CUDA Path Tracing with Reinforcement learning

PALASH BANSAL, Indraprastha Institute of Information Technology, Delhi

Implementation of Monte Carlo Path Tracing on CUDA with refractive, glossy and metallic material BRDF with approximations for rendering of caustics. Further optimization of the algorithm using reinforcement learning by learning the light transport [1].

CCS Concepts: • **Computer graphics**; • **GPU Computing**; • **Reinforcement learning** → Artificial Intelligence;

Additional Key Words and Phrases: Path tracing, Monte Carlo, Reinforcement Learning, CUDA

ACM Reference format:
Palash Bansal. 2018. CUDA Path Tracing with Reinforcement learning. 1, 1, Article 1 (April 2018), 13 pages.
https://doi.org/0000001.0000001

1 INTRODUCTION

Path tracing is a computer graphics Monte Carlo method of rendering images of three-dimensional scenes such that the global illumination is faithful to reality. Fundamentally, the algorithm is integrating over all the illuminance arriving to a single point on the surface of an object. This illuminance is then reduced by a surface reflectance function (BRDF) to determine how much of it will go towards the viewpoint camera. This integration procedure is repeated for every pixel in the output image. Since this process is executed independently for every pixel, it is embarrassingly parallel in nature, and can be implemented in CUDA to get great improvement in performance.
Furthermore, on combining path tracing with reinforcement learning, we will be able to memoize more data about the path of lights and will be able to simulate the light paths much faster than traditional monte-carlo path tracing. This technique is discussed in [1].

2 LITERATURE REVIEW

2.1 Path Tracing

Path tracing is a challenging task, that requires solving the rendering equation.
Theoretically it would take infinite time to solve the equation properly, but we use many approximations to speed up this process and get a good enough looking scene image. In path tracing, light rays are spawned from the camera and simulated in the scene to get the intensity at that camera point.
Since this is an embarrassingly parallel task, using GPU with CUDA makes a lot of sense, and it should speed up the process by tremendous amounts.
The most common parallel algorithm is monte-carlo path tracing, where one ray is sent from the camera, which

XXXX-XXXX/2018/4-ART1 $15.00
https://doi.org/0000001.0000001

bounces around randomly. This process is repeated to get the average intensity over millions of samples.

2.2 Reinforcement Learning

Reinforcement learning is an area of machine learning inspired by behaviourist psychology, concerned with how software agents ought to take actions in an environment so as to maximize some notion of cumulative reward. It is a kind of approximate dynamic programming, where agents try a lot of possible scenarios and save the resulting outcomes.
The environments are formulated as a Markov Decision Process, and while using reinforcement learning, it doesn't require much knowledge of the MDP, most of the things are derived from hit and try.

2.3 Path tracing with reinforcement learning

Ken Dahm and Alexander Keller in the paper Learning Light Transport the Reinforced way pointed out the structural similarities between the Light transport and the Q learning equation, and hence formalized a way to learn the important light paths using hit and trial and reinforcement learning by modelling a MDP and using Monte-Carlo path tracing to feed the process.
This is basically a replacement/improvement over importance sampling and related methods, where we feed the light position into the algorithms and sometimes lose the details and quality of the scene.
Since Path Tracing is already a parallel task, this could be accomplished on GPU using CUDA without much overhead.
[1] has shown that the equations of reinforcement learning and light transport simulation are related integral equations. Based on this correspondence, a scheme to learn importance while sampling path space has been derived. This approach is helpful in a consistent light transport simulation algorithm that uses reinforcement learning to progressively learn where light comes from. As using this information for importance sampling includes information about visibility, too, the number of light transport paths with zero contribution is dramatically reduced, resulting in much less noisy images within a fixed time budget.

3 ALGORITHM

In the algorithm, first one ray will be shot from every pixel of the camera to the scene, this ray will bounce around in the scene depending on the surface functions and final color will be accumulated and shown on the screen. This will be implemented in a work-queue fashion where each processor will get the job from the memory and process the ray till it encounters a light source or the environment/ambient.
With time a lot of rays will be shot, and the final color will be averaged to get a reduced noise image(Progressive improvement).
To incorporate reinforcement learning, on each step of the bounce, whenever we calculate the ray intersection, we update Q-learning table depending on the current light intensity. And subsequently while calculating the bounce direction, this Q-table is queried to randomly get one of the optimal bounce direction.

4 IMPLEMENTATION PLAN

Firstly the path tracing code will be coded on GPU with CUDA, taking reference from the original CPU code developed by me earlier. The speedup will be compared with reference to that CPU code. This will be further improved to support caustics and area-lights for soft shadows.
Since in path tracing, the ray can end at any time, I plan to use a work queue based strategy while implementing

ALGORITHM 1: Augmenting path tracing with RL

Input: The scene, and camera.
ray = rayForCurrentPixel(); *bounces* = 0;
repeat
  r2 = intersect(*ray*, *scene*);
  updateQTable(*r2*, *color*);
  if *isLightOrEnvironment(r2)* **then**
    break and return *color*;
  else
    direction = calculateDirectionProportionalToQ();
    color = calculateColor(*r2*, *direction*);
    ray = generateRay(*r2*, *direction*);
  end
until *bounces* > *maxBounces*;

the kernels to simulate the rays (this might change).
After successful implementation of Path Tracing on CUDA, will further move on improving performance by adding reinforcement learning. Inspiration will be taken from the algorithm template provided in [1].

5 DELIVERABLE

5.1 Intermediate 1

Implemented a path tracer on CUDA. The speedup achieved is 120x on GTX 970 as compared to a very similar CPU based implementation by myself.

5.1.1 Basic Code structure. In the main function, the code uses a renderengine to render one frame of the image, this data after being copied to the CPU, gets send to an OpenGL context as a texture, which is then rendered on the screen/window.
There are separate RenderEngines for the CPU and GPU code, and both take the same input, therefore hot switching is very easy. Complete code is written by me. Based on some initial code for structure of ray-tracer given by the instructor during the Computer Graphics course at IIIT Delhi.

5.1.2 Data Structures. In the CPU implementation classes were made for all parts of the project. In the corresponding CUDA version, structures were used and data copied to the device from the host in the beginning. Some main structures and classes:

- **RenderEngine:** Contains the base code for rendering a frame. There are 2 classes: RenderEngine and RenderEngine_GPU, which are the single point for a driver program.
- **Camera:** Contains the camera layout position and precomputed basis vectors. This is passed as a struct to the kernel.
- **World:** Contains the complete world layout with all the objects and light sources. Also has the helper functions for checking intersections with the scene, calculating parameter values, hitpoints etc.
- Ray: Main structure for a ray to simulate, contains the properties for origin, direction, hitpoint, normal etc.
- **Sphere:** Main structure for the objects right now, contains various properties of the sphere including material properties. Also has the member helper functions to check for intersection etc.

5.1.3 GPU implementation. GPU implementation details

- Each pixel is subsampled 64 times. For each ray, a GPU thread is assigned. So, each pixel is spread across 2 warps.

- Currently there is one kernel to simulate the light paths
- At every frame, the current bitmap is calculated by using the previous bitmap as the seed, which is then copied back to the host to display.
- Each block is divided into 3D threads, where xDim=8, yDim=8 and zDim is variable depening on the number of threads per block. The X and Y space is used directly to subsample a pixel, and the threads in the Z dimension are different pixel.s
- The blocks in the grid are arranged in a 1D fashion depending on the number of pixels to render.
- In the kernel, first the ray direction is calculated. Stochastic jittering is used to vary the starting position of the ray while sub sampling. After calculating the ray direction, the light path is simulated upto 64 bounces, for small scenes 8 bounces came to be enough.
- At each intersection with an object the color is multiplied.
- Finally after getting the final color in all the threads, the colors are reduced to one value per-pixel which is then written to the image-bitmap in the global memory.
- Since the threads used in subsampling are in a multiple of 32, which is the WARP_SIZE, I am using shuffle instructions to perform reduce in a warp of 32 threads. Then using shared memory to add values in subsequent warps. This final value is then written to the bitmap.

5.1.4 BRDF. The implementation supports the following BRDF for monte-carlo path tracing in both GPU and CPU implementation.

- **Diffuse:** The most common one. Samples the rays randomly in all the directions
- **Specular:** A completely specular object will behave as a mirror, refecting rays properly. There is a parameter to set the level which will make the object more or less shiny, by randomly choosing with diffuse.
- **Emissive:** For the area lights.
- **Glossy:** For the mettalic effect, samples the rays on a probability destribution based on an angle from the reflected ray.
- **Dielectric:** For the refractive objects, rays can pass through these objects and gets refracted based on the eta.

5.1.5 Speedup. For a basic scene designed to test the performance and the working of the algorithm, the speedup is great.
For all the experiments, the MAX_LEVELS = 8, i.e the rays are simulated upto 8 bounces.

Configuration for the experiment: Image size: 360x240, Samples per pixel: 64, Area lights: 1, Objects: 6

- Time taken to render 1 frame on GPU(GTX 970): 51ms
- Time taken to render 1 frame on CPU(i7 4.4GHz): 144308ms
- **Speedup: 2880x**

Configuration for the 2nd experiment: Image size: 640x480, Samples per pixel: 64, Area lights: 1, Objects: 9

- Time taken to render 1 frame on GPU(GTX 970): 251ms
- Time taken to render 1 frame on CPU(i7 4.4GHz): 819200ms
- **Speedup: 3264x**

Configuration for the 3nd experiment: Image size: 640x480, Samples per pixel: 64, Area lights: 2, Objects: 13

- Time taken to render 1 frame on GPU(GTX 970): 428ms
- Time taken to render 1 frame on CPU(i7 4.4GHz): 1041920ms
- **Speedup: 2435x**

5.2 Final deliverable

The current version is an optimized path tracer with online reinforcement learning.

5.3 Reinforcement learning

In the paper [1], reinforcement learning concept is explained, not the proper algorithm or any implementation details. So the complete implementation is designed my myself keeping GPU and CUDA concepts in mind.

5.3.1 State space. This is defined differently from the paper. The state space is the voxel representation of the entire scene. Where each sphere/object may lie in one or more voxels.

5.3.2 Action space. This is also defined differently from the paper. The actions from each state is a set of 8 directions represented by the 8 octants in the 3d space.

5.3.3 Updation. The QTable is updated everytime a ray hits a new object. The value is very large in case of a light and slightly less than the illumination in case of any other.

5.3.4 Access of action and state. Getting the Qindex(index in the q table for any point) and direction octant(the octant for a direction vector) is O(1), the algo is in the snippet.

5.4 Comparisons

5.4.1 Monte-Carlo Path tracing vs MC Path tracing with RL. Reinforcement learning perform a lot better as compared to simple monte-carlo based path tracing for finding the light path in case of challenging conditions.

5.4.2 Issues. In case of reflective surfaces and some dielectric surfaces, we get some artifacts due to voxelisation of state space.

5.4.3 Other implementation. Being a recent paper, the only other implementation I could find was by Tanya Chaudhary and Aashay Mittal, IIIT Delhi. Their results were very different from this, as is shown in an image.

5.5 Optimizations

5.5.1 Shuffle and shared memory. For adding the colors within a warp and across 2 warps.

5.5.2 SPP design. The warp and ray structure are designed in a way that rays from the same pixel lie in the same warp and therefore will perform in lock step for very long.

5.5.3 Random texture. Custom pseudo-random function is designed for GPU. For the seed a random texture is used that is precomputed once for every 64 samples per pixel, and also uses the timestamp.

5.5.4 RL algorithm. The complete algorithm is designed for fast access to state space and action space.

5.6 Code

The entry point of the GPU code is render_loop in render_engine.cu.
To run the code, the project need to be compiled using cmake with glew installed. It will be linked automatically. See CMakeLists.txt for details.
Reinforcement learning related functions are in rl_helper_functions.h and material and color computation functions are in material_functions.h
All the parameters related to the program can be changed from pathtracer_params.h

5.7 Code snippets

```
c = warpAddColors(c);
__shared__ float3 val[MAX_THREADS_IN_BLOCK/(SAMPLE*SAMPLE)];
val[threadIdx.z] = make_float3(1,0,0);
__syncthreads();
if(p==SAMPLE-1 && q==SAMPLE-1)
    val[threadIdx.z] = c;
__syncthreads();

if(p==0 && q==0){
    c = c+val[threadIdx.z];
    c = clamp(c/(SAMPLE*SAMPLE), 0, 1);
    int index = (i + j*cam.size.x)*3;
    float f = 1.0f / (steps+1);
    bitmap[index + 0] = (unsigned char) ((bitmap[index + 0] * (f * steps) + 255 * c.x * f));
    bitmap[index + 1] = (unsigned char) ((bitmap[index + 1] * (f * steps) + 255 * c.y * f));
    bitmap[index + 2] = (unsigned char) ((bitmap[index + 2] * (f * steps) + 255 * c.z * f));
}
```

Fig. 1. Snippet shows reduction of color and write to bitmap. Initial c is the thread color.

```
inline float3 warpAddColors(float3 val) {
#pragma unroll
    for (int offset = warpSize / 2; offset > 0; offset /= 2) {
        val.x += __shfl_xor(val.x, offset);
        val.y += __shfl_xor(val.y, offset);
        val.z += __shfl_xor(val.z, offset);
    }
    return val;
}
```

Fig. 2. Adding colors in a warp using shuffle xor instructions.

5.8 Some renders

Some sample renders, max number of bounces is 64:

REFERENCES

[1] K. Dahm and A. Keller. Learning Light Transport the Reinforced Way. *ArXiv e-prints*, January 2017.

Received March 2018

```
if(sp_mat == LIGHT){
    if(ENABLE_RL) updateQTable(q_table, last_index, dir_oct, clamp01(length(wor.spheres[sphere].col)));
    break;
}else {
    if(ENABLE_RL) updateQTable(q_table, last_index, dir_oct, clamp01(q_table[q_index].max * 0.8f));
    if (sp_mat == DIELECTRIC) {
        ray.dir = shadeDielectric(ray, seed, wor, c, sphere);
    } else if (sp_mat == GLOSSY) {
        ray.dir = shadeGlossy(ray, seed, wor, c, sphere);
    } else if (sp_mat == REFLECTIVE && Random_GPU(seed) < wor.spheres[sphere].param) {
        ray.dir = shadeReflective(ray, seed, wor, c, sphere);
    } else {
```

Fig. 3. QTable update routine in the path tracer.

```
float3 direction;
unsigned char t_index;
QNode q = q_table[q_index];
if(ENABLE_RL&&steps>3&&i) for(int li=0; li<8; li++) {
    direction = shadeDiffuse(ray, seed);
    t_index = getDirectionOctant(direction);
    if (q.v[t_index] > 0.75 * q.max) {
        break;
    }
}else direction = shadeDiffuse(ray, seed);
```

Fig. 4. Computing direction by considering Qtable values for diffuse objects.

```
int seed = 341*q + 253 * p * 8 + ( rand_tex[(i + j*cam.size.x)%(cam.size.x * cam.size.y)]) + 349*steps + clk;
__device__ __inline__ inline float Random_GPU(int &s) {
    s = (s * 238905729 + 729962271);
    return 1.0f*((s)&0x7fffff)/0x7fffff;
}
```

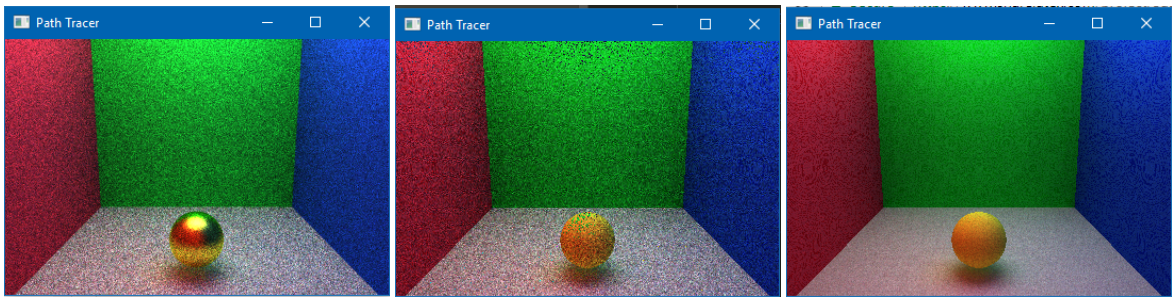
Fig. 5. Pseudo-Random function and seed heuristics

```
__device__ unsigned int getQIndex(float3 &r) {
    return clamp(static_cast<uint>((floor(r.x) + MAX_COORD) * MAX_COORD * MAX_COORD * 4
                                   + (floor(r.y) + MAX_COORD) * MAX_COORD * 2
                                   + floor(r.z) + MAX_COORD), (uint) 0, (uint) MAX_COORD * MAX_COORD * MAX_COORD * 8);
}

__device__ unsigned char getDirectionOctant(float3 &r) {
    return static_cast<unsigned char>(r.z > 0 ?
                                      (r.y > 0 ? (r.x > 0 ? 0 : 1) : (r.x > 0 ? 2 : 3)) :
                                      (r.y > 0 ? (r.x > 0 ? 4 : 5) : (r.x > 0 ? 6 : 7)));
}

__device__ __inline__ inline void
    updateQTable(QNode *&q_table, unsigned int &last_index, unsigned char &last_dir_quad, float newVal) {
        q_table[last_index].v[last_dir_quad] = q_table[last_index].v[last_dir_quad] * (1 - ALPHA) + newVal * ALPHA;
        q_table[last_index].max = fmax(q_table[last_index].v[last_dir_quad], q_table[last_index].max);
}
```

Fig. 6. Helper Functions for reinforcement learning operations



(a) CPU render 320x240, 144 sec, 64 samples per pixel

(b) GPU render 320x240, 0.051 sec, 64 samples per pixel

(c) GPU render 320x240, 0.5 sec, 640 samples per pixel

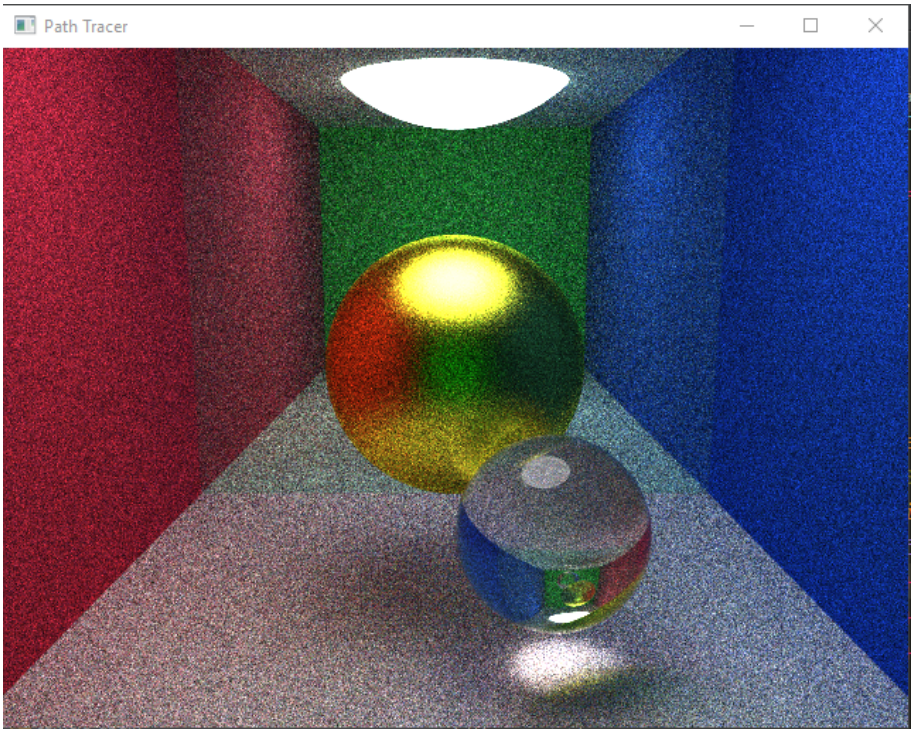


Fig. 8. CPU render 640x480, 819 sec, 64 samples per pixel

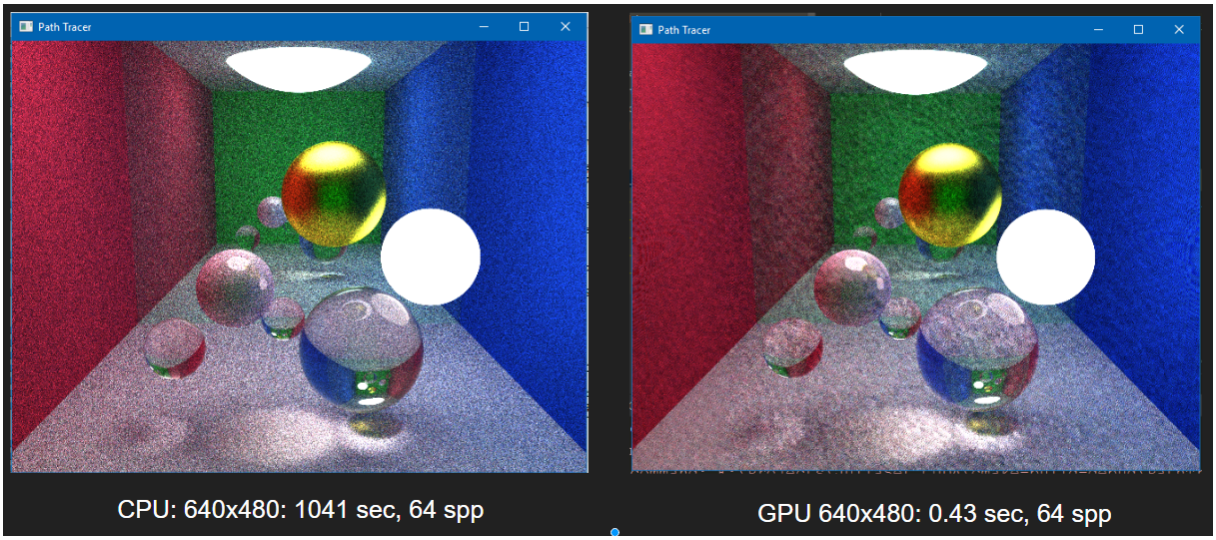


Fig. 9. CPU vs GPU comparison

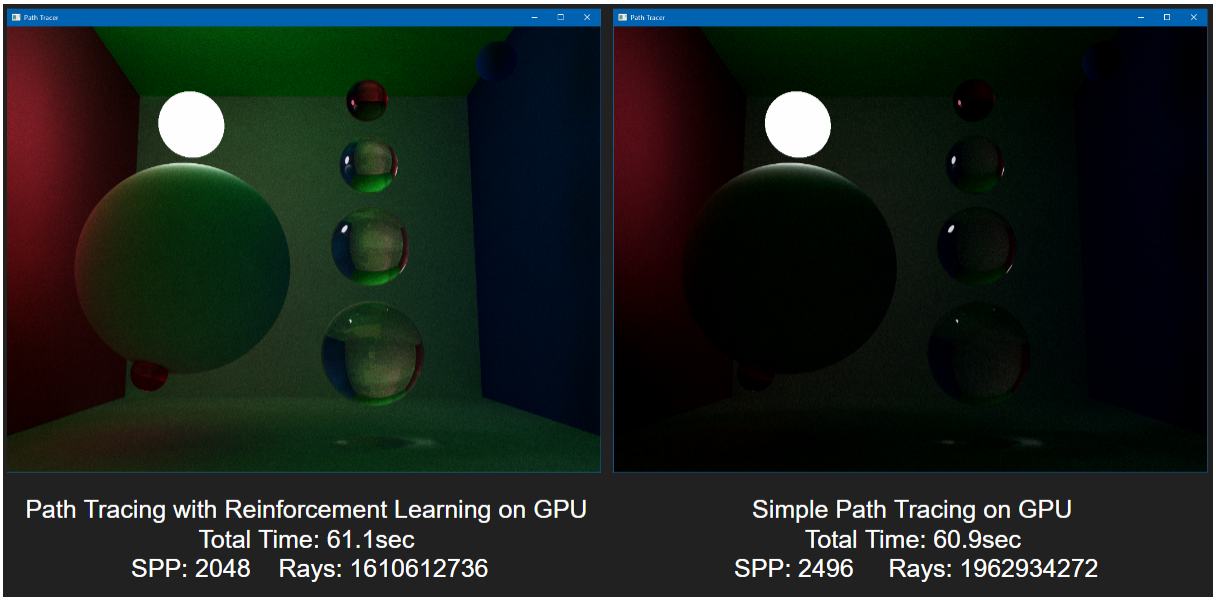


Fig. 10. Path tracing vs Reinforcement learning on the same scene

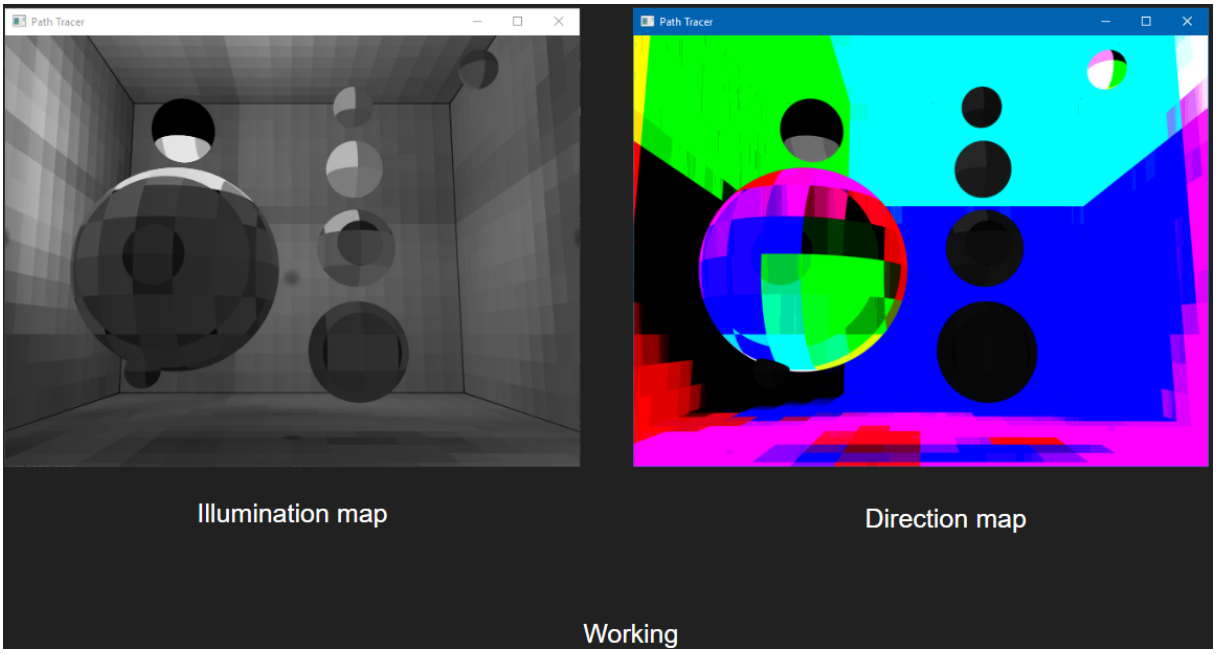


Fig. 11. The illumination map and the direction map where each color(including black) denote a direction

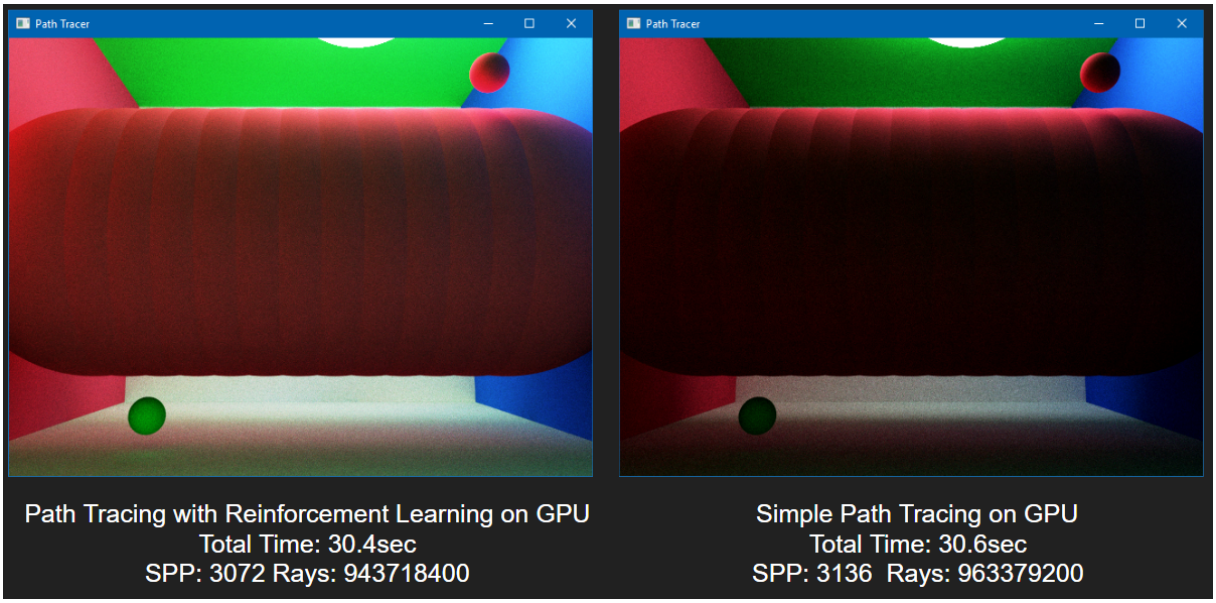


Fig. 12. Path tracing vs Reinforcement learning on the same scene

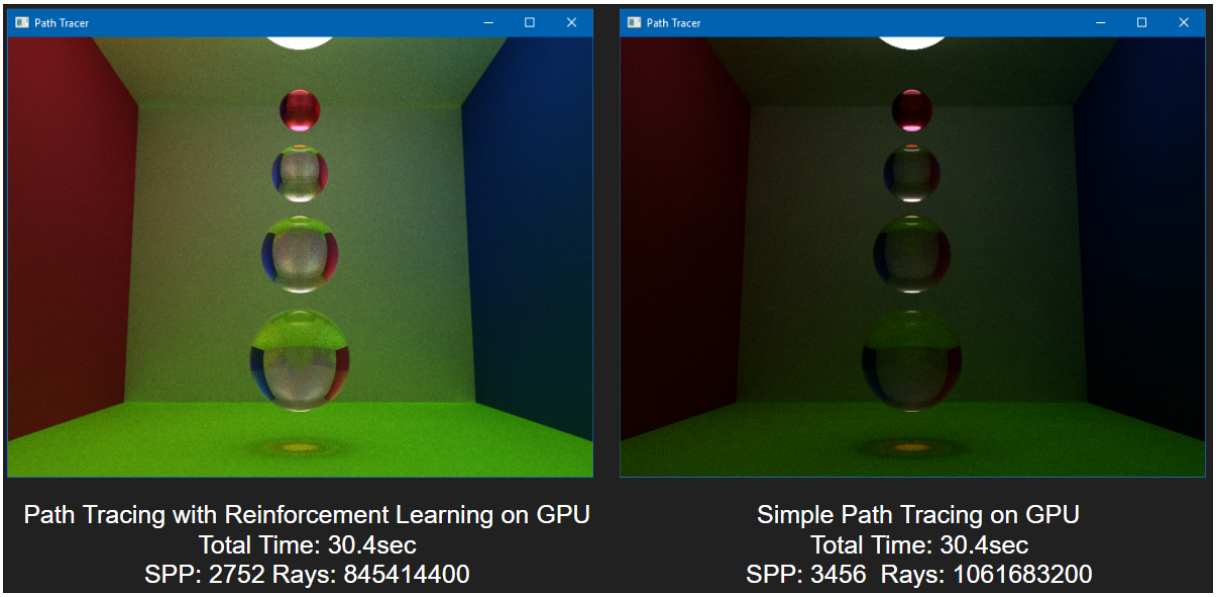


Fig. 13. Path tracing vs Reinforcement learning on the same scene

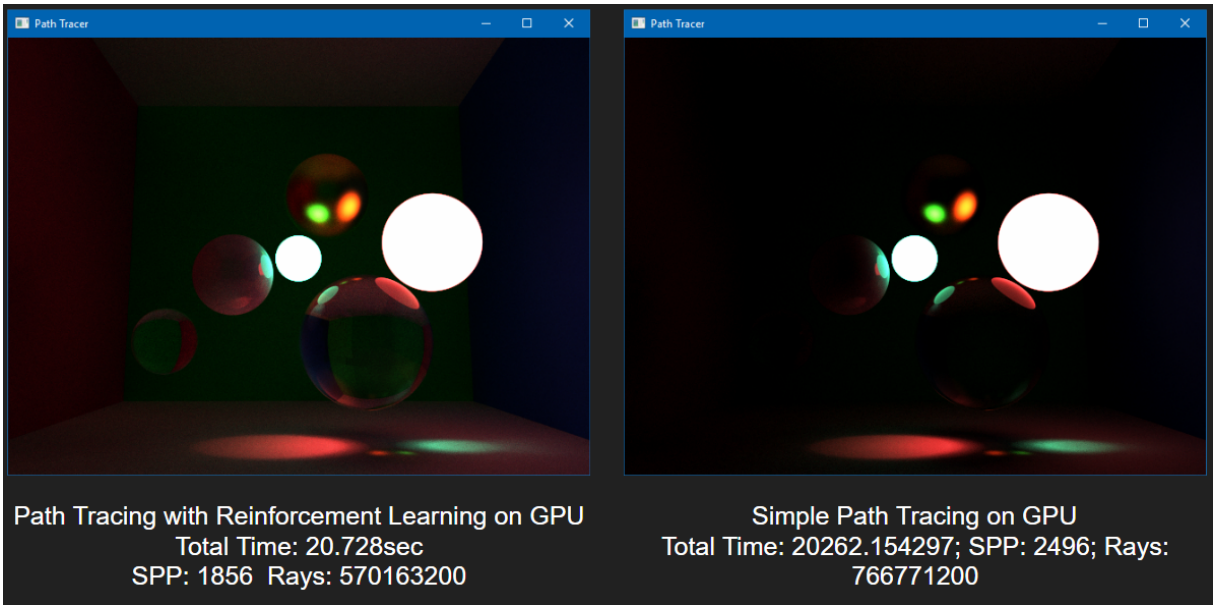


Fig. 14. Path tracing vs Reinforcement learning on the same scene

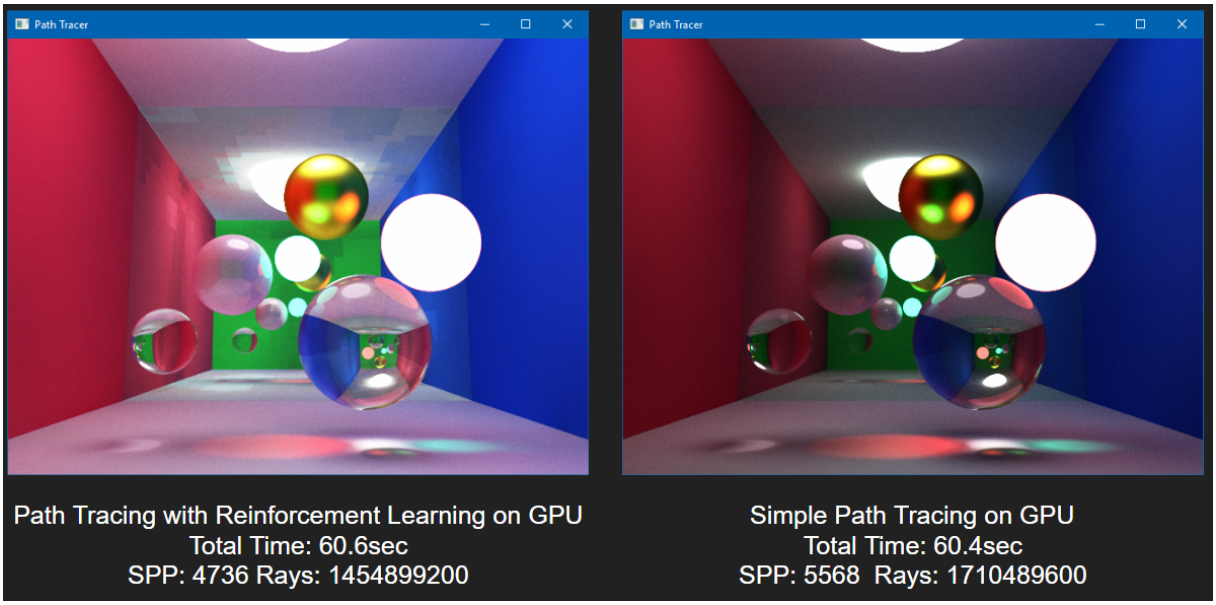


Fig. 15. Path tracing vs Reinforcement learning on the same scene. In the perfectly reflective wall, we can see some artifacts that are not present in monte-carlo path tracing.

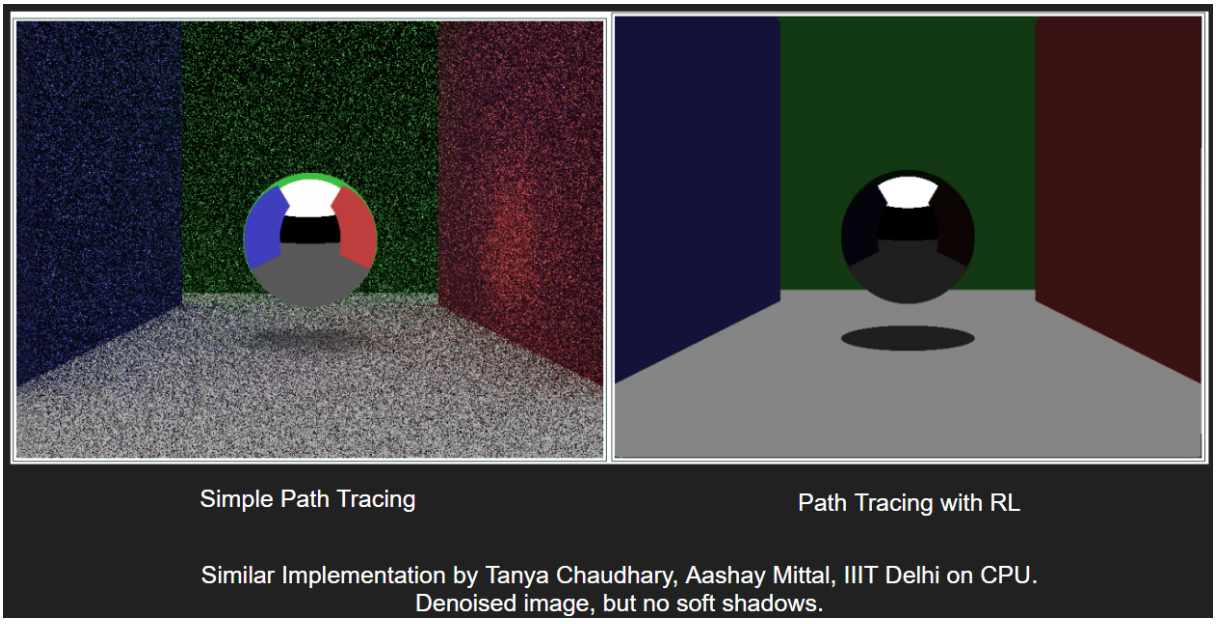


Fig. 16. Only implementation other than that of Nvidia that I could find.