$$I(x, x') = g(x, x') \left[ \epsilon(x, x') + \int_{s} \rho(x, x', x'')I(x', x'')dx'' \right]$$

Welcome!
Today's Agenda:

- Introduction
- GPU Architecture
- OpenCL Primer
- My First GPU Ray Tracer
- One Does Not Simply
Introduction

Supercomputing for the Masses*

---

Introduction

Supercomputing for the Masses

GPUs also have substantially more bandwidth:

AMD R9 Fury: 512GB/s
NVidia TitanX: 336.5GB/s
Intel Xeon: 118GB/s

Ray tracing requires compute power as well as bandwidth.
Introduction

Supercomputing for the Masses

GPUs have a unique way of dealing with latencies:

- CPUs rely on caches to reduce average memory access time
- GPUs rely on massive parallelism to hide latencies.

The CPU approach works well if memory access is somewhat coherent.

The GPU approach works well if there is sufficient parallelism (memory access coherence is irrelevant).
Introduction

Supercomputing for the Masses

And finally, GPUs are optimized for graphics. Although ray tracing is a GPGPU task, we still benefit:

- Texture filtering hardware
- Very fast sin/cos/tan and square root
- Efficient conversion between float and int
- Efficient interpolation and clamping
- Fast interop with OpenGL / DirectX

The GPU is a perfect match for ray tracing...

...but it comes with some peculiarities.
Today’s Agenda:

- Introduction
- GPU Architecture
- OpenCL Primer
- My First GPU Ray Tracer
- One Does Not Simply
GPU Architecture

**CPU**
- Small number of cores
- Optimized for generic tasks
- Hyperthreading: two threads per core

**GPU**
- Small number of cores (‘multiprocessors’)
- Optimized for parallel tasks
- Many threads per core, grouped in *warps*
Warp:

32 threads, running the same instructions in lock step – *SIMT*. In case of delays, the stalled warp is swapped for another warp.

A single multiprocessor manages several warps (up to 64). Switching between warps is governed by the hardware. Each warp (active or not) has its own registers; switching is ‘instant’.

A modern GPU can execute 4 warps simultaneously (while 60 wait). A modern GPU can have up to 24 multiprocessors.
GPU Architecture

Feeding the Beast

How do we feed such a processor sufficient work?

*We feed it many identical tasks.*

For a ray tracer:

- One thread per pixel.
GPU Architecture

GPU Memory Model

NVidia Maxwell architecture*:

Registers: 65536 per multiprocessor
1 cycle access time

Shared memory: 96Kb per multiprocessor
28 cycles

L1/texture cache: 24Kb per multiprocessor
1 cycle

L2 cache: 2MB
194 cycles

Global memory: X GB
350 cycles

Host memory: X GB

PCIe 3.0 Bandwidth: ~15GB/s

GPU Memory Model

Consequences of the memory model:

1. Caches are either very small (L1) or slow (L2).
2. This is compensated by ‘shared memory’, which we have to manage manually.
3. The memory hierarchy is (at least partially) \textit{explicit} rather than implicit as on the CPU.
4. We have to trade registers per thread for number of threads: at 2048 threads per multiprocessor, each thread can use only 32 registers (a single float4 is four registers). Beyond this count, ‘register spilling’ occurs.

\[ \Rightarrow \text{It’s probably better to feed the GPU small programs.} \]
\[ \Rightarrow \text{We have to be really careful when spending memory.} \]
Today's Agenda:

- Introduction
- GPU Architecture
- OpenCL Primer
- My First GPU Ray Tracer
- One Does Not Simply
Preparing to Run OpenCL Code

Step 1:
Update your drivers!

Step 2:
Weak / no GPU? Download CPU drivers:
(Select the OpenCL Runtime 15.1 version for your OS.)

Step 3:
If anything goes wrong (build freezes, crashes):
Update your drivers.
Template with OpenCL Support

In Game::Init:
Instantiate a Kernel. For the first kernel, this invokes InitCL (in opencl.cpp).

InitCL does the following:

- finds a GPU OpenCL platform in your system
- falls back to a CPU device if no OpenCL capable GPU is found
- loads OpenCL code from ‘../../../program.cl’
- builds the OpenCL code
- reports on errors in the console window.

Result:

- an OpenCL context (ComputeContext)
- a command queue (ComputeCommandQueue)
- a kernel (ComputeKernel).
Kernel

A kernel contains the code we will be executing in each thread.

Example:

```c
__kernel void device_function(__global int* a)
{
    // get thread id
    int idx = get_global_id(0);
    int idy = get_global_id(1);
    // calculate pixel index
    int id = idx + 512 * idy;
    // check if we're within the workset
    if (id >= (512 * 512)) return;
    // do calculations
    uint color = ((idx >> 1) << 16) + ((idy >> 1) << 8);
    // send result to output array
    a[id] = color;
}
```
Kernel

The kernel needs a buffer to write to. In this case:

- the buffer
- on the device
- needs to be copied to the host.

We create the buffer using OpenCL via the template:

```c
Buffer buffer = new Buffer( 512 * 512 );
```
Kernel

The kernel has a single argument. Since all threads will use the same argument, we set it once for the kernel:

```
kernel->SetArgument( 0, buffer );
```

Now we are ready to invoke the kernel:

```
kernell-Run( buffer );
```

Note:

- Kernel::Run first waits for OpenGL to finish
- we tell the GPU how many parallel tasks we have (workSize)
- we wait for the GPU to complete the work.
Kernel

Once the GPU has executed the work, the data is still on the GPU. We get it back to the host over the PCI bus:

```c
buffer->CopyFromDevice();
uint* data = buffer->GetHostPtr();
```

After this, we can operate on `int[]` data as usual.

```c
__kernel void device_function( __global int* a )
{
    // get thread id
    int idx = get_global_id( 0 );
    int idy = get_global_id( 1 );
    // calculate pixel index
    int id = idx + 512 * idy;
    // check if we're within the workset
    if (id >= (512 * 512)) return;
    // do calculations
    uint color = ... ;
    // send result to output array
    a[id] = color;
}
```

Buffer buffer = new Buffer( 512 * 512 );
kernel->SetArgument( 0, buffer );
kernel->Run( buffer );
Summary

Executing code on the GPU:

1. Initialize an OpenCL context;
2. Load and build OpenCL code to obtain a kernel;
3. Create input and output buffers;
4. Set kernel parameters;
5. Execute the kernel;
6. Get results from the GPU to the host.

```c
InitCL();

__kernel void device_function( __global int* a )
{
    // get thread id
    int idx = get_global_id( 0 );
    int idy = get_global_id( 1 );
    // calculate pixel index
    int id = idx + 512 * idy;
    // check if we're within the workset
    if (id >= (512 * 512)) return;
    // do calculations
    uint color = ...;
    // send result to output array
    a[id] = color;
}

Buffer buffer = new Buffer( 512 * 512 );
kernel->SetArgument( 0, buffer );
kernel->Run( buffer );
buffer->CopyFromDevice();
uint* data = buffer->GetHostPtr();
```
OpenCL Primer

Hidden Things

OpenCL takes care of:

1. Sending kernel code to the GPU;
2. Sending a ReadWrite buffer to the GPU;
3. Allocating memory in GPU global memory;
4. Distributing work items over hardware threads.

```c
InitCL();
__kernel void device_function( __global int* a )
{
    // get thread id
    int idx = get_global_id( 0 );
    int idy = get_global_id( 1 );
    // calculate pixel index
    int id = idx + 512 * idy;
    // check if we're within the workset
    if (id >= (512 * 512)) return;
    // do calculations
    uint color = ...;
    // send result to output array
    a[id] = color;
}
Buffer buffer = new Buffer( 512 * 512 );
kernel->SetArgument( 0, buffer );
kernel->Run( buffer );
buffer->CopyFromDevice();
uint* data = buffer->GetHostPtr();
```
Distributing Work over Hardware

We specified the amount of work when invoking queue.Execute:

```c
clEnqueueNDRangeKernel( queue, kernel, 2, 0, workSize, null, null, null, null );
```

Here, workSize is an array with 1, 2 or 3 elements. In our case, we want to work on a 512x512 set, so a good workSize is `{ 512, 512 }`.

We have control over the number of threads per multiprocessor, via a ‘local worksize’:

```c
long workSize[] = { 512, 512 };
long localSize[] = { 32 * 32 };
clEnqueueNDRangeKernel( queue, kernel, 2, 0, workSize, localSize, null, null, null );
```

Here we tell OpenCL to start 32 full warps on each multiprocessor.
Optimal Work Size / Local Size

The ideal number of threads we sent to each multiprocessor depends on:

1. The total amount of work (don’t leave some multiprocessors idling);
2. The complexity of our code (prevent register spilling);
3. The capabilities of the GPU (not every GPU can handle 32 warps).

Especially ‘2’ results in practice in manual tuning to get the best result.

GPU capabilities result in different work sizes per GPU.

(Obviously, this only affects performance, not correct operation.)
Today’s Agenda:

- Introduction
- GPU Architecture
- OpenCL Primer
- My First GPU Ray Tracer
- One Does Not Simply
Example: Simple Ray Tracer

```c
__kernel void TraceKernel( write_only image2d_t outimg )
{
    // get thread id
    int x = get_global_id( 0 );
    int y = get_global_id( 1 );
    int id = x + 512 * y;
    if (id >= (512 * 512)) return;

    // do calculations
    float3 color = Trace( x, y );
    // send result to output array
    write_imagef( outimg, (int2)(x, y), (float4)(color, 1) );
}
```

```c
// get survival probability; diffuse scattering - doing it properly, classical;
if ( (Radiance.x + Radiance.y + Radiance.z) > 0 ) // ensure
    E = true;
else
    E = false;

// for bounce n + 1
if ( scatPDF > EvaluatedScler( L, N ) ) * Survive;
    at = EvaluatedScatter ( L, N, n + 1 );
    if ( at > MI )
        return;
    E = (weight * cosThetaInc / directPDF) * (transmit); // march
    if ( E )
        return;
    // bounce walk - done properly, closely following radiance
    trace_brdf = SampleDiffuse( diffuse, N, c1, c2, RR, E);
    RR = 1.0 - (trace_brdf.a);
    pdf = trace_brdf.E;
    h = E * bvec3( dot( N, R ) / pdf );

    if ( h > 0.0 )
```
Example: Simple Ray Tracer

```cpp
float3 Trace( int x, int y )
{
    return (float3)( 0, 0, 0 );
}

__kernel void TraceKernel( write_only image2d_t outimg )
{
    ...
    float3 color = Trace( x, y );
    ...
}
```


Example: Simple Ray Tracer

```c
struct Ray { float3 O, D; float t; }

Ray GeneratePrimaryRay(int x, int y) {
    Ray r;
    r.O = (float3)( 0, 0, 0 );
    r.D = (float3)( 0, 0, 1 );
    r.t = 1;
    return r;
}

float3 Trace( int x, int y ) {
    Ray r = GeneratePrimaryRay();
    // intersect with scene
    // translate to color
    return (float3)( 0, 0, 0 );
}
```
Example: Simple Ray Tracer

```c
struct Ray { float3 O, D; float t; };

struct Ray GeneratePrimaryRay( int x, int y )
{
    struct Ray r;
    r.O = (float3)( 0, 0, 0 ); // for now
    r.D = (float3)( 0, 0, 1 ); // for now
    r.t = 1;
    return r;
}

float3 Trace( int x, int y )
{
    // generate primary ray
    struct Ray r = GeneratePrimaryRay( x, y );
    // intersect with scene
    // translate to color
    return (float3)( 0, 0, 0 );
}
```
Example: Simple Ray Tracer

typedef struct { float3 O, D; float t; } Ray;

Ray GeneratePrimaryRay( int x, int y )
{
    Ray r;
    r.O = (float3)( 0, 0, 0 ); // for now
    r.D = (float3)( 0, 0, 1 ); // for now
    r.t = 1;
    return r;
}

float3 Trace( int x, int y )
{
    // generate primary ray
    Ray r = GeneratePrimaryRay();
    // intersect with scene
    // translate to color
    return (float3)( 0, 0, 0 );
Example: Simple Ray Tracer

```c
Ray GeneratePrimaryRay( int x, int y )
{
    Ray r;
    float3 p1 = (float3)( -1, -1, -1 ); // top left
    float3 p2 = (float3)(  1, -1, -1 ); // top right
    float3 p3 = (float3)( -1,  1, -1 ); // bottom left
    float fx = (float)x / 512;
    float fy = (float)y / 512;
    float3 P = p1 + (p2 - p1) * fx + (p3 - p1) * fy;
    r.O = (float3)( 0, 0, 0 );
    r.D = normalize( P - r.O );
    r.t = 1e34f;
    return r;
}
```
Example: Simple Ray Tracer

Ray GeneratePrimaryRay( int x, int y, float3 pos, float3 target )
{
    Ray r;
    float3 E = normalize( target - pos );
    float3 up = (float3)( 0, 1, 0 );
    float3 right = normalize( cross( up, E ) );
    up = cross( E, right );
    float3 C = pos + E;
    float3 p1 = C - right + up; // top left
    float3 p2 = C + right + up; // top right
    float3 p3 = C - right - up; // bottom left
    float fx = (float)x / 512;
    float fy = (float)y / 512;
    float3 P = p1 + (p2 - p1) * fx + (p3 - p1) * fy;
    r.D = normalize( P - E );
    r.O = pos;
    r.t = 1e34f;
    return r;
}
Example: Simple Ray Tracer

```c
float3 Trace( int x, int y, float3 pos, float3 target )
{
    // generate primary ray
    Ray r = GeneratePrimaryRay( x, y, pos, target );
    // intersect with scene
    // translate to color
    return (float3)( r.D.x, r.D.y, r.D.z );
}
```

```c
__kernel void TraceKernel( write_only image2d_t outimg, float3 pos, float3 target )
{
    ...
}
```

In the current setup, the screen corners used for interpolation are recalculated for every ray.

This is where the ‘massive parallelism’ paradigm breaks: we want to do this once.

Solution: calculate the screen corners on the CPU, and pass them to the kernel.

Next problem: how do we pass a float3? It is generally not safe to pass complex types.
Example: Simple Ray Tracer

typedef struct
{
    float3 pos;    // 16 bytes
    float3 target; // 16 bytes
    float3 p1;     // 16 bytes
    float3 p2;     // 16 bytes
    float3 p3;     // 16 bytes
}
RenderData;

// DEVICE SIDE DATA

RenderData rd;
kernel.SetArgument( 1, rd );

__kernel void TraceKernel( write_only image2d_t outimg, RenderData rd )
{
    ... }

struct RenderData
{
    float posx, posy, posz, dummy1;
    float targetx, targety, targetz, dummy2;
    float p0x, p0y, p0z, dummy3;
    float p1x, p1y, p1z, dummy4;
    float p2x, p2y, p2z, dummy5;
};
Advanced Graphics – GPGPU Recap

My First

Take-away: OpenCL

1. Everything in the kernel happens for all threads
   \textit{Do initialization on the CPU}

2. Local data in the kernel exists for all active threads
   \textit{Don’t create your scene in a kernel}

3. Data beyond basic types cannot be safely transferred between host and GPU
   \textit{Consider using a ‘render state’ to easily pass complex data}

4. No ‘by reference’ passing
   \textit{Use pointers instead}

5. Debugging is hard
   \textit{Build incrementally; think of ways to visualize intermediate results}

6. OpenCL isn’t magic
   \textit{Most of the code will be very similar to a C++/C# version}

7. Convert to OpenCL; don’t develop in OpenCL
   \textit{Make it work first}
Today’s Agenda:

- Introduction
- GPU Architecture
- OpenCL Primer
- My First GPU Ray Tracer
- One Does Not Simply
Consequences

Food For Thought

How do you implement packet traversal on a GPU?

**You don’t.**

You don’t have to.

A warp contains 32 threads, which are responsible for 32 primary rays. This *is* a packet.

Suppose a multiprocessor works on 8 warps. It is processing 256 rays. This is a large packet.

- Efficiency is affected by ray coherence.
- The packet behavior is implicit.
Food For Thought

- Efficiency is affected by ray coherence.

Recall that threads in a warp execute the same instructions, in lock-step.

Conditional code is implemented (invisible to the programmer) using *masking*, which results in reduced *occupancy*.

We thus benefit when threads exhibit flow coherence.

This is a good reason for having a 2D `workSize`.

You will still benefit from a Morton curve! A warp will be 2 tiles of 4x4 rays.

```c
z = 5;
if (cond1)
    x = y + z;
if (cond2)
    w = t + u;
else
    w = t - u;
p = q + r;
else
    x = y - z;
x++;
```
Consequences

Food For Thought

What if a thread skips a large amount of work?

Scenario:

Of the 512x512 primary rays, ~50% wants to cast a shadow ray to the only point light source in the scene.

Approximately 10% of the primary rays require reflection rays.

The primary rays hit three very distinct materials, evenly distributed over the scene.
Consequences

Food For Thought

It’s not easy to build a BVH on the GPU.

BVH construction starts with finding the root split plane. Only after that we can work on two subtrees, in parallel.

Solution:

Maintain the BVH on the CPU.
Do this in parallel with rendering: build the BVH for the next frame.

(the alternative is an idling CPU anyway)
Consequences

Food For Thought

Primary rays are easily handled by rasterization hardware.

But: you shouldn't bother.

1. Primary rays are the cheapest rays (coherent!);
2. Splitting rendering in two phases is a complication;
3. Rasterization requires a pinhole camera.
Recommended literature:

You’ll find that many ‘tutorials’ explain mostly how to get a minimal OpenCL program running – focusing on setting up the environment rather than actual OpenCL coding.

OpenCL Tutorial – N-Body Simulation, Brown Deer Technology

Programming and Simulating Heterogeneous Devices – Ubal et al., 2012

Let me know if you find other good resources.
Today's Agenda:

- Introduction
- GPU Architecture
- OpenCL Primer
- My First GPU Ray Tracer
- One Does Not Simply
INFOMAGR – Advanced Graphics
Jacco Bikker - November 2017 - February 2018

END of “GPGPU Recap”
next lecture: “GPU Path Tracing (1)"