rics & (depth < ∧∞coo

: = inside ? 1 + 1.0 ht = nt / nc, ddn bs2t = 1.0f - nnt D, N); 3)

at a = nt - nc, b = 00 at Tr = 1 - (R0 + (1 - R0 rr) R = (D = nnt - N = 000

= * diffuse = true;

efl + refr)) && (depth < MAXDEPT

D, N); refl * E * diffuse; = true;

AXDEPTH)

survive = SurvivalProbability(diffuse estimation - doing it properly, closed) ff; and are - Sampleicht(Sample - Sample)

radiance = SampleLight(&rand, I, &L, &ll e.x + radiance.y + radiance.z) > 0) && (dl.)

v = true; at brdfPdf = EvaluateDiffuse(L, N) Psurvive at3 factor = diffuse * INVPI; at weight = Mis2(directPdf, brdfPdf); at cosThetaOut = dot(N, L); E * ((weight * cosThetaOut) / directPdf)

andom walk - done properly, closely following Small /ive)

; at3 brdf = SampleDiffuse(diffuse, N, r1, r2, &R, &pdf urvive; pdf; n = E * brdf * (dot(N, R) / pdf); sion = true:

/INFOMOV/ Optimization & Vectorization

J. Bikker - Sep-Nov 2019 - Lecture 10: "GPGPU (3)"

Welcome!



ics & (depth < Mode∈)

: = inside ? 1 + . . ht = nt / nc, ddn os2t = 1.0f - nnt O, N); 3)

at a = nt - nc, b = nt at Tr = 1 - (R0 + (1 - R0 Fr) R = (D ⁼ nnt - N - (00)

= * diffuse = true;

efl + refr)) && (depth < MODECT)

), N); refl * E * diffuse; = true;

AXDEPTH)

survive = SurvivalProbability(diffuse
estimation - doing it properly, closed
f;
radiance = SampleLight(&rand, I, &L, &light()
e.x + radiance.y + radiance.z) > 0) && (doing)

w = true; at brdfPdf = EvaluateDiffuse(L, N) Psurvive at3 factor = diffuse * INVPI; at weight = Mis2(directPdf, brdfPdf); at cosThetaOut = dot(N, L); E * ((weight * cosThetaOut) / directPdf) (radd)

andom walk - done properly, closely following Sec. /ive)

; t3 brdf = SampleDiffuse(diffuse, N, r1, r2, &R, &pdf urvive; pdf; n = E * brdf * (dot(N, R) / pdf); Sion = true:

Today's Agenda:

- GPU Execution Model
- GPGPU Flow
- GPGPU Low Level Notes
- P3



efl + refr)) && (dept

Recap

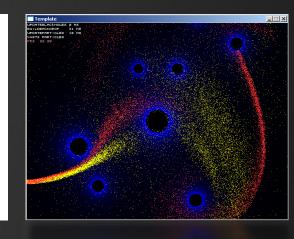
- The GPU is a co-processor, which needs a host.
- GPUs have a history of fixed-function pipelines.
- Typical GPU work is fundamentally data-parallel.
- GPU programming is similar to SIMD programming.
- For parallel tasks, a GPU is very fast (worth the effort!).

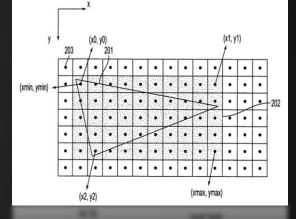


Theoretical Peak Performance, Single Precision

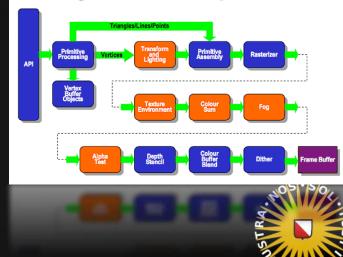
andom walk - done properly, closely following /ive)

; at3 brdf = SampleDiffuse(diffuse, N, r1, r2, &R, &pdf urvive; pdf; n = E * brdf * (dot(N, R) / pdf); sion = true:





Existing Fixed Function Pipeline



SIMT Recap

Model

at a = nt

), N); refl * E * diffuse; = true;

(AXDEPTH)

survive = SurvivalProbability(diff) if; radiance = SampleLight(&rand, I, &L,

e.x + radiance.y + radiance.z) > 0) 8

v = true;

at brdfPdf = EvaluateDiffuse(L, N) at3 factor = diffuse * INVPI; at weight = Mis2(directPdf, brdfPdf); at cosThetaOut = dot(N, L);

E * ((weight * cosThetaOut) / directPdf)

andom walk - done properly, closely follow /ive)

at3 brdf = SampleDiffuse(diffuse, N, r1, r2, &R, &p urvive; pdf; 1 = E * brdf * (dot(N, R) / pdf); sion = true:

for (float i = 0.0; i < 4095.0f; i += 1.0)

S.I.M.T.: *Single Instruction, Multiple Thread.*

dz = (float2)(2.0f * (z.x * dz.x - z.y * dz.y) + 1.0f, 2.0f * (z.x * dz.y + z.y * dz.x));z = cmul(z, z) + c;float a = sin(tm * 1.5f + i * 2.0f) * 0.3f + i * 1.3f; float2 t = (float2)($\cos(a) * z.x + \sin(a) * z.y$, $-\sin(a) * z.x + \cos(a) * z.y$);

if (fabs(t.x) > 2.0f && fabs(t.y) > 2.0f) { it = i; break; }

float $z^2 = z \cdot x + z \cdot y +$ float q = zoom * 0.016f * (1.0f / j.x + 1.0f / j.y), d = length(j), w = q * d / 400.0f; float s = q * d / 80.0f, f = 0.0f, g = 0.0f;







SIMT Recap

Model

at a = nt

), N); refl * E * diffuse; = true;

(AXDEPTH)

survive = SurvivalProbability(diffu if; radiance = SampleLight(&rand, I, &L,

e.x + radiance.y + radiance.z) > 0) {

v = true;

at brdfPdf = EvaluateDiffuse(L, N) at3 factor = diffuse * INVPI; at weight = Mis2(directPdf, brdfPdf); at cosThetaOut = dot(N, L);

E * ((weight * cosThetaOut) / directPdf)

andom walk - done properly, closely follow /ive)

at3 brdf = SampleDiffuse(diffuse, N, r1, r2, &R, & urvive; pdf; i = E * brdf * (dot(N, R) / pdf); sion = true:

for (float i = 0.0; i < 4095.0f; i += 1.0)

S.I.M.T.: *Single Instruction, Multiple Thread.*

dz = (float2)(2.0f * (z.x * dz.x - z.y * dz.y) + 1.0f, 2.0f * (z.x * dz.y + z.y * dz.x));z = cmul(z, z) + c;float a = sin(tm * 1.5f + i * 2.0f) * 0.3f + i * 1.3f;

float2 t = (float2)($\cos(a) * z.x + \sin(a) * z.y$, $-\sin(a) * z.x + \cos(a) * z.y$); if (fabs(t.x) > 2.0f && fabs(t.y) > 2.0f) { it = i; break; }

float z2 = z.x + z.y + z.y, t = log(z2) + sqrt(z2) / length(dz), r = sqrt(z2);float q = zoom * 0.016f * (1.0f / j.x + 1.0f / j.y), d = length(j), w = q * d / 400.0f; float s = q * d / 80.0f, f = 0.0f, g = 0.0f;





ics & (depth < NOCCS

: = inside } 1 = 1 ... ht = nt / nc, ddm = 0 bs2t = 1.0f - nmt = nm D, N); ≷)

at a = nt - nc, b = nt - n at Tr = 1 - (R0 + (1 - R0 Fr) R = (D ⁼ nnt - N = (dd

= * diffuse; = true;

efl + refr)) && (depth < NAXOED

D, N); refl * E * diffuse; = true;

AXDEPTH)

survive = SurvivalProbability(diffuse)
estimation - doing it properly, closed
if;
radiance = SampleLight(%rand, I, %L, %l)
e.x + radiance.y + radiance.z) > 0) %% ()

w = true; at brdfPdf = EvaluateDiffuse(L, N) * Psu at3 factor = diffuse * INVPI; at weight = Mis2(directPdf, brdfPdf); at cosThetaOut = dot(N, L); E * ((weight * cosThetaOut) / directPdf)

andom walk - done properly, closely following Source /ive)

```
t3 brdf = SampleDiffuse( diffuse, N, r1, r2, 8R, 8pdf
urvive;
pdf;
n = E * brdf * (dot( N, R ) / pdf);
sion = true:
```

SIMT Recap

S.I.M.T.: *Single Instruction, Multiple Thread*.

Adding two arrays, C/C++ way: for(int i = 0; i < N; i++) c[i] = a[i] + b[i]; Adding two arrays in MatLab: c = a + b

Adding two arrays using SIMD:

```
void add(int* a, int* b, int* c, int N)
{
```

for(int i = 0; i < N; i += 4)
{
 ___m128 a4 = ((___m128*)a)[i];</pre>

__m128 b4 = ((__m128*)b)[i]; ((__m128*)c)[i] = a4 + b4; Adding two arrays using SIMT:

```
void add(int* a, int* b, int* c)
{
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    c[i] = a[i] + b[i];
    c[i] += a[b[i]]; // via a lut
    // look ma, no loop!
```



tics & (depth < Monoson

: = inside ? 1 : 1 3 ht = nt / nc, ddn ps2t = 1.0f - nnt D, N); 3)

at a = nt - nc, b = nt at Tr = 1 - (R0 + (1 - Rc Fr) R = (D = nnt - N = (3

= * diffuse; = true;

-:fl + refr)) && (depth < MAXDEDI

), N); refl * E * diffuse; = true;

AXDEPTH)

survive = SurvivalProbability(diffuse
estimation - doing it properly
f;
radiance = SampleLight(&rand, I, &L, &L
e.x + radiance.y + radiance.z) > 0) &&

w = true; at brdfPdf = EvaluateDiffuse(L, N) * Psurvis at3 factor = diffuse * INVPI; at weight = Mis2(directPdf, brdfPdf); at cosThetaOut = dot(N, L); E * ((weight * cosThetaOut) / directPdf) * ();

andom walk - done properly, closely following Sec. /ive)

; t3 brdf = SampleDiffuse(diffuse, N, r1, r2, &R, dpdf) urvive; pdf; n = E * brdf * (dot(N, R) / pdf); sion = true:

SIMD versus SIMT

```
void add(int* a, int* b, int* c, int N)
{
   for( int i = 0; i < N; i += 4 )
   {
      __m128 a4 = ((__m128*)a)[i];
      __m128 b4 = ((__m128*)b)[i];
      ((__m128*)c)[i] = a4 + b4;
   }
}</pre>
```

Benefit of SIMT:

 Easier to read and write; similar to regular scalar flow. void add(int* a, int* b, int* c)
{
 int i = blockIdx.x * blockDim.x +
 threadIdx.x;
 c[i] = a[i] + b[i];
 c[i] += a[b[i]]; // via a lut
 // look ma, no loop!
}

Drawbacks of SIMT:

- Redundant data (here: pointers a, b and c).
- Redundant data (variable i).
- A 'warp' is 32-wide, regardless of data size.
- Scattered memory access is not discouraged.
- Control flow.
- We need *tons* of registers.



Model

), N);

AXDEPTH)

refl * E * diffuse;

survive = SurvivalProb

radiance = SampleLight(e.x + radiance.y <u>+ radi</u>

inte 13863 Δ89386DX-16 SX213 19311389 1NTEL©© '85 ΣΣ

RAX (64-bit)

RBX

RCX

RDX

RBP

RSI

RDI

RSP

R8..R15

XMM0..XMM15

YMM0..YMM15

ZMM0..ZMM31



Register Pressure

On a CPU:

	AX ('accumulator register') BX ('base register')	AH, AL (8-bit) BH, BL	EAX (32-bit) EBX
	CX ('counter register')	CH, CL	ECX
	CA (Counter register)	CH, CL	ECA
	DX ('data register')	DH, DL	EDX
	BP ('base pointer')	EBP	
	SI ('source index')		ESI
	DI ('destination index')		EDI
	SP ('stack pointer')		ESP
			st0st7
iffuse(L, N) * Psu INVPI;			XMM0XMM7

v = true; at brdfPdf = EvaluateDiffuse(L, N) Psurvive at3 factor = diffuse * INVPI; at weight = Mis2(directPdf, brdfPdf); at cosThetaOut = dot(N, L); E * ((weight * cosThetaOut) / directPdf) * (Fact

andom walk - done properly, closely following Soci /ive)

; t3 brdf = SampleDiffuse(diffuse, N, r1, r2, &R, Brdf urvive; pdf; n = E * brdf * (dot(N, R) / pdf); sion = true:

Model

os2t		
5. N		

at a = nt - nc, b = nt at Tr = 1 - (R0 + (1 - R0 Fr) R = (D * nnt - N * (2 ≣ * diffuse;

= true;

efl + refr)) && (depth < MAXDED

D, N); refl * E * diffuse; = true;

AXDEPTH)

survive = SurvivalProbability(diffuse .estimation - doing it properly, if; radiance = SampleLight(&rand, I, &L, .x + radiance.y + radiance.z) > 0) 30

w = true; at brdfPdf = EvaluateDiffuse(L, N) *

at3 factor = diffuse * INVPI; at weight = Mis2(directPdf, brdfPdf); at cosThetaOut = dot(N, L);

E * ((weight * cosThetaOut) / directPdf) * (real

andom walk - done properly, closely following Small /ive)

; at3 brdf = SampleDiffuse(diffuse, N, r1, r2, &R, Bpdf urvive; pdf; n = E * brdf * (dot(N, R) / pdf); sion = true:

Register Pressure

On a CPU: RAX (64-bit) RBX RCX RDX RBP RSI RDI RSP

R8..R15 YMM0..YMM15 (256-bit)









Model

sics & (depth < >>>>⊃

t = inside 7 1 1 1 0 nt = nt / nc, ddn us2t = 1.0f - nnt ∩ n D, N); ≫)

nt a = nt - nc, b = nt nt Tr = 1 - (R0 + (1 - Rc) ir) R = (D ⁼ nnt - N ⁻ (ddn

= * diffuse; = true;

• efl + refr)) && (depth < MAXO

D, N); refl * E * diffuse; = true;

(AXDEPTH)

survive = SurvivalProbability(diffu estimation - doing it properly, it if; radiance = SampleLight(&rand, I, &L = x + radiance.y + radiance.z) > 0)

w = true; at brdfPdf = EvaluateDiffuse(L, N) at3 factor = diffuse * INVPI; at weight = Mis2(directPdf, brdfPdf); at cosThetaOut = dot(N, L); E * ((weight * cosThetaOut) / directPd

andom walk - done properly, closely fo /ive)

; at3 brdf = SampleDiffuse(diffuse, N, r1, r2, &R, &pdf urvive; pdf; n = E * brdf * (dot(N, R) / pdf); sion = true:

Register Pressure

On a GPU:

- Each thread in a warp needs its own registers (32 * N);
- The GPU relies on SMT to combat latencies (32 * N * M).

SMT on the CPU: each core *avoids* latencies.

- Super-scalar execution
- Out-of-order execution
- Branch prediction
- Cache hierarchy
- Speculative prefetching

And, as a 'last line of defense', if a latency happens anyway:

SMT



Model

nics & (depth < 20000)

= inside ? 1 : 1 : 1 ht = nt / nc, ddn bs2t = 1.0f - nnt D, N); 2)

at a = nt - nc, b = nt at Tr = 1 - (R0 + (1 - R0 [r] R = (D ⁺ nnt - N ⁻ (dd)

= * diffuse; = true;

• •fl + refr)) && (denth s US)

D, N); refl * E * diffuse; = true;

AXDEPTH)

survive = SurvivalProbability(diffuse estimation - doing it properly, close if; radiance = SampleLight(&rand, I, **t**, e.x + radiance.y + radiance.z) > 0) & w = true; at brdfPdf = EvaluateDiffuse(L, N) at3 factor = diffuse * INVPI; at weight = Mis2(directPdf, brdfPdf); at cosThetaOut = dot(N, L); E * ((weight * cosThetaOut) / directPdf) andom walk - done properly, closely followin vive) st3 brdf = SampleDiffuse(diffuse, N, r1, r2, &R, appd grvive; pdf;

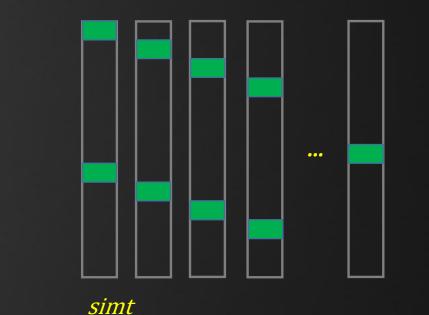
n = E * brdf * (dot(N, R) / pdf); sion = true;

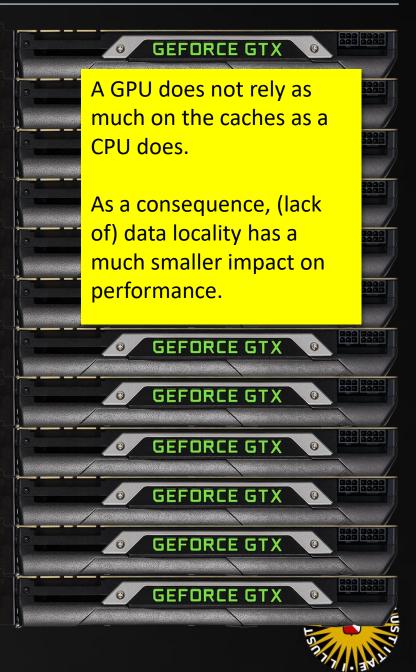
Register Pressure

On a GPU:

- Each thread in a warp needs its own registers (32 * N);
- The GPU relies on SMT to combat latencies (32 * N * M).

SMT on the GPU: *primary weapon* against latencies.





Model

rics & (depth < NoCCS

z = inside / 1 ht = nt / nc, ddh bs2t = 1.0f - nnt -D, N); ð)

at a = nt - nc, b = nt at Tr = 1 - (R0 + (1 - Re Fr) R = (D = nnt - N = (dd

= * diffuse = true;

• efl + refr)) && (depth < MAXDEDIII

), N); refl * E * diffuse; = true;

AXDEPTH)

survive = SurvivalProbability(diffuse estimation - doing it properly, closed if; radiance = SampleLight(&rand, I, &L, & e.x + radiance.y + radiance.z) > 0) &&

v = true; at brdfPdf = EvaluateDiffuse(L, N) * P at3 factor = diffuse * INVPI; at weight = Mis2(directPdf, brdfPdf); at cosThetaOut = dot(N, L); E * ((weight * cosThetaOut) / directPdf

andom walk - done properly, closely f /ive)

, H33 brdf = SampleDiffuse(diffuse, N, r1, r2, &R, &pdf urvive; pdf; n = E * brdf * (dot(N, R) / pdf); sion = true:

Register Pressure

On a CPU, hyperthreading typically *hurts* single thread performance → SMT is limited to 2, max 4 threads.

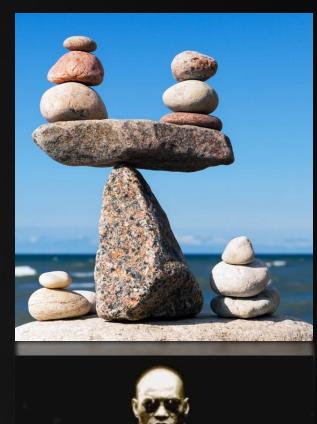
On a GPU, 2 warps per SM is not sufficient: we need 4, 8, 16 or more.

For 16 warps per SM we get:

32 * N * 16, where N is the number of registers one thread wishes to use.

On a typical CPU we have 32 registers ore more available, many of these 256-bit (8-wide AVX registers), others 64-bit.

On a modern GPU, we get 256KB of register space per SM: 32 * 32 * 64 = 65536 32-bit registers per SM.





Model

), N); refl * E * diffuse;

AXDEPTH)

survive = SurvivalProbability(diffu radiance = SampleLight(&rand, I, &L,) e.x + radiance.y + radiance.z) > 0) 88

v = true; at brdfPdf = EvaluateDiffuse(L, N) at3 factor = diffuse * INVPI; at weight = Mis2(directPdf, brdfPdf); at cosThetaOut = dot(N, L); E * ((weight * cosThetaOut) / directPdf)

/ive)

at3 brdf = SampleDiffuse(diffuse, N, r1, r2, &R, &p urvive; pdf; n = E * brdf * (dot(N, R) / pdf); sion = true:

Control Flow

if (threadIdx.x < 16)</pre>

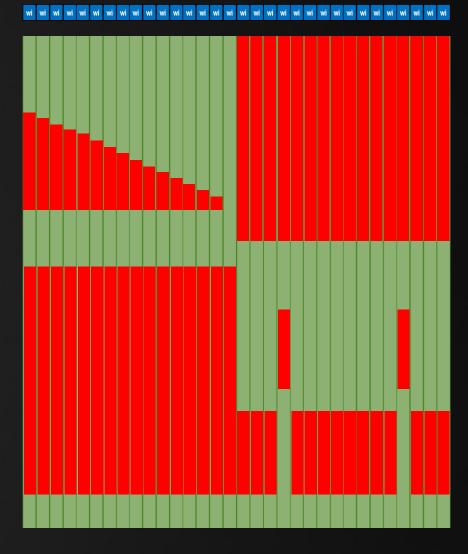
for(int i = 0; i < threadIdx.x; i++)</pre>

else

if (y == 5

// ...

else





nics & (depth < 200000

: = inside } 1 ; . . ht = nt / nc, ddn os2t = 1.0f - nmt 0, N); ∂)

at a = nt - nc, b = nt - m at Tr = 1 - (R0 + (1 - R0) Fr) R = (D ⁺ nnt - N ⁻⁻ (dd)

= * diffuse; = true;

. efl + refr)) && (depth < MOODEPIID

), N); refl * E * diffuse; = true;

AXDEPTH)

survive = SurvivalProbability(diffuse estimation - doing it properly, closed if; radiance = SampleLight(&rand, I, &L, &II e.x + radiance.y + radiance.z) > 0) && ()

v = true; at brdfPdf = EvaluateDiffuse(L, N) * Ps at3 factor = diffuse * INVPI; at weight = Mis2(directPdf, brdfPdf); at cosThetaOut = dot(N, L); E * ((weight * cosThetaOut) / directPdf)

andom walk - done properly, closely /ive)

```
;
t3 brdf = SampleDiffuse( diffuse, N, r1, r2
urvive;
pdf;
n = E * brdf * (dot( N, R ) / pdf);
sion = true:
```

Control Flow

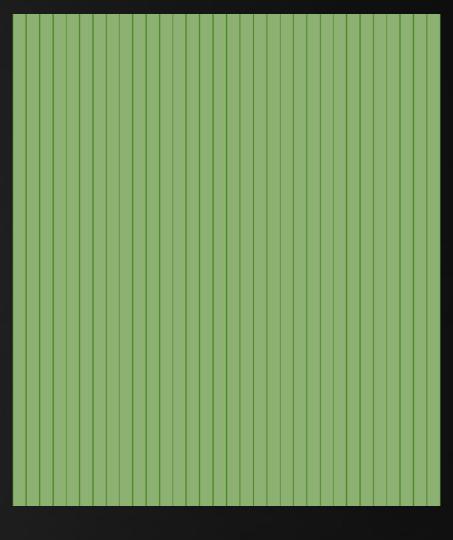
while (1)

// ...
if (Rand() < 0.05f) break;</pre>

while (1)

if (threadIdx.x == 0)
{
 if (Rand() < 0.05f) a[0] = 1;
}
if (a[0] == 1) break;</pre>

Careful: thread 0 is not necessarily the first one to reach the break.





nics & (depth < 200000

at a = nt - nc, b = nt at Tr = 1 - (R0 + (1 - nc) Fr) R = (D = nnt - N = (dom

= * diffuse; = true;

. efl + refr)) && (depth < MOODEPIID

), N); refl * E * diffuse; = true;

AXDEPTH)

survive = SurvivalProbability(diffuse
estimation - doing it properly, closed
f;
radiance = SampleLight(&rand, I, &L, &l
e.x + radiance.y + radiance.z) > 0) && (

w = true; at brdfPdf = EvaluateDiffuse(L, N) PSU at3 factor = diffuse * INVPI; at weight = Mis2(directPdf, brdfPdf); at cosThetaOut = dot(N, L); E * ((weight * cosThetaOut) / directPdf)

andom walk - done properly, closely following Small /ive)

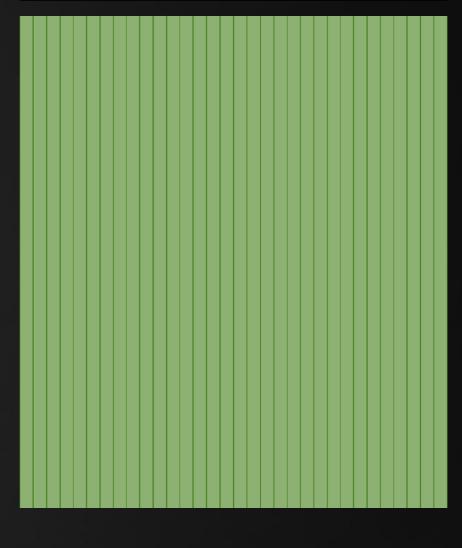
; at3 brdf = SampleDiffuse(diffuse, N, r1, r2, &R, &pd) ; urvive; pdf; n = E * brdf * (dot(N, R) / pdf); sion = true:

Control Flow

while (1)

// ...
if (Rand() < 0.05f) break;</pre>

while (1)





rics & (depth < Mo⊙

: = inside ? 1 ; 1 ; ht = nt / nc, ddn ss2t = 1.0f - nnt 2, N); 2)

at a = nt - nc, b = nt at Tr = 1 - (R0 + (1 - R0 Tr) R = (D = nnt - N = (d)

= * diffuse; = true;

efl + refr)) && (depth < NAXDEPTH

D, N); refl * E * diffuse; = true;

AXDEPTH)

v = true; at brdfPdf = EvaluateDiffuse(L, N) * Psurvive at3 factor = diffuse * INVPI; at weight = Mis2(directPdf, brdfPdf); at cosThetaOut = dot(N, L); E * ((weight * cosThetaOut) / directPdf) * (ref

andom walk - done properly, closely following Sould /ive)

; at3 brdf = SampleDiffuse(diffuse, N, r1, r2, &R, &pdf urvive; pdf; n = E * brdf * (dot(N, R) / pdf); sion = true:

Synchronization

CPU / GPU synchronization: *streams* (CUDA), *queues* (OpenCL).

An OpenCL command is executed *asynchronously*: it simply gets added to the queue.

Example:

void Kernel::Run()

glFinish(); // wait for OpenGL to finish
clEnqueueNDRangeKernel(queue, kernel, 2, 0, workSize, localSize, 0, 0, 0);
clFinish(queue); // wait for OpenCL to finish



nics & (depth < Mocean

: = inside ? 1 | 1.3 ht = nt / nc, ddn - 1 ps2t = 1.0f - nnt - 1 D, N); 3)

at a = nt - nc, b = nt at Tr = 1 - (R0 + (1 - Rc) fr) R = (D = nnt - N = (00)

= * diffuse; = true;

efl + refr)) && (depth < MAXDEPIN

D, N); refl * E * diffuse; = true;

AXDEPTH)

survive = SurvivalProbability(diff: estimation - doing it properly, clo if; radiance = SampleLight(&rand, I, &L, e.x + radiance.y + radiance.z) > 0) &

w = true; at brdfPdf = EvaluateDiffuse(L, N) * Psu at3 factor = diffuse * INVPI; at weight = Mis2(directPdf, brdfPdf); at cosThetaOut = dot(N, L); E * ((weight * cosThetaOut) / directPdf)

andom walk - done properly, closely followir /ive)

; at3 brdf = SampleDiffuse(diffuse, N, r1, r2, urvive; pdf; n = E * brdf * (dot(N, R) / pdf); sion = true:

Synchronization

Fundamental approach to synchronization of GPU threads: don't do it.

...But, if you must:

__syncthreads();

For free:

```
__shared__ int firstSlot;
if (threadIdx.x == 0) firstSlot = atomic_inc( &counter, 32 );
int myIndex = threadIdx.x;
array[firstSlot + myIndex] = resultOfComputation;
```

Warps execute in lockstep, and are therefore synchronized*.

*: On Volta and Turing use __syncwarp(), see: <u>https://devblogs.nvidia.com/inside-volta</u>, section "Independent Thread Scheduling".



nics & (depth < MADDar

: = inside ? | : | : ht = nt / nc, ddn >s2t = 1.0f - nnt 2, N); ≥)

at a = nt - nc, b = nt at Tr = 1 - (R0 + (1 - n) Tr) R = (D = nnt - N = (

= * diffuse; = true;

efl + refr)) && (depth < MAXDEPT

D, N); refl * E * diffuse; = true;

AXDEPTH)

survive = SurvivalProbability(diffuse estimation - doing it properly, close ff; radiance = SampleLight(&rand, I, &L, e.x + radiance.y + radiance.z) > 0) &&

v = true; at brdfPdf = EvaluateDiffuse(L, N) * Psum at3 factor = diffuse * INVPI; at weight = Mis2(directPdf, brdfPdf); at cosThetaOut = dot(N, L); E * ((weight * cosThetaOut) / directPdf)

andom walk - done properly, closely following Sec. /ive)

; at3 brdf = SampleDiffuse(diffuse, N, r1, r2, &R, apdi urvive; pdf; n = E * brdf * (dot(N, R) / pdf); sion = true:

Synchronization

Threads on a single SM can communicate via global memory, or via shared memory.

In CUDA:

__global___void reverse(int* d, int n)

__shared__ int s[64]; int t = threadIdx.x; int tr = n-t-1; s[t] = d[t]; __syncthreads(); d[t] = s[tr];

Warp Scheduler + Dispatch (32 thread/cli Warp Scheduler + Dispatch (32 thread/cik Register File (16,384 x 32-bit) Register File (16,384 x 32-bit) TENSOR TENSOR **FP32** FP32 **INT32 INT32** CORES CORES SFU LD/ST LD/ST LD/ST LD/ST LD/ST LD/ST LD/ST SFU LD/ST Register File (16,384 x 32-bit) Register File (16,384 x 32-bit) TENSOR TENSOR **INT32 FP32 INT32 FP32** CORES CORES LD/ST LD/ST LD/ST LD/ST SFU LD/ST LD/ST LD/ST LD/ST SFU 96KB L1 Data Cache / Shared Memory Tex Tex Tex Tex

nics & (depth < Morros

c = inside ? 1 : 5 ; nt = nt / nc, ddn ss2t = 1.0f - nnt D, N); B)

at a = nt - nc, b = 11 at Tr = 1 - (R0 + (1 - FFr) R = (D * nnt - N

= * diffuse; = true;

. efl + refr)) && (depth < MAXDEPT

D, N); refl * E * diffuse; = true;

AXDEPTH)

survive = SurvivalProbability(diffuse .estimation - doing it properly. df; radiance = SampleLight(&rand, I, &L, e.x + radiance.y + radiance.z) > 0) &&

v = true; at brdfPdf = EvaluateDiffuse(L, N) * Psurv at3 factor = diffuse * INVPI; at weight = Mis2(directPdf, brdfPdf); at cosThetaOut = dot(N, L); E * ((weight * cosThetaOut) / directPdf)

andom walk - done properly, closely following Small /ive)

; at3 brdf = SampleDiffuse(diffuse, N, r1, r2, &R, Epdi urvive; pdf; n = E * brdf * (dot(N, R) / pdf); sion = true:

Synchronization

Threads on a single SM can communicate via global memory, or via shared memory.

In OpenCL:

_kernel void reverse(global int* d, int n)

_local int s[64]; int t = get_local_id(0); int tr = n-t-1; s[t] = d[t]; barrier(CLK_LOCAL_MEM_FENCE); d[t] = s[tr];



ics & (depth < Mode∈)

: = inside ? 1 + . . ht = nt / nc, ddn os2t = 1.0f - nnt O, N); 3)

at a = nt - nc, b = nt at Tr = 1 - (R0 + (1 - R0 Fr) R = (D ⁼ nnt - N - (00)

= * diffuse = true;

efl + refr)) && (depth < MODECT)

), N); refl * E * diffuse; = true;

AXDEPTH)

survive = SurvivalProbability(diffuse
estimation - doing it properly, closed
f;
radiance = SampleLight(&rand, I, &L, &light()
e.x + radiance.y + radiance.z) > 0) && (doing)

w = true; at brdfPdf = EvaluateDiffuse(L, N) Psurvive at3 factor = diffuse * INVPI; at weight = Mis2(directPdf, brdfPdf); at cosThetaOut = dot(N, L); E * ((weight * cosThetaOut) / directPdf) (radd)

andom walk - done properly, closely following Sec. /ive)

; t3 brdf = SampleDiffuse(diffuse, N, r1, r2, &R, &pdf urvive; pdf; n = E * brdf * (dot(N, R) / pdf); Sion = true:

Today's Agenda:

- GPU Execution Model
- GPGPU Flow
- GPGPU Low Level Notes
- P3



Flow

A Typical GPGPU Program

Calculating anything using a GPU kernel:

- Setup input data on the CPU
- Transfer input data to the GPU 2.
- Operate on the input data 3.
- Transfer the result back to the CPU 4.
- 5. Profit.

), N); refl * E * diffuse; = true;

(AXDEPTH)

= true:

survive = SurvivalProbability(dif radiance = SampleLight(&rand, I, e.x + radiance.y + radiance.z) > (v = true:

at brdfPdf = EvaluateDiffuse() at3 factor = diffuse * INVPI; at weight = Mis2(directPdf, brdfPdf at cosThetaOut = dot(N, L); E * ((weight * cosThetaOut) / directPd

andom walk - done properly, closely fol /ive)

at3 brdf = SampleDiffuse(diffuse, N, r1, r2, &R, A urvive; pdf; 1 = E * brdf * (dot(N, R) / pdf);

Amdahl's law:

 $S_{peedup} < \frac{1}{1-p}$,

where *p* is the portion of the code that is parallelizable.



.

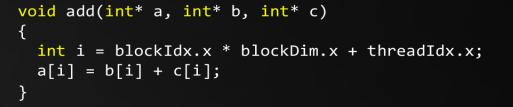
A Typical GPGPU Program

2. Transfer input data to the GPU.

Flow



n = E * brdf* (dot(N, R) / pd1 sion = tr



diffuse;		Context I [U]						
true;		Runtime API	\mathbb{Y}		cudaMemcpy [3636]	cudaMemcpy [3636]		cudaMemcpy [3636]
+ refr)) &		Nsight	\mathbf{Y}	Ranges			Flushing Re	
↓); L * E * dif		Memory	\mathbf{Y}	1	8.00 MB HostToDevice	8.00 MB HostToDevice		8.00 MB DeviceToHost
true;		Compute	\mathbf{Y}	Í			add	
DEPTH) vive = Surv		0.3% [1] a	\mathbf{Y}	Î			add	
imation -	=	Streams		8				
lance = Sam + radiance		Stream 1	\mathbb{Y}		8.00 MB HostToDevice	8.00 MB HostToDevice	add	8.00 MB DeviceToHost
factor = d weight = Mi cosThetaOut	iffuse * s2(dired = dot(M		ance		BOD ME HORIZODANKE	EXEMPTOR NUMBER OF STREET		- rele
om walk - de :)								
brdf = Sam ive; F;	pleDiffus	se(diffuse, N, r1, r2, 8R,						3.4

```
INFOMOV – Lecture 10 – "GPGPU (3)"
```

Flow

tics ≰(depth < Morran

at a = nt - nc, b = nt at Tr = 1 - (R0 + (1 - R0) Fr) R = (D [#] nnt - N [#] (30)

* diffuse; = true;

efl + refr)) && (depth < MAXDEPT

D, N); refl * E * diffuse; = true;

AXDEPTH)

```
survive = SurvivalProbability( diffuse
estimation - doing it properly
if;
radiance = SampleLight( &rand, I, &L
e.x + radiance.y + radiance.z) > 0
```

w = true; at brdfPdf = EvaluateDiffuse(L, N) * P: at3 factor = diffuse * INVPI; at weight = Mis2(directPdf, brdfPdf); at cosThetaOut = dot(N, L); E * ((weight * cosThetaOut) / directPdf)

andom walk - done properly, closely following /ive)

, H33 brdf = SampleDiffuse(diffuse, N, r1, r2, 8R, 8pd urvive; pdf; n = E * brdf * (dot(N, R) / pdf); sion = true:

A Typical GPGPU Program

2. Transfer input data to the GPU.

Optimizing transfers:

- Reduce the *number* of transfers first, then their size.
- Only send changed data.
 - Use asynchronous copies.

If possible:

Produce the input data on the GPU.

For visual results:

Store visual output directly to a texture.

```
void add(int* a, int* b, int* c)
{
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    a[i] = b[i] + c[i];
}
```



Flow

Asynchronous Copies

OpenCL supports multiple queues:

nt = nt / nc, ddn os2t = 1.0f - nnt 0, N); 3)

at a = nt - nc, b = nt at Tr = 1 - (R0 + (1 - R0 Fr) R = (D = nnt - N = (2

= * diffuse; = true;

. efl + refr)) && (depth < Maddep

), N); refl * E * diffuse; = true;

AXDEPTH)

survive = SurvivalProbability(diffus estimation - doing it properly if; radiance = SampleLight(&rand, I, &L, 2.x + radiance.y + radiance.<u>x) > 0) &</u>

w = true; at brdfPdf = EvaluateDiffuse(L, N) * Ps at3 factor = diffuse * INVPI; at weight = Mis2(directPdf, brdfPdf); at cosThetaOut = dot(N, L); E * ((weight * cosThetaOut) / directPdf)

andom walk - done properly, closely following See /ive)

; at3 brdf = SampleDiffuse(diffuse, N, r1, r2, &R, dpdf urvive; pdf; n = E * brdf * (dot(N, R) / pdf); sion = true:

queue = clCreateCommandQueue(context, devices[...], 0, &error);

Kernels and copy commands can be added to any queue:

```
clEnqueueNDRangeKernel( queue, kernel, 2, 0, workSize, 0, 0, 0, 0);
clEnqueueWriteBuffer( Kernel::GetQueue(), ... );
```

Queues can wait for a signal from another queue:

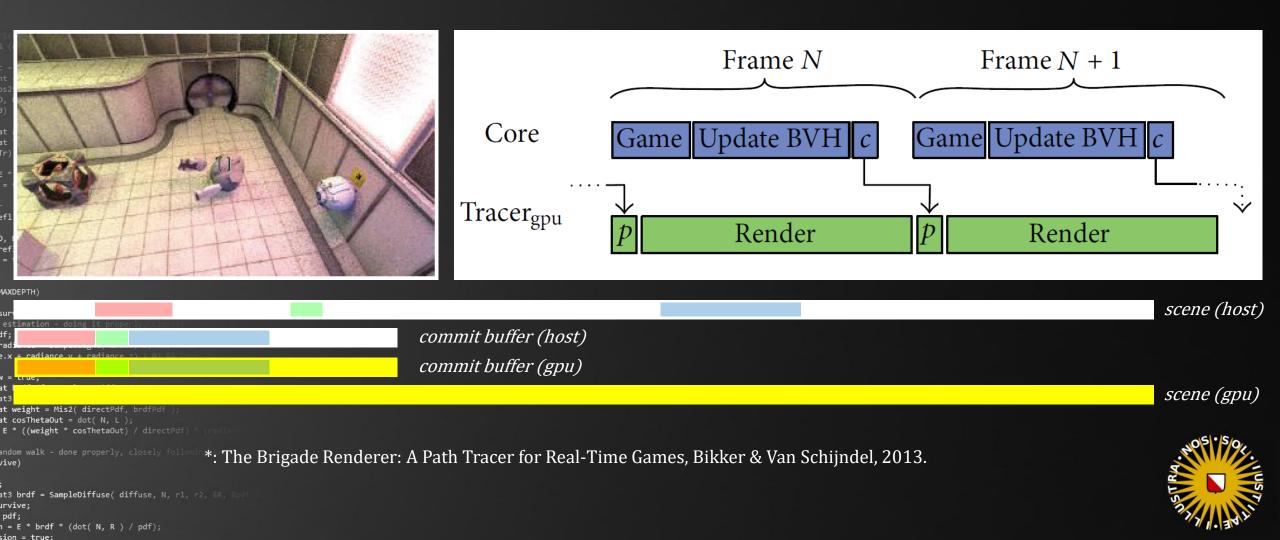
```
clEnqueueBarrierWithWaitList( ... );
```

CUDA provides similar functionality.



Flow

Asynchronous Copies





ics & (depth < Mode∈)

: = inside ? 1 + . . ht = nt / nc, ddn os2t = 1.0f - nnt O, N); 3)

at a = nt - nc, b = nt at Tr = 1 - (R0 + (1 - R0 Fr) R = (D ⁼ nnt - N - (00)

= * diffuse = true;

efl + refr)) && (death < MODESSI

), N); refl * E * diffuse; = true;

AXDEPTH)

survive = SurvivalProbability(diffuse
estimation - doing it properly, closed
f;
radiance = SampleLight(&rand, I, &L, &light()
e.x + radiance.y + radiance.z) > 0) && (doing)

w = true; at brdfPdf = EvaluateDiffuse(L, N) Psurvive at3 factor = diffuse * INVPI; at weight = Mis2(directPdf, brdfPdf); at cosThetaOut = dot(N, L); E * ((weight * cosThetaOut) / directPdf) (radd)

andom walk - done properly, closely following Sec. /ive)

; t3 brdf = SampleDiffuse(diffuse, N, r1, r2, &R, &pdf urvive; pdf; n = E * brdf * (dot(N, R) / pdf); Sion = true:

Today's Agenda:

- GPU Execution Model
- GPGPU Flow
- GPGPU Low Level Notes
- P3



ics & (depth < MAXDE

: = inside ? 1 : 1 . ht = nt / nc, ddn os2t = 1.0f - nnt ?), N); ?)

at a = nt - nc, b = N at Tr = 1 - (R0 + (1 - R0 Fr) R = (D ⁺ nnt - N - (1

* diffuse; = true;

• efl + refr)) && (depth < MAXO

D, N); refl * E * diffuse; = true;

AXDEPTH)

survive = SurvivalProbability(diffuse estimation - doing it properly, closed if; radiance = SampleLight(&rand, I, &L, &L e.x + radiance.y + radiance.z) > 0) &&

w = true; at brdfPdf = EvaluateDiffuse(L, N) * P at3 factor = diffuse * INVPI; at weight = Mis2(directPdf, brdfPdf); at cosThetaOut = dot(N, L); E * ((weight * cosThetaOut) / directPdf

andom walk - done properly, closely follo /ive)

, t33 brdf = SampleDiffuse(diffuse, N, r1, r2, &R, apdf) urvive; pdf; n = E * brdf * (dot(N, R) / pdf); sion = true:

Your Mission

"Optimize an application using the process and means discussed in INFOMOV."

"An application":

- 1. <u>One of your own</u>. Requirement: functionality must be 'done', optimization may not purely be a port to C/C++.
- 2. One of <u>Roland's Projects</u>. Additional benefit: goodies if you win. Also: winning. Will be introduced today in The Final Hour.
- 3. One of my projects. Options: <u>animation module</u> of Lighthouse 2, and a <u>simpler</u> <u>application</u>. Simple application grade will be capped at 7.
- 4. A <u>single-header library</u> from GitHub. Lists: <u>here</u> and <u>here</u>. You will have to setup your own test case, and you are expected to submit the optimized code (INFOMOV-branded) to the original repo.
- 5. Any <u>GitHub / open source project</u>, if you think you can handle it. Warning: last option on this list for a reason.



nics & (depth < MoxOS

: = inside 7 1 (). ht = nt / nc, ddn os2t = 1.0f - nnt 0, N); 2)

at a = nt - nc, b = nt at Tr = 1 - (R0 + (1 - RC Fr) R = (D * nnt - N * (dd

* diffuse; = true;

-:fl + refr)) && (depth < MAXDEPT

), N); refl * E * diffuse; = true;

(AXDEPTH)

survive = SurvivalProbability(diffuse estimation - doing it properly, close ff; radiance = SampleLight(&rand, I, &L, l e.x + radiance.y + radiance.z) > 0) &&

w = true; at brdfPdf = EvaluateDiffuse(L, N) * F at3 factor = diffuse * INVPI; at weight = Mis2(directPdf, brdfPdf); at cosThetaOut = dot(N, L); E * ((weight * cosThetaOut) / directPdf

andom walk - done properly, closely f /ive)

; at3 brdf = SampleDiffuse(diffuse, N, r1, r2, &R, &pdf urvive; pdf; n = E * brdf * (dot(N, R) / pdf); sion = true:

Your Mission

"Optimize an application using the process and means discussed in INFOMOV."

"The Process":

. Establish optimization goal (optional).

- 2. Profile.
- 3. Apply high-level optimization (on hotspot).
- 4. Profile.
- 5. Multi-thread / vectorize / apply GPGPU, if applicable.
- 6. Profile.
- 7. Apply low-level optimizations.
- 8. Repeat step 6 and 7 until time runs out.
- 9. Report.

Your report should provide clear proof that you approached the optimization in a structured manner, i.e. it will provide profiling information at every step.



ics & (depth < ∧∨∞co

: = inside } 1 ; ; ; ; ht = nt / nc, ddn os2t = 1.0f - nnt 0, N); 3)

at a = nt - nc, b = nt at Tr = 1 - (R0 + (1 - R0 Fr) R = (D * nnt - N * (dd)

* diffuse; = true;

. :fl + refr)) && (depth < MODE

D, N); refl * E * diffuse; = true;

(AXDEPTH)

survive = SurvivalProbability(diffuse estimation - doing it properly if; radiance = SampleLight(&rand, I, &L, &L) e.x + radiance.y + radiance.z) > 0) &&

w = true; at brdfPdf = EvaluateDiffuse(L, N) at3 factor = diffuse * INVPI; at weight = Mis2(directPdf, brdfPdf); at cosThetaOut = dot(N, L);

E * ((weight * cosThetaOut) / directPdf) = ()

andom walk - done properly, closely following Sec. /ive)

; t3 brdf = SampleDiffuse(diffuse, N, r1, r2, &R, lpdf) urvive; pdf; n = E * brdf * (dot(N, R) / pdf); sion = true:

Your Mission

"Optimize an application using the process and means discussed in INFOMOV."

"Means":

1. High-level optimizations (typically those that change algorithmic complexity).

- 2. Low-level optimizations (see "Rules of Engagement").
- 3. Data-Oriented Design.
- 4. Anything else to please the cache.
- 5. SIMD.
- 6. GPGPU.
- 7. Compiler output inspection, compiler choice, compiler settings.

Note that overclocking is not in this list.



nics & (depth < ≫occs

: = inside ? 1 : 1 : 3 ht = nt / nc, ddn os2t = 1.0f - nnt 0, N); 3)

at a = nt - nc, b = nt - n at Tr = 1 - (R0 + (1 - RC Fr) R = (D ⁼ nnt - N ⁻ (dd

= * diffuse; = true;

. efl + refr)) && (depth < MAXDE

D, N); refl * E * diffuse; = true;

AXDEPTH)

survive = SurvivalProbability(diffuse)
estimation - doing it properly, closed
f;
radiance = SampleLight(&rand, I, &L, &light
e.x + radiance.y + radiance.z) > 0) && (dot)

w = true; at brdfPdf = EvaluateDiffuse(L, N) Psurviv at3 factor = diffuse * INVPI; at weight = Mis2(directPdf, brdfPdf); at cosThetaOut = dot(N, L); E * ((weight * cosThetaOut) / directPdf) (();

andom walk - done properly, closely following Sec. /ive)

; at3 brdf = SampleDiffuse(diffuse, N, r1, r2, 8R, 8pdf urvive; pdf; n = E * brdf * (dot(N, R) / pdf); sion = true:

Your Mission

"Optimize an application using the process and means discussed in INFOMOV."

Notes:

- L. Do not alter functionality.
- 2. If you skip optimizations to maintain readability: indicate this in the report.
- 3. Multiple teams may work on the same base code. Do not share optimized code in these cases; sharing ideas is still allowed however.

Don't forget to maintain a healthy work/life balance. Or fix that after the deadline.



nics & (depth < MAXDE

at a = nt - nc, b = nt at Tr = 1 - (R0 + (1 - Rc) Fr) R = (D = nnt - N = (dd)

= * diffuse; = true;

. :fl + refr)) && (depth < MAXDEPT

D, N); refl * E * diffuse; = true;

AXDEPTH)

survive = SurvivalProbability(diffuse)
estimation - doing it properly, closed
if;

radiance = SampleLight(&rand, I, &L 2.x + radiance.y + radiance.z) > 0) &&

v = true; at brdfPdf = EvaluateDiffuse(L, N) * Psurvive at3 factor = diffuse * INVPI; at weight = Mis2(directPdf, brdfPdf); at cosThetaOut = dot(N, L); E * ((weight * cosThetaOut) / directPdf) * (reading);

andom walk - done properly, closely following Small /ive)

; at3 brdf = SampleDiffuse(diffuse, N, r1, r2, &R, &pdf) urvive; pdf; n = E * brdf * (dot(N, R) / pdf); sion = true:

/INFOMOV/

END of "GPGPU (3)"

next lecture: "fixed point"

