Optimizing Monte Carlo Path Tracer using CUDA programming: EE451 PROJECT REPORT

By: VANSH DHAR (2505769073)
ABHISHEK MISHRA (3432233339)

Introduction

Rendering or image synthesis is the process of generating a photorealistic or non-photorealistic image from a 2D or 3D model by means of a computer program. The resulting image is referred to as the render. Ray tracing is a rendering method that shoots rays from the camera to the scene. In a basic ray tracer, the rays reflect from the objects to the light source to compute the color value at the pixel position. Ray tracing can model the actual behaviour of light in a scene, but is computationally intensive because of exponential branching.

A Path Tracer on the other hand tries to solve the rendering equation proposed by Kajiya, which is as follows:

$$I(x,x') = g(x,x')[\varepsilon(x,x') + \int \rho(x,x',x'') \cdot I(x',x'')dx'']$$

Where x, x',x'' are points in the 3D environment, I(x,x') is the intensity of light falling on x from x', g(x,x') is a geometry term that models occlusion between x' and x, ε(x,x') is the intensity of light source emitted from x' to x. ρ(x,x',x'') is the surface reflectivity function(BRDF) of surface point x', given x and x''.

Path tracing builds on ray tracing, but unlike it, the albedo is calculated by the lights arriving from all possible directions to a pixel, each of which continuously bounced off from other objects in the scene. This simulates a more natural behaviour of light leading to effects such as soft shadows, caustics, ambient occlusion, and indirect lighting. Due to its accuracy and unbiased nature, path tracing is used to generate reference images when testing the quality of other rendering algorithms. But being a more sophisticated version of ray tracing, makes path tracing on of the most computationally expensive ways of rendering a 3D world.

Thus, in our project we built a path tracing rendering system and parallelized its workload using CUDA programming. We also finetuned the parameters that could affect its performance: Image Resolution, Block Size, Register usage per thread. We also varied the 3D world being renderered by our system to analyze how that would affect the workload and performance.

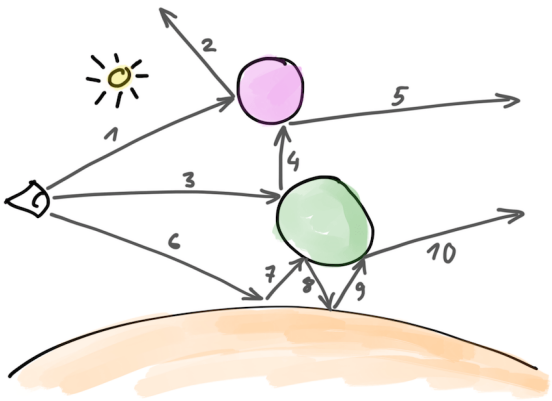

Fig: A Path Tracing Diagram

Algorithm:

The Algorithm below is executed by each thread in our CUDA program. Note: For simplicity purposes the number of threads is equal to the number of pixels in the image being rendered.

Algorithm 1 : Algorithm executed by each Thread/Pixel

$i \leftarrow ThreadIdx.x + BlockDim.x * BlockIdx.x$
$j \leftarrow ThreadIdx.y + BlockDim.y * BlockIdx.y$
Ensure: $i < Image\_Width$ and $j < Image\_Height$
$N \leftarrow No\_of\_Samples$
$Color_{ij} \leftarrow 0$
while $N \neq 0$ **do**
  $i+ = Random(-1, 1)$
  $j+ = Random(-1, 1)$
  $ray = Compute\_Ray(Camera\_Info, i, j)$
  $Color_{ij}+ = Compute\_Color(3D\_Scene, ray)$
  $N \leftarrow N - 1$
end while
$FrameBuffer[j, i] = Color_{ij}/No\_of\_Samples$

In the Algorithm, we first define the pixel position that the thread represents, then we make sure the value doesn't exceed the given image size. We then run a loop for the number of times we are going to sent a ray through one pixel. Afterwards, we move each rays position slightly from its original location (This helps us achieve a higher quality image) and then we compute the explicit ray equation based on the camera position in the 3D world and the pixels location. We then compute the color produced by that ray traveling the 3D world using the concept of Monte Carlo Path Tracing. Finally we take the average color over all our samples and the store the value.

Context/Implementation Details:

The in-depth working of the path tracer might be to lengthy to explain in this report and since our focus is on parallelization and performance optimization we will explain only those concepts of our rendering system that will help the reader understand our parallelization efforts. If more context is needed you can refer to Peter Shirley's "Ray Tracing in One Weekend Series", which served as an inspiration for building our Path Tracing Rendering System.

Ray-Object Intersection Computation: We defined a class to represent 3D geometric vectors, which includes vector operations. A ray is implemented as $p(t) = A + t * B$ where p is a 3D position on a 3D line whose origin is A and direction is B. Firstly, we place the camera at (0, 0, 0) and define the coordinates of our screen space. For a given sphere with center C = (Cx, Cy, Cz) and radius R, if our ray intersects the sphere, there will be some t for which p(t) will satisfy the equation:

$$t^* \cdot dot(B,B) + 2t \cdot dot(B, A - C) + dot(A - C, A - C) - R^* = 0.$$

We can use this to find t and color the pixel that hits the sphere with a designated color[2].

Surface Texture Computation: Lambertian/Diffuse objects which don't emit light take the

color of the surroundings and adjust it with their own color. Light reflects from a diffuse surface in random directions. We define a sphere with unit radius at the hit point and pick any random point in this sphere to get the direction of the reflected ray.

Metals are smooth, and they don't scatter rays randomly. The reflected ray for metals, have the direction of R which can be calculated as:

$$V - 2 * dot(V, N) * N$$

where V is the incoming light direction and N is the normal at the intersection.

Dielectrics are clear materials like glass, water etc. When a light ray hits them, it divides into a reflected and a refracted ray. On interaction with these materials we randomly choose either a reflected or refracted ray. In case it is the reflected ray, we use, while for the refracted ray, we apply the Snell's law: $n.\sin(\theta) = n'.\sin(\theta')$, where n and n' are refractive indices of the respective materials.

Camera Positioning: We place our camera at a position, look-from and point it to another position, look-at. To handle tilt of the camera, we specify an up-vector. The orthonormal basis to denote camera's coordinates, (u, v, w) can be calculated using cross-products on the vectors defined.

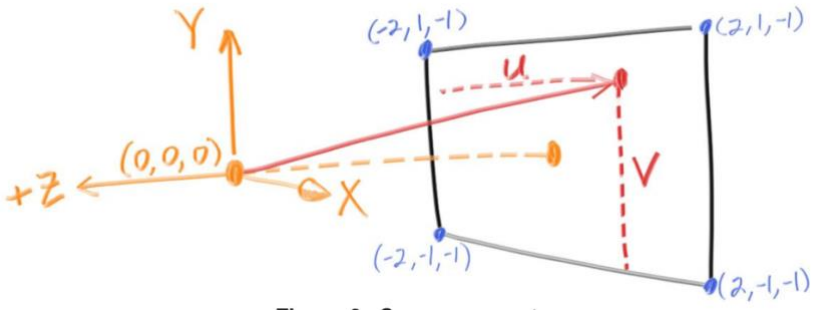

Fig: Camera Geometry

Monte Carlo and Importance Sampling: We can achieve the statistical estimation of Kajiya's rendering equation[5] by using the Monte Carlo integration. We compute four probability density functions (pdf) for a ray scattered on a diffuse surface, a ray directly going to an area light, a ray going to a sphere, and a ray reflecting on a metal surface. Each pdf is then used to compute attenuation with the corresponding reflected ray. We further implemented importance sampling by combining sampling methods along weighted average of the related pdfs to reduce noise in the rendered image.
The generalized form of the Monte Carlo integration is shown by Jensen[8] :

$$\int_{x \in S} g(x)\, d\mu \approx (1/N) * \sum_{i=1}^{N} (g(x_i)/p(x_i))$$

We use this estimation to calculate the color value of each pixel. Each randomly reflected ray is weighted with its pdf. For the rays scattered on a diffuse surface, the pdf is a $\cos(\theta)$ where the angle is between the normal and the reflected ray on the hemisphere:

$$pdf_{diffuse}(direction) = \cos\theta / \pi$$

For the direct light sampling, we map the pdf over a hemisphere knowing the area of the light, angle, and the distance between the intersection and light source:

$$pdf_{arealight}(direction) = distance\ (p, q)^2 / (\cos \alpha \cdot A)$$

The pdf for the sampling through a glass sphere is simply a solid angle which is equal to an area of a unit sphere as follows:

$$pdf_{sphere}(direction) = 1 / solid\_angle$$

$$solid\_angle = \int_0^{2\pi} \int_0^{\theta_{maz}} \sin\theta \ d\theta d\phi$$
$$= 2\pi(1 - \cos\theta_{max})$$

We can use an implicit sampling or a uniform pdf for the metal surface:

$$pdf_{metalbox}(direction) = 1$$

Finally, we mix the probability densities of the diffuse/cosine sampling and direct light sampling for the importance sampling to reduce noise in the image by taking a weighted average of these pdfs:

$$pdf_{mix}(direction) = 0.5 * pdf_{diffuse}(direction) + 0.5 * pdf_{arealight}(direction)$$

The importance sampling shows significant improvement on noise reduction as seen below (with 100 samples/pixel).

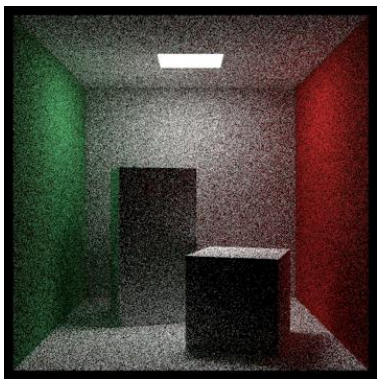

Fig: Random Sampling

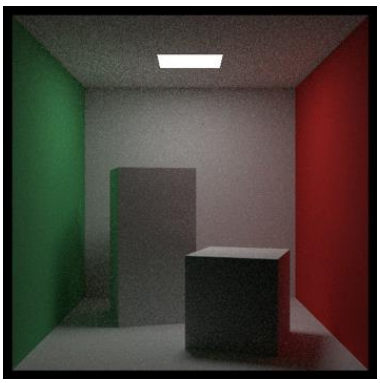

Fig: Important Sampling

Parallelization Strategy:

As we can conclude from the implementation details that, each ray's computation is independent from one another. So, in order to utilize parallel computing we divide the workload for rendering an image such that each pixel's color is computed by an individual thread. The algorithm that each thread executes is explained in detail in the algorithm section.

Note: The amount of workload each thread computes depends on the 3D scene being rendered by the camera. Therefore, in order to optimize the performance of our rendering system based on the hardware(GPU) being used we conducted experiments on three different 3D scenes.

In Scene1, all the rays are forced to bounce inside a square cube, making their path computation/workload comparatively the same. In Scene2, some rays directly intersect with

the background while some do bounce off between spheres, and thus there is an imbalance in the workload being performed by the threads. In Scene3, we have a scene similar to scene2 but the number of spheres has increased drastically (with some of them even moving), this causes an extreme imbalance of workload between those rays that directly intersect with the background and those rays that bounce of between spheres.

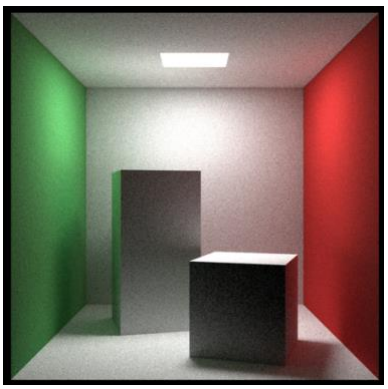
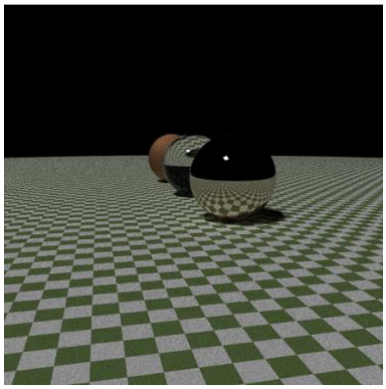
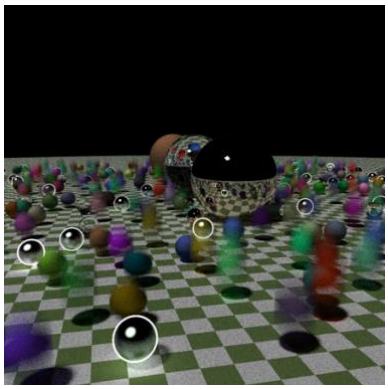

Fig: Scene1 Scene2 Scene3

Hypothesis:

In our parallelization efforts, we will utilize Data Parallelism techniques as the same task is being performed at different positions in the 3D Space. Note: Even though each thread does a completely independent task and thus, the block size shouldn't affect performance, but its our belief that since block size determines the amount of resources needed for each block and also how many blocks can be concurrently run on a single Streaming Multiprocessor:

1. Using a Block with several threads will lead to high resource utilization and will affect occupancy per Streaming Multiprocessor. Using only a few threads per Block will lead to most of the time being consumed by block/wrap management and overhead.

2. There should be a range of threads per block which would result in optimal utilization of hardware(GPU) resources resulting in maximum SpeedUp.

3. This might vary for images of different resolutions.(As total number of blocks change)

Experimental Setup:

In this section we provide details about our experimental setup.

- **Platform Used:**

  Hardware Specification:
  - System Type: 64-bit operating system, x64-based processor
  - Installed RAM: 16.0 GB
  - Processor: AMD Ryzen 9 5900HX with Radeon Graphics 3.30 GHz
  - GPU: Nvidia GeForce RTX 3070 Laptop version

- 5120 cores at 1.29 GHz,
- 48 wraps per SM, 64 SM total
- 8GB GDDR6 graphics memory w/ 256 bit memory bus at 14 GHz

Software Specification:

- OS Name: Windows 11
- Programming language: C++, CUDA
- Programming IDE: Microsoft Visual Studio v1.66.0
- Profiling Tool: Nvidia Nsight Compute v2022.1.1

- **Parameter Range:**
  - Image Resolution: 256x256, 512X512, 1024x1024
  - Number of Samples: 8
  - Block Size: 8x8, 16x16, 32x32, 64x8, 128x1, 256x1
  - Number of Objects in 3D world: 4, 7, 404

Results and Analysis:

In this section we discuss the results obtained while finetuning for the best parameter range. Note: We kept the Number of Samples fixed to reduce the complexity of our experimentation.

Scene1:

In this scene, where workload is evenly distributed, from the results we observe that SpeedUp remains close to the same for different resolutions and even for different block sizes in a particular resolution the speedup remains in the range of 450x-650x. This is highly plausible since as we increase the number of threads, i.e, the resolution size of the image the workload is also increase proportionally because each thread does equal amount of work. Thus, the change in Speedup for Different Resolution isn't Substantial. Also we, observe that '64x8' block size leads the best SpeedUp followed by '32x32' block size for all image resolutions. Thus, when rendering a scene in which each pixel has similar computation being done and the users hardware resembles closely to the one we use. We would recommend setting the block size to either '64x8' or '32x32' for maximum SpeedUp.

We also observe, that branch efficiency is close to 86%, this is due to the fact that importance sampling for one ray might choose the reflected path while for the other ray choose the light path. Furthermore, Nvidia Profiling Tool helped us understand that Theoritical Occupancy was limited due the registers utilised per thread. But since our CUDA program was written efficiently we were able to have our achieved occupancy be close to the theoretical limit for most of the different parameter configuration.

Note: In the SpeedUp Comparison Chart, different colors represent different resolution sizes. While the Y-axis represents different block sizes.

Resolution	Block size	No. of Samples	Parallel	Serial	SpeedUp	Branch Efficiency	Achieved/Theoretical Occupancy
256x256	8x8	8	0.085	46.515	547.2352941	86.88	62.42/66.67
256x256	16x16	8	0.09	46.515	516.8333333	86.99	69.61/83.33
256x256	32x32	8	0.084	46.515	553.75	86.96	58.7/66.7
256x256	64x8	8	0.082	46.515	567.2560976	86.96	55.84/66.67
256x256	128x1	8	0.095	46.515	489.6315789	86.96	68.12/83.33
256x256	256x1	8	0.085	46.515	547.2352941	86.96	67.23/83.33
Resolution	Block size	No. of Samples	Parallel	Serial	SpeedUp	Branch Efficiency	Achieved/Theoretical Occupancy
512x512	8x8	8	0.354	188.884	533.5706215	86.93	61.29/66.67
512x512	16x16	8	0.355	188.884	532.0676056	87.02	72.20/83.33
512x512	32x32	8	0.318	188.884	593.9748428	87.01	59.57/66.67
512x512	64x8	8	0.305	188.884	619.2918033	87.01	56.68/66.67
512x512	128x1	8	0.392	188.884	481.8469388	87.01	72.39/83.33
512x512	256x1	8	0.346	188.884	545.9075145	87.01	68.62/83.33
Resolution	Block size	No. of Samples	Parallel	Serial	SpeedUp	Branch Efficency	Achieved/Theoretical Occupancy
1024x1024	8x8	8	1.423	757.105	532.0484891	86.88	62.42/66.67
1024x1024	16x16	8	1.434	757.105	527.9672245	86.91	73.53/83.33
1024x1024	32x32	8	1.281	757.105	591.0265418	86.99	60.72/66.67
1024x1024`	64x8	8	1.229	757.105	616.0333605	86.99	58.34/66.67
1024x1024	128x1	8	1.58	757.105	479.1803797	86.99	73.74/83.33
1024x1024`	256x1	8	1.386	757.105	546.2518038	86.99	68.94/83.33

Fig: Scene1 Experimentation Results

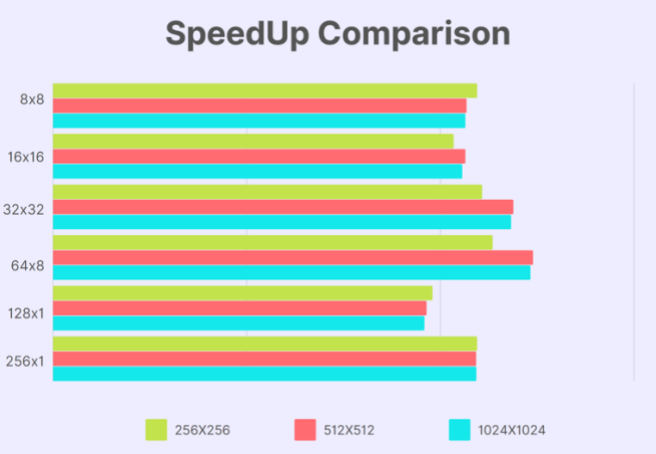

Fig: Scene1 SpeedUp Comparison

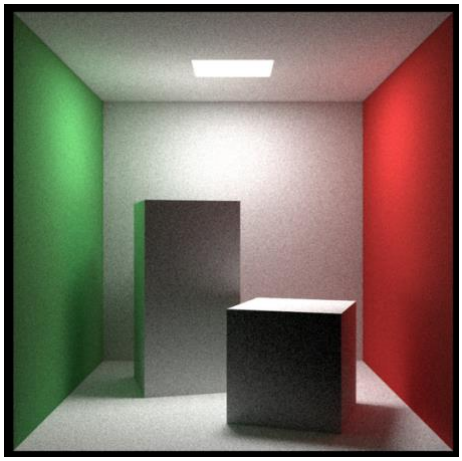

Fig: Scene1

Scene2:

In this scene, where there is a workload imbalance, from the results we observe that SpeedUp scales with image Resolution. We achieve a maximum SpeedUp of 1015x for image of Size 256x256, while we achieve a maximum SpeedUp of 1433x for image of Size 512x512 and for image size of 1024x1024 we achieve a maximum SpeedUp of 1663x. This behaviour is observed because increasing the image resolution increases the serial complexity proportionally, but due to the smart resource utilization in our CUDA program the increase in parallel complexity is significantly lower thus there is a scalabilty in SpeedUp as the resolution size increases. Also we, observe that '256x1' block size leads the best SpeedUp for all image resolutions. Thus, when rendering a scene in which pixels have varying amount of computations being done and the users hardware resembles closely to the one we use. We would recommend setting the block size to '256x1' for maximum SpeedUp.

Resolution	Block size	No. of Samples	Parallel	Serial	SpeedUp	Branch Efficiency	Achieved/Theoretical Occupancy
256x256	8x8	8	0.005	4.059	811.8	91.12	45.55/66.67
256x256	16x16	8	0.005	4.059	811.8	91.07	50.34/66.67
256x256	32x32	8	0.006	4.059	676.5	90.9	56.01/66.67
256x256	64x8	8	0.005	4.059	811.8	90.9	52.57/66.67
256x256	128x1	8	0.004	4.059	1014.75	90.9	46.27/75
256x256	256x1	8	0.004	4.059	1014.75	90.9	42.18/66.67
Resolution	Block size	No. of Samples	Parallel	Serial	SpeedUp	Branch Efficiency	Achieved/Theoretical Occupancy
512x512	8x8	8	0.014	17.201	1228.642857	91.33	55.36/66.67
512x512	16x16	8	0.013	17.201	1323.153846	91.28	57.61/66.67
512x512	32x32	8	0.013	17.201	1323.153846	91.1	59.11/66.67
512x512	64x8	8	0.013	17.201	1323.153846	91.1	56.86/66.67
512x512	128x1	8	0.013	17.201	1323.153846	91.1	60.16/75
512x512	256x1	8	0.012	17.201	1433.416667	90.9	42.18/66.67
Resolution	Block size	No. of Samples	Parallel	Serial	SpeedUp	Branch Efficiency	Achieved/Theoretical Occupancy
1024x1024	8x8	8	0.045	73.166	1625.911111	91.53	61.09/66.67
1024x1024`	16x16	8	0.045	73.166	1625.911111	91.49	60.83/66.67
1024x1024`	32x32	8	0.046	73.166	1590.565217	91.33	61.13/66.67
1024x1024`	64x8	8	0.045	73.166	1625.911111	91.33	58.78/66.67
1024x1024	128x1	8	0.046	73.166	1590.565217	91.33	65.34/75
1024x1024`	256x1	8	0.044	73.166	1662.863636	91.33	58.92/66.67

Fig: Scene2 Experimentation Results

We also observe, that branch efficiency is close to 91%, this is due to the fact that importance sampling for one ray might choose the reflected path while for the other ray choose the light path. Furthermore, Nvidia Profiling Tool helped us understand that Theoritical Occupancy was limited due the registers utilised per thread. But since our CUDA program was written efficiently we were able to have our achieved occupancy be close to the theoretical limit for most of the different parameter configuration.

Note: In the SpeedUp Comparison Chart, different colors represent different resolution sizes. While the Y-axis represents different block sizes.

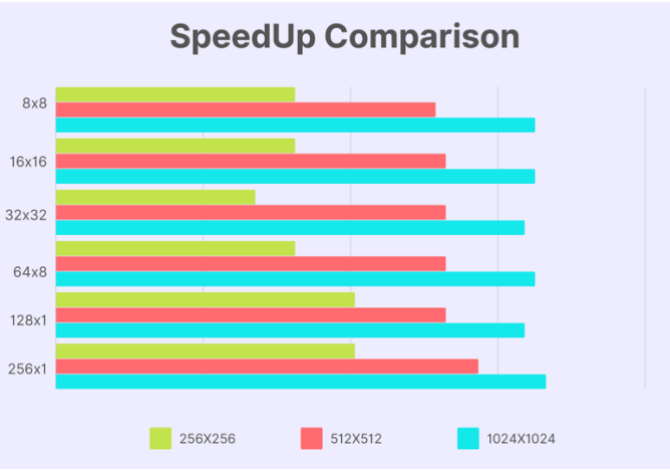

Fig: Scene2 SpeedUp Comparison

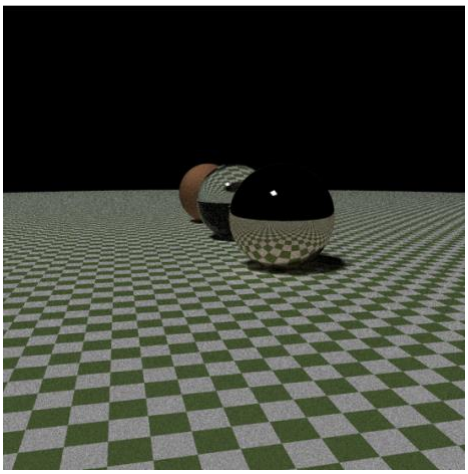

Fig: Scene2

Scene3:

In this scene, where there is an extreme workload imbalance, from the results we observe that SpeedUp scales with image Resolution. We achieve a maximum SpeedUp of 592x for image of Size 256x256, while we achieve a maximum SpeedUp of 1187x for image of Size 512x512 and for image size of 1024x1024 we achieve a maximum SpeedUp of 1511x. This behaviour is observed because increasing the image resolution increases the serial complexity proportionally, but due to the smart resource utilization in our CUDA program the increase in parallel complexity is significantly lower thus there is a scalabilty in SpeedUp as the resolution size increases. Also we, observe that '64x8' block size leads the best SpeedUp for 256x256 and 1024x1024 image resolutions, while '8x8' block size lead to the best SpeedUp for 512x512 image resolution. Thus, when rendering a scene in which pixels have an extreme imbalance in the amount of computations being done and the users hardware resembles closely to the one we use. We would recommend setting the block size to '64x8' or '8x8' for initial testing of ideal block size for maximum SpeedUp.

Resolution	Block size	No. of Samples	Parallel	Serial	SpeedUp	Branch Efficiency	Achieved/Theoretical Occupancy
256x256	8x8	8	0.206	104.805	508.7621359	99.82	37.44/66.67
256x256	16x16	8	0.201	104.805	521.4179104	99.81	39.04/66.67
256x256	32x32	8	0.199	104.805	526.6582915	99.81	43.36/66.67
256x256	64x8	8	0.177	104.805	592.1186441	99.81	42.78/66.67
256x256	128x1	8	0.187	104.805	560.4545455	99.81	39.79/75
256x256	256x1	8	0.19	104.805	551.6052632	99.81	36.46/66.67
Resolution	Block size	No. of Samples	Parallel	Serial	SpeedUp	Branch Efficiency	Achieved/Theoretical Occupancy
512x512	8x8	8	0.358	425.249	1187.846369	99.82	52.29/66.67
512x512	16x16	8	0.368	425.249	1155.567935	99.82	52.16/66.67
512x512	32x32	8	0.417	425.249	1019.781775	99.81	47.19/66.67
512x512	64x8	8	0.376	425.249	1130.981383	99.81	46.81/66.67
512x512	128x1	8	0.372	425.249	1143.142473	99.81	53.64/75
512x512	256x1	8	0.374	425.249	1137.029412	99.81	48.08/66.67
Resolution	Block size	No. of Samples	Parallel	Serial	SpeedUp	Branch Efficency	Achieved/Theoretical Occupancy
1024x1024	8x8	8	1.131	1680.53	1485.879752	99.82	58.36/66.67
1024x1024`	16x16	8	1.129	1680.53	1488.511957	99.82	57.94/66.67
1024x1024`	32x32	8	1.288	1680.53	1304.759317	99.82	51.42/66.67
1024x1024`	64x8	8	1.112	1680.53	1511.267986	99.82	51.93/66.67
1024x1024	128x1	8	1.145	1680.53	1467.71179	99.82	59.11/75
1024x1024`	256x1	8	1.113	1680.53	1509.910153	99.82	53.83/66.67

Fig: Scene2 Experimentation Results

We also observe, that branch efficiency is close to 99%, this is due to the fact that importance sampling for one ray might choose the reflected path while for the other ray choose the light path. Furthermore, Nvidia Profiling Tool helped us understand that Theoritical Occupancy was limited due the registers utilised per thread. But since our CUDA program was written efficiently we were able to have our achieved occupancy be close to the theoretical limit for most of the different parameter configuration.

Note: In the SpeedUp Comparison Chart, different colors represent different resolution sizes. While the Y-axis represents different block sizes.

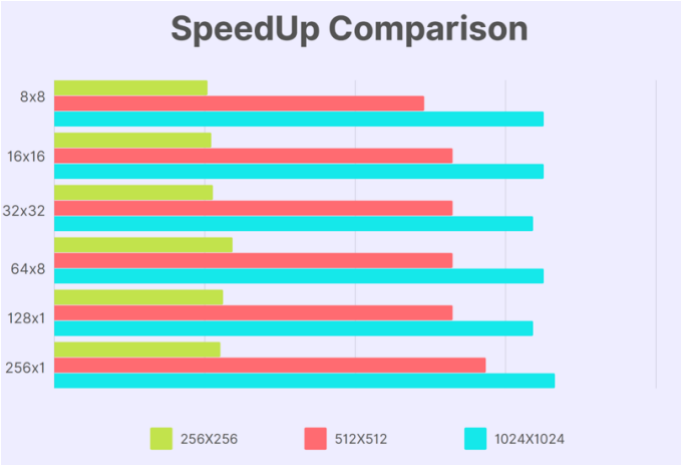

Fig: Scene3 SpeedUp Comparison

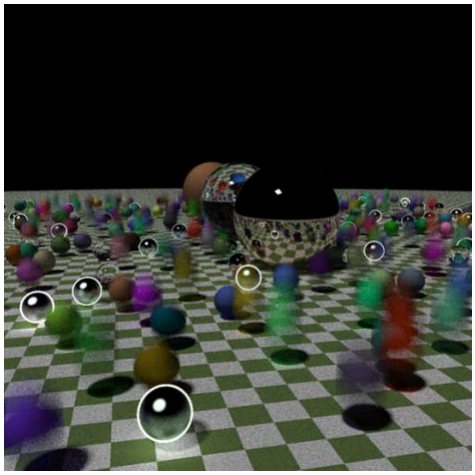
Fig: Scene3

Conclusion:

In this project, we develop a Monte Carlo Path Tracing Rendering System without using a graphics library with features such as materials like Lambertian/diffuse, metal, dielectric and camera effects like motion blur, defocus blur and Monte Carlo and Importance Sampling. We then optimize the execution time of our rendering system by developing a CUDA program to parallelize the rendering workload. We consider different 3D scenes that could be rendered and how they would impact performance. Thus, for different scenes, we come up with different parameter ranges that would yield high resource utilization of the underlying hardware (GPU) and offer maximum Speedup in terms of execution time.

References:

[1] “Path tracing.” [online]. Available: https://en.wikipedia.org/wiki/Path_tracing
[2] P. Shirley, “Ray tracing in one weekend.” [online]. Available: https://raytracing.github.io/books/RayTracingInOneWeekend.html
[3] P. Shirley, “Ray tracing the next week.” [online]. Available: https://raytracing.github.io/books/RayTracingTheNextWeek.html
[4] P. Shirley, “Ray tracing the rest of your life.” [online]. Available: https://raytracing.github.io/books/RayTracingTheRestOfYourLife.html
[5] J. T. Kajiya, “The rendering equation,” in Computer Graphics, vol. 20, no. 4, Aug. 1986, pp. 143-150.
[6] “Gpu accelerated computing with c and c++.” [online]. Available: https://developer.nvidia.com/how-to-cuda-c-cpp
[7] “Accelerated ray tracing in one weekend in cuda.” [online]. Available: https://devblogs.nvidia.com/accelerated-ray-tracing-cuda/
[8] H. W. Jensen et al., “State of the art in monte carlo ray tracing for realistic image synthesis,” in SIGGRAPH 2001 Course 29, Aug. 2001. [online]. Available: http://cseweb.ucsd.edu/~viscomp/classes/cse274/fa18/readings/course29 sig01.pdf