Performance Evaluation of CUDA Based Ray Tracer

Gandharv Sachdeva (https://orcid.org/0000-0002-6589-5576)

ABSTRACT

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. The aim of this paper is to present an analysis and comparison of a parallelized CUDA based parallel Ray Tracer against a serial ray tracer. CUDA is a parallel computing platform and application programming interface model created by Nvidia. Parallelization using CUDA allows us to achieve maximum speedups up to 18 times against the serial implementation.

KEYWORDS

Ray tracing, CUDA, photorealistic graphics, rendering, parallelization, ray tracing algorithm, GPU.

1. INTRODUCTION

One of the most important objective 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.

Thus, the main objective of our CUDA based ray tracer is to reduce time to perform the tracing while retaining the properties of creating realistic images.

The paper 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 parallel CUDA based implementation of a serial ray tracer. Finally, in Section 7 and 8, the results and conclusion are discussed.

2. CONTRIBUTION OF WORK

In parallel computing, an Embarrassingly Parallel problem is one where the problem can be divided into sub-problems with little or no extra work. This is also the case where there is no requirement of communication between two different processes to complete the tasks given to respective process.

We notice that in the process of ray tracing, the individual values of each pixel in the final image pane are independent of each other and hence require no communications between any two pixels. This points towards the fact that the problem of ray tracing is an Embarrassingly Parallel problem. This allows us to explore new approaches of parallelization towards this technique. This paper gives a comparative study of serial ray-tracing using CPU and parallelized ray-tracing utilizing GPU (using CUDA).

3. RELATED WORK

Ray tracing has been a field of study for over 25 years now, and its algorithms have been fine-tuned and optimized. Current methods aim to achieve real-time ray tracing methodologies, with speed as a primary concern whilst not doing too much of a trade-off for quality. And so, when dealing with real-time systems the performance does not match the offline rendered (ray-traced /path-traced) images. Offline systems take between a few seconds to a few days even to simulate a few images/graphics scenes. When coming to the accuracy, the offline rendered graphics are far better than graphics rendered in real time. Current ray-tracing systems are capable of producing highly realistic images and parallelizing its implementation decreases the compute cost.

So, it comes as no surprise that optimizing ray tracing algorithms is a widely studied field, so much so that it has its own top-tier conference (HPG: High Performance Graphics). There is generally a whole spectrum of methods to increase the efficiency of ray tracing. Some methods aim to create better sampling methods

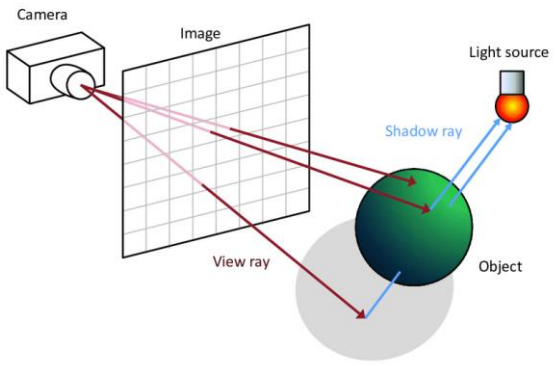
gandharv.sachdeva2019@vitstudent.ac.in
https://orcid.org/0000-0002-6589-5576

(Metropolis Light Transport) [5], while others try to reduce noise in the final image by filtering the output [6] (4D Sheared transform).

4. EXISTING SYSTEM DESCRIPTION

Ray tracing renders a photo-realistic 3D image by tracing the trajectories of light rays through pixels in a view 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 forward tracing, while in backward tracing, rays are traced from the light source to the camera. Backward tracing is much more computationally expensive as every ray emitted by the light source (even if it doesn't reach the camera) has to be traced. Figure 1 describes a simple scene to be ray traced.

Figure 1: A simple ray tracing diagram



Source: Dubla, Piotr. (2011). Interactive Global Illumination on the CPU.

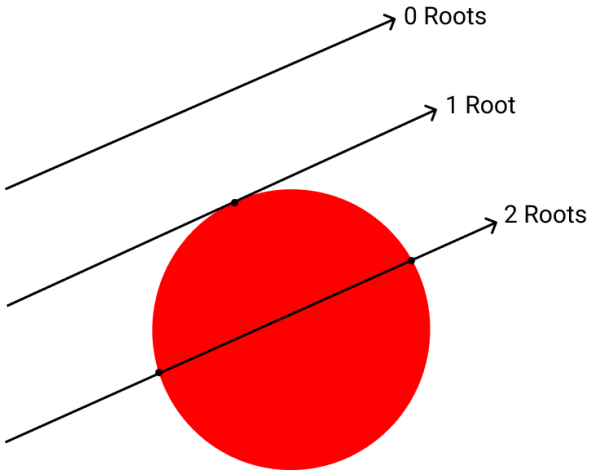
Algorithm 1 is a simple ray tracer algorithm wherein, the rays sent out by camera (for forward ray tracing) can be represented as a line (ray) with an origin (point A) and a direction (Vector B). At any time, a point on the line (ray) can be represented with

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

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 any sphere in our scene at any points.

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

Figure 2: Ray-Sphere interaction



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

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

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

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

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

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

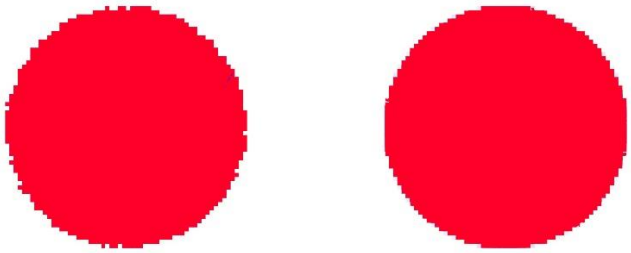
Algorithm 1: Simple ray tracing algorithm

```
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
```

Source: Turner Whitted. (2005). An improved illumination model for shaded display. In ACM SIGGRAPH 2005 Courses (SIGGRAPH '05). Association for Computing Machinery

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 that pixel. The solution to this is to introduce more rays into the scene and to make the ray-object interactions much more random.

Figure 3: Aliased and Anti-aliased ray traced spheres



Aliased Sphere | Anti-Aliased 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 reflections thus rendering a more photo-realistic image. The single ray used to compute the color of a pixel is instead replaced with multiple rays and an average of some of these randomly sampled rays is taken to render the anti-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 reflection and 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;
```

Source: A.E. Adewale. (2021), Performance Evaluation of Monte Carlo Based Ray Tracer. In Journal of Computational Science Education, Volume 12, Issue 1

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

5. RENDERED SCENE

The rendered scene which we used to test and analyze our serial and CUDA based parallel ray tracers consists of spheres placed in a 3D plane, with light sources casting rays into the scene while the camera captures the reflected and refracted rays. The rendered scene is anti-aliased by random sampling of rays through a pixel and then averaging them to compute a color. Three different types of materials have been implemented, namely diffuse or Lambertian, metal and dielectric. The metal objects show fuzzy reflection while the dielectrics show physically accurate phenomenon such as refraction and total internal reflection. The rendered scene is also gamma corrected to get accurate color intensities.

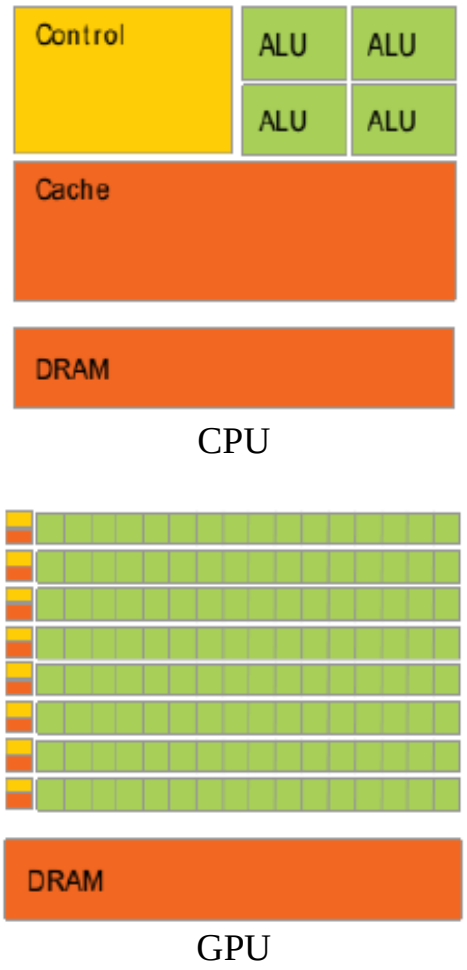
6. SYSTEM ARCHITECTURE

6.1. COMPARATIVE STUDY OF CPU AND GPU ARCHITECTURE

A GPU has a very small cache which is shared among threads but very high bandwidth main memory. A CPU has a relatively gigantic cache for each core, but slow main memory. Indirectly, this implies that sequential/streaming access works best on GPUs. A GPU has 32-way SIMD and no proper branch-prediction which works best when there is relatively no divergent if statements. A CPU tends to have 4 to 8-way SIMD and extremely powerful branch predictors and pre-fetchers which means conditions are no big deal. A GPU also has no cache coherency or any semblance of synchronization-specific hardware. Whereas, CPU is the opposite, and is built with a lot of embedded synchronization primitives. A GPU works best, therefore, with low-contention code.

gandharv.sachdeva2019@vitstudent.ac.in
https://orcid.org/0000-0002-6589-5576

Figure 4: A visual comparison of the difference between a CPU and a GPU



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

6.2. IMPLEMENTATION DETAILS

In the CUDA version, the processing of the image has been done in square batches of pixels (e.g., 8 by 8 batch). Four kernels are called, the first two of which are used for initializing and setting up the type of scene that needs to be rendered and the next two kernels have been used to initialize the random numbers for each thread while rendering the required image respectively.

In the serial implementation, we have used nested for loops to iterate over all of the pixels. In CUDA, the scheduler takes blocks of threads and schedules them on the GPU for us. CUDA allows us to maintain a Unified Memory frame buffer that is written by the GPU and read by the CPU.

The serial implementation to compute the color of a surface when translated to CUDA code results in a stack overflow since it can call itself many times. So, it had to be changed to iteratively compute the color of the desired pixel. The number of iterations can be varied as the max recursive depth of the serial implementation.

7. RESULTS

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 compare the serial and parallel implementations are as follows:

gandharv.sachdeva2019@vitstudent.ac.in
https://orcid.org/0000-0002-6589-5576

Table 1:System Configurations

System 1	System 2
Processor: Intel i5 – 10210u, 14nm, 1.60 Ghz Base frequency	Processor: AMD Ryzen 5 3550H, 12nm, 2.10 Ghz Base frequency
Number of cores: 4, Number of threads: 8	Number of cores: 4, Number of threads: 8
Memory: 8 GB, 2666 Mhz	Memory: 8 GB, 2400 Mhz
Graphics: Intel UHD Graphics 620	Graphics: NVIDIA GeForce GTX 1650
Number of Execution Units, Cores, Shaders: 24	Number of cores: 896 CUDA cores

7.1. SAMPLES PER PIXEL VS RENDERING TIME

Maximum recursive depth = 50

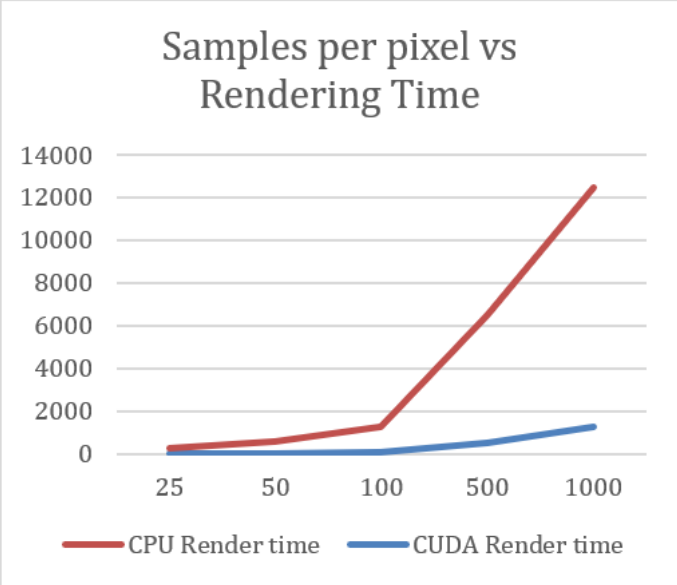
Dimensions = 1920 x 1080

CUDA block size = 8 x 8

Table 2: Samples per pixel vs Rendering Time

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

Figure 5: Samples per pixel vs Rendering Time



Average Speedup: 17.2180

7.2. MAXIMUM RECURSIVE DEPTH VS RENDERING TIME

Samples per pixel = 100

Dimensions = 1920 x 1080

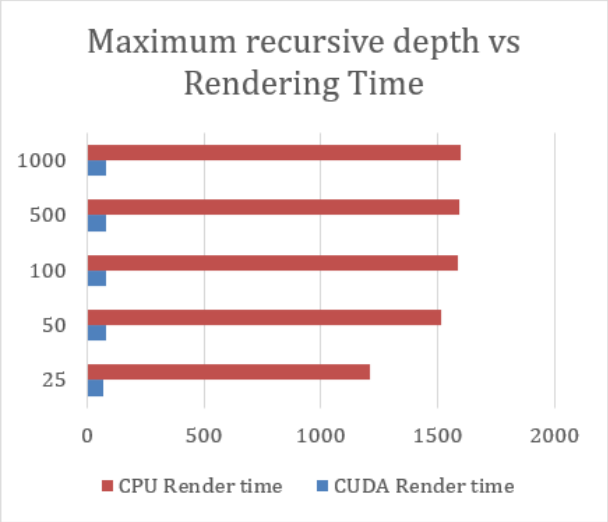
CUDA block size = 8 x 8

Table 3: Maximum recursive depth vs Rendering time

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

gandharv.sachdeva2019@vitstudent.ac.in
https://orcid.org/0000-0002-6589-5576

Figure 6: 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

7.3. CUDA BLOCK SIZE VS RENDERING TIME

Samples per pixel = 100

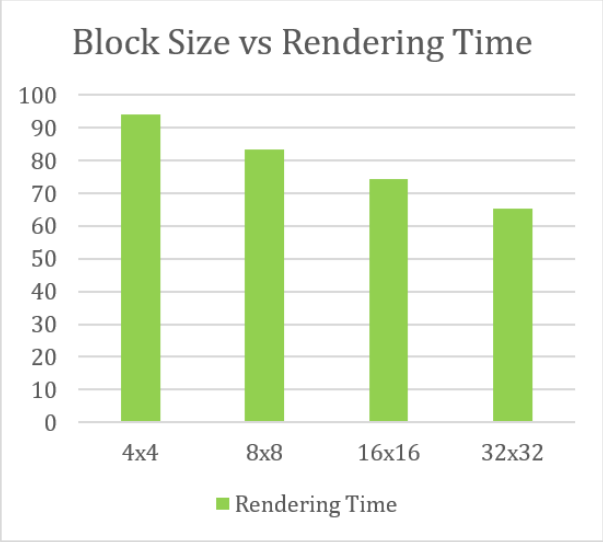
Maximum recursive depth = 50

Dimensions = 1920 x 1080

Table 4: Block size vs Rendering time

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

Figure 7: Block size vs Rendering time



With increasing block size i.e., Parallelization, the rendering time decreases, but we only have limited number of thread (892 for NVIDIA GTX 1650), so re-assignment of pixels will lead to stagnation in runtime as the block size increases.

gandharv.sachdeva2019@vitstudent.ac.in
https://orcid.org/0000-0002-6589-5576

7.4. DIMENSION SIZE VS RENDERING TIME

Samples per pixel = 100

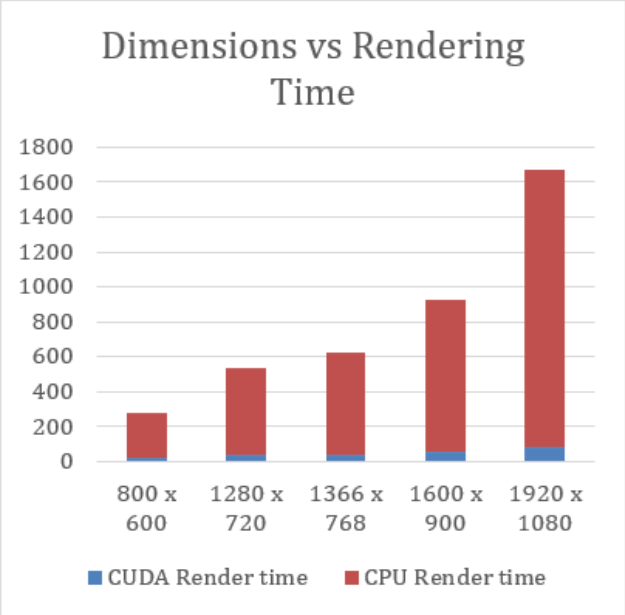
Maximum recursive depth = 50

CUDA block Size = 8 x 8

Table 5: Dimensions vs Rendering Time

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

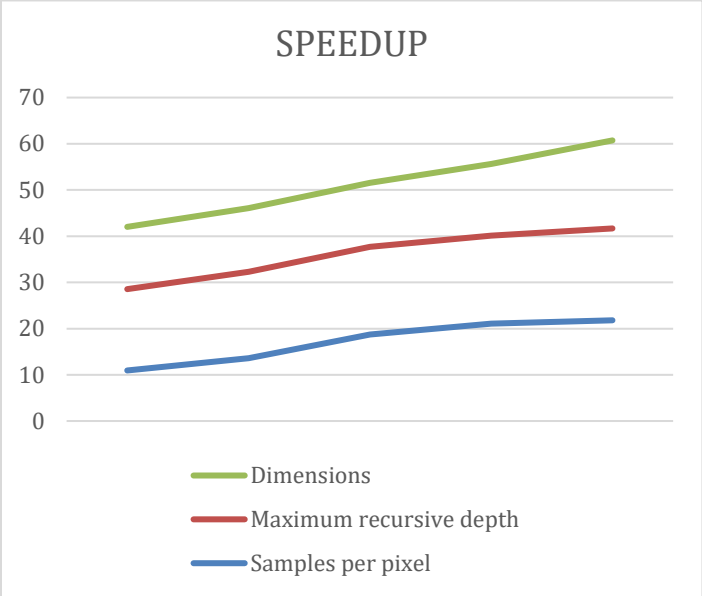
Figure 8: Dimensions vs Rendering Time



Average Speedup: 15.1261

7.5. SPEEDUP

Figure 9: Speedups Achieved



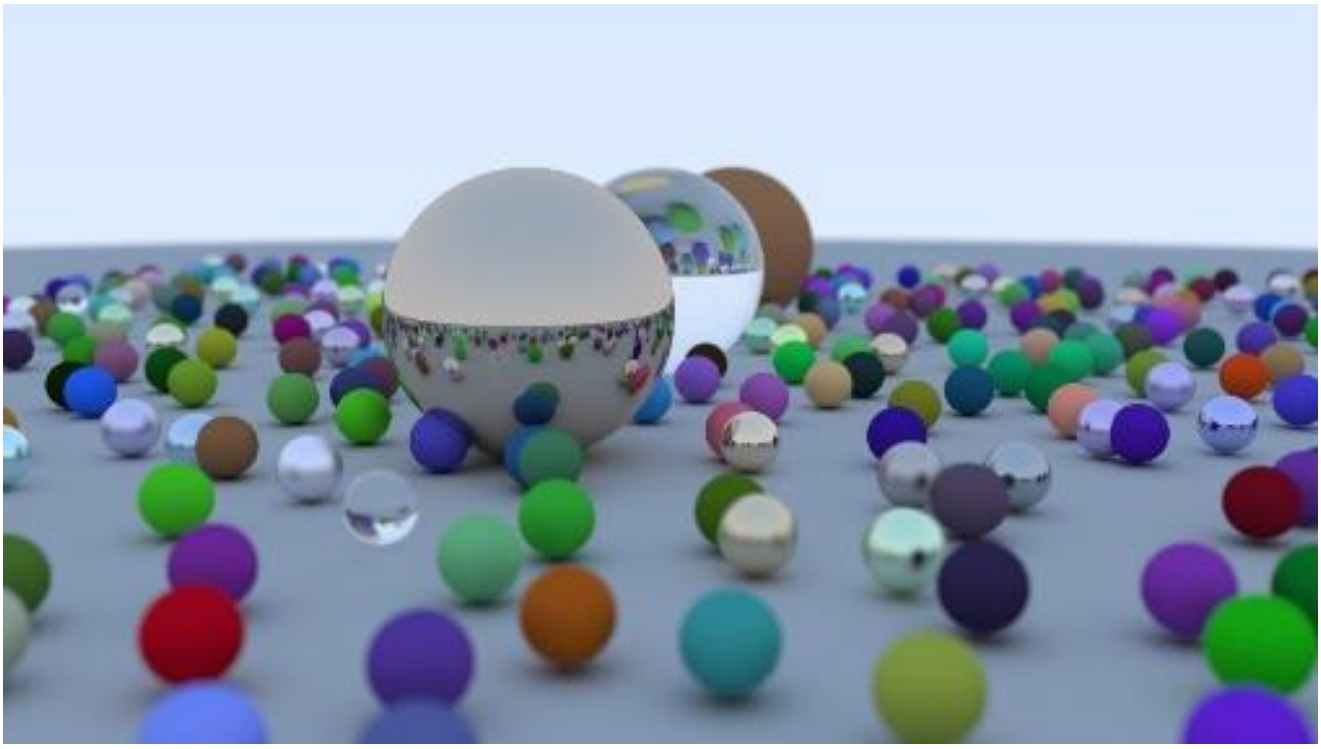
8. CONCLUSION

In conclusion, we have shown in this paper 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 new algorithm 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.

gandharv.sachdeva2019@vitstudent.ac.in
https://orcid.org/0000-0002-6589-5576

Figure 10: A complex ray traced scene, 1920x1080 image with 100 samples per pixel in 8x8 blocks



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., output size) see the greatest speedups on increasing degree of parallelism.

9. REFERENCES

[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