Embree Ray Tracing Kernels for the Intel® Xeon® and Intel® Xeon Phi™ Architectures

Sven Woop, Carsten Benthin, Ingo Wald
Intel
Legal Disclaimer and Optimization Notice

INFORMATION IN THIS DOCUMENT IS PROVIDED “AS IS”. NO LICENSE, EXPRESS OR IMPLIED, BY ESTOPPEL OR OTHERWISE, TO ANY INTELLECTUAL PROPERTY RIGHTS IS GRANTED BY THIS DOCUMENT. INTEL ASSUMES NO LIABILITY WHATSOEVER AND INTEL DISCLAIMS ANY EXPRESS OR IMPLIED WARRANTY, RELATING TO THIS INFORMATION INCLUDING LIABILITY OR WARRANTIES RELATING TO FITNESS FOR A PARTICULAR PURPOSE, MERCHANTABILITY, OR INFRINGEMENT OF ANY PATENT, COPYRIGHT OR OTHER INTELLECTUAL PROPERTY RIGHT.

Software and workloads used in performance tests may have been optimized for performance only on Intel microprocessors. Performance tests, such as SYSmark and MobileMark, are measured using specific computer systems, components, software, operations and functions. Any change to any of those factors may cause the results to vary. You should consult other information and performance tests to assist you in fully evaluating your contemplated purchases, including the performance of that product when combined with other products.

Optimization Notice

Intel’s compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804
Outline

• Embree Overview
• ISPC Overview
• Embree ISPC API
• Implementation Details
• Embree Performance
Embree Overview
Embree Ray Tracing Kernels

Motivation

• Ray tracing is used heavily for professional graphics applications (Movie Industry, Visualization, Digital Content Creation)
• Implementing a fast ray tracer is very difficult ➔ Implementations often don’t perform optimal on Intel Architecture

Goal

• Provide the fastest ray tracing kernels to application developers

Embree is an open source high fidelity visualization toolkit for application developers who want to create compelling visual applications to deliver an outstanding user experience on current and future computing architectures. Easy to integrate, Embree provides a blueprint for scalable and efficient Ray Tracing capabilities that are demanded by media and entertainment, product design, energy or scientific visualization applications.
Why Ray Tracing?

Physically based:
• Imagery can be trusted
• Can generate photorealistic images

Conceptually simple:
• Easy to build renderers with
• Effects combine naturally

Embarrassingly parallel:
• Scales to arbitrary number of compute units
• But also very compute hungry !!!!!
Problem: Writing a Fast Ray Tracer is Hard!

• **Need deep domain knowledge:** many different data structures (kd-trees, octrees, grids, BVH2, BVH4, ..., hybrid structures) and algorithms (single rays, packets, large packets, stream tracing, ...) to choose

• **Need low-level expert knowledge of hardware:** scalability to many cores/CPUs, efficient use of SIMD units, different ISAs (SSE, AVX, Xeon Phi™)

• **Need for multiple implementations:** Different ISAs/CPU types favor different data structures, data layouts, algorithms, and implementations

→ Embree is a “middleware” solution!
What is Embree?

- High fidelity visualization toolkit for application developers
- Easy to integrate, rapid prototyping
- Highly efficient and scalable Ray Tracing features and capabilities
- Compatible with present and future compute platforms
- Available as Open Source on http://embree.github.com (Apache 2.0 license)
Embree 1.x Overview

Support for Intel® Xeon® CPUs:
- Optimized for SSE 3-4.1 (and AVX 1)
- Most professional renders use this platform

“Single ray” Interface to Ray Tracing Kernels:
- Application developers can use scalar C++ code
- Easy to integrate into existing applications

High Performance:
- 1.5x – 6x rendering speedup achievable
- 15M triangles/s high quality spatial index build
Embree 2.0 New Features

• Support for latest Intel® Xeon® Processor family and Intel® Xeon Phi™ coprocessor products
• Support for “Ray Packets” (4, 8, or 16 rays per packet)
• Integration with Intel® SPMD program compiler (ISPC, http://ispc.github.com )
• Two-level Hierarchies and fast BVH builders

Embree 2.0 Released Today!

→ http://embree.github.com
How to use Embree?

- As a benchmark to identify performance issues in your own code
- As a library through the Embree API
- As example code by copying code
Single Ray Tracing not optimal for wide SIMD

Rendering Application
- Shaders leverage 4-wide SSE naturally, e.g. \((x,y,z,\_\,)\) or \((r,g,b,a)\)
- Using wider SIMD units is less efficient and code less readable

Ray Tracing Kernels
- Single ray kernels work well for 4-wide SSE
- Strongly diminishing return for vector widths wider than 4
How to support Intel® Xeon Phi™ Coprocessor?

Example: Intel® Xeon Phi™ 7120X (→http://ark.intel.com)

• Highly parallel architecture (61 cores, 4 threads per core, 1.238GHz)
• Peak floating point performance >2TFlops SP
• 16 GB of high memory bandwidth (320GB/s)
• 30 MB of L2 cache
• 16-wide SIMD ISA, 32 SIMD registers
• SIMD + scalar instr can co-issue

→ Great architecture for ray tracing!
Ray Packets and SPMD Programming Model is a Solution

**Single Program Multiple Data (SPMD) programming model**
- One „program“ per SIMD lane (e.g. one pixel per SIMD lane)
- Masking and sequentialization for diverging control flow
- Code „looks“ like scalar code (e.g. OpenCL)
- Automatic, efficient, and guaranteed vectorization

**Embree 2.0 supports SPMD programming model**
- Support for “Ray Packets” (4, 8, or 16 rays per packet)
- Integration with Intel® SPMD Program Compiler (ISPC, [http://ispc.github.com](http://ispc.github.com))
ISPC Overview
Intel® SPMD Program Compiler (ISPC)

• Support for scalar and vectorized control flow and data flow
• Masking done automatically
• Compilation to different vector ISAs (SSE, AVX, Xeon Phi™)
• Allows close coupling of C/C++ and ISPC code (Data structures shared with C/C++ code)
• Available as Open Source from http://ispc.github.com
ISPC Language

• **C-based syntax** (for, while, if, then else, int, float, ...) plus extensions

• Pointers, structs, new, delete, recursion, function pointers, ...

• **uniform** and **varying** type qualifiers to express scalars and vectors

• Rich standard library: vectorized transcendental functions, atomics, ...
Embree ISPC API
Embree ISPC API

• Simple low level Ray Tracing API (build, trace)
• Triangle Meshes and Two Level Scene support
• Support for different acceleration structures
• Support for different traversal algorithms
Embree ISPC Example: Mesh Creation

/* create triangle mesh */
uniform RTCGeometry* uniform mesh = rtcNewTriangleMesh (12, 8);

/* fill vertex buffer */
uniform RTCVertex* uniform vertices = rtcMapPositionBuffer(mesh);
vertices[0].x = -1; vertices[0].y = -1; vertices[0].y = -1;
...  
rtcUnmapPositionBuffer(mesh);

/* fill triangle index buffer */
uniform RTCTriangle* uniform triangles = rtcMapTriangleBuffer(mesh);
triangles[0].v0  = 0; triangles[0].v1  = 1; triangles[0].v2 = 2;
triangles[0].id0 = 0; triangles[0].id1 = 0;
...  
rtcUnmapTriangleBuffer(mesh);

/* launch and wait for build task */
launch rtcBuildAccel (mesh);
sync;

/* query default intersector */
uniform RTCIntersector* uniform intersector = rtcQueryIntersector(mesh);
/* loop over all screen pixels */
foreach (y=0 ... screenHeight-1, x=0 ... screenWidth-1)
{
    /* create primary ray */
    varying Ray ray;
    ray.org = p;
    ray.dir = normalize(add(mul(x,vx,mul(y,vy),vz));
    ray.tnear = 0;
    ray.tfar = inf;
    ray.id0 = ray.id1 = -1;

    /* trace ray */
    intersector->intersect(intersector,ray);

    /* uv-shading and framebuffer write */
    if (ray.id0 != -1)
        pixels[y*width+x] = make_vec3f(ray.u,ray.v,1.0-ray.u-ray.v);
    else
        pixels[y*width+x] = 0;
}
Implementation Details
Algorithms for Intel® Xeon® CPUs

Spatial index structures:
- BVH2, BVH4 (recommended), BVH8, BVH4MB (motion blur)

Triangle representations:
- triangle4 (recommended), triangle8, triangle4i (less memory), ...

Traversal algorithms:
- Single ray (recommended for incoherent workloads)
- SSE packet, AVX packet
- SSE hybrid, AVX hybrid (recommended for coherent workloads)

Ray/triangle interectors:
- Möller-Trumbore (recommended for performance)
- Plücker variant (recommended for accuracy)
BVH4 Spatial Index Structure

struct Node4 {
    ssef minx, miny, minz;
    ssef maxx, maxy, maxz;
    Node4* child[4];
}

struct Triangle4 {
    ssef v0x, v0y, v0z;
    ssef e1x, e1y, e1z;
    ssef e2x, e2y, e2z;
    ssef Nx, Ny, Nz;
    ssei id0, id1;
}
Optimizing BVH4 Traversal for CPUs

- Reduce number of executed instructions
- Reduce data dependencies of critical paths
- Take advantage of special instructions (e.g. SSE, bitscan, etc.)
- Optimize most frequently executed code paths
Optimizing BVH4 Traversal for CPUs (SSE)

• Load front/back plane based on direction sign of the ray.
• Balanced min/max trees
• Bitscans to iterate through hit children
  • Early exit for 0 children hit (20%)
  • 1 child hit (50%): keep next node in register (instead of push/pop sequence)
  • 2 children hit (20%): keep next node in register, sort using a branch

while (true) {
  if (isLeaf(node)) goto leaf;
  ssef nearX = (norg.x + node[nearX]) * rdir.x;
  ssef nearY = (norg.y + node[nearY]) * rdir.y;
  ssef nearZ = (norg.z + node[nearZ]) * rdir.z;
  ssef farX  = (norg.x + node[farX]) * rdir.x;
  ssef farY  = (norg.y + node[farY]) * rdir.y;
  ssef farZ  = (norg.z + node[farZ]) * rdir.z;
  ssef near  = max(max(nearX,nearY),
                   max(nearZ,ray.near));
  ssef far   = min(min(farX,farY),
                   min(farZ,ray.far));
  int hitmask = movemask(near <= far);
  if (hitmask == 0) goto pop;
  int c = bitscan(hitmask);
  hitmask = clearbit(hitmask,c);
  if (hitmask == 0) {
    node = node.child[c]; continue;
  } ...
}
Algorithms for Intel® Xeon Phi™ Coprocessor

Spatial index structure:
- BVH4AOS

Traversal algorithms:
- Single ray traversal
- Packet traversal
- Hybrid packet/single ray traversal

Triangle intersector:
- Möller-Trumbore
Single Ray Traversal for Intel® Xeon Phi™ Coprocessor

- 4-wide BVH in AoS layout, 2 x 64bit cachelines (4 x box min, 4 x box max)
- Use 16-wide SIMD as 4 x 4-wide ‘lanes’, ops using AoS layout: 4 x “xyzw”
- Horizontal reductions to determine hit node with shortest distance
- Critical path optimization for 0, 1, and 2 hit nodes
- Triangle intersection: 1 ray vs. 4 triangles (AoS layout)
- Single ray traversal best for incoherent rays
Packet Traversal for Intel® Xeon Phi™ Coprocessor

• 16 rays per packet (fits ISA width)

• 4-wide BVH in AoS layout (same layout as for single ray traversal)

• BVH node test: 16 rays vs. 1 box

• Triangle intersection: 16 rays vs. 1 triangle

• Packet traversal best for coherent rays
Hybrid Traversal for Intel® Xeon Phi™ Coprocessor

- Real world ray distributions neither fully incoherent nor fully coherent
- Combine packet and single ray traversal into single hybrid traversal

Hybrid traversal
- Switch between single and packet traversal based on ray coherence
- Use SIMD utilization as measure for ray coherence (bitcount of mask bits)
- Low #active rays per packet (incoherent rays) → single ray traversal
- High #active rays per packet (coherent rays) → packet traversal
- Switch multiple times per packet (need low switch overhead)
Hybrid Traversal for Intel® Xeon Phi™ Coprocessor

• SIMD util < 2 ➞ single ray traversal
Embree Performance
Embree Example Path Tracer

• Flexible modular system design
• Virtual interface to cameras, lights, materials, brdfs, etc.
• Materials build from multiple BRDF components
• Support for HDR environment lighting
• C++ and ISPC implementation of renderer

Entire „Embree 1.x path tracer“ runs on Xeon® and Xeon Phi™
Benchmark Settings

- Intel® Xeon® E5-2690 (8 cores @ 2.9 GHz)
- Intel® Xeon Phi™ 7120 (61 cores @ 1.238 Ghz)
- 1024x1024 image resolution, shading takes 25-40% of total rendering time

<table>
<thead>
<tr>
<th>Scene</th>
<th>#triangles</th>
</tr>
</thead>
<tbody>
<tr>
<td>Imperial Crown of Austria</td>
<td>4.3 M</td>
</tr>
<tr>
<td>Martin Lubich, <a href="http://www.loramel.net">www.loramel.net</a></td>
<td></td>
</tr>
<tr>
<td>Bentley 4.5l Blower (1927)</td>
<td>2.3 M</td>
</tr>
<tr>
<td>Asian Dragon</td>
<td>12.3 M</td>
</tr>
<tr>
<td>The Stanford 3D Scanning Repository</td>
<td></td>
</tr>
</tbody>
</table>
### BVH4 Build Performance on Xeon® and Xeon Phi™

<table>
<thead>
<tr>
<th>Scene</th>
<th>BVH Build*** [triangles/second]</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Xeon* SAH (best BVH quality)</td>
</tr>
<tr>
<td>![Scene Image]</td>
<td>14.9 M</td>
</tr>
<tr>
<td>![Scene Image]</td>
<td>15.6 M</td>
</tr>
<tr>
<td>![Scene Image]</td>
<td>15.0 M</td>
</tr>
</tbody>
</table>

* Intel® Xeon® E5-2690, 8 cores @ 2.9 GHz  ** Intel® Xeon Phi™ 7120, 61 cores @ 1.238 GHz
*** including triangle acceleration structure build, excluding memory allocation time
<table>
<thead>
<tr>
<th>Scene</th>
<th>BVH Build*** [triangles/second]</th>
<th>Speedup</th>
<th>Xeon Phi** Morton (reduced BVH qual)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Xeon* SAH (best BVH quality)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Xeon Phi** SAH (best BVH quality)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Xeon Phi** Morton (reduced BVH qual)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>14.9 M</td>
<td>32.3 M</td>
<td>2.16x</td>
</tr>
<tr>
<td></td>
<td>160.1 M</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>15.6 M</td>
<td>31.7 M</td>
<td>2.03x</td>
</tr>
<tr>
<td></td>
<td>140.4 M</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>15.0 M</td>
<td>35.1 M</td>
<td>2.34x</td>
</tr>
<tr>
<td></td>
<td>162.1 M</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

* Intel® Xeon® E5-2690, 8 cores @ 2.9 GHz
** Intel® Xeon Phi™ 7120, 61 cores @ 1.238 GHz
*** including triangle acceleration structure build, excluding memory allocation time
## BVH4 Build Performance on Xeon® and Xeon Phi™

<table>
<thead>
<tr>
<th>Scene</th>
<th>BVH Build*** [triangles/second]</th>
<th>Speedup</th>
<th>Xeon Phi** Morton (reduced BVH qual)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Xeon* SAH (best BVH quality)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Xeon Phi** SAH (best BVH quality)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><img src="image1.jpg" alt="Image" /></td>
<td>14.9 M</td>
<td>2.16x</td>
<td>160.1 M</td>
</tr>
<tr>
<td><img src="image2.jpg" alt="Image" /></td>
<td>15.6 M</td>
<td>2.03x</td>
<td>140.4 M</td>
</tr>
<tr>
<td><img src="image3.jpg" alt="Image" /></td>
<td>15.0 M</td>
<td></td>
<td>162.1 M</td>
</tr>
</tbody>
</table>

→ Can rebuild 4.3M crown from scratch 3.5 times per second!

* Intel® Xeon® E5-2690, 8 cores @ 2.9 GHz ** Intel® Xeon Phi™ 7120, 61 cores @ 1.238 GHz
*** including triangle acceleration structure build, excluding memory allocation time
## BVH4 Build Performance on Xeon® and Xeon Phi™

<table>
<thead>
<tr>
<th>Scene</th>
<th>BVH Build*** [triangles/second]</th>
<th>Speedup</th>
<th>Xeon Phi** Morton (reduced BVH qual)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Xeon* SAH (best BVH quality)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Xeon Phi** SAH (best BVH quality)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td><strong>Intel® Xeon® E5-2690, 8 cores @ 2.9 GHz</strong></td>
<td><strong>Intel® Xeon Phi™ 7120, 61 cores @ 1.238 GHz</strong></td>
<td><strong>including triangle acceleration structure build, excluding memory allocation time</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>14.9 M</td>
<td>32.3 M</td>
<td>2.16x</td>
</tr>
<tr>
<td></td>
<td>15.6 M</td>
<td>31.7 M</td>
<td>2.03x</td>
</tr>
<tr>
<td></td>
<td>15.0 M</td>
<td>35.1 M</td>
<td></td>
</tr>
</tbody>
</table>

→ ... and over **SEVEN** times per sec on a Xeon Phi!

* Intel® Xeon® E5-2690, 8 cores @ 2.9 GHz ** Intel® Xeon Phi™ 7120, 61 cores @ 1.238 GHz
*** including triangle acceleration structure build, excluding memory allocation time
## BVH Build Performance on Xeon® and Xeon Phi™

<table>
<thead>
<tr>
<th>Scene</th>
<th>BVH Build*** [triangles/second]</th>
<th>Speedup</th>
<th>Xeon Phi** Morton (reduced BVH qual)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Xeon* SAH (best BVH quality)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Xeon Phi** SAH (best BVH quality)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>King</td>
<td>14.9 M</td>
<td>2.16x</td>
<td>160.1 M</td>
</tr>
<tr>
<td>Car</td>
<td>15.6 M</td>
<td>2.03x</td>
<td>140.4 M</td>
</tr>
<tr>
<td>Dragon</td>
<td></td>
<td></td>
<td>162.1 M</td>
</tr>
</tbody>
</table>

→ ... and with reduced quality, at over 35 Hertz

* Intel® Xeon® E5-2690, 8 cores @ 2.9 GHz  ** Intel® Xeon Phi™ 7120, 61 cores @ 1.238 GHz
*** including triangle acceleration structure build, excluding memory allocation time
## Whitted Style Coherent Rays on Xeon® CPUs*

<table>
<thead>
<tr>
<th>Scene</th>
<th>Rendering [rays/second]</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>SSE Single Ray</td>
<td>SSE Packets</td>
</tr>
<tr>
<td>![Image 1]</td>
<td>32.0 M</td>
<td>31.5 M</td>
</tr>
<tr>
<td>![Image 2]</td>
<td>33.5 M</td>
<td>30.0 M</td>
</tr>
<tr>
<td>![Image 3]</td>
<td>33.6 M</td>
<td>31.8 M</td>
</tr>
</tbody>
</table>

→ Packets and SPMD not much help on SSE … but give significant speedup on AVX (wider SIMD)

* Intel® Xeon® E5-2690, 8 cores @ 2.9 GHz
Path Traced Incoherent Rays on Xeon® CPUs*

<table>
<thead>
<tr>
<th>Scene</th>
<th>Rendering [rays/second]</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>SSE Single Ray</td>
<td>SSE Packets</td>
</tr>
<tr>
<td></td>
<td>18.8 M</td>
<td>15.3 M</td>
</tr>
<tr>
<td></td>
<td>22.8 M</td>
<td>17.6 M</td>
</tr>
<tr>
<td></td>
<td>23.1 M</td>
<td>21.3 M</td>
</tr>
</tbody>
</table>

But: Packets/SPMD won’t help much on incoherent rays

* Intel® Xeon® E5-2690, 8 cores @ 2.9 GHz
## Whitted Style Coherent Rays on Xeon Phi™

<table>
<thead>
<tr>
<th>Scene</th>
<th>Rendering [rays/second]</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Xeon Phi* Single Ray</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Xeon Phi* Hybrid</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Speedup</td>
<td></td>
</tr>
<tr>
<td>Xeon Phi*</td>
<td>44.9 M</td>
<td>109.0 M</td>
</tr>
<tr>
<td></td>
<td>Hybrid</td>
<td></td>
</tr>
<tr>
<td></td>
<td>45.2 M</td>
<td>113.8 M</td>
</tr>
<tr>
<td></td>
<td>Hybrid</td>
<td></td>
</tr>
<tr>
<td></td>
<td>50.9 M</td>
<td>122.0 M</td>
</tr>
</tbody>
</table>

On Xeon Phi (16 wide SIMD) Packets/SPMD even more important than on Xeon with AVX → 2.4-2.5x speedup over single ray for coherent rays

* Intel® Xeon Phi™ 7120, 61 cores @ 1.238 GHz
### Path Traced Incoherent Rays on Xeon Phi™

<table>
<thead>
<tr>
<th>Scene</th>
<th>Xeon Phi* Single Ray</th>
<th>Xeon Phi* Hybrid</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image" alt="Crown" /></td>
<td>34.7 M</td>
<td>62.7 M</td>
<td>1.80x</td>
</tr>
<tr>
<td><img src="image" alt="Car" /></td>
<td>38.1 M</td>
<td>75.5 M</td>
<td>1.98x</td>
</tr>
<tr>
<td><img src="image" alt="Dragon" /></td>
<td>47.5 M</td>
<td>87.5 M</td>
<td>1.84x</td>
</tr>
</tbody>
</table>

* Intel® Xeon Phi™ 7120, 61 cores @ 1.238 GHz

... and still a significant 1.8-2x speedup for incoherent rays (⇒ hybrid)
Whitted Style Coherent Rays on Xeon® CPUs and Xeon Phi™

<table>
<thead>
<tr>
<th>Scene</th>
<th>Rendering [rays/second]</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Xeon* AVX Packets</td>
<td>Xeon Phi** Hybrid</td>
</tr>
<tr>
<td><img src="img1.png" alt="Scene" /></td>
<td>41.5 M</td>
<td>109.0 M</td>
</tr>
<tr>
<td><img src="img2.png" alt="Scene" /></td>
<td>48.0 M</td>
<td>113.8 M</td>
</tr>
<tr>
<td><img src="img3.png" alt="Scene" /></td>
<td>43.5 M</td>
<td>122.0 M</td>
</tr>
</tbody>
</table>

Best on Xeon (AVX, packet) vs best on Xeon Phi (hybrid)
→ Xeon Phi up to 2.8x faster than fastest Xeon for coherent rays

* Intel® Xeon® E5-2690, 8 cores @ 2.9 GHz ** Intel® Xeon Phi™ 7120, 61 cores @ 1.238 GHz
# Path Traced Incoherent Rays on Xeon® CPUs and Xeon Phi™

<table>
<thead>
<tr>
<th>Scene</th>
<th>Rendering [rays/second]</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Xeon* Single Ray</td>
<td>Xeon Phi** Hybrid</td>
</tr>
<tr>
<td><img src="image1.png" alt="Scene 1" /></td>
<td>18.8 M</td>
<td>62.7 M</td>
</tr>
<tr>
<td><img src="image2.png" alt="Scene 2" /></td>
<td>22.8 M</td>
<td>75.5 M</td>
</tr>
<tr>
<td><img src="image3.png" alt="Scene 3" /></td>
<td>23.1 M</td>
<td>87.5 M</td>
</tr>
</tbody>
</table>

Best on Xeon (SSE,single) vs best on Xeon Phi (hybrid) → Xeon Phi up to ~3.8x faster than fastest Xeon for incoherent rays

---

* Intel® Xeon® E5-2690, 8 cores @ 2.9 GHz  ** Intel® Xeon Phi™ 7120, 61 cores @ 1.238 GHz
Conclusion

• Kernels for both Xeon® (SSE, AVX) and Xeon Phi™
  • With support for both C/C++ and ISPC, fast BVH builds, etc

• Both single-ray and packet kernels
  • Single-ray for existing single ray based renderers
  • Packet kernels for packet/SPMD-enabled renderers
  • For Xeon Phi™: hybrid kernel that is fast for both coherent and incoherent rays

• One sample renderer each for both C++ and ISPC interfaces
  • Perfect example for how to write a complex renderer in ISPC

→ Application developer can freely chose which hardware architecture (Xeon® /Xeon Phi™) and software (C++/SIMD/ISPC) is best for his particular case!
Questions?

Embree 2.0 available on embree.github.com