Wavefront Path Tracing
15-618 Project Proposal

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

Summary
We are going to implement a parallel path tracer using a specific optimization technique named wavefront path tracing in CUDA. We will compare its performance to simpler parallel implementation using megakernels on CPU and GPU, and analyze how ray path and shader divergence affect parallelization performance. Project website.

Background
Path tracing, as one branch of the ray tracing methods, can potentially benefit from parallelization greatly by distributing the rays to different CPU threads using OpenMP or MPI. However, on GPUs or CPUs with SIMD support, naively assigning rays to each CUDA thread or SIMD lane can result in very poor speedup, given the highly divergent workload among rays.

A simplistic approach to parallelizing path tracing with CUDA or SIMD involves creating "megakernels" or "ray packets" that process an entire light path from beginning to end. However, this method faces challenges as rays in different threads might end at various points, encounter diverse object types, or interact with objects using different shaders. This can cause significant divergence within GPU warps since each thread might execute distinct instructions, leading to idle threads and low warp utilization. Additionally, when accessing acceleration structures like Bounding Volume Hierarchies (BVHs), the divergence in ray paths can adversely affect cache coherence.

In contrast, wavefront path tracing divides the process into several smaller kernels. Each kernel either advances the path to the next kernel, or terminates it. This allows opportunities to sort the rays by their origins, directions, hit surface types, or other properties, and compact the thread blocks when some rays terminate early, so that we can achieve better execution coherence. Smaller kernels also allow pipelining (streaming) and finer-scale work queueing, where we may get better work balancing. The idea is described in [1] in detail.

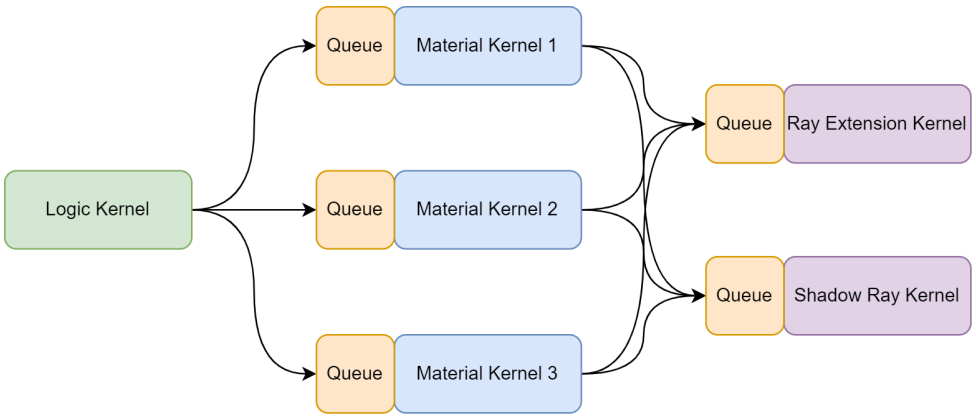

Figure 1: Diagram of a single wave in wavefront path tracing.

Figure 1 is an illustration of one possible implemention. We may divide a "wave" in a ray path into three stages. In the logic stage we accumulate the shading result, perform sampling, determine whether the ray should be terminated, and determine the material type of the hit location. In the material stage we

evaluate the material, and in the ray cast stage we extend the rays, which will form the next wave. We can use a work queue and maintain a pool of working threads for each kernel.

Challenges

As mentioned above, parallelizing path tracing with CUDA or SIMD is trivial, but getting maximum speedup is hard. Essentially, path tracing in practice is *not* a nice data parallel program, and there can be high divergence if we directly map each ray to a CUDA thread. This is the main challenge for our project, and also the main motivation of using a wavefront design.

While [1] gives one possible formulation of wavefront path tracing, we would like to note that the design space for this method is huge, and there is a lot of performance tradeoffs depending on the actual workload charateristics. We will describe some of the major challenges.

Execution Divergence

The goal of the wavefront design is to reduce divergence in GPU warps or CPU SIMD groups, however, the design *alone* does not resolve all the divergence issues. Within each kernel divergent execution and memory access might still exist, if we do not sort or reorder the rays. We also need to make trade-offs between divergence and kernel granularity, as dividing kernels into very small pieces helps with coherence but adds overhead for communication and orchestration.

Memory Access Locality

As mentioned, divergent rays can lead to low spatial locality in BVH traversing, which can quickly overwhelm GPU's thread local caches and global memory bandwidth. This implies that ray sorting strategies may be needed to maximize the locality within each thread block. The spatial locality of transferring data from stage to stage should be exploited as well.

Space and Communication Overhead

Wavefront design requires us to place all the data produced and consumed in each stage into global buffers. There can be a high space requirement for the buffers, and reading/writing them in each stage can be very time-consuming, lowering arithmetic intensity. Nonetheless these redundant data tranfers does not exist in megakernel design. We therefore need to carefully divide the stages and kernels to minimize the space and communication overhead.

Synchronization and Orchestration Overhead

There are multiple ways to implement and orchestrate the wavefront design. Apart from the one shown in Figure 1 using work queues, we may also launch the kernels in each stage only after all work has been generated, which is simpler and reduces overhead in work fetching, but has higher overhead in launching threads. Alternatively, we can pipeline all the stages, or even stream data between them, which requires us to think about how to synchronize stages.

If we decide to maintain a threadpool for each kernel, we then need to care about dynamically adjust the threadpool sizes based on the load. Additionally, there might be multiple consumers and producers writing to these buffers concurrently if we do streaming, thus we need proper synchronization for the queues and buffers.

Kernel and Pipeline Organization

We need to determine the granularity of kernels and pipeline stages to make sure the gain outweighs the overhead. For instance, we do not want to create two separate kernels and queues for two shaders that are largely the same. Fine granularity of kernels may yield higher coherence, but also suffers from high communication, space, and orchestration overhead, and low arithmetic intensity. This requires us to carefully analyze the path tracing scenes we plan to render.

Resources

We plan to use the GHC machines with RTX 2080Ti as our main platform of development and experimentation. We will likely start with an existing pure CPU-based path tracing codebase (e.g., Scotty3D, Nori, and Yocto/GL), and adapt it to CUDA, so that we can reuse the scene I/O and user inferfaces, and spend most of our time implementing the wavefront pipeline. We will use the design in [1] and [2] as a reference, but we almost surely will explore and choose different designs and implementations.

Goals and Deliverables

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, based on the results in [1].
- 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.

In case of slow progress, we stick with the first three goals.

Goals Hope to Achieve

- Explore more complex design choices like streaming, threadpool, etc.
- Implement wavefront path tracing also using SIMD on CPU and compare its performance to the megakernel version using SIMD, and without using SIMD.
- Accelerate BVH construction and/or refitting using GPU.
- Extend the path tracer to utilize heterogeneous architectures, e.g., offloading certain highly divergent rays to CPU, or using the RTX cores in the GPU for hardware acceleration.

Deliverables

- We may present an interactive demo to show the speedup of our design and how it varies with the scene, if our final implementation can achieve interactive framerates.
- 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.

Platform Choice

We choose CUDA as the problem of divergence only exists in data parallel programming models, which the main rationale of wavefront design. It makes little sense to apply the same design to shared address space or message passing models, like using OpenMP or MPI on CPUs.

Although wavefront design also benefits SIMD execution on CPU, the degree of parallelism is much lower than GPU, thus the issue is less pronounced. Also, implementing ray casting and shading using SIMD intrinsics or ISPC might be more complex than implementing them in CUDA.

Overall, even though normal path tracing has high divergence and does not match well with the data parallel programming model, it is naturally parallelizable and can still enjoy the massive parallelism of GPU. In practice, a good wavefront implementation of path tracing on GPU is also much faster than a CPU version using OpenMP, hence it is interesting to see how we can adapt the workload to better suit the CUDA programming model.

Schedule

- Nov 15 - Nov 21: Get started on the CPU path tracer codebase. Start to implement the megakernel CUDA version.
- Nov 21 - Nov 27: Complete the megakernel version. Start implementing the wavefront version.
- Nov 28 - Dec 3 (milestone due): Complete the wavefront version. Explore different pipelines, optimizations, and design choices.
- Dec 4 - Dec 10: Test and profile different implementations on different scenes. Analyze the workload and performance.
- Dec 11 - Dec 15: Prepare the final report and poster session.

References

[1] S. Laine, T. Karras, and T. Aila, "Megakernels considered harmful: Wavefront path tracing on GPUs", *Proceedings of the 5th High-Performance Graphics Conference*, pp. 137–143, 2013.

[2] A. Keller *et al.*, "The iray light transport simulation and rendering system", *ACM SIGGRAPH 2017 Talks*, pp. 1–2, 2017.