Fancy Machines

- baetis.cs.utah.edu
- heptagenia.cs.utah.edu

- Quad-core Haswell Xeon @ 3.5GHz
  - 8 threads
Creative
Creative
Box Intersection

Box::intersect(HitRecord& hit, const Ray& ray) const{
    float tnear, t2;
    Vector inv = 1.f / ray.direction();
    Vector p1 = ( c1 - ray.origin() ) * inv;
    Vector p2 = ( c2 - ray.origin() ) * inv;
    Vector mins = p1.vecMin( p2 );
    Vector maxs = p1.vecMax( p2 );
    tnear = max( mins.x(), max( mins.y(), mins.z() ));
    t2 = min( maxs.x(), min( maxs.y(), maxs.z() ));

    if(tnear < t2)
        Hit
        Make sure to account for inside hits!
BVH layout

```c
int start_bvh = GetBVH();
```

Single BVH node
(8 words)

start_bvh

box corner
(3 floats)

box corner
(3 floats)

child ID

num children

(-1 indicates interior node)

```
c_min c_max 1 -1 c_min c_max 3 -1
```

start_bvh + 8

box corner
(3 floats)
BVH layout

- Sibling nodes are next to each other in memory
- Right child’s ID is always left_id + 1

node 2 (child is 13)  ...  node 13  node 14

start_bvh + (2 * 8)  start_bvh + (13 * 8)
BVH Nodes

As with all data held in global memory, recommended:

```cpp
BVHNode::BVHNode(int addr) {
    box.c1 = loadVectorFromMemory(addr + 0);
    box.c2 = loadVectorFromMemory(addr + 3);
    num_children = loadi(addr + 6);
    child = loadi(addr + 7);
}
```
Leaf Nodes

- Implied differently:
  - `num_children > 0`
  - `child_ID = address of node’s first triangle`
    - Not ID of first triangle!
    - Leaf node’s triangles are consecutive in memory
Leaf Nodes

- Box corner (3 floats)
- Box corner (3 floats)
- Child
- Num tris
- Remaining BVH nodes

- C_min
- C_max
- 682
- 2

- …

- Triangles

- …
- T1
- T2
- …

682 (address, not ID!)
inline void intersect(HitRecord& hit, const Ray& ray) const {
    int stack[32];
    int node_id = 0;
    int sp = 0;
    while(true){
        int node_addr = start_bvh + node_id * 8;
        BVHNode node(node_addr);
        HitRecord boxHit;
        node.box.intersect(boxHit, ray);
        if(boxHit.didHit())
            // and so on...
left_id = node.child;
if ( node.num_children < 0 ) // interior node node
{
    stack[ sp++ ] = left_id + 1;
    continue;
}
// leaf node
tri_addr = left_id;
for ( int i = 0; i < node.num_children; ++i)
{
    // intersect triangles
}
// ... finish outer loop, manage stack
My bvh class contains just a pointer to start_bvh

```
BoundingVolumeHierarchy(int _start_bvh)
{
    start_bvh = _start_bvh;
}
```

- Nodes are loaded one at a time as needed
- Don’t pre-load all the nodes!
  - Will not fit on each thread’s stack
inline void intersect(HitRecord& hit,
               const Ray& ray)

- Note that this hit record passed in is for the final hit triangle (or none if background)
- Don’t use the same one for testing against boxes!
for each pixel...
Ray ray;
camera.makeRay(ray, x, y);
HitRecord hit;
scene.bvh.intersect(hit, ray);
result = shade(...);
Updated Scene

- Scene class (or struct) should no longer hold typed pointers to hard-coded scene

```c
int start_materials
PointLight the_light // only one light now
BoundingVolumeHierarchy bvh;
```

- Make sure you pass the scene as reference to any shade functions
Performance

- Remember, there are some optimizations:
- Traverse down closer child first
- Don’t traverse subtree if closer triangle already found
- The pseudo-code I’ve shown doesn’t do this
- Can be tricky!
  - What if boxes overlap, and intersection is inside box?
Program 3
Caches

- Why?

- Most naïve option: transfer a single word from DRAM when needed
  - This is one model a programmer can assume
Access Patterns (RT example)

- If we load \texttt{c\_min.x}, what is the likelihood we will load \texttt{c\_min.y}?

\begin{tabular}{|c|c|c|}
\hline
\texttt{c\_min} & \texttt{c\_max} & 1 & -1 \\
\hline
\end{tabular}

- Spatial locality
  - Almost all data/workloads have this property
Temporal Locality

- In general:
  - If we needed some data at time $T$, we will likely need it at $T+\epsilon$
- If ray 1 takes a certain path through BVH, ray 2 will likely take a similar path
- Becomes even more important with shared caches
Temporal Locality

ray1  ray2
Amortization

- Temporal and spatial locality are just the assumptions that allow us to amortize DRAM access

- Activating DRAM for a read has huge overhead
  - The read itself is somewhat insensitive to the amount of data

- “DRAM exists to refill cache lines” - Erik
  - Or: Cache lines exist hold DRAM bursts
Slightly More to it

- DRAM is extremely slow

- Caches are extremely fast
  - But they have to be small

- Ideal:
  - Hold a piece of data in cache for as long as it is possibly needed
In TRaX

- 65nm process
  - 1GHz
    - ~20 - 200 cycles
      (depends on pressure, access patterns)
    - ~20 – 70nJ / read
    - ~ 4GB
    - 3 cycles
      - ~1.2nJ / read
      - ~ 64KB – 4MB
    - 1 cycle
      - ~.13nJ / read
      - ~ 4KB – 32KB
Life of a Read

- `LOAD r2  r0, ...`

Diagram:
- RF
  - r0
  - r1
  - r2
  - r3
- L1
  - Check tag: Hit?
  - Map address to line number
  - yes
  - no
  - evict old line
- L2
  - Check tag: Hit?
  - yes
  - no
- DRAM
  - Map address to channel, then wait...
  - no
**Address ➔ Line**

- **Assuming 64B lines**
- **Address space >> cache size**
  - Physical cache line holds many different address ranges

<table>
<thead>
<tr>
<th>Line 0</th>
<th>Addresses 0 – 63 (tag = 0)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Addresses 256 – 319 (tag = 1)</td>
</tr>
<tr>
<td></td>
<td>...</td>
</tr>
<tr>
<td>Line 1</td>
<td>Addresses 64 – 127 (tag = 0)</td>
</tr>
<tr>
<td></td>
<td>...</td>
</tr>
<tr>
<td>Line 2</td>
<td>Addresses 128 – 191 (tag = 0)</td>
</tr>
<tr>
<td></td>
<td>...</td>
</tr>
<tr>
<td>Line 3</td>
<td>Addresses 192 – 255 (tag = 0)</td>
</tr>
<tr>
<td></td>
<td>...</td>
</tr>
</tbody>
</table>
Associativity

- A 2-way set-associative cache checks 2 possible lines for a given address
  - Why?
Associativity

- A 2-way set-associative cache checks 2 possible lines for a given address
  - Why?

- When evicting, we can now make an intelligent choice
  - Evict oldest line, LRU, etc…
Associativity

- TRaX cache model is “direct-mapped”
  - Only one address → line mapping

- Direct-mapped are smaller, cheaper, low-power
  - For RT specifically, seems to work well (91-94% hitrate)
Parallel Accesses (shared cache)

Thead A
read: 193 (miss)

Thead B

L1
Miss!
Incoming [A]

L2
Hit

3 cycles

Cycle
0
1
2
3
Parallel Accesses

0

1

2

3

Thead A
read: 193
(miss)

Thead B
read: 197
(miss)

L1
Miss!
Incoming
[A, B]

L2
Hit

3 cycles
Parallel Accesses

<table>
<thead>
<tr>
<th>Cycle</th>
<th>Thread A</th>
<th>Thread B</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>read: 193 (miss)</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td>read: 197 (miss)</td>
</tr>
<tr>
<td>2</td>
<td>complete</td>
<td>complete</td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

L1: cached

L2:
**MSHR**

- "Miss Status Handling Register" (one per line)
  - Tracks status/recipients for incoming line
- Thread B incurs "hit under miss"
  - Difference?

<table>
<thead>
<tr>
<th>Cycle</th>
<th>Thread A</th>
<th>Thread B</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>read: 193 (miss)</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td>read: 197 (miss)</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>complete</td>
</tr>
<tr>
<td>3</td>
<td>complete</td>
<td>complete</td>
</tr>
</tbody>
</table>
**Hit Under Miss**

- **Thread B incurs “hit under miss”**
- **Difference?**

**A:** one L1 access, one L2 access

**B:** one L1 access

<table>
<thead>
<tr>
<th>Cycle</th>
<th>Thread A</th>
<th>Thread B</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>read: 193 (miss)</td>
<td>read: 197 (miss)</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>complete</td>
<td>complete</td>
</tr>
</tbody>
</table>
Single Thread HUM

```
LOAD    r7,      r3,     0
LOAD    r9,      r3,     1
LOAD    r11,     r3,     2
LOAD    r6,      r3,     3
LOAD    r13,     r3,     4
LOAD    r8,      r3,     5
```

- Assume relevant lines are initially uncached
- Generates:
  - 6 L1 accesses
  - 1 L2 access
  - 1 DRAM access
Many-Core

All processed simultaneously

Suppose each of these nodes map to the same cache line (but different tag)
Ray Coherence

- Processing coherent rays simultaneously results in data locality
  - Lots of research involving collecting coherent rays
  - More on this later