CUDA Ray Tracer: Implementation, Comparison, and Profiling

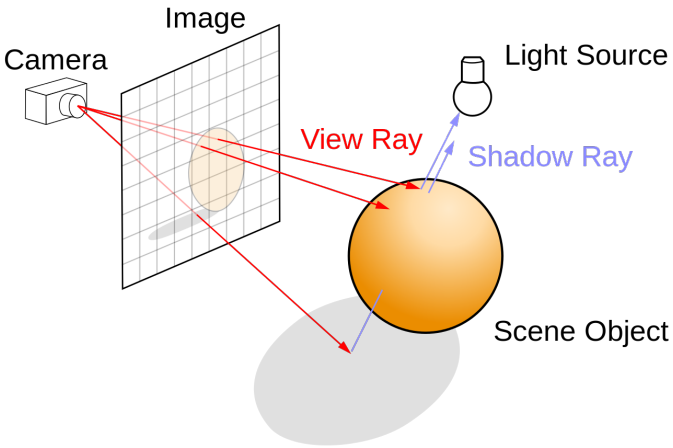
Manuel Pagliuca - 975169
December 13, 2022

Abstract

In this paper, I will illustrate how much is the Ray Tracing algorithm highly scalable on the GPUs, through a comparison with the **host only implementation** of the same code.

Everything started two years ago when I found out the field of *computer graphics* with the **ray tracing**, this was possible thanks to a really well-known online book called "*Ray Tracing in One Weekend*"[6] from Peter Shirley

The aim of this paper is to illustrate briefly the algorithm and to make a comparison between the CPU and the GPU implementation of the same ray tracer with the same scene.



1 Introduction

Currently, the algorithm is very well known due to the newest hybrid implementations on the *graphic pipelines* that wants to bring its peculiarities to the video games industry.[2]

Anyway this feature it's really difficult to achieve in real-time graphics since it is a really expensive algorithm, which allows us to obtain physically accurate images (in this particular case we speak of **path-tracing**).

The algorithm was not recently conceived, the first idea dates back to 1969 and finds its major use in **offline rendering** of animated films.

The algorithm is intensive since it uses an **image-oriented** rendering approach instead of using a graphic pipeline, for this reason, the use of the **GP-GPU** paradigm is a really good solution.

2 The algorithm

Ray Tracing is a 3D rendering algorithm that is based on the computation of the paths followed by the light, in such a way as to follow the rays through the interaction with the surfaces. This technique is capable of simulating a variety of optical effects such as reflection, refraction, diffraction, and dispersion with a high degree of realism.

The concept of the algorithm itself is even older since it derives in turn from the concept of the darkroom.

The algorithm works by tracing *rays* (or paths in the case of **path-tracing**) that start from an imaginary point of view called "***eye***" and cross one virtual screen (also called **frame**, **image-plane** or sometimes **sensor**, referring to the camera sensor).
Once you have crossed the image plane you reach the scene containing 3D objects that will be described by mathematical models by the programmer (or visual artist through intermediate tools). For each ray an intersection test is performed for each subset of objects in the scene, in the event that the test concludes with a positive result, the calculation of the light entering the intersection point is carried out, based on the type of material of the object that is intersecting (*lambertian*, *metallic*, *glass*, ...), in the end, all this information is combined and the color for the single pixel is obtained.
Already only in the case of a simple Ray Tracer, the calculations that occur for just a single pixel are too many, the speech can be considerably expanded when it comes to Path Tracing and therefore to Global Illumination, where the amount of calculations for each pixel increases considerably.
At the following link, there is a more explanatory image of the algorithm operation: `https://upload.wikimedia.org/wikipedia/commons/9/95/Ray_Tracing_Illustration_First_Bounce.png`.

3 CUDA Implementation

The CUDA[5] implementation is a transcription of the Peter Shirley code explained in his book for the GPUs, another resource that I also used is from Roger Allen which made a post[1] on the NVIDIA Developer Blog about the implementation of some chapters from "*Ray tracing in one weekend*".
By the way, the abstract logic of the code stays the same, anyway this wasn't the first attempt to port my previ-

ous Ray Tracer on GPU, I tried with **OpenCL**[3] from Khronos. The problem with OpenCL is that doesn't offer all the new and recent features that CUDA has, they are really helpful and great for this project.
Specifically, this project used:

- **Unified Memory**, for accessing the frame buffer directly from the CPU.
- **cuRAND**, for generating high-quality pseudo-random numbers (used for achieving the MSAA).
- **Dynamic Parallelism**, for calling different kernels during the rendering.

For obvious reasons, the grid used for implementing the Ray Tracer is a bi-dimensional grid of bidimensional blocks. The grid must fit the width and height of the **framebuffer** (the resulting render), each thread block is an 8x8 matrix of threads.
The general execution of the algorithm :

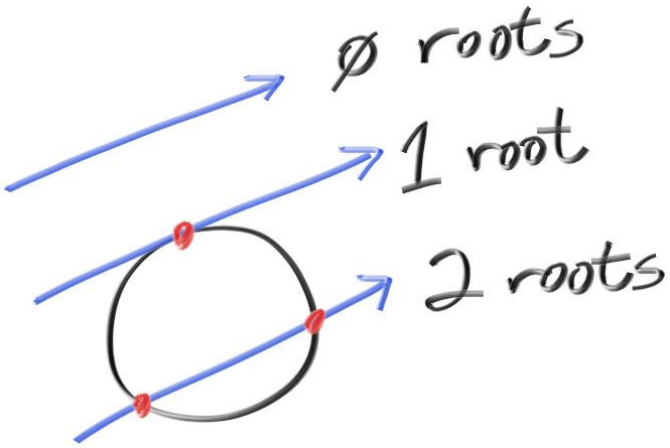
1. Setting up the size and number of samples to use by user input.
2. Allocate the needed objects: *framebuffer, random states for all pixels, hittable list, world, camera.*
3. Initialise the rendering (setting different seeds for each pixel) with a kernel.
4. Launching the rendering kernel (and the timer).
5. Collecting the results from the frame buffer and saving them in PNG file format.
6. Free the used resources (and device reset).

Generally, the images are saved in PPM file format, but I personally don't like it and prefer using a more common file format which is not lossy (so not JPEG) the PNG. This was trivial to obtain thanks to the *stb image* library[4].
The *render* kernel, which will perform these operations for all the pixels (this will also call other kernels thanks to dynamic parallelism) :

1. Sending rays to a pixel (expressed in row-major order) of the **view**, starting from the TLC (*top-left corner*). This operation occurs *ns* times, where *ns* is the number of the random samples that will be used for performing the **MSAA**.
   (a) Calculating the *u,v* random offsets (they are between [0,1]) through the **cuRAND** Library (this operation on the CPU sides needed an external library to obtain a high quality of uniform pseudo-random numbers).
   (b) Use the offsets for generating rays that will sample (calling the *color* kernel) the neighbors of the pixels.
   (c) Sum all the colors of the pixels in one vector, this will call the *color* kernel.
   - Essentially the *color* kernel will scroll the list of all the objects present in the world and check if the object is hit by a ray (performs the ray-sphere intersection). In that case will perform the scattering of the light in the base of his material, if there is no intersection this means that the ray is missed, so it will show a linear interpolation between white and light blue (to make it look like the sky).
     This kernel in particular has a limitation imposed by a loop of 50 because the attenuation was obtained recursively on the CPU version. So the solution proposed by Roger Allen was to make iterative to avoid reaching the limitations of the hardware imposed on the recursive calls.
2. Performs the average of *ns* samples on the pixel color vector.
3. Save the colored pixel on the framebuffer unified memory.

3.1 Spheres

The only objects which were implemented are the spheres because the ray-sphere intersection is simpler to obtain (and also *reflection and refraction* leads to better graphic effects on curved surfaces) with vector math and to reduce it to a second-grade equation (we will keep the first result since it is what the **eye** will see).



4 Comparisons and Metrics

4.1 Comparison with CPU implementation

It may seem obvious who will be the winner, but I think it is appropriate to make a comparison in order to better have the distance between the two implementations.
We are rendering the same scene which consists of 8 spheres with different materials (*metal, glass, lambertian*)

whose properties will be made them perform different light effects.
The GPU used is an *NVDIA STRIX 1070* and the CPU is an *Intel Quad-Core i7-6700K @ 4.00GHz*

Table 1: CPU vs GPU benchmarks with 10 samples.

Resolution	CPU time (sec)	GPU time (sec)
800x600	58.9532	0.0537
1280x720	110.9580	0.0825
1920x1080	248.112	0.174891

The comparison is pretty over there, of course, the GPU would have won but the results are really shocking.
The GPU implementation in respect of the CPU implementation is 1097 times faster in the first benchmark and 1418 times with the 1920x1080 resolution.
Let's see GPU-only benchmarks with different samples :

Table 2: GPU benchmarks with different samples.

Resolution	Samples	GPU time (sec)
800x600	10	0.0537
800x600	100	0.5110
800x600	500	2.4204
1280x720	10	0.0825
1280x720	100	0.7481
1280x720	500	3.6542
1920x1080	10	0.1749
1920x1080	100	1.6084
1920x1080	500	7.9574

4.2 Profiling

Using ***nvprof*** is possible to measure some technical metrics of the CUDA execution.
The metrics used :

- Occupancy (ratio of the average active warps per active cycle to the maximum number of warps supported on a multiprocessor)
- Number of overall instructions executed.
- Number of floating point operations.
- Number of double-digit operations.
- Number of integer operations.
- Device Memory Read Throughput.
- Device Memory Write Throughput.
- Instructions executed per cycle.
- Warp execution efficiency.

For the execution of **nvprof** the render proposed got an 800x600 resolution with 500 samples. The profiling tools metrics are related only for the *render* kernel, since the others are marginal in comparison.

Metric Description	Min	Max	Avg
Achieved Occupancy	0.444211	0.444211	0.444211
Instructions Executed	4.2153e+10	4.2153e+10	4.2153e+10
FP Instructions(Single)	1.4403e+11	1.4403e+11	1.4403e+11
FP Instructions(Double)	0	0	0
Integer Instructions	1.5631e+11	1.5631e+11	1.5631e+11
Device Memory Read Throughput	48.698GB/s	48.698GB/s	48.698GB/s
Device Memory Write Throughput	97.027GB/s	97.027GB/s	97.027GB/s
Executed IPC	0.657102	0.657102	0.657102
Warp Execution Efficiency	49.64%	49.64%	49.64%

4.3 Conclusions

In conclusion, even if the profiling results did not give the best results for **warp efficiency**, we obtained amazing results in terms of kernel execution speed.
The CUDA API together with the libraries proved to be extremely comfortable and easy to use in the implementation of this project, avoiding having to resort to external convenient libraries (in the CPU version the PCG library was used for the generation of random numbers).
So it is fair to say that the use of the GP-GPU paradigm is perfectly suited to the problem of Ray racing thanks to the possibility of generating ad hoc two-dimensional grids for processing images in parallel.

References

[1] Roger Allen. **Accellerated Ray Tracing in One Weekend in CUDA**, `https://developer.nvidia.com/blog/accelerated-ray-tracing-cuda/`, Novemer 5th, 2018.

[2] Microsoft Corporation. **DirectX Raytracing**, `https://en.wikipedia.org/wiki/DirectX_Raytracing`, October 10th, 2018.

[3] Khronos Group. **OpenCL**, open computing language, December 8th, 2008.

[4] nothings. **stb image library**, `https://github.com/nothings/stb`.

[5] NVDIA. **CUDA, Compute Unified Device Architecture**, `https://developer.nvidia.com/cuda-zone`, June 23th, 2007.

[6] Peter Shirley. ***Ray Tracing in One Weekend***, *https://raytracing.github.io/books/RayTracingInOneWeekend.html*. 2018.