Parallel Path-Tracing in OpenCL and OpenMP

Neel Gandhi

ngandhi@andrew.cmu.edu

Darwin Torres Romero

dtorresr@andrew.cmu.edu

Project Web Page: https://sirlegolot.github.io/ray_tracer.html

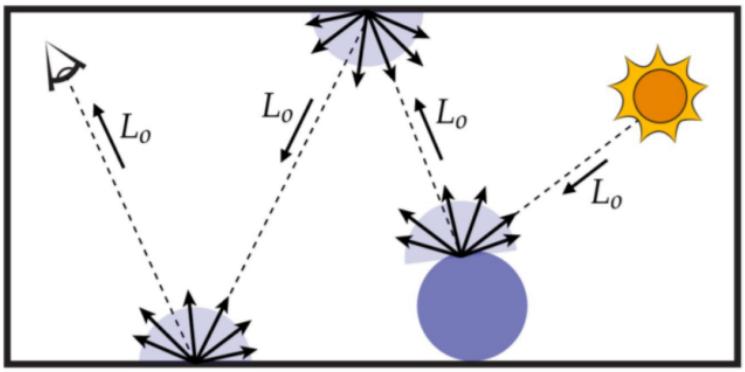
Summary

In this project, we implemented an OpenMP CPU and OpenCL GPU accelerated Monte-Carlo ray tracer and compared the performances of the two implementations. These were tested on an AMD Ryzen 7 3700x CPU (8 cores, 16 threads) and a Radeon RX 5700 XT GPU. Compared to a serial CPU implementation, we observed up to 10-11x speedup from OpenMP with 16 threads. The GPU implementation achieved over 60x improvement over the OpenMP implementation with no BVH acceleration and comparable performance with BVH acceleration.

Background

Ray tracing is a technique used in computer graphics to render more realistic lighting of objects in a scene. The basic idea, which can be simply gathered from the name, is to trace rays from the camera to the scene, and determine lighting based on how those rays intersect with objects in the scene, as well as the material property of the objects. The main computation of ray tracing thus boils down to being able to quickly send these rays and tell whether/how they intersect objects in the scene.

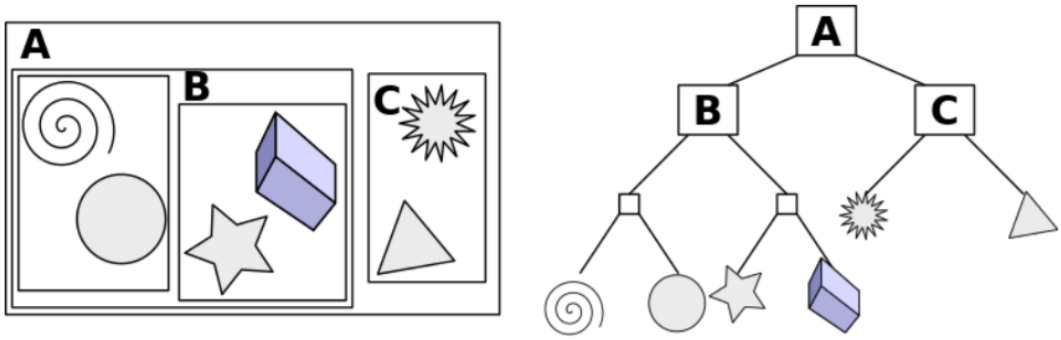
In a Monte Carlo path tracer, the light contribution for each pixel in the image is a function of the material the ray from that pixel intersects with as well as integrating the indirect light contribution from around the point of intersection. The "rendering equation" which describes this is recursive in nature as for a hit point on a material, we generate new rays whose direction is in the distribution of possible angles of incoming light for the material. The contribution of those new rays is accumulated into the light value for the pixel. The diagram below visualizes this concept



Below is pseudocode of the main trace ray function. Note its recursive nature, where it calls itself at the very end of the function.

```
def trace_ray(scene, ray):
    # Start off with no light
    accumulated_light = (0, 0, 0)

    # If the ray has bounced the max number of times, just return
    if ray.depth > max_depth:
        return accumulated_light

    # Intersect the scene with the ray. If it misses, just return
    hit_info = scene.intersect(ray)
    if not hit_info.hit:
        return accumulated_light

    # Otherwise, collect the light from the hit object (this involves
    # recursively tracing another ray from the hit point towards the light
    # source)
    accumulated_light += hit_info.emission # Simplified for pseudocode

    # Terminate the ray with random roulette
    if random.random() < 0.5:
        return accumulated_light

    # Otherwise, continue bouncing the ray
    ray.depth += 1
    accumulated_light += attenuation_multiplier*trace_ray(scene, ray)

    return accumulated_light
```

The most computationally expensive component of ray tracing is intersecting the scene. The first idea would be to have the rays loop through all the objects in the scene to determine if that ray intersects with an object. This can quickly get extremely slow if the number of objects in the scene is high. One widely used method to accelerate ray tracing is the use of a bounded volume hierarchy (BVH). A BVH aims to organize the objects in a scene in the form of a tree so that a ray only needs to traverse down a single/few paths in a tree to check which objects it intersects with, as shown in the diagram below.



Here is some pseudo code describing BVH traversal. Note its recursive and branching nature.

```
void find_closest_hit(Ray* ray, BVHNode* node, HitInfo* closest)
{
   if (node->leaf) {
      // process leaf
   } else {
      HitInfo hit1 = intersect(ray, node->child1->bbox);
      HitInfo hit2 = intersect(ray, node->child2->bbox);

      NVHNode* first = (hit1.t <= hit2.t) ? child1 : child2;
      NVHNode* second = (hit2.t <= hit1.t) ? child2 : child1;

      find_closest_hit(ray, first, closest);
      if (hit2.t < closest.t)
         find_closest_hit(ray, second, closest);
   }
}
```

Ray tracing stands to greatly benefit from parallelism simply from parallelizing the computation of rays, since there isn't any dependency between rays. However, optimizing for different architectures is the difficult aspect of doing so.

At the surface, since each ray computation can be done independently, it seems that getting a parallel implementation up and running will not be so difficult. The only interaction between different samples are the updates of the output pixel value at the end of each calculation. There is no changing state in the scene that each sample has to worry about, meaning we can immediately start new samples without synchronization. For a framework like OpenMP, this is all good news and makes it easy to get started. However, the same can not be said for a GPU. There are other problems that must be considered when using a GPU to build a ray tracer.

One big issue is that each ray can experience vastly different execution paths, resulting in heavy divergent execution. This can mean inefficient utilization of the GPU, limiting the potential speedup we can achieve. One challenge in implementing a good OpenCL ray tracer will be in preventing divergent execution by making sure we group rays experiencing similar execution

paths into the same thread warps. One major aspect of divergence is the fact that rays tend to finish tracing rays at completely different points in time. In SIMD style execution, this would mean that rays that are finished get dragged down with the rest of the active rays, limiting SIMD efficiency. Another source of divergence is in BVH traversal, where different rays will traverse down different paths in the tree. In our project we aimed to tackle these issues.

Notation

Note that because we are using OpenCL, some of the terms typically used in CUDA are slightly different:
OpenCL work group -> CUDA thread block
OpenCL wavefront -> CUDA warp (both are 32 threads)
OpenCL local memory -> CUDA shared memory

We try to stay consistent with the OpenCL terminology in this paper.

Development

The outline of this section is first a mostly chronological order in how this project was implemented and optimized. The following section provides an overall view summarizing the key components of the approach taken in the final solution.

We first started off with a version of a path tracer that Darwin had written for a computer graphics class, written in C++. It supports a limited number of object types (spheres and triangles), surface types (lambertian. mirror, refract, glass, and diffuse), and lights (rectangular, and point). The sequential implementation of it was used as a basis for comparison. It follows the basic algorithm as described in the previous section.

OpenMP implementation

We first began parallelizing the implementation via OpenMP. There are two main aspects with which we can parallelize: samples and pixels. In this path tracer, each pixel shoots `n_samples` many rays, and the output color in the image is the average of the colors returned by the rays for each pixel. The number of samples is adjustable from 1 to infinity (of course more realistically a couple thousand). Essentially, it would look like this, with the two for loops easily being able to be swapped around:

```
for pixel in pixels:
      total_light = 0
      for sample in n_samples:
            Total_light += trace_ray(scene, ray)
      image[pixel] = total_light / n_samples
```

In the case of a few samples per pixel, the approach we took was to parallelize by the pixel computations themselves, having each thread shoot `n_samples` many rays and averaging the color for the pixel it is assigned to. The threads would then in parallel write out the resulting colors in the output image. Assignment was done dynamically to threads because of the variability in how deep rays can go in the image. Some rays may just never intersect with any object in the scene and return immediately, while other rays would bounce around a couple of times. Thus dynamic assignment would allow for better load balancing. There are plenty of pixels in the image (in the order of hundreds of thousands or millions) to distribute the load evenly.

Another approach we tested out was to parallelize by the number of samples. This is only feasible when the number of samples is greater than the number of threads given, so that each thread can perform at least one sample for all the pixels in the image. We decided to perform a static assignment in this scenario since each thread would perform ray computation for all the pixels in the image at once, and with this high of a granularity, the cost of dynamic assignment was higher than simple static assignment. For example, if there are 64 samples per image, and 16 threads, each thread would perform 4 samples for all the pixels in the image. Finally, there is some synchronization cost as we have to atomically update and average the values of the pixels, which is minimal.

We thought that the first approach would prove to be better because of the dynamic assignment of pixels, but the fine granularity of the assignment proved to not provide much benefit in speedup. The second approach performs just as well, as the execution times for each sample for the same pixel would generally follow a similar path.

Overall, the OpenMP implementation was reasonably straightforward, and for 16 threads, provided close to 11x speedup. We are not able to achieve 16x speedup because of slightly uneven load balancing in ray computations, as well as some synchronization costs in reducing to get the average across all the samples. The speedup is also generally dependent on the type of scene, as there are differences as to how many ray bounces occur. The OpenMP implementation would also serve as a basis of comparison for the next part, which is accelerating this algorithm with OpenCL.

Initial OpenCL Implementation

The first task of accelerating via OpenCL was to deal with the fact that the original codebase was written all in C++ 17 syntax with templates and several recursive data structures and functions. This recursive aspect is what the typical implementation of a path tracer would look like, as described in the section above. While the overall logic and algorithm of the path tracer would stay the same, to use with the GPU, the code would have to be overhauled into OpenCL C (C99 syntax) and converted to iterative format. This includes the various math library functions on floating point vectors and matrices, the various ray intersection methods for

different types of primitives and light sources, the sampling schemes for BSDFs, BVH traversal, etc.

The overall path tracing algorithm could be modified to run in terms of iterations, where each iteration corresponds to a bounce/depth of a ray. At each iteration, a multiplier would be accumulated as well as the total light/color collected, and multiplied together to get the effective contribution of light at the depth of the ray. In terms of object representation, the original code was defined recursively in that for each object in a scene a BVH for the triangle mesh was constructed separately, followed by constructing a BVH of all the BVH. This nested BVH data structure had to be flattened out to be able to easily send over into GPU memory. Another aspect about BVHs is that as they are trees, a typical representation would be to use pointers to represent the left and right children of a node. Pointers in CPU will not translate to GPUs, so the representation used for the BVH was an array of nodes, where each node's left and right pointers are just indices into different parts of the node array. This is similar to a typical implementation of a fixed size heap.

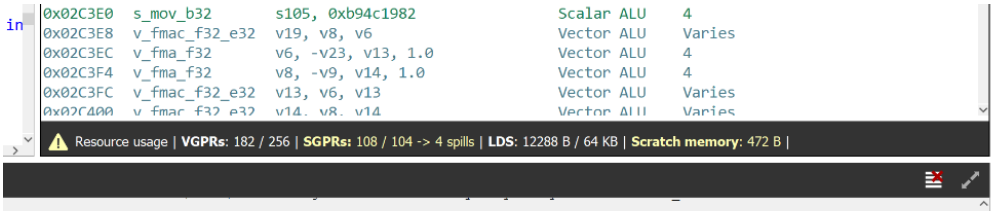
With some effort, an initial implementation of the path tracer was completed for the GPU. The mapping to parallel hardware was as follows. Each thread is assigned one ray computation. The total number of threads would be the number of samples per pixel times the number of pixels. Each work group (the equivalent of a thread block in OpenCL), contains `n_samples` many threads, with the minimum number of samples being 32, the size of a wavefront (the equivalent of a warp on AMD GPUs). On the 5700 XT, the maximum work group size was 256, so we were limited to a maximum of 256 samples running at once before we had to loop the entire process again to get the next samples. In parallel, the threads would execute the path tracing algorithm and shoot their rays to collect the light/color in each step. Initially, while the BVH was copied over into GPU memory, a simple looping traversal through the primitives was performed for ray intersection calculations. When all the rays returned, a work group reduction would be performed to add all the accumulated light together and finally average. We tried this reduction both via a compare and exchange atomic floating point addition as well as a built in OpenCL work group reduction. No noticeable difference in execution time was observed because the amount of threads contending was a maximum of 256, which was relatively small compared to the execution time of the remainder of the path tracing algorithm.

This initial implementation was all implemented as part of one big megakernel. Apart from a separate initializing kernel that sets up the global arrays and variables accessed throughout, the remainder of the functionality was all completed as one kernel invocation. The speedups measured here were limited (NOTE: our midpoint report was inaccurate for GPU speedups, as we discovered that our random number generator algorithm was consistently returning 0 or values close to 0, causing rays bounces to head in the same direction (especially for light calculations) and exit immediately after a single bounce). For a standard non-BVH ray intersection, the speedup observed was around 8-10x compared to the CPU implementation. While this is definitely a speedup, we would generally expect to observe far more as the GPU can potentially run over 2000 threads simultaneously, compared to only 16 threads on the CPU.

Following, we present various optimizations we attempted with varying degrees of success:

Kernel Pipeline

As stated earlier, the initial implementation of the GPU path tracer was structured as one single megakernel. While there were some speedups, this structure was severely limiting for performance. First of all, especially when there was no BVH traversal, running a single kernel for a lot of samples and depth for each ray was simply not possible, due to various reasons such as AMD driver timeouts freezing our PC. A single megakernel also severely limits any sort of modifications and improvements that are explained below. Finally, after conducting a GPU kernel analysis using the AMD GPU Analyzer, it was found that there was a little bit of register spilling and scratch memory usage occurring, as shown in the image below. Register spilling is when a given GPU thread does not have enough space to store all of the variables it needs in the registers it was assigned when scheduled. When this occurs, it has to store things in a separate, far slower piece of memory (usually this will be global memory on AMD GPUs). This made sense because our megakernel performs all the computations for ray tracing in one go, referencing variables defined at the very beginning of its code.



As such, we overhauled the path tracer implementation to be structured as a pipeline of several smaller kernels, each dedicated to a specific task of the path tracer algorithm. A simplified version of the pipeline is as follows (fully explained in the **Final GPU Approach** section):

Primary ray generation
v
Primary ray scene intersections
v
Primary ray processing & secondary ray generation
v
Secondary ray scene intersection
v
Secondary ray processing & next bounce ray generation

This pipeline of kernels would be run for each sample within a pixel and depth of a ray, with the same assignment strategy of a ray per thread. However, this time instead of grouping all the samples of the same pixel in the same work group, we performed only a small number of

samples (specifically 8) of each pixel in each iteration of the pipeline. This enabled tracing rays for higher sample counts than 256. Work groups were now a collection of more than one pixel.

Running the GPU kernel analyzer again, this pipeline system did not have any register spilling, but still used scratch memory in the ray scene intersection kernels. Scratch memory is used to temporarily store values during lengthy calculations involving registers. We speculate our matrix multiplications, among other expensive math operations, contribute toward the kernels' usage of scratch memory. In terms of speedup, there was little improvement, if at all when switching to the kernel pipeline, even when reducing register spilling. There is one big reason for this, which is the fact that now we were launching multiple kernels right after each other, instead of just once. To put this in perspective, if each iteration had four kernel launches, and the number of iterations was 4 for the max depth of a ray multiplied by 512 samples per pixel, this would be 8,192 kernel launches. While the cost of launching a kernel is very small, this might offset any improvement gained over fixing a small amount of register spillage.

However, the pipeline kernel enabled the implementation of a lot more optimizations, as described next. This is due to its far more flexible structure, allowing us to introduce newer kernels in between each section or even modifying one of the kernels which is a small section of the kernel pipeline.

BVH traversal

As explained earlier, the typical way to accelerate path tracing algorithms is to use a BVH to represent the objects in a scene as a tree. This makes scene intersections go from a linear operation to a logarithmic operation in time, as a ray only needs to go down a few paths in a tree. This results in remarkable speedups of the ray tracing pipeline. There are a couple of issues with BVH traversal implementations on a GPU. The standard algorithm is essentially a recursive depth-first search of the tree to find the closest intersecting object. As recursion is not possible on a GPU, the algorithm must be made iterative. The bigger issue is that BVH traversal will lead rays down completely different paths, leading to both execution and data divergence. Here are a couple of optimizations we attempted with varying success, with our main point of comparison being with the multithreaded CPU implementation, where branch predictors essentially make BVH traversal free of any major cost.

Stackless Traversal

The first issue we wanted to address is how GPUs have limited stack space, which can be impacted when using an iterative stack-based depth-first traversal. With trees with a large depth, the stack could grow really large, and would definitely cause register spilling to occur. As such, we looked into a stackless BVH traversal method proposed by Hapala et al. In this method, a simple state machine is used to determine which child node to visit next in the traversal. It requires storing the parent information in a BVH node, and keeps track of the last

visited node. With this information, a backtracking traversal can be implemented. While the order of nodes visited will end up the same as that of a stack based traversal, the backtracking does require going back up to a parent node in some cases to visit the other child node. On a GPU, this would mean extra global memory accesses when revisiting the parent node. When implemented in our GPU path tracer, this was about the same performance to that of the multithreaded CPU ray tracer. We had initially thought it would be way better, but it seems that extra global memory accesses, along with the branch divergence were limiting its performance.

Stack-based traversal

So instead of stackless traversal, we switched to using a stack-based BVH traversal. We used a fixed size stack of 32, which would allow for traversal of a tree of max depth 31. The code for stack-based traversal is a lot cleaner in terms of if-else cases compared to the state machine implementation from Hapala et al, and would have less global memory accesses as we are not revisiting parents. The general flow of execution is popping from the stack, intersecting with the left and right nodes, and pushing whichever children have hits onto the stack. In the case of a leaf node, the primitives in the leaf are intersected in order. This stack traversal gave a 5-10% performance compared to the stackless implementation. In this case flow execution was the same for all rays in the while loop, with only how long staying in the while loop being variable. The other source of divergence is data divergence, as rays will visit completely different nodes. But this cannot be avoided when using a BVH.

Experimenting with leaf size

The improvement of switching to stack-based traversal was not that big of a change however, and we suspected that perhaps it is because of how large the depth of the tree is, leading to a lot more varying degrees of traversal in terms of depth for different rays. One way to combat this is two increase the leaf size (i.e. the number of primitives stored at a leaf node). Changing from a leaf size of 1 to 32 decreases the amount of levels in the BVH by at least 5, and with this we noticed a huge boost in improvement of 1.5-1.8x in some scenes. Anything more would lead to more sequential traversal, which was slower.

Traversal Pruning

At the cost of a bit more branch divergence, BVH traversal can prune visiting nodes. For example, if both the left and right child intersect with the ray, and you hit a primitive on the left subtree, which turns out to be closer than the intersection with the right child node, there is no need to continue visiting the right child node anymore. Another heuristic is that if both the left and right child nodes intersect, we want to prioritize traversing whichever node intersects first. We added both pruning mechanisms to the GPU traversal code, and while the first did improve the performance by around 5-10%, adding the second reduced speedup by the same amount. This was expected due to increased branch divergence of rays. We ended up only performing the first method of pruning.

Local/Shared Memory optimization

When no BVH is used, rays will all traverse the same list of primitives in the same order to test for intersections. We can take advantage of this fact and use shared memory to speed up these ray intersections. We can split up the list of primitives into chunks that are the size of the work group of threads, and have threads collectively work together to load the primitives into shared memory. This prevents having each thread accessing global memory frequently. After loading in a chunk, the threads will perform intersection calculations and update the closest hit appropriately. This is then repeated for the next chunks until completion. With this optimization by itself, we were able to achieve significant improvement of 1.8x-2x compared to before.

Ray Reordering

One of the key difficulties of implementing a GPU ray tracer is the branch divergence due to the fact that rays have varying degrees of how many bounces they go through in a scene before they either reach their max depth or terminate early (for example via Russian Roulette). Some rays could bounce around for a while, while others could immediately miss the scene and return. This extremely variant aspect of rays can cripple performance as rays that have already returned do no useful work in a SIMD lane and get dragged along with others that are still bouncing in a scene. Another issue is other work-groups can not be scheduled until there are resources available due to previous work-groups completing their execution. When there are threads in a work-group that have completed their work, these lanes are not being utilized. This occurs across all active work-groups. When multiple threads are completed across multiple work-groups, we can see that they might take up enough resources that would allow another work-group full of active threads to be scheduled. To tackle these problems, we can reorder the rays so that active rays are packed together to improve SIMD utilization. All non-active rays are discarded. There are two levels of ray reordering that we implemented: work group level reordering and global level reordering.

Work-group Reordering

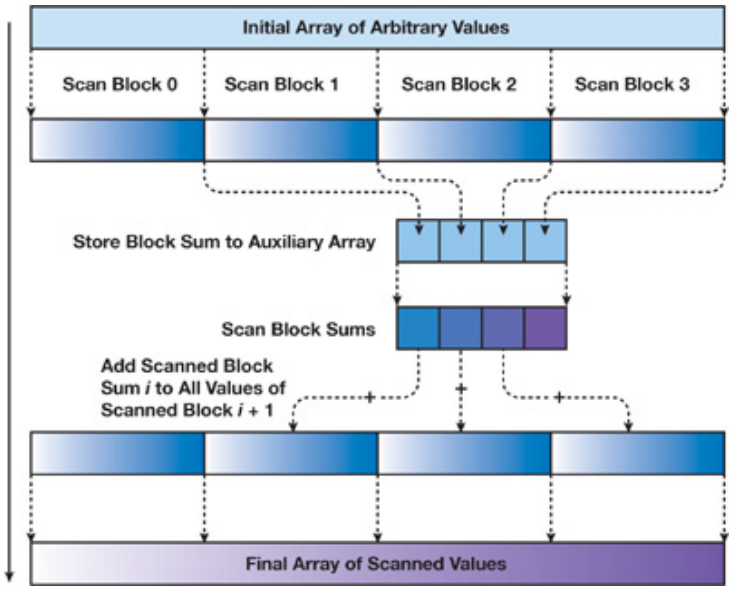
The simpler of the two ray reordering mechanisms is to only reorder rays within a work group. Looking at the AMD documentation, SIMD is assigned with linear work group id. That is, with a SIMD width of 32, the first 32 threads in the workgroup will get packed together and run with SIMD, followed by the next 32 and so on. If we can compact the active rays towards the front, we can improve SIMD utilization by making all active threads run at once. All other threads will be placed towards the end of the work group in terms of linear id, and they will all exit early together when they are completed.

We perform ray reordering with exclusive scan compaction. Two exclusive scans are performed for arrays indicating whether a ray is done and not done. They are compacted together so that all active rays are pushed to the front and finished rays at the end of the work group. This optimization by itself also gave a boost of about 1.8x, which is significant. However, there is still more work to be done. Work groups are of size 256 max, and with 32 wide SIMD lanes, local ray reordering can only help so much. All the threads at the end of a work group still get

scheduled by the kernel to do no useful work, which is a bottleneck. Also, it is possible for groups of 32 threads to eventually reach a point where only a few of them are active within a work group. We want to be able to compact these with active threads from other work groups.

Global Reordering

With a global reordering of rays, we have to perform a global scan compaction of active rays. Unfortunately GPU global memory has quite high latency in accesses, making a global scan similar as implemented in assignment 2 quite slow. There were no optimized libraries such as thrust that we could add for OpenCL, so we implemented a version of exclusive scan ourselves. The way this scan worked was that work groups would perform local/shared memory scans and put their results into the global output array. They would also put their final result in a different global array indicating the scan result for each work group. This new array would be scanned, and the scan output would be added back to the original output array to compute the global scan. This scan is recursively defined, with the base case being when the size of the array being scanned is smaller than the work group size. In that case, we can just perform a shared memory scan. While this scan implementation is more complicated, it aims to improve speedup by taking advantage of the much faster local memory exclusive scans and building the final result.



-> Nvidia

In the kernel pipeline, rays are reordered after the first primary ray scene intersection (in the case the rays miss) and after the secondary ray processing (when rays either finish their last bounce or are terminated via Russian Roulette). After getting the new number of active rays, all future kernel launches only launch as many threads as active rays. This optimization, by itself, produced a 3x speedup for performance for non-BVH traversal at a ray depth of 4.

Ray Scheduling

The path tracer kernel pipeline was originally formatted as two nested loops. The outer loop was over the number of samples per pixel in the image. The inner for loop is over the max depth of a ray. Inside the inner loop is the series of kernels to perform ray tracing (primary rays, secondary rays, processing, ray reordering, etc.). We noticed that by formatting the loops this way, rays tended to finish very quickly in the inner loop after leaving the scene or terminating via Russian Roulette. While they were being discarded as we perform ray reordering, they could instead immediately start the next sample before having to wait for the other rays to finish this round. This would keep SIMD utilization up.

By itself, this optimization improved speedup by 1.8x for non-BVH. However, when combined with ray reordering, it was slightly slower than with the old loop ordering. This was because rays were being active for far longer (as they immediately started the next sample), and the global exclusive scan on a large number of rays was expensive. The cost of global array accesses outweighed the benefit of keeping up utilization here. We experimented with assigning a granularity as to how frequently ray reordering occurred (i.e. once every 5 iterations or 10 iterations), but have too small of a value would be slow because of frequent global scan and large values would prevent the benefit of ray reordering to keep SIMD utilization up. For BVH traversal, this optimization crippled performance (going from 3 second render times to 350 second render times). This was due to increased branch divergence as rays that were originally just reordered away when they were finished were now restarting their traversal at completely different points with respect to the rest of the threads in a wavefront. After further research, we discovered that Aila et al. also pointed this scheduling mechanism out, determining that although SIMD utilization is maintained, the memory access incoherence outweighed any benefit. As such, we decided to forgo this optimization in the final approach.

Math optimizations

Switching sine, cosine, and sqrt functions to use native hardware instructions, along with compiler optimizations for floating point math resulted in a gain of 10-12% speedup.

Putting the optimizations together

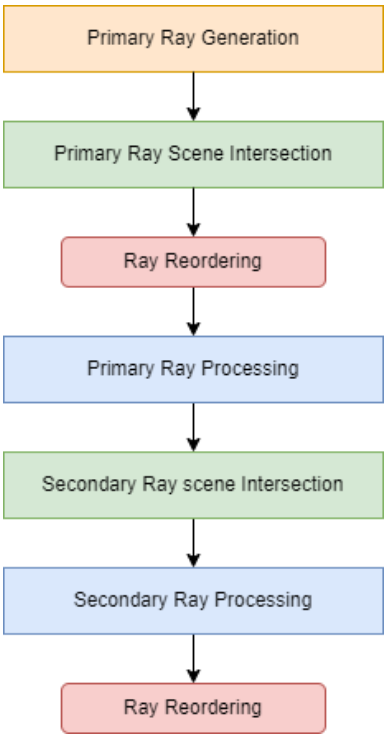
We presented a series of optimizations that we attempted, some of which were extremely successful and some that weren't. Overall, the non-BVH traversal stood to gain the most benefit from these optimizations, given that 99% of the time was spent on iterating through all the primitives in the scene to perform ray intersections. Together, we were able to get over 60x speedup compared to the 16 threaded CPU implementation when not using a BVH, which is around 600x speedup of the sequential implementation. For the BVH traversal, branch divergence severely limited speedup when compared to the CPU implementation, though it was still faster than the non BVHimplementation. Most of the optimizations did not have that much of an impact on the BVHimplementation since not as much time was spent on that as compared to non-BVH. As such, we observed only comparable performance to the CPU implementation of

about 1-3x speedup depending on the conditions. This was still 10-30x faster than the sequential implementation. We fully analyze our results in the results section.

Final GPU Approach

In this section we summarize the overall structure of the GPU path tracer pipeline, using all the optimizations discussed previously.

The overall flow is as follows:




```
# Initialize all GPU buffers, copy over data
...

# Main loops
for sample_chunk in range(n_samples / sample_gran):
    initialize_rays()
    for depth in range(max_depth):
        ray_primary_hit()
        ray_reorder()
        if num_active_rays == 0:
```

```
            break
        process_primary_hit()
        ray_secondary_hit()
        process_secondary_hit()
        ray_reorder()
        if num_active_rays == 0:
            break

# Copy over data to CPU and output image
...
```

Ray Initialization

This is the first step of the process, where we initialize rays that pass through random positions inside every pixel. Each thread is assigned a sample for pixel, with there being 8 samples per pixel (This is the maximum amount we could fit in one iteration without overflowing GPU memory for large image sizes). The ray information is stored in a global array, and contains the following:

- Origin (12 bytes)
- Direction (12 bytes)
- Time bounds (8 bytes)
- Depth (4 bytes)
- Accumulated Light (12 bytes)
- Potential Indirect Light (12 bytes)
- Light Value Multiplier (12 bytes) [Used to multiply with the accumulated light]
- Hit information (36 bytes) [Used for determining hit objects]
- BSDF sampling information (36 bytes) [Used for material information]
- Random seed (8 bytes) [Used for random number generation]
- Img linear index (4 bytes) [Used to access the respective pixel in the image]
- Done flag (1 byte)

While this is a lot of information per ray, when there are potentially millions of rays being traced in a scene, it is all necessary for color calculations

Primary Ray Hit

This is where the first scene intersection occurs. Each thread gets assigned to the ray that is at its global index in the global rays array, and there are 256 threads in a thread block. In this kernel, all threads perform a scene intersection. If not using a BVH, the 256 threads in a thread block work in chunks of 256 objects at a time, bringing them into shared memory and processing them accordingly as described above. This significantly reduces global memory

access times. In the case of using a BVH, each thread traverses the BVH on their own, with the traversal scheme aimed at minimizing as much execution divergence as possible. This was the stack based traversal with basic pruning, as described above. However, each ray will eventually be accessing different pieces of data in the tree, causing data divergence, as well as traversing for different amounts of time. After primary ray traversal is completed, rays that do not intersect with the scene add their accumulated light color to the output image buffer and mark themselves as done.

Process Primary Hit

In this step, we perform three actions if the ray hits an object:

- Sample the BSDF of the object, which is essentially the material information (reflection, refraction, etc), and place it in the ray struct.
- Sample the indirect light contribution and place it in the ray struct.
- Generate a “shadow ray” originating from the hit point and pointed towards the light source. This ray will also be shot to determine if we can add the indrect light. This is also placed in the ray struct.

Once again, each thread works on the ray that appears at its global index in the global rays array. This step is relatively small in computation.

Secondary Ray hit

The secondary rays are the “shadow rays” that were constructed in the previous step. We shoot these rays in the scene towards the light, and if they intersect with anything before hitting the light, it is considered as being in a shadow of the hit object. In this case, we do not add the potential indirect light. Otherwise the potential indirect light is added, multiplied by the weight multiplier.

Process Secondary Hit

This is the last step in the pipeline, where we check if the ray has reached its maximum depth or if it should be terminated via a probability based on the material it hit. In either case, the ray marks itself as done. All the still active rays then construct a new ray originating from the hit point and in some new direction based on the BSDF of the material it hit. This is the ray for the next bounce or depth, which will be traced in the next iteration of the pipeline

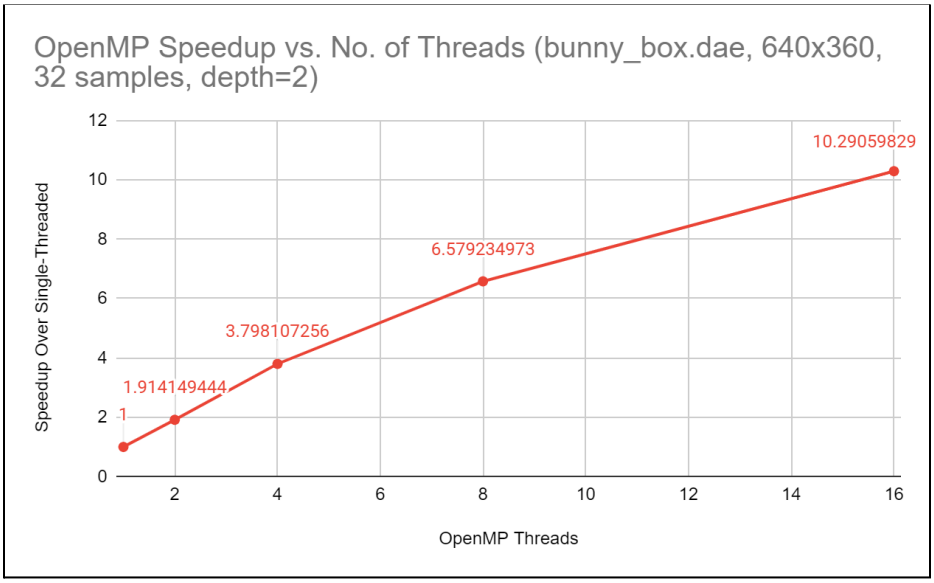
Ray Reordering

This step occurs after both primary ray hit and secondary ray processing. In both steps, as described, rays can mark themselves as completed. In these cases we want to reorder the list of global arrays so that only the active rays are packed together, which will boost SIMD efficiency. This is completed via an exclusive scan compaction that takes advantage of being able to

perform a scan of the larger array in smaller chunks in shared memory, as explained earlier. With this array, we can compact active rays together, so in the next kernel call, all the rays in the wavefronts are active. If all rays are completed in this step, the host will start the next iteration.

Results

OpenMP Speedups

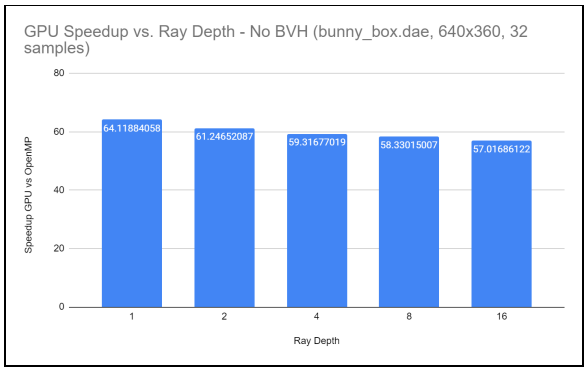


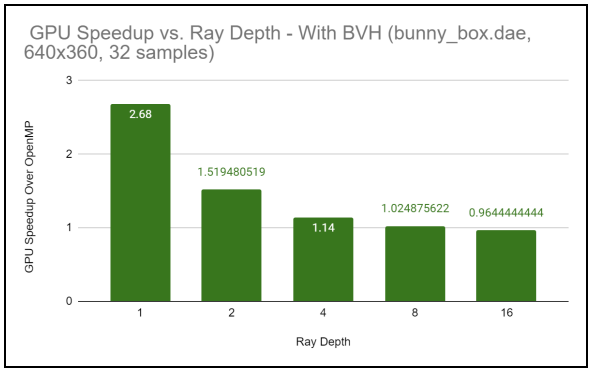
The chart above shows the speedup of the openmp implementation as the number of threads increases. As explained earlier in the development section, while speedup increases are mostly linear, we are unable to achieve a full 16x speedup due to the variation in how long each ray lives throughout the render.

GPU Speedups

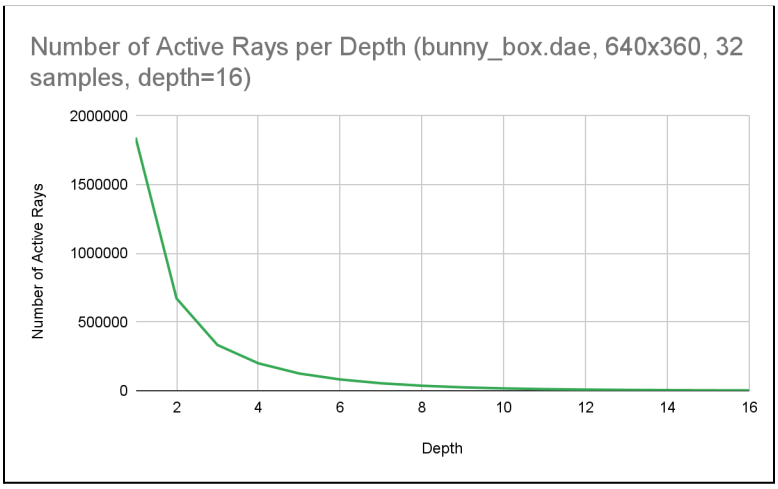
In our analysis, we consider 3 different types of problem sizes: Ray Depth, Samples per pixel, and Image size.

Ray Depth





The above plots describe the speedup for both our BVH and non-BVH implementations when depth is varied. For both versions, as ray depth increases, we see a drop in our speedup. The non-BVH implementation goes from 64 speedup to 57 speedup from 1 depth to 16, while the BVH implementation goes from 2.6 speedup to 0.96 speedup. While our ray reordering optimization strategy significantly improved the speedup observed so that we don't run threads that have already finished, the cost of performing the global array scan at each depth starts to have a more visible effect. Especially in these scenes, by 4 bounces, approximately 90% of the rays have already finished or exited the scene, as indicated by the table below. The very few remaining rays drag on the computation, and not much of the GPU is being utilized at these depths. As a result, parallelization decreases.

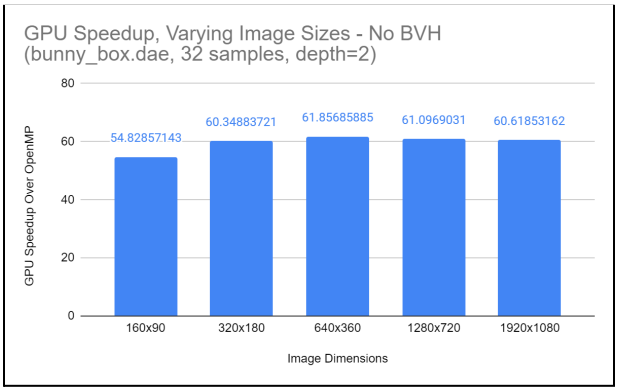


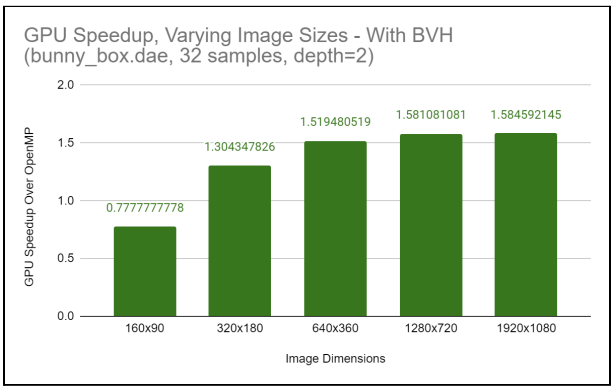
Another thing to note is that for the BVH implementation, the amount by which speedup decreases is more significant compared to the non-BVH implementation. With further analysis of the kernel pipeline timings, we determined that the time to traverse the BVH increases as ray depth increases. We detail the GPU profile execution in a later section. This is due to the fact that in the first bounce, rays that are next to each other tend to visit similar nodes in the BVH and have somewhat similar traversals. However, after reordering rays between each depth, and the fact that rays bounce off in random directions when hitting a lambertian BSDF surface (which is what the scene consisted of, unlike other scenes with mirrors), rays in the same SIMD

vector start to deviate in traversal paths in the BVH significantly. This leads to more execution and data divergence, causing performance to drop significantly.

In our experience with testing around with various scenes, to get a high quality image a ray depth of 2 is quite sufficient for most material types, except for glass, where a ray depth of 4 is needed to trace rays through a glass object and account for refraction. As shown in the table above, this accounts for 90% of the rays anyways, which is sufficient.

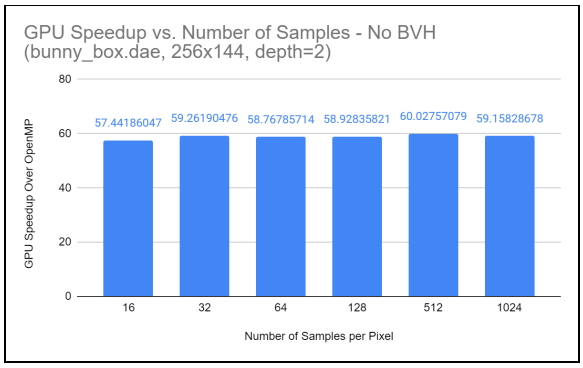
Image Size

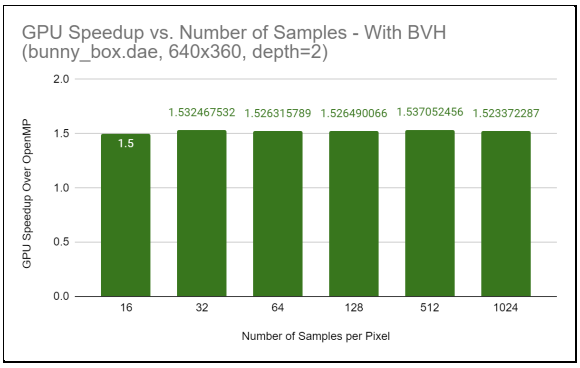




The above plots describe the GPU speedup for both our BVH and non-BVH implementations when image size is varied. For both non-BVH and BVH traversal, the speedup tended to increase as the image size increased. While this is a bit unexpected since all samples are independent of each other, we realize that it is due to the fact that the GPU is able to still perform more parallelization if we give it more samples to work with at the same time (i.e. it was not fully utilized). Currently our kernel pipeline iterates over chunks of samples, with the sample chunk size being fixed at 8. We can easily adjust this so that it modifies the chunk size based on the image size of the output so that at lower image sizes, we can do a lot more samples at once, and at higher image sizes we do less. Special attention should be given to memory usage as the array data of rays will be bigger the more samples that are run at once.

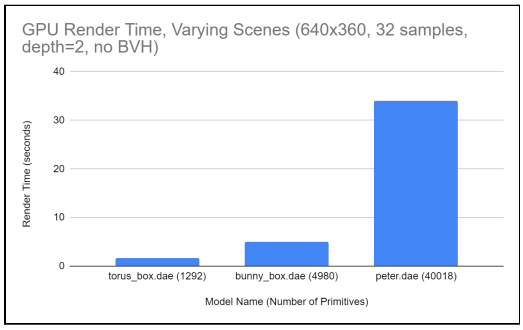
Number of Samples

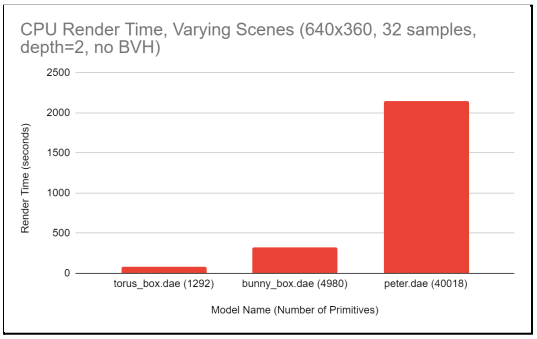


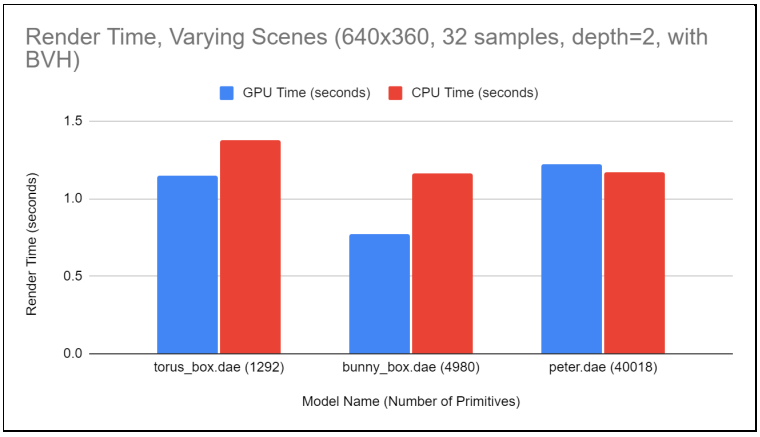


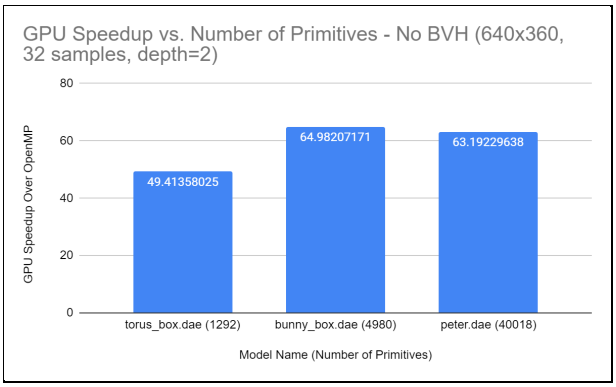
The plots above indicate the GPU speedup as we vary the number of samples per pixel in the image. As we increase the number of samples, we see that there is little change in the speedup of the GPU implementation over the CPU implementation. Both the CPU and GPU implementations have a linear increase in time as the number of samples increases.

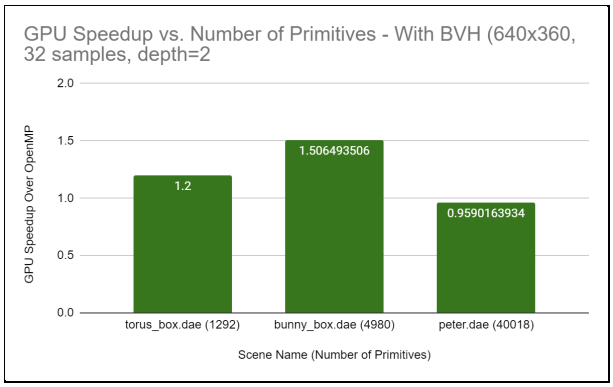
Number of Primitives











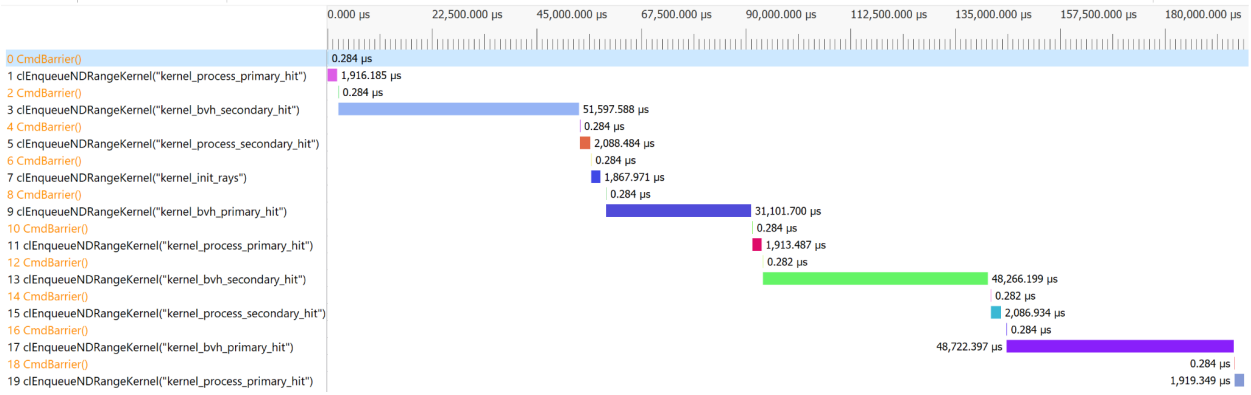
Finally, the last major aspect of problem size we measured is the number of primitives in the scene. For the non-BVH implementation, as we traverse all the primitives, we expect to see the render time being linear with respect to the number of primitives. The data supports this case. For a BVH, it is a lot harder to expect what the render times will look like, because rays will only intersect with the small number of primitives it needs. As such, shown in the plot above, the render times for the three scenes are about the same. In terms of speedup, the GPU implementation with no BVH performs as expected like in the other categories. For BVH, the GPU is comparable with the CPU implementation, sometimes under and sometimes over in render time. This variation simply has to do with the randomness of how rays will travel the scene.

GPU Profiling - Bottlenecks and Improvements

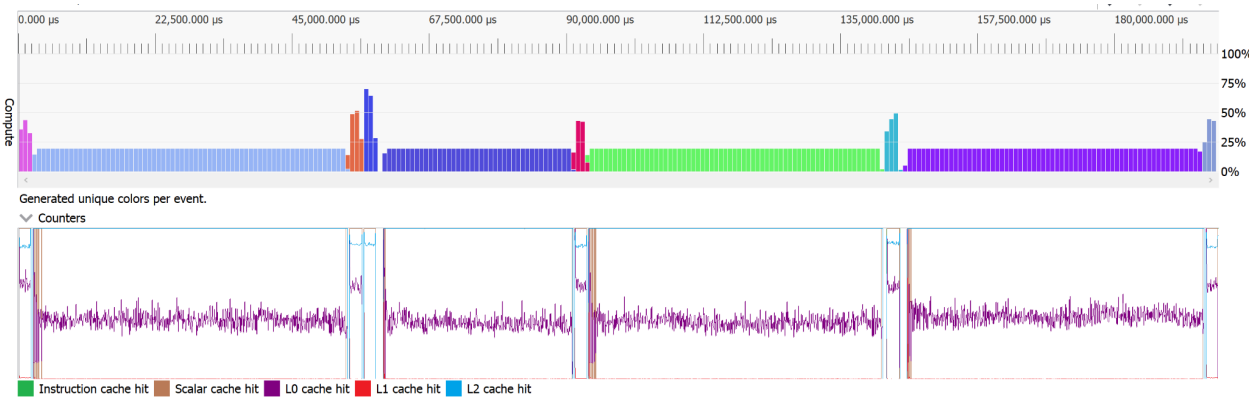
In this section, we break down the algorithm and analyse the various components with the AMD GPU Profiler.

No BVH Analysis

We first begin with non BVH ray tracing, shown below. This snapshot was taken when running on an extremely small scene with only a few primitives to be able to show the timings in one picture. We also don't include the intermediate kernels for exclusive scan compaction for simplicity here (Our profiler only let us trace 50 kernel launches max, and the scan kernels can take a lot of kernel launches due to its recursive nature), but they are included in the BVH traversal profile.

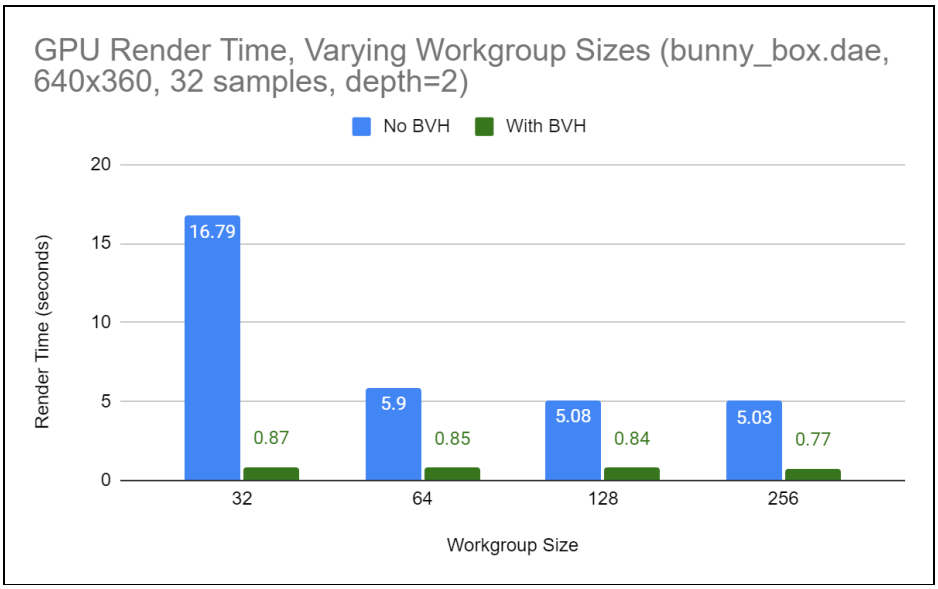


This picture shows two iterations of the kernel pipeline. As clearly shown, the majority of the time is spent on **primary_hit** and **secondary_hit** kernels. In fact, this accounts for 99% of the time for the execution of the entire algorithm. In this picture, because the list of primitives we are running with are small, the bars appear to be smaller. In a much larger scene, these bars will take up the majority of the time and the **process_hit** kernels in between will barely be visible. This all makes sense execution-wise because the most computationally expensive component of non-BVH ray tracing will be the looping through all the primitives in the scene to test for intersections. As such, priority for optimizations must be given mainly to these kernels.



This next picture (top half) shows the wavefront occupancy of the kernels run. Wavefront occupancy tells us how many wavefronts were able to be run at the same time, and can be used as a measure of GPU utilization. To get the most parallelism, you should strive to have the highest wavefront occupancy as possible. The results here are quite striking. We have only up to 50% wavefront occupancy for the **process_hit** kernels and 25% occupancy for the **primary_hit** and **secondary_hit** kernels. This seems to indicate heavy room for improvement. Digging deeper into these metrics, we can see that the reason for this 25% occupancy is because of local/shared memory usage. In non-BVH traversal, we take advantage of shared memory of the threads in a work group so that they can work collectively to load in object data from expensive global memory and perform ray intersection tests. With 256 threads running in a work group, the shared memory array used is 256 in size. Object data is quite large (214 bytes), so we end up using a lot of shared memory.

The picture above shows how we are limited by LDS, which is OpenCL local memory (the equivalent of shared memory in CUDA). In a work group, we are using 54,784 bytes of shared memory, which prevents the GPU from being able to schedule several wavefronts of threads at once. As seen in the blue bar indicator, we are only able to run 4 wavefronts at a time out of the total 20.



To test if decreasing the work group size would help with being able to parallelize more, we tested it with our renderer, and the results are shown in the figure above. Workgroup sizes 64, 128, 256 yield similar render times, however a workgroup size of 32 results in render times that are about three times slower. We speculate that this is because smaller workgroups result in a greater number of recursive levels for our global exclusive scan implementation. Workgroup sizes 64 and greater result in big enough chunks that minimize the amount of recursive levels needed. A small workgroup size like 32 would also hurt performance as more global accesses would be made as we are jumping by smaller chunks at a time.

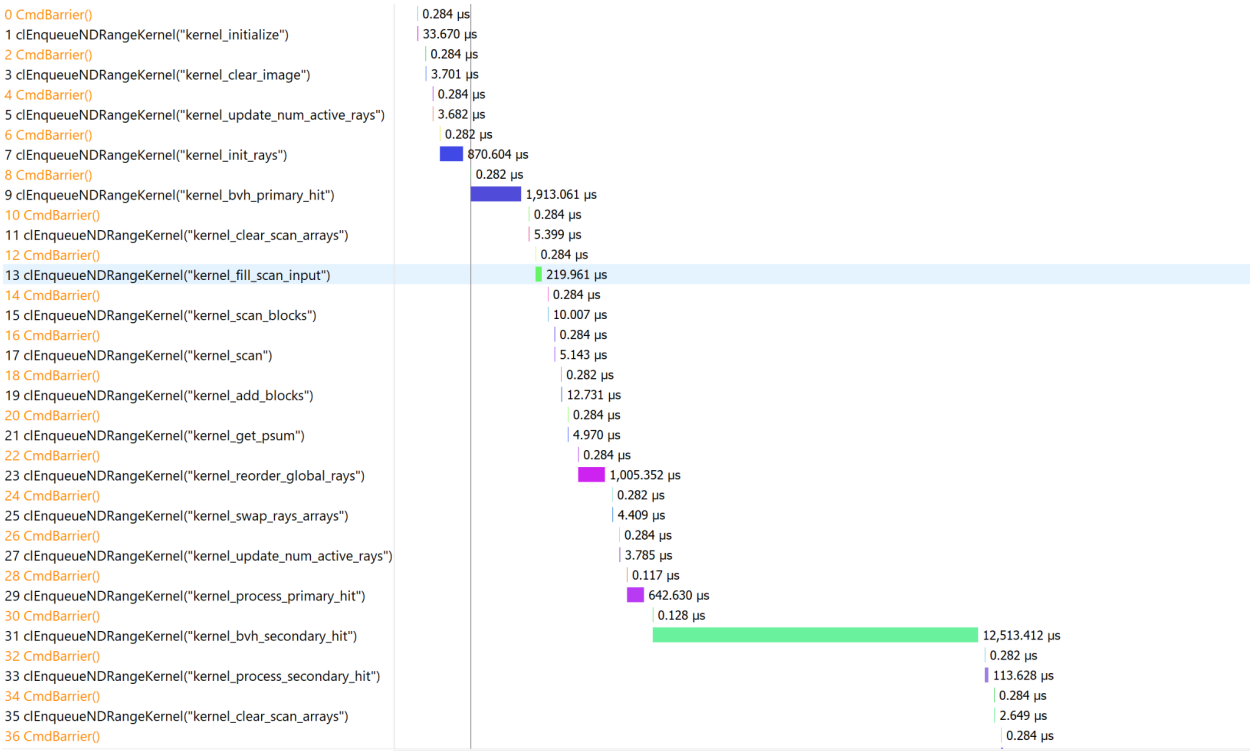
For future work, we should focus on optimizing how much shared memory we use, perhaps by manipulating the data constructs used to store other object data to decrease size. The other kernels which are at about 50% wavefront occupancy can also be optimized. The GPU profiler gives us information usage about register usage by the threads in a work group. As registers are a resource shared by threads in a wavefront, if each thread uses less registers we can manage to squeeze more wavefronts scheduled at the same time. Currently, wavefront occupancy is at about 12 out 20 for the other kernels (image not shown, since it is nearly

identical to the one above). If we can optimize around the code to use less registers, we can also see more parallelism.

Finally, for non-BVH traversal, we can probably see more improvements in speedup by overall reducing the global memory access usage. In our code, a lot of global memory accesses are being made, potentially in not the most optimal way. If we can introduce caching of object data or other pieces of information, perhaps we won't need to access global memory as much.

BVH Analysis

Next, we analyze our algorithm using BVH traversal, especially the bottlenecks due to branch divergence. The GPU profile below was taken when running on the medium sized scene bunny_box.dae. This time we include the exclusive scan kernels, but present the pipeline in two images because of how many intermediate kernels there are.

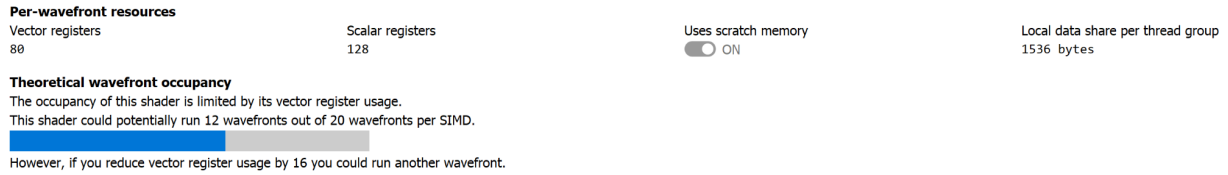


In this first image, we have a trace from the very beginning of the algorithm, where the initializer kernels are called. As shown, the majority of the timings are the **primary_hit** and **secondary_hit**, but not as much to the extent that non-BVH traversal has, at least in the beginning first iteration. One major thing to note is how different the execution times **primary_hit** and **secondary_hit** kernels are. These are the second blue bar from the top and the green bar, respectively. This is actually quite astounding, because after **primary_hit**, a lot of rays in the scene already get pruned (especially the rays on the sides of the image that never intersect anything in the scene). We would expect that after reordering, because a lot of rays are removed, the execution time of **secondary_hit** would be equal, if not less than the **primary_hit**.

This is clearly not the case, and can be attributed to branch divergence. In the first ray-scene intersection, threads in a work group will get assigned to either samples of the same pixel or to pixels that are next to each other. As such, their directions will be very similar, and thus traversal through the BVH will be similar. As they are accessing very similar nodes, there is little divergence. However, after the primary hit, some rays are pruned and the remaining rays are bounced in random directions towards the light source. At this point, rays will now traverse more different parts of the BVH, leading to higher execution and data divergence. In fact, the more bounces that occur, the worse that this divergence gets, as shown in the pictures below.

Looking at the timings for the **primary_hit** and **secondary_hit**, the execution times of the kernels gets longer and longer, from 1.9 ms, 12.5 ms 14.5 ms, 18 ms, 20 ms. This steady increase in execution time from increased branch divergence eventually results in the percentage time taken by these two kernels to also be over 99% of the entire algorithm, just like non-BVH traversal. Clearly the best action for improvement will be to significantly decrease the branch divergence (especially data divergence which causes incoherent memory accesses). We discuss this in the next steps section.

In this above picture (top half), we show the wavefront occupancies with BVH traversal. Except for the kernels used for exclusive scan compaction, the occupancy is much higher than that of the non-BVH traversal, over 50%. This makes sense because we are using less shared memory resources with this implementation, so we can run a bit more wavefronts in parallel.



Finally, we look at the wavefront occupancy more closely for the **primary_hit** (which is similar to **secondary_hit**). As shown, we are running 12 wavefronts at a time, which is greater than the 4 from non-BVH. Parallelism can also be increased by potentially manipulating the data constructs around so we don't hold as large of data in the registers when running these kernels. This will allow us to schedule more wavefronts.

Overall however, for the BVH implementation, the main area of improvement needs to be to reduce the branch divergence of ray BVH traversal. We discuss what we had planned in the next section.

Next Steps

We have successfully shown how ray reorganization techniques can drastically improve performance when used to compact active rays together., especially with non-BVH traversal. However, as detailed throughout this project, the current BVH traversal implementation severely lacks in speedup due to divergence issues. With limited time left, we were unable to tackle the issue fully. However, a plan of next steps was made.

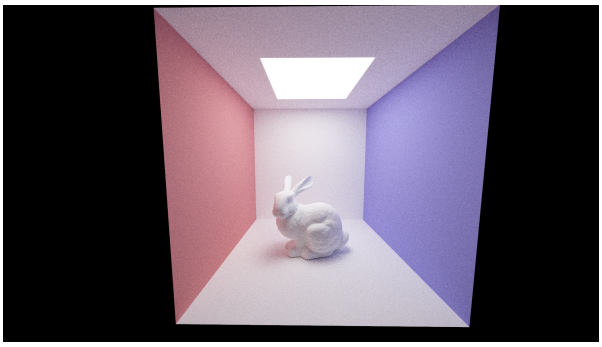
Similar to how rays are reordered so that active rays are put at the front, we can apply a reordering strategy to BVH traversal. In our BVH representation, nodes that are next to each other from left to right appear in increasing order of node index in the BVH array. If we can sort the index of nodes that the threads are accessing in their BVH traversal, this could help with decreasing data divergence by packing rays that are traversing similar paths together. Special

attention will have to be given to whether doing a global array sort or simply sorting within a work group will be better, given the slower global memory access times.

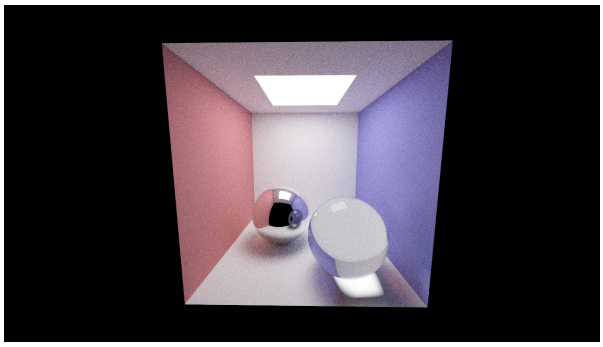
Another strategy that we considered promising was dynamic ray scheduling. Instead of a typical depth first search traversal, we can do a breadth first search in parallel at each level. This would require changing the view from a per-ray to per-node, where we maintain queues of rays visiting each node. Rays in a work group would all be accessing the same node in the BVH and perform intersection tests to determine whether to go left or right (or both), and add themselves to the respective queues of the children they want to visit. This can be performed via some smartly placed exclusive scans and atomics. At the next level, we would have a queue of rays for each node, and the process would continue.

With respect to our choice of using a GPU to run our ray tracer, it seems that in recent times, they are finally able to catch up to CPU renderers. Historically, photorealistic rendering via path tracing has been done on CPUs, with production studios using render farms of thousands of CPU cores to render the frames of their 3D animated films. As great as GPUs are for parallel computations, they have been considered unfavorable for ray tracing due to their limited memory and pitfalls with branched execution. From our research on the usage of GPUs for rendering and our experience with using software that supports rendering on GPUs, such as Blender, we see that GPUs can achieve just as good or sometimes even better performance than CPUs. A prime example of a popular GPU ray-tracing API is Nvidia Optix, which has been shown to allow modern GPU renderers to achieve comparable performance, if not better, to Intel Embree, which is a highly optimized CPU renderer that takes advantage of SIMD vector operations. Hopefully with the optimizations we strive to put in the future we can get closer to the speeds of these world-class renderers.

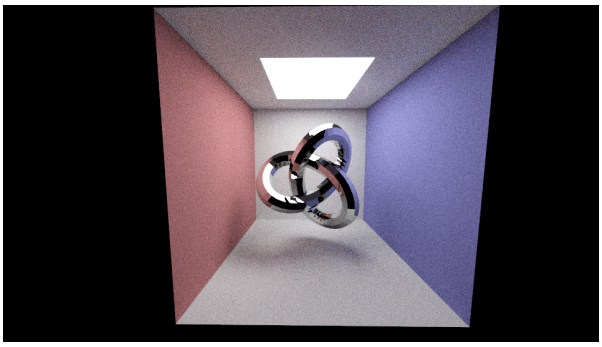
Renders



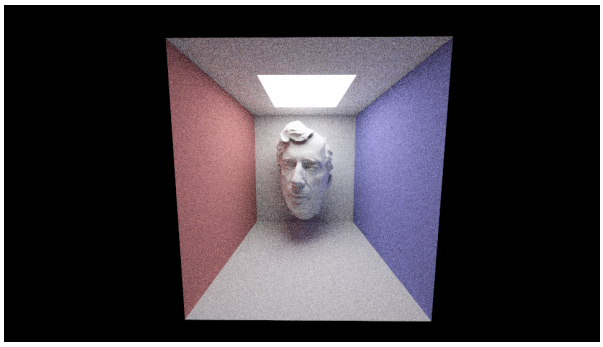
bunny_box.dae



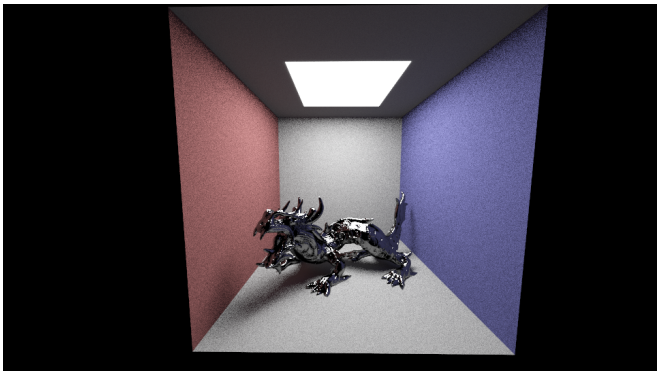
cbox.dae



torus_box.dae



peter.dae



dragon.dae

References

[1] Hapala, Michal, et al. "Efficient stack-less bvh traversal for ray tracing." Proceedings of the 27th Spring Conference on Computer Graphics. 2011.

[2] Aila, Timo, and Samuli Laine. "Understanding the efficiency of ray traversal on GPUs." Proceedings of the conference on high performance graphics 2009. 2009.

Project Split

Neel Gandhi (ngandhi) - 60%
Darwin Torres (dtorresr) - 40%