KU LEUVEN

Project: Advanced Computer Architecture

Raytracing in a fiber

Daan Vercammen
Klaas Meersman

ELICT: ELECFS
17/05/2025
Master in Engineering Technology
Academic year 2024-2025

1.1 Introduction

Simulating light propagation within optical fibers is computationally intensive, especially when modeling the emission and transport of a large number of rays from a realistic source such as an LED. This project addresses these challenges by leveraging GPU parallelism to accelerate raytracing. The goal is to simulate rays emitted from an LED at the fiber entrance, trace their multiple reflections inside the fiber, and analyze the spatial density of rays at the fiber's exit.

1.2 Motivation for GPU Acceleration

Raytracing is inherently parallel: each ray's trajectory is independent, making it ideal for GPU execution, where thousands of threads can run concurrently. In contrast, CPUs have far fewer cores, resulting in much longer runtimes for large-scale simulations. GPU acceleration enables efficient simulation of millions of rays, drastically reducing computation time.

1.3 Implementation

To simplify initial development, the simulation begins in 2D (Figure 1.1), defining the fiber's length and height, the number of rays, and their initial directions. Each ray's direction is updated at every wall collision. The final height of each ray at the fiber's exit is stored for analysis.

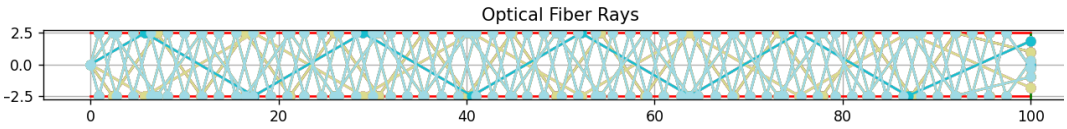

Figure 1.1: Initial 2D-model

The project then extends to 3D (Figure 1.2), introducing two angular parameters and a three-axis coordinate system. We will only focus on this part.

1.3.1 Project repository

The GPU version can be found on branch GPU_3D, the CPU version can be found on branch CPU_3D. Earlier versions of the 2D implementation can be found for both GPU and CPU.

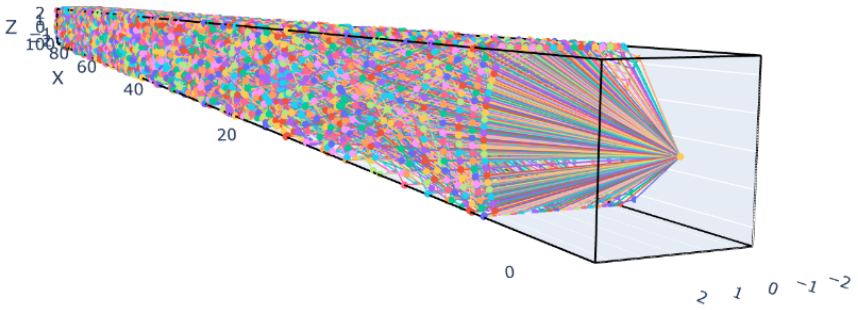

Figure 1.2: Plotly is used for 3D visualization in Python.

1.3.2 Code Structure

The project 3D GPU code is organized as follows:

- `srcGPU`: Contains CUDA C++ code for the GPU implementation (`main.cu`, `ray.hpp`, `fiber.hpp`, `coordinate.hpp`).
- `density.py`: Python script for running the simulation, parsing output, and plotting the density.

1.3.3 Raytracing Algorithm

Our implementation uses CUDA to run each ray on a separate GPU thread (in `__global__ void traceRayGPU()`). Initial positions and directions follow a Lambertian emission profile to mimic an LED, which are generated and initialized in the kernel (`__global__ void initRaysBinned()`). As rays propagate, collisions with fiber walls are computed using geometric optics (in `void Ray::propagateRay()`), and the process repeats until the ray exits the fiber. Only the exit positions are used for further analysis.

1.3.4 Parallelization and Optimization

To maximize performance, several optimization strategies are employed. First, the initialization and propagation of rays are performed entirely on the GPU (in `Ray* runTraceRayGPU()`), minimizing data transfer between host and device. Only the final endpoints of the rays are copied back to the host for analysis and visualization, thereby reducing memory traffic. Additionally, rays are grouped into bins based on their initial angles (in `Ray* runTraceRayGPU()`) to reduce divergence within GPU warps, as threads following similar execution paths are more efficiently processed by the hardware. We achieved about a 40% increase in performance doing this. The use of the cuRAND library ensures that random number generation for generating the Lambertian emission pattern is both efficient and statistically robust (in `__global__ void initCurandStates()`), further contributing to the accuracy and speed of the simulation. We optimized block size and number of bins for this specific case. These numbers were optimal for as long as we kept using a high amount of rays, and a sufficiently long fiber.

1.3.5 Attempts That Did Not Improve Performance

- Overlap data transfers and computation using CUDA streams (asynchronous execution) did not make a noticeable difference.
- Copying only endpoints versus entire ray objects made no measurable difference, likely due to a ray object still being within the minimal object size.
- Striding had no noticeable improvement in performance.

1.4 Results

Our approach gives good results. For one million rays, the GPU implementation is about ten times faster than the CPU version (Figure 1.4). The results (Figure 1.3) are processed and visualized in Python. The density of the rays is uniform on both CPU and GPU, which was also expected. Correct propagation was verified using a ray plot (Figure 1.2).

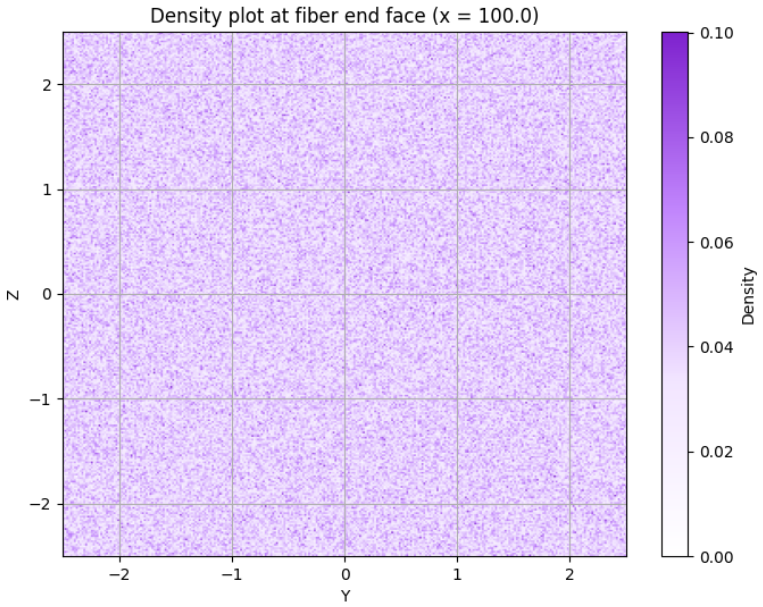

Figure 1.3: Density plot at the fiber end: uniform distribution

1.5 Discussion

1.5.1 Shortcomings

Possible performance improvement

Divergence within GPU warps, while mitigated by angle binning, cannot be entirely eliminated due to the stochastic nature of ray paths and the varying number of reflections experienced by different rays. Memory usage is another consideration, as each ray is represented as a full object, which could be further optimized by adopting a structure-of-arrays layout or by storing only essential data. These techniques are typically used

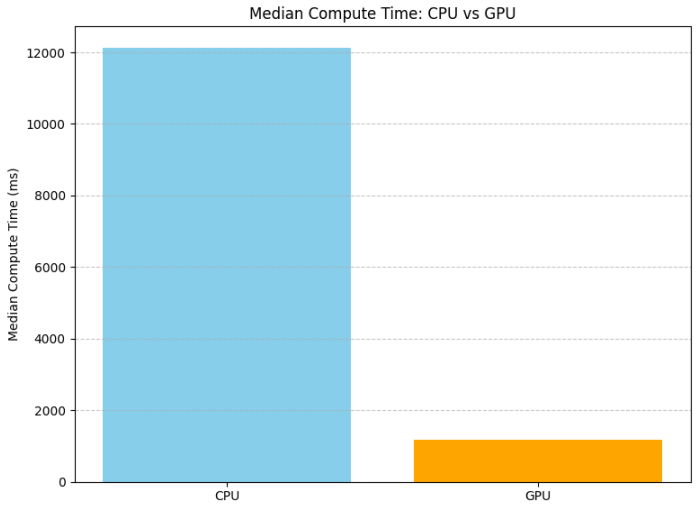

Figure 1.4: Performance comparison: CPU vs GPU runtime for 10^6 rays. The GPU implementation achieves a ∼10x speedup.

for programming in 3D space. Lastly, we could use CUDA profiling tools to identify and address remaining bottlenecks.

Features to add

The current implementation focuses on geometric optics. Extensions to include more complex physical phenomena, such as polarization or wavelength-dependent effects, would require additional computational resources and algorithmic enhancements. Modeling bending cylindrical fibers would require more complex mathematics for surface normals and intersection calculations. We could also include a dispersion graph, as our current implementation makes it straightforward to calculate the distance each ray travels. Using these distances, we can create a dispersion graph to visualize how spread out the initial signal is.

1.6 Conclusion

GPU acceleration makes large-scale raytracing in optical fibers practical and efficient. By exploiting parallelism, optimizing data movement, and reducing divergence, this project achieves significant performance gains without sacrificing accuracy. We could further improve our architecture by adding extra features and complexity.

REFERENTIELIJST

[1] *CUDA: Determining threads per block, blocks per grid*. Stack Overflow. Geraardpleegd op 17 februari 2025. 2010. URL: https://stackoverflow.com/questions/4391162/cuda-determining-threads-per-block-blocks-per-grid.

[2] NVIDIA Corporation. *CUDA C++ Programming Guide*. Geraadpleegd op 13 februari 2025. NVIDIA. verschillend. URL: https://docs.nvidia.com/cuda/cuda-c-programming-guide/.

[3] Alex Minnaar. *Grid-stride loops*. Geraadpleegd op 3 maart 2025. Aug. 2019. URL: https://alexminnaar.com/2019/08/02/grid-stride-loops.html.

[4] NVIDIA Corporation. *cuRAND API Reference Guide*. Geraadpleegd op 17 mei 2025. NVIDIA. 2025. URL: https://docs.nvidia.com/cuda/curand/index.html.