastly to capture the real-life aspects of an object for depiction in media such as games, movies, animations, etc. [2]. Due to the importance of lighting in the world of computer graphics, different techniques have been adopted to improve the time taken and efficiency of generating photorealistic graphics. Mainly two techniques namely rasterization and ray tracing are used for producing such graphics [3].

Rasterization is used to create pixelated images called raster images from a vector format whereas in ray tracing, rays are projected through a view plane and their trajectories are traced to get individual pixels in the view plane [4]. Among these two techniques, rasterization has been a staple in computer graphics for a long time, simply because it is much faster in rendering good quality images but it falls short in rendering 2D and especially 3D photo realistic images as it handles reflections poorly.

As compared to rasterization, ray tracing is much slower when it comes to rendering an image, but it shines in the department of photo realism as ray tracing can produce very accurate simulations of reflections, refractions and shadows. The pitfall of high computational cost and thus slow rendering time, can be overcame by the use of parallel programming and GPUs.

The main objective of this project is to reduce time to perform the tracing while retaining the properties of creating realistic images. In this project, we studied about the serial implementation and parallel implementation of a ray tracing algorithm, analyzed their rendering performances and stated our conclusions and reflections.

The report is organized as follows. Section 2 introduces our contribution of work towards parallelization of ray tracing. Section 3 includes the related works being conducted in ray tracing. Section 4 the existing paradigms in and parallel implementations (e.g., OpenMP) in ray tracing [1]. Section 5 and 6, deals with our CUDA implementation of a serial ray tracer. Finally, in Section 7 and 8, the results and conclusion are discussed.

2. LITERATURE SURVEY

2019 – 2021

TITLE	AUTHOR	JOURNALAND DATE	KEY CONCEPTS	ADVANTAGES	DISADVANTAGES	FUTURE ENHANCEMENT
Performanc eEvaluation of Monte Carlo Based Ray Tracer	Ayobami Ephraim Adewale	Journal of Computationa lScience Education Volume 12, Issue 1 ISSN 2153-4136 January 2021	The paper discusses 2 unique ray tracing techniques (Distributed algorithms) and performanc eevaluation of serial and parallel way of ray tracing.	Using multithreadin g i.e. In parallel implementatio nof ray tracing (OpenMP) leads to 10 times faster render times than in normal serial implementatio n .	The parallel implementation runs on the CPU whereas an implementation utilizing the GPU could be much more efficient due to the nature of thetask at hand.	Comparison of a serial real time distributed ray tracing implementation against a parallel implementation ofboth OpenMP andMPI could be included in future works.
A Ray-Tracing Algorithm Based on the Computatio n of (Exact) Ray Paths with Bidirectional Ray-Tracing	Mehmet Mert Taygur, Thomas F. Eibert	IEEE Transactions on Antennas and Propagation June 2020	This paper presents a novel ray-tracing algorithm to cope with the phase errors due to incorrect ray path computatio n s in currently existing ray-launching approaches.	The proposed new algorithm is effective in reducing the complexity of the simulation process by avoiding Keller cones for edges which are aligned with the interaction surface. Thus, the proposed algorithm is better and more efficient than other traditional ray	The proposed algorithm is successfully demonstrated on only 3 different problems; Thus, the algorithm requires to be stress tested to prove to be efficient for many other cases.	Future work includes, more rigorous testing of the proposed algorithm, this must be done to ensure its credibility towards other different simulations.

				tracing algorithms.		

Progressive path tracingwith bilateral-filtering-based denoising	Qiwei Xing, Chunyi Chen, Zhihua Li	Multimedia Tools and Applications (2021) 80, Springer 2021	This paper presents a novel denoising algorithm framework. This paper also proposes an improved bilateral filtering algorithm with use of the gradient feature to obtain the noise-free images.	The framework proposed is suitable for the PT denoising method based on a neural network. The framework performs well at a low sampling rate and with the neural network,more optimal filter parameters can be obtained.	The main limitation of the proposed framework is that the estimator may be inefficient at recognizing fine structures. This could lead to over blurred images, or there could be aliasing on the edge of textures produced.	The noise estimator could be modified to improve the adaptive samplingwhile neural networkscould be utilised to find better feature parameters for the pixels with different effects.

High-Quality Rendering of Glyphs Using Hardware-Accelerated Ray Tracing	S. Zellmann, M. Aumüller, N. Marshak and I. Wald	Eurographics Symposium on Parallel Graphics and Visualization 2020	This paper presents implementations of various glyph rendering techniques using hardware-accelerated ray tracing with RTX.	The brute force approach implemented in the paper allows to renderhigh Number of glyphs. It was observedthat the performance of the OpenGL renderer is dominated by vertex processing load and is very respectable.	The paper uses the information provided by already conducted researches which might not be as accurate and authentic. This a more thorough examination of the implemented algorithms is required.	In the future the authors aim to combine motion blur with glyphs other than spheres, so that velocity can be shown together with another quantity. Theauthors also want to enable evaluation of the effectiveness and usefulness of the proposed methods byintegrating them intoproduction visualization systems such as ParaView or Vistle.

2017 – 2018

SIMD Monte-Carlo Numerical Simulations Accelerated on GPU and Xeon Phi	Bastien Plazolles, Didier El Baz, Martin Spe, Vincent Rivol, Pascal Gegout	International Journal of Parallel Programming 2018	This paper presents optimisations of Monte-Carlo numerical simulations of stratospheric balloon envelope drift descent is. The optimization of the SIMD parallel codes on the K40 and K80 GPUs as wellas on the IntelXeon Phi are considered .	The paper putshigh emphasison loop and task parallelism, multi-threading and vectorization in SIMD optimisations. Computing accelerators appear to be very serious alternatives to clusters in order to solve real world problems in operational conditions.	The implementations of the parallel algorithm on the GPUs K40 and K80 are dependent on their respective architectures, in order to take advantage of the massive parallelism ability of the GPUs highly specific code is required, which can't be easily extrapolated for other architectures.	The application presented in the present paper isan illustration of a numerical integrator with Monte-Carlo perturbation of initial conditions. This is a general class of problems which has many fields of applications such as atmospheric re-entry of satellites, ballistic trajectory prediction, ray tracingof GNSS], etc. Many of these applications could benefit from the methodology presented in this paper.

Optimizing Ray Tracing Algorithm Using CUDA	Sayed Ahmadreza Razian, Hossein Mahvash Moham madi	Emerging Science Journal & Italian Journal of Science & Engineering Vol. 1, No. 3 October, 2017	The paper proposes an optimized algorithm which has been designed to generate images using ray tracing algorithm to running on the GPU using multiple threads.	The multi-threads implementation ,running on a graphic card can generate HD and Full HD images. A lot of time is also saved by transferring computations from CPU to GPU.	The proposed implementation cannot render images in real time. The results also show that the algorithm is not good for CPU processing as it takes too much timewith poor output. Thus, the algorithm is only suited for CUDA computing.	In the future the authors wish to improve the algorithm to include real time ray-tracing and to decrease the high time cost when the algorithm runs onthe CPU.

A Comprehens ive Review of Efficient Ray-Tracing Techniques for Wireless Communicat ion	Tan Kim Geok, Ferdous Hossain, Mohd. Nazeri Kamarud din, Noor Ziela Abd Rahman, Sharlene Thiagaraj ah, Alan Tan Wee Chiat, Jakir Hossen, Chia Pao Liew	International Journal on Communicatio ns Antenna and Propagation Vol. 8, N. 2 April 2018	The paper describes elemental ray-tracing methods and also reviews recently upgraded methods. Thepaper revies numerous ray-tracing progressive methods and then groups these methods into several related categories.	The paper reviews various ray-tracing methods for wireless communication on the basis ofsufficient categories like heuristic, effective deterministi cand special brute force.	The paper includes review of existing ray tracing techniques used in one of the real-world applications of ray tracing: Wireless communication.

Implementation of Image Enhancement Algorithms and Recursive Ray Tracing using CUDA	Mr. Diptarup Sahaa, Mr. Karan Darjib, Dr. Narendra Patelc, Dr. Darshak Thakored	Procedia Computer Science Volume 79, 2016, Pages 516-524 2016	This paper presents analysis of several image processing algorithms implemented on both CPU and GPU (CUDA). This paper also includes implementation of recursive raytracing implemented on GPU.	The proposed recursive ray tracing algorithms is very computationally expensive, so it is only possible using parallelisation of existing ray tracing algorithms. This paper concludes thatin parallel applications better results can be achieved with lower number of threads per block.	The recursive ray tracing algorithm cannot be implemented on the CPU and it can only be compared with a sequential CPU implementation.	The ray tracing algorithm is implemented for 300 passes, with each pass producing a clearer image. Thus, the upper bound in the quality of the image produced by recursive ray tracing can be determined by increasing the number of passes the algorithm runsfor.

2015 – 2016

Ray tracing method for stereo image synthesis using CUDA	Al-Oraiqat Anas M., Zori S. A	International Journal of Engineering Science and Technology Vol. 8 No.06 Jun 2016	The paper presents an implementation to visualize 3Dimages with the use of parallel Graphics processing units. The paper also gives the main features of volume visualization and introduces stereo visualization by method of tracing beams on parallel GPU based on experimental studies.	The proposed algorithm includes several of the optimizations of modern ray tracing methods such as the ability touse binary search to find the primitive intersected by ta ray. The algorithm also requires implemented using Kd tree also requires very little memory and is able to performbatch tracing.	The proposed algorithm is very computationall yexpensive and cannot be used without the use of high-performance parallel computing solutions in the case of real timerendering. With the increase in computing complexity of a scene to be rendered, time for its processing increases too.	Future work could include optimizations toreduce the computing complexity of a scene which will reduce the total time taken in processing the image.

A Machine Learning Approach for Filtering Monte Carlo Noise	Nima Khademi Kalantari, Steve Bako, Pradeep Sen	ACM Transactions on Graphics, 2015	The paper proposes a machine learning approach to filter Monte Carlo rendering noise as a post-process.	Several distributed effects are used to train a multilayer perceptron (MLP) neural network (as a nonlinear regression model) to output correct filter parameters to denoise a Monte Carlo rendered image.	This paper implements a neural network as a nonlinear regression model. It does not consider the usefulness of other non-linear regressions models.	This paper presents one of the first attempts to apply machine learning to Monte Carlo noise reduction, future works could include analysis of other non-linear regression models such as a support vector machine. Adaptive sampling could also be introduced to Monte Carlo denoising.

Recent Advances in Adaptive Sampling and Reconstructionfor Monte Carlo Rendering	M. Zwicker, W. Jarosz, J. Lehtinen, B. Moon, R. Ramamoorthi, F. Roussell, P. Sen, C. Soler and S.-E. Yoon	ComputerGraphics Forum 2015	This paper surveys recent advances in adaptive sampling and reconstruction. The methodologies surveyed have proven to be very effective at reducing the computational cost of Monte Carlo techniques in practice.	A wide variety of noise or variance reduction strategies havebeen surveyed.Such as, different path sampling strategies, statistical techniques andsignal processing methods.	This paper doesnot include analysis and comparison of data-driven andlearning based approaches.	Many largely unexplored techniques and methodologies can be covered in the future. These could include data-driven and learning-basedapproaches, or general frameworks for sparse sampling and reconstruction based on work in compressive sensing and matrix completion.

Massively Parallel Ray Tracing Algorithm Using GPU	Yutong Qin, Jianbiao Lin,Xiang Huang	[IEEE 2015 Science and Information Conference (SAI) 2015	The paper introduces a GPU algorithm aims to parallelise the computationall yexpensive process of Ray tracing.	The algorithm takes 6-10 seconds to create a photorealistic image (SD resolution) with efficient lighting and shadows.	Because of the high computation complexity, theproposed algorithm can't reach the requirement of real-time rendering. In initial stages the image generated has amounts of black colour noise.	In the future the authorwants to improve the results of the algorithmby incorporating real time raytracing.

Gradient-Domain Path Tracing	Markus Kettunen ,Marco Manzi, Miika Aittala, Jaakko Lehtinen, Fredo Durand, Matthias Zwicker	ACM Transactions on Graphics July 2015	The paper introduces gradient domain rendering forMonte Carloimage synthesis.	The gradient path tracer shifts each base path to its fourvertical and horizontal neighbors, resulting in standard finite difference gradient estimates, this provides small additional error reduction.	This paper proposes gradient domainpath Tracing, a relatively simple modification of the standard path tracing algorithm that can yield far superior results, since this is a very new approach more analysis of this methodology is required.	In the future, conceptssuch as gradient domain bidirectional path tracingand related approaches can be explored. In addition tothis development of more advanced shift mappings,custom tailored shift mappings can be explored.

Range Sensors Simulation Using GPU Ray Tracing	Karol Majek and Janusz Bedkowski	Proceedings of the 9th InternationalConference on Computer Recognition Systems CORES, Springer 2015	The paper includes a GPU accelerated range sensors simulation and discusses various optimizations.	Due to high performance of the simulation other additional processing like adding noise or discretization are also implemented.	The simulation is currently usedonly for performance check. The simulation of structured light is considered to be future work by the authors.	The future work could include the integration ofthe simulator with a robotics framework such as the ROS (Robot Operating System). Simulation of the structured-light could also be considered as afuture enhancement.

2.1 PROBLEM DEFINITION

Enhancing ray tracers and tracing algorithms by varying various parameters and determining the right combination of parallelization techniques and using distributed tracing, and selecting the right variation of hyperparameters to further optimize the renderers.

3. OVERVIEW OF WORK

3.1 OBJECTIVE

In the field of computer graphics, Ray Tracing is a rendering technique used to generate an image by tracing the path of light rays as pixels on a plane and simulating various effects and encounters with virtual objects. This technique is capable of producing high degrees of visual realism but at the sacrifice of computational costs. In this project we make an attempt at reducing the time taken to render an image using parallelization techniques. The aim of our project is to present an implementation and analysis of a CUDA based parallel Ray Tracer. CUDA is a parallel computing platform and application programming interface model created by Nvidia. CUDA parallelization paradigms allow us to use multi-threading to achieve maximum speedups up to 18 times against the serial implementation.

3.2 HARDWARE REQUIREMENTS

- Processor: Intel i5 – 10210u, 14nm, 1.60 Ghz Base frequency, Intel i5 – 10300H, 14nm, 2.50 Ghz Base frequency
- Number of cores: 4, Number of threads: 8
- Memory: 8 GB, 2666 Mhz, 16 GB 3200Mhz
- Graphics: Intel UHD Graphics 620 Number of Execution Units, Cores, Shaders: 24, Nvidia GTX 1650, Number ofCoder: 896 CUDA cores

3.3 SOFTWARE REQUIREMENTS

- Nvidia GeForce Experience (To ensure up to date drivers)
- CUDA C++
- Nvidia CUDA Toolkit
- Visual Studio
- .ASP NET
- CMake

4. SYSTEM DESIGN

4.1 EXISTING SYSTEM DESCRIPTION

Ray tracing renders a photo-realistic 3D image by tracing the trajectories of light rays through pixels in aview image. Tracing the light in ray tracing can be achieved with to methodologies namely, forward tracing and backward tracing. When the rays are traced from camera to light source, it is known as forwardtracing, while in backward tracing, rays are traced from the light source to the camera. Backward tracingis much more computationally expensive as every ray emitted by the light source (even if it doesn't reachthe camera) has to be traced. Figure 1 describes a scene to be ray traced.

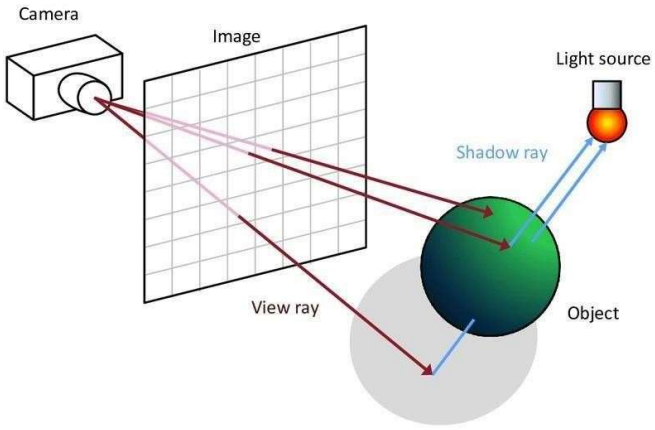


Figure 1: A simple ray tracing diagram

Algorithm 1 is a simple ray tracer algorithm wherein, the rays sent out by camera (for forward ray tracing)can be represented as a line (ray) with an origin (point A) and a direction (Vector B). At any time, a pointon the line (ray) can be represented with

$$P(t) = A + B \cdot t \quad (1)$$

where P is a 3D point lying along a line in 3D. The ray parameter t is the distance of the point from the ray origin. Using the origin and direction of the ray, all points lying on that ray can be computer. Using these points on a ray and the 3D position of a sphere, we can compute whether the ray intersects with anysphere in our scene at any points

The ray-sphere interactions can be summarized using figure 2.

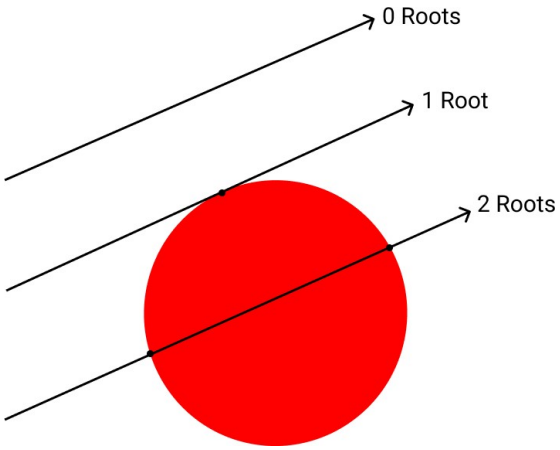


Figure 2: Ray – Sphere interactions

A sphere in 3D space can be defined vectors as:

$$(P - C) \cdot (P - C) = r^2 \qquad (2)$$

Where P is any point on the surface of the sphere and C is the center of the sphere, with r being the radiusof the sphere. Using equations 1 and 2, the ray sphere interactions can be defined as:

$$(P(t) - C) \cdot (P(t) - C) = r^2 \qquad (3)$$

$$((A + B \cdot t) - C) \cdot ((A + B \cdot t) - C) = r^2 \qquad (4)$$

The quadratic equation (4) can be solved for t to get different possibility in number of real roots, asdepicted in Figure 2.

4.2 ALGORITHM

```
for each pixel in the viewing plane do
  for each object in the scene do
    if ray intersects an object in the scene, then
      select min (d1, d2);
      recursively ray trace the rays;
      calculate color;
    end
end
end
```

Algorithm 1: Simple Ray Tracing Algorithm

The most significant pitfall of this ray tracing method is that it causes aliasing. When the ray does not intersect with a sphere (no roots), the color of the background is returned, and thus for every pixel in the view frame a deterministic color is returned, which gets overwritten by interactions of other rays with thatpixel. The solution to this is to introduce more rays into the scene and to make the ray-object interactionsmuch more random.

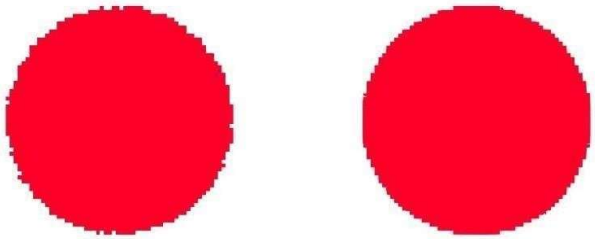


Figure 4: Aliased and Anti – aliased ray traced sphere

Distributed ray tracing further expands on traditional ray tracing method by the introduction of sampling to get rid of aliasing. Anti – aliasing an image leads to much higher shadow quality and better reflectionsthus rendering a more photo-realistic image. The single ray used to compute the color of a pixel is insteadreplaced with multiple rays and an average of some of these randomly sampled rays is taken to render theanti-aliased pixel. Traditional ray tracing, does not trace the rays after they hit a diffused surface, but in the distributed ray tracing, many forked rays are generated randomly based on the bi-directional reflectionand refraction distribution function of the diffuse surface [7].

Algorithm 2: Distributed Ray Tracing Algorithm

```
for each pixel in the viewing plane do
 for each ray in random rays do
  for each object in the scene do
   if ray intersects an object in the scene, then
    select min (d1, d2);
    recursively ray trace the rays;
    return calculated color;
   end
   if no intersection then
   return background color;
   end
  end
 end
 random sample rays with Monte Carlo;
 calculate color average;
end
```

The time complexity of a distributed ray tracing algorithm will definitely take much more time to renderan image all the while producing higher quality and more photo-realistic renders.

4.3 SYSTEM ARCHITECTURE

A comparative study of CPU and GPU architecture

A GPU has a small cache that is shared among threads, but its main memory has a high bandwidth. Each core of a CPU has a relatively large cache, but main memory is slow. Indirectly, this means that GPUs are excellent for sequential/streaming access. When there is very few diverging if statements, a GPU's 32-way SIMD and lack of appropriate branch prediction perform best. Conditions aren't a problem because most CPUs feature 4 to 8-way SIMD and extremely powerful branch predictors and pre-fetchers. In addition, a GPU lacks cache coherency and any synchronization-specific hardware. The CPU, on the other hand, is created with a large number of inbuilt synchronization primitives. As a result, low-contention code works best with a GPU.

Source: Thambawita, Vajira & Ragel, Roshan & Elkaduwe, Dhammika. (2014). To Use or Not to Use: Graphics Processing Units for Pattern Matching Algorithms.

Figure 5: A visual comparison of the difference between a CPU and a GPU.

4.4 RENDERED SCENE

The generated scenario that we utilized to test and analyze our serial and CUDA-based parallel ray tracers is made up of spheres on a 3D plane with light sources casting rays into the scene and the camera capturing the reflected and refracted rays. Anti-aliasing is achieved by sampling rays through a pixel at random and then averaging them to produce a colour. There were three different types of materials used: diffuse or Lambertian, metal, and dielectric. Metal objects exhibit fuzziness in reflection, whereas dielectrics exhibit physically correct phenomena like refraction and total internal reflection. To acquire appropriate color intensities, the produced scene is additionally gamma adjusted.

5. IMPLEMENTATION

5.1 DESCRIPTION OF MODULES/PROGRAMS

The image processing in the CUDA version was done in square batches of pixels (e.g., an 8 by 8 batch). The first two kernels are used to set up and establish the sort of scene that has to be drawn, while the next two kernels are used to set up and initialise the random numbers for each thread while rendering the required image.

To traverse over all of the pixels in the serial implementation, we used nested for loops. The scheduler in CUDA accepts thread blocks and schedules them on the GPU for us. We can keep a Unified Memory frame buffer that is written by the GPU and read by the CPU thanks to CUDA.

When converted to CUDA code, the serial implementation for computing the colour of a surface results in a stack overflow because it can call itself several times. As a result, it has to be altered to compute the colour of the target pixel iteratively. The maximum recursive depth of the serial implementation can be changed by changing the number of iterations.

5.2 Source Code

This code tells us about how we are rendering the image.

```
#include <stdio.h>
#include <stdint.h>
#include <float.h>
#include <time.h>
const int width = 2048;
const int height = width;
const int maxReflect = 5;
struct Color {
  unsigned char b, g, r, a;
};

Color* image;
void writebmpheader(FILE* f, int width, int height) {
  int size = width * height * sizeof(Color);
  struct {
    uint32_t filesz;
    uint16_t creator1;
    uint16_t creator2;
    uint32_t bmp_offset;
  } bmpheader =
    { size + 54, 0, 0, 54};
  struct {
    uint32_t header_sz;
    int32_t width;
    int32_t height;
    uint16_t nplanes;
    uint16_t bitspp;
    uint32_t compress_type;
    uint32_t bmp_bytesz;
    int32_t hres;
    int32_t vres;
    uint32_t ncolors;
    uint32_t nimpcolors;
  } dibheader =
    {40, width, height, 1, 32, 0, size, 0, 0, 0, 0};
  fwrite("BM", 2, 1, f);
```

```
42.  fwrite(&bmpheader, sizeof(bmpheader), 1, f);
43.  fwrite(&dibheader, sizeof(dibheader), 1, f);
44.}
45.void writebmp(const char* filename, const Color* data, int width, int
46.             height) {
47.  FILE* f = fopen(filename, "wb");
48.  if (!f) return;
49.  writebmpheader(f, width, height);
50.  fwrite(data, sizeof(Color), width * height, f);
51.  fclose(f);
52.}
53.
54._device_ _host_ inline float3 operator+(float3 a, float3 b) {
55.  return make_float3(a.x + b.x, a.y + b.y, a.z + b.z);
56.}
57.
58._device_ _host_ inline float3 operator-(float3 a, float3 b) {
59.  return make_float3(a.x - b.x, a.y - b.y, a.z - b.z);
60.}
61.
62._device_ inline float3& operator+=(float3& a, float3 b) {
63.  a.x += b.x;
64.  a.y += b.y;
65.  a.z += b.z;
66.  return a;
67.}
68._device_ _host_ inline float3 cross(float3 a, float3 b) {
69.  return make_float3( -a.z * b.y + a.y * b.z,
70.                      a.z * b.x - a.x * b.z,
71.                      -a.y * b.x + a.x * b.y );
72.}
73._device_ inline float dot(float3 a, float3 b) {
74.  return a.x * b.x + a.y * b.y + a.z * b.z;
75.}
76._device_ _host_ float3 inline operator*(float3 a, float b) {
77.  return make_float3(a.x * b, a.y * b, a.z * b);
78.}
79._device_ _host_ float3 inline operator*(float b, float3 a) {
80.  return a * b;
81.}
82.
83._device_ _host_ inline float sqrlength(float3 v) {
84.  return v.x * v.x + v.y * v.y + v.z * v.z;
85.}
86._device_ _host_ inline float length(float3 v) {
87.  return sqrtf(sqrlength(v));
88.}
```

```
89._device_ _host_ inline float3 normalize(float3 v) {
90.  float invlen = 1 / length(v);
91.  return make_float3(v.x * invlen, v.y * invlen, v.z * invlen);
92.}
93._device_ inline float3 modulate(float3 a, float3 b) {
94.  return make_float3(a.x * b.x, a.y * b.y, a.z * b.z);
95.}
96.struct Ray {
97.  float3 origin, direction;
98.
99.  _device_ float3 getPoint(float t) {
100.         return origin + t * direction;
101.       }
102.     };
103.
104.     struct PerspectiveCamera {
105.       float3 eye, front, right, up;
106.       float fovScale;
107.
108.       _device_ Ray generateRay(float x, float y) const {
109.         float3 r = right* ((x - 0.5) * fovScale);
110.         float3 u = up * ((y - 0.5) * fovScale);
111.         Ray ray = {eye, normalize(front + r + u)};
112.         return ray;
113.       }
114.     };
115.
116.     PerspectiveCamera makePerspectiveCamera(float3 e, float3 f,
   float3 u, float v) {
117.       PerspectiveCamera c = {e, f, cross(f, u), cross(cross(f,u), f)
   , tan(v * 0.5 * 3.1415926 / 180) * 2};
118.       return c;
119.     };
120.     enum g_t {G_SPHERE, G_PLANE} ;
121.     struct IntersectResult {
122.       g_t g_type;
123.       int g_id;
124.       float distance, reflectiveness;
125.       float3 position, normal;
126.     };
127.
128.     struct Sphere {
129.       float3 center;
130.       float radius, sqrRadius;
131.       float3 diffuse, specular;
132.       int shininess;
133.       float reflectiveness;
```

```
134.        float3 lightDir;
135.        float3 lightColor;
136.
137.        _device_ inline bool intersect(Ray& ray, IntersectResult&
   result) const {
138.          float3 v = ray.origin - center;
139.          float a0 = sqrlength(v) - sqrRadius;
140.          float DdotV = dot(ray.direction, v);
141.          if (DdotV <= 0) {
142.            float discr = DdotV * DdotV - a0;
143.            if (discr >= 0) {
144.              result.g_type = G_SPHERE;
145.              result.distance = -DdotV - sqrt(discr);
146.              result.position = ray.getPoint(result.distance);
147.              result.normal = normalize(result.position - center);
148.              result.reflectiveness = reflectiveness;
149.              return true;
150.            }
151.          }
152.          return false;
153.        }
154.
155.        _device_ inline float3 sample(Ray ray, float3 position, float3
   normal) const {
156.
157.          float NdotL = dot(normal, lightDir);
158.          float3 H = normalize(lightDir - ray.direction);
159.          float NdotH = dot(normal, H);
160.          float3 diffuseTerm = diffuse * fmaxf(NdotL, 0.0);
161.          float3 specularTerm = specular * __powf(fmaxf(NdotH, 0.0),
   shininess);
162.          return modulate(lightColor, diffuseTerm + specularTerm);
163.
164.        }
165.      };
166.
167.      Sphere makeSphere(float3 c, float r, float3 d, float3 sp, int sh,
   float re = 0.0) {
168.        Sphere s = {
169.          c, r, r * r, d, sp, sh ,re,
170.          normalize(make_float3(1, 1, 1)),
171.          make_float3(1, 1, 1)
172.        };
173.        return s;
174.      };
175.      struct Plane {
176.        float3 normal, position;
```

```
177.       float scale, reflectiveness;
178.       _device_ inline bool intersect(Ray ray, IntersectResult&
   result) const {
179.         float a = dot(ray.direction, normal);
180.         if (a >= 0.0)
181.           return false;
182.         float b = dot(normal, ray.origin - position);
183.         float d = -b / a;
184.         result.g_type = G_PLANE;
185.         result.distance = d;
186.         result.position = ray.getPoint(d);
187.         result.normal = normal;
188.         result.reflectiveness = reflectiveness;
189.         return true;
190.       }
191.
192.       _device_ inline float3 sample(Ray ray, float3 position, float3
   normal) const {
193.         if (fmodf(fabsf(floorf(position.x * 0.1) + floorf(position.z
   * scale)), 2) < 1)
194.           return make_float3(0, 0, 0);
195.         else
196.           return make_float3(1, 1, 1);
197.       }
198.     };
199.
200.     Plane makePlane(float3 n, float d, float s, float r) {
201.       Plane p = {n, n * d, s, r};
202.       return p;
203.     };
204.     struct RayTracingParam {
205.       PerspectiveCamera camera;
206.       int spheres_n;
207.       Sphere spheres[10];
208.       int planes_n;
209.       Plane planes[10];
210.       int maxReflect;
211.     } cpuparam =
212.     {
213.       makePerspectiveCamera(make_float3(0, 5, 15), make_float3(0, 0,
   -1),
214.                             make_float3(0, 1, 0), 90),
215.       2,
216.       {makeSphere(make_float3(-10, 10, -10), 10,
217.               make_float3(1, 0, 0), make_float3(1, 1, 1), 16, 0.25),
218.        makeSphere(make_float3(10, 10, -10), 10,
219.               make_float3(0, 0, 1), make_float3(1, 1, 1), 16, 0.25)},
```

```
220.         1,
221.         {makePlane(make_float3(0, 1, 0), 0, 0.1, 0.25)}
222.     };
223.
224.     _constant_ RayTracingParam param;
225.     template <typename T>
226.     _device_ inline bool intersect(T* geometries, int n, Ray r,
   IntersectResult& result) {
227.       IntersectResult ir;
228.       bool ok = false;
229.       for (int i = 0; i < n; ++i) {
230.         ir.g_id = i;
231.         if (geometries[i].intersect(r, ir) && ir.distance <
   result.distance) {
232.           result.distance = ir.distance;
233.           result = ir;
234.           ok = true;
235.         }
236.       }
237.       return ok;
238.     }
239.
240.     _device_ inline bool intersect(Ray r, IntersectResult& result) {
241.       bool ok = false;
242.       result.distance = FLT_MAX;
243.       ok = intersect(param.spheres, param.spheres_n, r, result) ||
   ok;
244.       ok = intersect(param.planes, param.planes_n, r, result) || ok;
245.       return ok;
246.     }
247.
248.     _device_ inline float3 sample(Ray r, int g_type, int g_id, float3
   position,
249.                                   float3 normal) {
250.       if (g_type)
251.         return param.planes[g_id].sample(r, position, normal);
252.       else
253.         return param.spheres[g_id].sample(r, position, normal);
254.     }
255.
256.     _device_ inline float3 gpuSample(Ray ray) {
257.       float3 color = make_float3(0, 0, 0);
258.       float reflectiveness = 1.0;
259.       float r = 1.0;
260.       float3 c = make_float3(0, 0, 0);
261.       IntersectResult ir;
262.
```

```
263.        for (int i = 0; i < maxReflect + 1; ++i) {
264.          if (!intersect(ray, ir)) break;
265.          color += reflectiveness * (1 - r) * c;
266.          reflectiveness = reflectiveness * r;
267.          r = ir.reflectiveness;
268.          c = sample(ray, ir.g_type, ir.g_id, ir.position, ir.normal);
269.          if (r > 0) {
270.            Ray newray = {ir.position,
271.                          ir.normal * (-2*dot(ir.normal,ray.direction)) +
    ray.direction};
272.            ray = newray;
273.          } else
274.            break;
275.        }
276.        return color + reflectiveness * c;
277.      }
278.
279.      _global_ void gpuRayTracing(unsigned* out) {
280.        int index = blockIdx.x * blockDim.x + threadIdx.x;
281.
282.        int y = index / width, x = index % width;
283.        float sx = x / float(width), sy = y / float(height);
284.        Ray r = param.camera.generateRay(sx, sy);
285.        float3 c = gpuSample(r);
286.        unsigned char c4[] = {
287.            __saturatef(c.z) * 255,
288.            __saturatef(c.y) * 255,
289.            __saturatef(c.x) * 255,
290.            255};
291.
292.        unsigned ct = reinterpret_cast<unsigned>(c4);
293.        out[index] = ct;
294.      }
295.
296.      int main() {
297.        unsigned* gpuout;
298.        cudaSetDevice(0);
299.        cudaMallocHost(&image, width * height * sizeof(Color));
300.        cudaMalloc(&gpuout, sizeof(Color) * width * height);
301.        cudaMemcpyToSymbol(param, &cpuparam, sizeof RayTracingParam);
302.        clock_t t1 = clock();
303.        gpuRayTracing<<<width * height / 256, 256>>>(gpuout);
304.        cudaMemcpy(image, gpuout, sizeof(Color) * width * height,
    cudaMemcpyDeviceToHost);
305.        clock_t t2 = clock();
306.
307.        printf("%f\n", (t2 - t1) / float(CLOCKS_PER_SEC));
```

```
308.        cudaFree(gpuout);
309.        writebmp("raytracing_cuda.bmp", image, width, height);
310.        cudaFreeHost(image);
311.        return 0;
312.      }
```

This code tells us about what image we are rendering.

```
#include <fstream>
#include <cstring>
#include <iostream>
#include <cmath>
#include <memory>
#include <vector>
#include <cfloat>
#include <ctime>
#include <cstdint>
using namespace std;
const int width = 2048, height = 2048;
const int maxReflect = 5;
inline float clampf(float v, float min, float max) {
  return v < min ? min : v > max ? max : v;
}

struct Color {
  float r, g, b;
  Color(float r_ = 0.0, float g_ = 0.0, float b_ = 0.0) :
      r(r_), g(g_), b(b_){}
  Color(const Color& c) : r(c.r), g(c.g), b(c.b) {}
  Color clamp() const {
    return Color( clampf(r, 0.0, 1.0),
                  clampf(g, 0.0, 1.0),
                  clampf(b, 0.0, 1.0));
  }

  Color operator+(const Color& o) const{
    return Color(r + o.r , g + o.g, b + o.b);
  }

  Color& operator+=(const Color& o) {
    r += o.r; g += o.g; b += o.b;
    return *this;
  }
  Color operator*(float v) const {
    return Color(v * r, v * g, v * b);
  }

  Color modulate(const Color& o) const {
```

```
    return Color(r * o.r, g * o.g, b * o.b);
  }
};
inline Color operator*(float v, const Color& o) {return o * v;}
Color image[width * height];
class BMPWriter {
 public:
  struct bmpheader {
    uint32_t filesz;
    uint16_t creator1;
    uint16_t creator2;
    uint32_t bmp_offset;
  };

  struct bmpinfo{
    uint32_t header_sz;
    int32_t width;
    int32_t height;
    uint16_t nplanes;
    uint16_t bitspp;
    uint32_t compress_type;
    uint32_t bmp_bytesz;
    int32_t hres;
    int32_t vres;
    uint32_t ncolors;
    uint32_t nimpcolors;
  };

  struct bgracolor {
    unsigned char b,g,r,a;
  };

  static void write_header(ofstream& f, int width, int height) {
    bmpheader header;
    bmpinfo info;
    int size = width * height * 4;
    memset(&header, 0, sizeof(header));
    memset(&info, 0, sizeof(info));
    header.filesz = size + 54;
    header.bmp_offset = 54;
    info.header_sz = 40;
    info.width = width;
    info.height = height;
    info.nplanes = 1;
    info.bitspp = 32;
    info.bmp_bytesz = size;
    f.write("BM", 2);
```

```
    f.write((char*)&header, sizeof(header));
    f.write((char*)&info, sizeof(info));
  }

  static void conv_color(bgracolor* out, const Color* in, int size) {
    for (int i = 0; i < size; ++i) {
      Color c = in[i].clamp();
      out[i].r = c.r * 255;
      out[i].g = c.g * 255;
      out[i].b = c.b * 255;
      out[i].a = 255;
    }
  }

  static void write_color(ofstream& f, bgracolor* bgra, int width, int height)
{
    f.write((char*)bgra, width * height * sizeof(*bgra));
  }
  static void write(const std::string& filename, const Color* data, int width,
int height) {

    ofstream f(filename.c_str(), ios::binary);
    if (!f) return;

    write_header(f, width, height);

    bgracolor* bgra = new bgracolor[width * height];
    conv_color(bgra, data, width * height);
    write_color(f, bgra, width, height);

    delete[] bgra;
  }
};

struct Vector {
  float x, y, z;
  Vector(float x_, float y_, float z_) : x(x_), y(y_), z(z_) {}
  Vector(const Vector& r) : x(r.x), y(r.y), z(r.z) {}
  float sqrLength() const {
    return x * x + y * y + z * z;
  }

  float length() const {
    return sqrt(sqrLength());
  }

  Vector operator+(const Vector& r) const {
```

```
    return Vector(x + r.x, y + r.y, z + r.z);
  }

  Vector operator-(const Vector& r) const {
    return Vector(x - r.x, y - r.y, z - r.z);
  }

  Vector operator*(float v) const {
    return Vector(v * x, v * y, v * z);
  }

  Vector operator/(float v) const {
    float inv = 1 / v;
    return *this * inv;
  }

  Vector normalize() const {
    float invlen = 1 / length();
    return *this * invlen;
  }

  float dot(const Vector& r) const {
    return x * r.x + y * r.y + z * r.z;
  }

  Vector cross(const Vector& r) const {
    return Vector(-z * r.y + y * r.z,
                  z * r.x - x * r.z,
                  -y * r.x + x * r.y);
  }

  static Vector zero() {
    return Vector(0, 0, 0);
  }
};
inline Vector operator*(float l, const Vector& r) {return r * l;}

struct Ray {
  Vector origin, direction;
  Ray(const Vector& o, const Vector& d) : origin(o), direction(d) {}

  Vector getPoint(float t) const {
    return origin + t * direction;
  }
};

class Geometry;
```

```
struct IntersectResult {
  const Geometry* geometry;
  float distance;
  Vector position;
  Vector normal;

  IntersectResult() : geometry(NULL), distance(0), position(Vector::zero()),
normal(Vector::zero()) {}
  IntersectResult(const Geometry* g, float d, const Vector& p, const Vector&
n) :
      geometry(g), distance(d), position(p), normal(n) {}
};

struct Scene {
  virtual ~Scene() {}
  virtual IntersectResult intersect(const Ray& ray) const = 0;
};

struct Material {
  float reflectiveness;
  virtual ~Material() {}
  Material(float r = 0.0) : reflectiveness(r) {}
  virtual Color sample(const Ray& ray, const Vector& position, const Vector&
normal) const = 0;
};

struct Geometry : public Scene {
  virtual ~Geometry() {}
  unique_ptr<Material> material;
  Geometry(Material* m = NULL) : material(m) {}
};
struct Sphere : public Geometry {
  Vector center;
  float radius, sqrRadius;

  Sphere(const Vector& c, float r, Material* m = NULL) :
      Geometry(m), center(c), radius(r), sqrRadius(r * r) {}
  IntersectResult intersect(const Ray& ray) const {
    Vector v = ray.origin - center;
    float a0 = v.sqrLength() - sqrRadius;
    float DdotV = ray.direction.dot(v);
    if (DdotV <= 0.0) {
      float discr = DdotV * DdotV - a0;
      if (discr >= 0.0) {
        float d = -DdotV - sqrt(discr);
        Vector p = ray.getPoint(d);
```

```
        Vector n = (p - center).normalize();
        return IntersectResult(this, d, p, n);
      }
    }
    return IntersectResult();
  }
};

struct Plane : public Geometry {
  Vector normal, position;
  std::auto_ptr<Material> material;

  Plane(const Vector& n, float d, Material* m = NULL) :
      Geometry(m), normal(n), position(normal * d) {}

  IntersectResult intersect(const Ray& ray) const {
    float a = ray.direction.dot(normal);
    if (a >= 0.0)
      return IntersectResult();
    float b = normal.dot(ray.origin - position);
    float d = -b / a;
    return IntersectResult(this, d, ray.getPoint(d), normal);
  }
};

struct Union : public Scene {
  unique_ptr<Geometry> geometries[4];
  int n;
  /*
  template <typename... T>
  Union(T... gs) {
    addGeometries(gs...);
  }

  void addGeometries() {

  }
  template <typename... T>
  void addGeometries(Geometry* g, T... gs) {
    geometries.push_back(shared_ptr<Geometry>(g));
    addGeometries(gs...);
  }
  */
  Union(Geometry* g1 = NULL, Geometry* g2 = NULL, Geometry* g3 = NULL,
Geometry* g4 = NULL) {
    n = 0;
    if (g1) geometries[n++].reset(g1);
```

```
    if (g2) geometries[n++].reset(g2);
    if (g3) geometries[n++].reset(g3);
    if (g4) geometries[n++].reset(g4);
  }
  IntersectResult intersect(const Ray& ray) const {

    float minDistance = FLT_MAX;
    IntersectResult minResult;
    for (int i = 0; i <n; ++i) {
      IntersectResult result = geometries[i]->intersect(ray);
      if (result.geometry && result.distance < minDistance) {
        minDistance = result.distance;
        minResult = result;
      }
    }
    return minResult;
  }
};

struct PerspectiveCamera {
  Vector eye, front, right, up;
  float fovScale;
  PerspectiveCamera(const Vector& e, const Vector& f, const Vector& u, float
fov)
      :eye(e), front(f), right(f.cross(u)), up(right.cross(f)),
fovScale(tan(fov * 0.5 * 3.1415926 / 180) * 2) {}

  Ray generateRay(float x, float y) const {
    Vector r = right * ((x - 0.5) * fovScale);
    Vector u = up * ((y - 0.5) * fovScale);
    return Ray(eye, (front + r + u).normalize());
  }
};

inline int iabs(int x) {
  return x < 0 ? -x : x;
}
struct CheckerMaterial : public Material {
  float scale;
  CheckerMaterial(float s, float r = 0.0) : Material(r), scale(s) {}
  Color sample(const Ray& ray, const Vector& position, const Vector& normal)
const {
    if (iabs(floor(position.x * 0.1) + floor(position.z * scale)) % 2 < 1)
      return Color();
    else
      return Color(1.0, 1.0, 1.0);
  }
```

```
};
Vector lightDir = Vector(1, 1, 1).normalize();
Color lightColor = Color(1, 1, 1);
inline float max(float a, float b) {
  return a > b ? a : b;
}
struct PhongMaterial : public Material {
  Color diffuse, specular;
  int shininess;
  PhongMaterial(const Color& d, const Color& sp, int sh, float r = 0.0) :
      Material(r), diffuse(d), specular(sp), shininess(sh) {}

  Color sample(const Ray& ray, const Vector& position, const Vector& normal)
const {
    float NdotL = normal.dot(lightDir);
    Vector H = (lightDir - ray.direction).normalize();
    float NdotH = normal.dot(H);
    Color diffuseTerm = diffuse * max(NdotL, 0.0);
    Color specularTerm = specular * pow(max(NdotH, 0.0), shininess);
    //Color r = lightColor.modulate(diffuseTerm + specularTerm);
    //cerr << r.r << " " << r.g << " " << r.b << endl;
    return lightColor.modulate(diffuseTerm + specularTerm);
  }
};

template <int maxReflect>
struct RayTracer {
  Color operator()(const Scene& scene, const Ray& ray_) const {
    Color color;
    float reflectiveness = 1.0;
    int reflect_times = 0;
    float r = 1.0;
    Color c;
    Ray ray = ray_;
    for (int i = 0; i < maxReflect; ++i) {
      IntersectResult result;
      result = scene.intersect(ray);
      if (!result.geometry) {
        reflect_times = i;
        break;
      }
      color += reflectiveness * (1 - r) * c;
      reflectiveness = reflectiveness * r;
      r = result.geometry->material->reflectiveness;
      c = result.geometry->material->sample(ray, result.position,
result.normal);
      if (reflectiveness > 0) {
```

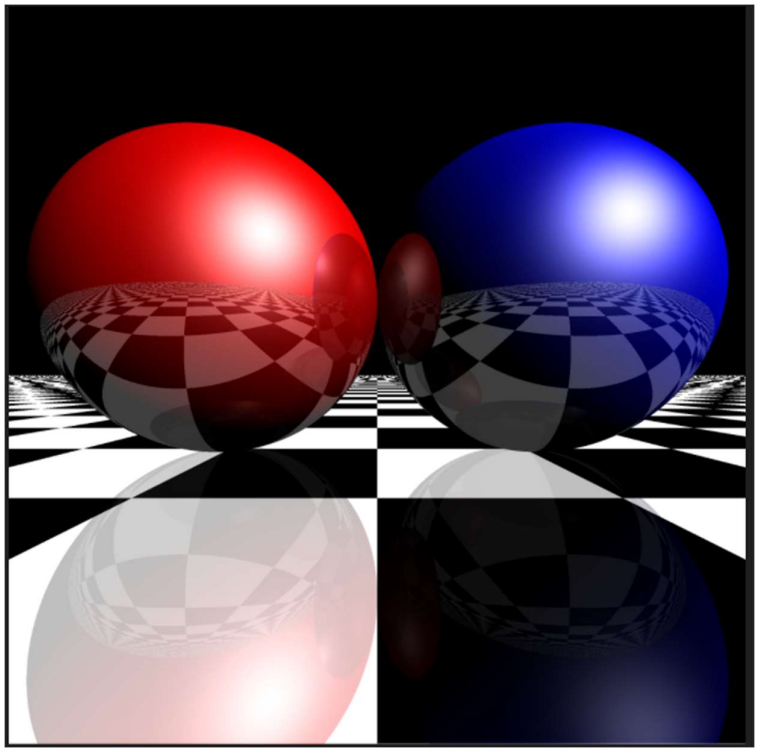
```
        ray = Ray(result.position, result.normal * (-2 *
result.normal.dot(ray.direction)) + ray.direction);
      } else
        break;
    }
    return color + reflectiveness * c;
  }
};

template <typename RenderFuncT>
void render(const Scene& scene, const PerspectiveCamera& camera, const
RenderFuncT& f) {
#pragma omp parallel for num_threads(4)
  for (int i = 0; i < height * width; ++i) {
    int y = i / width;
    int x = i % width;
    float sy = y / float(height);
    float sx = x / float(width);
    Ray ray = camera.generateRay(sx, sy);
    image[i] = f(scene, ray);
  }
}

int main() {
  clock_t t1 = clock();
  render(Union(new Plane(Vector(0, 1, 0), 0, new CheckerMaterial(0.1, 0.25)),
               new Sphere(Vector(-10, 10, -10), 10, new PhongMaterial(Color(1,
0, 0), Color(1, 1, 1),16, 0.25)),
               new Sphere(Vector( 10, 10, -10), 10, new PhongMaterial(Color(0,
0, 1), Color(1, 1, 1), 16, 0.25))),
         PerspectiveCamera(Vector(0, 5, 15), Vector(0, 0, -1), Vector(0, 1,
0), 90),
         RayTracer<maxReflect>());
  clock_t t2 = clock();
  cerr << (t2 - t1) / float(CLOCKS_PER_SEC) << endl;
  BMPWriter::write("raytracing.bmp", image, width, height);
}
```

6. OUTPUT AND PERFORMANCE ANALYSIS

6.1 EXECUTION SNAPSHOTS



6.2 OUTPUT – IN TERMS OF PERFORMANCE METRICS

Several parameters such as samples per pixel, maximum recursive depth, dimensions of output image, block size (CUDA specific) and number of threads (CUDA specific) can be varied to get a definitive speedup achieved by parallelizing serial code using CUDA. The configuration used to test and comparethe serial and parallel implementations are as follows:

System 1	System 2
Processor: Intel i5 – 10210u, 14nm, 1.60 Ghz Base frequency	Processor: Intel i5 – 10300H, 14nm, 2.50 GhzBase frequency
Number of cores: 4, Number of threads: 8	Number of cores: 4, Number of threads: 8
Memory: 8 GB, 2666 Mhz	Memory: 16 GB, 3200 Mhz
Graphics: Intel UHD Graphics 620 Number of Execution Units, Cores, Shaders: 24	Graphics: NVIDIA GeForce GTX 1650, Number of Coder: 896 CUDA cores

1.1. SAMPLES PER PIXEL VS RENDERING TIME

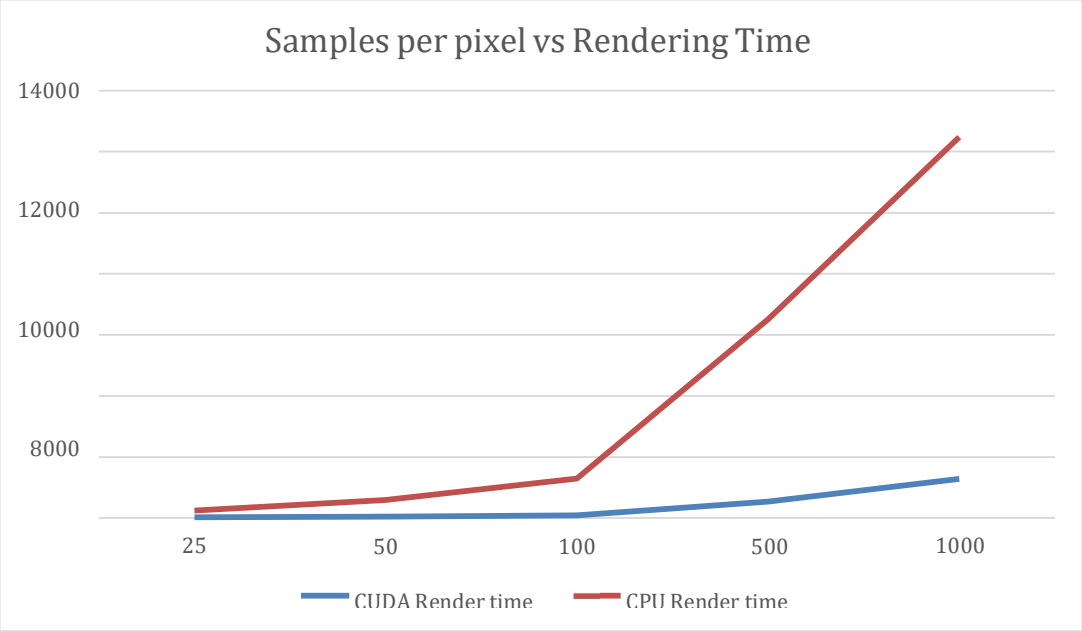
1. Performance comparison with existing works

Maximum recursive depth = 50

Dimensions = 1920 x 1080

CUDA block size = 8 x 8

Samples Per Pixel	CUDA Rendering time (seconds)	CPU Rendering time (seconds)
25	19.8488	217.1873
50	40.5943	542.2117
100	84.1484	1,573.3264
500	538.3345	11,272.6368
1000	1,271.1760	27,704.1128



Average Speedup: 17.2180

2. MAXIMUM RECURSIVE DEPTH VS RENDERING TIME

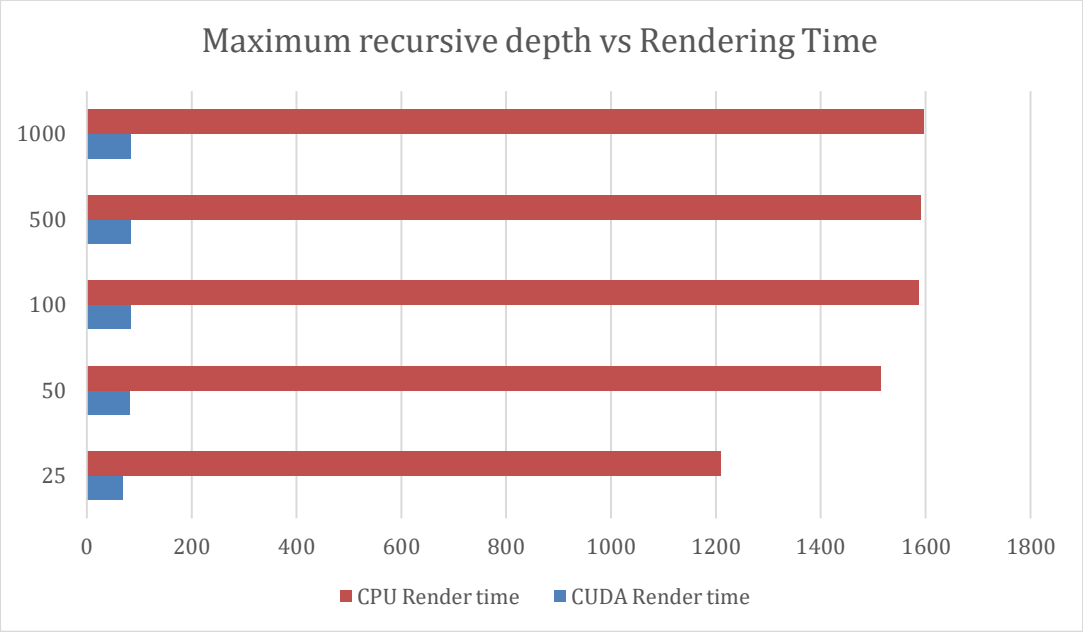
Samples per pixel = 100

Dimensions = 1920 x 1080

CUDA block size = 8 x 8

Maximum Recursive Depth	CUDA Rendering time (seconds)	CPU Rendering time (seconds)
5	68.663	1,208.7892
10	80.9458	1514.2833
20	83.3557	1586.3806
40	83.5947	1590.7325
50	83.6799	1596.7325

Table 2: Maximum Recursive Depth vs Rendering Time



A very small number of pixels benefit from increased maximum recursion depth, this is because a ray undergoes attenuated after every reflection/refraction,

initially the increased recursion depth leads to massive improvements in photo-realism but it reaches stagnation soon.

Average Speedup: 18.8508

3. CUDA BLOCK SIZE VS RENDERING TIME

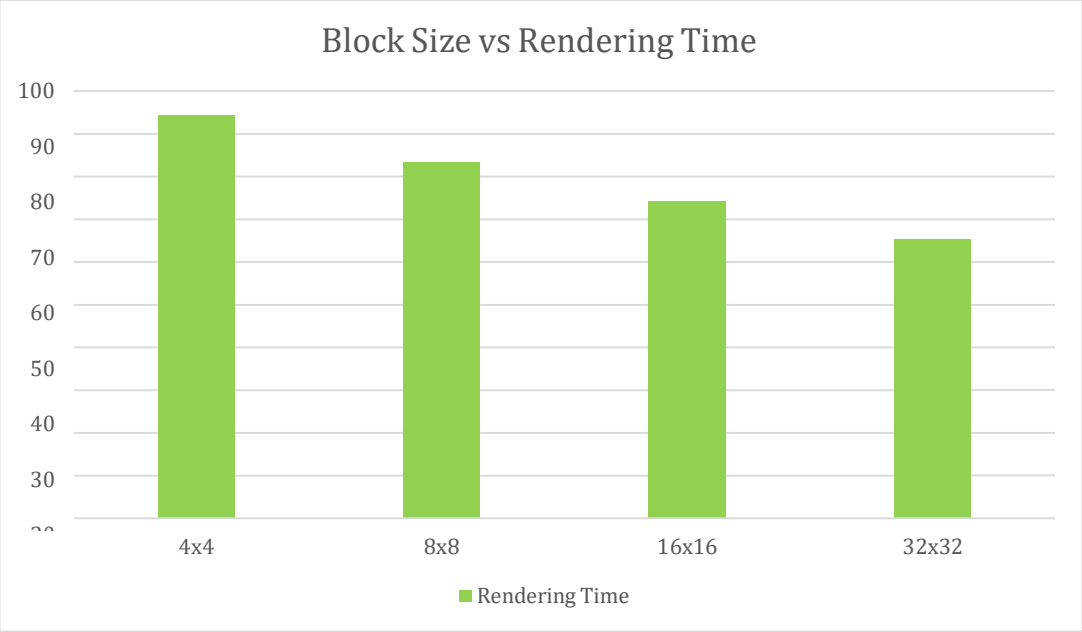
Samples per pixel = 100

Maximum recursive depth = 50

Dimensions = 1920 x 1080

CUDA block size	Rendering Time (Seconds)
4 x 4	94.2454
8 x 8	83.3508
16 x 16	74.2572
32 x 32	65.2454

Table 3: Block size vs Rendering Time



With increasing block size i.e., Parallelization, the rendering time decreases, but we

only have limitednumber of thread (892 for NVIDIA GTX 1650), so re assignment of pixels will lead to stagnation in runtime as the block size increases.

4. DIMENSION SIZE VS RENDERING TIME

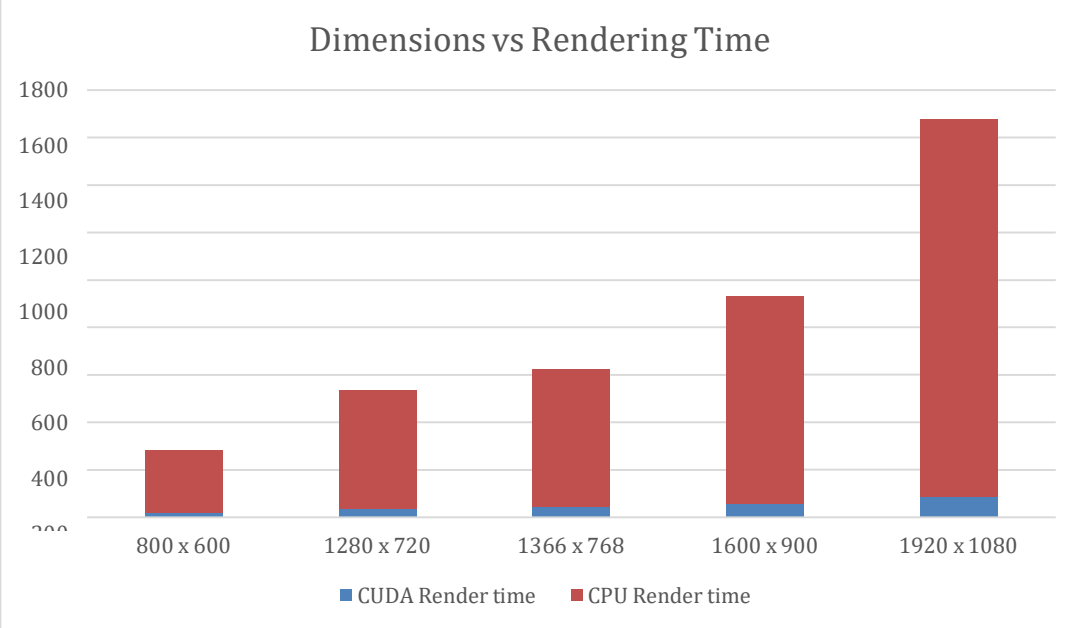
Samples per pixel = 100

Maximum recursive depth = 50

CUDA block Size = 8 x 8

Dimensions	CUDA Rendering time (seconds)	CPU Rendering time (seconds)
800 x 600	19.4233	261.643
1280 x 720	36.2494	498.2781
1366 x 768	42.1525	581.3977
1600 x 900	56.3919	873.3341
1920 x 1080	83.5451	1,591.2672

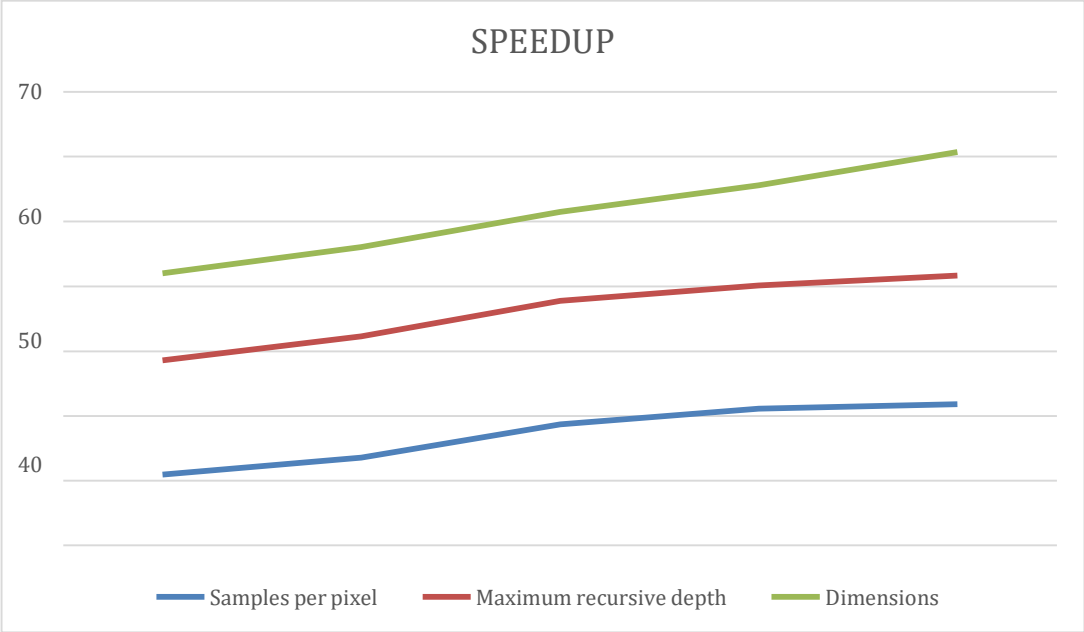
Table 4: Dimensions vs Rendering Time



Average Speedup: 15.1261

6.3 PERFORMANCE COMPARISON WITH EXISTING WORKS

1. SPEEDUP



7. CONCLUSION AND FUTURE DIRECTIONS

In conclusion, we have shown in this report that the right combination of parallelization techniques and using distributed ray tracing we can make a very fast ray tracer. We experimented with various hyperparameters and selected the best of each to further optimize our renderer.

We have also learnt that most CPU algorithms rarely map well to the GPU, and in most cases a newalgorithm needs to be devised, one that follows the spirit of the original one, but whose primitive operations are better suited to the GPU's skillset.

It can be concluded that on varying the above-mentioned parameters, and thus the degree of parallelism,we can only achieve speedup until we have to start reassigning threads for remaining computations. Variations in samples per pixel and dimensions of image (i.e., data size) see the greatest speedups onincreasing degree of parallelism.

8. REFERENCES

- M. M. Taygur and T. F. Eibert, "A Ray-Tracing Algorithm Based on the Computation of (Exact) Ray Paths with Bidirectional Ray-Tracing," in IEEE Transactions on Antennas and Propagation, vol. 68, no. 8, pp. 6277-6286, Aug. 2020, doi: 10.1109/TAP.2020.2983775.

- Adewale AE. (2021). Performance Evaluation of Monte Carlo Based Ray Tracer, Journal of Computational Science

- [1] Adewale AE. (2021). Performance Evaluation of Monte Carlo Based Ray Tracer, Journal of Computational Science
-
- [2] Jan Škoda and Martin Motyčka. (2018). Lighting Design Using Ray Tracing. In 2018 VII. Lighting Conference of the Visegrad Countries (Lumen V4). IEEE, 1–5. https://ieeexplore.ieee.org/document/8521111
-
- [3] Chun-Fa Chang, Kuan-Wei Chen, and Chin-Chien Chuang. (2015). Performance comparison of rasterization-based graphics pipeline and ray tracing on GPU shaders. In 2015 IEEE International Conference on Digital Signal Processing (DSP). 120–123. https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=7251842
-
- [4] Sadraddin A. Kadir and Tazrian Khan. (2008). Parallel Ray Tracing using MPI and OpenMP. Technical Report. Stockholm, Sweden.
-
- [5] Eric Veach and Leonidas J. Guibas.(1997). SIGGRAPH 97 Proceedings (August 1997), Addison-Wesley, pp. 65-76.
-
- [6] Ling-Qi Yan, Soham Uday Mehta, Ravi Ramamoorthi, and Fredo Durand. (2016). Fast 4D Sheared Filtering for Interactive Rendering of Distribution Effects. ACM Trans. Graph. 35, 1, Article 7 (December 2015), 13 pages. DOI:https://doi.org/10.1145/2816814
-
- [7] Balázs Csébfalvi. 1997. A Review of Monte Carlo Ray Tracing Methods. Retrieved April 4, 2020 from http://www.cescg.org/CESCG97/csebfalvi/index.html

9. ADDITIONAL REFERENCES

[8] Xing, Qiwei & Chen, Chunyi & Li, Zhihua. (2021). Progressive path tracing with bilateral-filtering-based denoising. Multimedia Tools and Applications. 80. 1-16. 10.1007/s11042-020-09650-7.

[9] M. M. Taygur and T. F. Eibert, "A Ray-Tracing Algorithm Based on the Computation of (Exact) Ray Paths with Bidirectional Ray-Tracing," in IEEE Transactions on Antennas and Propagation, vol. 68, no. 8, pp. 6277-6286, Aug. 2020, doi: 10.1109/TAP.2020.2983775.

[10] Zellmann, Stefan & Aumüller, Martin & Marshak, Nathan & Wald, Ingo. (2020). High-Quality Rendering of Glyphs Using Hardware-Accelerated Ray Tracing.

[11] Bastien, Plazolles & El Baz, Didier & Spel, Martin & Rivola, Vincent & Gegout, Pascal. (2018). SIMD Monte-Carlo Numerical Simulations Accelerated on GPU and Xeon Phi. International Journal of Parallel Programming. 46. 1-23. 10.1007/s10766-017-0509-y.

[12] Li, Tzu-Mao & Aittala, Miika & Durand, Frédo & Lehtinen, Jaakko. (2018). Differentiable Monte Carlo ray tracing through edge sampling. ACM Transactions on Graphics. 37. 1-11. 10.1145/3272127.3275109.

[13] Razian, Sayed & MahvashMohammadi, Hossein. (2017). Optimizing Raytracing Algorithm Using CUDA. Italian Journal of Science & Engineering. 1. 167-178. 10.28991/ijse-01119.

[14] Geok, Tan & Hossain, Ferdous & Kamaruddin, Mohd & Abd Rahman, Noor Ziela & Thiagarajah, Sharlene & Chiat, Alan & Hossen, Jakir & Liew, Chia. (2018). A Comprehensive Review of Efficient Ray- Tracing Techniques for Wireless Communication. International Journal on Communications Antenna and Propagation (IRECAP). 8. 123. 10.15866/irecap.v8i2.13797.

[15] Saha, Mr & Darji, Mr & Patel, Narendra & Thakore, Darshak. (2016). Implementation of Image Enhancement Algorithms and Recursive Ray Tracing using CUDA. Procedia Computer Science. 79. 516-524. 10.1016/j.procs.2016.03.066.

[16] Al-Oraiqat, Anas & Zori, S.. (2016). Ray Tracing Method for Stereo Image Synthesis Using CUDA. 8.

[17] Khademi Kalantari, Nima & Bako, Steve & Sen, Pradeep. (2015). A Machine Learning Approach for Filtering Monte Carlo Noise. ACM Transactions on Graphics. 34. 122:1-122:12. 10.1145/2766977.

[18] Zwicker, Matthias & Jarosz, Wojciech & Lehtinen, Jaakko & Moon, B. & Ramamoorthi, Ravi & Rousselle, Fabrice & Sen, Pradeep & Soler, Cyril & Yoon, Sung-eui. (2015). Recent Advances in Adaptive Sampling and Reconstruction for Monte Carlo Rendering. Computer Graphics Forum. 34. 10.1111/cgf.12592.

[19] Qin, Yutong & Lin, Jianbiao & Huang, Xiang. (2015). Massively Parallel Ray Tracing Algorithm Using GPU

[20] Kettunen, Markus & Manzi, Marco & Aittala, Miika & Lehtinen, Jaakko & Durand, Frédo & Zwicker, Matthias. (2015). Gradient-Domain Path Tracing. ACM Transactions on Graphics.34. 123:1-123:13. 10.1145/2766997.

[21] Majek, Karol & Bedkowski, Janusz. (2015). Range Sensors Simulation Using GPU Ray Tracing.