Agenda

1. Brief Introduction
2. Dispelling Myths
3. GPU Ray Tracing Facts
4. Out of Core results
5. mental ray GPU results
6. iray futures (time willing)
7. OptiX 2.5 with David McAllister
Evolving Views on GPU Ray Tracing (it)

4 SIG’s ago: The future is ray tracing and a GPU can’t do it

3 SIG’s ago: NVIDIA can do it, but we can’t

2 SIG’s ago: Now everyone can do it

Last SIGGRAPH: Many companies are doing it

This SIGGRAPH: You can do it with most anything

Next year: You can do it anywhere
GPU Ray Tracing Myths

1. The only technique possible on the GPU is path tracing  
   **False**: you’re techniques are only limited by C

2. You can only use (expensive) pro GPUs  
   **False**: GPU computing languages run on all GPUs

3. A GPU farm is more expensive than a CPU farm  
   **False**: perf/dollar is well in GPUs favor

4. A GPU isn’t that much faster than a good CPU  
   **False**: unless you consider 4-12X a quad-core “not much”

5. GPU Ray Tracing is hard  
   **Can be true**: that’s why we created OptiX

6. Your scene has to fit into GPU memory – and that’s finite  
   **True**: until now
Similarities for today’s GPU Ray Tracing

- Speed is linear to GPU cores and clock – for a given GPU architecture. Gains between GPU generations will vary per solution, but they’re BIG.
- Most scale well across system GPUs (“SLI” not needed), but Scaling efficiency will vary per solution and/or technique.
- DP an app choice, ECC a user choice – neither usually needed.
- GPU Computing (ray tracing) steals from system graphics – care is needed to achieve balanced interaction (or multi GPU).
- GTX GPUs – designed for Ultimate Game performance, and not for GPU computing longevity.
- Entire scene must fit onto the GPU’s memory (geometry, textures, acceleration structures) – to work, or go at top speed.
# GPU Computing Overview

## GPU Computing Applications

<table>
<thead>
<tr>
<th>CUDA C/C++</th>
<th>OpenCL</th>
<th>Direct Compute</th>
<th>Fortran</th>
<th>Python, Java, .NET, ...</th>
</tr>
</thead>
<tbody>
<tr>
<td>Over 90,000 developers</td>
<td>1st GPU demo</td>
<td>Microsoft API for GPU Computing</td>
<td>PGI Accelerator</td>
<td>PyCUDA</td>
</tr>
<tr>
<td>Running in Production since 2008</td>
<td>Shipped 1st OpenCL Conformant Driver</td>
<td>Supports all CUDA-Architecture GPUs (DX10 and DX11)</td>
<td>PGI CUDA Fortran</td>
<td>jCUDA</td>
</tr>
<tr>
<td>SDK + Libs + Visual Profiler and Debugger</td>
<td>Public Availability</td>
<td>NOAA Fortran bindings</td>
<td>CUDA.NET</td>
<td>CUDA.NET</td>
</tr>
</tbody>
</table>

**NVIDIA GPU**
with the CUDA Parallel Computing Architecture

---

**Broad Adoption**

- Over 250,000,000 installed CUDA-Architecture GPUs
- Over 100,000 GPU Computing Developers
- Windows, Linux and MacOS Platforms supported
- GPU Computing spans HPC to Consumer
- 250+ Universities teaching GPU Computing on the CUDA Architecture

© 2010
<table>
<thead>
<tr>
<th>Programming Approach</th>
<th>Programming Language</th>
</tr>
</thead>
<tbody>
<tr>
<td>iray</td>
<td>CUDA C, C Runtime</td>
</tr>
<tr>
<td>finalRender</td>
<td>CUDA C, C Runtime</td>
</tr>
<tr>
<td>Furry Ball</td>
<td>CUDA C, C Runtime</td>
</tr>
<tr>
<td>Arion</td>
<td>CUDA C, driver API</td>
</tr>
<tr>
<td>Octane</td>
<td>CUDA C, driver API</td>
</tr>
<tr>
<td>V-Ray RT (in our booth)</td>
<td>OpenCL</td>
</tr>
<tr>
<td>Brazil</td>
<td>OpenCL</td>
</tr>
<tr>
<td>CentiLeo</td>
<td>CUDA C, driver API</td>
</tr>
<tr>
<td>Panta Ray</td>
<td>CUDA C, driver API</td>
</tr>
<tr>
<td>OptiX (2.5)</td>
<td>CUDA C, driver API &amp; PTX</td>
</tr>
<tr>
<td>Adobe Research (in our booth)</td>
<td>CUDA C, OptiX API</td>
</tr>
<tr>
<td>Works Zebra, etc. (at our booth)</td>
<td>CUDA C, OptiX API</td>
</tr>
<tr>
<td>mental ray</td>
<td>CUDA C, OptiX API</td>
</tr>
</tbody>
</table>
Solutions Vary in their GPU Exploitation

- Speed-ups vary, but a top end Fermi GPU will typically ray trace 6 to 15 times faster than on a quad-core CPU

- A GPGPU programming challenge is to keep the GPU “busy”

- Gains on complex tasks often greater than for simple ones

- Particularly evident with multiple GPUs, where data transfers impact simple tasks more

- Can mean the technique needs to be rethought in how it’s scheduling work for the GPU

- OptiX 2.1 example – first tuned for simple, now tuned for complex, with a 30-80% speed increase
NVIDIA Goals for Advanced Rendering

- Make the GPU an essential part of *ALL advanced rendering*
- Create engines and libraries to make it easier for everyone to exploit the GPU (e.g., OptiX); *learning what’s needed for future GPU languages and architectures*
- Create compelling commercial offerings to spur GPU adoption (e.g., iray); *learning what’s needed for success, so to help other developers and improve the GPU platform for it*
- If you’re developing solutions for Advanced Rendering, *NVIDIA wants to help*
NVIDIA ARC: mental ray AO

- mental ray 3.9 code & pipeline accelerated w/ OptiX

~21mil faces

+3 minutes
2 CPU

+18X

20 seconds
1 GPU

Subsequent frames can be *far* faster yet...

Model courtesy NVIDIA Creative

Rendered with mental ray 3.9
NVIDIA iray: roadmap

- **iray 2** now
  - Layered Material Model
    (car paint, subsurface scattering, decals, etc.)
  - Increased Performance & Interactivity, Daylight Portals, Clipping Planes, more...

- **iray 3** next year
  - Better convergence for more difficult lighting conditions
  - Increased interactivity...
  - Increased flexibility for production use cases...
  - Much more in the works...
Agenda

1. What is OptiX
2. Why OptiX
3. What’s New
NVIDIA ARC: OptiX ray tracing engine

- A ray tracing framework for developers
  - Similar to OpenGL in doing the “heavy lifting” of ray tracing and leaving capability and technique to the developers
  - Very general and applicable to many markets
  - Proven to speed development as well as performance

- Being used by Adobe in our booth
- Being used internally in our commercial software
The Beauty of OptiX as Middleware
The Beauty of OptiX as Middleware

- *We do the heavy lifting for you.*
The Beauty of OptiX as Middleware

- We handle acceleration structure build and traversal.
The Beauty of OptiX as Middleware

- We handle multiple GPUs transparently.
The Beauty of OptiX as Middleware

- We handle thread scheduling and reconvergence.
The Beauty of OptiX as Middleware

- We automatically page datasets too large for GPU memory.
The Beauty of OptiX as Middleware

- We enable interactive scene editing via fast rebuilds
The Beauty of OptiX as Middleware

- We virtualize texture resources and use software texturing as needed.
Life of a ray

1. Ray Generation
2. Intersection
3. Shading
RT_PROGRAM void pinhole_camera() {
    float2 d = make_float2(launch_index) / make_float2(launch_dim) * 2.f - 1.f;
    float3 ray_origin = eye;
    float3 ray_direction = normalize(d.x*U + d.y*V + W);
    optix::Ray ray = optix::make_Ray(ray_origin, ray_direction, scene_epsilon, RT_DEFAULT_MAX);
    PerRayData_radiance prd;
    rtTrace(top_object, ray, prd);
    output_buffer[launch_index] = make_color( prd.result );
}

RT_PROGRAM void intersect_sphere() {
    float3 O = ray.origin - center;
    float3 D = ray.direction;
    float b = dot(O, D);
    float c = dot(O, O) - radius*radius;
    float disc = b*b - 4.c;
    if(disc > 0.0f){
        float sdisc = sqrtf(disc);
        float root1 = (-b - sdisc) / 2.f;
        shading_normal = geometric_normal = (O + root1*D)/radius;
        if( rtPotentialIntersection(0) ) {
            shading_normal = geometric_normal = (O + root1*D)/radius;
            if( rtReportIntersection(0) ){
                shading_normal = geometric_normal = (O + root1*D)/radius;
                rtReportIntersection(0);
            }
        }
        if(check_second){
            float root2 = (-b + sdisc) / 2.f;
            if( rtPotentialIntersection( root2 ) ) { 
                shading_normal = geometric_normal = (O + root2*D)/radius;
                rtReportIntersection(0);
            }
        }
    }
}

RT_PROGRAM void closest_hit_radiance3() {
    float3 world_geo_normal   = normalize( rtTransformNormal( RT_OBJECT_TO_WORLD, geometric_normal ) );
    float3 world_shade_normal = normalize( rtTransformNormal( RT_OBJECT_TO_WORLD, shading_normal ) );
    float3 ffnormal     = faceforward( world_shade_normal, ray.direction, world_geo_normal );
    float3 color = Ka * ambient_light_color;
    float3 hit_point = ray.origin + t_hit * ray.direction;
    for(int i = 0; i < lights.size(); ++i){
        BasicLight light = lights[i];
        float3 L = normalize(light.pos - hit_point);
        float nDl = dot( ffnormal, L);
        if( nDl > 0.0f ){ // cast shadow ray
            PerRayData_shadow shadow_prd;
            float Ldist = length(light.pos - hit_point);
            optix::Ray shadow_ray( hit_point, L, shadow_ray_type, scene_epsilon, Ldist );
            rtTrace(top_shadower, shadow_ray, shadow_prd);
            float3 light_attenuation = shadow_prd.attenuation;
            if( fmaxf(light_attenuation) > 0.0f ){ // has attenuation
                float3 Lc = light.color * light_attenuation;
                color += Kd * nDl * Lc;
                float3 H = normalize(L - ray.direction);
                float nDh = dot( ffnormal, H );
                if(nDh > 0) {
                    color += Kn * Lc * pow(nDh, phong_exp);
                }
            }
        }
    }
    prd_radiance.result = color;
}
Program objects (shaders)

- Input “language” is based on CUDA C/C++
- No new language to learn
- Powerful language features available immediately
- Can also take raw PTX as input
- Interconnection of programs defines the outcome
- Data associated with ray is programmable
- Caveat: still need to use it responsibly to get performance

```c
RT_PROGRAM void pinhole_camera()
{
    float2 d = make_float2(launch_index) / make_float2(launch_dim) * 2.f - 1.f;
    float3 ray_origin = eye;
    float3 ray_direction = normalize(d.x*U + d.y*V + W);

    optix::Ray ray = optix::make_Ray(ray_origin, ray_direction,
     radiance_ray_type, scene_epsilon, RT_DEFAULT_MAX);

    PerRayData_radiance prd;
    rtTrace(top_object, ray, prd);
    output_buffer[launch_index] = make_color( prd.result );
}
```
# Programmable Operations

<table>
<thead>
<tr>
<th>Rasterization</th>
<th>Ray Tracing</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fragment</td>
<td>Closest Hit</td>
</tr>
<tr>
<td>Vertex</td>
<td>Any Hit</td>
</tr>
<tr>
<td>Geometry</td>
<td>Intersection</td>
</tr>
<tr>
<td>Hull/Domain (Tessellation)</td>
<td>Selector</td>
</tr>
<tr>
<td></td>
<td>Ray Generation</td>
</tr>
<tr>
<td></td>
<td>Miss</td>
</tr>
<tr>
<td></td>
<td>Exception</td>
</tr>
</tbody>
</table>
Acceleration Structure Choices

- **Sbvh**: Splits slivery triangles  
  *Fastest ray tracing*
- **Bvh**: Standard CPU build
- **MedianBvh**: Fast CPU build
- **Lbvh**: Built on GPU  
  *Fastest build time*
Fast BVH Build on GPU

- Fairy Forest (174K triangles) – 4.8ms
- Turbine Blade (2M triangles) – 10.5ms
- Power Plant (12M triangles) – 62ms
Large Dataset Paging

- App is not written to be paging aware
- Handles models somewhat larger than GPU memory
- Modest slowdown vs. in-core

- App is written for large datasets
- Handles models 3X size of GPU memory
- More substantial slowdown but fast vs. CPU
• Create SW page table
• Rewrite LD/ST instructions to virtualize
• On page hit, translate virtual address and load it
• On page miss save thread state to GPU memory and pause ray, grab a new ray
• Copy requested pages from CPU to GPU on page fault
• Restart kernel, restore state, resume rendering
OptiX Roadmap

Version 2.5  later this year
- Out-of-core support, paging to system RAM
- Unlimited number of textures
- Very fast device-side BVH builds
- Speedups!

Version 3  first half of next year
- Optimized for Kepler GPU Architecture
- CPU fallback (for interactive rendering)
- Why? - required by major commercial products (including NVIDIA ARC’s)
Preliminary tests with OptiX 2.5

Texture Out of Core (Whitted)

Geometry Out of Core (with AO)

# of 4k Images

 Millions of Textured & Smoothed Faces

Quadro 6000 = 6GB on board memory
Quadro 5000 = 2.5GB on board memory