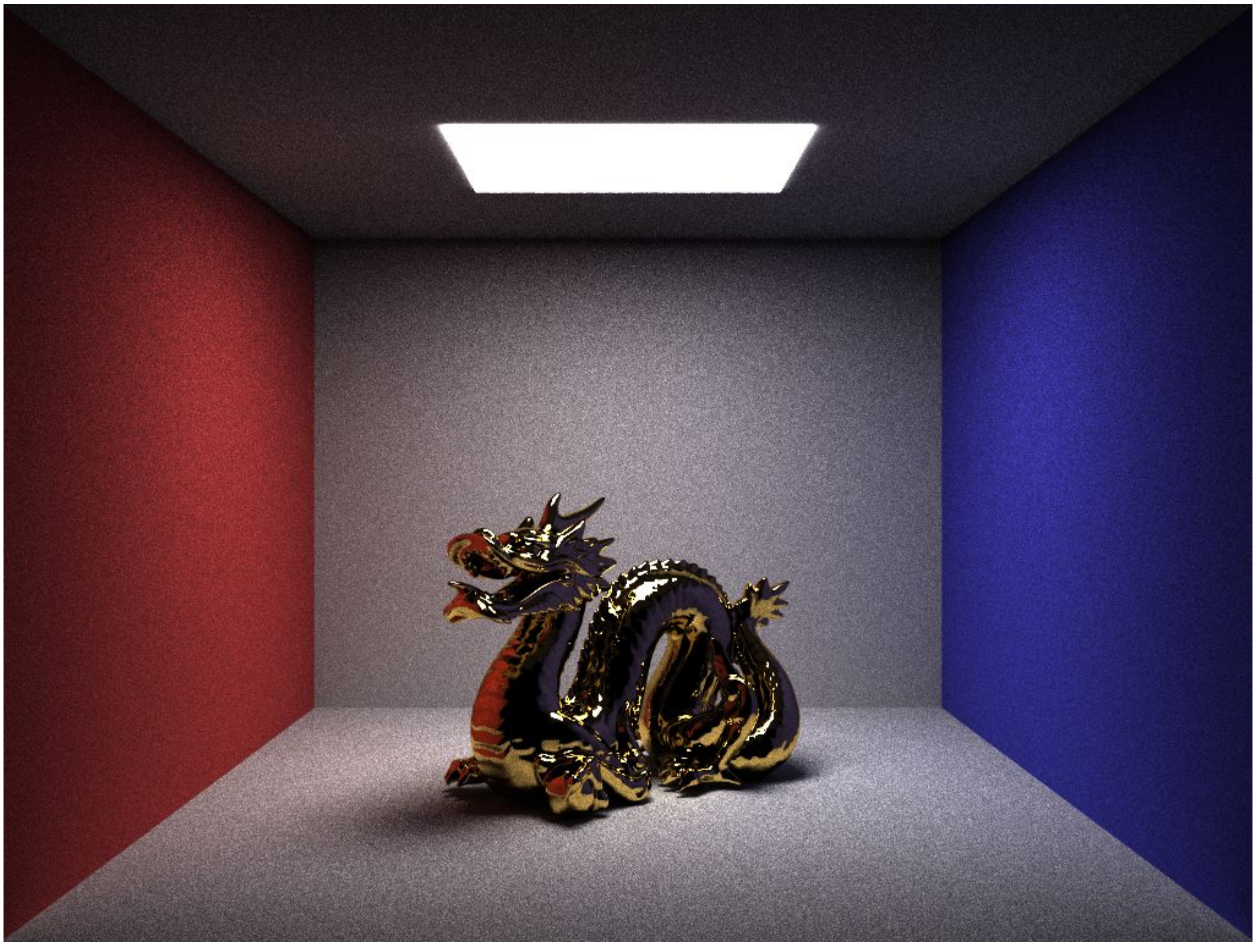
15-618: Parallel Computer Architecture and Programming

CUDA-SCOTTY

Fast Interactive CUDA Path Tracer using Wide Trees and Dynamic Ray Scheduling



“Golden Dragon”

TEAM MEMBERS

Sai Praveen Bangaru (Andrew ID: saipravb)

Sam K Thomas (Andrew ID: skthomas)

Introduction

Path tracing has long been the select method used by the graphics community to render photo-realistic images. It has found wide uses across several industries, and plays a major role in animation and filmmaking, with most special effects rendered using some form of Monte Carlo light transport.

It comes as no surprise, then, that optimizing path tracing algorithms is a widely studied field, so much so, that it has it's own top-tier conference (**HPG;** High Performance Graphics).

There are generally a whole spectrum of methods to increase the efficiency of path tracing. Some methods aim to create **better sampling methods (Metropolis Light Transport),** while others try to **reduce noise** in the final image by filtering the output **(4D Sheared transform)**.

In the spirit of the parallel programming course **15-618**, however, we focus on a third category: system-level optimizations and leveraging hardware and algorithms that better utilize the hardware (**Wide Trees, Packet tracing, Dynamic Ray Scheduling)**.

Most of these methods, understandably, focus on the **ray-scene intersection** part of the path tracing pipeline, since that is the main bottleneck. In the following project, we describe the implementation of a hybrid **non-packet** method which uses **Wide Trees** and **Dynamic Ray Scheduling** to provide an **80x** improvement over a **8-threaded CPU implementation.**

Summary

Over the course of roughly 3 weeks, we studied two **non-packet** BVH traversal optimizations: **Wide Trees**, which involve non-binary BVHs for shallower BVHs and better Warp/SIMD Utilization and **Dynamic Ray Scheduling** which involved changing our perspective to process rays on a per-node basis rather than processing nodes on a per-ray basis.

We then merged these algorithms and replaced any synchronization requirements with smartly placed **atomics** and **exclusive scan** operations to create a GPU friendly pipeline.

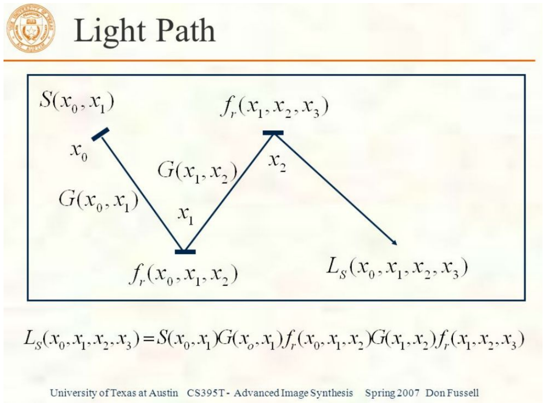
The final pipeline gave us a massive boost (of around **90x**) over the **multi-threaded** implementation of Scotty3D. While Scotty3D is not massively optimized, it's relative simplicity gives it rather high throughput.

We compare our renderer with the **8-threaded** (presumably, this ran on a Xeon processor that is currently installed on the GHC machines) **reference implementation** provided by the organizers of the **15-442: Computer Graphics** course, to avoid any bias due to a possibly suboptimal student implementation. All our results are computed on a **NVIDIA GTX 1080 Ti**.

Background

The only relevant background ideas one needs to understand our implementation are the **Monte Carlo Path Tracing** algorithm and the working of a **BVH (Bounding Volume Hierarchy)** acceleration structure.

Monte Carlo Path Tracing


Light Path

$$L_S(x_0,x_1,x_2,x_3)=S(x_0,x_1)G(x_o,x_1)f_r(x_0,x_1,x_2)G(x_1,x_2)f_r(x_1,x_2,x_3)$$

University of Texas at Austin CS395T - Advanced Image Synthesis Spring 2007 Don Fussell

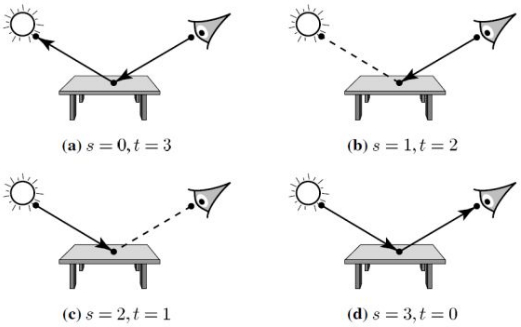


Figure: (Left) Computing the total contribution for a single path. Note how it can be done by iteratively accumulating an **importance** value formed by **f** and **G**.

Figure: (Right) Possible ways of creating light paths. A simple path tracer only uses the **[s=1, t=2]** and **[s=0, t=3]** paths.

Monte carlo path tracing is a fairly popular technique used in the Computer Graphics industry to produce physically realistic images (See first page for an example).

The primary idea behind this is to estimate the **rendering** equation based on the **Monte Carlo,** which is to estimate the total contribution to each pixel by randomly generating path from an unbiased random distribution of possible paths and evaluating it's contribution.

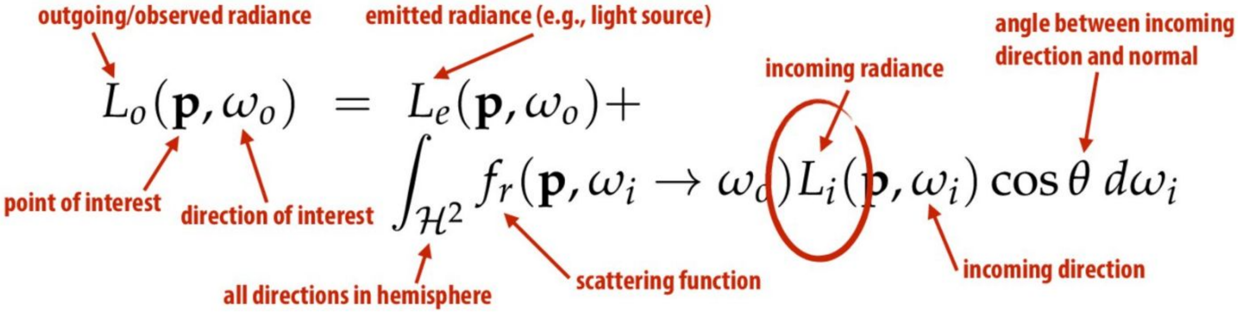

outgoing/observed radiance
emitted radiance (e.g., light source)
incoming radiance
angle between incoming direction and normal
$L_o(\mathbf{p},\omega_o) = L_e(\mathbf{p},\omega_o) + \int_{\mathcal{H}^2} f_r(\mathbf{p},\omega_i \to \omega_o) L_i(\mathbf{p},\omega_i) \cos\theta \, d\omega_i$
point of interest
direction of interest
all directions in hemisphere
scattering function
incoming direction

Figure: The recursive rendering equation.

The recursive rendering equation can be solved iteratively as well, which is important for a GPU ray tracer, because of a lack of stack space.

The important part of the rendering equation (bottleneck) that we will attempt to speed up here is the geometry **G(x)** term which involves **visibility**, which means we need to find the point of intersection with the scene geometry.

For this we use an acceleration data structure, the most common of which, is a **BVH.**

Bounding Volume Hierarchy (BVH)

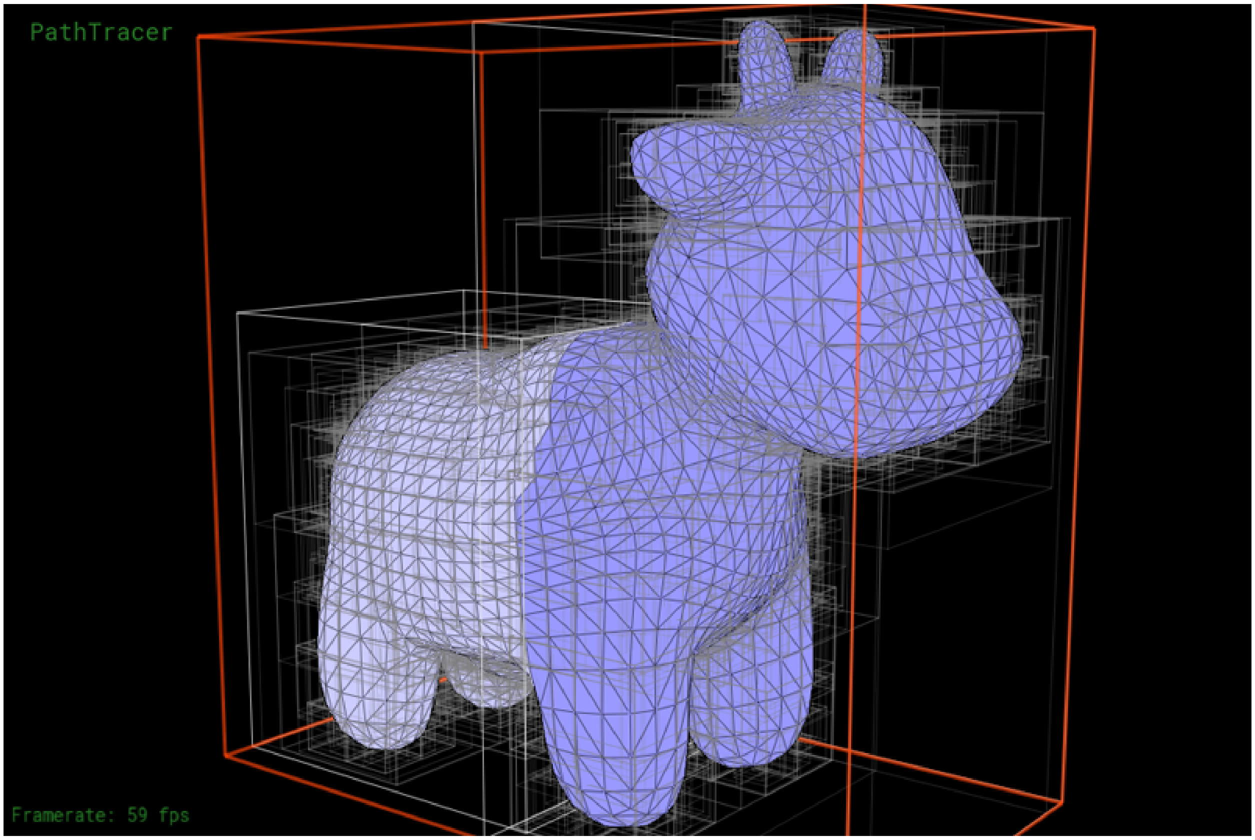


Figure: BVH visualization on the Scotty model. Visualization taken from the github for the course.

A BVH is a spatial tree data structure that is used to organize large scale scenes involving thousands or even millions of primitives. The primary unit or **node** contains an axis aligned bounding box that encompasses all the primitives of the subtree that it's rooted at.

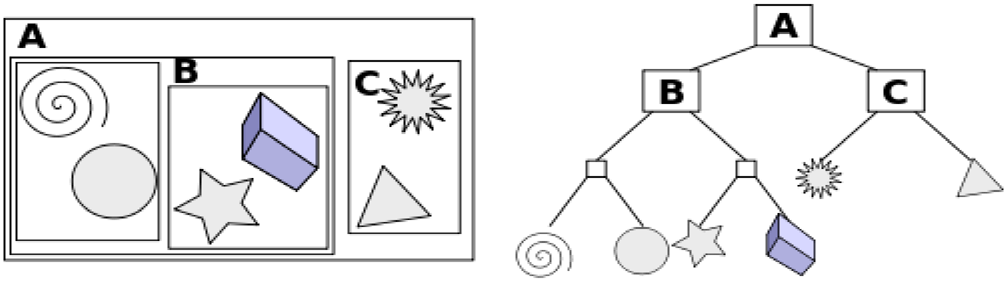


Figure: Illustration of primitives in a BVH

Each internal node contains a bounding box, which is the easiest primitive to intersect a bounding box with.

A leaf node simply contains a list of primitives.

One important property of a BVH is that while no primitive can be in more than one leaf node, a ray can intersect any number of nodes since bounding boxes can overlap.

This means that the BVH is useless if we consider worst case complexity. However, in practice, using a decent BVH construction algorithm provides an almost drastic reduction in traversal time, usually scaling as ***O(log n)***.

It should be noted that other data structures like Octrees and KD-trees *do* have theoretical logarithmic bounds, however they are harder to work with, less general and sometimes slower in practice.

Also, BVH construction code was readily available, so we went with that.

The BVH construction is done on the **CPU** since we work with static scenes, and then transferred into the **GPU**.

Motivation

The first and most important thing about porting code to GPUs is to throw away the CPU code, because in most cases, it will not work well with GPUs because of the radically different operating characteristics.

What are the differences?

A GPU has a **very small cache** which is shared among threads but very **high bandwidth main memory**. A CPU has a relatively **gigantic cache** for each core, but **slow main memory**. Indirectly, this implies that sequential/streaming access works best on GPUs.

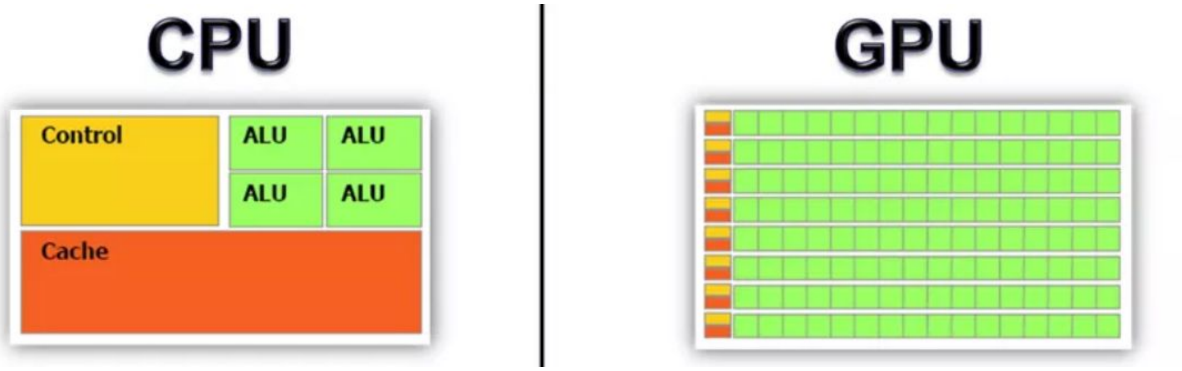


Figure: A visual comparison of the difference between a CPU and a GPU.

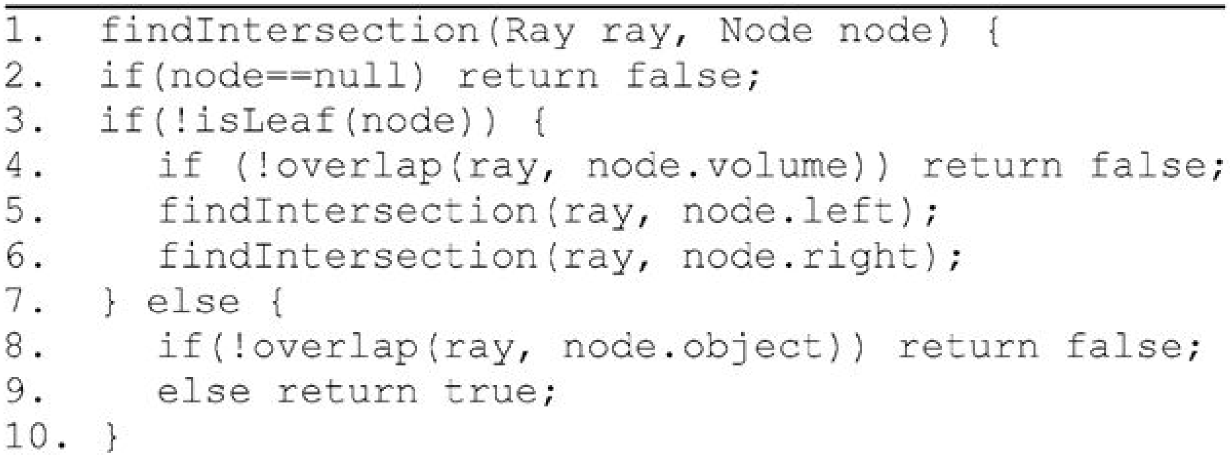
Image credit:

A GPU has **32-way SIMD** and **no proper branch-prediction** which works best when there is relatively no divergent if statements. A CPU tends to have **4 to 8-way SIMD** and extremely powerful **branch predictors** and **prefetchers** which means conditions are no big deal.

A GPU also has **no cache coherency** or any semblance of synchronisation-specific hardware. A CPU is the opposite, and is built with a lot of embedded synchronisation primitives. A GPU works best, therefore, with **low-contention** code.

The Design: Overview

A typical **CPU** implementation of BVH traversal involves a **recursive** function that tests each bounding box rooted at that subtree with the ray and recursively calls itself on whichever nodes intersect the ray. This is similar to a **depth-first** traversal of the tree (the difference being that some (most) paths are pruned because the ray doesn't intersect it's bounding box.)



```
findIntersection(Ray ray, Node node) {
if(node==null) return false;
if(!isLeaf(node)) {
  if (!overlap(ray, node.volume)) return false;
  findIntersection(ray, node.left);
  findIntersection(ray, node.right);
} else {
  if(!overlap(ray, node.object)) return false;
  else return true;
}
```

Figure: A typical CPU implementation of a BVH traversal. This approach does not carry over smoothly to GPUs because of limited stack size, and high divergence.

A common optimization in this case, is to first traverse the subtree that has a closer intersection with the ray. It usually gives significant gains on a CPU. (In the **Optimizations** section, we describe why we ignore this).

According to the characteristic differences between a CPU and GPU mentioned in the previous section, we see that the standard CPU implementation of a ray tracer falls prey to most of the above pitfalls.

It uses **random accesses** (since a BVH is a tree, there's no way to sequentially index every possible path).

It also uses **divergent conditions** (Rays on adjacent threads could go a completely different way, reducing both SIMD utilization and Memory utilization)

Both these problems prevent the GPU from achieving full utilization.

How do we combat it?

The first and most importance bit is to change the perspective from **per-ray** to **per-node**. This is referred to as **Dynamic Ray Scheduling** (which has found successful implementations on CPU code, but not implemented on a GPU, yet.), and in this mode, we process a list of rays per node and compute their target nodes and transfer the rays to that node.

Assuming we keep the rays at each node in order, the access (within a block) is fully sequential. This mitigates one problem that we had. However, we do run into another problem. The rays have to be fetched for **every** node of the BVH. There usually are several thousand nodes in a BVH (not counting leaves). That can significantly slow down the implementation.

This is where **Wide Trees** come in. **Wide Trees** are essentially non-binary BVHs. Each node can have 4, 8, 16 or more trees at each stage. This means that we will be reducing the number of nodes in total and the number of levels in the BVH too. For a GPU, memory access is a bigger issue than raw compute power. So the additional computation at each step is usually small compared to the gain from a smaller number of nodes. For larger tree branching indices, we find that the tradeoff becomes less certain and at some point, one effect dominates the other. **4-way BVHs** tend to give the best results (a considerable reduction of almost **30%** in total rendering times compared to **2-way** and **8-way**).

Algorithm

We present the stages algorithm of the algorithm in full detail in the same order that it appears in the code.

BVH Compaction and Serialization

There are two steps that happen on the BVH, **Compaction** and **Serialization**.

Compaction refers to the conversion of a binary BVH to an **N-way** BVH. This can be done using a fairly simple ***O(n)*** breadth-first traversal of the BVH. Note that unlike binary BVHs, it's possible to have fewer than **N** branches at each node. These are just replaced with **NULL**s.

Compression refers to converting the **pointers** into list indices, because host pointers differ from **device** pointers. It's not possible to directly send the tree data to the device. This step allots a unique integer ID to each node, places them into a contiguous list and replaces all branch pointers with their respective IDs.

Primary Ray Generation

This step is fairly simple and parallelized it is fairly straightforward. Each pixel is handled by one thread. Each thread creates a specified number of samples for that pixel.

We avoid having to access any global structure by storing all required information within the ray structure. Each ray carries the following important data:

- Origin vector (12 bytes)
- Direction vector (12 bytes)
- Max Distance (4 bytes)
- Ray ID (4 bytes)
- Current Importance (12 bytes)
- Accumulated Light (12 bytes)
- Potential Light (12 bytes) [Used for Direct Light Estimate, This holds the light that will be added to “Accumulated Light” if the visibility step is successful]
- Screen Space sample (8 bytes)
- Depth (4 bytes)
- Additional Debug Stuff + Padding (50 bytes)

BVH Ray Traversal

Each call to the kernel handles one level of the BVH. (There are usually around 8-10 levels in the BVH) Each level could have a lot of nodes (Anywhere from 1-1000). Since we use a level-by-level approach (with synchronisation between levels), we do not need to worry about the order of execution within each kernel execution.

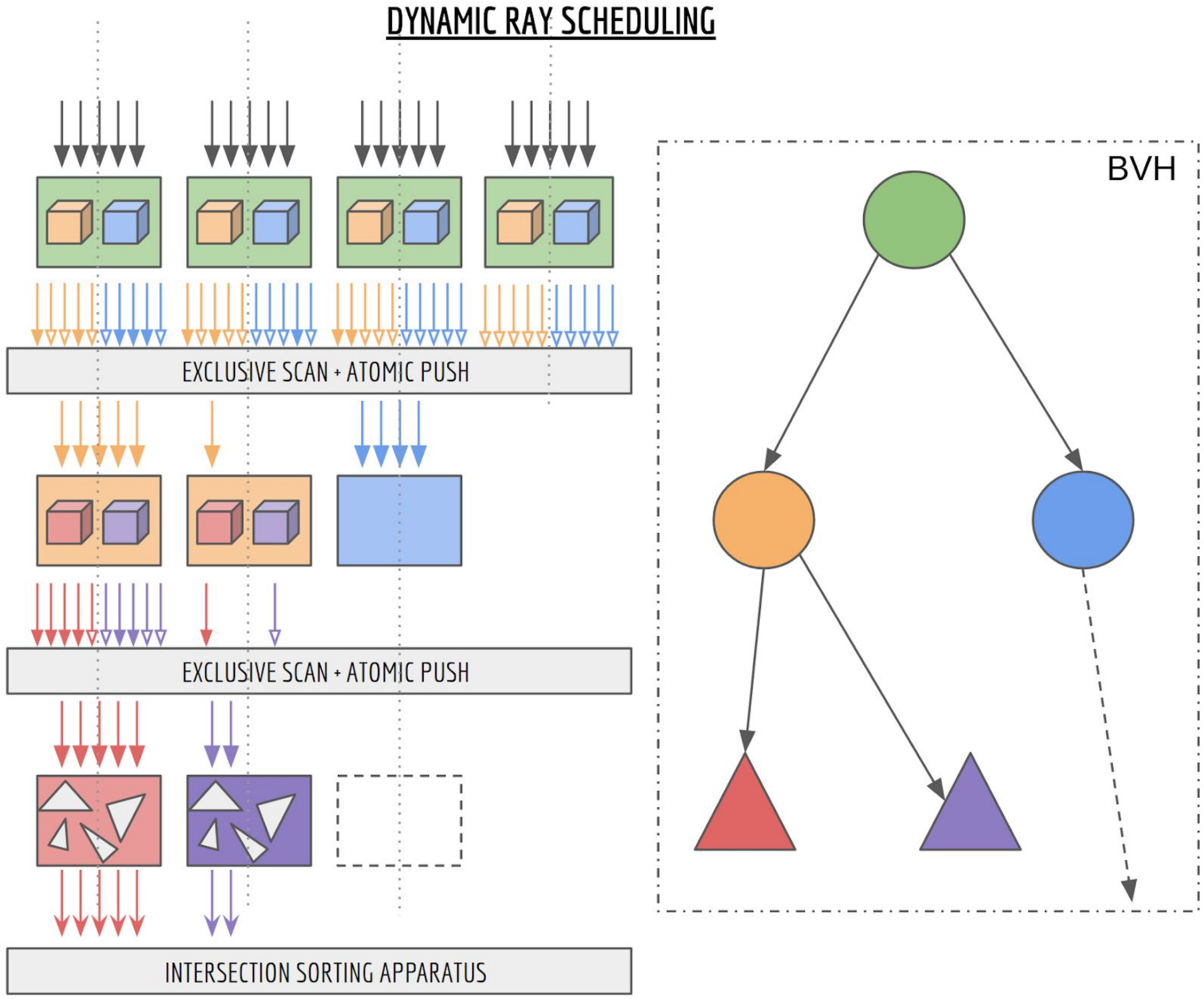

DYNAMIC RAY SCHEDULING
BVH
EXCLUSIVE SCAN + ATOMIC PUSH
EXCLUSIVE SCAN + ATOMIC PUSH
INTERSECTION SORTING APPARATUS

Figure: A flowchart illustrating the BVH traversal pipeline. Each horizontal set of blocks represents one execution of the kernel and processes one full set of rays through a **single level**. Each square represents one CUDA **block**. Note: The rays are colored to indicate a comparison with a specific **bounding box**, a ray with a **solid arrowhead** intersects the bounding box while a ray with an **empty arrowhead** doesn't. **Curved arrowheads** represent intersections with a **primitive**.

Each block processes a fixed number of rays (variable upto 1024). At the very beginning, the **BVH node** and the **bounding boxes** for the children are loaded into **shared** memory.

Each box is tested with each ray.

To compress the assignment of rays to child nodes, we use an **Shared Memory Array Compaction** (uses **Exclusive Scan)**. At this point we have a queue for each child node, but this queue is **only** for this particular block. There is a **race condition** between this block and others that are also processing the **same node**.

To handle this, we use **atomicAdd** to request space in the global queue for each child node. There is only one **atomicAdd** necessary per **block** per **child node**, so while it doesn't slow it down much, it seems like this is still one of the **bottlenecks**. But it can't be avoided.

The effect of this **bottleneck** reduces when the number of rays per block is increased.

Alternatively, If a block encounters a **Leaf Node**, the steps are very different:

It simply loads all the triangles into shared memory and then intersects each ray with each triangle.

Once an intersection is found, the intersection is placed into a buffer called the **Multi-intersection buffer**. This is required because a ray can intersect multiple nodes and thus parallel blocks can find intersections at different triangles. Only one of these intersections is correct (**minimum depth**). Unfortunately, because we do breadth-first traversal, it's hard to perform intersections one after the other, so we accumulate all intersections into a buffer and perform the **minimum operation** later.

There is one **atomicAdd** operation that is used to place the intersections into the buffer without race conditions. Fortunately, because the **atomicAdd** operates on a unique location for each ray, there are rarely any collisions.

Intersection Merging

The **Multi-intersection buffer** is now processed to find the least distance intersection for each ray. This process is called intersection merging.

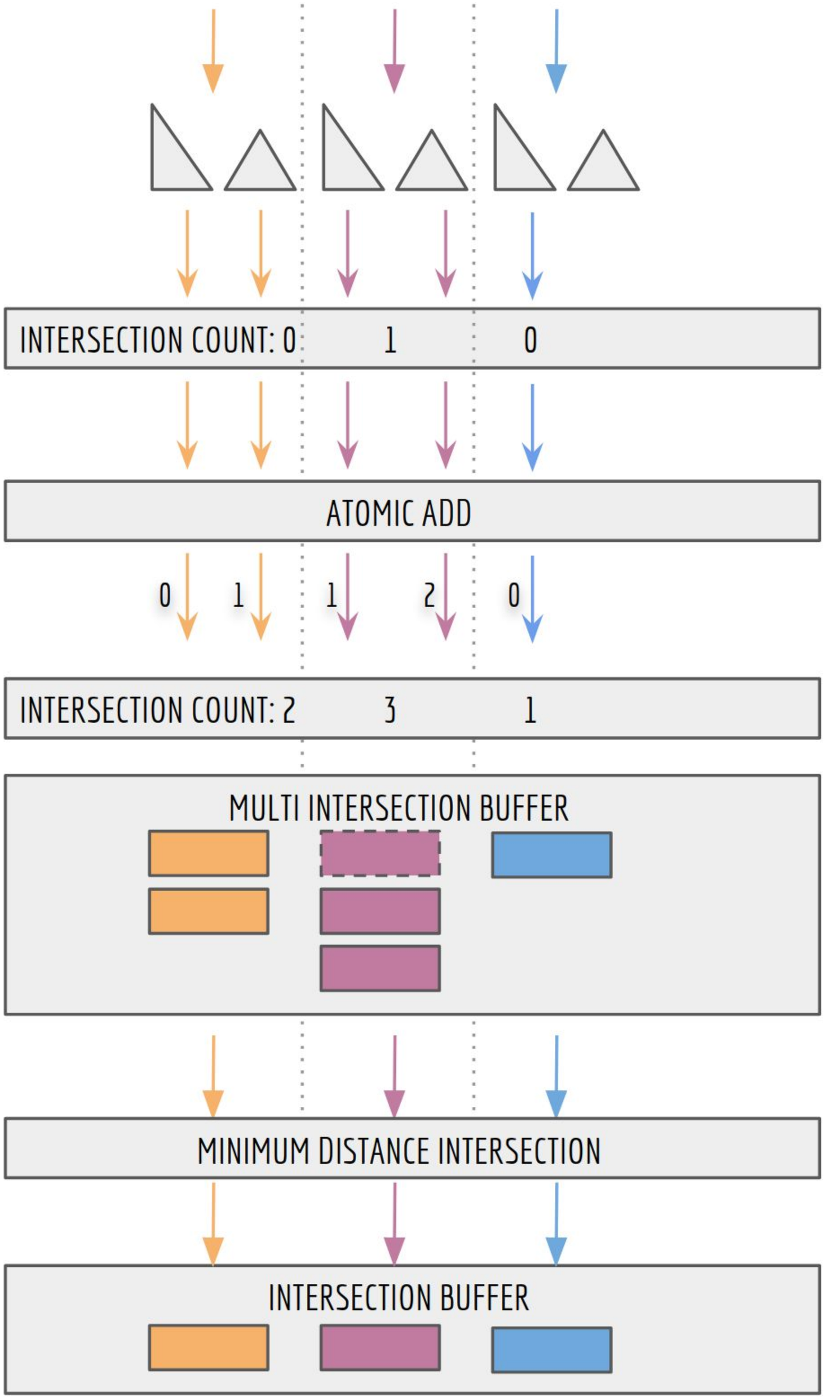


Figure: A representation of the **intersection sorting apparatus** that is used to avoid synchronization constructs due to the fact that a single ray can have **multiple intersections** that are detected by blocks running in parallel.

Direct Light Ray Generation

This step is used to create rays that are used to evaluate the direct light component at this point. The light in the scene is randomly sampled and connected to this point using a ray. The **potential light contribution** is computed at this stage and placed in the ray.

After this the ray intersection step is repeated for these new rays. If there is no intermediate intersection, this **potential light contribution** is added to the total contribution to this ray.

Note that all the light and importance data are carried with the ray and intersection constructs, to avoid having to randomly access any other global data structure.

Upon finding an intersection, it is important to note that other than the **light** contribution component, the rest of the intersection is **not modified**. This is because we don't want to advance the ray to a new point, but only check if the light contributes to this point.

Scene Ray Generation

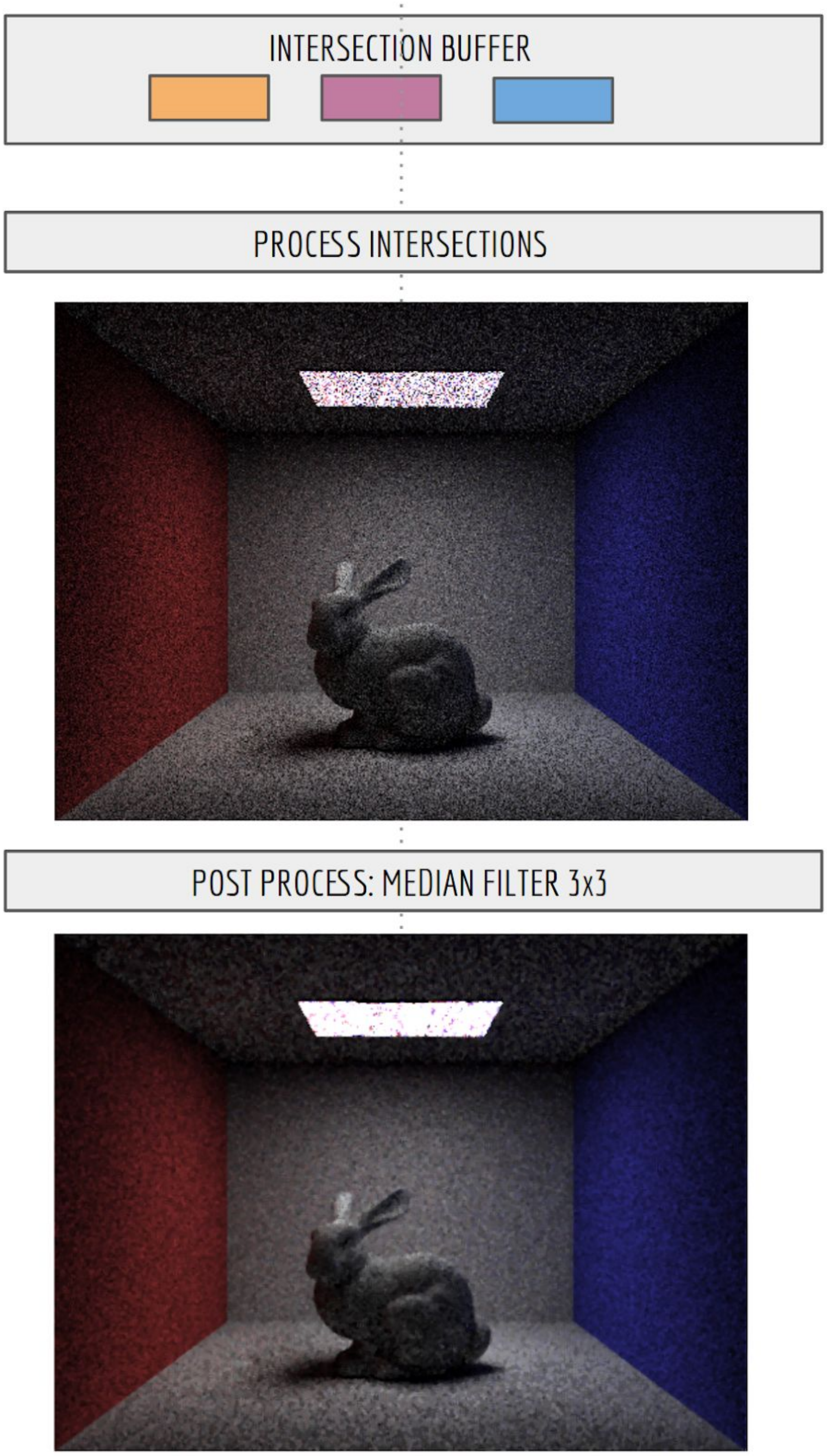
This process is mostly the same as **Direct Light Ray Generation** step, except that instead of sampling a light, the BSDF at that point is sampled.

We only fully support two BSDFs for now:

- **Mirror BSDF**: Use reflected ray.
- **Diffuse BSDF:** Randomly sampled ray.
- **(Experimental) Glass BSDF:** Refracted ray + Reflected ray based on Fresnel component.

We plan on adding a few others soon. At this point it is fairly simple to do so.

Image Reconstruction



INTERSECTION BUFFER

PROCESS INTERSECTIONS

POST PROCESS: MEDIAN FILTER 3x3

After the final intersections are computed (the **light** attribute accumulates the light at each bounce), then we just extract the radiance value from each intersection and place it in the right spot on the final image.

The final image, however, is extremely noisy if the number of samples are low. In order to produce a better image for real-time display, we apply a filter to the final image. While there are very complex filters which can reconstruct a full image from just one sample per pixel, implementing those is likely to be an entire project on it's own.

For now, we have implemented three simple filters:

- **Gaussian Blur** - Standard average. Blurs everything. Might as well render at lower resolution and higher sample count.
- **Bilateral Filter** - Compute weighted sum of neighbourhood, good for preserving edges. Bad at white noise removal.
- **Median Filter** - Compute median of neighborhood. Perfect for white noise, which is exactly what path traced images posses.

We finally settled on the **Median Filter**, which helps remove extreme noise to a great extent. Note that this post processing effect only comes into effect if the total samples is under a specific threshold.

Micro and Macro Optimizations

We now describe the loads of micro and macro optimizations that we did to improve the speed of the renderer. A lot of these were done in batches without a lot of checkpointing, so we are not entirely sure how well these worked, but together, these optimizations gave a **60%** reduction in total time taken.

BVH Traversal: "Talk first, Think later"

The original implementation of Dynamic Ray Scheduling (on the CPU) involved sorting the bounding boxes in order of distance from ray and then placing the ray into a special queue based on the **order** in which they intersect the ray. (Example: For a binary BVH, there are theoretically four queues, one for **left only**, one for **right only**, one for **left then right**, one for **right then left**.)

This is a problem in GPUs since we don't have nearly as much memory and we would like to keep these queues in shared memory for as long as possible. Adding to this problem, **Wide Trees** have multiple branches, which leads to way more possible orders (The paper describes a technique with which this can be greatly reduced, but it's still too many to fit into shared memory). So, we simply ignore the order and maintain just one queue per child node. This means we may get more redundant rays in the short term, but this optimization allows us to handle wider trees and a lot more rays per block, which gives us significant performance boosts.

We also describe a tradeoff mechanism below ("**Early Out**") that offsets the increase in rays due to this optimization, albeit in a non-deterministic way.

BVH Traversal: Early Out

This is a way of emulating the speculative traversal that is used in serial BVH traversal implementations. Since we cannot wait for one child node to return the full intersection before deciding whether to place the ray into the queue to the next child node, the same ray is likely processed several times even though only one intersection is correct.

To mitigate this (to an extent), we maintain a global array that holds the **minimum distance** discovered so far for that ray. If a bounding box or triangle intersection exceeds this minimum distance, that instance of the ray is invalidated to prevent further wasteful processing.

This helps to prevent wasteful processing if:

1. If the minimum distance triangle is found in the previous level.
2. If the minimum distance triangle is found in the same level but before the current node is processed (race condition).

There are race conditions that arise, but we do not use any atomics or synchronisation constructs because this array is simply a heuristic. The race condition, if any, cannot give rise to a value **lower** than the lowest distance currently found. Therefore, a race condition does not lead to incorrect execution but may lead to slower execution.

BVH Traversal: Block Striding

One major problem, as described before, with the intersection routine is the **atomicAdd** used to allocate space in a child node's **queue**. This contention actually slows the ray intersection procedure quite a bit. After some analysis we determined that the cause for this is that the blocks are scheduled linearly (For example the first **x** blocks processed all the rays for the first node in that level. The next **y** blocks process all rays for the second node). Given that the GPU roughly schedules the blocks in order, at any given time, almost all of the **Simultaneous Multiprocessors (SMs)** are executing the same node and attempting to access the same child nodes' queues.

We fixed this problem by striding the blocks. Instead of linearly indexing them, we are computing a **stride** index by taking square root of the total blocks in that level. We then create a **2D grid** of blocks **(stride, stride).** In the ray intersection procedure, the index is computed using the **y** coordinate as the minor and the **x** coordinate as the major.

This makes it so that adjacent indices correspond to very different nodes. Since each node points to a unique set of **child nodes**, there is no contention between them. This reduces the total ray intersection time by nearly **25%**, especially at levels **1** and **2**. Level **0** is unaffected because there is only one node, and beyond level **2**, there are so many nodes that most blocks are processing different nodes anyway.

BVH Traversal: Limited Blocks

The initial implementation involved a fixed number of blocks being generated for each level independent of the actual number of rays in each node. This meant that since theoretically each node could have all the rays in it, the number of blocks created was extremely large, e even though **99%** of the time, the block was simply reading a few values and quitting.

Even though this does not seem like a big deal, the sheer scale of the number of blocks (several **million**, when we only need a **couple hundred**) slowed the code down considerably.

To work around this, the blocks need to be **dynamically** assigned to nodes based on the number of rays in the queue for each node. This can be achieved fairly easily by using an **exclusive scan** to sum the ray counts at each node, followed by a parallel search to find the node index for each block. Since the number of nodes per level is usually in the ballpark of **10-1000**, this step is negligible compared to the rest of the ray intersection procedure.

Micro-optimizations

Totally these give about a **10-15%** improvements in speed.

1. Replace **sqrt()** with **__sqrtf(), cos()** and **sin()** with **__sincosf()** etc.. i.e their approximate FP32 counterparts.
2. Always load the data structures to memory first and avoiding the use of pointers since it leads to multiple accesses. This gave a decent boost.

Performance Analysis and Results

Path Tracer Render Times

From the graph we can see that our path tracer gives us a speed improvement of roughly around **90x** to **100x** the **multi-threaded CPU** ray tracer. As mentioned before, we use the **reference implementation of Scotty3D** as our baseline and we use the performance benchmarks provided by the **15-462** course with a fully functioning **BVH**, running on a modern **8-core Xeon** (GHC machines).

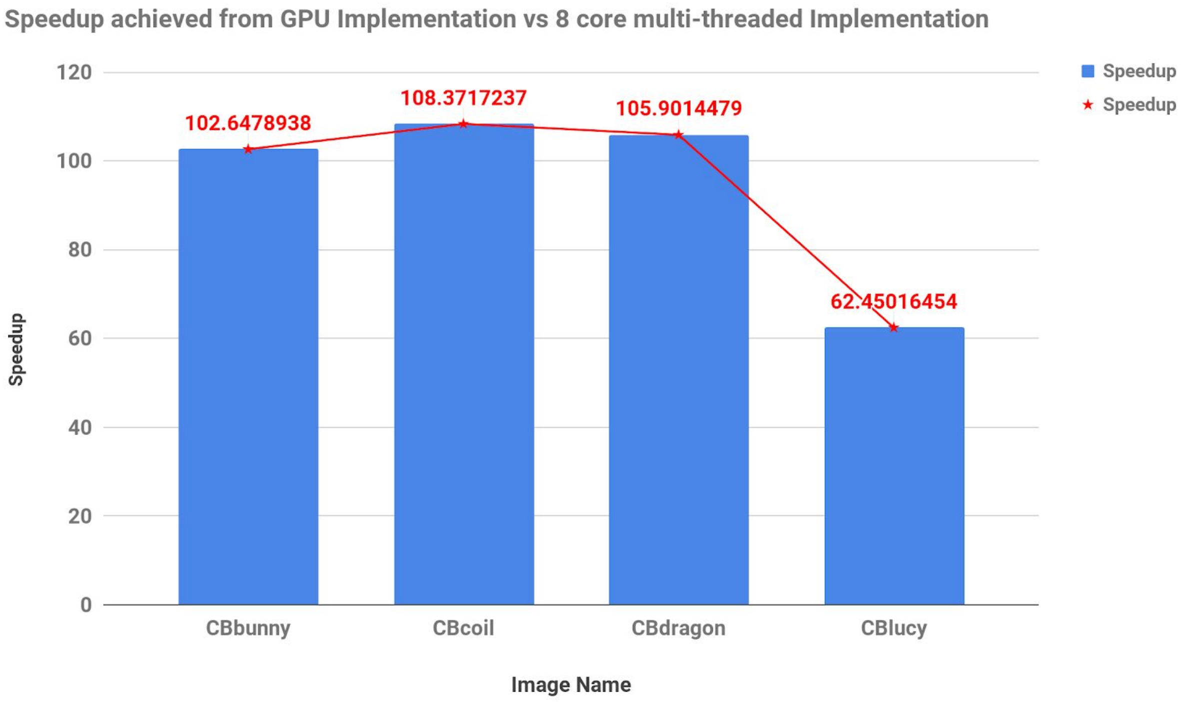
We also note that the images provided by the reference implementation were **640x480** by default, out of which (by manual estimation), only **40%** of the scene pixels were covered and the CUDA renderer created a **256x256** image with **90%** coverage of the **same scene**

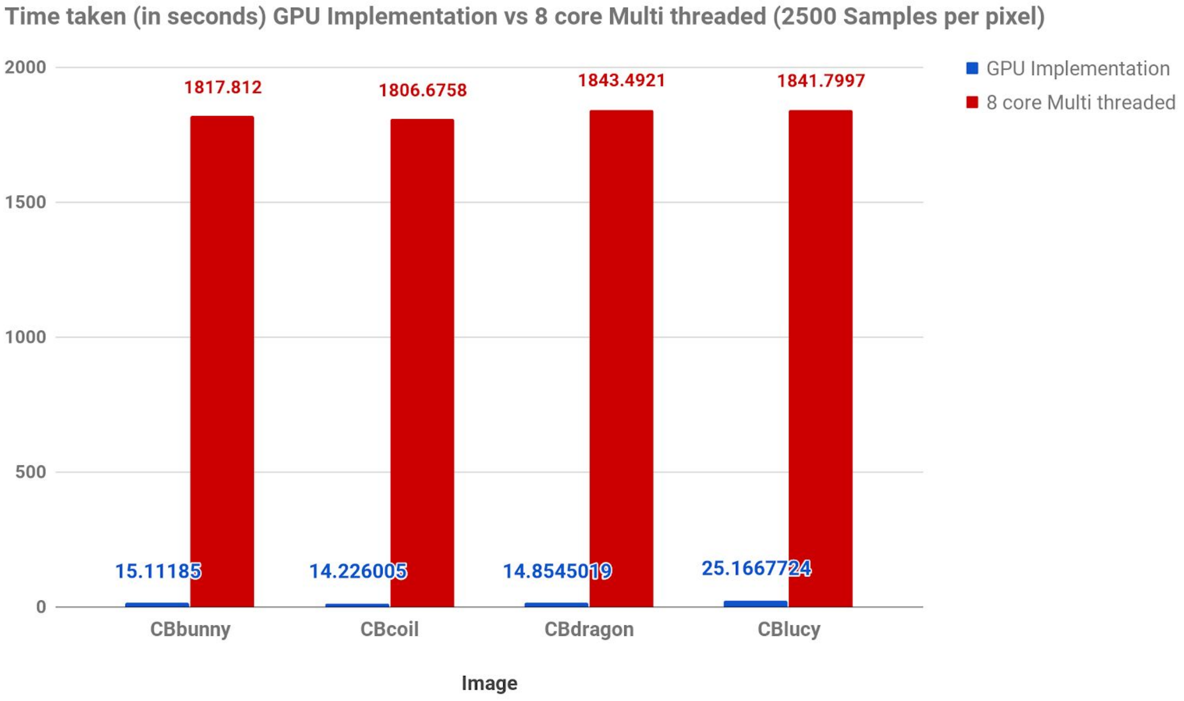
(Unfortunately our renderer is restricted to powers of 2 and equal width and height since this gives a computation boost). Thus the values provided by our renderer were reduced by a factor of **1.17** to account for this disparity in useful rays.

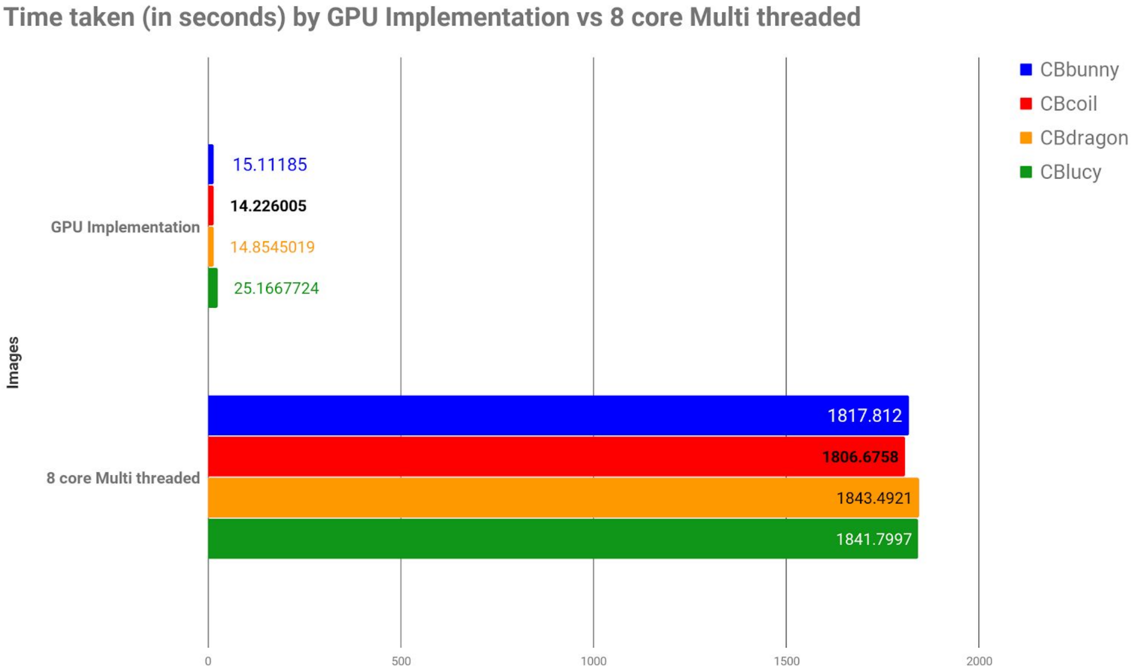
The **CBlucy** scene alone was too large for our GPU, so we had to reduce our **Tree width** to **4**. Thus the running time was suboptimal.

Notes:

We understand that it is possible that we miss out on the extra computation, however small, performed by the reference renderer to handle the black areas (boundary rays), but we argue that this is a negligible part of the total time taken, since they all do not intersect the root node.




Time taken (in seconds) GPU Implementation vs 8 core Multi threaded (2500 Samples per pixel)
2000
1500
1000
500
0
1817.812
1806.6758
1843.4921
1841.7997
15.11185
14.226005
14.8545019
25.1667724
CBbunny
CBcoil
CBdragon
CBlucy
Image
GPU Implementation
8 core Multi threaded


Time taken (in seconds) by GPU Implementation vs 8 core Multi threaded
CBbunny
CBcoil
CBdragon
CBlucy
GPU Implementation
8 core Multi threaded
Images
15.11185
14.226005
14.8545019
25.1667724
1817.812
1806.6758
1843.4921
1841.7997
0
500
1000
1500
2000

Analysis: Variation with Total Rays per Traversal

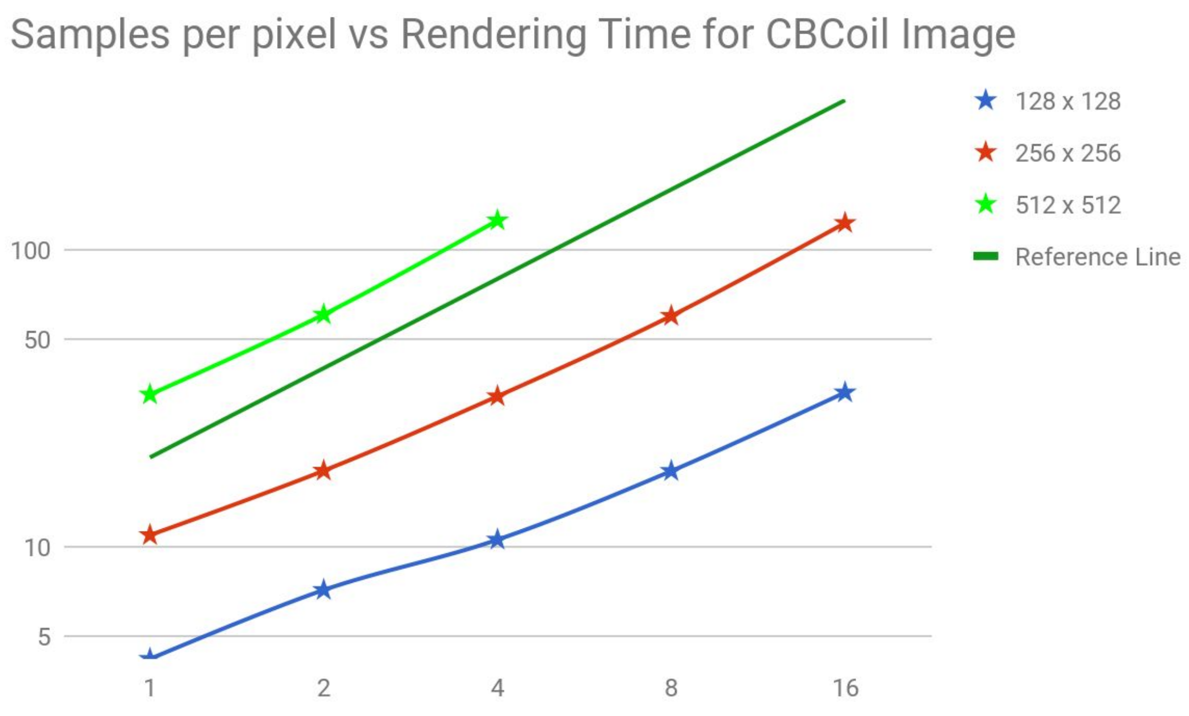


Figure: Rendering time w.r.t total rays per traversal. The graph is represented as a **log-log** graph (both axes are log scale). The **green** reference line has a slope of **1**. The **red** and **blue** lines have lower slope than the reference line which implies that they have **sublinear** growth

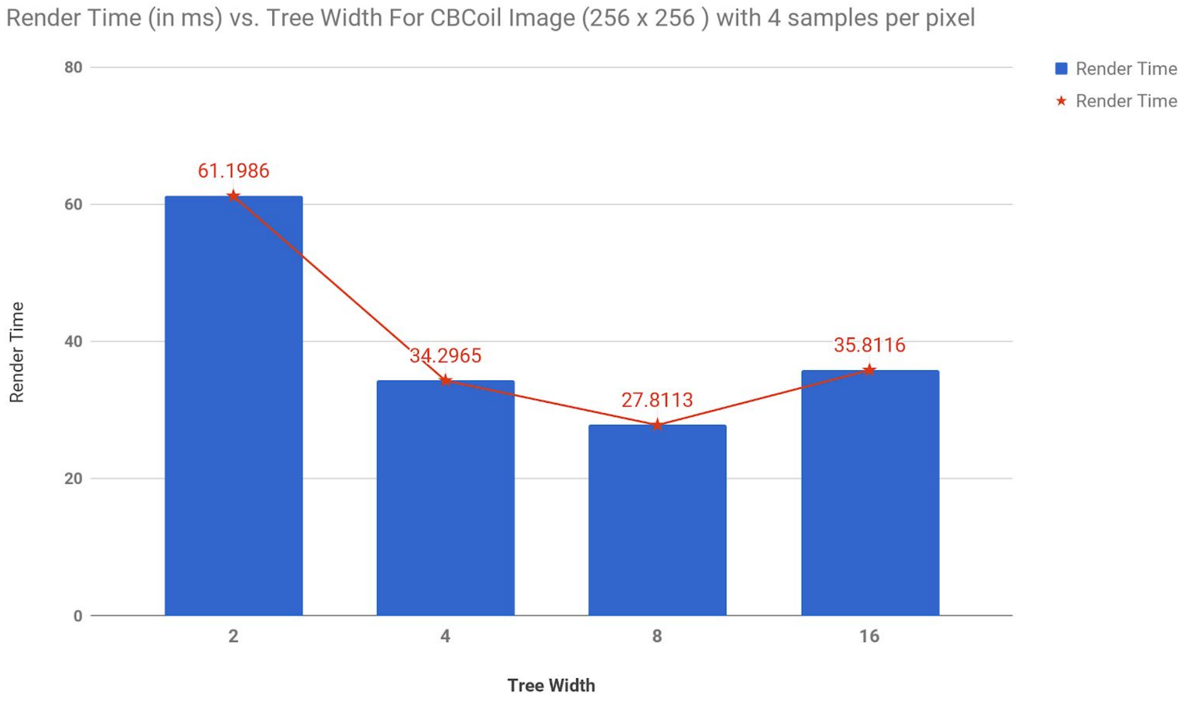
The **Total Rays per Traversal** is computed as the product of **width, height** and **samples per pixel**. This represents the total number of rays that are in the queue at the top level initially.

Pushing more rays through in a single pass lowers the per-ray time. From the plot above, we see that increasing the number of rays by a factor of **4** only increases the total time by about **3**.

So what prevents us from pushing through more samples by increasing samples per pixel or image dimensions? That would be **device memory.** Physical memory needs to be allocated to store the queue of rays (we technically need **two queues** one for input and one for output, which are switched with each other every level). This memory scales linearly with the number of rays.

By our analysis, at **1 million rays** and **4-way** BVHs (or alternatively **500,000 rays** and **8-way BVHs**), we use 6GB of device memory. This is one example of a situation where more memory directly increases performance.

Analysis: Variation with Tree Widths

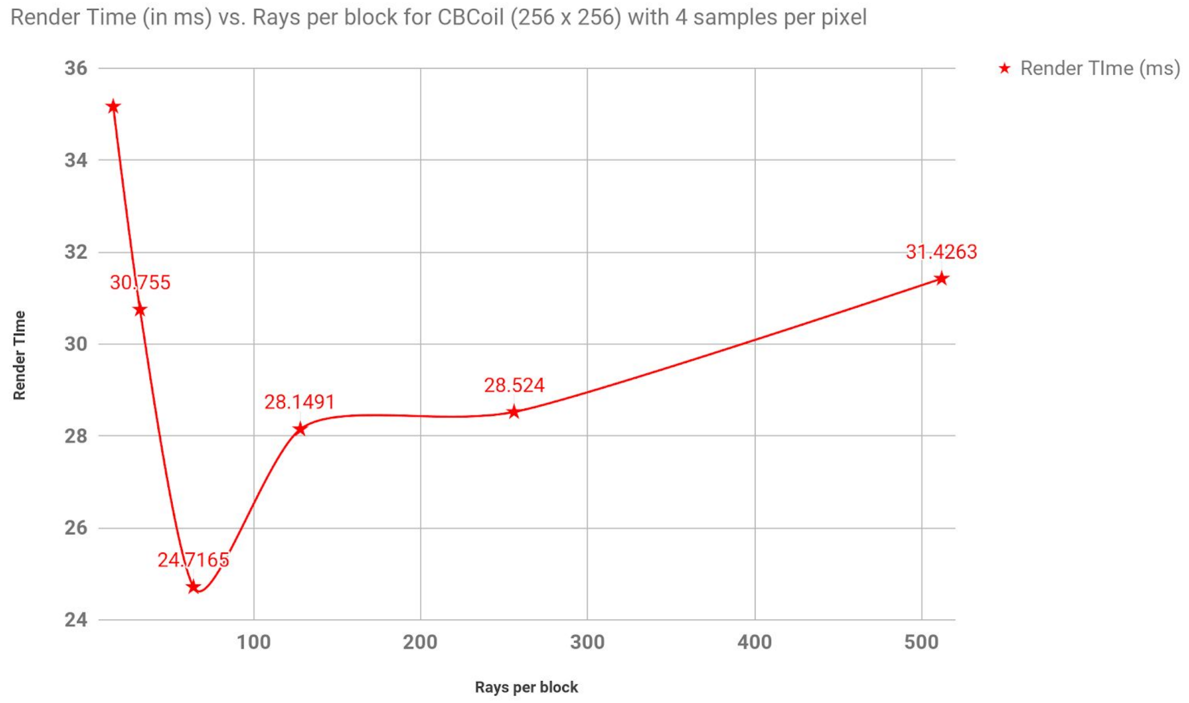


This is a very interesting hyperparameter because there are two opposing forces that trade-off at some point. The final optimal value turned out to be an **8-way BVH.**

1. Higher tree branching factor means fewer nodes and fewer levels, thus reducing the time taken to perform a full traversal. Thus **higher is better**
2. But higher tree branching factor also means that each node has more queues, thus more atomicAdd operations and more exclusive scans are performed. Thus **lower is better.**

These two forces tradeoff at **8 branches per node.**

Analysis: Variation with Rays per Block



We see that the peak throughput is at RPB **64.**

One explanation for this is that the code is **compute bound**. Given that an **SM** can only execute **64** threads at a single point in time (**64 ALUs**), it makes sense that overloading the **SM** will not give any significant boost to processing time.

Analysis: Primary vs Secondary Bounces

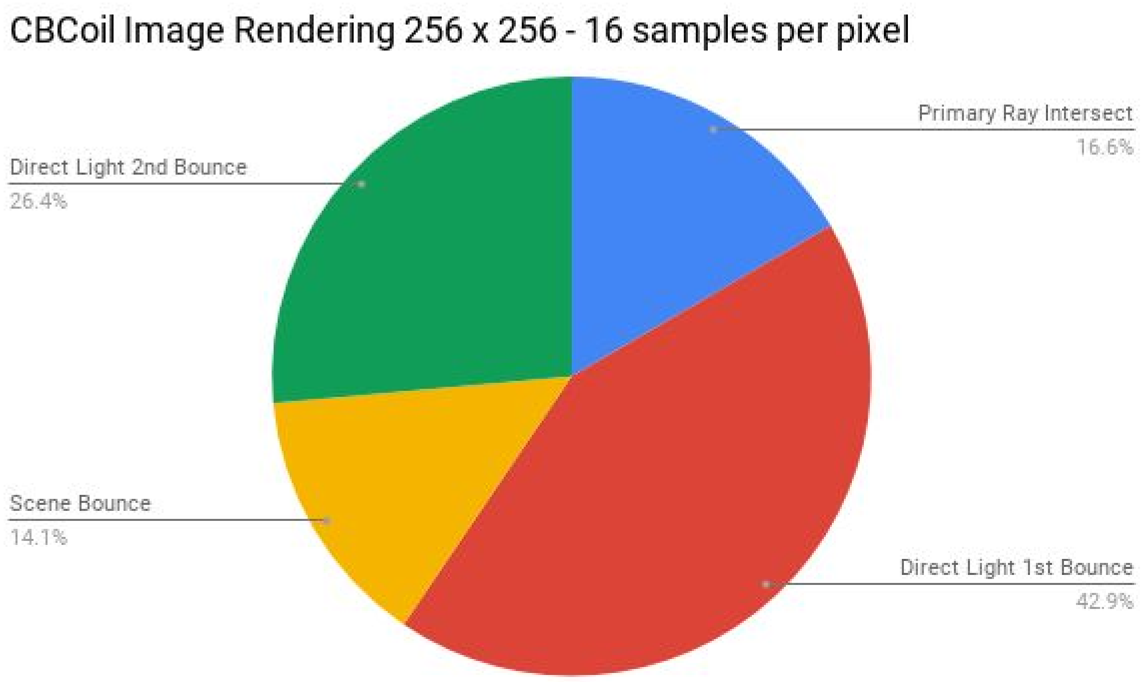
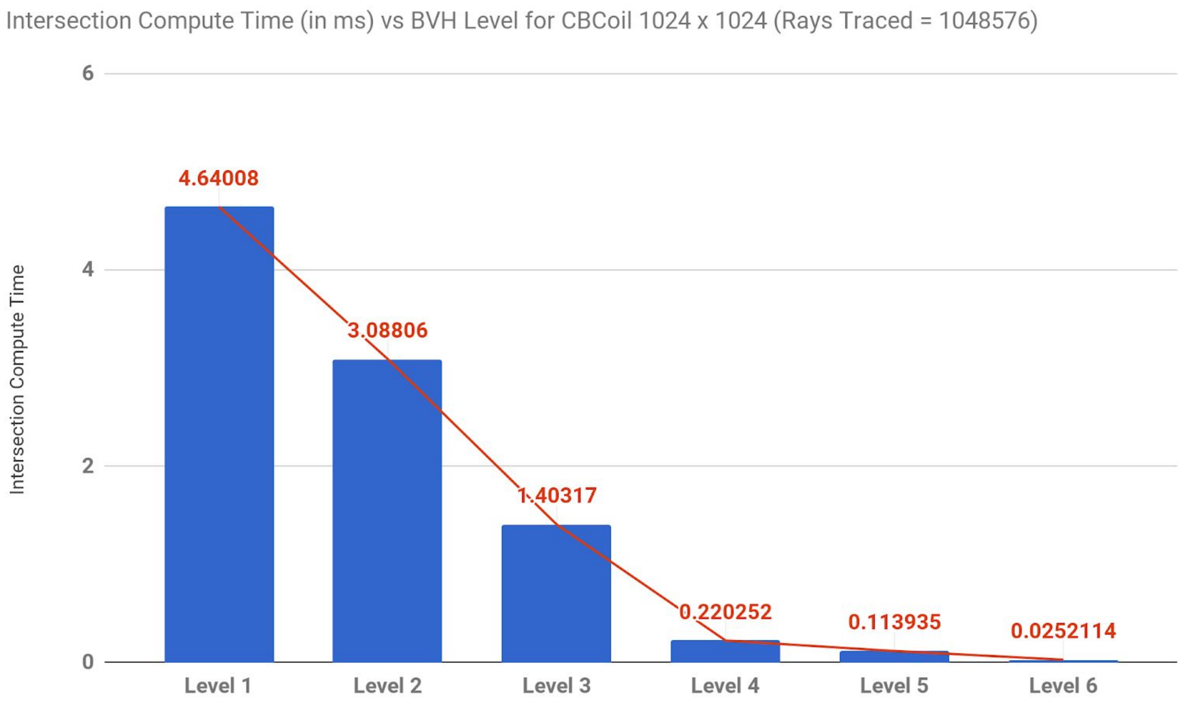


Figure: A pie chart illustrating the times taken by each ray-scene intersection. Note that the direct light bounces involve **two** ray-scene intersections because we take **two** area light samples.

One important thing to note here is that **Secondary** throughput (**Direct Light 2nd Bounce + Scene Bounce**) is not much lower than the **Primary** throughput(**Direct Light 1st Bounce + Primary Ray Intersect**). This is an important feature of our renderer. Since "sorting" the rays at each stage is an integral part of the algorithm, every block usually operates at full capacity with **relevant** rays. It should be noted that it's possible to maybe get a significant boost in **primary** throughput if we used a different method for **primary** rays alone. (It could even be rasterization)

Analysis: Variation within Levels



This is a fairly straightforward analysis. The top 3 levels tend to have a lot of mostly **unavoidable** contention (due to fewer nodes). Below this, we have **fewer rays** (some of them don't intersect anything) *and* **less contention** (because of more nodes), so the time taken drops off very quickly.

Note that at the very end (Level 6), there are actually very few nodes because only a few paths in the tree reach the max path length. The most number of nodes tend to be in Levels 3, 4 and 5.

As a side node, we found that with **4-way** BVH build with a max node size of **20** triangles, there are usually not more than 10 levels and because of the falloff at higher levels, there is only a small increase in total time with increasing scene complexity.

Comparison with State-of-the-art Renderers

We haven't been able to render the exact same scene with other renderers, so we're using rough estimates of their ray tracer throughputs than rendering times for a comparison.

For reference, our renderer gives an output of around **70-82** MRPS depending on the scene.

Intel's **Embree** is notable for being a highly optimized CPU renderer that usually achieves about a **100** MRPS (million rays per second) on a single Intel CPU (multi-threaded). Our path tracer is in the same range, although slightly lower sometimes.

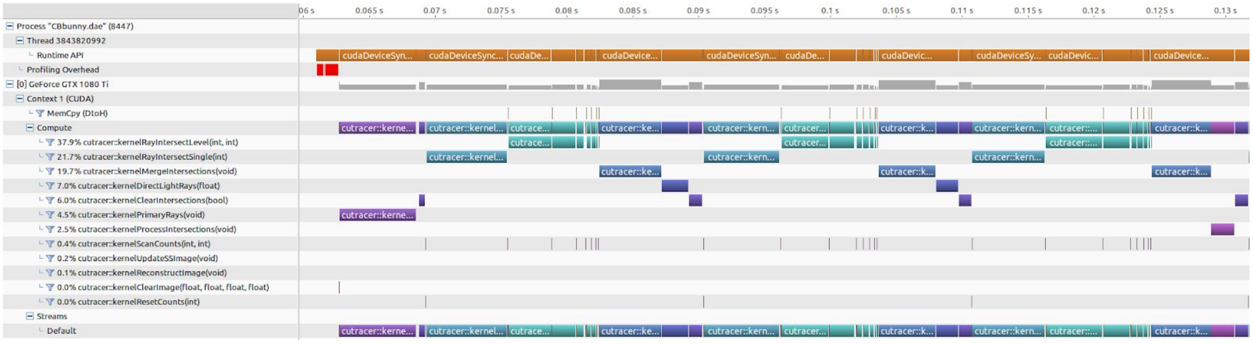
A popular research oriented renderer **mitsuba** is fairly well optimized, though not as much as **Embree**. In our experiments with mitsuba, we estimated a stable **11** MRPS on a scene with the full resolution bunny mesh (we used mitsuba's internal counter of rays traced to get a robust estimate). This is still 6-10 times slower than our GPU implementation.

On the flip side, a robust **GPU** path tracer and the current state-of-the-art ray tracer is **NVIDIA**'s **Optix** (it's technically only a 'framework' / 'SDK'), which gives **300-350** MRPS on a GTX TITAN X. This will be the number to beat in the future after we fix utilization issues with our current version.

For a full analysis of optimality and utilization, see the next section.

Optimality and Bottlenecks

We used the **NVIDIA Visual Profiler** to analyze our performance numbers.



06 s
0.065 s
0.07 s
0.075 s
0.08 s
0.085 s
0.09 s
0.095 s
0.1 s
0.105 s
0.11 s
0.115 s
0.12 s
0.125 s
0.13 s
Process "CBbunny.dae" (8447)
Thread 3843820992
Runtime API
Profiling Overhead
[0] GeForce GTX 1080 Ti
Context 1 (CUDA)
MemCpy (DtoH)
Compute
37.9% cutracer::kernelRayIntersectLevel(int, int)
21.7% cutracer::kernelRayIntersectSingle(int)
19.7% cutracer::kernelMergeIntersections(void)
7.0% cutracer::kernelDirectLightRays(float)
6.0% cutracer::kernelClearIntersections(bool)
4.5% cutracer::kernelPrimaryRays(void)
2.5% cutracer::kernelProcessIntersections(void)
0.4% cutracer::kernelScanCounts(int, int)
0.2% cutracer::kernelUpdateSSImage(void)
0.1% cutracer::kernelReconstructImage(void)
0.0% cutracer::kernelClearImage(float, float, float, float)
0.0% cutracer::kernelResetCounts(int)
Streams
Default

Figure: Timeline of execution and GPU Usage for every kernel (The gray bars represent GPU usage)

(Please zoom in if the above image is not visible)

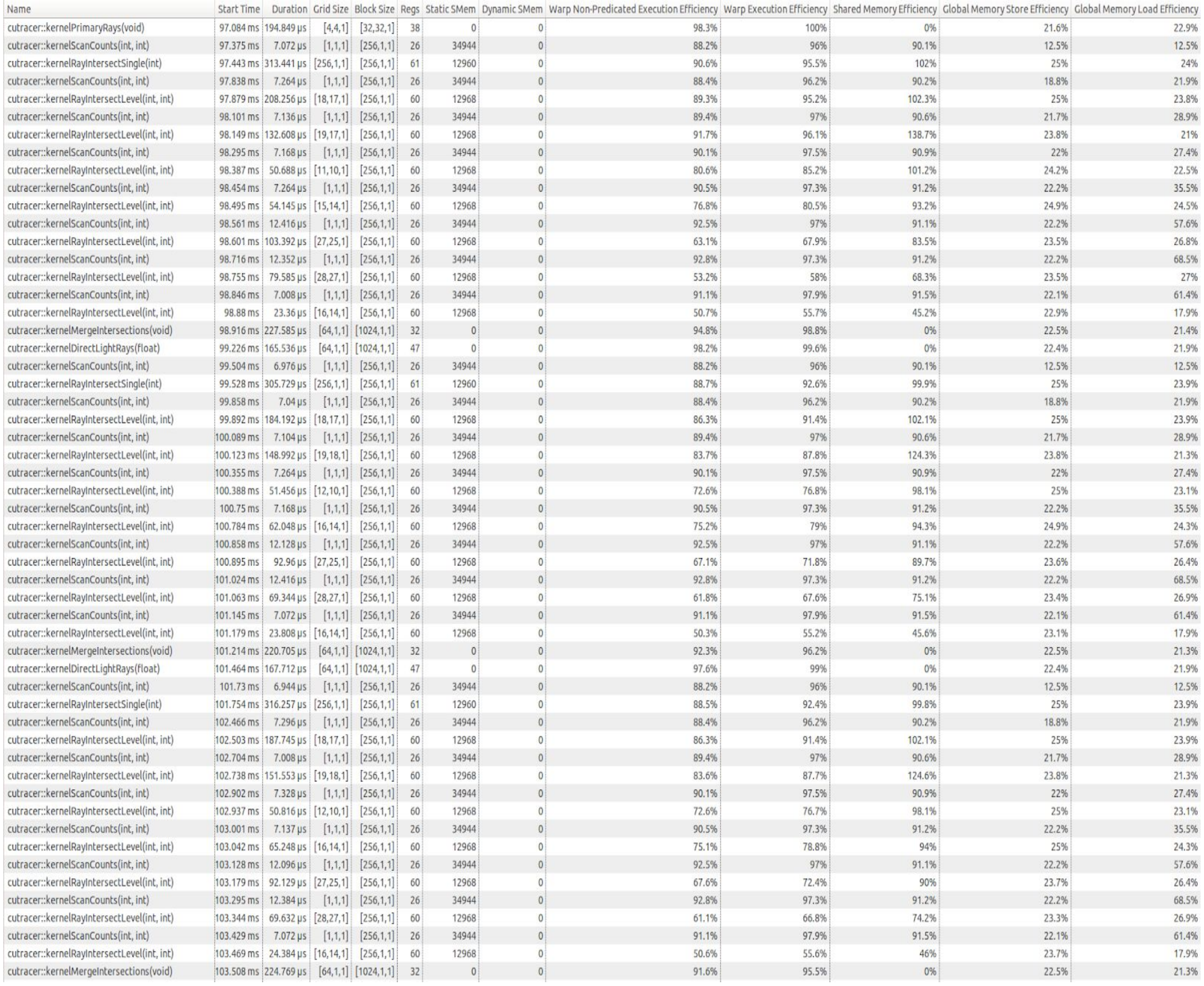
At first glance, it appears that there is still a lot of room for improvement since the main ray intersection routines only use about **50%** of the GPU.

We also see that the **Merge Intersections** phase actually takes about half as much time as the full **BVH traversal** itself. This is a very long time allocation for something that is supposed to be a small post process step. In the future, we will be addressing this issue. This is likely to require some algorithmic change or code optimization because the way it is written right now, it has full **100%** utilization with extremely high warp execution efficiency.

As for the BVH Traversal kernels (**kernelRayIntersectSingle()** for the top level and **kernelRayIntersectLevel()** for the other levels), we will need to further investigate them to understand their limitations.

For this, we use the **per-kernel analysis tool**:

(Next page)



Name	Start Time	Duration	Grid Size	Block Size	Regs	Static SMem	Dynamic SMem	Warp Non-Predicated Execution Efficiency	Warp Execution Efficiency	Shared Memory Efficiency	Global Memory Store Efficiency	Global Memory Load Efficiency
cutracer::kernelPrimaryRays(void)	97.084 ms	194.849 μs	[4,4,1]	[32,32,1]	38	0	0	98.3%	100%	0%	21.6%	22.9%
cutracer::kernelScanCounts(int, int)	97.375 ms	7.072 μs	[1,1,1]	[256,1,1]	26	34944	0	88.2%	96%	90.1%	12.5%	12.5%
cutracer::kernelRayIntersectSingle(int)	97.443 ms	313.441 μs	[256,1,1]	[256,1,1]	61	12960	0	90.6%	95.5%	102%	25%	24%
cutracer::kernelScanCounts(int, int)	97.838 ms	7.264 μs	[1,1,1]	[256,1,1]	26	34944	0	88.4%	96.2%	90.2%	18.8%	21.9%
cutracer::kernelRayIntersectLevel(int, int)	97.879 ms	208.256 μs	[18,17,1]	[256,1,1]	60	12968	0	89.3%	95.2%	102.3%	25%	23.8%
cutracer::kernelScanCounts(int, int)	98.101 ms	7.136 μs	[1,1,1]	[256,1,1]	26	34944	0	89.4%	97%	90.6%	21.7%	28.9%
cutracer::kernelRayIntersectLevel(int, int)	98.149 ms	132.608 μs	[19,17,1]	[256,1,1]	60	12968	0	91.7%	96.1%	138.7%	23.8%	21%
cutracer::kernelScanCounts(int, int)	98.295 ms	7.168 μs	[1,1,1]	[256,1,1]	26	34944	0	90.1%	97.5%	90.9%	22%	27.4%
cutracer::kernelRayIntersectLevel(int, int)	98.387 ms	50.688 μs	[11,10,1]	[256,1,1]	60	12968	0	80.6%	85.2%	101.2%	24.2%	22.5%
cutracer::kernelScanCounts(int, int)	98.454 ms	7.264 μs	[1,1,1]	[256,1,1]	26	34944	0	90.5%	97.3%	91.2%	22.2%	35.5%
cutracer::kernelRayIntersectLevel(int, int)	98.495 ms	54.145 μs	[15,14,1]	[256,1,1]	60	12968	0	76.8%	80.5%	93.2%	24.9%	24.5%
cutracer::kernelScanCounts(int, int)	98.561 ms	12.416 μs	[1,1,1]	[256,1,1]	26	34944	0	92.5%	97%	91.1%	22.2%	57.6%
cutracer::kernelRayIntersectLevel(int, int)	98.601 ms	103.392 μs	[27,25,1]	[256,1,1]	60	12968	0	63.1%	67.9%	83.5%	23.5%	26.8%
cutracer::kernelScanCounts(int, int)	98.716 ms	12.352 μs	[1,1,1]	[256,1,1]	26	34944	0	92.8%	97.3%	91.2%	22.2%	68.5%
cutracer::kernelRayIntersectLevel(int, int)	98.755 ms	79.585 μs	[28,27,1]	[256,1,1]	60	12968	0	53.2%	58%	68.3%	23.5%	27%
cutracer::kernelScanCounts(int, int)	98.846 ms	7.008 μs	[1,1,1]	[256,1,1]	26	34944	0	91.1%	97.9%	91.5%	22.1%	61.4%
cutracer::kernelRayIntersectLevel(int, int)	98.88 ms	23.36 μs	[16,14,1]	[256,1,1]	60	12968	0	50.7%	55.7%	45.2%	22.9%	17.9%
cutracer::kernelMergeIntersections(void)	98.916 ms	227.585 μs	[64,1,1]	[1024,1,1]	32	0	0	94.8%	98.8%	0%	22.5%	21.4%
cutracer::kernelDirectLightRays(float)	99.226 ms	165.536 μs	[64,1,1]	[1024,1,1]	47	0	0	98.2%	99.6%	0%	22.4%	21.9%
cutracer::kernelScanCounts(int, int)	99.504 ms	6.976 μs	[1,1,1]	[256,1,1]	26	34944	0	88.2%	96%	90.1%	12.5%	12.5%
cutracer::kernelRayIntersectSingle(int)	99.528 ms	305.729 μs	[256,1,1]	[256,1,1]	61	12960	0	88.7%	92.6%	99.9%	25%	23.9%
cutracer::kernelScanCounts(int, int)	99.858 ms	7.04 μs	[1,1,1]	[256,1,1]	26	34944	0	88.4%	96.2%	90.2%	18.8%	21.9%
cutracer::kernelRayIntersectLevel(int, int)	99.892 ms	184.192 μs	[18,17,1]	[256,1,1]	60	12968	0	86.3%	91.4%	102.1%	25%	23.9%
cutracer::kernelScanCounts(int, int)	100.089 ms	7.104 μs	[1,1,1]	[256,1,1]	26	34944	0	89.4%	97%	90.6%	21.7%	28.9%
cutracer::kernelRayIntersectLevel(int, int)	100.123 ms	148.992 μs	[19,18,1]	[256,1,1]	60	12968	0	83.7%	87.8%	124.3%	23.8%	21.3%
cutracer::kernelScanCounts(int, int)	100.355 ms	7.264 μs	[1,1,1]	[256,1,1]	26	34944	0	90.1%	97.5%	90.9%	22%	27.4%
cutracer::kernelRayIntersectLevel(int, int)	100.388 ms	51.456 μs	[12,10,1]	[256,1,1]	60	12968	0	72.6%	76.8%	98.1%	25%	23.1%
cutracer::kernelScanCounts(int, int)	100.75 ms	7.168 μs	[1,1,1]	[256,1,1]	26	34944	0	90.5%	97.3%	91.2%	22.2%	35.5%
cutracer::kernelRayIntersectLevel(int, int)	100.784 ms	62.048 μs	[16,14,1]	[256,1,1]	60	12968	0	75.2%	79%	94.3%	24.9%	24.3%
cutracer::kernelScanCounts(int, int)	100.858 ms	12.128 μs	[1,1,1]	[256,1,1]	26	34944	0	92.5%	97%	91.1%	22.2%	57.6%
cutracer::kernelRayIntersectLevel(int, int)	100.895 ms	92.96 μs	[27,25,1]	[256,1,1]	60	12968	0	67.1%	71.8%	89.7%	23.6%	26.4%
cutracer::kernelScanCounts(int, int)	101.024 ms	12.416 μs	[1,1,1]	[256,1,1]	26	34944	0	92.8%	97.3%	91.2%	22.2%	68.5%
cutracer::kernelRayIntersectLevel(int, int)	101.063 ms	69.344 μs	[28,27,1]	[256,1,1]	60	12968	0	61.8%	67.6%	75.1%	23.4%	26.9%
cutracer::kernelScanCounts(int, int)	101.145 ms	7.072 μs	[1,1,1]	[256,1,1]	26	34944	0	91.1%	97.9%	91.5%	22.1%	61.4%
cutracer::kernelRayIntersectLevel(int, int)	101.179 ms	23.808 μs	[16,14,1]	[256,1,1]	60	12968	0	50.3%	55.2%	45.6%	23.1%	17.9%
cutracer::kernelMergeIntersections(void)	101.214 ms	220.705 μs	[64,1,1]	[1024,1,1]	32	0	0	92.3%	96.2%	0%	22.5%	21.3%
cutracer::kernelDirectLightRays(float)	101.464 ms	167.712 μs	[64,1,1]	[1024,1,1]	47	0	0	97.6%	99%	0%	22.4%	21.9%
cutracer::kernelScanCounts(int, int)	101.73 ms	6.944 μs	[1,1,1]	[256,1,1]	26	34944	0	88.2%	96%	90.1%	12.5%	12.5%
cutracer::kernelRayIntersectSingle(int)	101.754 ms	316.257 μs	[256,1,1]	[256,1,1]	61	12960	0	88.5%	92.4%	99.8%	25%	23.9%
cutracer::kernelScanCounts(int, int)	102.466 ms	7.296 μs	[1,1,1]	[256,1,1]	26	34944	0	88.4%	96.2%	90.2%	18.8%	21.9%
cutracer::kernelRayIntersectLevel(int, int)	102.503 ms	187.745 μs	[18,17,1]	[256,1,1]	60	12968	0	86.3%	91.4%	102.1%	25%	23.9%
cutracer::kernelScanCounts(int, int)	102.704 ms	7.008 μs	[1,1,1]	[256,1,1]	26	34944	0	89.4%	97%	90.6%	21.7%	28.9%
cutracer::kernelRayIntersectLevel(int, int)	102.738 ms	151.553 μs	[19,18,1]	[256,1,1]	60	12968	0	83.6%	87.7%	124.6%	23.8%	21.3%
cutracer::kernelScanCounts(int, int)	102.902 ms	7.328 μs	[1,1,1]	[256,1,1]	26	34944	0	90.1%	97.5%	90.9%	22%	27.4%
cutracer::kernelRayIntersectLevel(int, int)	102.937 ms	50.816 μs	[12,10,1]	[256,1,1]	60	12968	0	72.6%	76.7%	98.1%	25%	23.1%
cutracer::kernelScanCounts(int, int)	103.001 ms	7.137 μs	[1,1,1]	[256,1,1]	26	34944	0	90.5%	97.3%	91.2%	22.2%	35.5%
cutracer::kernelRayIntersectLevel(int, int)	103.042 ms	65.248 μs	[16,14,1]	[256,1,1]	60	12968	0	75.1%	78.8%	94%	25%	24.3%
cutracer::kernelScanCounts(int, int)	103.128 ms	12.096 μs	[1,1,1]	[256,1,1]	26	34944	0	92.5%	97%	91.1%	22.2%	57.6%
cutracer::kernelRayIntersectLevel(int, int)	103.179 ms	92.129 μs	[27,25,1]	[256,1,1]	60	12968	0	67.6%	72.4%	90%	23.7%	26.4%
cutracer::kernelScanCounts(int, int)	103.295 ms	12.384 μs	[1,1,1]	[256,1,1]	26	34944	0	92.8%	97.3%	91.2%	22.2%	68.5%
cutracer::kernelRayIntersectLevel(int, int)	103.344 ms	69.632 μs	[28,27,1]	[256,1,1]	60	12968	0	61.1%	66.8%	74.2%	23.3%	26.9%
cutracer::kernelScanCounts(int, int)	103.429 ms	7.072 μs	[1,1,1]	[256,1,1]	26	34944	0	91.1%	97.9%	91.5%	22.1%	61.4%
cutracer::kernelRayIntersectLevel(int, int)	103.469 ms	24.384 μs	[16,14,1]	[256,1,1]	60	12968	0	50.6%	55.6%	46%	23.7%	17.9%
cutracer::kernelMergeIntersections(void)	103.508 ms	224.769 μs	[64,1,1]	[1024,1,1]	32	0	0	91.6%	95.5%	0%	22.5%	21.3%

(Please zoom in if the above image is not visible)

As the above profile suggests, the main kernels have **low global memory load/store** efficiency. This is likely because of the bad access patterns that can arise from certain random accesses. These can be fixed by rearranging some data constructs. However, time constraints prevent us from applying those optimizations before the deadline. We will be fixing those errors in the **future** and aiming for **100%** under every column to improve throughput.

We note that because of the way our algorithm works, the BVH traversal kernels at the bottom half (**Secondary Rays**) have only slightly lower **Warp Execution Efficiency** than the top half (**Primary Rays**). In most other CUDA implementations which work on a per-ray basis, they suffer from alarmingly low Warp Efficiency for secondary rays.

Conclusions

In conclusion, we have shown in this report that the right combination of **Wide Trees** and **Dynamic Ray Scheduling** can make a very fast path tracer. We experimented with various hyperparameters and selected the best of each to further optimize our renderer.

We have also learnt that most CPU algorithms rarely map well to the GPU, and in most cases a new algorithm needs to be devised, one that follows the spirit of the original one, but whose primitive operations are better suited to the GPU's skillset.

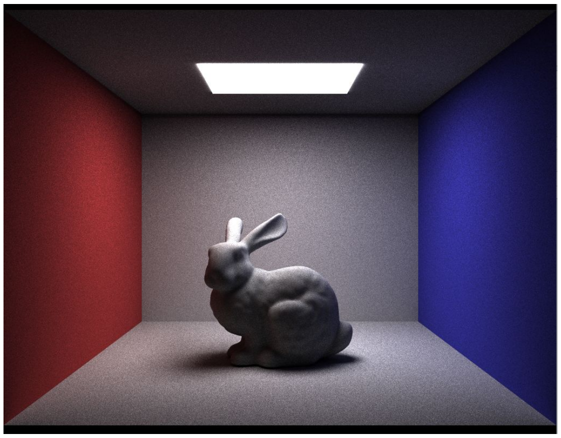
For our **demo** we have also created a real-time version of the renderer (which was our original idea) which uses some elementary post processing to reduce noise.

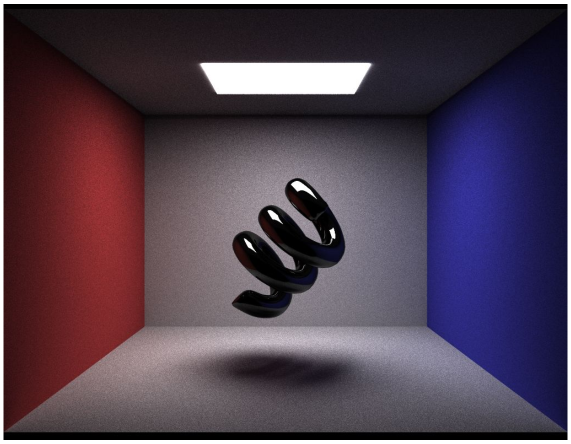
Finally, we also show that our renderer is not as optimal as it could be but with a lot more effort. It only uses around 60% of the GPU and it's possible, by smartly rearranging the data around to reduce bandwidth consumption, we could make our raytracer even faster and some day, truly interactive.

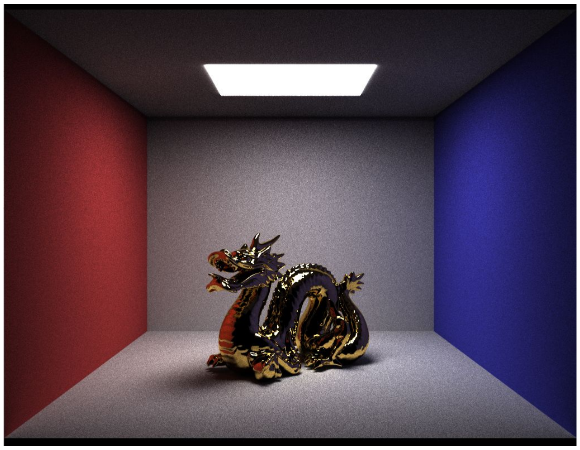
Gallery

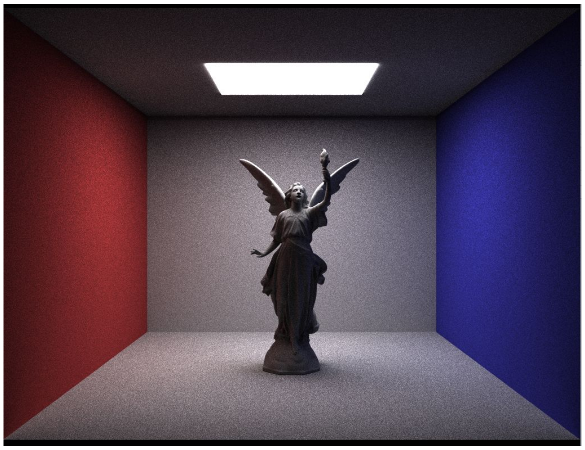
No path-tracer report is complete without a gallery of renderings created by the path tracer. Here they are:

Note that the radiance values differ from the original versions of the scene files, so the scene may not look the same as the originals. (They're darker than usual)









References

1. Getting rid of packets: Efficient SIMD single-ray traversal using multibranching BVHs., Wald et al., *Eurographics Symposium on Ray Tracing 2008*
2. Understanding the Efficiency of Ray Traversal on GPUs, Aila et al., *HPG 2009*
3. Dynamic Ray Scheduling to Improve Ray Coherence and Bandwidth Utilization, Navratil et al, *RT 2007*
4. Dynamic Ray Stream Traversal, Barringer et. al.

Division of Work

Equal work was performed by both project members.