Real-time CUDA Raytracer

Alexandre Gbaguidi Aïsse, Thibaud Michaud

Abstract—Raytracing is a highly data-parallel task. This makes it very fit to GPU rendering since each pixel can easily be computed on a separate GPU thread. The purpose of this report is to present our implementation of a C++11 and a CUDA version of the same raytracer, compare their performance and suggest optimizations to improve the CUDA version.

Index Terms—Raytracing, CUDA, real-time, rendering, GPU

1 Introduction

In the context of computer graphics, *rendering* [1] is the automatic process of generating an image from a 2D or a 3D model. Two major rendering categories exist: algorithms based on rasterization and those based on raytracing. Historically, raytracing has been used exclusively to produce high-quality images. This approach is well known for its poor performances and long rendering time. If more precision is needed then more time-consumption will occur because more rays will be casted.
Since modern graphical processing units (GPUs) are designed for intensive data-parallel computing applications, they receive special attention in the raytracing field [2].
In this report, we present two versions of a real-time raytracer: a C++11 one and a CUDA one. We compare their performance and show that the GPU version outperforms the C++ version by a large factor, but we propose solutions to optimize the GPU version even further.

2 Specifications

- *Alexandre Gbaguidi Aïsse*
  E-mail: aga@lrde.epita.fr,
- *Thibaud Michaud,*
  E-mail: tmichaud@lrde.epita.fr
  LRDE (Research and Development Laboratory of EPITA).

As part of the Introducing course to CUDA provided by Alexandre Fender.

This section gives an overview of the project and explains different ways to use it.

2.1 Description

The main purpose of this project is to implement a real-time raytracer. Usually in raytracing, photo-realistic images are incompatible with real-time rendering. We will have to make hard choices that can affect both rendering time and image quality. From a given input which describes a scene, we give two possibilities:

- render the scene by producing an image and end the process.
- enter in an interactive mode with a camera (point-of-view) that can be moved around the scene.

2.2 Input

The input file looks like this:

screen width height
camera pos_x pos_y pos_z u_x u_y u_z v_x v_y v_z
sphere radius pos_x pos_y pos_z diff refl spec shin $color_r$ $color_g$ $color_b$ refr opac
plane a b c d diff refl spec shin $color_r$ $color_g$ $color_b$ refr opac
triangle A_x A_y A_z B_x B_y B_z C_x C_y C_z diff refl spec shin $color_r$ $color_g$ $color_b$ refr opac
plight pos_x pos_y pos_z $color_r$ $color_g$ $color_b$
dlight dir_x dir_y dir_z $color_r$ $color_g$ $color_b$
alight $color_r$ $color_g$ $color_b$

where:

- camera has a position pos_x pos_y pos_z and looks at the plane formed by the vectors (u_x, u_y, u_z) and (v_x, v_y, v_z).

- plane, triangle, sphere, etc. are the shapes that will be rendered
- (a-d-p)light is for ambiant, directional or point light
- diff is the diffusion coefficient, between 0 and 1.
- refl is the reflection coefficient, between 0 and 1.
- spec is the specular coefficient, between 0 and 1.
- shin is the shininess, between 0 and ∞. This property is used when implementing the specular light.
- color$_r$, color$_g$, color$_b$ contain three integer values, respectively the red, green and blue components, between 0 and 255.
- refr is the refraction coefficient, between 0 and ∞.
- opac is the opacity coefficient, between 0 and 1.

For example, the input file of the Figure 1 is the following:

```
screen 600 600
camera 0 0 0 1 0 0 0 1 0
sphere 2 -3 0 20 1 0 0 100 0 255 255 0 0
sphere 3 3 0 20 1 0 0 100 255 0 255 0 0
plane 0 1 0 -4 1 0 0 100 150 0 0 0 0
alight 255 255 255
dlight 1 1 0 255 255 255
dlight -1 1 0 255 255 255
```

2.3 Output

As said before, we offer two ways to use our raytracer: a basic image rendering and an interactive mode. The basic mode outputs a picture using the PPM [3] format. In the interactive mode, each time the camera moves, its new coordinates are taken into account and the scene is recomputed. The graphics and events are handled using the SFML [4] library. To move the camera, use the keyboard arrows.

3 Implementation

The Figure 2 shows the camera position and its field-of-view (fov). Here are the steps[1] to compute outgoing screen vectors:

- Normalize u and v.
- Compute w using cross-product between u and v.
- Compute L, the distance between the eye and the center of the screen, according to $\tan(fov) = \frac{screenwidth}{2}$
- Compute C, the center of the screen according to $C - Lw = eye\_pos$.

1. This part is heavily based on a subject written by the ACU 2016 of EPITA

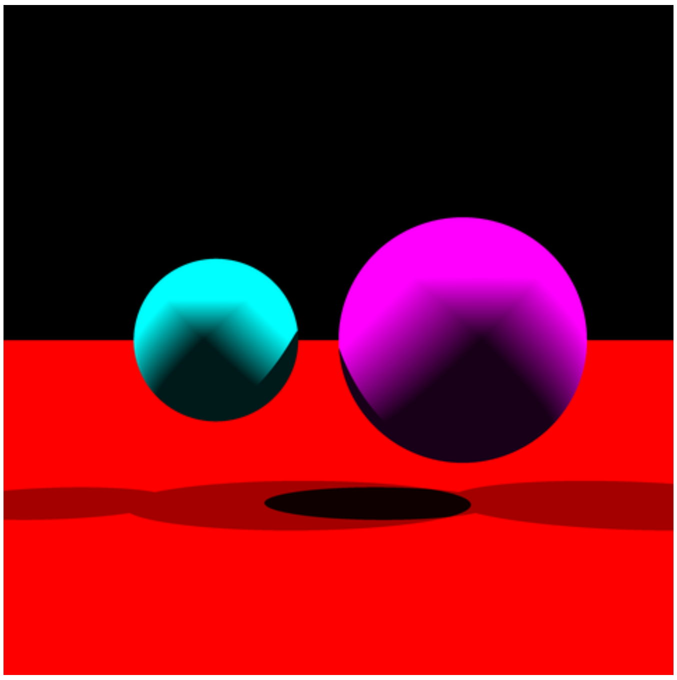

Figure 1. Example of a scene rendered with our raytracer to demonstrate ponctual lights and shadows

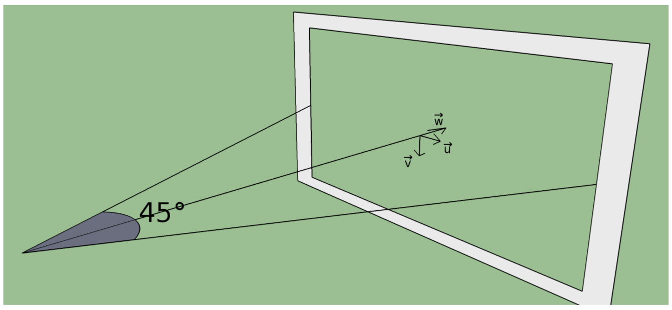

Figure 2. Raytracing concept

- Now, from $\frac{-screenwidth}{2}$ to $\frac{screenwidth}{2}$ and from $\frac{-screenheight}{2}$ to $\frac{screenheight}{2}$:
  - Compute the point on the screen using u and v.
  - Using this point and the eye position, compute the outgoing vector.
  - Using the position of the camera and the vector from the eye to the point on the screen, cast the ray.

3.1 Architecture

A class named data (Figure 5) is instantiated at the beginning of the program. Then the program parses the input file and fills the data. And since Sphere, Plane and Triangle are some

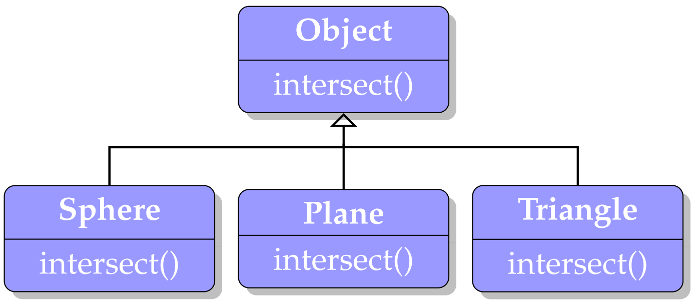

Figure 3. Base class Object and each kind of object that inherits from it.

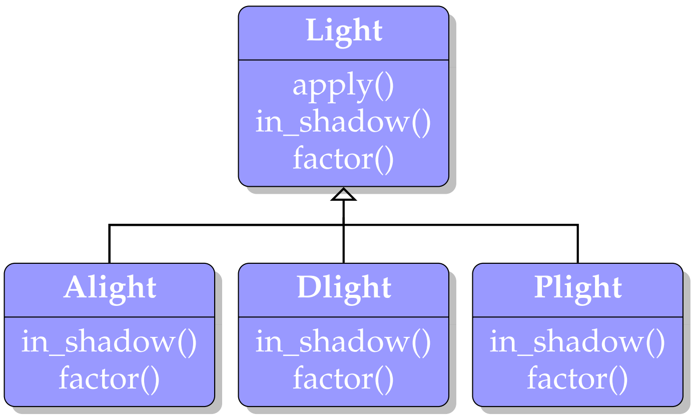

Figure 4. Base class Light and each kind of light that inherits from it.

type of Object, those specific objects inherit from a base class Object (Figure 3. Such an implementation makes the rest of the code more generic. We did the same with lights (Figure 4).

On the host side, only parsing and preparation of the data is done. The rest of the computations are done on the device. We associate one thread to each pixel of the picture. Let $N = width \times height$ be the total number of pixels to handle and $M = 32$ be the number of threads per block. The number of blocks is evaluated as follows: $(M + N - 1)/M$.

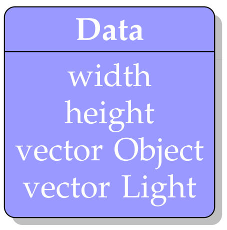

Figure 5. Data class

4 Performance

The main purpose of this project is to analyze and optimize the performance of a GPU implementation of a raytracer, and compare that to its CPU version. In this section we describe the current state of the project, the performance analysis that has been done, and the optimizations that are planned.

4.1 CPU vs GPU

The current progress of our project consists in two implementations of the raytracer: a CPU version (C++11) and a GPU version (CUDA 9.0 on GeForce GTX 1080 Ti). We tried to make them as similar as possible to compare them fairly. Figure 6 shows the time taken by both versions to render the same scene with varying image sizes.

At 250,000 pixels (500x500), the GPU version becomes faster than the CPU version. Taking this as a reference point, at 9Mpx (3000x3000) the CPU version has multiplied its runtime by 24.7 while the GPU version only multiplied it by 2.5. This is a clear win for the GPU, but the runtime of the GPU version is still too high for the interactive mode. We have already identified key bottlenecks for which we propose some solutions in the next section.

4.2 Optimizations

The rendering of a single frame contains three time-consuming operations (see figure 7):

- Device memory allocation (`cudaMalloc()`) for the scene data (objects, lights, etc.),
- Main kernel (`rt_eachpx()`),
- Data transfer from device to host (`cudaMemcpy()`).

In non-interactive mode, the bottleneck is `cudaMalloc()`. This is explained by the fact that the first call to `cudaMalloc()` incurs an inevitable cost that is orders of magnitude higher than the computation time of the main kernel or the data transfer. When rendering a single frame, this is very significant and this allocation represents between 80% and 90% of the total runtime of our application. But in interactive mode this cost is only paid once,

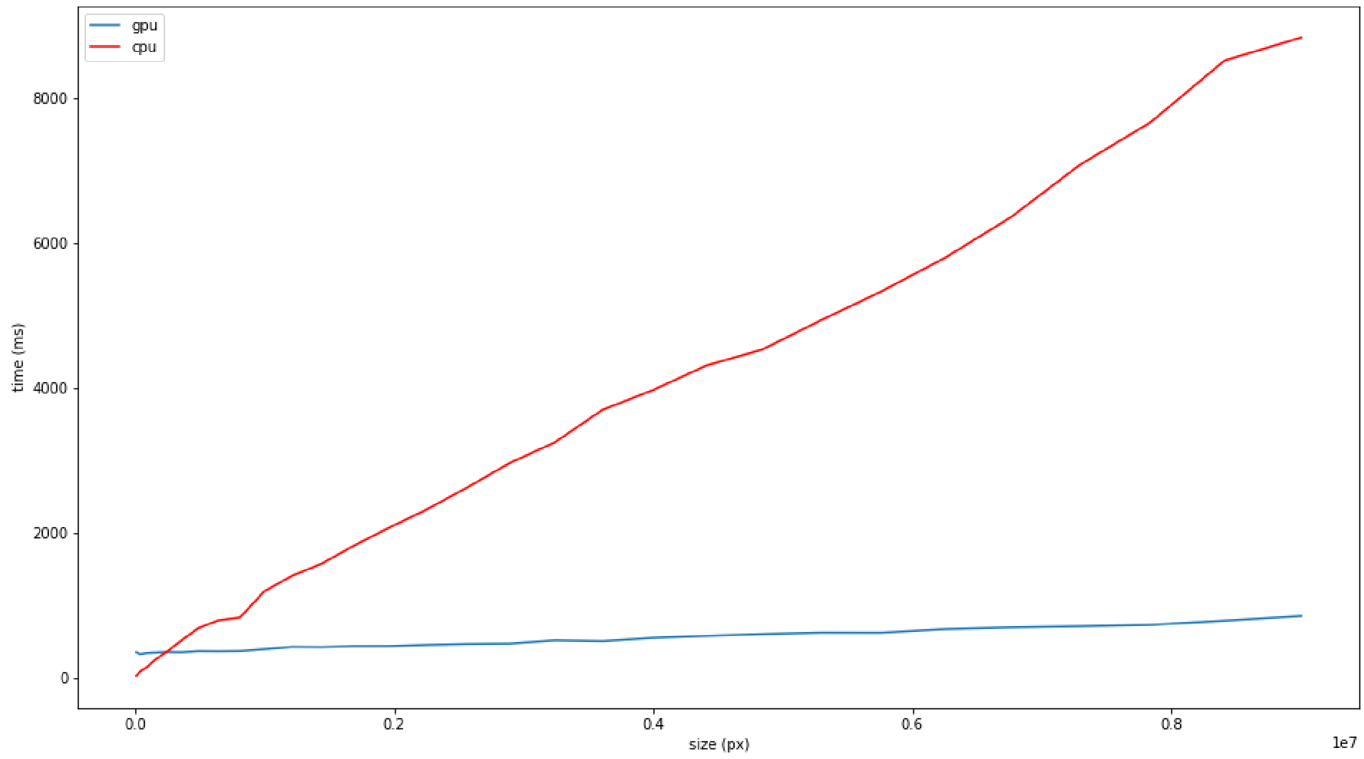

Figure 6. CPU vs GPU rendering of a scene

and the subsequent allocations take less than a millisecond each.

The kernel itself takes between 50ms and 100ms with a 1Mpx image (1000x1000). The `cudaMemcpy()` is negligible for now (below 1ms), which means that we can consistently display between 10 and 20 frames per second in interactive mode.

4.2.1 Next steps

There is a lot to do in terms of performance analysis and optimization. The main points on which we want to focus next are:

- Warp divergence: the main kernel contains many branching statements that probably slow down the application because of warp divergence. We want to measure the impact of this divergence in terms of performance and rewrite the kernel accordingly.
- Try more complex scenes and analyze the performance of the kernel itself.
- Play with different block sizes, dimensions, and number of threads per blocks.

There is not much to do in terms of streamlining since the only data transfer (image back to host) is extremely fast compared to the kernel itself.

5 Difficulties Encountered

We faced a few technical issues when translating our first C++ implementation to CUDA.

5.1 Derived classes

Instances of derived class (such as `sphere`, `plane` and `triangle` which inherit from `object`) turned out to be harder to handle than expected. Here are the main problems we had:

- Because of object slicing, we cannot create a vector of objects since this would call the copy constructor of the base class, and not of the derived classes. Our solution is to use a vector of pointers and to copy each object individually. Because of this we have to manage three separate entities: the host vector of host pointers, the host array

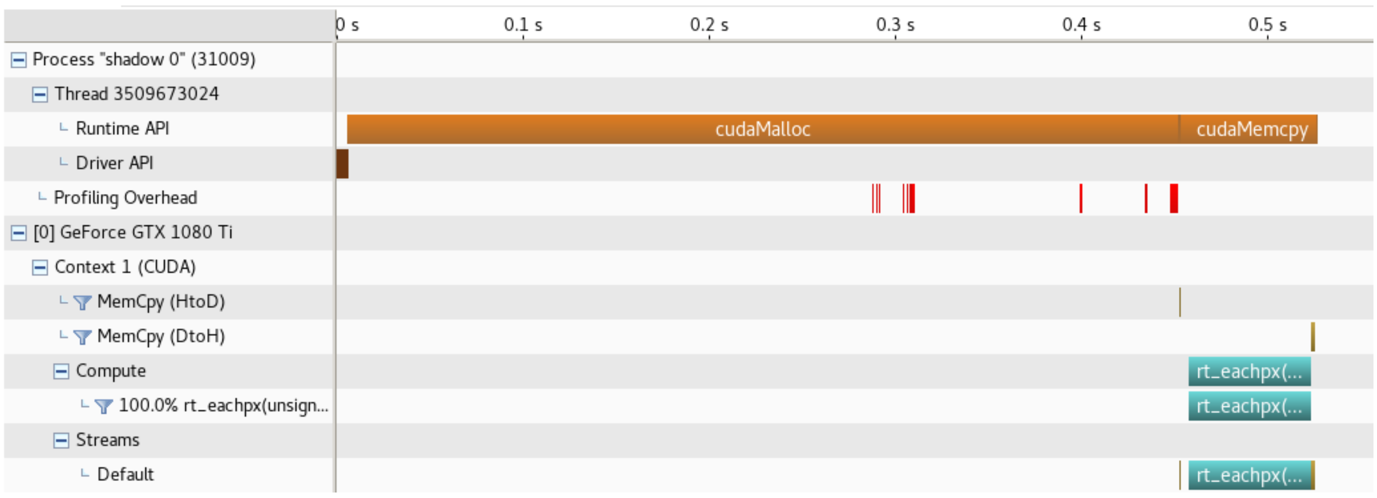

Figure 7. Visual Profiling of data obtained from the rendering of a single 3000x3000 frame

of device pointers, and the device array of device pointers.

- Instances of derived classes can be bigger than their base class, which means we have to `dynamic_cast` them individually to check their type and allocate the necessary space.
- Most importantly, the pointer to the vtable is invalidated when transferred to the device since it is a *host* pointer. As-is, the objects are useless after the transfer. Our solution is to check the objects individually again, and reinstantiate them from the kernel. This is made harder by the fact that `dynamic_cast` is not allowed from the device, preventing us from checking the dynamic types of the objects and instantiating them accordingly.

Of course, all these problems apply to the `light` class hierarchy as well.

5.2 Thrust library

To easily transfer vectors of objects/lights from host to device, we wanted to use the thrust library. But we found out that thrust's `device_vectors` are not meant to be used *from the device*: none of their member functions have the `__device__` attribute except for the constructor. Instead, they are meant to be manipulated from the host using the set of provided functions such as `thrust::transform`, `thrust::reduce` or `thrust::for_each`. Although we probably could have rewritten our raytracer using only these primitives, this would have defeated the purpose of the assignment.

6 Conclusion

As expected, the CUDA version of our raytracer is way faster than the CPU version on realistic image sizes, without any particular optimization. We can still improve the performance for a smoother rendering of our real-time interactive mode. We will start by analyzing the divergence of the warps, which we expect to be very high because of the many branching statements in the device code.

References

[1] "Rendering (computer graphics)," https://en.wikipedia.org/wiki/Rendering_(computer_graphics), accessed: 2017-11-13.

[2] H. Ludvigsen and A. C. Elster, "Real-Time Ray Tracing Using Nvidia OptiX," in *Eurographics 2010 - Short Papers*, H. P. A. Lensch and S. Seipel, Eds. The Eurographics Association, 2010.

[3] "PPM description," https://fr.wikipedia.org/wiki/Portable_pixmap#PPM, accessed: 2017-11-12.

[4] "SFML library," https://www.sfml-dev.org, accessed: 2017-11-12.