rics & (depth < MOOSE

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

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

= * diffuse = true;

efl + refr)) && (depth < MAXDED1

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

AXDEPTH)

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

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 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 4: "Caching (1)"

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 - RC) fr) R = (D = nnt - N = (dd)

= * diffuse = true;

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

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) && (dott)

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) (1860)

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

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

Today's Agenda:

- The Problem with Memory
- Cache Architectures



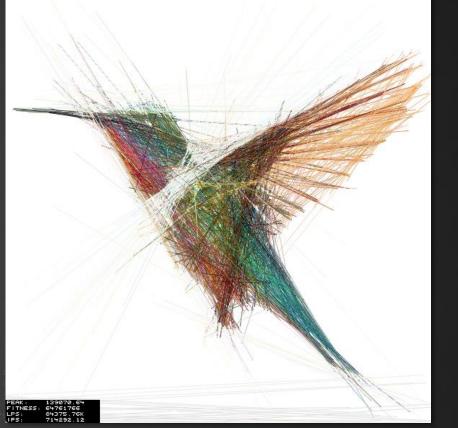
D, N); refl * E * diffuse; = true;

AXDEPTH)

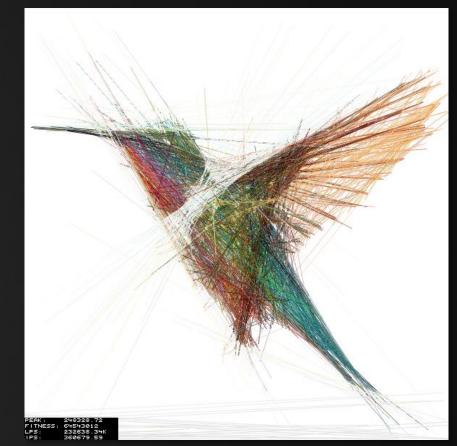
survive = SurvivalProbability(diffus estimation - doing it properly if; radiance = SampleLight(&rand, I, &L, 2.x + radiance.y + radiance.z) > 0) &

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

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



1 3 9 0 7 0



2 4 8 3 2 8

× 1.7856



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 - RC) fr) R = (D = nnt - N = (dd)

= * diffuse = true;

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

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) && (dott)

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) (1860)

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

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

Today's Agenda:

- The Problem with Memory
- Cache Architectures



Introduction

nics & (depth < Motosi

c = inside ? 1 : 5 ; ht = nt / nc, ddn ss2t = 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 < MAX

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

AXDEPTH)

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

v = 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 fo /ive)

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

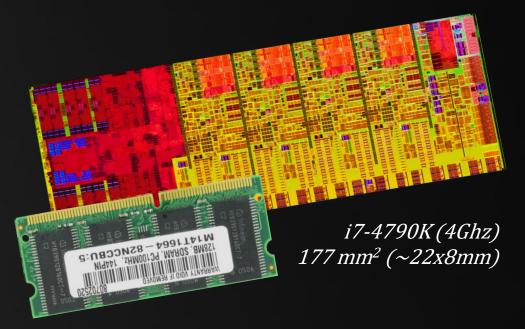
Let's assume our CPU runs at 4Ghz. What is the maximum physical distance between memory and CPU if we want to retrieve data every cycle?

Speed of light (vacuum): 299,792,458 m/s Per cycle: ~0.075 m

→ \sim <u>3.75cm</u> back and forth.

In other words: we cannot physically query RAM fast enough to keep a CPU running at full speed.

Note: signals in electronics do *not* propagate at the speed of light: https://en.wikipedia.org/wiki/Velocity_factor#Typical_velocity_factors ²TL;DR: It's between 0.6c .. 0.99c.







Introduction

sics & (depth < Monos-

= inside / 1 ht = nt / nc, ddn ss2t = 1.0f - nnt 2, N); 8)

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

= * diffuse; = true;

efl + refr)) && (depth < MAXDEPTH

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

(AXDEPTH)

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

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 Soul /ive)

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

Feeding the Beast

Sadly, we can't just divide by the physical distance between CPU and RAM to get the cycles required to query memory.

Factors include (stats for DDR4-3200/PC4-25600):

- RAM runs at a much lower clock speed than the CPU
 - 25600 here means: theoretical *bandwidth* in MB/s
 - 3200 is the number of *transfers* per second (1 transfer=64bit)
 - We get two transfers per cycle, so actual I/O clock speed is 1600Mhz
 - DRAM cell array clock is $\sim 1/4$ th of that: 400Mhz.

Latency between query and response: 20-24 cycles.





Introduction

tics & ¢depth < ™005511

t = ntside i ot = nt / nc, ddm = i os2t = 1.0f - nmt i 0, N); 8)

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

: * diffuse; = true;

efl + refr)) && (depth < MAX

D, N); refl * E = true;

AXDEPTH)

survive = estimati if:

radiance = e.x + radia

w = true; at brdfPdf at3 factor at weight =

at cosTheta E * ((weig andom walk vive)

; at3 brdf urvive; pdf;

pur; = E * brdf * (dot(N, R) / pdf) tion = true;

BL

Feeding the Beast

M

BL

WL

Ma

Sadly, we can't just divide by the physical distance between CPU and RAM to get the cycles required to query memory.

Factors include (stats for DDR4-3200/PC4-25600):

• Latency between query and response: 20-24 cycles.

SRAM:

- Maintains data as long as V_{dd} is powered (no refresh).
- Bit available on *BL* and *BL* as soon as *WL* is raised (fast).
- Six transistors per bit (\$).
- Continuous power (\$\$\$).





Introduction

tics & (depth < MAXDERT

z = inside / 1 ht = nt / nc, ddm os2t = 1.0f - nnt -D, N); 3)

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

= * diffuse; = true;

. efl + refr)) && (depth < 100000

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

AXDEPTH)

survive = estimatio f; radiance =

e.x + radia w = true; at brdfPdf at3 factor

at weight = at cosTheta E * ((weig andom walk

vive)

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

at3 brdf = SampleDiffuse(diffu

Feeding the Beast

 $DL \xrightarrow{=} M = C$

Sadly, we can't just divide by the physical distance between CPU and RAM to get the cycles required to query memory.

Factors include (stats for DDR4-3200/PC4-25600):

Latency between query and response: 20-24 cycles.

DRAM:

- Stores state in capacitor C.
- Reading: raise AL, see if there is current flowing.
- Needs rewrite.
- Draining takes time.
- Slower but cheap.
- Needs refresh.





DL ·

Introduction

ics (depth < MAXDERT

z = inside / 1 ht = nt / nc, ddh os2t = 1.0f - nnt / n O, N); 3)

nt a = nt - nc, b = nt - n nt Tr = 1 - (R0 + (1 - R0 T) R = (D [#] nnt - N [#] (dd

= * diffuse; = true;

efl + refr)) && (depth < NOODE

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

AXDEPTH)

survive = estimatio f; radiance =

e.x + radia w = true; at brdfPdf at3 factor

at weight

at cosThetaO E * ((weigh andom walk vive)

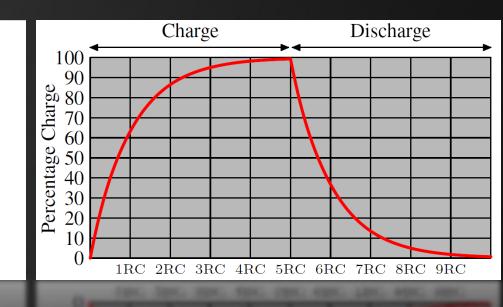
; at3 brdf = SampleDiffuse(difi urvive; pdf;

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

Sadly, we can't just divide by the physical distance between CPU and RAM to get the cycles required to query memory.

Factors include (stats for DDR4-3200/PC4-25600):

Latency between query and response: 20-24 cycles.







Introduction

nics & (depth < MAXDER

_ = inside + 1 ht = nt / nc, ddn ps2t = 1.0f - nnt D, N); B)

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

= * diffuse; = true;

• •fl + refr)) && (depth < MADE

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

AXDEPTH)

survive = SurvivalProbability
estimation - doing it proper
if;
adiance = SampleLight(&rand
e.x + radiance.y + radiance.z

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

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

; t3 brdf = SampleDiffuse(diffuse, N, r1 .rvive; .pdf; n = E * brdf * (dot(N, R) / pdf); sion = true: Feeding the Beast

Sadly, we can't just divide by the physical distance between CPU and RAM to get the cycles required to query memory.

Additional delays may occur when:

- Other devices than the CPU access RAM;
 - DRAM must be refreshed every 64ms due to leakage.

For a processor running at 2.66GHz, latency is roughly 110-140 CPU cycles.

Details in: "What Every Programmer Should Know About Memory", Chapter 2.





Introduction

nics & (depth < MODE⊂

c = inside f | ht = nt / nc, ddn os2t = 1.0f - nnt O, N); 8)

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

= * 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, clister r.x + radiance.y + radiance.z) > 0) && (doit)

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) (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:

Feeding the Beast

"We cannot physically query RAM fast enough to keep a CPU running at full speed."

How do we overcome this?

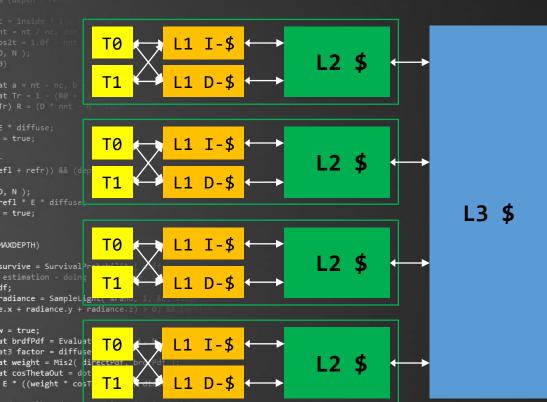
We keep a copy of frequently used data in fast SRAM memory, close to the CPU: the *cache*.



11

Introduction

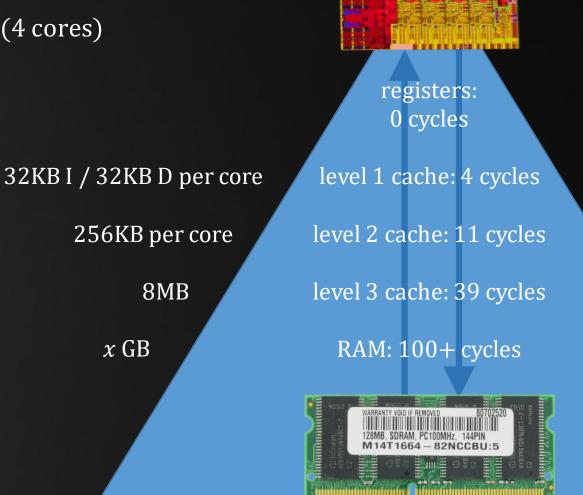
The Memory Hierarchy – Core i7-9xx (4 cores)



andom walk - done broperly, closely followin /ive)

, t33 brdf = SampleDiffuse(diffuse, N, r1, r2, &R, &pdf) µrvive; pdf; n = E * brdf * (dot(N, R) / pdf);

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





Introduction

Caches and Optimization

nics & (depth < MODE⊡

: = inside } 1 | | | | ht = nt / nc, ddn | | bs2t = 1.0f - nnt | n1 D, N); 3)

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

= * 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) * 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 Soci /ive)

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

Considering the cost of RAM vs L1\$ access, it is clear that the cache is an important factor in code optimization:

- Fast code communicates mostly with the caches
- We still need to get data into the caches
- But ideally, only once.

Therefore:

- The working set must be small;
- Or we must maximize *data locality*.



13

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 - RC) fr) R = (D = nnt - N = (dd)

= * diffuse = true;

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

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) && (dott)

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) (1860)

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

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

Today's Agenda:

- The Problem with Memory
- Cache Architectures



};

Architectures

nics & (depth < ⊅00000

: = inside / l | | | ht = nt / nc, ddh ss2t = 1.0f - nnt 2, N); 2)

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

* diffuse; = true;

efl + refr)) && (depth < MAXDEP1

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

AXDEPTH)

survive = SurvivalProbability(diffus
estimation - doing it properly
f;
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 foll /ive)

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

Cache Architecture

The simplest caching scheme is the *fully associative cache*.

struct CacheLine

uint address; // 32-bit for 4G
uchar data;
bool valid;

CacheLine cache[256];

This cache holds 256 bytes.

This cache has an administrative overhead of 33x256 bits = ~ 8 KB.

address	data	valid
0x0000000	OxFF	0
0x0000000	OxFF	0

Notes on this layout:

- We will rarely need 1 byte at a time
- So, we switch to 32bit values
- We will never read those at addresses that are not a multiple of 4
- So, we drop 2 bits from the address field (and: we rename it to *tag*).



};

Architectures

nics & (depth < Mannes

: = inside / l | | | ht = nt / nc, ddh ss2t = 1.0f - nnt 2, N); 2)

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

* diffuse; = true;

efl + refr)) && (depth < MAXDEPT

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

AXDEPTH)

survive = SurvivalProbability(diffus estimation - doing it properly if; radiance = SampleLight(&rand, I, & .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 f /ive)

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

Cache Architecture

The simplest caching scheme is the *fully associative cache*.

struct CacheLine

uint tag; // 30 bit for 4G uint data; bool valid, dirty;

CacheLine cache[64];

This cache holds 64 dwords (256 bytes).

This time, the administrative overhead is 32 * 64 = 2KB.

tag	data	valid	dirty
0x0000000	OxFFFFFFF	0	0
0x0000000	OxFFFFFFF	0	0
0x0000000	OxFFFFFFF	0	0
0x00000000	OxFFFFFFF	0	0
0x0000000	OxFFFFFFF	0	0
	•••		
0x0000000	OxFFFFFFF	0	0



};

Architectures

at a = nt ·

efl + refr)) && (depth (

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

(AXDEPTH)

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

v = true; at brdfPdf = EvaluateDiffuse(L 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; 1 = E * brdf * (dot(N, R) / pdf); sion = true:

Cache Architecture

The simplest caching scheme is the fully associative cache.

struct CacheLine

// 30 bit for 4G uint tag; uint data; bool valid, dirty; CacheLine cache[64];

This cache holds 64 dwords (256 bytes).

2 31 1 0 offs tag address for (int i = 0; i < 64; i++)</pre> if (cache[i].valid) if (cache[i].tag == tag) return cache[i].data[offs]; uint d = RAM[tag].data; // cache miss WriteToCache(tag, d); return d[offs];

Single-byte read operation:



};

Architectures

at a = nt

efl + refr)) && (depth

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

AXDEPTH)

```
survive = SurvivalProbability
radiance = SampleLight( &rand,
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) / directPdf)

andom walk - done properly, closely follo /ive)

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

Cache Architecture

The simplest caching scheme is the fully associative cache.

struct CacheLine

// 30 bit for 4G uint tag; uint data; bool valid, dirty; CacheLine cache[64];

This cache holds 64 dwords (256 bytes).

One problem remains... We store one byte, but the slot stores 4. What should we do with the other 3?

Single-byte write operation:

```
for ( int i = 0; i < 64; i++ )</pre>
   if (cache[i].valid)
      if (cache[i].tag == a)
         cache[i].data[offs] = d;
         cache[i].dirty = true;
         return;
```

```
for ( int i = 0; i < 64; i++ )</pre>
   if (!cache[i].valid)
      cache[i].tag = a;
      cache[i].data[offs] = d;
      cache[i].valid|dirty = true;
      return;
```

i = BestSlotToOverwrite();

if (cache[i].dirty) SaveToRam(i); cache[i].tag = a;

cache[i].data[offs] = d; cache[i].valid[dirty = true;



Architectures



c = inside / 1 ht = nt / nc, ddn ss2t = 1.0f - nnt nn D, N); B)

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

= * diffuse; = true;

-:fl + refr)) && (depth < 1900

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

AXDEPTH)

survive = SurvivalProbability(diffuestimation - doing it properly, if; radiance = SampleLight(&rand, I, &: 2.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) = (ra

andom walk - done properly, closely following Soli /ive) *. ba

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

BestSlotToOverwrite() ?

The best slot to overwrite is the one that will not be needed for the longest amount of time. This is known as Bélády's algorithm, or the *clairvoyant* algorithm.

Alternatively, we can use:

- LRU: least recently used
- MRU: most recently used
- Random Replacement
- LFU: Least frequently used

• …

AMD and Intel use 'pseudo-LRU' (until Ivy Bridge; after that, things got complex*).



In case thit isn't obvious: this is a hypothetical algorithm; the best option if we actually had a crystal orb.



Architectures

at a = n at Tr = 1

), N);

AXDEPTH)

v = true;

/ive)

at weight = Mis2(directPdf, brdfP at cosThetaOut = dot(N, L); E * ((weight * cosThetaOut) / direc

andom walk - done properly, closely

f:

efl + refr)) && (de

refl * E * diffuse;

The Problem with Being Fully Associative

Read / Write using a fully associative cache is O(N): we need to scan each entry. This is not practical for anything beyond $16 \sim 32$ entries.



An alternative scheme is the *direct mapped cache*.

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



};

Architectures

nics & (depth < NOCCS

: = inside } 1 ; ; ; ht = nt / nc, ddn bs2t = 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 < MAXE

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)

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:

Direct Mapped Cache

struct CacheLine

uint tag; // 24 bit for 4G uint data; bool dirty, valid;

CacheLine cache[64];

This cache again holds 256 bytes.

In a direct mapped cache, each address can only be stored in a single cache line.

Read/write access is therefore O(1).

For a cache consisting of 64 cache lines:

31	8	7 2	1 0
	tag	slot	offs
address			

- Bit 0 and 1 still determine the offset within a slot;
- 6 bits are used to determine which slot to use;
- The remaining 24 bits form the tag.



31

Architectures

Direct Mapped Cache

nics & (depth < Modeen

c = inside ? 1 ()) nt = nt / nc, ddn ps2t = 1.0f - nnt (), N); »)

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

= * diffuse; = true;

efl + refr)) && (depth < MAXDEPT

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

AXDEPTH)

survive = SurvivalProbability(diffuse estimation - doing it properly, closed H; radiance = SampleLight(&rand, I, &L, \$11 e.x + radiance.y + radiance.z) > 0) && (d)

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:

M+N M+N-1 7 32-bit address

Ν

N-1

0

In general:

 $N = \log_2(cache \ line \ width)$ $M = \log_2(number \ of \ slots \ in \ the \ cache)$

Bits 0..N-1 are used as offset in a cache line; Bits N..M-1 are used as slot index;

Bits M..31 are used as tag.

22

Architectures



: = inside } 1 | 1 | 1 ht = nt / nc, ddn | 1 ss2t = 1.0f - nnt | n), N); ≫)

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

= * diffuse; = true;

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

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

(AXDEPTH)

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

v = 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) * (r)

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

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

The Problem with Direct Mapping

In this type of cache, each address maps to a single cache line, leading to O(1) access time. On the other hand, a single cache line 'represents' multiple memory addresses.

This leads to a number of issues:

- A program may use two variables that occupy the same cache line, resulting in frequent cache misses (collisions);
- A program may heavily use one part of the cache, and underutilize another.



0000000		
0000004		
0000008		
000000C	\checkmark	
0000010		
0000014		1
0000018		cache
000001C		
0000020	/ /	
0000024		
0000028		
000002C		
0000030	/	
0000034		
0000038		
000003C		
RAM		



Architectures

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

AXDEPTH)

survive = SurvivalProbability(diff) lf; radiance = SampleLight(&rand, I, & e.x + radiance.y + radiance.z) > 31

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

/ive)

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

sion = true:

N-Way Set Associative Cache

struct CacheLine

uint tag; uint data; bool valid, dirty; }; CacheLine cache[16][4];

This cache again holds 256 bytes.

4

address

tag

3

set

2 1

0

offs

set

In an N-way set associative cache, we use N slots (cache lines) per set.

		slot 0	slot 1	slot 2	slot 3
t	0000				
	0001				
	0002				
	0003				
	0004				
	0005				
	0006				
	0007				
	8000				
	0009				
	0010				
	0011				
	0012				
	0013				
	0014				1405.5%
	0015				

};

Architectures

at a = nt

efl + refr)) && (depth

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

(AXDEPTH)

survive = SurvivalProbability(diff f: radiance = SampleLight(&rand, I e.x + radiance.y + radiance.z) 31 v = true: at brdfPdf = EvaluateDiffuse

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

andom walk - done properly /ive)

urvive; pdf;

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

N-Way Set Associative Cache

struct CacheLine

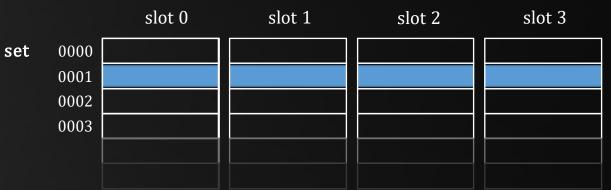
uint tag; // 28 bit for 4G uint data; bool valid, dirty;

CacheLine cache[16][4];

This cache again holds 256 bytes.



In an N-way set associative cache, we use N slots (cache lines) per set.



When reading / writing data, we check each of the N slots that may contain the data.

Example: Address 0x00FF1004

Offset: lowest 2 bits \rightarrow 0. Set: next 4 bits \rightarrow 1. Tag: remaining bits.



Architectures

Caching Architectures

The Intel i7 processors use three on-die caches:

L1: 32KB 4-way set associative instruction cache + 32KB 8-way data cache per core L2: 256KB 8-way set associative cache per core L3: 2MB x cores global 16-way set associative cache.

The AMD Phenom also uses three on-die caches:

L1: 64KB 2-way set associative (32+32) per core L2: 512KB 16-way set associative per core L3: 1MB x cores global 48-way set associative cache.

Both AMD and Intel currently use 64 byte cache lines.

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

at a = nt

), N);

= true;

AXDEPTH)

v = true;

f:

efl + refr)) && (depth <

survive = SurvivalProbability(d:

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

refl * E * diffuse;

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:



Architectures

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

AXDEPTH)

survive = SurvivalProbability(diff lf; radiance = SampleLight(&r e.x + radiance.y +

v = true; at brdfPdf = Evalua at3 factor = diffus at weight = Mis2(d at cosThetaOut = do E * ((weight * cos]

/ive)

at<u>3</u> brdf = SampleDif urvive; pdf; n = E * brdf * (dot(N, R) / pdf);

sion = true:

32KB, 4-Way Set Associative Cache

struct CacheLine

uint tag; // 19 bit for 4G uchar data[64]; bool valid, dirty;

}; CacheLine cache[128][4];

This cache holds 32768 bytes in 512 cachelines, organized in 128 sets of 4 cachelines.

radiance.z) > 0) 28 (00) 31 HteDiffuse(13	12	6	5	0
HEDINGS() HirectPdf, b bt(N, L); ThetaOut) /	tag	set		off	S
properly, cl <u>esely following same</u>					
ffuse(diffuse, N, r1, r2, 8R, Spdf	addre	ess			

slot 0	slot 1	slot 2	slot 3
			siller.
			25/1.301

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 - RC) fr) R = (D = nnt - N = (dd)

= * diffuse = true;

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

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) && (dott)

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) (1860)

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

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

Today's Agenda:

- The Problem with Memory
- Cache Architectures



rics & (depth < ™0060

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

= * diffuse = true;

efl + refr)) && (depth < MODEPTH

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

AXDEPTH)

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

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:

/INFOMOV/

END of "Caching (1)"

next lecture: "SIMD (1)"

