GOING PARALLEL

Coding for massively parallel GPUs

RAY TRACING: EMBARRASSINGLY PARALLEL

- Defined: Little to no effort needed to separate into parallel tasks

RAY TRACING: EMBARRASSINGLY PARALLEL

- Defined: Little to no effort needed to separate into parallel tasks

- Rendering often a prototypical example of *embarrassingly parallel*
  - One obvious way: assign one CPU or GPU core per pixel

RAY TRACING: EMBARRASSINGLY PARALLEL

- On CPU, call fork() or spawn() to create multiple threads
  - Each thread works on separate pixels
  - Wait for all threads to complete
  - Some threads take longer → may need load balancing

RAY TRACING: EMBARRASSINGLY PARALLEL

- GPU programming model hides thread spawning and load-balancing
  - Code *appears* serial, but you have access to current pixel index

RAY TRACING: EMBARRASSINGLY PARALLEL

- GPU programming model hides thread spawning and load-balancing
  - Code *appears* serial, but you have access to current pixel index

```
// Simple DirectX-like ray tracing shader
RWTexture<float4> rayColors;

[shader("raygeneration")]
void SimpleRayTracer() {
   uint2       curPixel = DispatchRaysIndex().xy;
   RayDesc     ray      = { GetRayOrigin(curPixel), 0.0f, GetRayDir(curPixel), 1e+38f };
   RayPayload  payload  = { float3(0, 0, 0) };
   TraceRay( ..., ray, payload );
   rayColors[curPixel]  = float4( payload.rayColor, 1.0f );
}
```

RAY TRACING: EMBARRASSINGLY PARALLEL

- GPU programming model hides thread spawning and load-balancing
  - Code *appears* serial, but you have access to current pixel index

```
// Simple DirectX-like ray tracing shader
RWTexture<float4> rayColors;

[shader("raygeneration")]
void SimpleRayTracer() {
   uint2       curPixel = DispatchRaysIndex().xy;
   RayDesc     ray      = { GetRayOrigin(curPixel), 0.0f, GetRayDir(curPixel), 1e+38f };
   RayPayload  payload  = { float3(0, 0, 0) };
   TraceRay( ..., ray, payload );
   rayColors[curPixel]  = float4( payload.rayColor, 1.0f );
}
```

Identify the current pixel

RAY TRACING: EMBARRASSINGLY PARALLEL

- GPU programming model hides thread spawning and load-balancing
  - Code *appears* serial, but you have access to current pixel index

```
// Simple DirectX-like ray tracing shader
RWTexture<float4> rayColors;

[shader("raygeneration")]
void SimpleRayTracer() {
   uint2       curPixel = DispatchRaysIndex().xy;
   RayDesc     ray      = { GetRayOrigin(curPixel), 0.0f, GetRayDir(curPixel), 1e+38f };
   RayPayload  payload  = { float3(0, 0, 0) };
   TraceRay( ..., ray, payload );
   rayColors[curPixel]  = float4( payload.rayColor, 1.0f );
}
```

Setup the ray

RAY TRACING: EMBARRASSINGLY PARALLEL

- GPU programming model hides thread spawning and load-balancing
  - Code *appears* serial, but you have access to current pixel index

```
// Simple DirectX-like ray tracing shader
RWTexture<float4> rayColors;

[shader("raygeneration")]
void SimpleRayTracer() {
   uint2       curPixel = DispatchRaysIndex().xy;
   RayDesc     ray      = { GetRayOrigin(curPixel), 0.0f, GetRayDir(curPixel), 1e+38f };
   RayPayload  payload  = { float3(0, 0, 0) };
   TraceRay( ..., ray, payload );
   rayColors[curPixel]  = float4( payload.rayColor, 1.0f );
}
```

Initialize ray return values

RAY TRACING: EMBARRASSINGLY PARALLEL

- GPU programming model hides thread spawning and load-balancing
  - Code *appears* serial, but you have access to current pixel index

```
// Simple DirectX-like ray tracing shader
RWTexture<float4> rayColors;

[shader("raygeneration")]
void SimpleRayTracer() {
   uint2       curPixel = DispatchRaysIndex().xy;
   RayDesc     ray      = { GetRayOrigin(curPixel), 0.0f, GetRayDir(curPixel), 1e+38f };
   RayPayload  payload  = { float3(0, 0, 0) };
   TraceRay( ..., ray, payload );
   rayColors[curPixel]  = float4( payload.rayColor, 1.0f );
}
```

Trace your ray

RAY TRACING: EMBARRASSINGLY PARALLEL

- GPU programming model hides thread spawning and load-balancing
  - Code *appears* serial, but you have access to current pixel index

```
// Simple DirectX-like ray tracing shader
RWTexture<float4> rayColors;

[shader("raygeneration")]
void SimpleRayTracer() {
   uint2       curPixel = DispatchRaysIndex().xy;
   RayDesc     ray      = { GetRayOrigin(curPixel), 0.0f, GetRayDir(curPixel), 1e+38f };
   RayPayload  payload  = { float3(0, 0, 0) };
   TraceRay( ..., ray, payload );
   rayColors[curPixel]  = float4( payload.rayColor, 1.0f );
}
```

Output your results

DOING IT YOURSELF V.S. EXISTING APIS

- Embarrassingly parallel ≠ easy to parallelize

DOING IT YOURSELF V.S. EXISTING APIS

- Embarrassingly parallel ≠ easy to parallelize
  - — ***Not*** the same as getting best performance

DOING IT YOURSELF V.S. EXISTING APIS

- Embarrassingly parallel ≠ easy to parallelize
  - ***Not*** the same as getting best performance
- Many performance considerations:
  - # intersections, data structures, coherence, caching, load-balancing, SIMD

DOING IT YOURSELF V.S. EXISTING APIS

- Embarrassingly parallel ≠ easy to parallelize
  - — ***Not*** the same as getting best performance
- Many performance considerations:
  - — # intersections, data structures, coherence, caching, load-balancing, SIMD
- APIs can leverage best-known methods behind your back

DOING IT YOURSELF V.S. EXISTING APIS

- Embarrassingly parallel ≠ easy to parallelize
  - — ***Not*** the same as getting best performance
- Many performance considerations:
  - — # intersections, data structures, coherence, caching, load-balancing, SIMD
- APIs can leverage best-known methods behind your back
- APIs allow you to shoot yourself in the foot without knowing it

DOING IT YOURSELF V.S. EXISTING APIS

- Embarrassingly parallel ≠ easy to parallelize
  - — ***Not*** the same as getting best performance
- Many performance considerations:
  - — # intersections, data structures, coherence, caching, load-balancing, SIMD
- APIs can leverage best-known methods behind your back
- APIs allow you to shoot yourself in the foot without knowing it
- APIs come at many levels (e.g., use of CUDA without ray tracing API)

SOME RAY TRACING APIS

- Hardware vendor specific:
  - OptiX, Embree, FireRays
- Cross-vendor APIs:
  - DirectX Raytracing, Vulkan RT
- Game engine APIs:
  - Unity, Unreal
- Different:
  - Audiences, learning curves, flexibility, performance, built-in optimizations

TODAY: USING DIRECTX FOR SAMPLE CODE

- Why?
  - DirectX widely used API for interactive graphics
  - Similar to Vulkan model
  - Abstracts some bits tricky for novices' ray tracers
  - Tutorial frameworks for easy experimentation

DIRECTX RAY TRACING RESOURCES

- Some DirectX Ray Tracing tutorials:
  - Tutorial framework that hides the C++ API (http://intro-to-dxr.cwyman.org)
    - Easy to get started, not targeted at optimal performance
    - Used for my sample code today
    - Builds on Falcor for abstraction
  - Lower-level tutorial covering DirectX API
    - From the "Introduction to DirectX Ray Tracing" *Ray Tracing Gems* article
  - A simple getting started blog post
  - Microsoft's DXR samples
    - A DirectX Raytracing functional specification

WE'LL FOCUS ON GPU SHADER CODE

- Why?
  - Focus on tracing rays, identifying where to trace rays
  - Where interesting rendering algorithms mostly live

WE'LL FOCUS ON GPU SHADER CODE

- Why?
  - Focus on tracing rays, identifying where to trace rays
  - Where interesting rendering algorithms mostly live
- The CPU has vital infrastructure…
  - But it's largely reusable stuff like asset loaders
  - Not interesting (to me) to re-write

WE'LL FOCUS ON GPU SHADER CODE

- Why?
  - Focus on tracing rays, identifying where to trace rays
  - Where interesting rendering algorithms mostly live
- The CPU has vital infrastructure…
  - But it's largely reusable stuff like asset loaders
  - Not interesting (to me) to re-write
- For parallel GPU ray tracer, CPU code is mostly glue:
  - Pass configuration and data to GPU
  - Launch GPU processes