rics & (depth < Modes

: = inside ? 1 + 1 . 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
if;
radiance = SampleLight(&rand, I, &L, &liet)

e.x + radiance.y + radiance.z) > 0) 88 (doctor)

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:

/INFOMOV/ Optimization & Vectorization

J. Bikker - Sep-Nov 2019 - Lecture 3: "Caching (1)"

Welcome!



tics & (depth < Moder)

: = inside ? | ht = nt / nc, ddn os2t = 1.0f - nnt ? 0, N); 8)

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

= * diffuse = true;

efl + refr)) && (depth < MAXDEPID

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

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) * (Psd);

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 < NOOS

c = inside ? 1 : . . ht = nt / nc, ddn ss2t = 1.0f - nmt 2, N); 3)

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

= * diffuse; = true;

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

), 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) * Psurve 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:

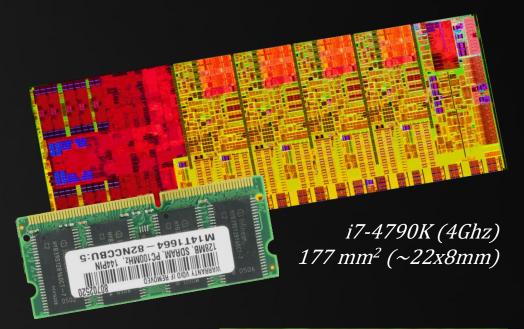
Feeding the Beast

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.







Introduction

nics & (depth < Moore

: = inside / 1 (1 / ht = nt / nc, ddh os2t = 1.0f - nnt (), N); »)

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

= * diffuse; = true;

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

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

AXDEPTH)

survive = SurvivalProbability(diffuse estimation - doing it properly, content 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 sour /ive)

; t3 brdf = SampleDiffuse(diffuse, N, r1, r2, &R, %pdf) 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 ~1/4th of that: 400Mhz.

Latency between query and response: 20-24 cycles.





Introduction

efl + refr)) && (dept

), N); refl * E true:

AXDEPTH)

survive : f:

adiance = e.x + radia

v = true: at3 factor at weight

at cosTheta andom walk /ive)

at3 brdf urvive; pdf;

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

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

es.

at a = nt - nc, b = nt at Tr = 1 - (R0 + (1 - R6) Fr) R = (D * nnt - N * (33)

= * diffuse; = true;

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

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

AXDEPTH)

survive = estimatio f; adiance =

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

at cosTheta E * ((weig andom walk

/ive)

at weight

; at3 brdf = SampleDiffuse(diffu ırvive;

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

Feeding the Beast

 $DL \xrightarrow{\overline{M}} C$

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 < Madem

: = inside ? 1 = 1 d nt = nt / nc, ddn = 0 os2t = 1.0f - nnt = n), N); ∂)

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

= * diffuse; = true;

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

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

(AXDEPTH)

survive = estimatio f; adiance =

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

at weight = at cosTheta(E * ((weigh andom walk · vive)

; at3 brdf = SampleDiffuse(diff ırvive; pdf;

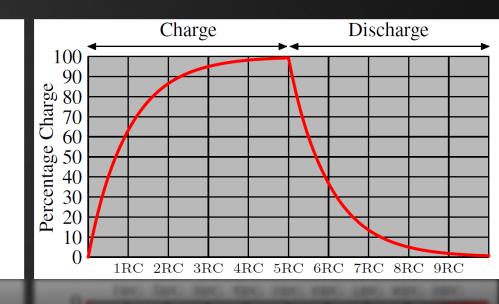
n = E * brdf * (dot(N, R) / pdf); sion = true:

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 < Moore

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

nt a = nt - nc, b = nt nt 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
estimation - doing it proper
if;
adiance = SampleLight(&rand
a.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 Sea /ive)

; at3 brdf = SampleDiffuse(diffuse, N, rl 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.

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

hics ≰j(depth < NOCCS

c = inside ? 1 : . . ht = nt / nc, ddn ss2t = 1.0f - nnt 2, N); 3)

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

= * diffuse; = true;

• efl + refr)) && (depth < MAXDEPT

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) && (doit)

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)

; t3 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."

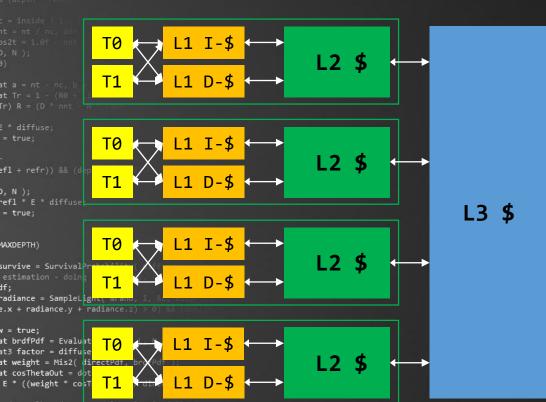
How do we overcome this?

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



Introduction

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



andom walk - done properly, closely fellen: /**ive)**

sion = true

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

registers: 0 cycles 32KB I / 32KB D per core level 1 cache: 4 cycles level 2 cache: 11 cycles 256KB per core level 3 cache: 39 cycles 8MB RAM: 100+ cycles x GB**WHIMIMIN** M14T1664-82NCCBU:5



Introduction

Caches and Optimization

tics ≰ (depth < Moc08

: = inside ? 1 1 1 3 ht = nt / nc, ddn - 3 552t = 1.0f - nmt - m 2, N); 3)

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

= * diffuse; = true;

efl + refr)) && (depth < NACESS

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

AXDEPTH)

v = true; at brdfPdf = EvaluateDiffuse(L, N) * Peu 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, dodf 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

tics & (depth < Moder)

: = inside ? | ht = nt / nc, ddn os2t = 1.0f - nnt ? 0, N); 8)

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

= * diffuse = true;

efl + refr)) && (depth < MAXDEPID

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

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) * (Psd);

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 < Modes

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

at a = nt - nc, b = nt at Tr = 1 - (R0 + (1 - R0 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, close if; radiance = SampleLight(&rand, I, &L, e.x + radiance.y + radiance.z) > 0) \$28

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

; t3 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.

address	data	valid
0x0000000	OxFF	0
0x0000000	OxFF	0
0x0000000	OxFF	0
0x00000000	OxFF	0
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 rarely read those at odd addresses
- So, we drop 2 bits from the address field.



};

Architectures

nics ≹j(depth < Notos

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

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

= * diffuse; = true;

-:fl + 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) 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 Soul /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).

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



};

Architectures

rics & (depth < Monos

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

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

= * diffuse; = true;

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

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

AXDEPTH)

survive = SurvivalProbability(diffue estimation - doing it properly, if; radiance = SampleLight(&rand, I, &L,) e.x + radiance.y + radiance.z) 0 88

```
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 Soci /ive)

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

Cache Architecture

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

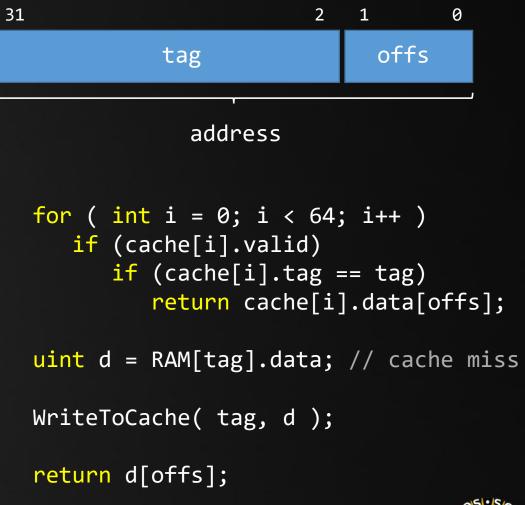
struct CacheLine

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

CacheLine cache[64];

This cache holds 64 dwords (256 bytes).

Single-byte read operation:





};

Architectures

nics ≩ (depth < Modes

: = 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 Fr) R = (D = nnt - N - (00)

= * diffuse; = true;

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

), 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(at3 factor = diffuse * INVPI;

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

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

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

; t3 brdf = SampleDiffuse(diffuse, N, r1, r2, &R, apd 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).

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

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

i = BestSlotToOverwrite();

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

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



Architectures



: = inside } 1 a 1.0 ht = nt / nc, ddn 000 ss2t = 1.0f - nnt 00 2, N); 8)

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

= * diffuse; = true;

-:fl + refr)) && (depth < 1000

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

AXDEPTH)

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

; at3 brdf = SampleDiffuse(diffuse, N, r1, r2, &R, dodf) 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

• ...

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

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



Architectures

at Tr = 1

), N);

= true;

AXDEPTH)

v = true;

/ive)

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

andom walk - done properly, closely

f:

efl + refr)) && (dep

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

at a = nt

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

(AXDEPTH)

survive = SurvivalProbability(diff) f: 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, I urvive; pdf; 1 = E * brdf * (dot(N, R) / pdf); sion = true:

Direct Mapped Cache

struct CacheLine

// 24 bit for 4G uint tag; 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
	addre	255	

- 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 < MAXDEF)

: = inside ? l : . . ht = nt / nc, ddn os2t = 1.0f - nnt 2, N); 3)

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

= * diffuse; = true;

efl + refr)) && (depth < NAXDEDII

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

AXDEPTH)

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



Architectures

nics & (depth < MODE)

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

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

= * diffuse; = true;

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

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

(AXDEPTH)

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

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) (n);

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

; t33 brdf = SampleDiffuse(diffuse, N, r1, r2, &R, &pdf 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		
0000018		cache
000001C		
0000020	/ /	
0000024		
0000028		
000002C	/	
0000030	/	
0000034		
0000038		
000003C		
RAM		



};

Architectures

nics & (depth < AAACO

: = inside ? 1 + 1.0 ht = nt / nc, ddn - ps2t = 1.0f - nnt -D, N); 2)

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

= * diffuse; = true;

efl + refr)) && (depth < MAXDE

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) 88 - - touc: 31

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

andom walk - done properly, cl<u>esely follo</u> /ive)

; it3 br

sion = true:

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

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

0000	
0001	
0002	
0003	
0004	
0005	
0006	
0007	
8000	
0009	
000A	
000B	
000C	
000D	
000E	
000F	
	0001 0002 0003 0004 0005 0006 0007 0008 0008 0008 0008 0008

slo



N-Way Set Associative Cache

bool valid, dirty;

CacheLine cache[16][4];

This cache again holds 256 bytes.

4

tag

3

2 1

set

0

offs

struct CacheLine

uint tag;

uint data;

};

Architectures

at a = nt

efl + refr)) && (depth

), N); refl * E * diffuse;

(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;

sion = true:

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

N-Way Set Associative Cache

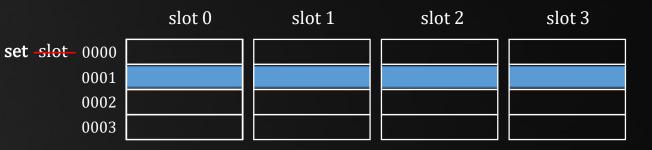
struct CacheLine

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 2 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);

AXDEPTH)

v = true:

f:

refl * E * diffuse;

survive = SurvivalProbability(d:

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

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:



26

};

Architectures

nics & (depth < Motos

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

= * diffuse; = true;

efl + refr)) && (depth < NOXDEF

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

AXDEPTH)

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, cl<u>asely follow</u> /ive)

```
t3 brdf
```

sion = true:

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

tag

32KB, 4-Way Set Associative Cache

uint tag; // 19 bit for 4G

This cache holds 32768 bytes in 512 cachelines,

set

65

offs

0

struct CacheLine

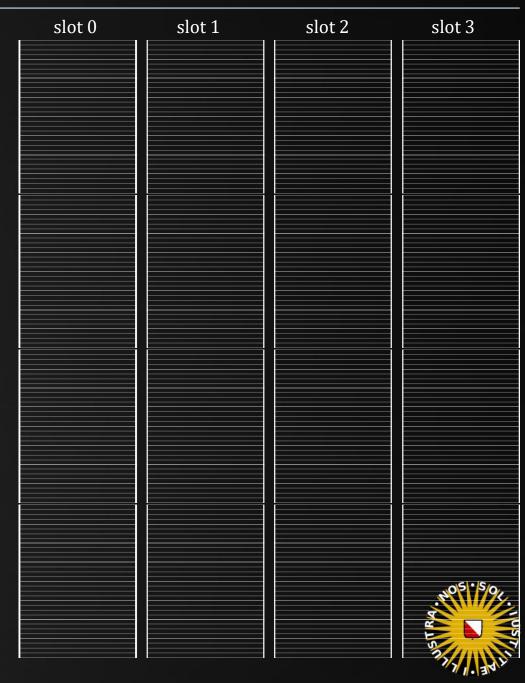
uchar data[64];

bool valid, dirty;

CacheLine cache[128][4];

organized in 128 sets of 4 cachelines.

13 12



tics & (depth < Moder)

: = inside ? | ht = nt / nc, ddn os2t = 1.0f - nnt ? 0, N); 8)

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

= * diffuse = true;

efl + refr)) && (depth < MAXDEPID

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

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) * (Psd);

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



tics & (depth < Mox06

at a = nt - nc, b = nt at Tr = 1 - (R0 + (1 - Rc) Fr) 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 closely
f;

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) (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/

END of "Caching (1)"

next lecture: "Caching (2)"



nics & (depth < MAXOS

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

at a = nt - nc, b = nt - nc 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, closed Hf:

radiance = SampleLight(&rand, I, &L, &lished a.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) (1800)

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:

/INFOMOV/ PRACTICAL SLIDES





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

wark - done broberty, crosery

/ive)



ics & (depth < ≫occs

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

= * diffuse = true:

. •fl + refr)) && (death < MAXISSIN

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

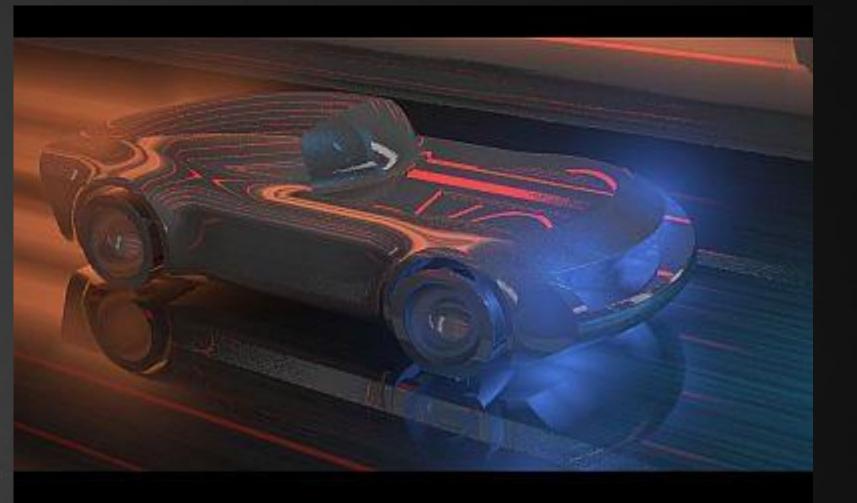
AXDEPTH)

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

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



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



On optimizing the galaxy:

- 1. You don't need to match 90% of my performance to pass. Only 55%.
- 2. Yes you can use SIMD.
- 3. Without SIMD you can score an 8.
- 4. You may share ideas.
- 5. You may not share code.
- 6. Optimal sharing means everyone gets the same grade (and learned the most).
- 7. I'm almost always on Discord.
- 8. Use the reference app to predict your score (within 1%).
- 9. Use PRTSC to verify your output against reference.
- w = true; at brdfPdf = EvaluateDiffuse(L, N) * Psurvive at3 factor = diffuse * INVPI; at cosThetaOut = diffuse * INVPI; at cosThetaOut = dot(N, L); E * ((weight * cosThetaOut) / directPdf) * (radiance)
- andom walk done properly, closely following Soul /ive)

at a = nt - nc,

), N);

AXDEPTH)

efl + refr)) && (depth <)

survive = SurvivalProbability(diff

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

refl * E * diffuse;

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

