15-418 Final Project Proposal

Ray Tracing Parallelization

Jonathan Ke & Kavya Tummalapalli

Project Web Page

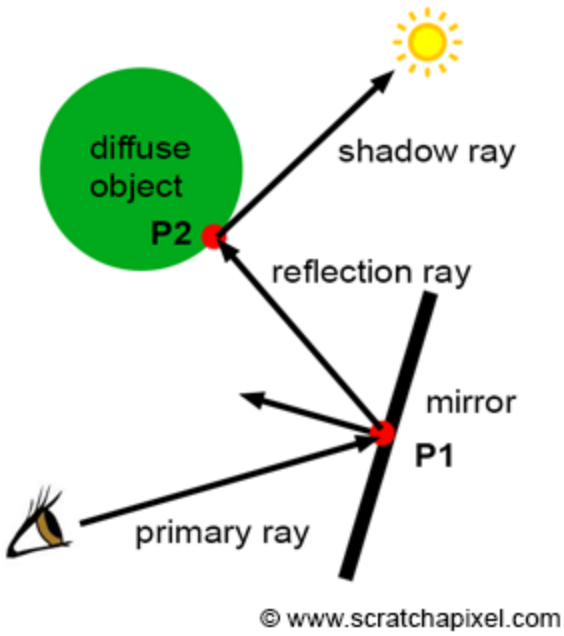
Summary

Our project will implement the ray-tracing graphics algorithm using CUDA on a single GPU and using OMP and AVX instructions on a multicore CPU architecture to achieve speedup. We will compare the difference between the two implementations and against a single-threaded naive benchmark implementation reference from another source.

Background

We will be accelerating a naive ray-tracing algorithm. The ray-tracing algorithm is a way to create a 2D image from a 3D scene given a camera angle, image size, and light source. The algorithm is extremely compute intensive compared to rasterization when it comes to shading objects in the space onto the image, but has the benefit of being able to generate shading and shadows, which creates more realistic looking images.

The idea behind the algorithm is to simulate light rays from a single point entering a scene. Whenever the light ray hits an object in the scene, a complex calculation is done to figure out what angle the light ray will bounce off at (reflection), whether a ray changes its direction once it contacts an object (refraction), whether a ray contacts a light source (shadow rays), and other complex characteristics of real light rays interacting with matter. The computation spawns new rays at each intersection based on the result of the computation, and the new rays are bounced into the scene. This process is repeatedly done until an upper bound. Then, each ray returns its color computation back to the ray that spawned it, where a complex opacity computation is done at the parent ray to create an updated color computation. This operation is recursively performed until we arrive at an initially spawned ray, which contains the color we will update the pixel that is mapped to the ray with.



Very roughly, the algorithm can be implemented recursively like below:

```
# Convert the image and camera angle into a set of initial rays
rays = empty list
for pixel in pixels:
    rays.add(makeRayFromPixel(pixel, cameraAngle))

# Simulate ray behavior until MAX_DETH collisions per ray
def simulateRay(ray, depth):
    for object in objects:
        collidedObject = None
        if collides(object, ray) and isCloser(object, collidedObject):
            collidedObject = object
            spawnedRays = simulateCollision
    if noCollisionsNeeded(ray) or depth == MAX_DEPTH:
        # This ray no longer requires further simulation.
        return computeColor(ray)
    else:
        colors = empty list
        for r in spawnedRays:
            colors.add(simulateRay(r, depth + 1))
        # Complex simulation blending the opacity
        # of all spawned rays and parent ray
        return computeColor(colors, ray)

colors = empty list
for ray in rays:
    colors.add(simulateRay(ray))

for pixel in pixels:
    pixel.update(getColorFromPixel(colors, pixel))
```

Non-recursively, it can be roughly implemented as below:

```
# Convert the image and camera angle into a set of initial rays
rays = empty list
for pixel in pixels:
    rays.add(makeRayFromPixel(pixel, cameraAngle))

# Simulate ray behavior until MAX_DETH collisions per ray
for _ in MAX_DEPTH:
    newRays = empty list
    for ray in rays:
        collidedObject = None
        for object in objects:
            if collides(object, ray) and isCloser(object, collidedObject):
                collidedObject = object
                spawnedRays = simulateCollision
        if noCollisionsNeeded(ray):
            # This ray sets to a default color, but no longer requires further simulation
            # We save it in our newRays so it is not discarded
            newRays.add(ray)
        else:
            newRays.add(spawnedRays)
    rays = newRays

# Compute the final pixel color by backtracking from the rays
# back to the original rays
for _ in MAX_DEPTH:
    parentRays = empty list
    for ray in rays:
        parentRay = ray.getParent()
        # Factor in the child rays color into the ray that spawned it
        parentRay.updateColor(ray.color)
        parentRays.add(parentRay)
    rays = parentRays

for ray in rays:
    pixel = ray.getMappedPixel()
    pixel.update(ray.color)
```

The recursive implementation can gain speedup from parallelizing the simulateRay operation per pixel with good load balancing. Meanwhile, a complex look up data structure could be created for the collision checks to be done in parallel or to minimize work per parallel simulateRay computation.

The non-recursive implementation can benefit from parallelism since each ray can be traced independently of all others per iteration. Furthermore, the ray and object collision is also parallelizable as a filter reduce operation if done carefully.

The Challenge

This problem is challenging as there will be a lot of memory accesses and if statement checks on many objects involved that will be difficult to parallelize in a SIMD architecture. One thing we must consider is how to reduce the number of pixels we need to check for our ray collisions as checking each object against each pixel will take unnecessarily long even if parallelized. Also, different materials, such as mirror-like vs. opaque objects, and how they compare in ray simulation will need to be taken into account. Another complicating factor is the algorithm itself, as we

are essentially working backwards from an output ray to arrive at our initial array, which is difficult to wrap our heads around and the recursive nature may complicate parallelism.

The challenges we may run into when considering the workload are that there will be problems with load balancing for each thread or processor as some rays may take longer to compute than others, in addition to the load changing through iterations of the algorithm depending on how many rays collide or scatter. Also, due to the abundance of rays needed to render a high quality image, there could be lots of cache misses associated with reading and writing the data. Some functions or aspects, such as computing scatter, will be hard to parallelize as they are recursive and so we will have to think about how to work around these nested dependencies.

Resources

We plan to use starter code from Scratchapixel, so that we have a correct but non-parallel implementation of ray-tracing to benchmark against and verify the correctness of our parallel implementations. The code we will be using is https://www.scratchapixel.com/code.php?id=3&origin=/lessons/3d-basic-rendering/introduction-to-ray-tracing.

We will be implementing our OMP and AVX ray tracer on the GHC machines and the supercomputing clusters. We will implement the CUDA ray tracer on the NVIDIA 2070 GPUs in the GHC clusters. We may also try the CUDA ray tracer implementation on a NVIDIA 3060Ti if time permits.

Goals and Deliverables

Plan to Achieve

- Fully working sequential, CUDA parallelized, and openMP/SIMD parallelized implementations of ray tracing
- Supports basic 3D sphere rendering from all angles
- Support light reflection and refraction on one specific material
- Speedup and performance analysis of the three implementations
    - Our primary goal is to analyze performance between a data parallel versus task parallel implementation on a task that could benefit from both.
- Demonstrate live ray-traced animation of a simple circular movement around a static model across the three implementations

Hope to Achieve

- Supports other basic 3D geometry rendering (cones, cubes, pyramids, etc.)
- Support rendering of multiple materials
- Parallelize new features on CUDA and openMP/SIMD and conduct a more in depth performance analysis
- Achieve further speedup: rendering target of 20 FPS on both parallel CPU and GPU implementations
- Demonstrate a live ray-traced animation of moving objects
- Demonstrate a live-ray-traced animation where the user can control camera angle

Platform Choice

We chose to use OMP and AVX in our CPU implementation because the starting code we will be using is highly recursive and not particularly data parallel but can benefit from parallelism across pixels. These properties mean that for parallel computing CPU platforms, OMP would be a better use case for parallelism over other platforms like MPI.

For the CUDA implementation, the CUDA API already provides a 3D coordinate system to allow for 3D space locality to be mapped into the GPU cores. Furthermore, given that GPUs are already accelerating graphics rendering, it seems fitting to attempt ray-tracing on CUDA cores. If we convert the starting code to a non-recursive implementation, similar to the pseudocode above, we can assign all GPU SMs small tasks to complete in iterations and massively parallelize ray simulating workloads such that each SM only simulates a small group of rays per iteration.

Schedule

Week 1: 11/7 - 11/13

- ➢ Research existing parallelization methods for path tracing using SIMD, CUDA, and OpenMP.
- ➢ Test initial sequential ray tracing algorithm for spheres and create some initial benchmark tests that will work with the initial algorithm.
- ➢ Begin parallelization implementation using CUDA.

Week 2: 11/14 - 11/20

- ➢ Finish parallelization implementation using CUDA.
- ➢ Begin parallelization implementation using OpenMP and SIMD intrinsics.

Week 3: 11/21 - 11/27

- ➢ Finish parallelization implementation using OpenMP and SIMD intrinsics.
- ➢ Measure speedup and compare performance for two parallel implementations against sequential and each other.
- ➢ Implement more features and functionality to sequential ray tracing algorithms noted in the **Hope to Achieve** section.
- ➢ Begin working on the milestone report.

Week 4: 11/28 - 12/4

- ➢ Finish milestone report. (*due* ***Wednesday, November 30th, 9:00am***)
- ➢ Parallelize code using CUDA and OpenMP + SIMD intrinsics on new features.

Week 5: 12/5 - 12/11

- ➢ Buffer time for weeks 1-4.
- ➢ Begin working on the poster, website, and writeup.

Week 6: 12/12 - 12/18

- ➢ Clean up code.
- ➢ Finish poster, website, and writeup. (*due* ***Saturday, December 17th, 11:59pm***)
- ➢ Present poster. (***Sunday, December 18th, 1:00-4:00pm***)