15-418 Final Project Proposal

Ray Tracing Parallelization

Jonathan Ke & Kavya Tummalapalli

Summary

Our project implements the ray-tracing graphics algorithm using CUDA on a single 2060 Nvidia GPU and using OMP and AVX instructions on a multicore CPU architecture in the GHC cluster to achieve speedup. We compare the differences between the two implementations and against a single-threaded naive benchmark implementation reference from another source.

Background

The algorithm we are parallelizing is drawn from an introductory ray tracing algorithm found from Scratchapixel.com. The algorithm cannot scale for higher resolutions or more objects in the scene and is only focused on simulating simple reflections, refractions, and diffuse spheres in the scene.

The basic algorithm in the source code doesn't attempt to be overly complex. For each pixel in the screen, we compute the ray going through the point from some input origin. Each ray is passed into a *trace* function that does a couple actions: check if the ray conflicts with any sphere in the scene, trace the reflected ray if it exists, trace the refracted ray if it exists, compute the diffuse color if no reflection or refraction occurs, and return a surface color based on the simulation of the ray. The reflection and refraction computations are recursively computed using *trace* until a maximum depth. Notice that diffuse objects and detecting no sphere collisions means we can stop recursing. Please refer to https://www.scratchapixel.com/code.php?id=3&origin=/lessons/3d-basic-rendering/introduction-to-ray-tracing for the exact starting code.

The simulation algorithm can roughly be pseudo-coded as shown below.

```
# 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 noCollision(ray) or depth == MAX_DEPTH:
        # This ray no longer requires further simulation.
        return computeColor(ray)
    else if collidedObject is diffuse:
        intersectPoint = simulateCollision()
        finalColor = 0
        for object in objects:
            if object is light source:
                for object2 in objects:
                    if object2 blocks intersectPoint from object:
                        continue
                finalColor += computeColor(object, collidedObject, ray)
        return finalColor
    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)
```

Suppose our input resolution is L x W and we have N spheres in the scene. We also bound our maximum depth to k. We can analyze the total runtime of this code in parts.

To detect a sphere collision, the *collide* functionality above, the naive algorithm loops across all spheres and detects if it intersects the ray, represented as an origin and direction. Collision is a O(1) computation per ray per object, so we expect O(N) runtime per ray to check collisions.

To compute a diffusion color, we need to loop through all objects, tracing only from any light emitting objects, toward the intersection point of the current ray and the diffuse object, then check if any other object blocks this light ray. The block checking per diffuse ray is O(N) runtime, while looping through all objects filtering for light sources is O(N), for a total runtime of O(NxN) runtime when computing a diffuse color result.

The total number of rays we may need to simulate is dependent on the depth k. For any initial ray, it and all of its children can generate a new reflecting, and refracting ray, leading to a total of O(2^k) rays per initial ray. Every final ray could lead to a diffuse computation, so given we had L x W initial rays, we have O(2^k x L x W x N^2) total runtime for this algorithm to generate a single frame of our scene.

The original algorithm is implemented recursively, meaning no real data structures other than the array of spheres in the scene are needed to implement the ray tracing. Each initial ray can be traced independent of any other ray, but it is difficult to predict how deep each ray will need to recurse or if it hits the depth k. Each ray can bounce and spawn 2 more rays, and the initial ray cannot complete its computation until the spawned rays are computed, so per initial ray, we need to maintain bookkeeping about that ray's data and children until all the children rays don't bounce, hit a diffuse object, or are at max depth k. This book keeping is implicitly done through the stack within the naive implementation.

Within the naive implementation, the bulk of the workload is focused on the collision detection between rays and spheres in the scene. This work is always performed per ray when trying to discern if it should bounce or diffuse and if in the diffuse case, an additional set of collision detections need to be performed on all light vectors from the collision point. The collision detection is a very data parallel operation, as for any set of rays, we can repeatedly run the collision check for each sphere in the scene. However, we then have a divergence to task parallelism after collision checking, rays can either diffuse, reflect/refract, or go into the background. The behavior of each ray is vastly different after each collision check step for a group of rays.

Heuristically, we expect rays that are spatially local to each other exhibit very similar behavior both in the conflict checking stage and in the divergence of ray behavior after checking for collisions. This is expected since rays from the same origin and going a similar direction will likely collide in the same area. Following this logic, however, after a couple of bounces, an initial spatially local bundle of rays will have more and more divergent behavior as the reflected and refracted rays will scatter in different directions.

For most scenes, we expect a significant number of our initial set of rays to either diffuse or not collide with any sphere since it is unlikely that in all directions we will encounter a reflecting/refracting object. Meanwhile, some rays could reflect and refract until the maximum ray depth. In our design, nearly all spheres in the scene both reflect and refract, which means we need to optimize well for this case.

Approach

CUDA:

The initial design of the CUDA implementation followed the pseudocode presented in the proposal and copied 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)
```

We will map each ray to a thread and do a scan/reduce across all rays for each iteration up to the maximum depth. The key idea here is to ensure rays that do not spawn a refraction and reflection rays don't lead to generating new excess work and that each iteration/layer of rays can be computed in a massively data parallel fashion.

Once we have recursed to the maximum depth, we iterate from the last layer of rays back to the initial layer of rays to compute the final surface colors that are part of the frame. The initial implementation provided significant speedup compared to the naive implementation since the naive implementation was unicore. A major optimization on memory usage was done immediately, we bound the maximum number of objects allowed in the scene, we can employ constant memory partitions to preload all spheres and mitigate the cost of loading spheres into memory during collision checks. However, the initial implementation had a couple significant shortfalls that should be optimized.

- The initial implementation dynamically allocates and frees memory on the fly since each scene has very different and unpredictable memory usage.
- The initial implementation makes no usage of shared memory.
- The initial implementation does not filter different surface color computations to increase the benefit of data parallelism.
- The initial implementation does not minimize communication costs between stages of the implementation.
- The initial implementation does not precompute reusable information.

Further improvements were made to fix some of the problems above. The first optimization was to minimize dynamic allocations during rendering. To accomplish this, we allocate the final image space during startup and create a group of buffers scaled with the maximum ray trace depth. Each buffer is dynamically allocated the first time the render function is called and then only ever reallocated if the buffer is not big enough to store a new workload. This is a very effective optimization since between render frames of the same scene, the memory utilization per layer of rays is relatively similar, so one single allocation can be reused multiple times before a new allocation needs to be made. We try to predict overall memory usage by the scene by over allocating by a factor of 1.3x the currently needed space aligned to a granularity of 4096 ray data structures each time we are required to allocate to capture needing to reallocate when only a slight increase in memory usage is needed. This led to a 10% to 20% speed up on test cases relative to the initial CUDA implementation.

The next optimization was to precompute the set of light emitting objects and load them separately into constant memory. This allows us to save a factor of N spheres during the diffuse computation since we know beforehand which spheres will generate a light ray. This allowed scenes with more spheres to perform a little faster since we do not duplicate computing which spheres emit light during each diffuse computation.

At this point, fine-grained analysis of what steps of the algorithm take the most time and it was revealed that the movement of data between ray tracing iterations to consolidate the set of new reflected and refracted rays was taking up twice the amount of time even the collision checking was taking on the worst case.

An additional implementation was attempted here to minimize the communication overhead to the maximum, we implemented a CUDA ray tracer that traced all rays using only shared memory and only wrote the resulting surface

colors to global memory. The results of doing this turned out to be worse performing than the previous implementation because of limitations to the shared memory count upper bounded the number of threads we could allocate to below 10 threads, which led to the overall implementation to suffer significantly from lacking enough warps to fully utilize each core. The results of that experiment are reported later. Our original implementation scheduled 256 threads per block, which creates enough partitioned work for the GPU scheduler to fully utilize CUDA cores.

We then moved our attention to minimizing the amount of memory used to represent a ray. The data needed to track a ray are the ray origin, ray direction, surface color, a pointer to the colliding object, a pointer to the spawned reflecting and refracting children rays, the number of children spawned, and the point of hitting, normal of the hitting point, and some other miscellaneous computation data. This structure was significantly cut down by trading off some extra compute and recalculating certain values on the fly. By doing so, all the miscellaneous data like the hit point and the normal of the hit point could be removed and replaced with a single float, the intersecting distance along the direction ray. Furthermore, some data values could be shrunk in size, like the number of children spawned could be represented as only 8 bits. By doing these optimizations, the approximate speedup is 1.7 relative to before, as the total amount of data transferred between stages from global memory is nearly halved.

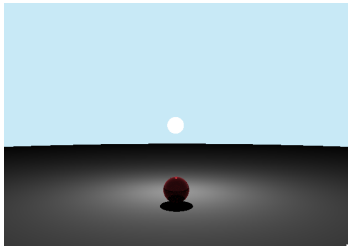
OpenMP:
The initial desi

Results

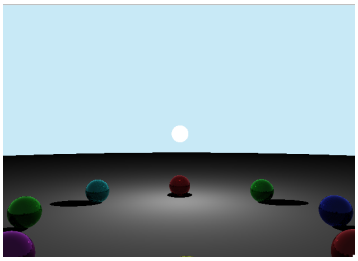
Benchmarking

We use thirteen different self-generated benchmarks for testing and collecting data.

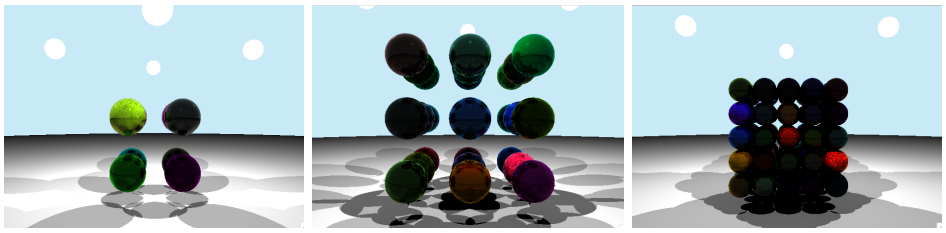
1. single.txt - A simple scene involving a surface, a light source, and a reflective and transparent sphere.



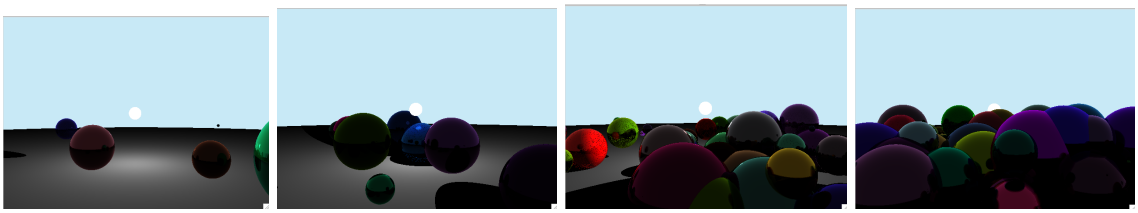
2. ring.txt - A ring of 8 spheres with random transparency and reflective properties spaced equally on a surface, and a light source.



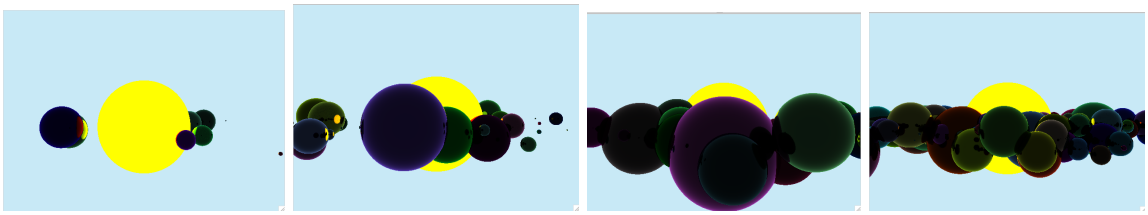
3. cube_x.txt - Cubes composed of x number of randomly configured equal radius circles in each dimension, four lights, and a surface.



4. random_x.txt - A single light source and x number of randomly generated spheres on a surface.



5. solarsystem_x.txt - A single light source and x number of randomly generated spheres encircling the light source.



Each scene is rendered for 90 frames with each frame rendering the camera angle moving 1 degree over in counterclockwise rotation and the total time is logged for rendering all 90 frames. This benchmarking allows us to render the same scene from multiple angles and get a more holistic and normalized view of rendering performance in real-time. We chose to have different numbers of spheres in our sample scenes to provide different scene workload sizes.

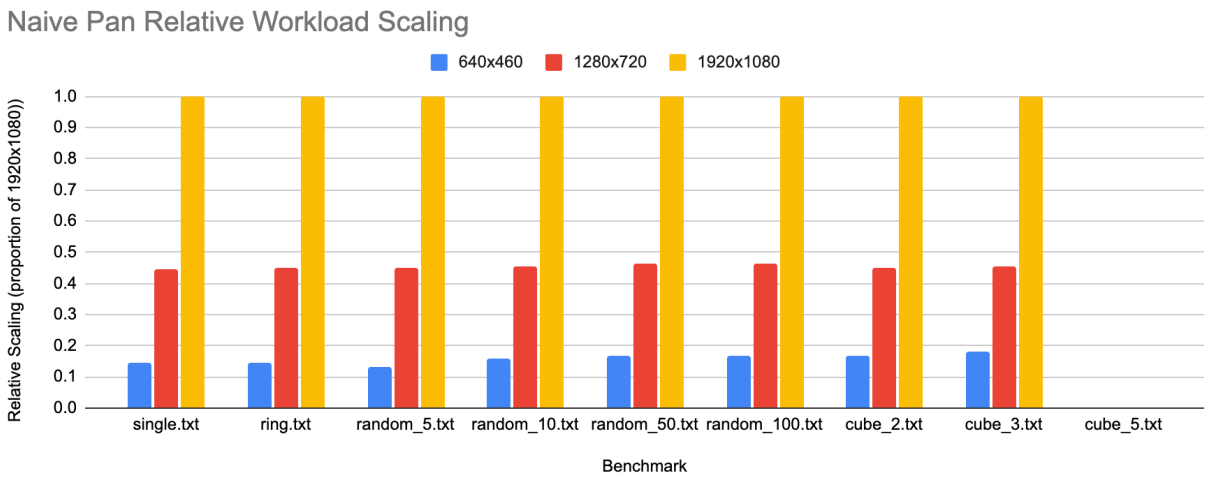
Our scenes are also rendered in 3 different screen sizes: 640x460p, 1280x720p, and 1920x1080p to provide data on how the ray tracer implementations scale with increasing screen size workloads.

Finally, we generate two kinds of movements within our benchmarking, the first is a panning movement, where the camera circles around all the objects in the scene and the second is placing the camera at the origin and rotating the scene around the camera. We use two motions here to because having the camera point to the center of all the objects tends to lead to more ray bounces and having the camera encircle the objects creates recognizable scenes and tends to lead to less ray bouncing as most rays will either hit the floor or does not collide with the scene, which allows us to simulate two different workloads.

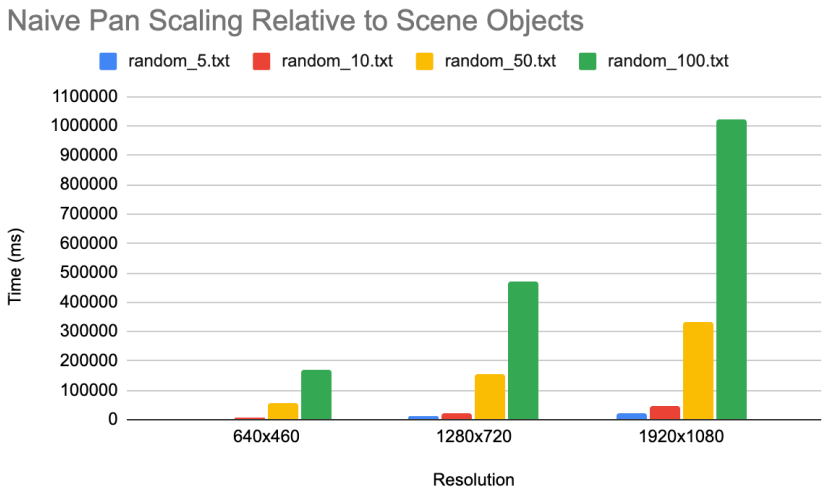
Naive Implementation Performance

The naive single-threaded implementation scales with different runtimes as the benchmarks increase in all three dimensions: frame size, scene size, and toggling between spinning and panning.

When we scale the frame dimensions, from (a) 640x460p to (b) 1280x720p (~3x the number of pixels compared to (a)) to (c) 1920x1080p (~7x the number of pixels compared to (a)), we see universally that the increase in relative runtime is proportionate to the increase in number of pixels, which aligns with our expectations.

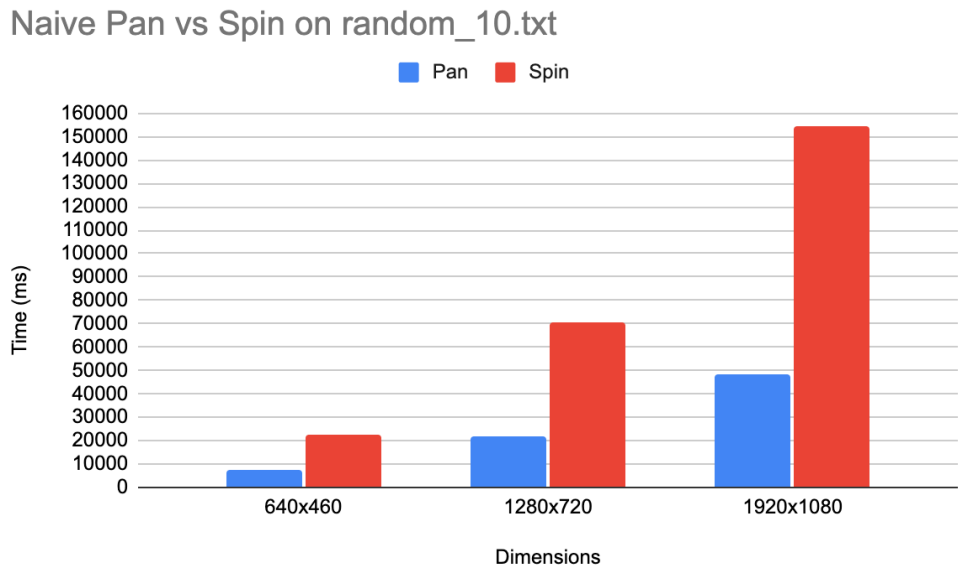


If we scale the number of objects, we see a non-linear scaling in runtime. Between 50 objects and 100 objects, we see a 3x jump in render time when the number of objects only approximately doubles.



This is expected since increasing the number of objects increases the number of rays that will be collide with objects multiple times, where each bounce will yield two new rays, so an exponential number of rays will be simulated relative to the number of objects in the scene.

We can also analyze how the naive sequential solution scales when panning versus spinning.

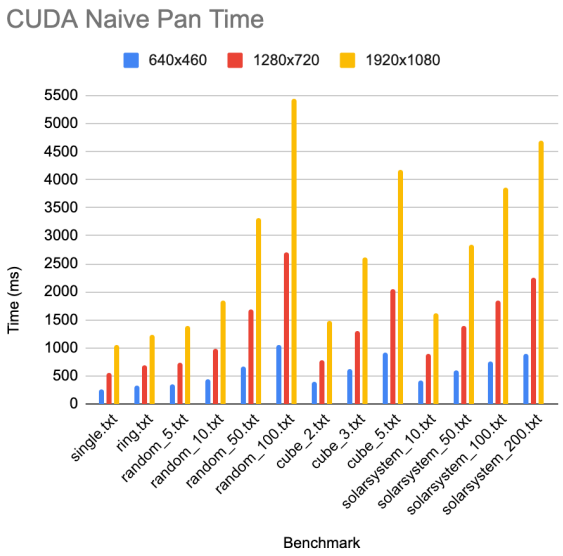


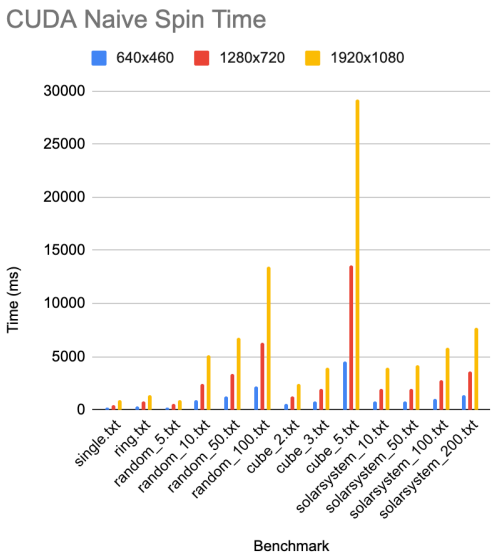
For random_10.txt, if we spin from the center, we take up significantly more time than if we panned around the scene. Furthermore, looking at the ratio between panning render time and spinning render time for each dimension, we see there is no constant ratio. Larger frames lead to disproportionately longer spinning times compared to panning times. This can be explained by the higher chance a ray will bounce in the spin case versus the pan case. With more collisions, exponentially more rays are recursively created and simulated, leading to a blow up of work for the subset of rays that bounce around the scene, which explains why the two rotation schemes don't scale similarly. For example, we expect the spinning case to have a larger number of rays that bounce off of the object directly in front of it, leading to more rays scattering versus diffusing with the scattered rays taking up exponentially more work. With higher resolution, even more rays get scattered, meaning allowing the exponentially scaling work to be more dominant in the total runtime according to Ahmdahl's Law.

CUDA

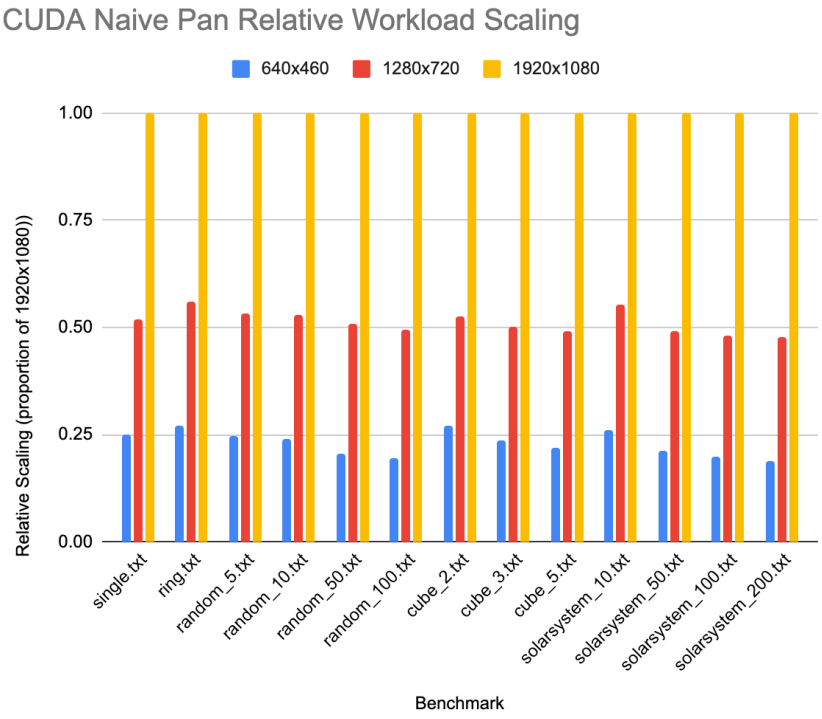
We compare the initial implementation using CUDA and the final implementation using CUDA to demonstrate the GPU parallelism specific optimizations performed since a GPU is a vastly different computing system compared to a single threaded CPU implementation.

The CUDA naive pan and spin had the following runtimes.

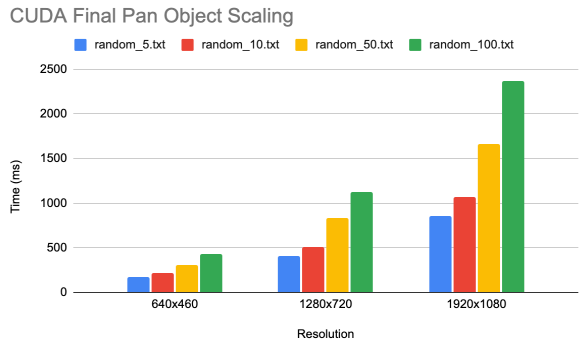


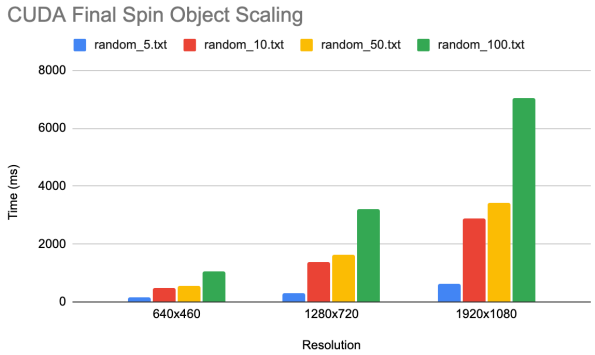


Observe as with the single-threaded case, that naive spinning takes significantly longer than panning, which is again attributed to the disproportionately more time bouncing rays take up relative to diffused rays or rays that don't collide. We also observe a similar scaling with increasing resolutions. Doubling the resolution doubles the work time.



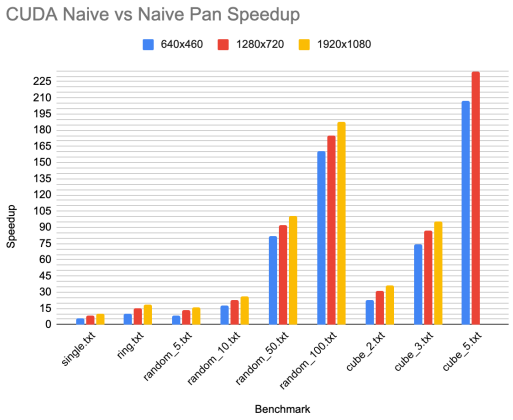
When increasing the number of objects in the scene, we see different scaling behavior depending on spin versus pan.

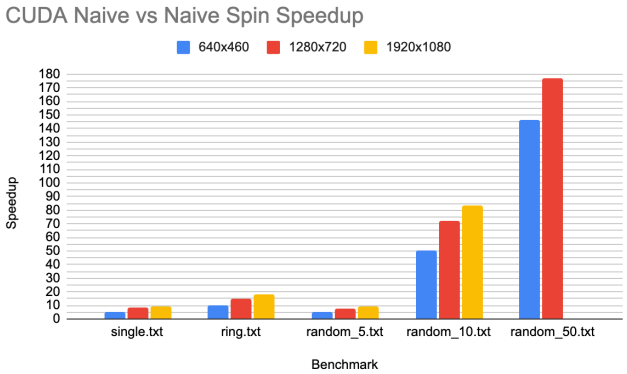




For panning, we observe much better scaling behavior relative to the single-threaded case. This can be attributed to optimizing all objects to stay in constant memory and precomputing the emitting objects array from the set of all objects. Furthermore, our CUDA implementation actually becomes bandwidth limited on our benchmark even with over 200 objects in the scene.

These statistics, however, do not take away from the overall extremely high speedup the naive implementation has over a single threaded CPU implementation.

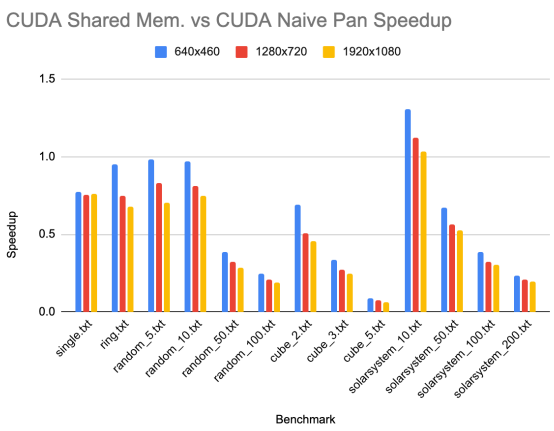


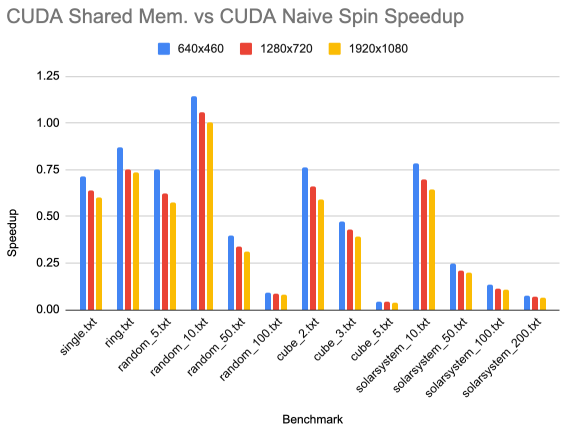


We can observe speedups from 5x to 220x relative to the single-threaded CPU implementation. We can also observe we get better speedup the higher the resolution and the more objects that are within the scene. This is an indicator the current CUDA solution is likely has relatively low arithmetic intensity and high overhead setup time as we have not hit a limit in the parallelism we are achieving on the GPU.

Given the low arithmetic intensity problem, an implementation using only shared memory was created to attempt to minimize memory overhead. This implementation could only assign 8 threads per SM due to how much memory each

ray computation required, meaning the amount of shared memory per thread block was only enough for 8 threads.

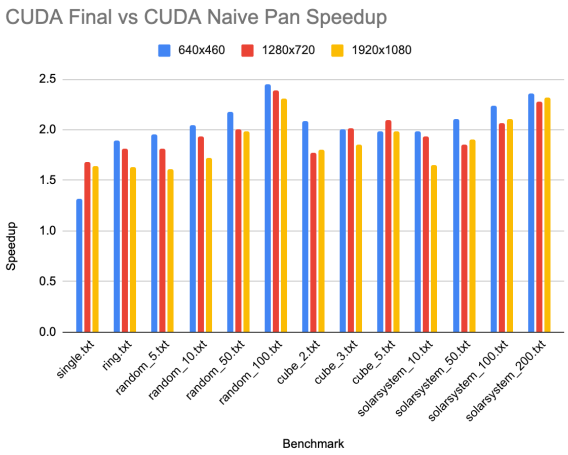


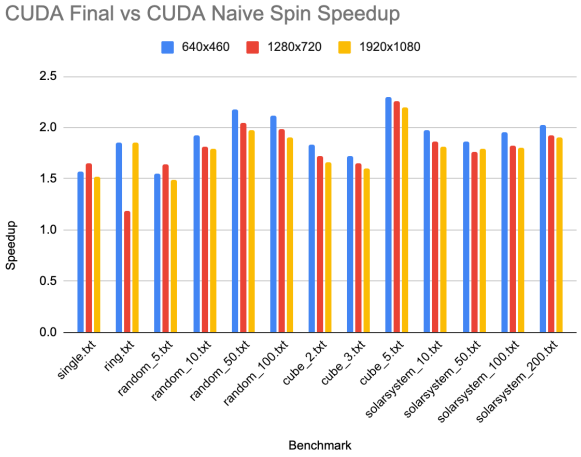


Observe we are getting hardly any speedup in any of our benchmarks, and most of the time we observe a massive decrease in performance. We can attribute these to a couple factors.

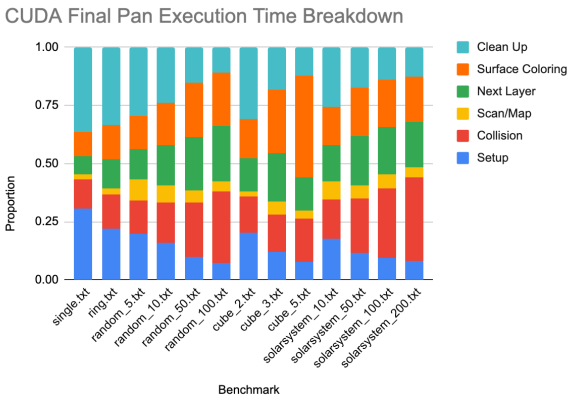
- Scheduling only 8 threads per block massively decreases the GPU scheduler's ability to pipeline threads.
- This one off attempt to implement everything in shared memory does not allow for early termination if all rays diffuse and assumes all rays will probably traverse to maximum depth. This means a lot more shared memory 32 ray data slots is required per ray and a lot of excess work is performed for rays that will have massively divergent behavior after one collision.
- The compute per warp is very non-data parallel, compounding an already long kernel execution with more branching.
- Some memory optimizations were not made, like reducing the required amount of data a ray uses during simulation.

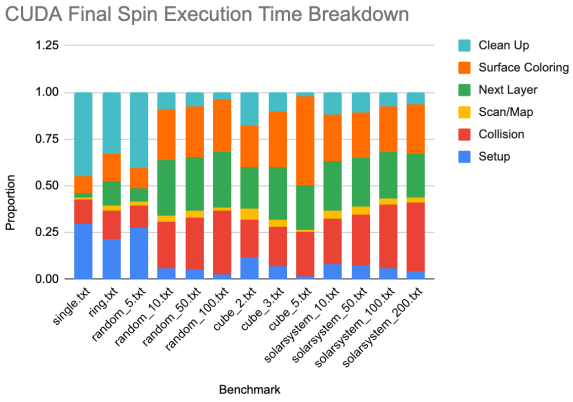
Given the observations above, we return to the old implementation and try to reduce memory usage as much as possible by shrinking data structures, specifically the amount of data needed to represent a ray. Doing so yielded good speedup, as shown below.





Minimizing memory utilization yielded anywhere from 1.2x to 2.5x speedup across all benchmarks, most notably, for benchmarks with more objects, we observe at least 1.5x speedup. Clearly, we are bandwidth limited in the CUDA implementation. Timing our code demonstrates this.



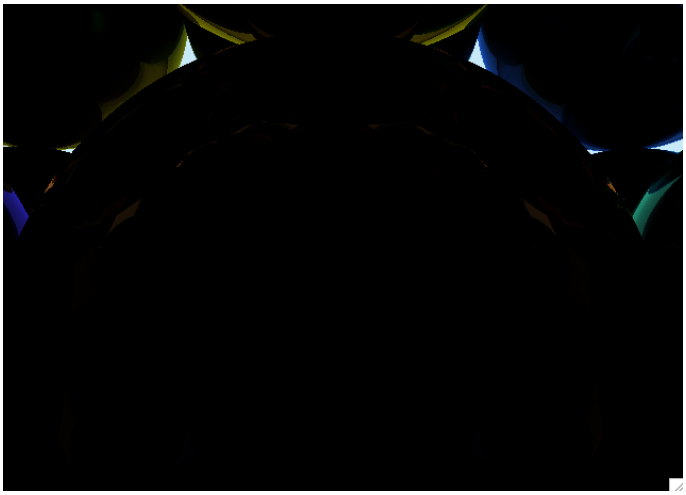


We time six major parts of the CUDA implementation. Setup time is all initialization time, collision time is all time spent comparing rays against objects, scan/map time is counting the number of rays to trace in the next interaction, next layer time is the time to initialize the next layer of rays, surface coloring time is all time to compute the color generated by each ray, and clean up time is any deallocation and copy image out time.

We can observe one of the most distinctly expensive parts of the execution time is the next layer time, which can take up as much as 20% of total execution time. Notice this step is primarily about reading and writing memory and is heavily memory bound. Before optimizing out memory operations, the CUDA implementation was twice as slow, meaning the particularly memory bound next layer initialization would've accounted for the largest proportion of runtime. This

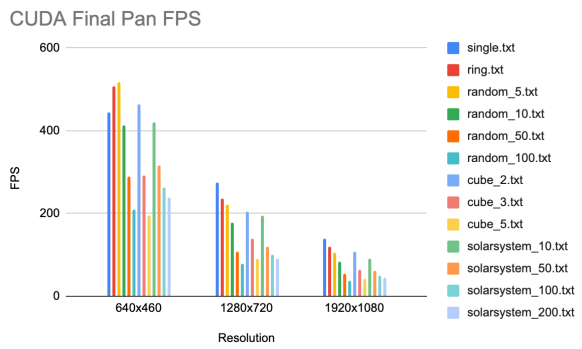
justifies optimizing memory usage first before optimizing collision checking with something like a BVH tree.

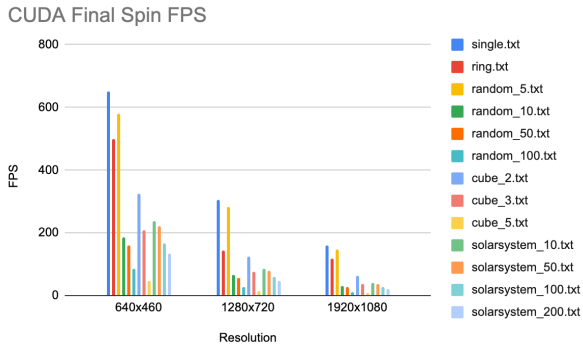
As expected, adding more spheres into the scene increases the surface coloring time and collision time. Interestingly, this value is relatively stable for most of the benchmarks, but for spinning cube_5.txt, we observe a much larger surface coloring time, which can be attributed to spinning in the middle of the 5x5x5 cube, which will mean every ray will traverse to maximum depth. We see rendered images like the one below. Just about every initial ray is going to collide with a sphere.



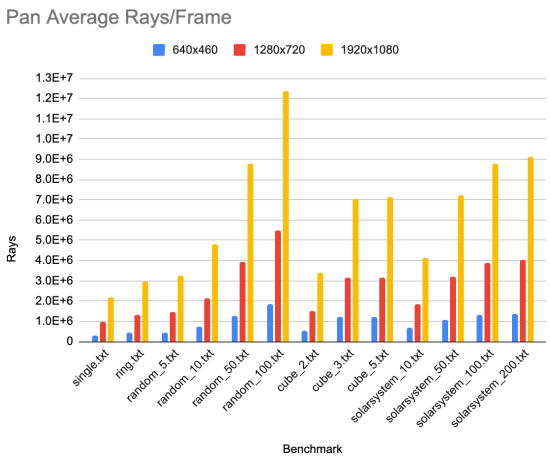
Remarkably, even with around 100 spheres, we don't see a drastic difference in the collision checking time relative to the surface coloring time despite one being significantly more expensive than the other in theoretical runtime (O(N) for all rays versus mostly O(1) runtime sometimes O(N) runtime per ray, which means the arithmetic intensity for 100 spheres isn't even that high versus the memory overhead of computing a parent ray surface color from its refracted and reflected child. Should our benchmarks have more spheres, this behavior would likely change to have the collision checking be even more dominant in total execution time.

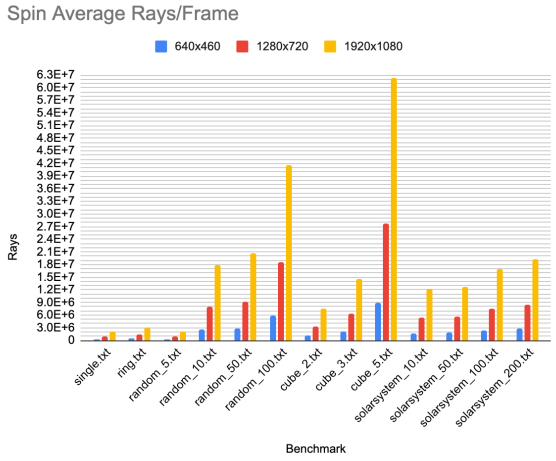
The final frames per second by the CUDA implementation for each benchmark is graphed below.





As expected, adding more spheres and increasing resolution decreases the obtained FPS. Placing the camera angle in a position to cause a lot of sphere collisions also significantly decreases FPS. Observe that most scenes at 640x460 were rendered at over 200 FPS for panning and panning always outperforms spinning in every state. This can be attributed to the fact that significantly more rays diffuse immediately in the pan case, as observed below, where there can be 6x as many rays generated by spinning compared to panning, which accounts for why panning can run multiple times as fast as spinning.





Surprisingly, with software ray tracing and relatively simple scenes, we are obtaining reasonable and visually acceptable rendering speeds on the benchmarks with CUDA. Using a GPU to compute ray tracing is a sound choice. While this is purely speculation, this suggests with modern hardware and good parallel implementations along with relatively few reflective/refractive surfaces, it is not unreasonable to see why ray tracing is becoming an increasingly popular technique in modern games.

OMP/AVX

References

Introduction to Ray Tracing: A Simple Method for Creating 3D Images (Source Code). https://www.scratchapixel.com/code.php?id=3&origin=/lessons/3d-basic-rendering/introduction-to-ray-tracing. Accessed 17 Dec. 2022.

Work Distribution

Jonathan Ke

- Created repository/initial template to host website.
- Constructed test cases.
- Constructed benchmarking harness.
- Constructed test harness.
- Implemented common code for all ray tracing versions.
- Implemented, optimized, and benchmarked all CUDA implementations.
- Ported reference raytracer into test harness.
- Contributed to proposal and project milestone report.
- Contributed summary, background, CUDA approach, benchmarking section, naive results, and CUDA results in final report.
- Implemented OpenGL live display and other rendering harness features for demo.

Kavya Tummalapalli

Percentage Distribution:

70% Jonathan Ke - 30% Kavya Tummalapalli