Ray Tracing on GPU

Based on “Fast Ray Sorting and Breadth-First Packet Traversal for GPU Ray Tracing”
By Kirill Garanzha and Charles Loop

Rodolfo Schulz de Lima - nov/2010
Motivation

- GPU not designed for ray tracing
- Several limitations:
  - Threads must not diverge too much within a warp
  - Memory access should be coherent
- Solution: new pipeline that mitigate these issues
Basic Idea

- Take set of generated rays (also incoherent)
- Sort them into packets of coherent rays
Basic Idea

- Traverse these frustums through the BVH for scene and get a sorted list of intersected BVH-leaves per frustum
Basic Idea

- For each ray in a frustum perform intersection tests against all primitives contained in the leaves.
New Pipeline

- Primary Rays
  - Generate Rays
  - Sort Rays
  - Build Frustums
  - Frustums’ Traversal
  - Local Intersection Tests
  - Accumulate Shading

Secondary Rays
Ray Hashing

- Given a ray we compute a hash value quantizing the ray origin and direction and assuming a virtual grid with scene’s bounding box extents.
Ray Hashing

- All the rays with the same hash value assigned are coherent in the 3D-space
Ray Sorting

Ray IDs: 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19
Hash values: 0 3 7 9 14 17 3 4 2 5 3 3

Compress the data
Chunk Hash: 0 3 7 9 14 17
Chunk Base: 0 3 7 9 14 17
Chunk Size: 3 4 2 5 3 3

Apply sorting (radix, bitonic)
Chunk Hash: 3 9 7 17 0 14
Chunk Base: 3 9 7 17 0 14
Chunk Size: 4 5 2 3 3 3

Decompress sorted data
Reordered IDs: 3 4 5 6 9 10 11 12 13 7 8 17 18 19 0 1 2 14 15 16
Hash values: 0 3 7 9 14 17 3 4 2 5 3 3
Ray Sorting: Compression

Compressed data is created using Head Flags and Prefix sum for them as the offsets for new elements.
Ray Sorting: Decompression

Ray IDs:

Hash values:

Chunk Hash:

Chunk Base:

Chunk Size:

Scan(Chunk Size):

Sorted Chunks

Head Flags:

SegScan(Skeleton, H.Flags):

Reordered IDs = SegScan(Skeleton, Head Flags)
Frustum Creation

• Creation kernel: 1 warp per frustum

• Steps:
  - Define dominant axis
  - Calculate near and far plane positions
  - Calculate near and far rectangles extents
  - Calculate parameters for all planes

Intra-warp Brent-Kung scan algorithm is your friend here
Breadth-first Frustum Traversal

Init a queue with Frustum IDs vs. BVH Root node

Test Frustum intersection with Node’s children

Repeat for all BVH-levels, Swap(Qin, Qout)

Scatter Frustums and intersected children IDs

© Kirill Garanzha and Charles Loop
Ray-primitive Localized Intersections

- When traversal is done we decompose all active frustums into chunks of 32 rays
- These chunks are mapped to CUDA thread warps
- All rays within a warp share the same Frustum ID that determines computations and memory reads
- This eliminates branches within a CUDA warp of threads (only implicit masking remains in a SIMT)
- Store transform from world space triangle to normalized instead of vertex positions → faster ray/triangle intersection
while (ray warps are available) { // persistent threads
    RayWarp = fetch_next_warp();
    FrustumId = frustum_id(RayWarp);
    Ray = fetch_ray(RayWarpBase + threadIdx.x);
    for (all leaves (FrustumId))
        if (Ray intersects AABB(Leaf_i)) // mask rays
            for (all primitives (Leaf_i)) // coherent reads
                intersect Ray with a primitive_j;
}
Results – Ray Casting

OptiX: 124 FPS, 54.6 Mrays/s

RTGPU: 151 FPS, 64.4 Mrays/s

1.21x
Results – Ray Casting

OptiX: 99 FPS, 43.6 Mrays/s

RTGPU: 90 FPS, 39.6 Mrays/s
Results – Hard Shadows

OptiX: 91 FPS, 40.0 Mrays/s

0.47x

RTGPU: 43 FPS, 18.92 Mrays/s
Results – Hard Shadows

OptiX: 53.67 FPS, 23.6 Mrays/s

RTGPU: 25.5 FPS, 11.2 Mrays/s
Results – Soft Shadows 16x

OptiX: 3.5 FPS, 26.2 Mrays/s

RTGPU: 8.34 FPS, 62.4 Mrays/s

2.38x
Hard shadows... slower??

Back frustums with few rays in it, that's why.
Ray Packet Filling

- Soft Shadows - Sponza
- Hard Shadows - Sponza
Results - Overview

![Bar chart showing performance comparison between OptiX and RTGPU for different rendering types on different scenes.]

- Ray casting - teapot
- Ray casting - sponza
- Hard shadows - teapot
- Hard shadows - sponza
- Soft shadows - sponza

The chart shows the Mrays/s performance of OptiX and RTGPU for each category, with OptiX generally performing better than RTGPU.
Conclusion

- Promising framework for ray tracing on GPU
- No reflections/refractions documented
  - Might be slow due to high number of frustums with few rays in it
- Memory usage is high
  - Breadth-first traversal is the culprit
  - Delicate balance between frustum count and ray coherency inside frustum
- My implementation needs an urgent refactoring
- Implementation can made more efficient
  - Memory bandwidth optimization
  - Merging of some kernels

THANKS