Understanding the Efficiency of Ray Traversal on GPUs

Timo Aila       Samuli Laine
NVIDIA Research
Agenda

- What limits the performance of fastest traversal methods on GPUs?
  - Memory speed (bandwidth, latency)?
  - Computation?
  - Resource conflicts (serialization, scoreboard, …)?
  - Load balancing?
- How far from “theoretical optimum”?
  - How much performance on table?
- Solutions that help today
- Solutions that may help tomorrow
Terminology

- **Trace()**
  - Unpredictable sequence of *acceleration structure traversal* and *primitive intersection*

- **SIMT**
  - SIMD with execution divergence handling built into hardware
  - Computes everything on all lanes

- **Warp**
  - Group of threads that execute simultaneously in a SIMD/SIMT unit, 32 in NVIDIA hardware
“Theoretical optimum”

- Peak FLOPS as quoted by marketing?
  - Achievable only in special cases
  - Too conservative bound

- Instructions issued/sec for a *given kernel* (i.e. program), assuming:
  - Infinitely fast memory
  - Complete absence of resource conflicts
  - Tight upper bound when limited by computation
Simulator

- Custom simulator written for this project

**Inputs**

- Sequence of operations for each ray (traversal, intersection, enter leaf, mainloop)
- Native asm instructions for each operation

**Execution**

- Mimics GTX285 instruction issue [Lindholm et al. 2008]
- Assumes all instructions have zero latency
- Configurable SIMD width, default 32

**Outputs**

- Practical upper bound of performance
Test setup

- NVIDIA GeForce GTX285, CUDA 2.1
- BVH (64 bytes per 2 child nodes)
  - Always tested together, proceeds to closer
  - Greedy SAH construction, early triangle splits
  - Max 8 triangles per leaf
- Woop’s unit triangle test (48 bytes per tri)
- Nodes in 1D texture -- cached
- Triangles in global memory -- uncached
- Hierarchical optimizations not used in traversal
Test scenes

- **Conference**
  (282K tris, 164K nodes)

- **Fairy**
  (174K tris, 66K nodes)

- **Sibenik**
  (80K tris, 54K nodes)

- 1024x768, 32 secondary rays (Halton)
- Average of 5 viewpoints
- All timings include only `trace()`
Packet traversal

- Assign one ray to each thread
- Follows Günther et al. 2007
  - Slightly optimized for GT200
- All rays in a packet (i.e. warp) follow exactly the same path in the tree
  - Single traversal stack per warp, in shared mem
  - Rays visit redundant nodes
  - Coherent memory accesses

- Could expect measured performance close to simulated upper bound
Packet traversal

<table>
<thead>
<tr>
<th></th>
<th>Simulated Mrays/s</th>
<th>Measured Mrays/s</th>
<th>% of simulated</th>
</tr>
</thead>
<tbody>
<tr>
<td>Primary</td>
<td>149.2</td>
<td>63.6</td>
<td>43</td>
</tr>
<tr>
<td>AO</td>
<td>100.7</td>
<td>39.4</td>
<td>39</td>
</tr>
<tr>
<td>Diffuse</td>
<td>36.7</td>
<td>16.6</td>
<td>45</td>
</tr>
</tbody>
</table>

- Only 40%!? Similar in other scenes.
- Not limited by computation
- Memory speed even with coherent accesses?
- Simulator broken? Resource conflicts? Load balancing?
- 2.5X performance on table
Per-ray traversal

- Assign one ray to each thread
- Full traversal stack for each ray
  - In thread-local (external) mem [Zhou et al. 2008]
  - No extra computation on SIMT
- Rays visit exactly the nodes they intersect
  - Less coherent memory accesses
  - Stacks cause additional memory traffic
- If memory speed really is the culprit
  - Gap between measured and simulated should be larger than for packet traversal
### Per-ray traversal

<table>
<thead>
<tr>
<th>Type</th>
<th>Simulated Mrays/s</th>
<th>Measured Mrays/s</th>
<th>% of simulated</th>
</tr>
</thead>
<tbody>
<tr>
<td>Primary</td>
<td>166.7</td>
<td>88.0</td>
<td>53</td>
</tr>
<tr>
<td>AO</td>
<td>160.7</td>
<td>86.3</td>
<td>54</td>
</tr>
<tr>
<td>Diffuse</td>
<td>81.4</td>
<td>44.5</td>
<td>55</td>
</tr>
</tbody>
</table>

55% -- getting *closer* with all ray types

Memory not guilty after all?
while-while vs. if-if

**while-while trace()**
```
while ray not terminated
  while node does not contain primitives
    traverse to the next node
  while node contains untested primitives
    perform ray-node intersection
```

**if-if trace()**
```
while ray not terminated
  if node does not contain primitives
    traverse to the next node
  if node contains untested primitives
    perform ray-node intersection
```
Per-ray traversal (if-if)

- ~20% slower code gives same measured perf
  - Memory accesses are *less* coherent
  - Faster than while-while when leaf nodes smaller

- Neither memory communication nor computation should favor if-if
  - Results possible only when some cores idle?
Work distribution

- Histograms of warp execution times
  - Fewer extremely slow warps in if-if
  - Slowest warp 30% faster in if-if
  - Otherwise similar

- CUDA work distribution units
  - Optimized for homogeneous work items
    - Applies to all NVIDIA’s current cards
  - `Trace()` has wildly varying execution time
    - May cause starvation issues in work distribution
    - Need to bypass in order to quantify
Persistent threads

- Launch only enough threads to fill the machine once
  - Warps fetch work from global pool using atomic counter until the pool is empty
  - Bypasses hardware work distribution
  - Simple and generic solution

- Pseudocode in the paper
Persistent packet traversal

<table>
<thead>
<tr>
<th></th>
<th>Simulated Mrays/s</th>
<th>Measured Mrays/s</th>
<th>% of simulated</th>
</tr>
</thead>
<tbody>
<tr>
<td>Primary</td>
<td>149.2</td>
<td>122.1</td>
<td>82</td>
</tr>
<tr>
<td>AO</td>
<td>100.7</td>
<td>86.1</td>
<td>86</td>
</tr>
<tr>
<td>Diffuse</td>
<td>36.7</td>
<td>32.3</td>
<td>88</td>
</tr>
</tbody>
</table>

~2X performance from persistent threads

~85% of simulated, also in other scenes

Hard to get much closer

- Optimal dual issue, no resource conflicts, infinitely fast memory, 20K threads…
Persistent while-while

<table>
<thead>
<tr>
<th></th>
<th>Simulated Mrays/s</th>
<th>Measured Mrays/s</th>
<th>% of simulated</th>
</tr>
</thead>
<tbody>
<tr>
<td>Primary</td>
<td>166.7</td>
<td>135.6</td>
<td>81</td>
</tr>
<tr>
<td>AO</td>
<td>160.7</td>
<td>130.7</td>
<td>81</td>
</tr>
<tr>
<td>Diffuse</td>
<td>81.4</td>
<td>62.4</td>
<td>77</td>
</tr>
</tbody>
</table>

- ~1.5X performance from persistent threads
- ~80% of simulated, other scenes ~85%
- Always faster than packet traversal
  - 2X with incoherent rays
Speculative traversal

“If a warp is going to execute node traversal anyway, why not let all rays participate?”
- Alternative: be idle
- Can perform redundant node fetches
- Should help when not bound by memory speed

5-10% higher performance in primary and AO
No improvement in diffuse
- Disagrees with simulation (10-20% expected)
- First evidence of memory bandwidth issues?
  - Not latency, not computation, not load balancing…
Two further improvements

- Currently these are slow because crucial instructions missing
  - Simulator says 2 warp-wide instructions will help

- ENUM (prefix-sum)
  - Enumerates threads for which a condition is true
  - Returns indices \([0, M-1]\)

- POPC (population count)
  - Returns the number threads for which a condition is true, i.e. \(M\) above
1. Replacing terminated rays

- Threads with terminated rays are idle until warp terminates

- Replace terminated rays with new ones
  - Less coherent execution & memory accesses
  - Remember: per-ray kernels beat packets

- Currently helps in some cases, usually not
  - With ENUM & POPC, +20% possible in ambient occlusion and diffuse, simulator says
  - Iff not limited by memory speed
2. Local work queues

- Assign 64 rays to a 32-wide warp
  - Keep the other 32 rays in shared mem/registers
  - 32+ rays will always require either node traversal or primitive intersection
    - Almost perfect SIMD efficiency (% threads active)
  - Shuffling takes time
    - Too slow on GTX285

- With ENUM + POPC, in Fairy scene
  - Ambient occlusion +40%
  - Diffuse +80%
  - Iff not limited by memory speed
Conclusions (1/2)

- Primary bottleneck was load balancing
- Reasonably coherent rays not limited by memory bandwidth on GTX285
  - Even without cache hierarchy
  - Larger scenes should work fine
  - Only faster code and better trees can help
    - E.g. Stich et al. 2009
- ~40% performance on table for incoherent rays
  - Memory layout optimizations etc might help
  - Wide trees, maybe with enum & popc?
Conclusions (2/2)

- Encouraging performance
  - Especially for incoherent rays
  - *Randomly shuffled* diffuse rays 20-40M/sec
  - GPUs not so bad in ray tracing after all

- Persistent threads likely useful in other applications that have heterogeneous workloads
Acknowledgements

- David Luebke for comments on drafts, suggesting we include section 5
- David Tarjan, Jared Hoberock for additional implementations, tests
- Reviewers for clarity improvements
- Bartosz Fabianowski for helping to make CUDA kernels more understandable
- Marko Dabrovic for Sibenik
- University of Utah for Fairy Forest
CUDA kernels available

- http://www.tml.tkk.fi/~timo/
- NVIDIA Research website (soon)

- Download, test, improve

Thank you for listening!