Real-Time Path Tracing Engine in CUDA + OpenGL

Enrique Gomez (exg), Maxwell Guo (mwguo)
https://mwguo15.github.io/path-tracing/

Summary

We propose to build a real-time global illumination renderer using Monte Carlo path tracing entirely on a GPU. Our engine will use CUDA to trace rays in parallel and produce progressively refined images of a 3D scene. An OpenGL front-end will display the rendered image in real time, continuously updating the view and resetting the accumulated result whenever the user interacts (e.g. moves the camera). The goal is to achieve interactive frame rates at low-to-moderate resolutions by leveraging massive parallelism and careful optimizations.

Background

In path tracing, rays are cast from the camera into the scene and bounce around, interacting with surfaces and light sources to compute the final image. This method naturally simulates global illumination, including reflections, refractions, and shadows, producing highly realistic lighting effects.

The application consists of several core components:

1. **Ray Generation**: Rays are cast from the camera through each pixel, representing potential paths of light
2. **Intersection Tests**: For each ray, intersection tests are performed to determine which object in the scene the ray hits. This includes testing against bounding volumes and maybe more detailed geometry
3. **Path Tracing**: Upon hitting a surface, additional rays are cast to simulate reflection, refraction, and shadows, with each bounce continuing the light simulation
4. **Lighting and Shading**: Each intersection contributes to the final color of the pixel based on material properties, light source information, and possibly other environmental factors

If we parallelize ray generation, intersection tests, and path tracing bounces, we can significantly speed up the path tracing process, which is essential for real-time rendering.

The Challenge

While a basic ray tracer is often considered data parallel because each pixel's ray is independent, a full path tracer introduces a host of nontrivial parallel challenges that make it an ideal project for exploring advanced GPU optimization techniques. In a path tracer, every pixel spawns many rays that bounce through the scene in a stochastic manner to capture global illumination, meaning that each ray can follow a different path and terminate at varying bounce depths. This results in highly irregular workloads and severe warp divergence on GPUs—some threads finish their work quickly while others are burdened with many bounces—leading to load imbalance. To tackle this, our project will implement advanced techniques such as active ray compaction, which filters out terminated rays using parallel prefix sums or stream compaction, thereby dynamically rebalancing the workload across GPU threads.

Additionally, the use of acceleration structures like bounding volume hierarchies (BVHs) further complicates the problem. Although BVHs are conceptually simple—grouping primitives into hierarchical bounding boxes to quickly eliminate large portions of the scene—their efficient traversal on the GPU is challenging. Recursive BVH traversal must be restructured into iterative, data-parallel algorithms that maintain coherent memory access patterns while minimizing divergence as different rays traverse different parts of the tree. Furthermore, managing irregular memory accesses during the BVH traversal and during the progressive accumulation of samples per pixel demands a deep understanding of GPU memory hierarchies and cache behavior.

In summary, our project is far more than just shooting rays. It requires us to design algorithms that can adapt to non-uniform, dynamic workloads and to optimize memory access and thread scheduling for maximum throughput on modern GPU architectures. These challenges are not typically encountered in a simple ray tracer, making the path tracer an excellent case study in advanced parallel programming and GPU optimization techniques.

Resources

Our development and testing will primarily occur on our personal machines, which are equipped with NVIDIA GPUs (an RTX 3060 and an RTX 3070), while we will also leverage the GHC lab machines with RTX 2080 GPUs for further performance evaluation. We plan to write the path tracer largely from scratch, drawing on foundational concepts from resources such as "Ray Tracing in One Weekend" and various GPU path tracing tutorials available online. To aid development, we will utilize GLM (OpenGL Mathematics) for vector operations and CUDA's curand library for high-quality random number generation. These resources, combined with the CUDA Toolkit and OpenGL libraries (likely using GLFW or GLUT for window management),

provide us with a complete ecosystem to explore and optimize the irregular, dynamic workload of path tracing on modern GPUs.

Goals and Deliverables

By the end of the project, we plan to deliver a working GPU path tracer along with a thorough performance analysis. Below we outline our expected outcomes:

- Core Deliverables (Plan to Achieve): We plan to deliver a fully functional path tracing engine running entirely on the GPU that can render images with global illumination effects (e.g. soft shadows, indirect lighting, color bleeding) in real time or near real time. The interactive viewer will allow a user to navigate a scene (move the camera) and see the image progressively refine. We will demonstrate the renderer on a simple yet illustrative scene (for example, a Cornell Box or a few reflective objects and area light sources) where global illumination is clearly visible. Essentially, if all goes as planned, we expect to show that our GPU implementation achieves a significant speedup (potentially tens to hundreds of times faster) over a CPU approach and meets the real-time performance threshold at low resolution.

- Stretch Goals (Hope to Achieve): If we get ahead of schedule, we have a couple of extension ideas to further enhance our project. One stretch goal is to incorporate a denoising technique or smarter sampling methods to improve image quality per sample. For example, we could integrate a filter-based denoiser to the output so that fewer samples are needed for a clean image, thus effectively boosting the visual performance. Another possible extension is to support more complex scene features, such as textured surfaces or additional material types (dielectrics, thin glass) that we initially might simplify. These features would increase the realism of our renderer. We would also try to scale our implementation to handle higher resolutions.

- Minimum Viable Product: In case we encounter unexpected difficulties, we will ensure at minimum to deliver a correct parallel path tracer that may not reach real-time frame rates but can still generate correct images given enough time. For instance, if performance falls short, we would still have a GPU path tracer that outperforms a CPU reference for the same number of samples, and we would analyze where the bottlenecks occurred. This fallback ensures we have something substantial even if optimizations prove trickier than expected.

- Live Demo Plan: We plan to perform a live demo at the poster session to showcase our real-time path tracing. The demo will involve running our program on a laptop or PC and allowing viewers to interact with the scene. For example, we'll let users move the camera around a Cornell Box scene with colored walls and see the indirect lighting (color bleed)

update progressively. As the view moves, the accumulated image will reset and then quickly converge to a visually pleasant result. This interactive demonstration will prove that our renderer is indeed running in real time. We will also prepare a few rendered images or short video clips as backup, highlighting before-and-after comparisons (e.g. without vs. with global illumination, or low vs. high sample count) to illustrate the effects of our project. The live demo is an exciting optional component; regardless of it, our evaluation will focus on the quantitative performance results as described.

Overall, by the end of the 5-week project, we expect to deliver a complete GPU-based path tracing system that not only runs efficiently but also serves as a learning vehicle for parallel programming techniques. We will have applied concepts from the class (such as exploiting parallelism, managing memory hierarchy, and handling divergent control flow) to a challenging graphics problem, and we'll document the lessons learned on our project webpage. This project will be both a functional real-time renderer and a case study in optimizing an irregular parallel workload on GPU hardware.

Platform Choice

We have chosen to implement our renderer in C++ using CUDA, targeting NVIDIA GPUs for several compelling reasons. NVIDIA's GPUs are designed for high-throughput, data-parallel workloads, making them ideally suited for the massive number of independent computations required by path tracing. Each pixel in a rendered image spawns many rays, and NVIDIA's architecture—with thousands of cores and efficient thread scheduling—enables us to handle these irregular and dynamic workloads effectively. Additionally, by integrating CUDA with OpenGL via interoperability (using pixel buffer objects), we can efficiently transfer data from the GPU to an OpenGL texture for real-time display, allowing for interactive visualization without incurring significant overhead. This combination not only meets our performance goals but also provides a robust platform for tackling the advanced parallel challenges inherent in our project.

Schedule

Week 1 (March 26 – April 1): Setup, Research, and CPU Prototype
During the first week, we will focus on setting up our development environment and building a solid foundation. We will establish our repository, configure our build system, and study references on ray and path tracing (e.g. "Ray Tracing in One Weekend"). Our goal is to implement a simple CPU-based ray tracer that renders a basic scene (using spheres and a ground plane) and outputs an image (PPM format). This prototype will help us understand the core math behind ray generation, intersection, and diffuse shading, and it will serve as a reference for our CUDA implementation.

Week 2 (April 2 – April 8): Porting to CUDA and Initial GPU Implementation
In the second week, we will port the core functionality of our CPU path tracer to CUDA. We will write a naive CUDA kernel that maps one thread per pixel to perform ray generation, intersection tests, and color computation for our simple scene. Our primary goal is to ensure that our GPU version produces similar results to the CPU version, albeit at lower performance initially. We'll also begin experimenting with CUDA's memory management and verify that our kernels run correctly on our target NVIDIA GPUs.

Week 3 (April 9 – April 15 – Milestone Deadline): Multi-Bounce Extension and Preliminary Optimizations
With a working single-bounce GPU path tracer in place, we will extend our implementation to support multiple bounces per ray, simulating global illumination. At this stage, we'll also begin incorporating basic material support (such as Lambertian diffuse and a simple specular reflection) and start designing a Bounding Volume Hierarchy (BVH) for accelerating intersection tests. We will profile our initial GPU code to identify performance bottlenecks and document preliminary performance data. All this progress will be compiled into our milestone report, which is due by April 15th.

Week 4 (April 16 – April 22): Acceleration Structures and Advanced Optimization
The fourth week is dedicated to performance improvements. We will integrate the BVH into our GPU pipeline to reduce the number of intersection tests per ray, which is critical for complex scenes. Alongside BVH integration, we will optimize memory access patterns by reorganizing our data structures for coalesced global memory accesses and explore active ray compaction techniques to handle divergent workloads from varying bounce counts. Using profiling tools (e.g. NVIDIA Nsight), we'll iterate on these optimizations to maximize GPU occupancy and throughput, updating our project webpage with performance improvements and findings.

Week 5 (April 23 – April 28): OpenGL Integration, Interactivity, and Finalization
In the final week, we will integrate CUDA–OpenGL interoperability by creating an OpenGL context (using GLFW) and setting up a pixel buffer object or texture for real-time display of the rendered image. We will implement basic interactive controls, such as camera movement and accumulation reset, so that the image progressively refines as the user navigates the scene. Final testing, debugging, and performance measurements will be conducted during this week. We will prepare our final report and poster materials, ensuring that our live demo clearly showcases our system's functionality and the advanced parallelization techniques we implemented.