CUDA Accelerated Raytracer

Victor Guichard
IMAGE, EPITA
victor.guichard@epita.fr

Gaël de Francony
IMAGE, EPITA
gael.de-francony@epita.fr

Guillaume Valente
IMAGE, EPITA
guillaume.valente@epita.fr

Abstract

In this project we present three approaches to accelerate a simple **C++ raytracer** using **CUDA**. The first approach is almost a 1 to 1 translation. The second one is a complete rewritten of a recursive algorithm into an iterative one. Finally, the last approach tries to aim for a better memory managment.

1 Introduction

The raytracing rendering algorithm is now often used as a real-time rendering technique. For that purpose, optimizing this algorithm allows to render more beautiful images than a traditional real time rendering technique (z buffer for instance) witha minimum real-time rate (30 FPS).

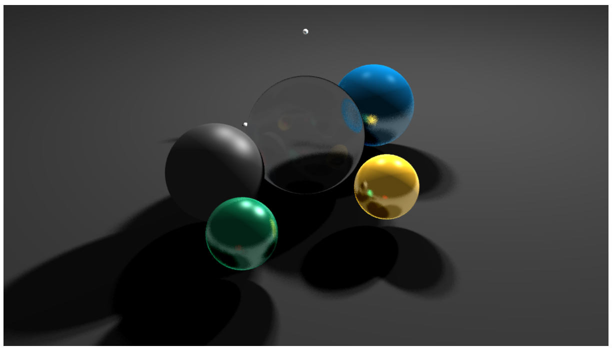


Figure 1: Render made using our raytracer

The raytracing algorithm is pretty straightforward. First the camera shoots width × height rays at different angles, each ray will describe one pixel. The rays then iterate over all objects in the scene to determine which hit object is the closest to the camera. Light is then computed and a check applies to see if an object is obstructing the light which creates a shadow. The light computation is done by shooting another ray at the intersection with a reflection angle and the algorithm same computation is then done.

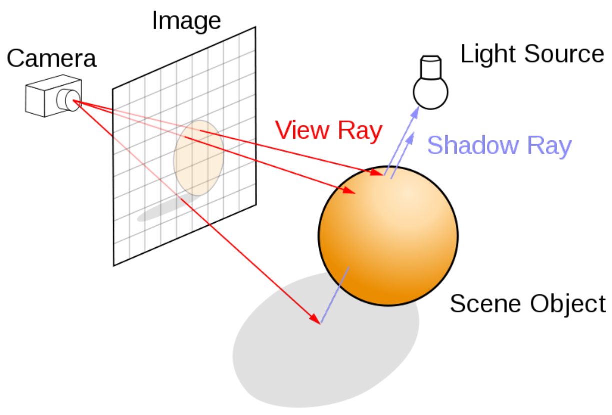


Figure 2: Raytracing algorithm explained

Raytracing is well suited for optimization like parallelization since each ray is casted independently and does not interfere with other rays. The unique common variable is the final buffer. For a multi-threading approach, each thread can render one pixel.

2 Description of CPU implementation (CPU Baseline)

2.1 Implementation Details

For our cpu implementation we tried to keep it simple as most optimizations are algorithmic and do not bring value for a GPU accelerated project (ex: BHV). It supports most features a raytracer might have such as refraction, reflection, multiple lights, sphere intersection as well as triangle intersection, shadows and depth.

2.1.1 Shape Interface

Sphere and **Triangle** classes will both implement the interface **Shape** that contains two methods: **getNormal** and **intersect** that will return the normal to the surface and return whether or not a ray hits a surface.

2.1.2 Light Interface

SpotLight, AreaLight, Directional light, etc.. implements the interface **Light** that contains required information to compute colors and an attenuation method. This polymorphic design helps having a better and more readable code.

2.1.3 Color Computation

The ray color will be computed using the object color and its position to each lights. Moreover, the color will be nulled (not visible by any light) or lowered (visible by some of the lights) if by casting a ray from the hit position and each lights an object is intersected.

Depending on the **Shape** proprieties and the current depth, a reflection ray and a refraction ray might be sent on collision and the two new colors will be merged with a reflection factor and a refraction factor by this simple formula:

$$Color(ray) = \begin{cases} object * shadow \\ \text{if } reflectivity = 0 \text{ and } transparency = 0 \\ \\ object * shadow + \\ Color(reflection) * reflectivity \\ \text{if } reflectivity > 0 \text{ and } transparency = 0 \\ \\ Color(reflection) * reflectivity + \\ Color(refraction) * transparency \\ \text{if } reflectivity > 0 \text{ and } transparency > 0 \end{cases}$$

2.1.4 Anti Aliasing

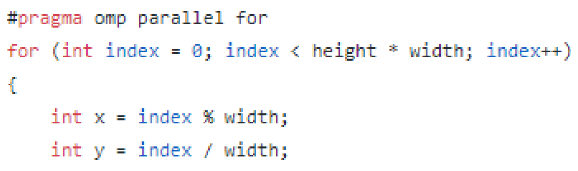
Anti Aliasing was made possible by scattering multiple ray for each pixel of the image instead of one. Usually at around 10 rays per pixel the aliasing issue is mostly fixed however this technique has a huge performance cost.

2.1.5 OBJ parser

Finally, the CPU implementation also contains a scene loader that parses **OBJ** file which allows to quickly create scene on Blender and then render the scene using this raytracer.

2.2 Parallel Computing (CPU OMP)

As the raytracing algorithm can be very simply paralleled, we decided to add a second CPU implementation that is using **OpenMP** and the compiler instruction **parallel for** before the main en-rolled loop[3].



```
#pragma omp parallel for
for (int index = 0; index < height * width; index++)
{
    int x = index % width;
    int y = index / width;
```

Figure 3: OpenMP instruction to parallelize the CPU raytracer

2.3 Performance

In order to benchmark these two cpu implementations we randomly create a scene containing 100 spheres with random properties and at a random position in front of the camera. Rendering a **1280x720** image with a depth of 5 and 5 rays per pixel was 14s long on the **CPU Baseline** and 7s long on the **CPU OMP** which is a 50% decrease in rendering time.

3 Bench-marking Experiment and Protocol

3.1 Comparing Execution Performance

As many versions were implemented during this project, A tool to quickly compare each version is needed. A benchmarking tool was therefore created that extract the overall execution time of the each raytracer.

In order to run a benchmark a **yaml** file[figure 4] needs to be created that contains:

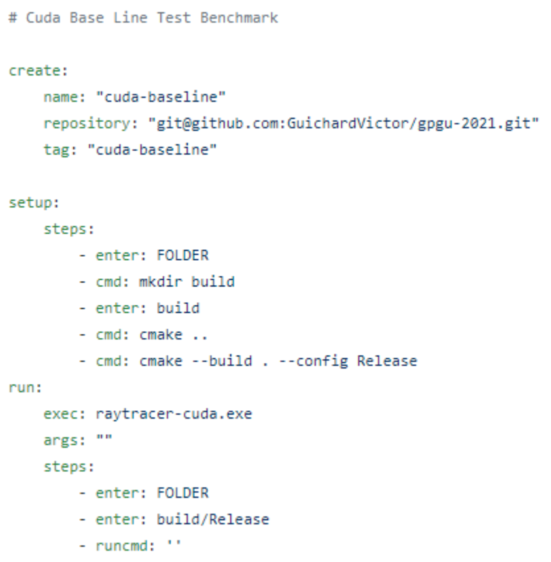
- A project location (git url, tag, branch, or local directory).
- A setup section that is used to compile or installed required package before running the benchmark.
- A run section that contains the setup to run the executable.

The tool will parse as many config file given as input and will run the executable a certain amount of times and output a summary of the execution time of each version as well as a relative time performance (figure 5).

However a part from comparing rendering time, this tool is not able to go deeper into the GPU usage.

3.2 Metrics and performance indicator used for CUDA

Performance metrics related to GPU usage are gathered using nvprof, a NVIDIA profiler avail-

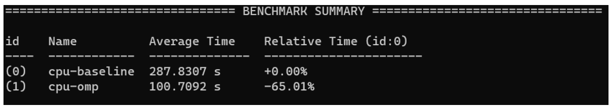


```
# Cuda Base Line Test Benchmark

create:
    name: "cuda-baseline"
    repository: "git@github.com:GuichardVictor/gpgu-2021.git"
    tag: "cuda-baseline"

setup:
    steps:
        - enter: FOLDER
        - cmd: mkdir build
        - enter: build
        - cmd: cmake ..
        - cmd: cmake --build . --config Release
run:
    exec: raytracer-cuda.exe
    args: ""
    steps:
        - enter: FOLDER
        - enter: build/Release
        - runcmd: ''
```

Figure 4: CUDA baseline benchmark config



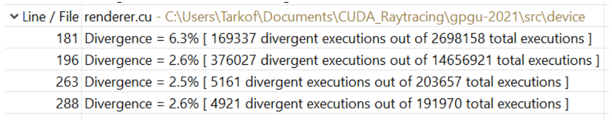
```
============================== BENCHMARK SUMMARY ==============================

id    Name          Average Time    Relative Time (id:0)
----  ------------  --------------  ----------------------
(0)   cpu-baseline  287.8307 s      +0.00%
(1)   cpu-omp       100.7092 s      -65.01%
```

Figure 5: Benchmark summary: **CPU Baseline** - **CPU OMP** 1000 spheres, rays per pixel: 5, max depth: 5, 1280x720, run 5 times

able for Volta and previous architectures.

Nvprof can be combined with NVDIAs Visual Profiler to automatically generate analysis reports. These metrics allows us to have a deeper understanding of the versions implemented and the possible impact optimizations can have. Kernel sampling distribution (as seen in figure 7) indicates which parts of the code the program spends most of its time. We use these metrics to identify critical sections of our code and gives us an indication on what we should focus our attention to improve performances. We are also interested in memory statistics (example figure 11). With this kind of reports we can understand how memory is managed and identify possible memory bottlenecks. Visual Profiler also generates reports on divergent execution (seen in figure 6). This can indicate inefficient use of computational resources in specific parts of the codes. Ideally all threads in a warp should branch together.



Line / File	renderer.cu - C:\Users\Tarkof\Documents\CUDA_Raytracing\gpgu-2021\src\device
181	Divergence = 6.3% [169337 divergent executions out of 2698158 total executions]
196	Divergence = 2.6% [376027 divergent executions out of 14656921 total executions]
263	Divergence = 2.5% [5161 divergent executions out of 203657 total executions]
288	Divergence = 2.6% [4921 divergent executions out of 191970 total executions]

Figure 6:

4 Baseline of GPU implementation

4.1 Description

Most of the code base is similar between the **CPU Baseline** and the **CUDA Baseline**, this is allowed by the nature of the raytracer algorithm. Therefore most classes and iterface have been able to be kept with some minor changes (such as adding **__device__** and **__host__** on methods). However, small parts of the code base still needed some changes.

4.1.1 renderScene kernel

The loop is removed as this kernel is run by every threads. Therefore the thread id is used to get the framebuffer index but also to compute the rays origin and direction.

4.1.2 setupScene kernel

As it was not possible to simply init a scene from the cpu, the setupScene kernel creates a scene, the camera and the renderer objects. Due to this limitation and to save time, we decided to drop the features of reading and parsing **OBJ** files as it did not provide values on performance nor CUDA programming.

4.1.3 Random

For both Area Lights and Anti Aliasing, randomness is involved in scattering rays. Random in CUDA is a bit different from random in CPU. a **local_rand_state** buffer was first initialized with **curand_init** and shared to kernels. Every method or functions that required random values pick its random state at the same index of the framebuffer. Random initialization can take a long time but is required once and can be reused for multiple renders. **curand_uniform** was used to generate a random number but different algorithms can also be used and might some time yield to better performance.

4.2 Performance analysis

From the performance analysis, we detected several issues. Regarding the memory, we discovered that the memory dependency was huge (79%). Using flop_count_dp nvprof metric, we noticed some double precision operations remaining.
Performances in terms of speed compared to CPU implementations were a lot better (2). Despite having better result than CPU, the CUDA recursive implementation has a new limitation which is the depth of the trace function. CUDA compiler

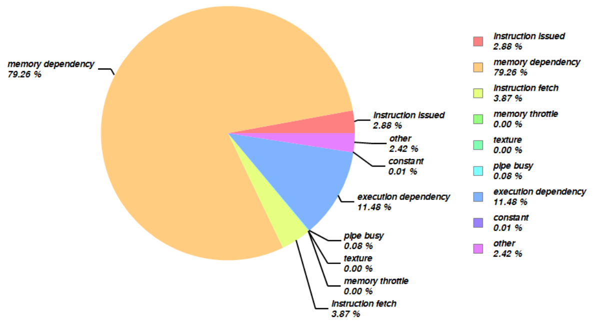


Figure 7: Performance with CUDA Baseline

sets a maximum depth for the stack size, giving limitations for recursive functions (in our case, the maximum depth is 7).

4.3 Proposed solutions

First of all, from the issues found in the performance analysis, we were able to fix most of the simple mistake made such as some double precision calculus, or the incorrect usage of the framebuffer as it was accessed many times instead of once in the **renderScene kernel**. However, from this issues we were still limited in ray depth and the global memory usage were still not tackled. Therefore, we proposed two other CUDA versions, one that is a complete rewrite of the raytracing algorithm: **CUDA Iterative** and one that tries to reduce the global memory usage by using the shared memory of the GPU.

5 Iterative CUDA implementation

5.1 Description

The iterative implementation simulates a recursion implementation. We were forced to implement that approach since the raytracer algorithm structure is recursive. To do so, we use different arrays to act as a tree. Arrays are filled within a for loop that increments. Since it's a tree that is represented as an array, the following applies:

$$\begin{cases} parent = i \\ leftchild = 2 \times i \\ rightchild = 2 \times i + 1 \end{cases}$$

The code is structured in two parts: the first one computes the color and rays, second uses the last depth colors to compute the colors at depth -1.

5.2 Performance analysis

The first implementation done gave bad results. For the same scene, the iterative version rendered in 10s and for recursive 1.5s (Geforce GTX 860m). We then conclude that memory allocation the main bottleneck. First implementation was using malloc which took time to allocate memory on the heap.

5.3 Issued solutions

To overcome the memory allocation problem, we tried static array allocation. To do so, the main issue is knowing the allocating size at compile time. Hopefully the recursion was made with 2 branches and power of two are the only ones that can be computed at compilation using bit-wise operators ($1 << depthasked$).
This solution raises other issues, first we have to know the depth at compile time, second, the allocating arrays are not entirely used (when ray does not touch any objects for instance). With this solution, if the depth is too high, the array cannot be allocated. The ideal solution would be one that does not depend on the depth but reallocate necessary space without slowing the render.

5.4 Iterative improvement

With the new iterative implementations, the trace function can reach higher depth. Some issues are still remaining; the memory depedency is the same as the CUDA baseline(7) and although the iterative implementation gives better results with high depth and a lot of objects (2), with simple scenes and high depth, the performances are bad compared to recursive implementation (0.092s for iterative, 0.056s for recursive with 9 objects and depth 7).

6 Shared Memory CUDA implementation

Shared memory is an on-chip memory. This makes it much faster than local and global memory. In fact, in good condition, shared memory latency is in orders of magnitude lower than uncached global memory. Shared memory is allocated per block and, as its name suggests, it is shared between the threads of the same block. It enables user-managed data caches to developers.

The goal of this implementation is to prioritize the use of shared memory over global memory. In previous implementation, most of our data structures were allocated in global memory (see figure **??**. We decide to focus on the Scene structure as the number of memory access to other structures are negligible.

The Scene can be loaded into shared memory with little change to previous implementations. Only two additional steps are required. First, dynamic shared memory allocation is done at kernel launch. The size of the buffer to allocate needs to be specified at that moment. Thus, the size needs to be computed before the launch. This is done when building the scene in the first kernel, setupScene. Then, at the start of the renderScene kernel one thread per block traverse the scene and copy it into the buffer. The new scene allocated in shared memory is then used by all threads for rendering.

6.1 Performance analysis

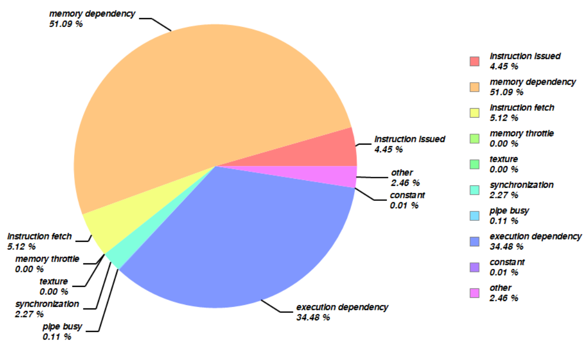


Figure 8: Performance with CUDA Baseline

The majority of global memory usage was successfully replaced by shared memory. The global load throughput decreased by a factor of 6, as seen in table[1]. However there is a decrease in the kernels performance with a reduced rendering speed to 28.7% of the baseline speed (table[2]. Shared usage efficiency is low, at 3.07%, indicating a wrong usage of the memory. This is caused by bank conflicts that serialize memory access. There are two other causes to this lack of efficiency. First, object structures are very large. Spheres have a size of 24B and triangles of 60B. In a system with 32 banks of 4B only 5 spheres or 2 triangles can be loaded per transaction. Secondly, in the current implementation all objects are loaded into shared memory, so its maximum size can easily be reached.

6.2 Proposed solutions

The use of an acceleration structure such as a BVH (Bounding Volume Hierarchy) could improve the performance. It could be possible to load only a portion of the objects into shared memory and thus reduce the load on shared memory. Objects size could also be reduced by removing polymorphism or by changing the approach to how they are stored in memory. For example, triangles could be stored using a vertex array / index.

7 Results And Benchmarks

7.1 Rendering Time

Using the tool and the metrics previously declared, results can be retrieved efficiently and can present meaningful insight on the performance of each versions.

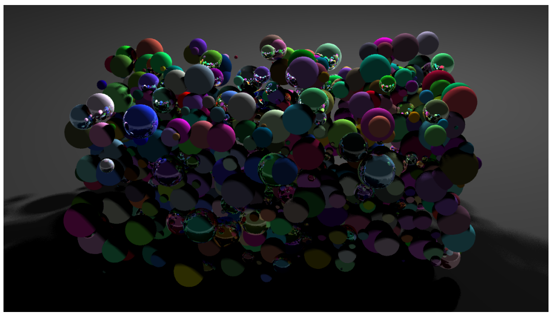


Figure 9: Scene used to benchmark render time.

The table (2) displays the render time of a 1001 spheres scene(4) of every version and the relative time compared to the **CPU baseline**. From the two **CPU** versions we can see how the algorithm take full advantage of parallel programming as the time to render a frame is divided by half. The performance increase is even more striking (-99% of time to render a frame) when comparing the **CPU baseline** with the **CUDA baseline** without improvements as it is almost a 1 to 1 translation. When comparing the two **CUDA Baselines** we can see how much simple fixes such as double precision operations removal can impact on the performance. Even if the code base is very similar the fixes divide by two the rendering time.

7.2 Image Quality

During this project, we made sure that a new version must not decrease the visual quality of the image. By zooming on a render(10) of the **CPU** and **CUDA** versions we can see no difference a part from the randomness of the scattered rays.

Metric Name	Description	GPU baseline	Shared
shared_efficiency	Shared memory usage efficiency	0.00%	3.07%
shared_load_throughput	shared_load_throughput	0.00000B/s	13.595GB/s
gld_efficiency	global load efficiency	18.19%	19.80%
gld_throughput	global load throughput	2.6201GB/s	438.33MB/s

Table 1: Metrics of shared memory implementation

Render Settings	Version Name	Average Time	Relative Time
Resolution: 1280x720 Max depth: 5 Rays per pixel: 5 Sphere count: 1001 Executed: 10 times	CPU Baseline	273.46s	+0.00%
	CPU OMP	95.42s	-65.11%
	CUDA Baseline	4.18s	-98.50%
	CUDA Baseline + fixes	2.55s	-99,07%
	CUDA Iterative	2.40s	-99,12%
	CUDA Shared	12.74s	-95,34%

Table 2: Rendering Time of each versions (CPU: AMD 3700x, GPU: NVIDIA 1660 super)

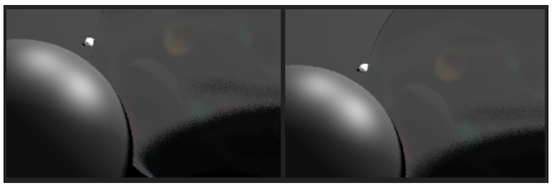


Figure 10: Left CUDA Baseline, Right CPU Baseline

7.3 In Depth GPU utilization

Each CUDA implementation has its own strength and weaknesses.

1. CUDA baseline: This implementation allows a fast rendering for scenes with few objects. This version is limited on the depth.

2. CUDA iterative: This implementation allows a fast rendering for big scene and allows to reach **deeper depth** but has bad performances with scene containing few objects.

3. CUDA shared memory: This implementation tries to reduce the memory depency but it still requires improvement.

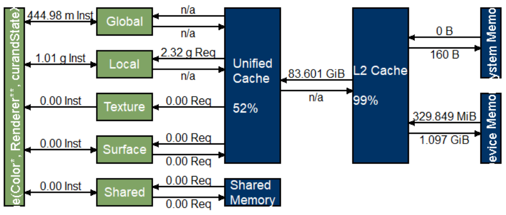


Figure 11: CUDA baseline : Memory statistics

8 Conclusion and Future Work

Ray tracing benefits a lot from GPUs. Rendering is accelerated by orders of magnitude in comparison to CPU only implementations. The algorithm is able to take full advantage of their highly parallel structure. However, a naive implementation leaves a lot of room for improvement. Changing the ray bouncing recursion to an iterative loop decreases the risk of stack smashing in the device and can increase the available maximum depth. Performance can be easily improved by reducing the number of double precision operations. It can also be improved by reducing the amount of global memory access using locally allocated buffers or used-managed cache like shared memory.

Previously cited optimizations focus solely in GPU accelerations yet, ray tracing performance can also be improved algorithmically. An example could be the use of acceleration structures such as a bounding volume hierarchy (BVH). It is used to reduce the number of intersection candidates by ignoring triangles within a non intersecting volume.

Ray tracing also opens the door to a number of others physically based rendering techniques such as path tracing, bi-directional path tracing or metropolis sampling. There is also work on advanced material rendering such as microfacet models or subsurface light scattering. All these only scratch the surface of currently available techniques.

In recent years, ray tracing has had a special surgence in popularity for its application on real-time rendering. A feat made possible with the release of

NVIDIAs Turing architecture and their RT cores. These cores enable hardware specific acceleration for triangle intersection and BVH.

8.1 Members contribution

CPU and CUDA baseline were made separately by each of us and then we kept each parts that we identified as relevant. When the baselines were confirmed we tried different approach on optimizing the CUDA version. The overall contribution can be seen in the table[3].

Project Main Aspect	Victor Guichard	Guillaume Valente	Gael De Francony
CPU Baseline	X (+OMP)	X	X
CUDA Baseline	X	X	X
CUDA Iterative	X	X	
CUDA Shared Memory		X	X
Benchmark Tooling	X		

Table 3: Members Contribution