Software Engineering Work Term Report

GPU-Parallelized Ray Tracing

Thomas George

Relevant Courses: CS 247, CS 488

Table of Contents

Terminology

Ray Tracer – a program which simulates light reflections, refractions, absorption (and other interactions) with a set of mathematically defined 3D objects. The Ray Tracer outputs an image of these objects which appears 3D due to the realistically generated shadows and lighting effects.

GPU (Graphics Processing Unit) – a processor specialized for highly-concurrent, math intensive operations. GPUs are vital in computer graphics, where functions commonly operate on a large volume of pixels simultaneously.

CUDA (Compute Unified Device Architecture) – a parallel computing platform and API for C, C++, Python and Fortran. CUDA is proprietary to Nvidia Corp., and it directly leverages their GPU hardware.

Kernel – a function compiled to execute on the GPU. In CUDA, kernels declarations/definitions are prefixed with either *__device__* or *__global__*.

STL (Standard Template Library) – a C++ library developed at Bell Labs providing templated algorithms, containers, iterators and functions.

1. Introduction

During the past co-op term, I worked for Nvidia Corp., where I developed video encoding capabilities in the Direct3D Windows Driver. Due to the strict confidentiality of the implementation details and driver code base, I am prevented from divulging any of my meaningful design decisions and rationales. Therefore, the topic of this report is aimed toward on an open-source project which I independently developed during the summer term: The A-Tracer.

The A-Tracer, or Accelerated-Tracer, is a ray tracer parallelized for GPU execution written in C++ (source code is on GitHub). I built this project because I am eager to take CS 488 (Introduction to Computer Graphics) in fourth year.

In this report, I will focus on one specific aspect of the A-Tracer which was very involved from an engineering standpoint: parallelizing the slow Monte Carlo sampling technique on a GPU.

2. Initial Implementation

In this section, I will provide a background on the state of the A-Tracer codebase prior to extending support to the GPU. This will set the stage for Section 4, where I will detail the design decisions and modifications to the existing program structure to incorporate GPU-Accelerated Monte Carlo sampling.

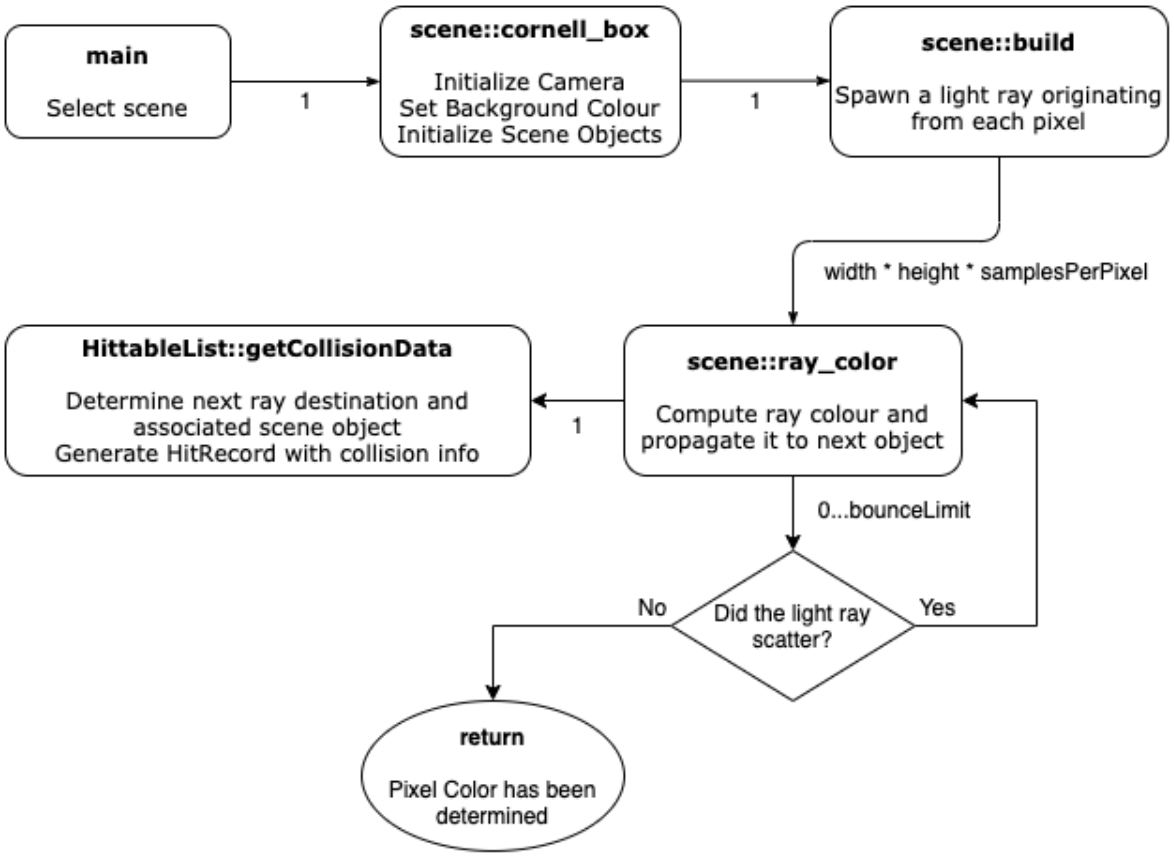

Figure 1 – High-level control flow governing the A-Tracer. The numbers beneath the transition arrows indicate the number of times this transition occurs.

Consider the A-Tracer's control flow in Figure 1. The program begins in the main function, where the scene to be drawn is selected. Then, a scene-specific function (e.g. *scene::cornell_box*) is called, which initializes the camera object (representing the plane of the image and the field of view), background colour and a list of 3D objects that make up the scene.

Next, *scene::build* generates a colour for each pixel in the image by generating a light ray that initially passes through every pixel position. *scene::ray_color* receives each light ray as a parameter and invokes *HittableList::getCollisionData*, which returns the ray after it has interacted with the scene objects (along with the collision metadata). If the ray had not been fully absorbed by an object or sent into the surrounding void, this procedure is recursively repeated on the scattered ray. There is a light ray bounce limit enforced to prevent stack overflow caused by too many successive reflections.

```
void scene::build(unsigned imgWidth, unsigned imgHeight, unsigned
maxReflections, unsigned samplesPerPixel, double aspectR)
{
    ...
    for (int i = static_cast<int>(imgHeight) - 1; i >= 0; --i)
    {
        for (unsigned j = 0; j < imgWidth; ++j)
        {
            pixelColor.zero();
            for (unsigned sample = 0; sample < samplesPerPixel; ++sample)
            {
                fractionX = (j + utils::random_double()) / width;
                fractionY = (i + utils::random_double()) / height;

                outgoingRay = camera->updateLineOfSight(fractionX,
fractionY);

                pixelColor = scene::ray_color(outgoingRay, background,
world, pdf, maxReflections);
            }
            pixelColor.formatColor(std::cout, samplesPerPixel);
        }
    }
    ...
}
```

Listing 1 – Source code for constructing each pixel's RGB channels

Examining the source code for *scene::build* in Listing 1 provides context on how *scene::ray_color* is invoked. For every pixel position in the image, a ray is generated by *Camera::updateLineOfSight* whose origin is at a fixed point (the "eyes" of the viewer) and which passes through the current pixel of interest (indicated by its fractional coordinates). Then, the ray, background, object list and bounce limit are passed to the recursive scene::ray_color. After re-iterating the ray trace for *samplesPerPixel* times, *Vec3::formatColor* averages the colour and appends its RGB components to the *.ppm output file.

3. Problem Context

In this section, I will outline how the A-Tracer's problem of generating low-quality images evolved into a runtime inefficiency. I will additionally describe the design constraints I needed to consider when adapting the Ray Tracer to run on a GPU, while retaining support for exclusive CPU execution.

3.1. Motivation

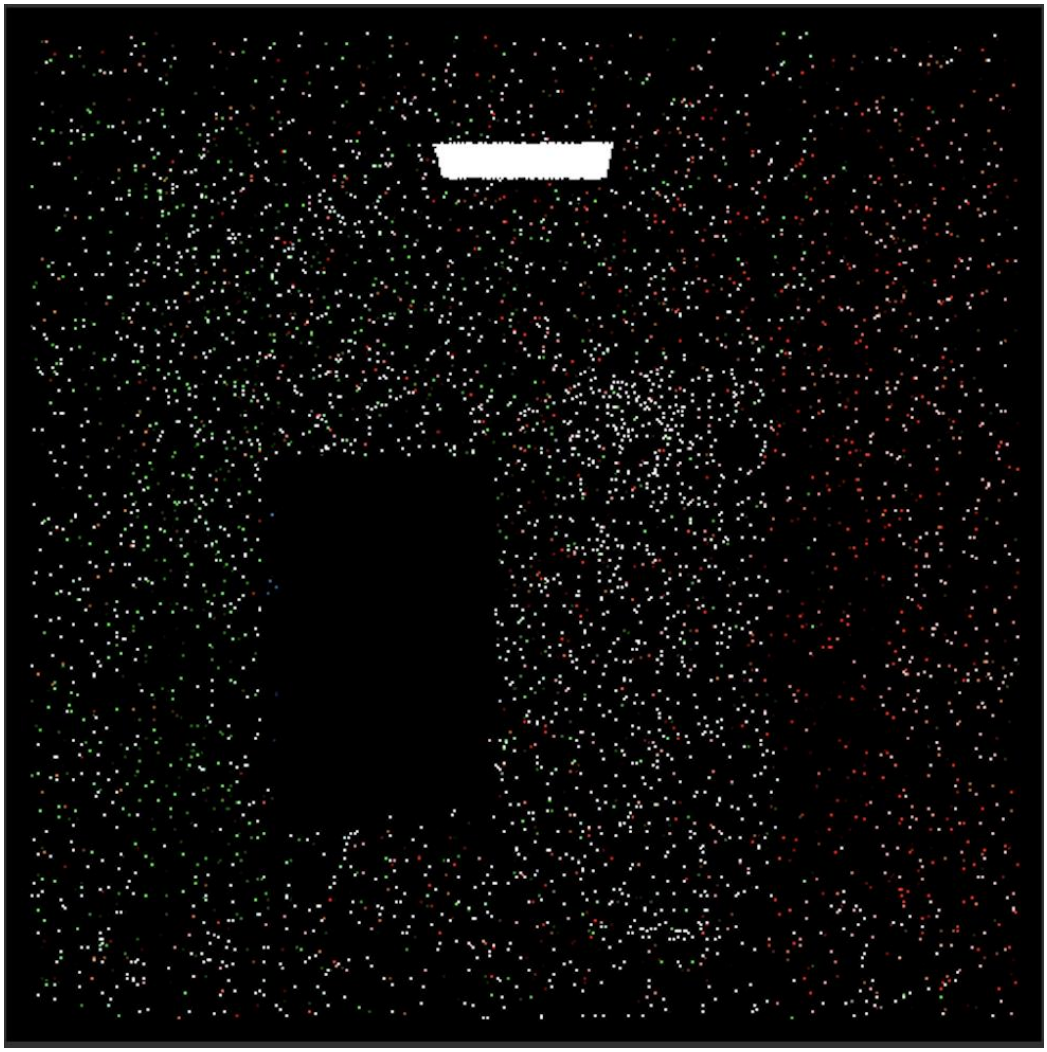

Figure 2 – 500x500 pixel image, each pixel sampled 1 time

"A picture is worth a thousand words" is an ancient phrase that stands true with the unappealing Figure 2. Figure 2 attempts to depict a modified Cornell Box (a scene used to demonstrate some of the different shapes, materials and textures supported by the A-Tracer), but it is speckled with frequent black pixels, and the two prisms and sphere are imperceptible. This unacceptable level of image noise can be attributed to the fact that a single round of calculation was done for each pixel colour. Since there is randomness involved in the light ray's interactions with the virtual environment, taking a one-color sample for each pixel allows for maximal statistical fluctuation and results in an uneven image.

To improve the quality, I employed a Monte Carlo strategy which is more likely to reflect light rays towards the light source of the Cornell Box [1]. By averaging multiple colour samples for each pixel and ensuring that most of these samples introduce luminance into the scene, I achieved Figure 3.

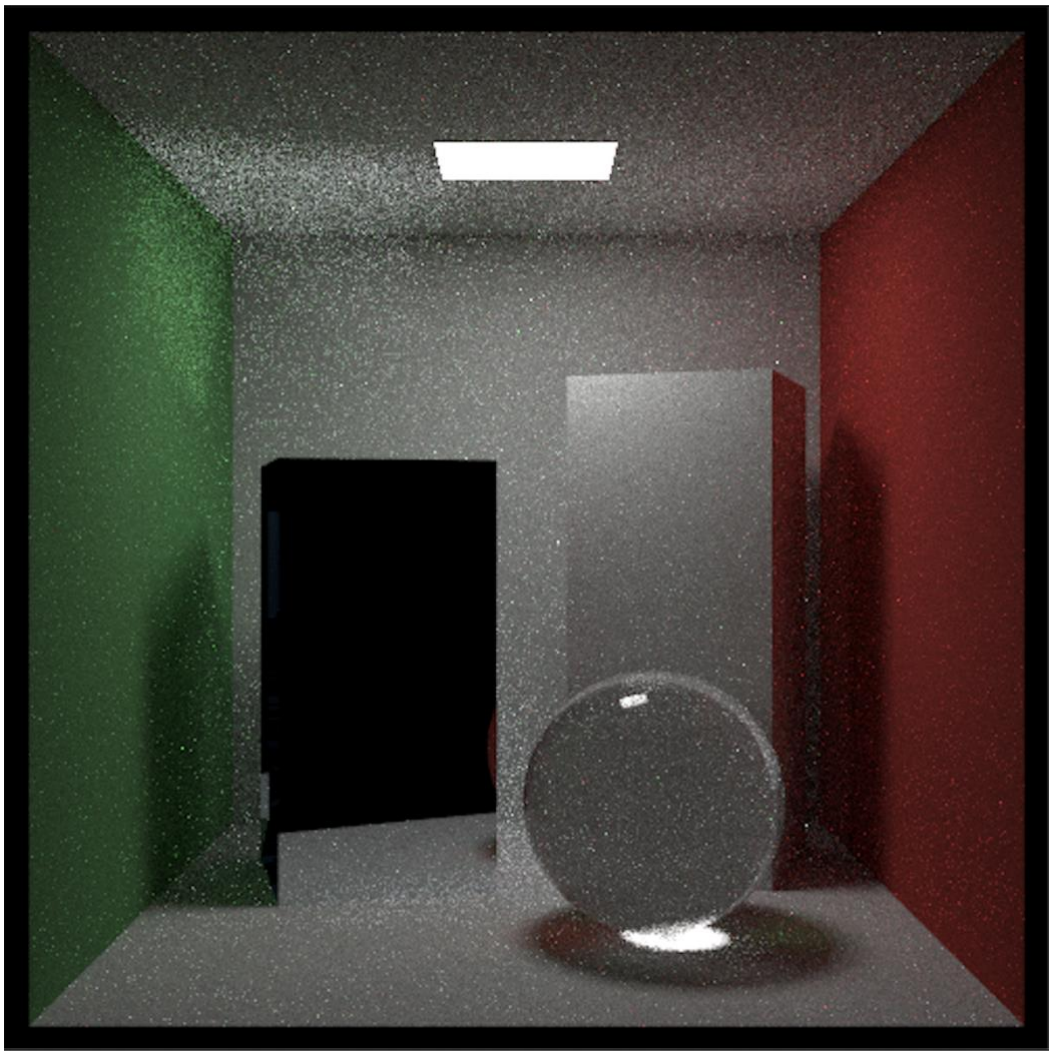

Figure 3 – 500x500 pixel image, each pixel sampled 100 times. The metallic prism on the right, diffuse prism on the right and glass ball in front are significantly more visible.

While multisampling each pixel resolved one problem, it introduced another: the time to generate the image had been roughly scaled up by the number of samples per pixel. The A-Tracer takes each ray individually and serially simulates its trajectory through the scene multiple times, recording the attenuation, surface normal and emitted ray. This is highly inefficient since every pixel and every sample is independent. The scene objects are not modified by the rays, so no race conditions arise if pixel Y can be computed before pixel X, even if pixel X appears first in the pixel array.

3.2. Constraints

3.2.1. Design

One of my development goals is to allow the user to compile the A-Tracer for both GPU and CPU execution. For the GPU case, all function declarations and definitions need to be prefixed with:

- *__host__* - the function is being compiled for execution on the CPU, callable from the CPU
- *__device__* - the function is being compiled for execution on the GPU, callable from the GPU
- *__global__* - the function is being compiled for execution on the GPU, callable from the CPU

Note that any method that overrides a virtual method must have matching __host__/__device__/__global__ tags.

3.2.2. Runtime Environment

In addition to the compilation considerations detailed in Section 3.2.1, there is one major additional restriction which will cause a runtime error if not addressed: device memory allocation. Namely, the CPU host and GPU device have disjointed heaps and accessing a particular address from the wrong processor will crash the A-Tracer.

This constraint invalidates the use of STL smart pointers (which would allocate memory only from the CPU heap).

3.2.3. Standard Template Library

This constraint is directly implied from the compilation constraints detailed in Section 3.2.1. The Standard Template Library has been instrumental to the A-Tracer's initial development. Since the STL's functions were not designed for CUDA use, they are not prefixed with the *__device__* keyword, so they cannot be invoked from a CUDA kernel running on the GPU.

While std::array<T> and std::vector<T> were used sparsely throughout the code base (and could be easily replaced with custom-built containers), every single pointer in the A-Tracer was a smart pointer (std::shared_ptr<> or std::unique_ptr<>). This signifies the need for a complete overhaul of the garbage collection system.

3.2.4. Performance

Since this project was done on my own initiative, there were no external performance requirements or hardware restrictions. My goal was to simply improve upon the time the Ray Tracer took to generate an image when Monte-Carlo importance sampling was used. My baseline was a 500x500 pixel Cornell Box image from Section 3.1 with 100 samples per pixel. It took roughly 7.4 minutes to generate.

4. The Solution

In this section, I will detail how I remodeled the A-Tracer from its CPU implementation in Section 2 to accommodate for GPU execution while adhering to the constraints of Section 3.2.

4.1. Design

Prior to following the CUDA API, I considered its alternative: Open Computing Language (OpenCL). While OpenCL is open-sourced and cross-platform (meaning that I could use the built-in AMD GPU in my laptop), CUDA was still the obvious choice. In addition to CUDA being more performant, it supports object-oriented and templated programming paradigms. OpenCL extends C99 [2] and does not support these C++ features, implying a massive overhaul of the A-Tracer's existing codebase. Moreover, the A-Tracer consists of real-life objects and systems that interact with each other, which will not be as elegantly encompassed in the procedural C language.

In CUDA, the *__host__*, *__device__* and *__global__* keywords are necessary, but a common C++ compiler such as g++ or clang++ will not recognize them. Therefore, I needed to use nvcc (Nvidia Cuda Compiler), while ensuring that any file containing a GPU kernel has the .cu extension.

```
ifeq ($(target), gpu)
        CPPC := nvcc
        CPPFLAGS := -g $(INCLUDES) -D GPU=1
        SOURCES := $(shell find src -name "*.cu")
else
        CPPC := clang++
        CPPFLAGS := -g -std=c++17 -g -Wall -Werror
$(INCLUDES) -D GPU=0
        SOURCES := $(shell find src -name "*.cpp")
endif
```

Listing 2 – Differences in compilation for the CPU vs. the GPU, as seen in the A-Tracer's Makefile

```
#ifdef GPU
#define DEV __device__
#define HOST __host__
#define GLBL __global__

#else
#define DEV
#define HOST
#define GLBL
#endif
```

Listing 3 – Platform.cuh keyword definitions

To incorporate these keywords, I added the GPU macro definition in the Makefile of Listing 2. GPU gets set to 1 if the user specifies target=gpu in their make command and GPU = 0 otherwise. Then in header file Platform.cuh of Listing 3, the *DEV, HOST* and *GLBL* macros are defined. Platform.cuh is included in every parent class of the A-Tracer, and all of the class methods are prefixed with *DEV, HOST* and *GLBL* as appropriate. When target=gpu is not specified, these macros are just empty spaces and the function declarations/definitions as seen by the compiler are standard C++.

4.2. Runtime Environment

Memory managed by *new/delete* will cause a runtime crash if accessed from the GPU. CUDA provides two alternatives to memory allocation: segregated memory and unified memory.

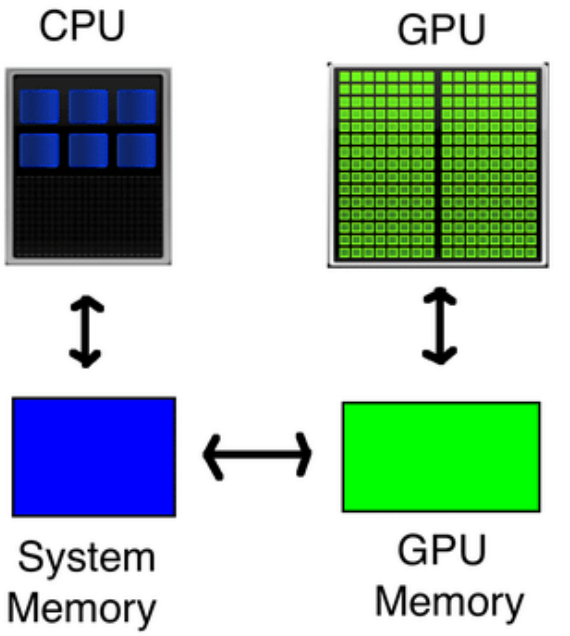

Figure 4 – Segregated CPU-GPU Memory Model [3]

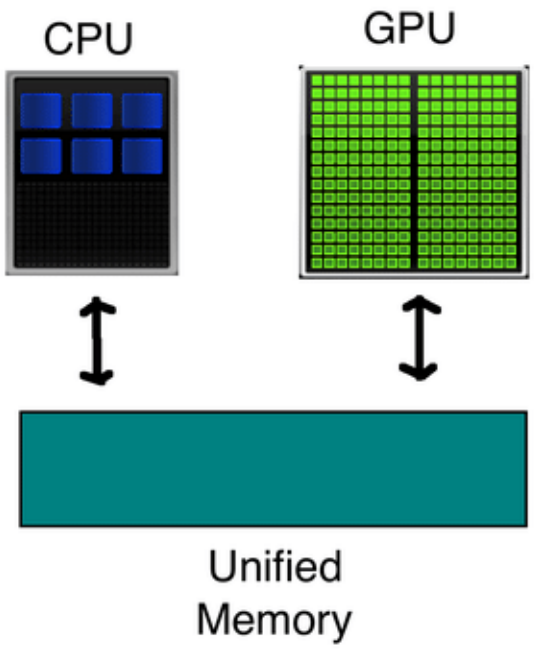

Figure 5 – Unified CPU-GPU Memory Model [3]

With segregated memory (illustrated in Figure 4), *cudaMalloc* allocates memory for exclusive GPU access. *cudaMalloc* is only callable from the CPU, so the data instantiated on the CPU heap must be copied into the GPU heap using *cudaMemcpy*. In addition to increasing the line count, segregated memory requires that you manage two pointers to your memory (one for CPU access, one for GPU access) and that you consolidate it if its contents are modified.

The alternative to managing segregated memory is using the unified memory model from Figure 5. In this model, the GPU can access any page of the entire system memory, and it copies frequently accessed address ranges into its local GPU heap for better performance [4]. Instead of *cudaMalloc*, *cudaMallocManaged* is used, and the resultant pointer to memory is accessible by both CPU functions and GPU kernels.

If both memory models were employed optimally in the A-Tracer, segregated memory would perform slightly better. However, the major performance bottleneck from Figure 1 (calling *scene::ray_color* serially for every independent sample and pixel) is already being solved by the GPU, regardless of the memory model used. In this case, over-optimization of performance is undesired since it overcomplicates the code. Therefore, the unified memory model was more appropriate for the A-Tracer.

4.3. Standard Template Library

Since no STL methods are callable from the GPU, there are two alternate memory management patterns. One option is to override the *new* and *delete* operators to use *cudaMallocManaged* and *cudaFree* if the A-Tracer has been compiled with the target=gpu flag. This option is not favorable, since every class would need to override these operators for the GPU=1 case and provide a non-default destructor to manually free all heap resources.

The other option is to implement custom smart pointers which encapsulate the garbage collection and abstract out the need to swap allocation/deallocation APIs between CPU or GPU accesses. By consolidating all of this logic into a small subset of the A-Tracer's classes, the codebase becomes more maintainable and less prone to memory leak bugs.

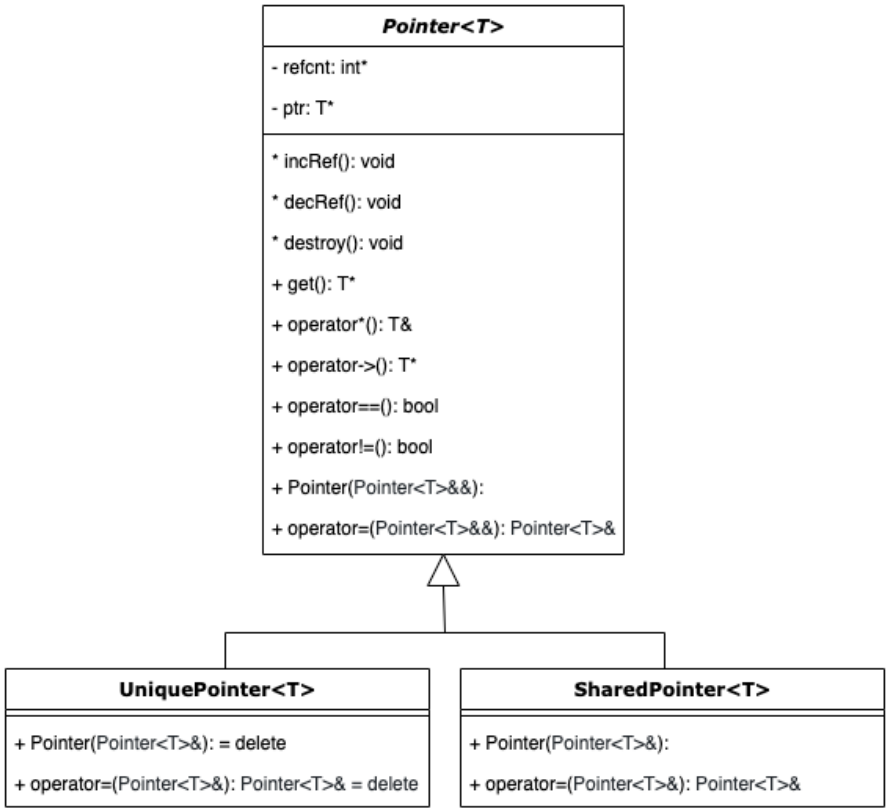

Figure 6 – UML depiction of the A-Tracer's custom smart pointer implementations

Figure 6 depicts the templated smart pointer inheritance hierarchy. The abstract *Pointer* class implements a reference count for the underlying resource, *ptr*. The *Pointer<T>* constructor transfers the object's contents from stack to heap (using *memmove* and *new* if GPU == 0 or *cudaMallocManaged* if GPU == 1). The *Pointer<T>::destroy* method uses *delete* if GPU == 0 and *cudaFree* if GPU == 1.

The *UniquePointer*'s copy constructors and assignment operators are explicitly deleted, limiting its **refcnt* to at most 1, and *SharedPointer* possesses a custom copy constructor and assignment operator that manages the **refcnt*. Since *refcnt* is a pointer, updating it in one *SharedPointer* updates the **refcnt* of all *SharedPointer* objects encapsulating that resource. If **refcnt* reaches 0, *ptr* is no longer in use and that memory gets automatically freed without manual intervention from any other class.

4.4. Performance

Ultimately, following the major CUDA constraints highlighted in Section 3.2.1 – Section 3.2.3 enabled the following modification in *scene::ray_color*'s invocation, depicted in Listing 4.

```
HOST void scene::build(unsigned imgWidth, unsigned imgHeight, unsigned
maxReflections, unsigned samplesPerPixel, double aspectR)
{
    ...
    dim3 blockD = {.x = SAMPLES_PER_PIXEL, .y = 1, .z = 1};
    dim3 gridD = {.x = imgWidth, .y = imgHeight, .z = 1};

    scene::sample_pixel <<<gridD, blockD>>> (d_img, imgWidth, imgHeight,
maxReflections, camera, pdf, background, world);
    ...
}
```

Listing 4 – *scene::sample_pixel* invoked using the CUDA API.

The modified *scene::sample_pixel __global__* kernel allows for all samples of each pixel to be computed independently, as decided by the GPU's hardware thread scheduler. CUDA's <<< … >>> execution configuration parameters provide a hardware thread abstraction by launching the kernel in a 3D grid of 3D blocks of independent GPU threads. In Listing 4, the block grid is imgWidth by imgHeight and each block contains a thread for each pixel sample. Therefore in *scene::sample_pixel*, the pixel coordinates are exactly the block coordinates. The current thread index specifies an offset into an array shared amongst all threads within a block. Once all samples have been written into this array, a GPU gather operation is performed to compute the sample average.

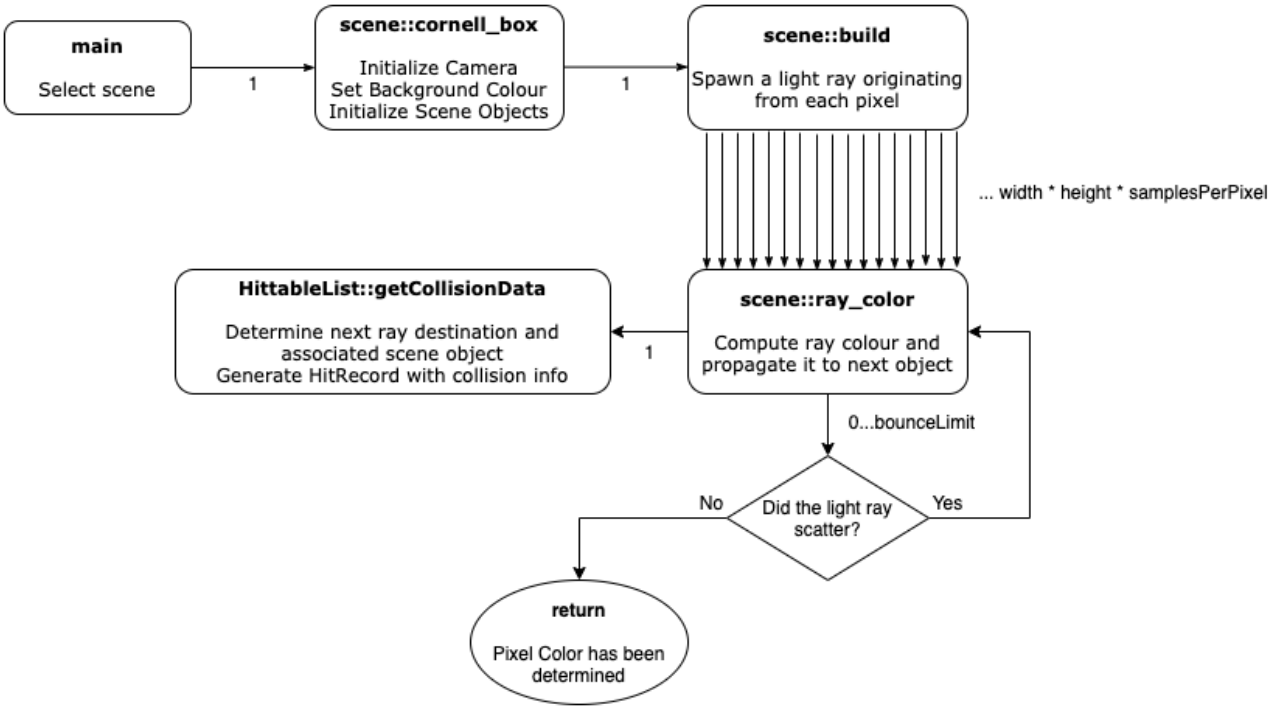

Figure 7 – High-level control flow governing the A-Tracer, now parallelized for the GPU.

Observe that in Figure 7, *scene::ray_color* is called as many times as it was in the serial implementation (Figure 1). But now, the calls are occurring concurrently, greatly elevating the throughput. The ray bounce limit loop in *scene::ray_color* still exists, since each scattered ray's origin depends on the origin and direction of the previous ray.

The time to beat when generating the Cornell Box benchmark image was 7.4 minutes as stated in Section 3.2.4. The time was indeed drastically reduced to 2.1 minutes for that same image, running on a low-end Nvidia Tesla T4 GPU.

All four constraints that defined porting the A-Tracer to the GPU as a success were fulfilled.

5. Conclusion

Overall, the A-Tracer is far from perfect and requires many more samples per pixel to output a truly crisp image. Nonetheless, it has made great progress in its output quality, with a side effect of heightened computational complexity. To reduce the running time to a more reasonable level, the ray tracing algorithm was adapted to run on a highly concurrent Nvidia GPU. This entailed a significant refactor in order for the compilation and execution to succeed, all while preserving the existing code design. My key takeaway from this project is that I should have planned for GPU support during initial development, as it would have saved a lot of time. I look forward to applying all that I have learned in my upcoming Capstone project.

References

[1] Ray Tracing: The Rest of Your Life, E-book, URL: raytracing.github.io/books/RayTracingTheRestOfYourLife.html

[2] OpenCL vs. CUDA: Which Has Better Application Support?, Article, URL: create.pro/opencl-vs-cuda/

[3] 'Unified Memory' Concept of CUDA 6.0 Programming Language, Graphic, URL: www.researchgate.net/figure/View-of-Unified-Memory-Concept-of-CUDA-60-Programming-Language_fig5_265851311

[4] Unified Memory Programming, CUDA Documentation, URL: docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#um-unified-memory-programming-hd