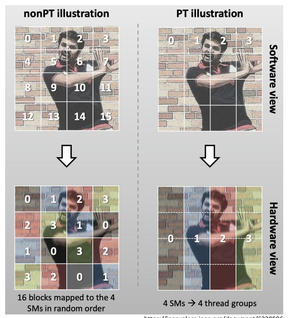
Analysis of Global Queues for Recursive Methods on GPUs

Project 16: Justin Kalloor and Cheng Cao

Problem

- Traditional recursive programs can cause huge performance issues due to cache divergence and control flow divergence
- Non-Nvidia GPUs will need to manually write a stack themselves, or convert their algorithms to an iterative one.
- There is no simple way to to specify recursion in a DSL (Domain Specific Language) for GPU programming, nevertheless create something performant.

Background

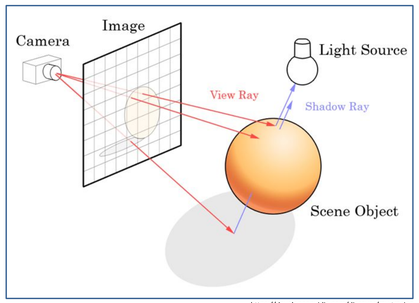


Persistent Threads

- Existing techniques to override scheduling use the persistent thread model.
- Not officially supported in any ways, relying on a few assumptions of GPU execution model.
- These programs run a while loop using a queue to assign work.

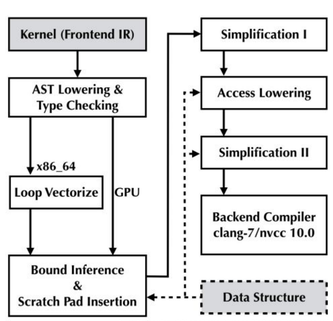
Ray Tracing

- Unique recursive problem where every pixel can be run simultaneously
- However, there is massive divergence within threads
- GPUs can not achieve full potential due to divergence and register pressure



Taichi

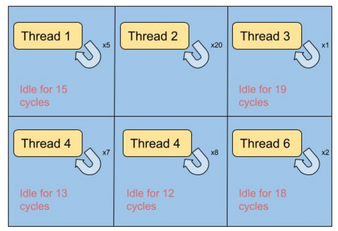
- Taichi is a DSL targeted towards allowing easy to use GPU programming.
- Loops are auto parallelized through the taichi compiler, which then connects tasks to the GPU while adding in various optimizations along the way.
- Due to the issues listed above, Taichi does not currently support recursion.

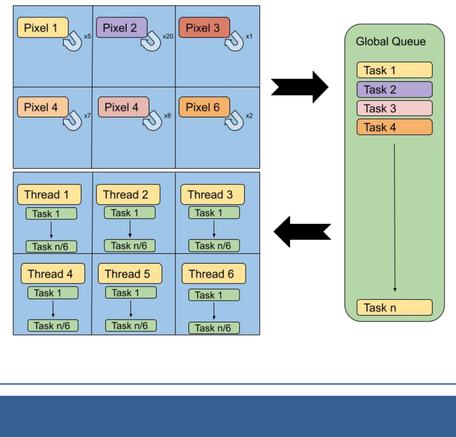


Hypothesis

Naive Ray Tracing

- Assign a HW thread to each pixel
- Each pixel will have varying recursion depths based on collisions with objects
- Divergence causes inefficiency in thread





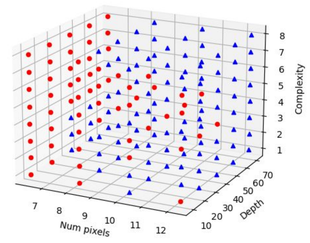
Global Queue

- Throw each task associated with a pixel into a global queue
- Allocate tasks from global queue evenly among HW threads

Next Steps

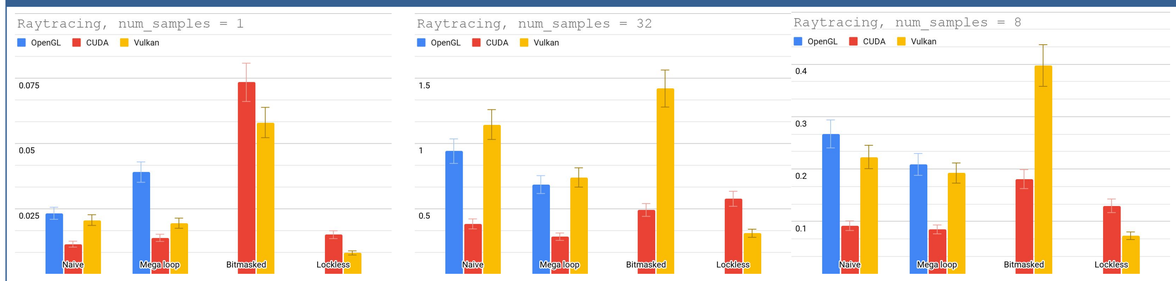
Because we observed very weird performance scaling on different compute backends, while also observing very poor performance with wavefront style raytracing, more investigation is needed to truly understand the reasons of such observations.

We have constructed a variable performance model where we can vary the number of pixels (maximum throughput), the max ray depth (control flow divergence), and the complexity of the hit function (register pressure & instruction complexity). From this, we can run our various queue algorithms as well as a mega kernel implementation to find performance. With these datapoints, a simple k Nearest Neighbors model can be selected to run any corresponding benchmark.

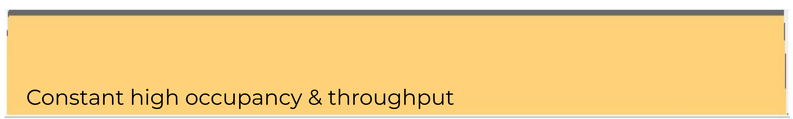


With these data and correlations with more lower level hardware performance counters, we should be able to find out the reasons of our performance observations, and derive a model where we can select the best algorithm for other workloads. Since Taichi is an GPU focused DSL, it should be possible to do runtime analysis and optimizations to achieve higher performance.

Analysis & Findings

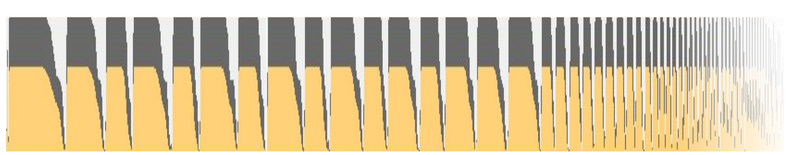


1. Host side queues (wavefront processing) has significant host-device communication and synchronization delays.
2. Device side queues has potential atomics contention issues. Due to the lack of clearly defined execution model, and the lack of scheduling APIs make it difficult to write efficient queues.
3. It is very hard to tune persistent thread parameters (such as thread count). Systematic approach is needed to avoid manual tuning / trial and error.



Lockless Queue (Vulkan) - An atomic SPSC queue paired with persistent thread model. We can see the GPU resources are almost fully utilized the entire time. However this model does not work as well as our hypothesis on CUDA. We are still trying to figure out why, but our suspicion is on heavier atomics traffic and heavy spin-waiting for threads. (CUDA might have treated spin waiting as useful work and therefore schedule less time for actual work)

Mega loop (Vulkan) - Each thread will move on to the next sample without waiting. In this setup, each thread iterate through all the different bounces of each ray, and also move to the next sample if possible. However we can observe that some pixels are no longer active after all samples are done, causing divergence and lowered occupancy. However we suspect that CUDA schedules the threads more granularly than Vulkan, and thus mega loop appears to be the fastest on CUDA while not that case in Vulkan.



Bitmasked (Vulkan) - Wavefront, the host will launch more threads if wavefront is not empty. This method try to reduce divergence by launching a single wave front a time, and a reduction step will be run to collect the still active pixels, and launch another wavefront. However we can see the overhead of draining the GPU pipeline due to host-device communication is too large for this to actually improve occupancy.