nics & (depth < NAXOER

at a = nt - nc, b = nt at Tr = 1 - (R0 + (1 - R0 Tr) 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 close
if;
radiance = SampleLight(&rand, I, &L, &light)
extinue.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) * (Fad

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 - April-June 2024 - Lecture 2: "Low Level"

Welcome!



ics & (depth < Modern

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

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

= * diffuse = true;

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

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

AXDEPTH)

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

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

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

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

Today's Agenda:

- The Cost of a Line of Code
- CPU Architecture: Instruction Pipeline
- Data Types and Their Cost
- Rules of Engagement



Instruction Cost

nics & (depth < Notic

: = inside } l ht = nt / nc, ddn bs2t = 1.0f - nnt 2, N); 3)

t a = nt - nc, b = nt t Tr = 1 - (R0 + (1 - R0 r) R = (D = nnt - N = (3

= * diffuse; = true;

efl + refr)) && (depth < ⊨

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

AXDEPTH)

survive = SurvivalProbability(diffs: estimation - doing it properly, if; radiance = SampleLight(&rand, I, &L 2.x + radiance.y + radiance.<u>z) > 0) 8</u>

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)

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

What is the 'cost' of a multiply?

starttimer();
float x = 0;
for(int i = 0; i < 1000000; i++) x *= y;
stoptimer();</pre>

- Actual measured operations:
 - timer operations;
 - initializing 'x' and 'i';
 - comparing 'i' to 1000000 (x 1000000);
 - increasing 'i' (x 100000);
 - jump instruction to start of loop (x 100000).
 - Compiler outsmarts us!
 - No work at all unless we use x
 - x += 1000000 * y

Better solution:

- Create an arbitrary loop
- Measure time with and without the instruction we want to time



Instruction Cost

nics & (depth < Moors

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

= * diffuse; = true;

efl + refr)) && (depth < MAXDEDI

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) &&

v = 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 See /ive)

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

What is the 'cost' of a multiply?

float x = 1, y = 1; unsigned int i = 0, j = 0x28929227; for(int k = 0; k < ITERATIONS; k++)</pre>

// ensure we feed our line with fresh data
x += y, y *= 0.9999f;
// integer operations to free up fp execution units
i += j, j ^= 0x17737352, i >>= 1, j /= 28763;
// operation to be timed
if (with) x *= y;
// integer operations to free up fp execution units
i += j, j ^= 0x17737352, i >>= 1, j /= 28763;

dummy = x + (float)i;

	Generate Graph of Include Files	
X	Show Call Stack on Code Map	Ctrl+Shift+`
ta	Insert Snippet	Ctrl+K, Ctrl+X
ta	Surround With	Ctrl+K, Ctrl+S
I	Peek Definition	Alt+F12
1	Go To Definition	F12
⇒≣	Go To Declaration	Ctrl+Alt+F12
	Find All References	
Ξ.	View Call Hierarchy	Ctrl+K, Ctrl+T
	Toggle Header / Code File	Ctrl+K, Ctrl+O
	Intel Advisor XE 2015	•
	Breakpoint	•
60	Add Watch	
⇔	Add Parallel Watch	
60	QuickWatch	Shift+F9
	Pin To Source	
	View Array	
→	Show Next Statement	Alt+Num *
k	Run To Cursor	Ctrl+F10
-le-	Run Flagged Threads To Cursor	
1	Set Next Statement	Ctrl+Shift+F10
Ģ	Go To Disassembly	
ж	Cut	Ctrl+X
ŋ	Сору	Ctrl+C
â	Paste	Ctrl+V
	Outlining	•
	Intel Compiler	•
	Options	Ctrl+1



Instruction Cost

ics & (depth < Motoss

t a = nt - nc, b = nt t Tr = 1 - (R0 + (1 - 1 r) R = (D ⁺ nnt - N ⁻

: * diffuse; = true;

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

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

AXDEPTH)

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

w = true; at brdfPdf = EvaluateDiffuse(L, at3 factor = diffuse * TM/DT.

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)

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

x86 assembly in 5 minutes

Modern CPUs still run x86 machine code, based on Intel's 1978 8086 processor. The original processor was 16-bit, and had 8 'general purpose' 16-bit registers*:

AX ('accumulator register')	AH, AL (8-bit)	EAX (32-bit)	RAX (64-bit)
BX ('base register')	BH, BL	EBX	RBX
CX ('counter register')	CH, CL	ECX	RCX
DX ('data register')	DH, DL	EDX	RDX
BP ('base pointer')		EBP	RBP
SI ('source index')		ESI	RSI
DI ('destination index')		EDI	RDI
SP ('stack pointer')		ESP	RSP
		st0st7	R8R15
		XMM0XMM7	XMM0XMM
following Section			YMM0YMM
* More info: <u>http://www.swansonte</u> 1, r1, r2, &R, &pdf J	ec.com/sregisters.html		ZMM0ZMM3





Instruction Cost

rics & (depth < ⊁VXCC)

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

= * diffuse; = true;

efl + refr)) && (depth < MAXDEP

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

AXDEPTH)

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

w = true; at brdfPdf = EvaluateDiffus<u>e(L, N)</u>

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

E * ((weight * cosThetaOut) / directPdf

andom walk - done properly, closely fol /ive)

```
st3 brdf = SampleDiffuse( diffuse, N, r1, r2, 8R, 8pdf
prvive;
pdf;
n = E * brdf * (dot( N, R ) / pdf);
sion = true:
```

x86 assembly in 5 minutes:

Typical assembler:

loop:

mov	eax,	[0x1008FFA0
shr	eax,	5
add	eax,	edx
dec	ecx	
jnz	loop	
fld	[esi]	
fld	st0	
fadc	lp	

// read from address into register // shift eax 5 bits to the right // add registers, store in eax // decrement ecx // jump if not zero // load from address [esi] onto FPU // duplicate top float // add top two values, push result

More on x86 assembler: <u>http://www.cs.virginia.edu/~evans/cs216/guides/x86.html</u> A bit more on floating point assembler: <u>https://www.cs.uaf.edu/2007/fall/cs301/lecture/11_12_floating_asm.html</u>



Instruction Cost

What is the 'cost' of a multiply?

float x = 0, y = 0.1f;unsigned int i = 0, j = 0x28929227; for(int k = 0; k < ITERATIONS; k++)</pre>

fl + refr)) && (depth

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

AXDEPTH)

survive = SurvivalProbability ff: radiance = SampleLight(&rand e.x + radiance.y + radiance.z)

v = true; at brdfPdf = EvaluateDiffus at3 factor = diffuse * INV* at weight = Mis2(directPdf brace x + (float)i;

E * ((weight * cosThetaOut) / directPd

andom walk - done properly, closely follow /ive)

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





Instruction Cost

ics & (depth < MODOR

: = inside 7 1 1 1 0 nt = nt / nc, ddn 1 ps2t = 1.0f - nnt 1 p, N); 2)

t a = nt - nc, b = nt t Tr = 1 - (R0 + (1 - R0 r) R = (D ⁼ nnt - N = (d0

= * diffuse; = true;

efl + refr)) && (depth < MODEP

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

AXDEPTH)

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

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) (Psurviv);

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

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

What is the 'cost' of a multiply?

Observations:

- Compiler reorganizes code
- Compiler cleverly evades division
- Loop counter *decreases*
- Presence of integer instructions affects timing (to the point where the mul is free)

But also:

It is really hard to measure the cost of a line of code.



Instruction Cost

nics & (depth < Moorean

t a = nt - nc, b = mt t Tr = 1 - (R0 + (1 - R0 r) R = (D * nnt - N * (3

* diffuse; = true;

efl + refr)) && (depth < MOXDEP

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

AXDEPTH)

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

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

What is the 'cost' of a single instruction?

Cost is highly dependent on the surrounding instructions, and many other factors. However, there is a 'cost ranking':

bit shifts simple arithmetic, logical operands multiplication division

sin, cos, tan, pow, exp

<< >>

+ - & | ^

*

sqrt

This ranking is generally true for any processor (including GPUs).



INFOMOV – Lecture 2 – "Low Level"	Instruction	Operands	Ops	Latency	Reciprocal throughput	Execution unit	Notes	10
Instruction Cost	Arithmetic instructions ADD, SUB ADD, SUB ADD, SUB ADC, SBB ADC, SBB	r,r/i r,m m,r r,r/i	1 1 1 1 1 1	1 1 7 1	1/3 1/2 2,5 1/3	ALU ALU, AGU ALU, AGU ALU ALU		
<pre>sics & (depth < MAXDEFIN : = inside ? 1 1 1 1 1 it = nt / nc, ddn it = nt / nc, ddn it = nt / nc, ddn os2t = 1.0f - nnt sot Tr = 1 - (R0 + (1 - R0 Tr) R = (D * nnt - N * (don t * diffuse; = true;</pre>	ADC, SBB CMP FSI CMP INC, DEC, NEG INC, DEC, NEG AAA, AAS DAA DAA DAA FSI AAA, AAS FPA DAA FSI AAD AAD FSI FSI FSI FSI FSI FSI FSI FSI	DRT N DS NCOS TAN ATAN CALE TRACT (M1 -2X -2XP1			1 44 90 51 90 76 10 46 10 72 16 5 7 8 49 63	35 12 0-100 0-100 0-150 0-200 0-170 8 11 27 126 147		
<pre>fil + refr)) && (depth < PAUDEPTH) fil + refr)) && (depth < PAUDEPTH) fil + E * diffuse; = true; AXXDEPTH) survive = SurvivalProbability(diffuse) estimation - doing it properly. Closed fi; radiance = SampleLight(&rand, I, &L, & filence, e.x + radiance.y + radiance.z) > 0) && (dot) x = true; th brdfPdf = EvaluateDiffuse(L, N) * Paurvive at3 factor = diffuse * INVPI; at weight = Mis2(directPdf, brdfPdf); at cosThetaOut = dot(N, L); E * ((weight * cosThetaOut) / directPdf) * (radiance andom walk - done properly, closely following Section vive) it3 brdf = SampleDiffuse(diffuse, N, r1, r2, &R, &pdf urvive; pdf; pdf = i brdf * (dot(N, R)) / pdf); </pre>	MUL, IMUL MUL, IMUL IMUL IMUL IMUL IMUL IMUL DIV DIV DIV DIV IDIV IDIV IDIV IDIV	r16/m16 r32/m32 r16,r16/m16 r32,r32/m32 r16,(r16),i r32,(r32),i r16,m16,i r32,m32,i r8/m8 r16/m16 r32/m32 r8 r16 r32 m8 m16	3 2 2 2 2 3 3 32 47 79 41 56 88 42 57	3 4 3 4 5 24 24 40 17 25 41 17 25	2 3 2,5 1 2 2 2 2 2 3 23 40 17 25 41 17 25	ALU0_1 ALU0 ALU0 ALU0 ALU0 ALU0 ALU0 ALU0 ALU ALU ALU ALU ALU ALU ALU ALU ALU ALU	dx=4	

Instruction Cost

nics & (depth < Motos

: = inside ? | ; | ; ht = nt / nc, ddn os2t = 1.0f - nnt 2, N); ∂)

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

= * diffuse; = true:

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

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

AXDEPTH)

sion = true:

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

v = 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, clos Note: Two micro-operations can vive) execute simultaneously if they at3 brdf = SampleDiffuse(diffuse, go to different execution pipes pdf; n = E * brdf * (dot(N, R) / pdf);

AMD Jaguar

2013

IDIV

IDIV

r32/m32

r64/m64

2

2

12-27

12-43

Instruction Reciprocal Operands Ops Execution Notes Latency throughput pipe Arithmetic instructions ADD, SUB r,r/i 1 0.5 10/11 ADD, SUB r.m 1 ADD, SUB 1 6 m,r ADC. SBR 1 1 10/1r r/i 1 ADC. Math FSQRT 35 35 FP1 ADC. 1 FLDPI, etc. FP0 CMP 1 FSIN 30-139 30-151 FP0, FP1 4-44 CMP FCOS 11-51 38-93 FP0, FP1 INC. E FSINCOS 11-76 55-122 55-180 **FP0, FP1** INC. E FPTAN 11-45 55-177 55-177 FP0, FP1 AAA FPATAN FP0, FP1 9-75 44-167 44-167 AAS FSCALE 27 FP0, FP1 5 DAA FXTRACT FP0, FP1 7 9 6 DAS F2XM1 32-37 FP0, FP1 8 AAD FYL2X 30-120 FP0, FP1 8-51 30-120 AAM FYL2XP1 61 ~160 ~160 FP0, FP1 MUL. I MUL, IMUL r16/m16 3 3 10 3 MUL. IMUL r32/m32 2 3 2 10 MUL, IMUL r64/m64 6 2 5 10 IMUL r16,r16/m16 3 10 IMUL r32,r32/m32 3 10 IMUL r64,r64/m64 6 10 1 IMUL r16,(r16),i 2 4 10 IMUL r32,(r32),i 3 10 1 IMUL r64,(r64),i 6 10 1 4 DIV r8/m8 1 11-14 11-14 10 DIV r16/m16 2 12-19 12-19 10 DIV r32/m32 2 12-27 12-27 10 DIV 12-43 r64/m64 2 12-43 10 IDIV 11-14 11-14 10 r8/m8 1 IDIV 12-19 r16/m16 2 12-19 10

12-27

12-43

10

10



Instruction Cost

nics & (depth < MAXDE

c = inside ? | ; ; ; ht = nt / nc, ddn ps2t = 1.0f - nnt 2, N); 2)

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

* diffuse; = true:

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

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

AXDEPTH)

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

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) (1865)

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:

Intel Skylake 2015

ADD SUB		r,r/i	1	1	p0156	23	1	0.25			12
ADD SUB		m,r/i	2	4	2p0156 2p2	37 p4	5	1			
ADC SBB		r,r/i	1	1	p06		1	1			
ADC SBB		r,m	2	2	p06 p2	3		1			
ADC SBB		m,r/i	4	6	3p0156 2p2	37 p4	5	2			
CMP		r,r/i	1	1	p0156	;	1	0.25			
CMP		m,r/i	1	2	p0156 p	23	1	0.5			
INC DEC NE NOT	G	r	1	1	p0156	6	1	0.25			
INC DEC NO	т	m	3	4	p0156 2p2	37 p4	5-6	1			
NEG		m	2	4	p0156 2p2	37 p4	5-6	1			
AAA			2	2	p1 p56	6 [.]	4		not 64 bit		
AAS	1 I		<u>^</u>	2		ا م	1				
DAA DAS	Math	h									
AAD	FSC	ALE			27	27	'		130	130	
AAM	FXT	RACT			17	17	·		11	11	
MUL IMU	FSQ	RT			1	1		p0	14-21	1 4-7	
MUL IMU	FSIN	J			53-105				50-12	0	
MUL IMU	FCO	S			53-105				50-13	0	
MUL IMU	FSIN	ICOS			55-120				55-15	0	
MUL IMU	F2X	M1			16-90				65-80	D	
MUL IMU	FYL 2	2X			40-100				103		
MUL IMU	FYL	2XP1			56				77		
MUL IMU	FPT/	AN			40-112				140-16	60	
IMUL	FPA	TAN			30-160				100-16	60	
IMUL	<u> </u>	r,m	1	2	p1 p23	3		1	- ¹	1	
IMUL		r16,r16,i	2	2	p1 p015	56	4	1			
IMUL		r32,r32,i	1	1	p1		3	1			
IMUL		r64,r64,i	1	1	p1		3	1			
IMUL		r16,m16,i	2	3	p1 p0156	p23		1			051.50
IMUL		r32,m32,i	1	2	p1 p23	3		1		, Š	SW//y
IMUL		r64,m64,i	1	2	p1 p23	3		1		5	
MULX		r32,r32,r32	3	3	p1 2p08	56	4	1	BMI2	E.	
MULX		r32,r32,m32	3	4	p1 2p056	p23		1	BMI2	2	
MULX		r64,r64,r64	2	2	p1 p5		4	1	BMI2		1. 3N
								1 · · · ·			

INFOMOV – Lecture 2 – "L	low Level"	Arithmetic i	instructions	3							13
		ADD, SUB		r,r	1	1	0.25				
Instruction Co	ct	ADD, SUB		r,i	1	1	0.25				
	31	ADD, SUB		r,m	1		0.33				
		ADD, SUB		m,r8/16	2	7-8	1				
		ADD, SUB		m,r32/64	2	1	1		may mirror		
		ADC, SBB		r,r	1	1	1				
		ADC, SBB		r,i	1	1	1				
		ADC, SBB		r,m	1	1	1				
		ADC, SBB		m,r8/16	2	8	1				
		ADC, SBB		m,r32/64	2	1	1		may mirror		
		ADCX ADO	x	r,r	1	1	1		ADX		
		CMP	FSQRT		'	1	1	25	['] 10	1	
r = 1 - (R0 + (1 - R0) R = (D ⁺ nnt - N ⁻ (dd)		CMP	FLDPI, et	tc.			1		1		
diffuse; AMD "	Zen 4"	CMP	FSIN				12-60	50-200		P0 P1	
true;		CMP	FSINCOS	2			12-100	80-150		P0 P1	
+ refr)) && (depth < MAXDEPIII)	2022	INC. DEC. I	FPTAN				10-60	60-120		P0 P1	
N); L * E * diffuse:		INC DEC I	FPATAN				10-100	50-190		P0 P1	
true;		AAA AAS	FSCALE				8	11	4	P0 P1	
рертн)			FXTRAC	Т			13	12	5	P0 P1	
vive = SurvivalProbability(diffuse		DAS	FZXM1 FYL2X				10-18	40-60		P0 P1 P0 P1	
inmation - doing it properly			FYL2XP1	I			70	~170		P0 P1	
+ radiance.y + radiance.z) > 0) 88 (doct					1	13	1				والكري
true; ordfPdf = EvaluateDiffuse(L, N) * Psurvives				r8/m8	1	3	1				
<pre>factor = diffuse * INVPI; weight = Mis2(directPdf, brdfPdf);</pre>				r16/m16	2	2	2				
cosThetaOut = dot(N, L); * ((weight * cosThetaOut) / directPdf) * (radiance				r22/m22	2	2	2				
om walk - done properly, closely following Section				r64/m64	2	2	4			×.	M ////////////////////////////////////
				104/1104	2	3					
<pre>brdf = SampleDiffuse(diffuse, N, r1, r2, &R, apdition ive;</pre>				r,r		3				SI	
f; E * brdf * (dot(N, R) / pdf);		INUL		r,m	1					~7/	1. 3412
			I	r16 r16			1	1	1		

Instruction Cost

What is the 'cost' of a single instruction?

The cost of a single instruction depends on a number of factors:

- The arithmetic complexity (sqrt > add);
- Whether the operands are in register or memory;
- The size of the operand (16 / 64 bit is often slightly slower);
- Whether we need the answer immediately or not (latency);
- Whether we work on signed or unsigned integers (DIV/IDIV).

On top of that, certain instructions can be executed simultaneously.

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

at a = nt

), N);

AXDEPTH)

efl + refr)) && (depth

refl * E * diffuse; <u>= tr</u>ue;

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) * (red

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

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



ics & (depth < Modern

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

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

= * diffuse = true;

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

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

AXDEPTH)

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

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

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

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

Today's Agenda:

- The Cost of a Line of Code
- CPU Architecture: Instruction Pipeline
- Data Types and Their Cost
- Rules of Engagement



CPU Instruction Pipeline

Fetch

Decode

Execute

1.

2.

3.

4.

Instruction execution is typically divided in four phases:

- efl + refr)) && (depth < M
-), N); refl * E * diffuse;
- AXDEPTH)
- estiXtOrroingesij, esi adiance = SampleLight(&rand, I, a childnce.y@r@iaXe.z) > 0 v = true; at brdfPdf = Evaluated if \mathbf{G}
- at weight = Mis2(directPdf, brdfPd at cosThetaOut = dot(N, L); E * ((weight * cosThetaOut) / directPdf
- andom walk done properly, closely follo /ive)
- at3 brdf = SampleDiffuse(diffuse, N, r1, r2 $ext{CPI}=4$ pdf; n = E * brdf * (dot(N, R) / pdf);
- sion = true:

- Get the instruction from RAM
- Decode the byte code
- Execute the instruction
- Write result to RAM/registers Writeback





nics & (depth < ≫0000

: = inside ? 1 0 0 0 ht = nt / nc, ddn bs2t = 1.0f - nnt 0 2, N); 2)

t a = nt - nc, b = nt t Tr = 1 - (R0 + (1 - R0 r) R = (D = nnt - N = (ddn

= * diffuse; = true;

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

D, 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) &&

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 following

/ive)

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

CPU Instruction Pipeline

For each of the stages, different parts of the CPU are active. To use its transistors more efficiently, a modern processor overlaps these phases in a *pipeline*.

	i	1	2	3	4	5	6	7	8	9
	i _o	IF	ID	EX	WB					
	i ₁		IF	ID	EX	WB				
	i ₂			IF	ID	EX	WB			
	i ₃				IF	ID	EX	WB		
	i ₄					IF	ID	EX	WB	
	i ₅						IF	ID	EX	WB
_								4440		

At the same clock speed, we get four times the throughput (CPI = IPC = 1).



nics Midenthe a Marriett

z = inside } 1 = 1 ... ht = nt / nc, ddn bs2t = 1.0f - nnt = n D, N); B)

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

= * diffuse = true;

efl + refr)) && (depth < MONDEPTH

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

AXDEPTH)

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

v = true;

at brdfPdf = EvaluateDiffuse(L, N) * H at3 factor = diffuse * INVPI;

at weight = Mis2(directPdf, brdfPdf); at cosThetaOut = dot(N, L);

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

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:

CPU Instruction Pipeline

In practice, each of the pipeline phases takes several cycles to complete.

i	1	2	3	4	5	6	7	8	9
i _o	IF	ID	ID	ID	EX	WB			
i ₁		IF			ID	ID	ID	EX	WB



rics & (depth < MoDE⊨

: = inside ? 1 ; 1 ; 1 ht = nt / nc, ddn bs2t = 1.0f - nnt 0, N); ∂)

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

= * diffuse = true;

efl + refr)) && (depth < MAX

), 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) &&

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 /ive)

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

CPU Instruction Pipeline

Maximum clock speed is determined by the most complex of the four stages. For higher clock speeds, it is advantageous to increase the number of stages (thereby reducing the complexity of each individual stage).

i	1	2	3	4	5	6	7	8	9	10	11	12	13
i _o					EX	EX	EX						
i ₁		P E				EX	EX	EX					
i ₂							EX	EX	EX		i i		
i ₃								EX	EX	EX			
i ₄									EX	EX	EX		

Stages

7	PowerPC G4e
3	Cortex-A9
L0	Athlon
12	Pentium Pro/II/III, Athlon 64
L4	Core 2, Apple A7/A8
L4/19	Core i2/i3 Sandy Bridge
16	PowerPC G5, Core i*1 Nehalem
18	Bulldozer, Steamroller
20	Pentium 4
31	Pentium 4E Prescott

Super-pipelining allows higher clock speeds and thus higher throughput, but it also increases the latency of individual instructions.

nics & (depth ≪ MoxDEAT

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

at a = nt - nc, b = nt = nt at Tr = 1 - (R0 + (1 - R0 Tr) R = (D = nnt - N = (d0)

= * diffuse; = true;

efl + refr)) && (depth < NOCES

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

CPU Instruction Pipeline

FP I M

Here, one execution unit handles floats; one handles integer; one handles memory operations.

Since the execution logic is typically the most complex part, we might just as well duplicate the other parts:

Different execution units for different (classes of) instructions:







efl + refr)) && (depth <

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

AXDEPTH)

survive = SurvivalProbability(diffu lf; 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) / directPd

andom walk - done properly, closely follo /ive)

at3 brdf = SampleDiffuse(diffuse, N, r1, r2 IPC = 3 (or: ILP = 3)

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

CPU Instruction Pipeline

This leads to the *superscalar* processor, which can execute multiple instructions in the same clock cycle, assuming not all instruction require the same execution logic.

i	1	2	3	4	5	6
fp _o			EX			
int _o			EX			
m _o			EX			
fp1				EX		
int ₁				EX		
m ₁				EX		
fp ₂					EX	
int ₂					EX	
m ₂					EX	

CPU Instruction Pipeline

fics & (depth < Modes

: = inside ? 1 ; 1 ; ht = nt / nc, ddn os2t = 1.0f - nnt 2, N); >)

t a = nt - nc, b = nt t Tr = 1 - (R0 + (1 - R0 r) R = (D = nnt - N = (3

= * diffuse; = true;

efl + refr)) && (depth < MAXDEPT

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

AXDEPTH)

survive = SurvivalProbability(diffuse
estimation - doing it properly, closed)
if;
radiance = SampleLight(&rand, I, &L, &ll
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 So /ive)

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

Using a pipeline has consequences. Consider the following situation:

a = b * c; d = a + 1; y = y >> 1; z = 0x1a4;

+

	i	1	2	3	4	5	6	7	8	9	10
	i _o	IF	ID	EX	EX	WB					
	i ₁		IF	ID		EX	WB				
	i ₂			IF	ID		EX	WB			
ve	i ₃				IF	ID		EX	WB		



CPU Instruction Pipeline

fics & (depth < ≯VOCCS

:= inside ? 1 ; 1 ; ht = nt / nc, ddn hs2t = 1.0f - nnt 2, N); 2)

t a = nt - nc, b = nt t Tr = 1 - (R0 + (1 - R0 r) R = (D = nnt - N = (0

= * diffuse; = true;

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

AXDEPTH)

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

v = 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) = (r

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

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

Using a pipeline has consequences. Consider the following situation:

a = b * c; d = a + 1; y = y >> 1; z = 0x1a4;

+

i	1	2	3	4	5	6	7	8	9	1
i _o	IF	ID	EX	EX	WB					
i ₁		IF	ID			EX	WB			
i ₂			IF	ID			EX	WB		
i ₃				IF	ID			EX	WB	

We now lose one cycle!

.0



CPU Instruction Pipeline

fics & (depth < ≯∕0000

:= inside 7 1 ; ... ht = nt / nc, ddn hs2t = 1.0f - nnt 2, N); 2)

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

= * diffuse; = true;

efl + refr)) && (depth < MAXDED11

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

AXDEPTH)

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

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

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

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

Using a pipeline has consequences. Consider the following situation:

a = b * c; d = a + 1; y = y >> 1; z = 0x1a4;

	i	1	2	3	4	5	6	7	8	9	10
	i _o	IF	ID	EX	EX	WB					
	i ₁		IF	ID			EX	WB			
	i ₂			IF	ID	EX			WB		
vest	i ₃				IF	ID		EX		WB	

Out-of-order execution requires instructions to be *retired* in-order. An instruction is retired when its result is written back to memory.



CPU Instruction Pipeline

a = b * c;

+

y = y >> 1; d = a + 1;

ics & (depth < ™000000

: = inside ? | . . ht = nt / nc, ddn h52t = 1.0f - nnt), N); >)

t = nt - nc, b = 0.0t Tr = 1 - (R0 + (1 - 0.0))r) R = (D = nnt - 0.0)

= * diffuse; = true;

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

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

AXDEPTH)

survive = SurvivalProbability(diffuse estimation - doing it properly, closed H; 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 following S /ive)

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

A good compiler re-organizes your code to maximize throughput.





at a = nt

), N);

AXDEPTH)

refl * E * diffuse;

CPU Instruction Pipeline

So how do we utilize out-of-order execution:

- A compiler reorganizes code to prevent latencies
- The CPU reorganizes instructions to prevent latencies
- Feeding mixed code provides compiler and CPU with opportunities for shuffling

One little problem remains:

What if the CPU doesn't know what the next instruction is?

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

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

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



Speculative Execution

The CPU guesses what a branch will do:

if (a < b) foo(); else bar();</pre>

For this it uses *branch prediction*. Instructions at the predicted location will be executed even if it is not sure that this is correct.

A *branch misprediction* requires that the pipeline is flushed and re-populated...



e.x + radiance.y + radiance.z) > 0) && (dot)
w = 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)

survive = SurvivalProbability(diff.

radiance = SampleLight(&rand, I, &L

efl + refr)) && (depth

refl * E * diffuse; = true;

), N);

AXDEPTH)

lf;

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

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

ics & (depth < Modern

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

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

= * diffuse = true;

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

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

AXDEPTH)

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

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

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

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

Today's Agenda:

- The Cost of a Line of Code
- CPU Architecture: Instruction Pipeline
- Data Types and Their Cost
- Rules of Engagement



Data Types

), N);

= true;

AXDEPTH)

v = true;

/ive)

urvive;

sion = true:

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

lf;

Red = u4 & (255 << 16); Data types in C++ Green = u4 & (255 << 8); Blue = u4 & 255; int unsigned int 16 15 31 24 23 8 7 Size: 32 bit (4 bytes) efl + refr)) && (depth < MA) union { unsigned int u4; int s4; char s[4]; }; Access: refl * E * diffuse; unsigned char v = 100;s[1] = v;survive = SurvivalProbability(diff. u4 = (a4 ^ (255 << 8)) | (v << 8); radiance = SampleLight(&rand, I, &L, e.x + radiance.y + radiance.z) > 0) & u4 ^= 1 << 31; Altering sign bit of s4: at brdfPdf = EvaluateDiffuse(L, N) at3 factor = diffuse * INVPI; (*note: -1 = 0xfffffff*) at weight = Mis2(directPdf, brdfPdf) at cosThetaOut = dot(N, L); E * ((weight * cosThetaOut) / directPdf) andom walk - done properly, closely follow at3 brdf = SampleDiffuse(diffuse, N, r1, r2, &R, &p

Data Types

Data types in C++

float

Value:



* diffuse; = true;

efl + refr)) && (depth <

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

AXDEPTH)

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

v = 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 see /ive)

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

Size: 32 bit (4 bytes)

Exponent:8 bit; $-127 \dots 128$ Mantissa:23 bit;0 ... 2^{23} -1

sign * mantissa * 2^exponent

Exercise: write a function that replaces array $a = \{0.5, 0.25, 0.125, 0.0625, ...\}$.



Data Types

nics & (depth < ∧∧∞s

nt a = nt - nc, b = nt nt Tr = 1 - (R0 + (1 - R6 r) R = (D ⁼ nnt - N ⁼ (dd

* diffuse; = true;

efl + refr)) && (depth < MAXDEPII

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

AXDEPTH)

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

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 Sm /ive)

; at3 brdf = SampleDiffuse(diffuse, N, r1, r2*: More on <u>http://www.catb.org/esr/structure-packing</u> urvive;

pdf; n = E * brdf * (dot(N, R) / pdf); sion = true: double char, unsigned char short, unsigned short LONG LONG LONG, __int64 bool

Data types in C++

Padding*:

};

struct Test

unsigned int u; bool flag;

sizeof(Test) is 8

64 bit (8 bytes) 8 bit 16 bit 32 bit (same as int) 64 bit 8 bit (!)

```
struct Test2
{
    double d;
    bool flag;
};
// sizeof( Test2 ) is 16
```



Data Types

nics & (depth < 2000

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

at a = nt - nc, b = 00 at Tr = 1 - (R0 + (1 - 00 fr) R = (D - nnt - 0 - 00

= * diffuse; = true;

efl + refr)) && (depth < MOCCEPIU

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

AXDEPTH)

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

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

st3 brdf = SampleDiffuse(diffuse, N, r1, r2, 8R, 8pdf urvive; pdf; n = E * brdf * (dot(N, R) / pdf); sion = true:

Data types in C++ - Conversions

Explicit:

float fpi = 3.141593; int pi = (int)(1024.0f * fpi);

bitmap[i].g *= 0.5f;

bitmap[i].b *= 0.5f;

Implicit:

struct Color { unsigned char a, r, g, b; }; Color bitmap[640 * 480]; for(int i = 0; i < 640 * 480; i++) { bitmap[i].r *= 0.5f;

// bitmap[i].r *= 0.5f;

novzx	eax,byte ptr [ecx-1]
nov	dword ptr [ebp-4],eax
Fild	dword ptr [ebp-4]
fnstcw	word ptr [ebp-2]
novzx	eax,word ptr [ebp-2]
or	eax,0C00h
nov	dword ptr [ebp-8],eax
Fmul	st,st(1)
Fldcw	word ptr [ebp-8]
fistp	dword ptr [ebp-8]
novzx	eax,byte ptr [ebp-8]
nov	byte ptr [ecx-1],al



Data Types

tics & (depth < 2000

: = inside } 1 ; . . . ht = nt / nc, ddn bs2t = 1.0f - nnt D, N); ∂)

nt a = nt - nc, b = nt nt Tr = 1 - (R0 + (1 - R0 Tr) R = (D - nnt - N - (0

= * diffuse; = true;

efl + refr)) && (depth < MAXDEPTH

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

AXDEPTH)

survive = SurvivalProbability(diffuse)
estimation - doing it properly
ff;
radiance = SampleLight(&rand, I, &L, &I
2.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 followinn∫o ∕ive)

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

Data types in C++ - Conversions

Explicit:

float fpi = 3.141593; int pi = (int)(1024.0f * fpi);

Avoiding conversion:

struct Color { unsigned char a, r, g, b; }; Color bitmap[640 * 480]; for(int i = 0; i < 640 * 480; i++)</pre>

bitmap[i].r >>= 1; bitmap[i].g >>= 1; bitmap[i].b >>= 1; // bitmap[i].r >>= 1; shr byte ptr [eax-1],1 // bitmap[i].g >>= 1; shr byte ptr [eax],1 // bitmap[i].b >>= 1; shr byte ptr [eax+1],1



Data Types

nics & (depth < ₩000

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

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

= * diffuse = true;

efl + refr)) && (depth < MAXDEPIN

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

AXDEPTH)

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

v = true; at brdfPdf = EvaluateDiffuse(L, N) * Psurvi 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, &pdF) urvive; pdf; n = E * brdf * (dot(N, R) / pdf); sion = true:

Data types in C++ - Conversions

Explicit:

float fpi = 3.141593; int pi = (int)(1024.0f * fpi);

Avoiding conversion (2):

struct Color { union { struct { unsigned char a, r, g, b; }; int argb; }; }; Color bitmap[640 * 480]; for(int i = 0; i < 640 * 480; i++)</pre>

bitmap[i].argb = (bitmap[i].argb >> 1) & 0x7f7f7f;



ics & (depth < Modern

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

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

= * diffuse = true;

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

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

AXDEPTH)

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

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

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

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

Today's Agenda:

- The Cost of a Line of Code
- CPU Architecture: Instruction Pipeline
- Data Types and Their Cost
- Rules of Engagement



nics & (depth < Motos)

: = inside ? 1 ()) ht = nt / nc, ddn bs2t = 1.0f - nnt) 2, N); 3)

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

* 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, &light) e.x + radiance.y + radiance.z) > 0) && (doing)

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) * (rectPdf)

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:

Common Opportunities in Low-level Optimization

RULE 1: Avoid Costly Operations

- Replace multiplications by bitshifts, when possible
- Replace divisions by (reciprocal) multiplications
- Avoid sin, cos, sqrt



Common Opportunities in Low-level Optimization

Adapt previous results (interpolation, reprojection, ...)

RULE 2: Precalculate

Loop hoisting

Lookup tables

Reuse (partial) results

- : = inside } 1 1 1 ht = nt / nc, ddn bs2t = 1.0f - n∩t 0, N); 3)
- at a = nt nc, b = nt at Tr = 1 - (R0 + (1 - R0) fr) R = (D = nnt - N = (00)
- = * diffuse; = true;
- -:fl + refr)) && (depth < MAXDE
- D, N); refl * E * diffuse; = true;

AXDEPTH)

- survive = SurvivalProbability(diffuse
 estimation doing it properly closed
 if;
 radiance = SampleLight(&rand, I, &L, &light()
 e.x + radiance.y + radiance.z) > 0) && (doing)
- 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) * (rad
- andom walk done properly, closely following Sec. /ive)
- ; at3 brdf = SampleDiffuse(diffuse, N, r1, r2, &R, &pdf urvive; pdf; n = E * brdf * (dot(N, R) / pdf); sion = true:



Common Opportunities in Low-level Optimization

RULE 3: Pick the Right Data Type

- Avoid byte, short, double
- Use each data type as a 32/64 bit container that can be used at will
- Avoid conversions, especially to/from float
- Blend integer and float computations
- Combine calculations on small data using larger data

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

= * diffuse; = true;

efl + refr)) && (depth < ≀

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

AXDEPTH)

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

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

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



Common Opportunities in Low-level Optimization

- efl + refr)) && (depth < MA
-), N); refl * E * diffuse; = true;
- AXDEPTH)
- survive = SurvivalProbability(diff. lf; 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, & urvive; pdf; n = E * brdf * (dot(N, R) / pdf); sion = true:

RULE 4: Avoid Conditional Branches

- if, while, ?, MIN/MAX
- Try to split loops with conditional paths into multiple unconditional loops
- Use lookup tables to prevent conditional code
- Use loop unrolling
- If all else fails: make conditional branches predictable



Rules of Engagement

char c = 'p';

int position = -1;

if (a[t] == c)

position = t;

Common Opportunities in Low-level Optimization

RULE 5: Early Out

char a[] = "abcdfghijklmnopqrstuvwxyz";

for (int t = 0; t < strlen(a); t++)</pre>

```
t = nt / nc, ddm
s2t = 1.0f - nnt
, N );
)
```

```
r) R = (D * nnt -
```

```
: * diffuse;
= true;
```

```
∘
efl + refr)) && (depth < MAXDED)
```

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

```
AXDEPTH)
```

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

```
v = true;
at brdfPdf = EvaluateDiffuse( L, N ) * Psurvio
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)

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

```
char a[] = "abcdfghijklmnopqrstuvwxyz";
char c = 'p';
int position = -1, len = strlen( a );
for ( int t = 0; t < len; t++ )
{
    if (a[t] == c)
    {
        position = t;
        break;
    }
}
```



Common Opportunities in Low-level Optimization

RULE 6: Use the Power of Two

- A multiplication / division by a power of two is a (cheap) bitshift
- A 2D array lookup is a multiplication too make 'width' a power of 2
- Dividing a circle in 256 or 512 works just as well as 360 (but it's faster)
 - Bitmasking (for free modulo) requires powers of 2

1-2-4-8-16-32-64-128-256-512-1024-2048-4096-8192-16384-32768-65536

Be fluent with powers of 2 (up to 2^{16}); learn to go back and forth for these: $2^{9} = 512 = 2^{9}$. Practice counting from 0..31 on one hand in binary.

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

at a = nt

), N);

= true;

AXDEPTH)

efl + refr)) && (depth

survive = SurvivalProbability(d

radiance = SampleLight(&rand, I

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

refl * E * diffuse;

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

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



Common Opportunities in Low-level Optimization

RULE 7: Do Things Simultaneously

- - An integer holds four bytes; use these for instruction level parallelism

efl + refr)) && (depth < M

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

AXDEPTH)

survive = SurvivalProbability(diff. lf; 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 followi /ive)

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

Use those cores

- More on this later.



Common Opportunities in Low-level Optimization

- 1. Avoid Costly Operations
- 2. Precalculate
- 3. Pick the Right Data Type
- 4. Avoid Conditional Branches
- 5. Early Out
- 6. Use the Power of Two
- 7. Do Things Simultaneously
- efl + refr)) && (depth < MAXE D, N); refl * E * diffuse;
- = true;
- AXDEPTH)
- survive = SurvivalProbability(diffuse)
 estimation doing it properly, closed
 if;
 radiance = SampleLight(&rand, I, &L, &light
 e.x + radiance.y + radiance.z) > 0) && (dot)
- 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) * (rad
- andom walk done properly, closely following Sec. /ive)
- ; at3 brdf = SampleDiffuse(diffuse, N, r1, r2, &R, &pdf urvive; pdf; n = E * brdf * (dot(N, R) / pdf); sion = true:



ics & (depth < Modern

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

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

= * diffuse = true;

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

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

AXDEPTH)

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

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

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

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

Today's Agenda:

- The Cost of a Line of Code
- CPU Architecture: Instruction Pipeline
- Data Types and Their Cost
- Rules of Engagement



tics & (depth < →0005)

: = inside ? 1 | | | | ht = nt / nc, ddn | | os2t = 1.0f - nnt | nn D, N); B)

at a = nt - nc, b = nt at Tr = 1 - (R0 + (1 - R0 Tr) R = (D = nnt - N = (100

= * diffuse; = true;

efl + refr)) && (depth < MOXDEPT

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

AXDEPTH)

survive = SurvivalProbability(diffuse estimation - doing it properly closed If; radiance = SampleLight(&rand, I, &L, &Light) 2.x + radiance.y + radiance.z) > 0) && (dot)

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

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 "Low Level"

next lecture: "Profiling"

