Lecture 8:
High-Performance Ray Tracing

Image Synthesis
Stanford CS348b, Spring 2014
Topics

- Motivation recap
  4 cores x 8-wide SIMD: 32x perf. difference available if parallelism is used effectively

- Topics:
  - Multi-core parallelism for ray tracing
  - Applications of SIMD to ray tracing
    - Hierarchy traversal and ray/object intersection
    - Coherent and incoherent rays
  - Real-time ray tracing?
Assumptions for Today

- Prebuilt acceleration structures
  - However: high-performance construction is an important topic, especially for real-time ray tracing
  - And remember Amdahl’s law...

- Read-only scene database
  - No dynamic tessellation, texture caching, ...

These are both unreasonable assumptions for production rendering, but will keep things simpler today
Multi-Core Parallelism For Ray Tracing
Identifying Work To Perform in Parallel

- This is generally straightforward with ray tracing
  - Computation for each eye ray is completely independent from computation for every other eye ray
    - (Given our initial working assumptions)
- e.g. 2M pixels x 64 samples per pixel gives 128M independent pieces of work
  - In practice, better to consider collections of eye rays
- This abundance of independent work is why ray tracing has been called “embarrassingly parallel”
Static Screen-Space Decomposition

- If N processing cores, each one gets 1/N of the image
  - +s: straightforward implementation
  - -s: ?

Static screen-space decomposition for 8 cores
The Problem With Static Work Assignment

Core

Time (seconds)

Poor load balancing, idle processors
Dynamic Work Assignment

- Observation: if we have more screen tiles than processing cores, we can hand out more work to cores as they finish previous tiles.
- Leads to much better load-balancing, at a cost of communication and coordination to hand out work.
Useful Abstraction: Task Systems

- Programmer provides many independent units of work ("tasks")
- Task system abstracts processing cores, scheduling, and synchronization among the cores
- Task system schedules work across available cores
  - Independence of tasks allows arbitrary scheduling of concurrent execution
  - Having \#tasks >> \#cores helps with load balancing
- Examples: Apple Grand Central Dispatch, Microsoft Concurrency Runtime, Intel TBB
- pbrt uses a task system as the basis for parallelization
Task System Implementation

- One hardware thread is launched for each processing core
  - Expectation is that task system manages all computation
  - Avoid context switches from having \( \# \) threads \( \geq \) \( \# \) cores

- Localizes low-level issues like OS and HW interaction
  - Programmer can focus on decomposing work into tasks

- Each thread pulls work from a queue, runs it, until all tasks have finished
Task System Implementation

- Single shared work deque:

| task0 | task1 | task2 | task3 | task4 | task5 | task6 | task7 | task8 | task9 | ... |

Worker threads pull tasks from head of deque

New tasks are added to tail of deque

- +s: ~straightforward implementation
- -s: doesn’t scale well with:
  - Large numbers of processing cores
  - Relatively short tasks
## Task System Implementation

- **Distributed work deques:**

<table>
<thead>
<tr>
<th>core 0</th>
<th>task0</th>
<th>task4</th>
<th>task8</th>
<th>task12</th>
<th>task16</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>core 1</td>
<td>task1</td>
<td>task5</td>
<td>task9</td>
<td>task13</td>
<td>task17</td>
<td>...</td>
</tr>
<tr>
<td>core 2</td>
<td>task2</td>
<td>task6</td>
<td>task10</td>
<td>task14</td>
<td>task18</td>
<td>...</td>
</tr>
<tr>
<td>core 3</td>
<td>task3</td>
<td>task7</td>
<td>task11</td>
<td>task15</td>
<td>task19</td>
<td>...</td>
</tr>
</tbody>
</table>

- Task threads initially pull work from their local deques
  - Efficient: doesn’t require coordination across all threads
- Once local work is exhausted, work stealing to grab tasks from other cores deques
Task API In pbrt

class Task {
public:
    virtual void Run() = 0;
};
void EnqueueTasks(const vector<Task *> &tasks);
void WaitForAllTasks();

class SqrtTask : public Task {
public:
    SqrtTask(float *p) : ptr(p) { }
    void Run() { *ptr = sqrtf(*ptr); }
private:
    float *ptr;
};
...

float array[1024];
// init array...
std::vector<Task *> tasks;
for (int i = 0; i < 1024; ++i)
    tasks.push_back(new SqrtTask(&array[i]));
EnqueueTasks(tasks);
WaitForAllTasks();
// delete Task *s...
Shortcomings of The Task API In pbrt

class Task {
public:
    virtual void Run() = 0;
};
void EnqueueTasks(const vector<Task *> &tasks);
void WaitForAllTasks();

class SqrtTask : public Task {
public:
    SqrtTask(float *p) : ptr(p) { }
    void Run() { *ptr = sqrtf(*ptr); }
private:
    float *ptr;
};
...
float array[1024];
// init array...
std::vector<Task *> tasks;
for (int i = 0; i < 1024; ++i)
    tasks.push_back(new SqrtTask(&array[i]));
EnqueueTasks(tasks);
WaitForAllTasks();
// delete Task *s...
void ParallelFor(const std::function<void(int)> &func, int count);

float array[1024];
// init array...
ParallelFor([array](int index) {
    array[index] = sqrtf(array[index]);
}, 1024);
Effect of Task Granularity

- Time (s) vs. Log2 (# Tasks)
- # tasks = # cores
- Best perf. w/ 4096 tasks (~300 pixels)

Independent task per pixel:
Per-task overhead > better load balancing
What Requires Coordination in a Ray Tracer?

- Updating the output image
- Updating the scene representation
  - On demand tessellation, BVH construction, ...
  - Caches: texture and geometry
  - Dynamic caching of illumination information, ...
- Administrivia
  - Statistics, counters, ...
Output Image Updates Need Coordination

- If we’re using a pixel reconstruction filter wider than a pixel’s extent, samples from neighboring pixel areas contribute.
  - Some of these will be computed by other tasks.

![Diagram showing output image updates and coordination between tasks A and B.](image-url)
Coordinating Updates of Shared Data

- If multiple threads concurrently update the same memory location without coordination, results may be undefined/incorrect.
- Consider two threads executing this code:

```c
float img[FB_SIZE];
// ...
float v = ...;
img[offset] = img[offset] + v;
```

With this execution schedule, thread 2’s update is lost completely.
Coordinating Updates of Shared Data

- **Atomic update instructions**
  - HW instructions that ensure consistent results with a number of associative ops
  - Not available for float (why?)

- **Locks to ensure mutual exclusion**
  - Only one thread at a time is allowed to execute code protected by a given mutex

```cpp
// C++11
std::atomic<int> img[FB_SIZE];
// ...
int v = ...;
img[offset] += v;
```

```cpp
// C++11
std::mutex mutex;
float img[FB_SIZE];
// ...
float v = ...;
mutex.lock();
img[offset] += v;
mutex.unlock();
```
Coordinating Updates of Shared Data

- **Transactional memory**
  - A set of memory updates is marked as a transaction
  - If there is no conflict with other concurrently-executing threads, the transaction is allowed to commit
  - If conflict, discard all memory writes within the transaction, then repeat, or use a mutex

- **HW support for transactional memory starting in 2013 CPUs**
Coordinating Image Updates

- **Single mutex**
  - May be a source of contention
  - Lock elision on modern CPUs helps, though...

- **Multiple mutexes, covering different subsets of image**
  - More work to acquire mutexes, reduces contention

- **Atomic operations**

- **Transactional memory**
  - This should be a great fit; common case is uncontended
SIMD And Ray Tracing
Context for SIMD

- Assume that multi-core parallelism has been applied

- Now, consider a single thread/task:
  - How can we apply SIMD within that task?

- Focus today will be SIMD for ray/object intersection acceleration
SIMD Ray-Triangle Intersection

- **SIMD-ization of scalar code** (right) is fairly straightforward
  - Can only take early outs when true for all lanes

- **Options:**
  - One ray, N triangles
  - N rays, 1 triangle
  - N rays, N triangles

```c
float TriangleIntersect(const Point p[3],
const Ray &ray) {
    Vector e1 = p[1] - p[0], e2 = p[2] - p[0];
    Vector s1 = Cross(ray.d, e2);
    float divisor = Dot(s1, e1);
    if (divisor == 0.)
        return false;
    float invDivisor = 1.f / divisor;
    // Compute first barycentric coordinate
    Vector d = ray.o - p[1];
    float b1 = Dot(d, s1) * invDivisor;
    if (b1 < 0. || b1 > 1.)
        return false;
    // Compute second barycentric coordinate
    Vector s2 = Cross(d, e1);
    float b2 = Dot(ray.d, s2) * invDivisor;
    if (b2 < 0. || b1 + b2 > 1.)
        return false;
    // Compute _t_ to intersection point
    float t = Dot(e2, s2) * invDivisor;
    return t;
}
```
SIMD Ray-Triangle Intersection

```cpp
struct SOAPoint {
    vec<float, 8> x, y, z;
    // operator overloads, etc...
};

struct SOAVector {
    vec<float, 8> x, y, z;
    // operator overloads, etc...
};

struct SOARay {
    SOAPoint o;
    SOAVector d;
};

vec<float, 8> Dot(const SOAVector &a, const SOAVector &b) {
    return a.x * b.x + a.y * b.y + a.z * b.z;
}
// ...

vec<float, 8>
TriangleIntersect(const SOAPoint p[3], const SOARay &ray) {
    SOAVector e1 = p[1] - p[0], e2 = p[2] - p[0];
    SOAVector s1 = Cross(ray.d, e2);
    vec<float, 8> divisor = Dot(s1, e1);
    vec<bool, 8> no_hit = (divisor == 0.);
    vec<float, 8> invDivisor = 1.f / divisor;

    SOAVector d = ray.o - p[1];
    vec<float, 8> b1 = Dot(d, s1) * invDivisor;
    no_hit |= (b1 < 0. || b1 > 1.);

    SOAVector s2 = Cross(d, e1);
    vec<float, 8> b2 = Dot(ray.d, s2) * invDivisor;
    no_hit |= (b2 < 0. || b1 + b2 > 1.);

    float t = Dot(e2, s2) * invDivisor;
    return select(no_hit, -INFINITY, t);
}
```
Application of SIMD to Ray Intersection

- SIMD ray/bounding box tests are similarly straightforward
- The main challenge:
  - Structuring the system to provide memory-coherent data to process in SIMD...
N Rays, 1 Triangle / BVH node: packet tracing

Program explicitly intersects a collection of rays against BVH at once

[Wald et al. 2001]

RayPacket
{}
  Ray rays[PACKET_SIZE];
  bool active[PACKET_SIZE];
};

trace(RayPacket rays, BVHNode node, ClosestHitInfo packetHitInfo)
{}
if (!ANY_ACTIVE_intersect(rays, node.bbox) ||
  (closest point on box (for all active rays) is farther than hitInfo.distance))
  return;

update packet active mask

if (node.leaf) {
  for (each primitive in node) {
    for (each ACTIVE ray r in packet) {
      (hit, distance) = intersect(ray, primitive);
      if (hit && distance < hitInfo.distance) {
        hitInfo[r].primitive = primitive;
        hitInfo[r].distance = distance;
      }
    }
  }
} else {
  trace(rays, node.leftChild, hitInfo);
  trace(rays, node.rightChild, hitInfo);
}
Packet Tracing

Blue = active ray after node box test

r6 does not pass node F box test due to closest-so-far check
Advantages of Packets

- **SIMD execution**
  - One vector lane per ray

- **Amortize fetch: all rays in packet visit node at same time**
  - Load BVH node once for all rays in packet
  - Note: value to making packets much bigger than SIMD width!

- **Amortize work (packets are hierarchies over rays)**
  - Use interval arithmetic to conservatively test entire set of rays against node bbox (e.g., think of a packet as a beam)
  - Further optimizations possible when all rays share origin
  - Note: value to making packets much bigger than SIMD width!
Disadvantages of Packets

- If any ray must visit a node, it drags all rays in the packet along with it.

- Loss of efficiency: node traversal, intersection, etc., amortized over less than a packet's worth of rays.

- Not all SIMD lanes doing useful work.

Blue = active ray after node box test
When rays are incoherent, benefit of packets can decrease significantly. This example: packet visits all tree nodes. (All rays visit all tree nodes)
Incoherence is a property of both the rays and the scene.

Random rays are “coherent” with respect to the BVH if the scene is one big triangle!
Incoherence is a property of both the rays and the scene.

Camera rays become “incoherent” with respect to lower nodes in the BVH if a scene is overly detailed.

(note importance of geometric level of detail)
Improving packet tracing with ray reordering

Idea: when packet utilization drops below threshold, resort rays and continue with smaller packet

- Increases SIMD utilization
- Still loses amortization benefits of large packets

Example: 8-wide SIMD processor, 16-ray packets
(2 SIMD instructions required to perform operation on all rays in packet)

16-ray packet: 7 of 16 rays active

Reorder rays
Recompute intervals/bounds for active rays

Continue tracing with 8-ray packet:
7 of 8 rays active
Improving packet tracing with ray reordering

Idea: when packet utilization drops below threshold, resort rays and continue with smaller packet
  - Increases SIMD utilization
  - Still loses amortization benefits of large packets

Benefit of higher utilization/tighter packet bounds must overcome overhead of reordering operation

10-18% speedup over standard packet tracing for glossy reflection rays
25-50% speedup for 2-bounce diffuse interreflection rays
(4-wide SSE implementation)
Giving up on packets

- Even with reordering, ray coherence during BVH traversal will diminish
  - Little benefit to packets (can decrease performance compared to single ray code)

- Idea: exploit SIMD execution within single ray-BVH intersection query
  - Interior: use wider-branching BVH
    (test single ray against multiple node bboxes in parallel)
    - Branching factor 4 has similar efficiency to branching factor 2
    - Branching factor 16 exhibits significant reduction in efficiency
  - Leaf: test ray against multiple triangles in parallel

[Wald et al. 2008]
Giving up on packets

- Even with reordering, ray coherence during BVH traversal will diminish
  - Little benefit to packets (can decrease performance compared to single ray code)

- Idea: exploit SIMD execution within single ray-BVH intersection query
  - Interior: use wider-branching BVH
  - Leaf: test ray against multiple triangles in parallel

- SIMD efficiency independent of ray coherence

- But no work/bandwidth reduction due to amortization across rays
  - Weren’t getting much benefit from packets of incoherent rays anyway
Packet tracing best practices

- Use large packets for higher levels of BVH
  - Ray coherence always high at the top of the tree

- Switch to single ray (intra-ray SIMD) when packet utilization drops below threshold
  - For wide SIMD machine, a single branching-factor 4 BVH works well for both packet and single ray traversal

- Can use packet reordering to postpone time of switch
  - Reordering allows packets to provide benefit deeper into tree
Global ray reordering

Idea: batch up rays in the same part of the scene. Process these rays together to increase locality

Partition BVH into treelets (treelets sized for L1 or L2 cache)

1. When ray (or packet) enters treelet, add rays to treelet queue

2. When treelet queue is sufficiently deep, intersect enqueued rays with treelet

[Pharr 1997, Navratil 07, Alia 10]
Lots of academic work + some industry attempts
Still not common in major ray tracing implementations
Summary

- Visibility: determine which scene geometry contributes to the appearance of which screen pixels
  - “Basic” rasterization: given polygon, find samples(s) it overlaps
  - “Basic” ray tracing: given ray, find triangle(s) that it intersects

- In practice, not as different as you might think

- Just different ways to solve the problem of finding interacting pairs between two hierarchies **
  - Hierarchy over point samples
  - Hierarchy over geometry

** A great analogy is collision detection (credit Tim Foley)
Consider performant, modern solutions for primary-ray visibility

- "Rasterizer"
  - Hierarchical rasterization (uniform grid over samples)
  - Hierarchical depth culling (quad-tree over samples)
  - Application scene graph, hierarchy over geometry
    - Modern games perform conservative coarse culling, only submit potentially visible geometry to the rendering pipeline
      (in practice, rasterization not linear in amount of geometry in scene)

- "Ray tracer"
  - BVH: hierarchy over geometry
  - Packets form hierarchy over samples (akin to frame buffer tiles). Breaking packets into small packets during traversal adds complexity to the hierarchy
  - Wide packet traversal, high-branching BVH: decrease work efficiency for better machine utilization
    (in practice, significant constants in front of that \( \lg(N) \))
Acknowledgement

- Thanks to Kayvon Fatahalian for the content in slides 28-42.