Wavefront Path Tracing
15-618 Project Milestone Report

Fulun Ma (fulunm) **Benran Hu (benranh)**

Updated Schedule

- Nov 15 - Nov 21: **(Completed)** Get started on the CPU path tracer codebase. Start to implement the megakernel CUDA version.
- Nov 21 - Nov 27: **(Completed)** Complete the megakernel version. Start implementing the wavefront version.
- Nov 28 - Dec 3 **(Ongoing, milestone due)**: Complete the wavefront version. Explore different pipelines, optimizations, and design choices.
- Dec 4 - Dec 7: Add wavefront design (pipelining) to BVH traversal, switch to "wider" BVH, and parallel ray-triangle intersection test.
- Dec 8 - Dec 11: Add wavefront design to ray generation and material/shading evaluation.
- Dec 12 - Dec 15: Test and profile different implementations on different scenes. Analyze the workload and performance. Prepare the final report and poster session.

Work Done So Far

We have completed implementing a CUDA version of an existing CPU path tracer using the naive megakernel approach, i.e., one CUDA thread is responsible for the whole procedure of ray generation, intersection tests, shading evaluation, ray extension, and image updating. All scene data (geometry, acceleration structures, lightings, materials, cameras) are stored in GPU global memory.

We also tried to directly use the CPU version of the BVH (i.e., a binary tree with large tree height and small leaf node size) in CUDA for ray-instance intersection, and a sequential ray-triangle intersection test in the leaf nodes. We measured some preliminary performance and compared to the CPU and the OptiX hardware accelerated version.

Goals and Deliverables

We are slightly behind our previous schedule, but the next direction is clear and the overall progress is smooth. Hence, we are still confident that we can achieve the following goals and present the listed deliverables.

As for the "nice to have" goals in case we have extra time, we might not be able to try SIMD implementation on CPU, or implement a heterogeneous version using both CPU and GPU, but we may still try to accelerate BVH construction using GPU, or including BVH refitting for animated scenes. We may also try to incorporate the hardware accelerated ray-triangle intersection test into our pipeline.

Goals Plan to Achieve

- Implement two path tracers in CUDA using wavefront design and megakernel design, respectively.
- Compare their performance and analyze how our wavefront design alleviate execution divergence, and how that affects the final performance. We target a speedup over 1.3x in simple scenes, and over 2x in more complex scenes.
- Test under multiple scenes with different configurations to analyze when wavefront path tracing benefits the most.

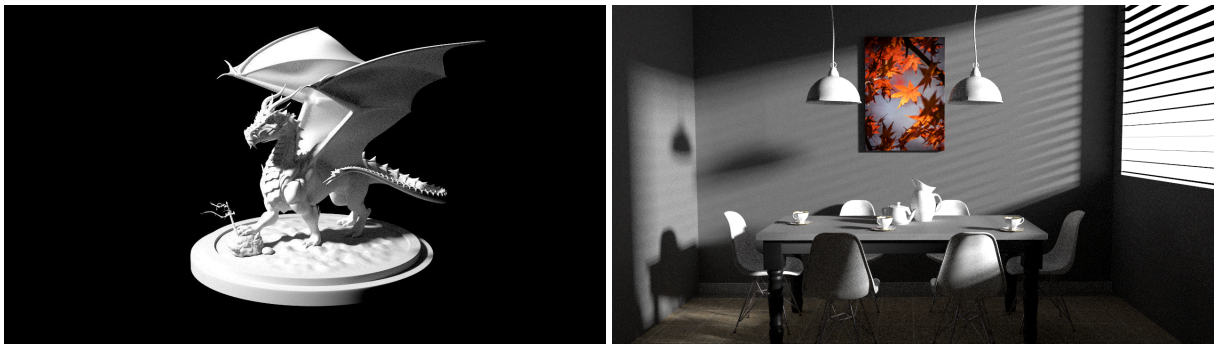
- Experiment with some basic ray sorting and work queueing methods and analyze and performance and tradeoffs.

Deliverables

- We may present an interactive demo to show the speedup of our design and how it varies with the scene. Our current naive implementation already achieves interactive framerates on some simple scenes.
- Speedup plots and other profiling metrics (warp utilization, cache hit rate, memory usage, etc.) of our wavefront design compared to the megakernel design on GPU, under different scene configurations.
- Performance comparison and analysis of different design choices mentioned in the challenges section.

Preliminary Results

We ran our current megakernel version of the path tracer on several different scenes and compare its performance to the CPU version and the implementation using OptiX (thus NVIDIA's RT cores) hardware accelerated intersection tests. Note that the OptiX version uses the same megekernel design.



a) Dragon b) The Breakfast Room

c) Country Kitchen d) Landscape

Figure 1: **Test Scenes Used in Prelinimary Results.** Rendered using our codebase with 512 spp. The first three scenes come from Benedikt Bitterli's rendering resources. The last is from PBRT-v4's repo.

We use scenes of different geometry complexity and shading complexity (variety of materials and lightings). The *dragon* scene contains a single object with simple material, whereas the *landscape* contains 23,241 plant models and 3.1B triangles. The other two scenes sit inbetween and feature different sets of materials.

As this is simply the preliminary results to get a sense of the performance bottleneck and optimization direction, we only time the total rendering time for rendering 1 sample-per-pixel (spp) for a 1280x720 image, and average the rendering time over 64 samples.

Time (s)	Dragon	The Breakfast Room	Country Kitchen	Landscape
CPU	0.0811	0.4537	0.6665	5.7601
CUDA	0.0180	0.1346	0.1292	2.7245
OptiX	0.0076	0.0203	0.0243	0.0888

Table 1: **The rendering time of 1 spp in different scenes.** CUDA refers to our current megakernel implementation.

From Table 1, we can see that even with the naive megakernel design, the CUDA implementation can still achieves 2x-5x speedup compared to the CPU implementation. However, considering that path tracing is massively parallelizable, this speedup is clearly unsatisfactory, and the main reason is the extremely high divergence occured during BVH traversal, ray extension, and shading evaluation.

Comparing the performance of CUDA and of OptiX implementation, we found that the performance of ray-scene intersection test is crucial for the overall rendering performance, especially in scenes with highly complex geometries. Therefore, the first optimization we plan to implement for our wavefront version is to make the BVH traversal a separate pipeline stage. We need to make the BVH "wider" by increasing the fan-out width and decreasing the tree height, which reduces the costly pointer chasing and allows us to parallelize ray-box/ray-triangle intersection test over all threads within a group. We do not expect that this can reach the same speedup as the OptiX version, but it will be a significant optimization to consider first.

Concerning Issues

We do not have any outstanding concerns at this point. One potential problem is how we acquire low-level profiling results such as warp utilization and cache hit rate, which might occur at a later stage of the project. The NVIDIA CUDA Profiling Tools and the Visual Profiler might suffice if we can make them work with our project.