INFOMAGR – Advanced Graphics

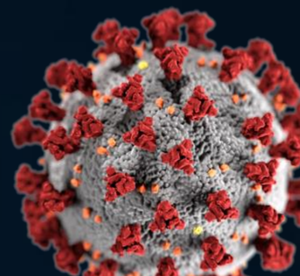
Jacco Bikker - November 2021 - February 2022

Lecture 10 - "GPU Ray Tracing (2)"

Welcome!

Today's Agenda:

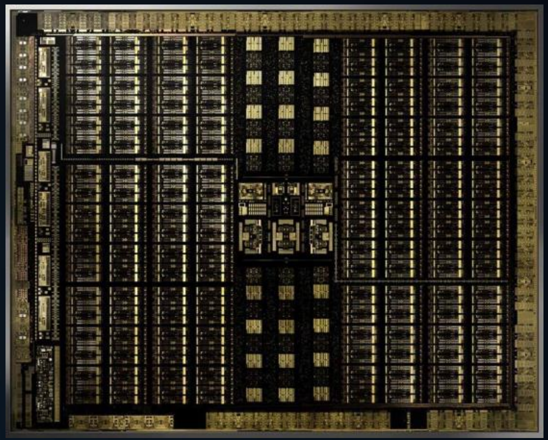
- State of the Art
- Wavefront Path Tracing

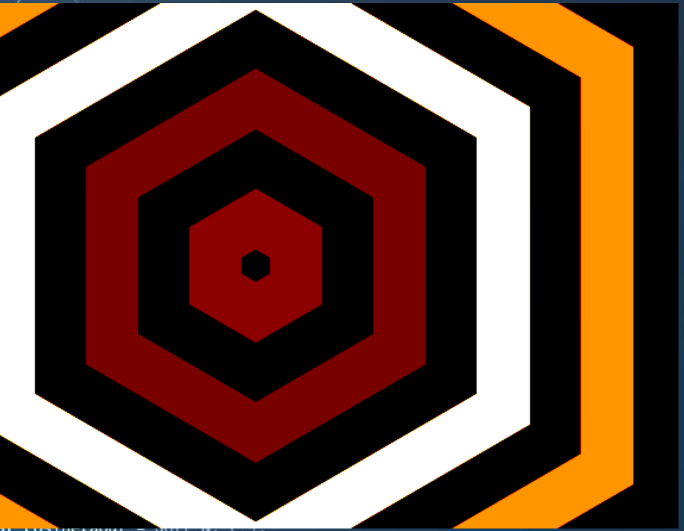


STAR

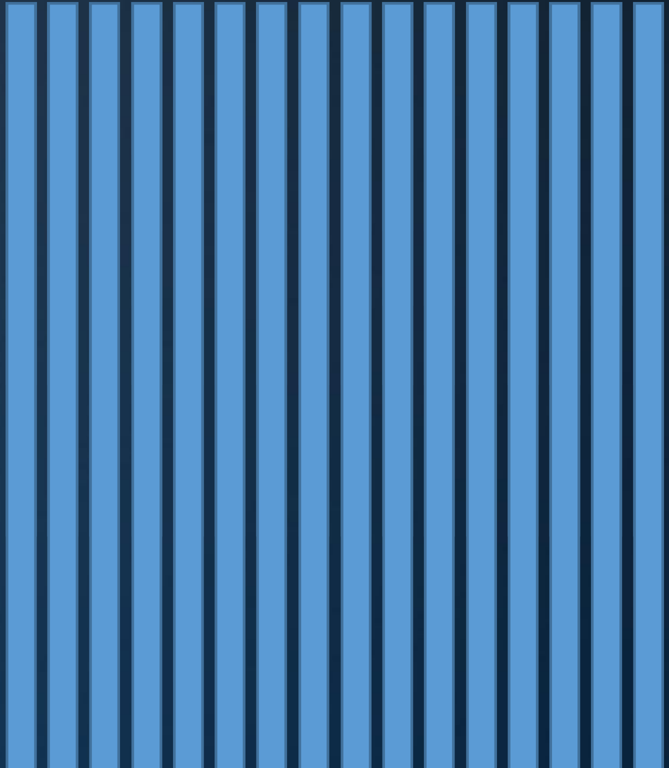
Previously in Advanced Graphics

GPU Architecture





https://www.shadertoy.com/view/wdcBW2



```
void mainImage( out vec4 fragColor, in vec2 fragCoord )
{
    vec2 uv = (fragCoord-.5*iResolution.xy)/iResolution.y;
    uv.y += .355;
    vec2 mouse = iMouse.xy/iResolution.xy;
    uv *= .29;
    vec3 col = vec3(0);
    uv.x = abs(uv.x);
    uv.y += tan(((5./6.)*3.1415))*.68;
    vec2 n = N((5./6.)*3.1415);
    float d = dot(uv-vec2(.5, 0), n);
    uv -= n*max(0., d)*2.;
    n = N((2./3.)*3.1415);
    float scale = 1.;
    uv.x += .5;
    for(int i=0; i < 1; i++) {
        uv *= 3.;
        scale *= 3.;
        uv.x -= 1.5;
        uv.x = abs(uv.x);
        uv.x -= 2.1;
        uv -= n*min(0., dot(uv, n))*1.;
```

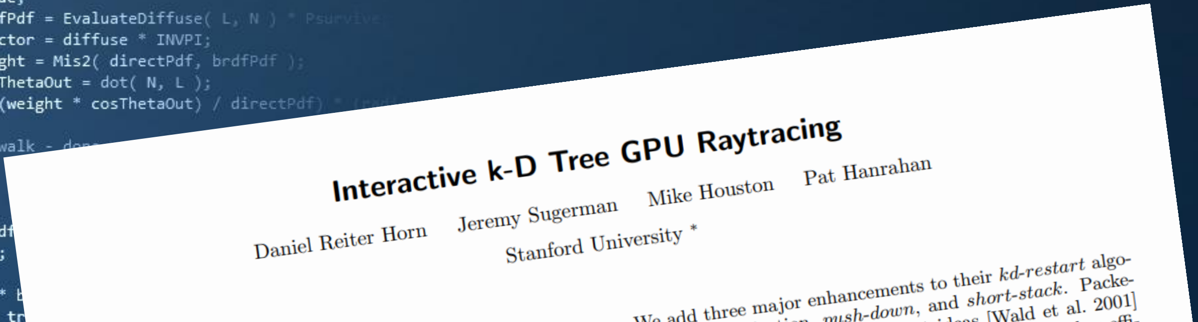
STAR

Previously in Advanced Graphics

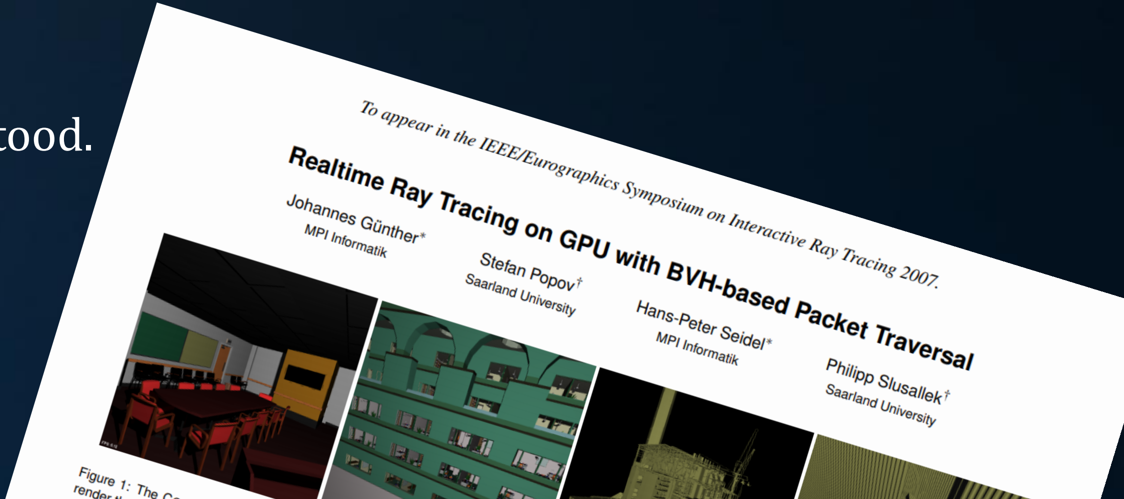
A Brief History of GPU Ray Tracing

2002: Purcell et al., multi-pass shaders with stencil, grid, low efficiency
2005: Foley & Sugerman, kD-tree, stack-less traversal with kdrestart
2007: Horn et al., kD-tree with short stack, single pass with flow control
2007: Popov et al., kD-tree with ropes
2007: Günther et al., BVH with packets.

- The use of BVHs allowed for complex scenes on the GPU (millions of triangles);
- CPU is now outperformed by the GPU;
- GPU compute potential is not realized;
- Aspects that affect efficiency are poorly understood.


Interactive k-D Tree GPU Raytracing

Daniel Reiter Horn Jeremy Sugerman Mike Houston Pat Hanrahan
Stanford University *


To appear in the IEEE/Eurographics Symposium on Interactive Ray Tracing 2007.

Realtime Ray Tracing on GPU with BVH-based Packet Traversal

Johannes Günther* MPI Informatik
Stefan Popov† Saarland University
Hans-Peter Seidel* MPI Informatik
Philipp Slusallek† Saarland University

STAR

Understanding the Efficiency of Ray Traversal on GPUs*

Observations on BVH traversal:

Ray/scene intersection consists of an unpredictable sequence of node traversal and primitive intersection operations. This is a major cause of inefficiency on the GPU.

Random access of the scene leads to high bandwidth requirement of ray tracing.

BVH packet traversal as proposed by Gunther et al. should alleviate bandwidth strain and yield near-optimal performance.

Packet traversal doesn't yield near-optimal performance. Why not?

*: Understanding the Efficiency of Ray Tracing on GPUs, Aila & Laine, 2009.
and: Understanding the Efficiency of Ray Tracing on GPUs – Kepler & Fermi addendum, 2012.

STAR

Understanding the Efficiency of Ray Traversal on GPUs

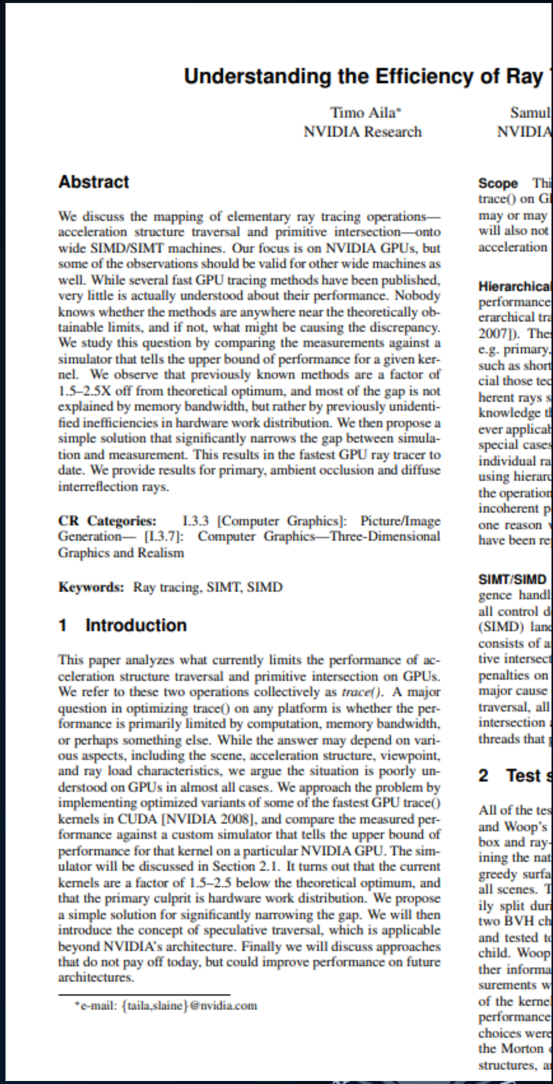
Simulator:

1. Dump sequence of traversal, leaf and triangle intersection operations required for each ray.
2. Use generated GPU assembly code to obtain a sequence of instructions that need to be executed for each ray.
3. Execute this sequence assuming ideal circumstances:

- Execute two instructions in parallel;
- Make memory access ‘free’.

The simulator reports on estimated execution speed and SIMD efficiency.

➔ The same program running on an actual GPU can never do better;
➔ The simulator provides an upper bound on performance.



Understanding the Efficiency of Ray

Timo Aila*
NVIDIA Research

Abstract

We discuss the mapping of elementary ray tracing operations—acceleration structure traversal and primitive intersection—onto wide SIMD/SIMT machines. Our focus is on NVIDIA GPUs, but some of the observations should be valid for other wide machines as well. While several fast GPU tracing methods have been published, very little is actually understood about their performance. Nobody knows whether the methods are anywhere near the theoretically obtainable limits, and if not, what might be causing the discrepancy. We study this question by comparing the measurements against a simulator that tells the upper bound of performance for a given kernel. We observe that previously known methods are a factor of 1.5–2.5X off from theoretical optimum, and most of the gap is not explained by memory bandwidth, but rather by previously unidentified inefficiencies in hardware work distribution. We then propose a simple solution that significantly narrows the gap between simulation and measurement. This results in the fastest GPU ray tracer to date. We provide results for primary, ambient occlusion and diffuse interreflection rays.

CR Categories: I.3.3 [Computer Graphics]: Picture/Image Generation— [I.3.7]: Computer Graphics—Three-Dimensional Graphics and Realism

Keywords: Ray tracing, SIMT, SIMD

1 Introduction

This paper analyzes what currently limits the performance of acceleration structure traversal and primitive intersection on GPUs. We refer to these two operations collectively as *trace()*. A major question in optimizing trace() on any platform is whether the performance is primarily limited by computation, memory bandwidth, or perhaps something else. While the answer may depend on various aspects, including the scene, acceleration structure, viewpoint, and ray load characteristics, we argue the situation is poorly understood on GPUs in almost all cases. We approach the problem by implementing optimized variants of some of the fastest GPU trace() kernels in CUDA [NVIDIA 2008], and compare the measured performance against a custom simulator that tells the upper bound of performance for that kernel on a particular NVIDIA GPU. The simulator will be discussed in Section 2.1. It turns out that the current kernels are a factor of 1.5–2.5 below the theoretical optimum, and that the primary culprit is hardware work distribution. We propose a simple solution for significantly narrowing the gap. We will then introduce the concept of speculative traversal, which is applicable beyond NVIDIA's architecture. Finally we will discuss approaches that do not pay off today, but could improve performance on future architectures.

*e-mail: {taila,slaine}@nvidia.com

STAR

Understanding the Efficiency of Ray Traversal on GPUs

Test setup

Scene: "Conference", 282K tris, 164K nodes

Ray distributions:

1. Primary: coherent rays
2. AO: short divergent rays
3. Diffuse: long divergent rays

Hardware: NVidia GTX285.

STAR

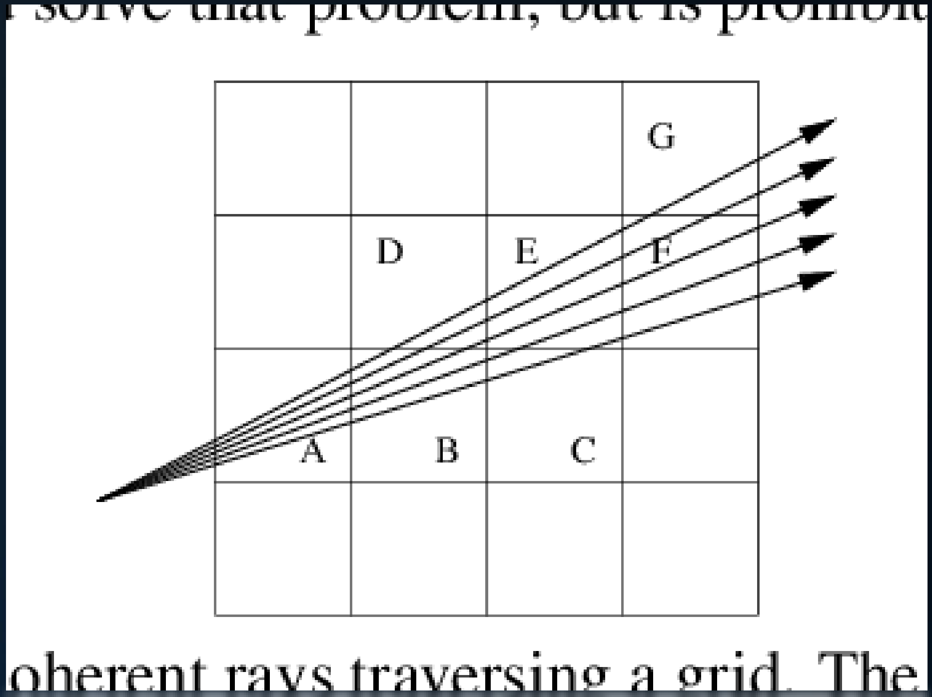
Understanding the Efficiency of Ray Traversal on GPUs

Test setup

Scene: “Conference”, 282K tris, 164K nodes

Ray distributions:

1. Primary: coherent rays
2. AO: short divergent rays
3. Diffuse: long divergent rays



Hardware: NVidia GTX285.

STAR

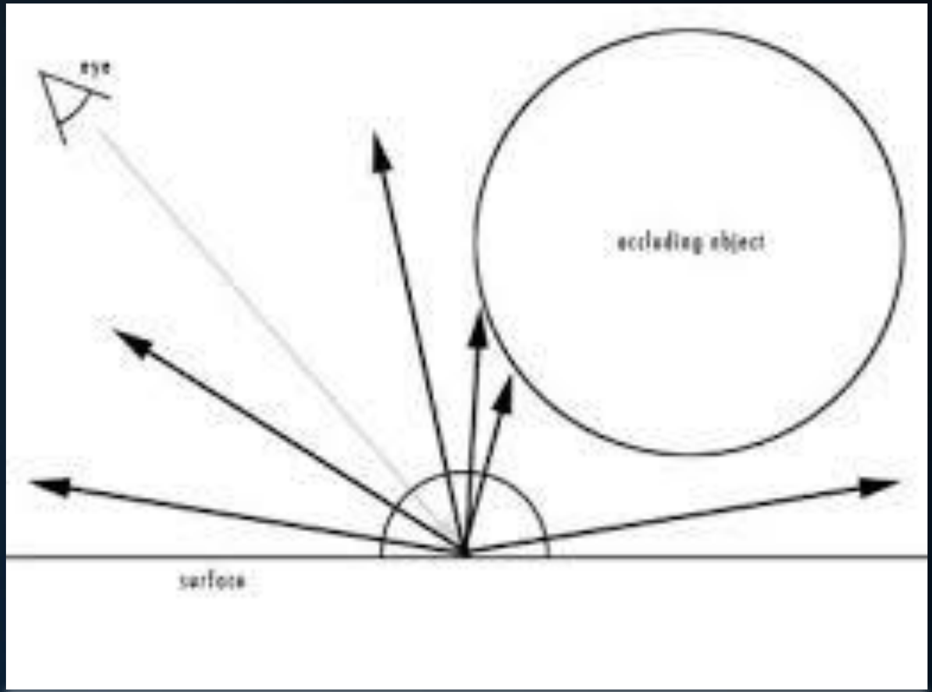
Understanding the Efficiency of Ray Traversal on GPUs

Test setup

Scene: “Conference”, 282K tris, 164K nodes

Ray distributions:

1. Primary: coherent rays
2. AO: short divergent rays
3. Diffuse: long divergent rays



Hardware: NVidia GTX285.

STAR

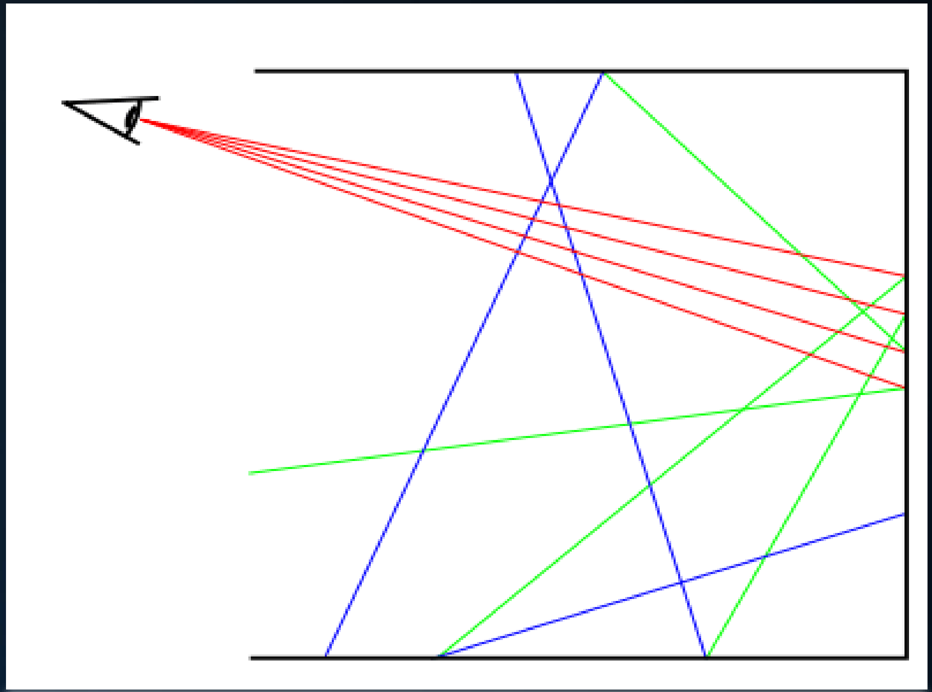
Understanding the Efficiency of Ray Traversal on GPUs

Test setup

Scene: “Conference”, 282K tris, 164K nodes

Ray distributions:

1. Primary: coherent rays
2. AO: short divergent rays
3. Diffuse: long divergent rays



Hardware: NVidia GTX285.

STAR

Understanding the Efficiency of Ray Traversal on GPUs

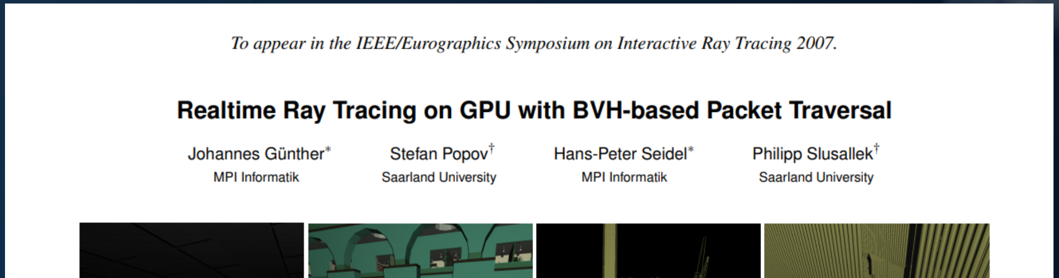
Simulator results, in MRays/s:

Packet traversal as proposed by Gunther et al. is a factor 1.7-2.4 off from simulated performance:

	Simulated	Actual	%
Primary	149.2	63.6	43
AO	100.7	39.4	39
Diffuse	36.7	16.6	45

(this does not take into account algorithmic inefficiencies)

Hardware: NVidia GTX285.



To appear in the IEEE/Eurographics Symposium on Interactive Ray Tracing 2007.

Realtime Ray Tracing on GPU with BVH-based Packet Traversal

Johannes Günther* MPI Informatik
Stefan Popov† Saarland University
Hans-Peter Seidel* MPI Informatik
Philipp Slusallek† Saarland University

STAR

Simulating Alternative Traversal Loops

Variant 1: *'while-while'*

```
while ray not terminated
   while node is interior node
      traverse to the next node
   while node contains untested primitives
      perform ray/prim intersection
```

Results:

		Simulated		Actual		%
Primary	149.2	166.7	63.6	88.0	43	53
AO	100.7	160.7	39.4	86.3	39	54
Diffuse	36.7	81.4	16.6	44.5	45	55

numbers in green: Packet traversal, Gunther-style (from previous slide).

Hardware: NVidia GTX285.

Here, every ray has its own stack; This is simply a GPU implementation of typical CPU BVH traversal.

Compared to packet traversal, memory access is <u>less</u> coherent.

One would expect a larger gap between simulated and actual performance. However, this is not the case (not even for divergent rays).

Conclusion: *bandwidth is not the problem.*

STAR

Simulating Alternative Traversal Loops

Variant 2: *'if-if'*

```
while ray not terminated
   if node is interior node
      traverse to the next node
   if node contains untested primitives
      perform a ray/prim intersection
```

Results:

		Simulated		Actual		%
Primary	166.7	129.3	88.0	90.1	53	70
AO	160.7	131.6	86.3	88.8	54	67
Diffuse	81.4	70.5	44.5	45.3	55	64

numbers in green: while-while.

Hardware: NVidia GTX285.

This time, each loop iteration either executes a traversal step or a primitive intersection.

Memory access is even less coherent in this case.

Nevertheless, it is *faster* than while-while. Why?

While-while leads to a small number of long-running warps. Some threads stall while others are still traversing, after which they stall again while others are still intersecting.

STAR

Simulating Alternative Traversal Loops

Variant 3: *'persistent while-while'*

Idea: rather than spawning a thread per ray, we spawn the ideal number of threads for the hardware.

Each thread increases an atomic counter to fetch a ray from a pool, until the pool is depleted*.

Benefit: we bypass the hardware thread scheduler.

Results:

		Simulated		Actual		%
Primary	129.3	166.7	90.1	135.6	70	81
AO	131.6	160.7	88.8	130.7	67	81
Diffuse	70.5	81.4	45.3	62.4	64	77

numbers in green: if-if.

Hardware: NVidia GTX285.

This test shows what the limiting factor was: thread scheduling. By handling this explicitly, we get much closer to theoretical optimal performance.

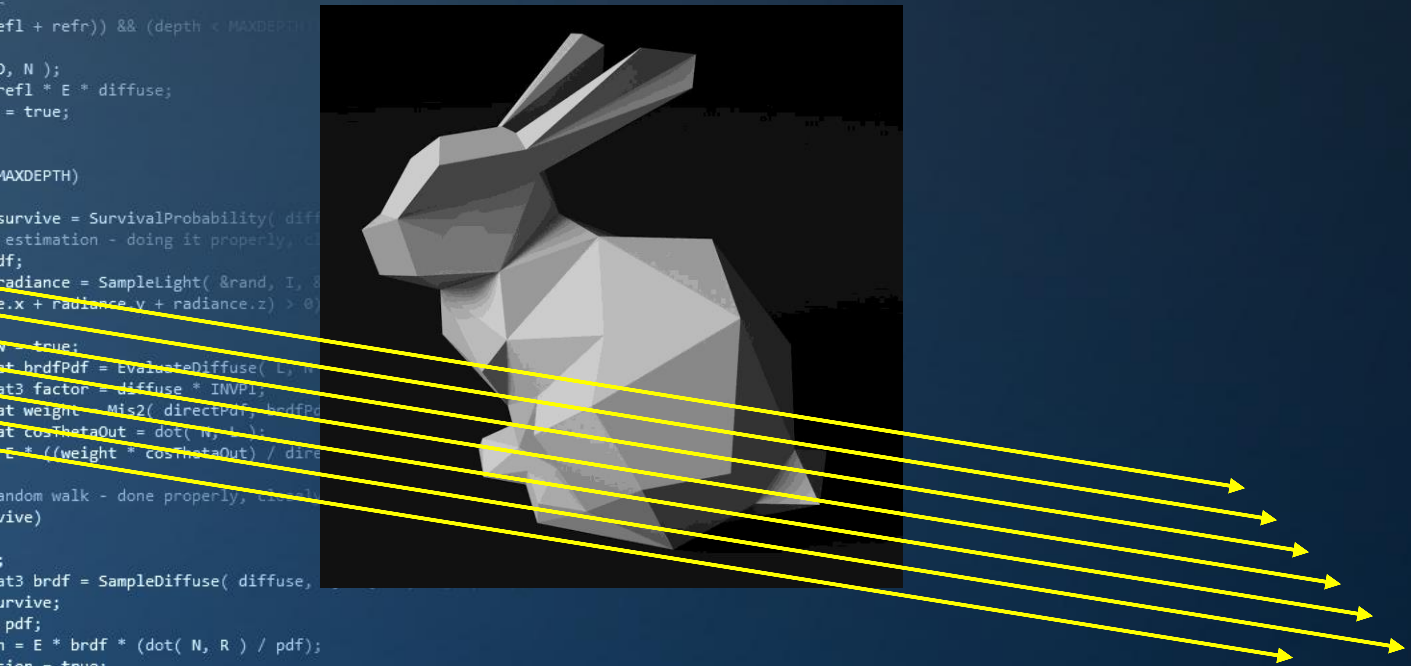
*: In practice, this is done per warp: the first thread in the warp increases the counter by 32. This reduces the number of atomic operations.

STAR

Simulating Alternative Traversal Loops

Variant 4: *'speculative traversal'*

Idea: while some threads traverse, threads that want to intersect prior to (potentially) continuing traversal may just as well traverse anyway – *the alternative is idling*.



STAR

Simulating Alternative Traversal Loops

Variant 4: *'speculative traversal'*

Idea: while some threads traverse, threads that want to intersect prior to (potentially) continuing traversal may just as well traverse anyway – *the alternative is idling*.

Drawback: these threads now fetch nodes that they may not need to fetch*. However, we noticed before that bandwidth is not the issue.

Results for persistent speculative while-while:

		Simulated		Actual		%
Primary	166.7	165.7	135.6	142.2	81	86
AO	160.7	169.1	130.7	134.5	81	80
Diffuse	81.4	92.9	62.4	60.9	77	66

numbers in green: persistent while-while.

Hardware: NVidia GTX285.

For diffuse rays, performance starts to differ significantly from simulated performance. This suggests that we now start to suffer from limited memory bandwidth.

*: On a SIMT machine, we do not get redundant calculations using this scheme. We do however increase implementation complexity, which may affect performance.

STAR

Understanding the Efficiency of Ray Traversal on GPUs

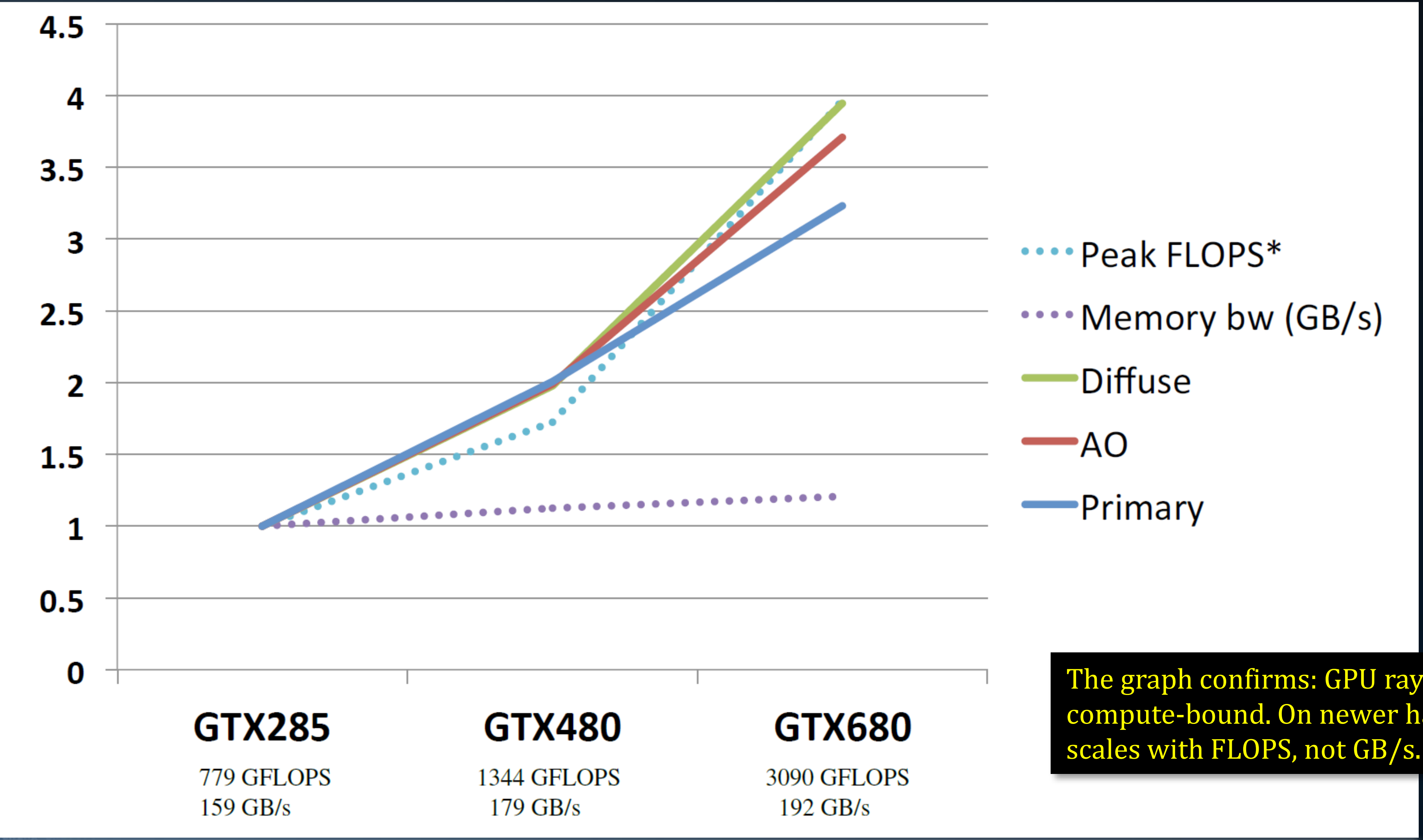
- Three years later* -

In 2009, NVidia‘s Tesla architecture was used (GTX285).
Results on Tesla (GTX285), Fermi (GTX480) and Kepler (GTX680):

	Tesla	Fermi	Kepler
Primary	142.2	272.1	432.6
AO	134.5	284.1	518.2
Diffuse	60.9	126.1	245.4

*: Aila et al., 2012. Understanding the efficiency of ray traversal on GPUs - Kepler and Fermi Addendum.

STAR



The graph confirms: GPU ray tracing is compute-bound. On newer hardware, it scales with FLOPS, not GB/s.

STAR

Latency Considerations of Depth-first GPU Ray Tracing*

A study of GPU ray tracing performance in the spirit of Aila & Laine has been published in 2014 by Guthe. Three optimizations are proposed:

1. Using a shallower hierarchy;
2. Loop unrolling for the while loops;
3. Loading data at once rather than scattered over the code.

	Titan (AL'09)	Titan (Guthe)	+%
Primary	605.7	688.6	13.7
AO	527.2	613.3	16.3
Diffuse	216.4	254.4	17.6

*: Latency Considerations of Depth-first GPU Ray Tracing, Guthe, 2014

STAR

Shallow Bounding Volume Hierarchies*

Idea:

We can cut the number of traversal steps in half if our BVH nodes have 4 instead of 2 child nodes.

Additional benefits:

- A proper layout allows for SIMD intersection of all four child AABBs;
- We increase the arithmetic density of a single traversal step.

*: Shallow Bounding Volume Hierarchies for Fast SIMD Ray Tracing of Incoherent Rays, Dammertz et al., 2008
Getting Rid of Packets - Efficient SIMD Single-Ray Traversal using Multi-branching BVHs, Wald et al., 2008

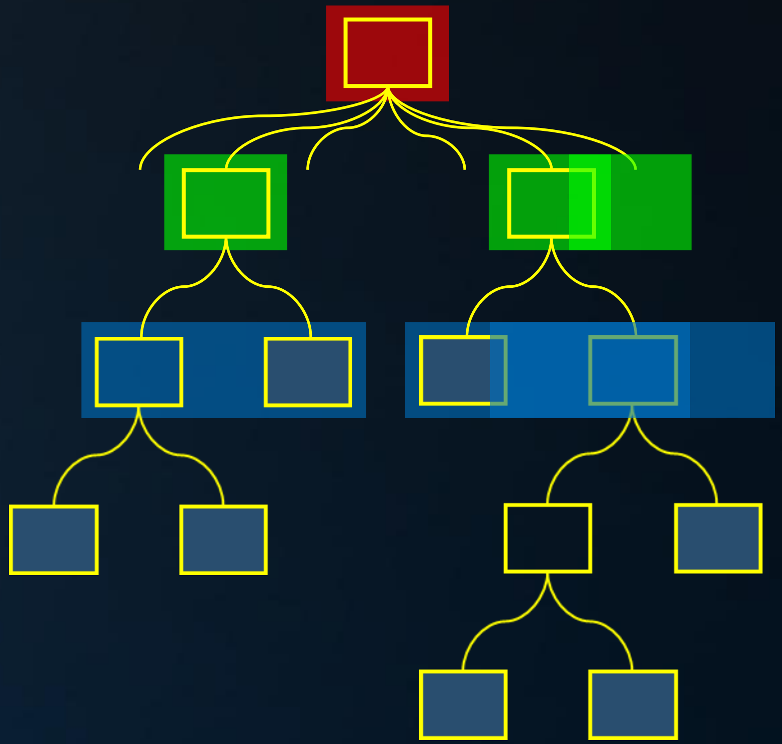
STAR

Building the MBVH

Collapsing a regular BVH

For each node n: iterate over the children c_i:

1. See if we can 'adopt' the children of c_i:
   $N_n - 1 + N_{c_i} \leq 4$;
2. Select the child with the greatest area;
3. Replace node c_i with its children;
4. Repeat until no merge is possible.



Repeat this process for the children of n.

Note that for this tree, the end result has one interior node with only 2 children, and one with only 3 children.

STAR

Building the MBVH

Data structure:

```
struct SIMD_BVH_Node
{
    __m128 bminx4, bmaxx4;
    __m128 bminy4, bmaxy4;
    __m128 bminz4, bmaxz4;
    int child[4], count[4];
};
```

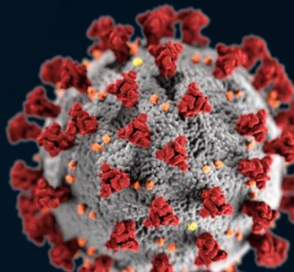
To traverse a regular BVH front-to-back, we can use a single comparison to find the nearest child. For an MBVH, this is not as trivial.

Pragmatic solution:

1. Obtain the four intersection distances in t4;
2. Overwrite the lowest bits of each float in t4 with binary 00, 01, 10 and 11;
3. Use a small sorting network to sort t4;
4. Extract the lowest bits to obtain the correct order in which the nodes should be processed.

Today's Agenda:

- State of the Art
- Wavefront Path Tracing



Wavefront

Mapping Path Tracing to the GPU

The path tracing loop from lecture 8 is straight-forward to implement on the GPU.

However:

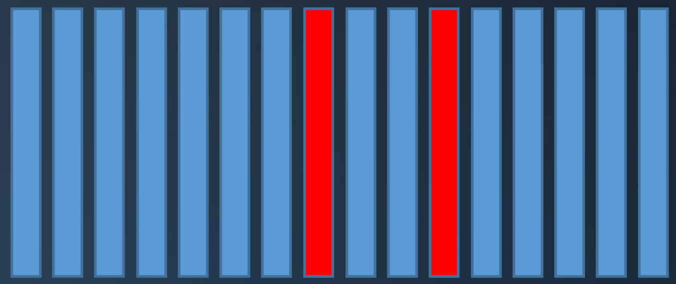
- Terminated paths become idling threads;
- A significant number of paths will not trace a shadow ray.

```
Color Sample( Ray ray )
{
   T = ( 1, 1, 1 ), E = ( 0, 0, 0 );
   while (1)
   {
      I, N, material = Trace( ray );
      BRDF = material.albedo / PI;
      if (ray.NOHIT) break;
      if (material.isLight) break;
      // sample a random light source
      L, Nl, dist, A = RandomPointOnLight();
      Ray lr( I, L, dist );
      if (N·L > 0 && Nl·-L > 0) if (!Trace( lr ))
      {
         solidAngle = ((Nl·-L) * A) / dist²;
         lightPDF = 1 / solidAngle;
         E += T * (N·L / lightPDF) * BRDF * lightColor;
      }
      // continue random walk
      R = DiffuseReflection( N );
      hemiPDF = 1 / (PI * 2.0f);
      ray = Ray( I, R );
      T *= ((N·R) / hemiPDF) * BRDF;
   }
   return E;
}
```

Wavefront

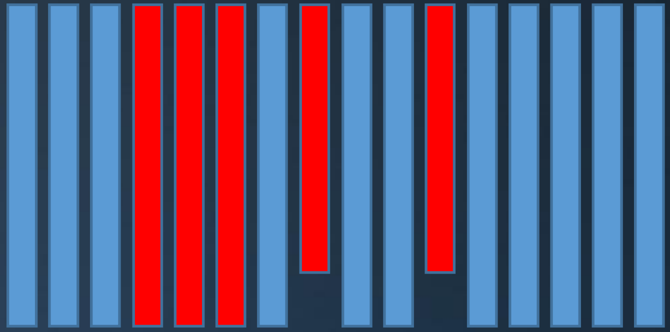
```
Color Sample( Ray ray )
{
   T = ( 1, 1, 1 ), E = ( 0, 0, 0 );
   while (1)
   {
      I, N, material = Trace( ray );
      BRDF = material.albedo / PI;
      if (ray.NOHIT) break;
      if (material.isLight) break;
      // sample a random light source
      L, Nl, dist, A = RandomPointOnLight();
      Ray lr( I, L, dist );
      if (N·L > 0 && Nl·-L > 0) if (!Trace( lr ))
      {
         solidAngle = ((Nl·-L) * A) / dist²;
         lightPDF = 1 / solidAngle;
         E += T * (N·L / lightPDF) * BRDF * lightColor;
      }
      // continue random walk
      R = DiffuseReflection( N );
      hemiPDF = 1 / (PI * 2.0f);
      ray = Ray( I, R );
      T *= ((N·R) / hemiPDF) * BRDF;
   }
   return E;
}
```

Wavefront



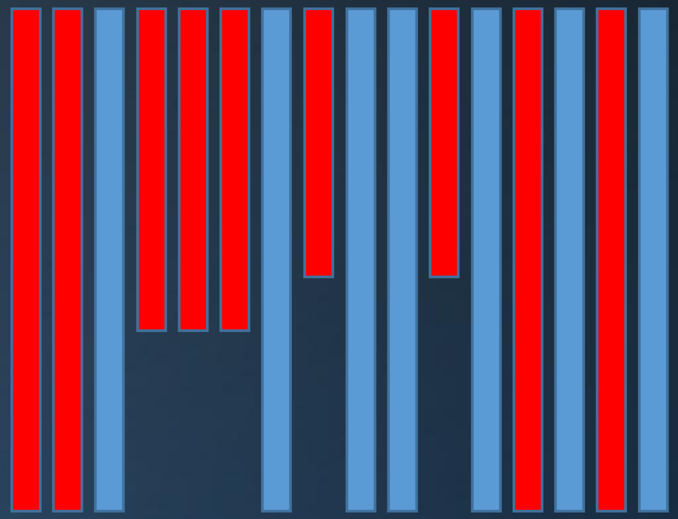
```
Color Sample( Ray ray )
{
   T = ( 1, 1, 1 ), E = ( 0, 0, 0 );
   while (1)
   {
      I, N, material = Trace( ray );
      BRDF = material.albedo / PI;
      if (ray.NOHIT) break;
      if (material.isLight) break;
      // sample a random light source
      L, Nl, dist, A = RandomPointOnLight();
      Ray lr( I, L, dist );
      if (N·L > 0 && Nl·-L > 0) if (!Trace( lr ))
      {
         solidAngle = ((Nl·-L) * A) / dist²;
         lightPDF = 1 / solidAngle;
         E += T * (N·L / lightPDF) * BRDF * lightColor;
      }
      // continue random walk
      R = DiffuseReflection( N );
      hemiPDF = 1 / (PI * 2.0f);
      ray = Ray( I, R );
      T *= ((N·R) / hemiPDF) * BRDF;
   }
   return E;
}
```

Wavefront



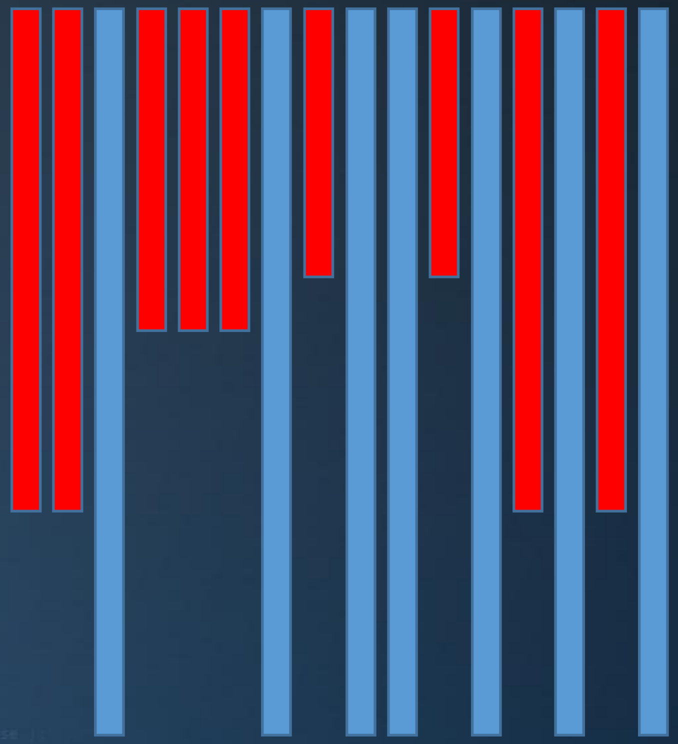
```
Color Sample( Ray ray )
{
   T = ( 1, 1, 1 ), E = ( 0, 0, 0 );
   while (1)
   {
      I, N, material = Trace( ray );
      BRDF = material.albedo / PI;
      if (ray.NOHIT) break;
      if (material.isLight) break;
      // sample a random light source
      L, Nl, dist, A = RandomPointOnLight();
      Ray lr( I, L, dist );
      if (N·L > 0 && Nl·-L > 0) if (!Trace( lr ))
      {
         solidAngle = ((Nl·-L) * A) / dist²;
         lightPDF = 1 / solidAngle;
         E += T * (N·L / lightPDF) * BRDF * lightColor;
      }
      // continue random walk
      R = DiffuseReflection( N );
      hemiPDF = 1 / (PI * 2.0f);
      ray = Ray( I, R );
      T *= ((N·R) / hemiPDF) * BRDF;
   }
   return E;
}
```

Wavefront



```
Color Sample( Ray ray )
{
   T = ( 1, 1, 1 ), E = ( 0, 0, 0 );
   while (1)
   {
      I, N, material = Trace( ray );
      BRDF = material.albedo / PI;
      if (ray.NOHIT) break;
      if (material.isLight) break;
      // sample a random light source
      L, Nl, dist, A = RandomPointOnLight();
      Ray lr( I, L, dist );
      if (N·L > 0 && Nl·-L > 0) if (!Trace( lr ))
      {
         solidAngle = ((Nl·-L) * A) / dist²;
         lightPDF = 1 / solidAngle;
         E += T * (N·L / lightPDF) * BRDF * lightColor;
      }
      // continue random walk
      R = DiffuseReflection( N );
      hemiPDF = 1 / (PI * 2.0f);
      ray = Ray( I, R );
      T *= ((N·R) / hemiPDF) * BRDF;
   }
   return E;
}
```

Wavefront



```
Color Sample( Ray ray )
{
   T = ( 1, 1, 1 ), E = ( 0, 0, 0 );
   while (1)
   {
      I, N, material = Trace( ray );
      BRDF = material.albedo / PI;
      if (ray.NOHIT) break;
      if (material.isLight) break;
      // sample a random light source
      L, Nl, dist, A = RandomPointOnLight();
      Ray lr( I, L, dist );
      if (N·L > 0 && Nl·-L > 0) if (!Trace( lr ))
      {
         solidAngle = ((Nl·-L) * A) / dist²;
         lightPDF = 1 / solidAngle;
         E += T * (N·L / lightPDF) * BRDF * lightColor;
      }
      // continue random walk
      R = DiffuseReflection( N );
      hemiPDF = 1 / (PI * 2.0f);
      ray = Ray( I, R );
      T *= ((N·R) / hemiPDF) * BRDF;
   }
   return E;
}
```

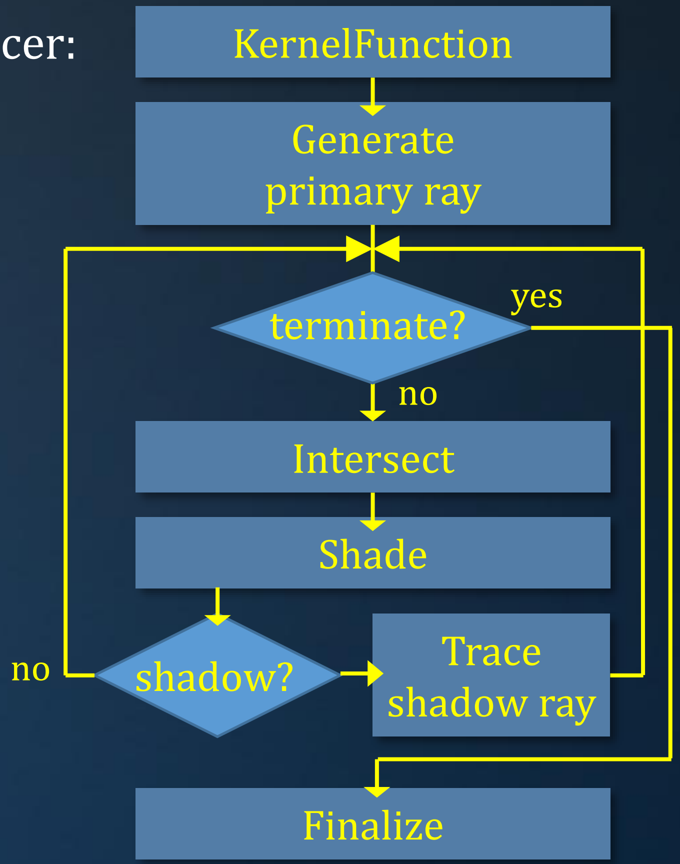
Wavefront

```
Color Sample( Ray ray )
{
   T = ( 1, 1, 1 ), E = ( 0, 0, 0 );
   while (1)
   {
      I, N, material = Trace( ray );
      BRDF = material.albedo / PI;
      if (ray.NOHIT) break;
      if (material.isLight) break;
      // sample a random light source
      L, Nl, dist, A = RandomPointOnLight();
      Ray lr( I, L, dist );
      if (N·L > 0 && Nl·-L > 0) if (!Trace( lr ))
      {
         solidAngle = ((Nl·-L) * A) / dist^2;
         lightPDF = 1 / solidAngle;
         E += T * (N·L / lightPDF) * BRDF * lightColor;
      }
      // continue random walk
      R = DiffuseReflection( N );
      hemiPDF = 1 / (PI * 2.0f);
      ray = Ray( I, R );
      T *= ((N·R) / hemiPDF) * BRDF;
   }
   return E;
}
```

Wavefront

Megakernels Considered Harmful*

Naïve path tracer:



Translating this to CUDA or OpenCL code yields a single kernel: individual functions are still compiled to one monolithic chunk of code.

Resource requirements (registers) - and thus parallel slack - are determined by ‘weakest link’, i.e. the functional block that requires most registers.

Conditional code leads to idling threads that wait until others are done.

*: Megakernels Considered Harmful: Wavefront Path Tracing on GPUs, Laine et al., 2013

Wavefront

Megakernels Considered Harmful

Solution: *split the kernel*.

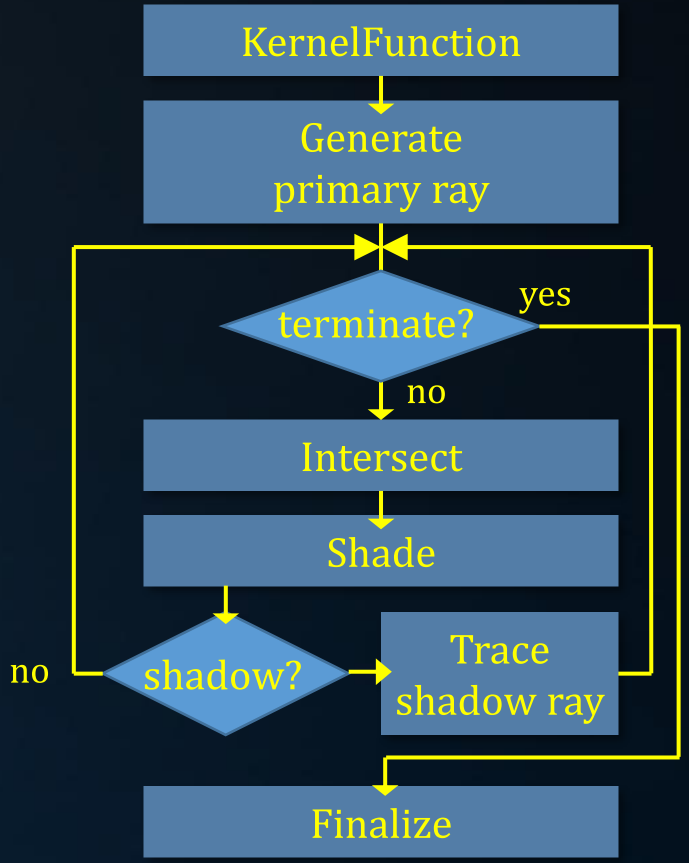
Example:

Kernel 1: Generate primary rays.
Kernel 2: Trace paths.
Kernel 3: Accumulate, gamma correct, convert to ARGB32.

Consequence:

Kernel 1 generates *all* primary rays, and stores the result.
Kernel 2 takes this buffer and operates on it.

➔ *Massive memory I/O.*



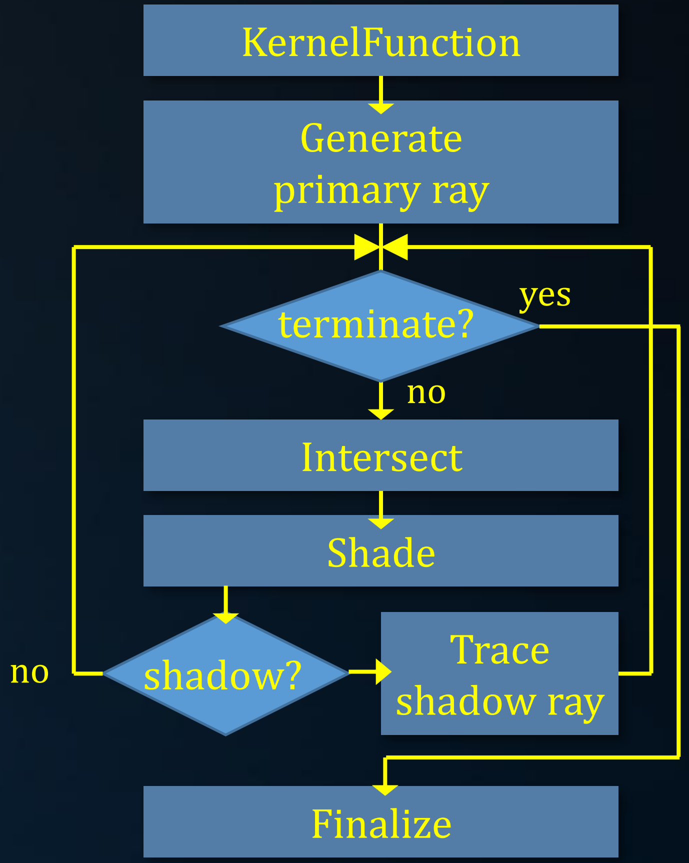
Wavefront

Megakernels Considered Harmful

Taking this further: streaming path tracing*.

Kernel 1: generate primary rays.
Kernel 2: extend.
Kernel 3: shade.
Kernel 4: connect.
Kernel 5: finalize.

Here, kernel 2 traces a set of rays to find the next path vertex (the random walk).
Kernel 3 processes the results and generates new path segments and shadow rays (2 separate buffers).
Kernel 4 traces the shadow ray buffer.
Kernel 1, 2, 3 and 4 are executed in a loop until no rays remain.



*: Improving SIMD Efficiency for Parallel Monte Carlo Light Transport on the GPU, van Antwerpen, 2011

Wavefront

Megakernels Considered Harmful

Zooming in:

The `generate` kernel produces *N* primary rays:

0, 1, …	…, N-1

Buffer 1: path segments (*N* times O,D,t,primIdx)

The `extend` kernel traces extension rays and produces intersections*.
The `shade` kernel processes intersections, and produces new extension paths as well as shadow rays:

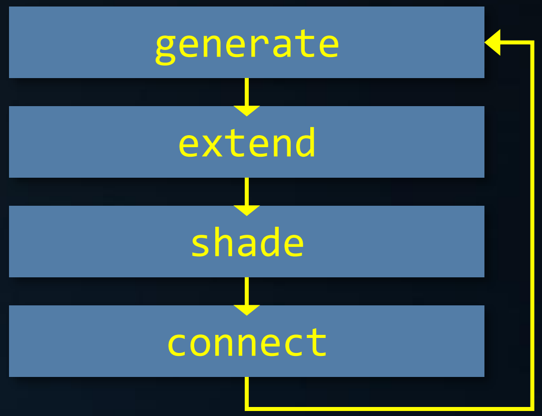
0, 1, …	…, N-1

Buffer 2: generated path segments (*N* times O,D,t,primIdx)

0, 1, …	…, N-1

Buffer 3: generated shadow rays (*N* times O,D,t, E,pixelIdx)

Finally, the `connect` kernel traces shadow rays.



Note: here, the loop is implemented on the host. Each block is a separate kernel invocation.

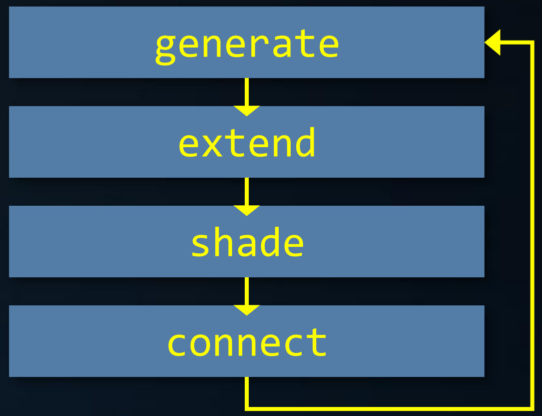
*: An intersection is at least the t value, plus a primitive identifier.

Wavefront

Megakernels Considered Harmful

Generate:

```
for each screen pixel i
{
   O,D = GenerateRayDirection(i)
   rayBuffer[i] = Ray( O, D, infinity, -1 )
}
```

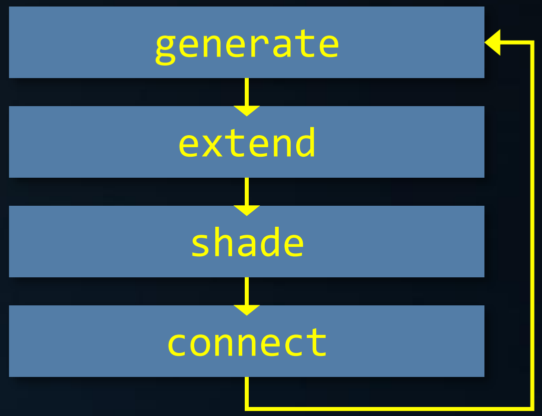


Wavefront

Megakernels Considered Harmful

Extend:

```
for each buffered ray r
{
   O,D,dist = rayBuffer[i]
   dist, primIdx = FindNearestIntersection( O, D, dist )
   rayBuffer[i].dist = dist
   rayBuffer[i].primIdx = primIdx
}
```

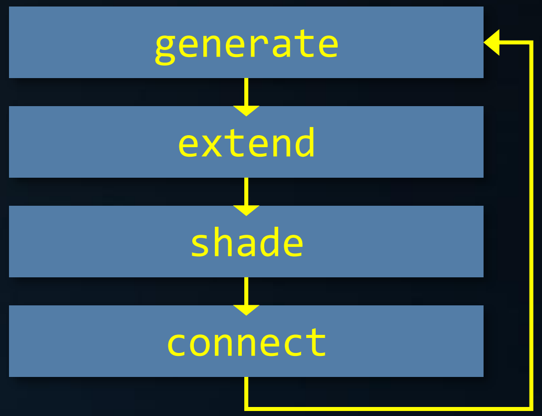


Wavefront

Megakernels Considered Harmful

Shade:

```
for each buffered ray r
{
    O,D,dist,primIdx = rayBuffer[i]
    I = IntersectionPoint( O, D, dist )
    N = PrimNormal( primIdx, I )
    if (NEE) {
        si = atomicInc( shadowRayIdx )
        shadowBuffer[si] = ShadowRay( … )
    }
    if (bounce) {
        ei = atomicInc( extensionRayIdx )
        newRayBuffer[ei] = ExtensionRay( … )
    }
}
```

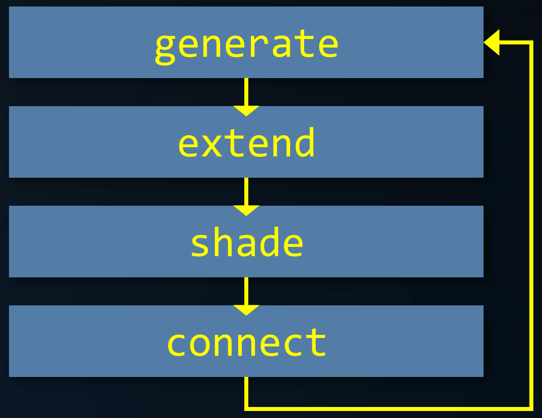


Wavefront

Megakernels Considered Harmful

Connect:

```
for each buffered shadowRay r
{
   O,D,dist,E, pixelIdx = shadowBuffer[i]
   if (!Occluded( O, D, dist ))
   {
      accumulator[pixelIdx] += E;
   }
}
```



Wavefront

Megakernels Considered Harmful

Digest:

Streaming path tracing introduces seemingly costly operations:

- Repeated I/O to/from large buffers;
- A significant number of kernel invocations per frame;
- Communication with the host.

The Wavefront paper claims that this is beneficial for complex shaders. In practice, this also works for (very) simple shaders.

Also note that the megakernel paper (2013) presents an idea already presented by Dietger van Antwerpen (2011).

Today's Agenda:

- State of the Art
- Wavefront Path Tracing

INFOMAGR – Advanced Graphics

Jacco Bikker - November 2021 - February 2022

END of “GPU Ray Tracing (2)”

next lecture: “Variance Reduction (2)”