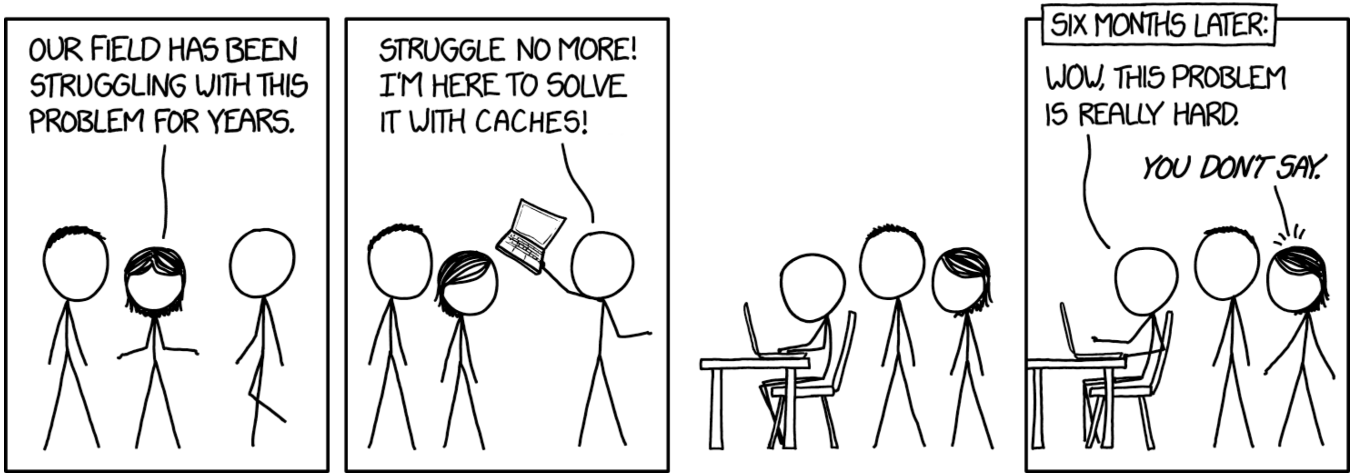

OUR FIELD HAS BEEN STRUGGLING WITH THIS PROBLEM FOR YEARS.
STRUGGLE NO MORE! I'M HERE TO SOLVE IT WITH CACHES!
SIX MONTHS LATER:
WOW, THIS PROBLEM IS REALLY HARD.
YOU DON'T SAY.

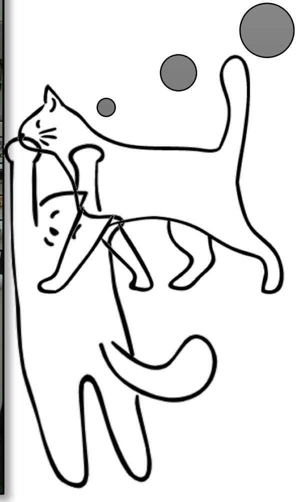
xkcd #1831
(slightly edited)

caches

CPEN212   2022 W2

a place of mind
THE UNIVERSITY OF BRITISH COLUMBIA

Q. what is a cache?



images: Bear-commonswiki/Wikimedia; 南無観自在/Wikimedia; TTTNIS/Wikimedia; J Amadeo/NounProject

key idea: reuse

temporal reuse

- **observation**: some data often accessed **repeatedly**
  - e.g., loop counter + other variables accessed in loop

- **idea**: keep frequently accessed data nearby

- **cache**: small, fast storage near use location
  - e.g., on-chip memory near CPU = cache for off-chip DRAM
  - e.g., physical DRAM = cache for on-disk virtual memory
  - e.g., proxy web server / CDN = cache for remote webpages

spatial reuse

- **observation**: data often accessed **in local clusters**
  - e.g., iterating through an array
- **idea**: also cache data **spatially near** recent accesses
- **cache line**: a range of data brought in cache at once
  - e.g., several contiguous addresses (e.g., 64–128 bytes)
  - e.g., nearby pixels (in a GPU texture cache)

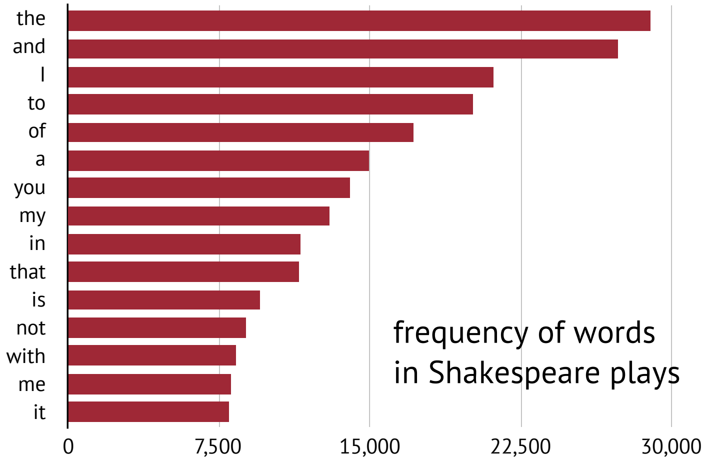
Q. what kind of reuse was the cat's wallet?

key idea: hierarchy

cache hierarchies

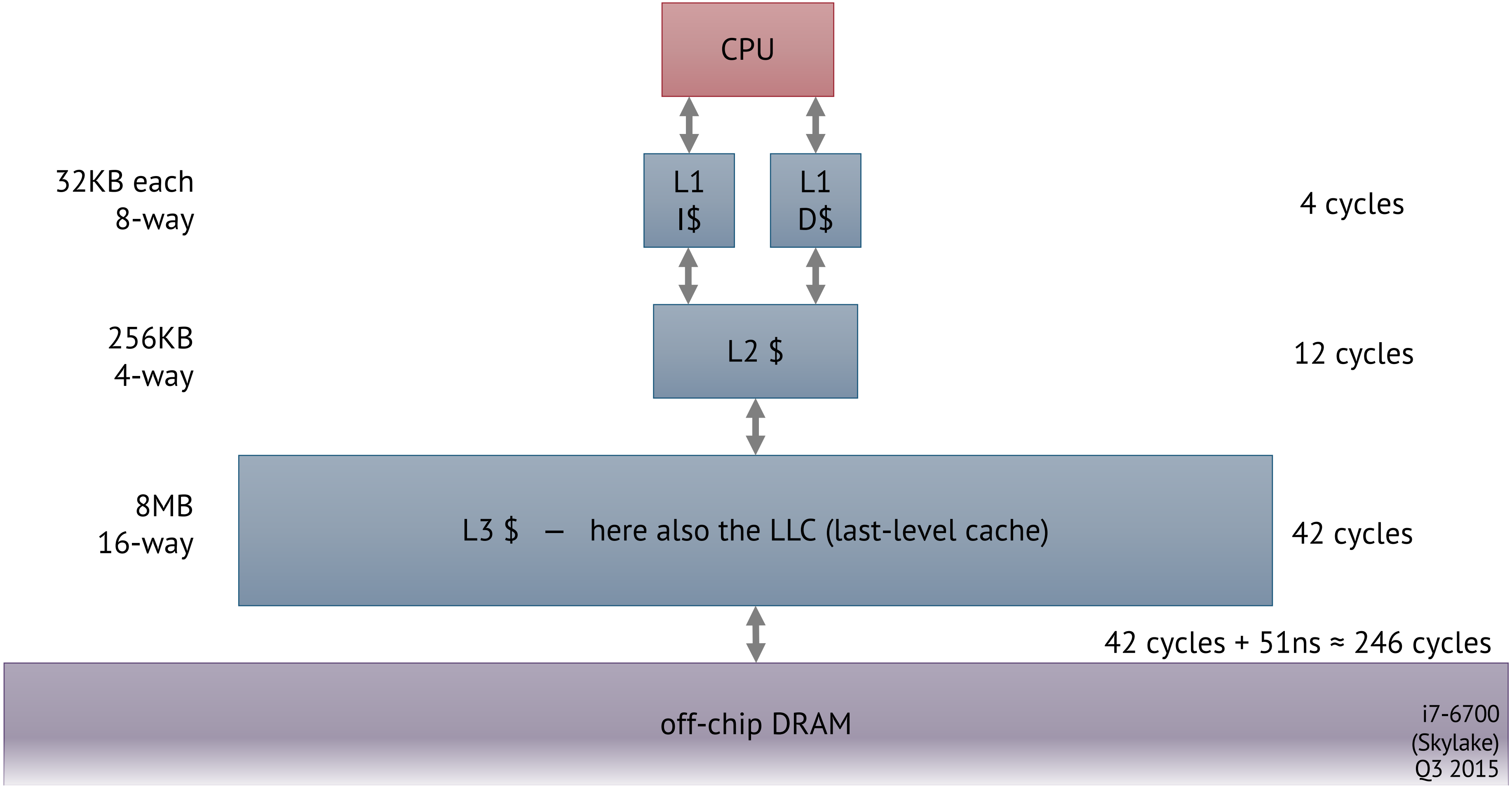
- memory latency ~ $\sqrt{\text{size}}$
  (w/ same technology)

- few data accessed very often
  - e.g., words in language



frequency of words
in Shakespeare plays

- **idea:** a fast small $ in front of a larger, slower $
  - and maybe another, even larger, even slower $...



CPU

L1
I$

L1
D$

L2 $

L3 $ – here also the LLC (last-level cache)

off-chip DRAM

32KB each
8-way

256KB
4-way

8MB
16-way

4 cycles

12 cycles

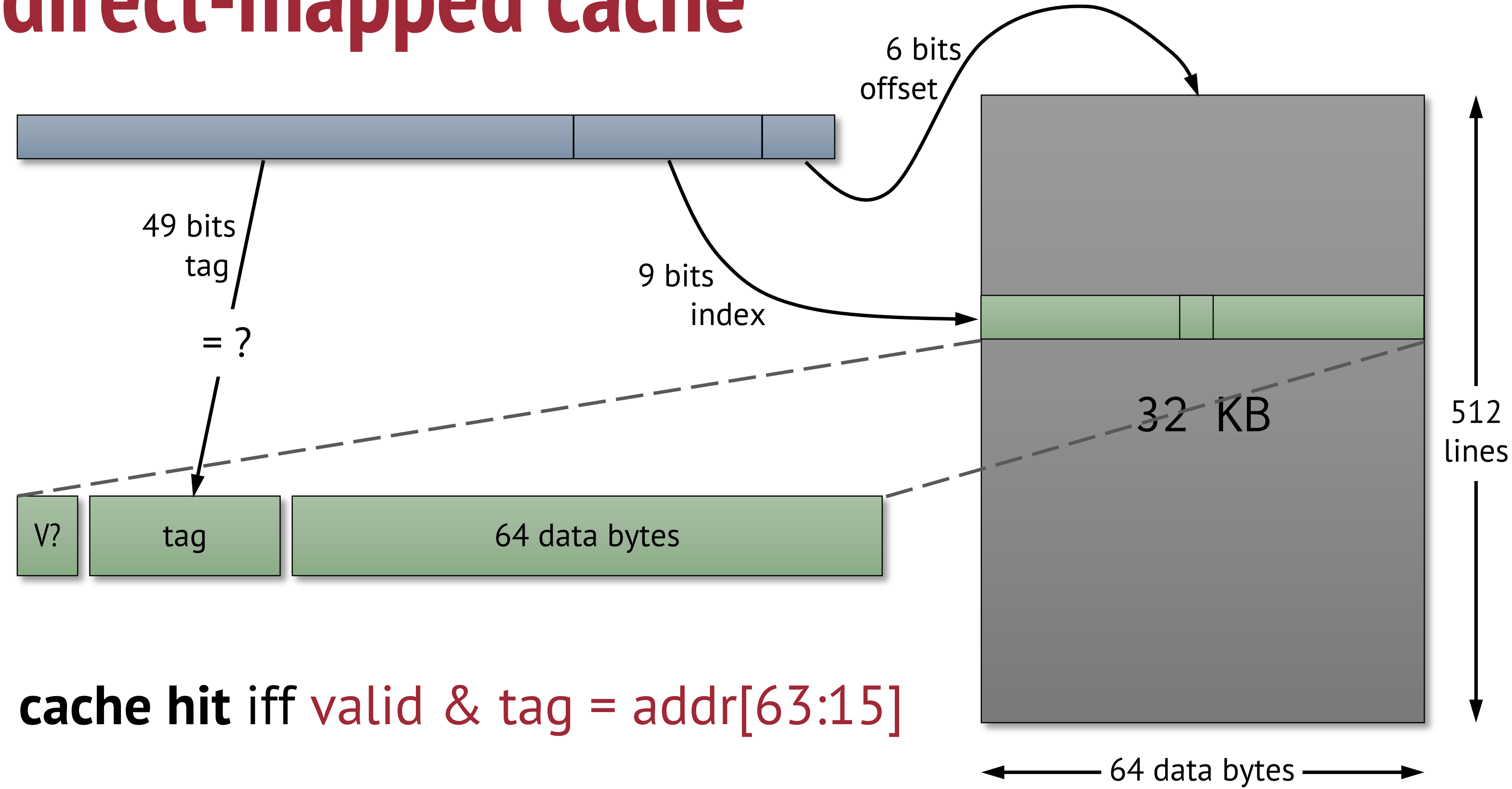
42 cycles

42 cycles + 51ns ≈ 246 cycles

i7-6700
(Skylake)
Q3 2015

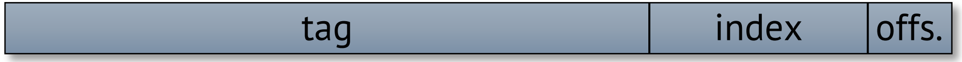
cache organization

direct-mapped cache



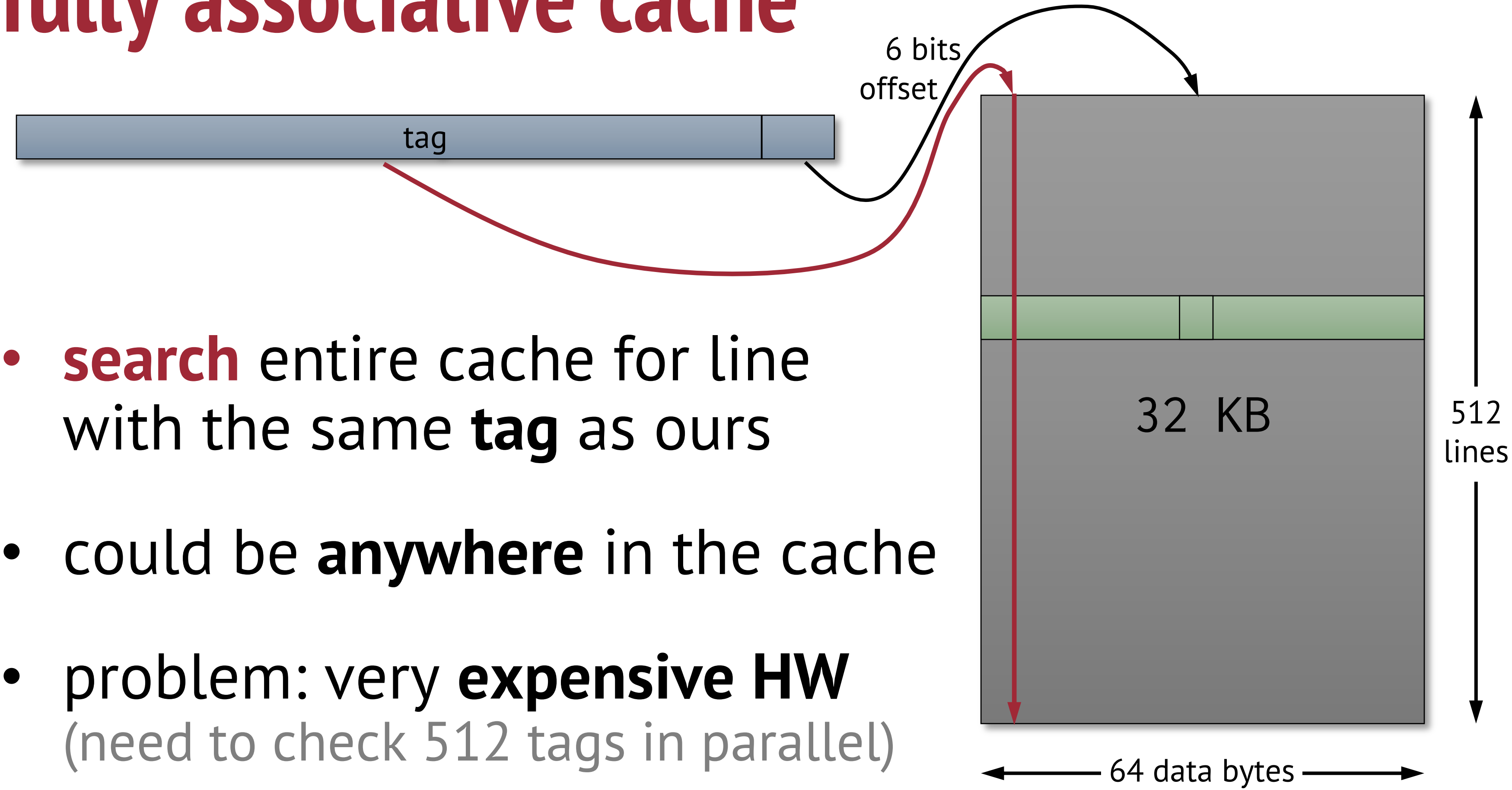
cache hit iff valid & tag = addr[63:15]

direct-mapped cache problems



- same index bits → **conflict** → eviction
- if indices random, Pr[conflict] > 50% if only **27 lines**
- even worse: **pathological patterns**
  - e.g., if accessing many words 4096 bytes apart
  - **only 8** cache lines used even if the cache has 512!
  - all accesses past the 8th **evict** some prior cache line

fully associative cache



- **search** entire cache for line with the same **tag** as ours

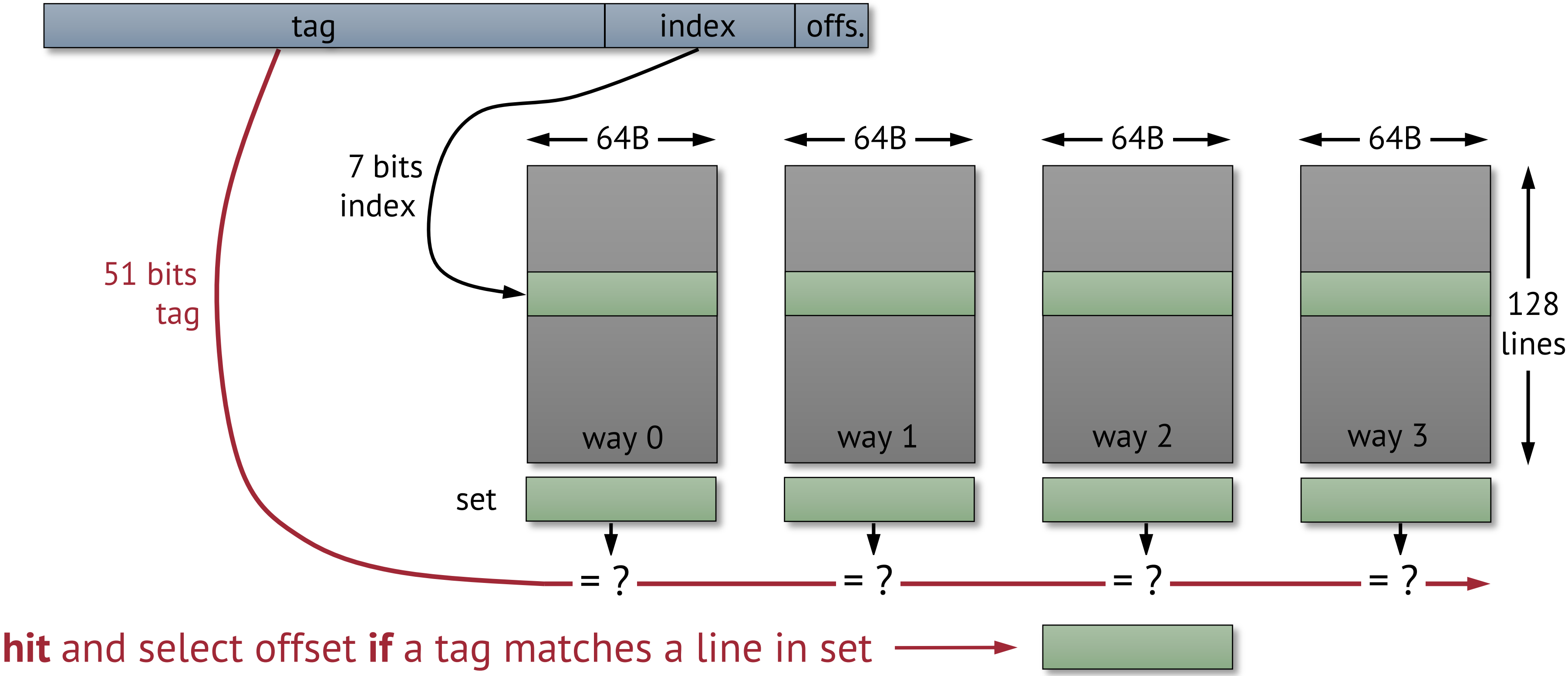
- could be **anywhere** in the cache

- problem: very **expensive HW**
  (need to check 512 tags in parallel)

compromise: set-associative caches

- **idea**: organize cache lines in **sets** (e.g., 4 lines / set)
- use index bits to **select sets**
- **search** for tag in all **ways** in the selected set
    - some extra hardware but # ways << # indices so not bad
- now conflict harder: need **|set|+1** equal indices

set-associative cache



hit and select offset **if** a tag matches a line in set

replacement policies

- if want to insert but set full, which line to evict?

- ideal replacement policy (Bélády MIN algorithm): evict line re-referenced **furthest** in the future

- but, can't know the future   T_T

- usually **least-recently used** (LRU) or **random**

*actually, pseudo-LRU (pLRU)

writing cache-friendly code

using caches efficiently

- **lay out** data accessed together inside one cache line
  - after the first miss the rest of cache line will hit
  - **Q**: **spatial or temporal locality?**

- reuse data in chunks **that fit in the cache** (tiles)
  - data likely to still be in cache when reused
  - **Q**: **spatial or temporal locality?**

- focus on **innermost cache first (L1)**, then outer caches

finding reuse: matrix multiplication

key idea: find matrix indices **constant** across a loop

need to loop through r, c

need to loop through i

$$\forall\ r, c: \quad z[r, c] \leftarrow \sum_{i} x[r, i] \cdot y[i, c]$$

z **not indexed** with i

→ z **reused** for every i
(for accumulation)

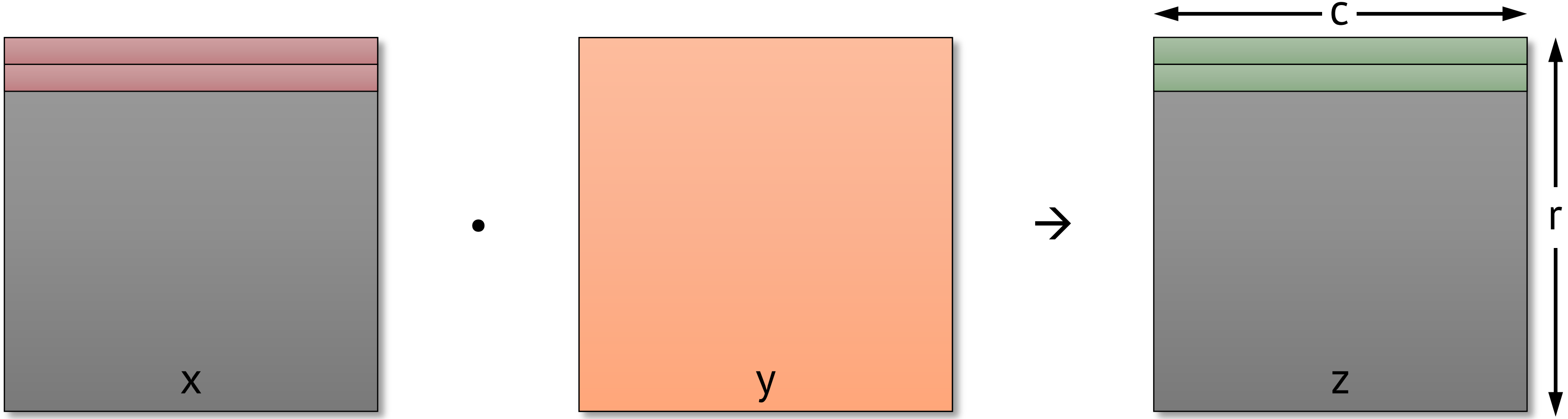
x **not indexed** with c

→ x **reused** for every c

y **not indexed** with r

→ y **reused** for every r

let’s check: is y reused for every r?

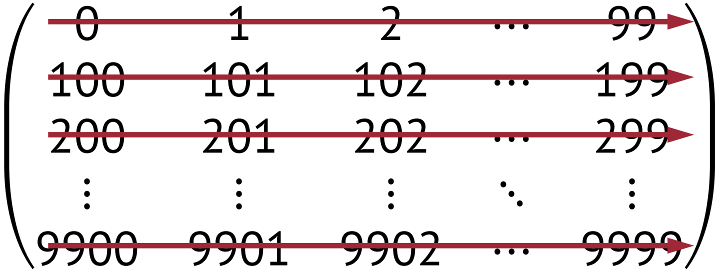


verify *x* and *z* reuse yourself (same logic)

Q. spatial or temporal reuse?

interlude: 2D matrix representations

row-major order



$$\begin{pmatrix} 0 & 1 & 2 & \cdots & 99 \\ 100 & 101 & 102 & \cdots & 199 \\ 200 & 201 & 202 & \cdots & 299 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 9900 & 9901 & 9902 & \cdots & 9999 \end{pmatrix}$$

```
double *matrix = { 0, 1, 2, ..., 99, 100, 101, 102, ..., 9999 };

double element = matrix[100 * row + col];
```

interlude: 2D matrix representations

column-major order

$$\begin{pmatrix} 0 & 1 & 2 & \cdots & 99 \\ 100 & 101 & 102 & \cdots & 199 \\ 200 & 201 & 202 & \cdots & 299 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 9900 & 9901 & 9902 & \cdots & 9999 \end{pmatrix}$$

```
double *matrix = { 0, 100, 200, ..., 9900, 1, 101, 201, ..., 9999 };

double element = matrix[row + 100 * col];
```

example: matrix multiplication

- problem setup: z = x · y
  - x, y, z are 64×64 matrices of `doubles` (8B each)
  - 16KB cache, 4-way set-associative

- Q. does each matrix fit in the cache?

- Q. does one row or column fit in the cache?

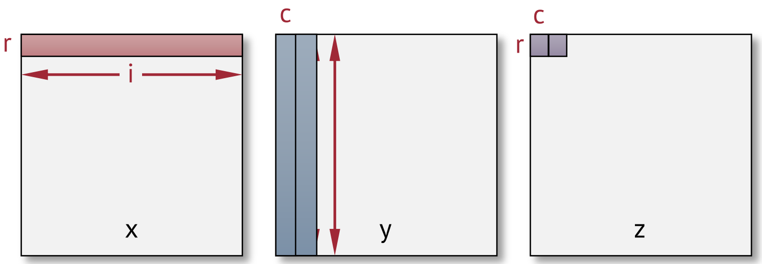
- plan: maximize reuse at **row / column** level

```
N = 64; // 3 × 4096-elt matrices (32KB each); 16KB cache
for (r = 0; r < N; ++r) // output row
    for (c = 0; c < N; ++c) // output col
        for (i = 0; i < N; ++i) // x col, y row
            z[N * r + c] += x[N * r + i] * y[N * i + c];
```

read *and* write

read

read



Q. expected reuse?

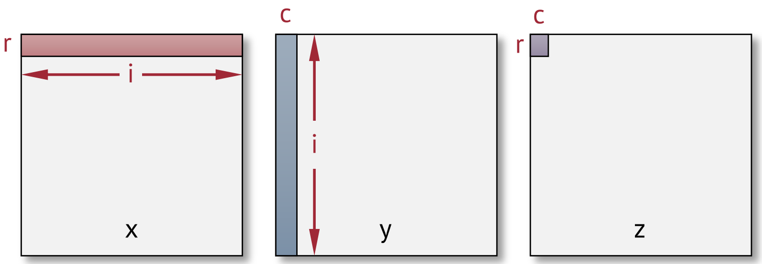
Q. cache miss rates?

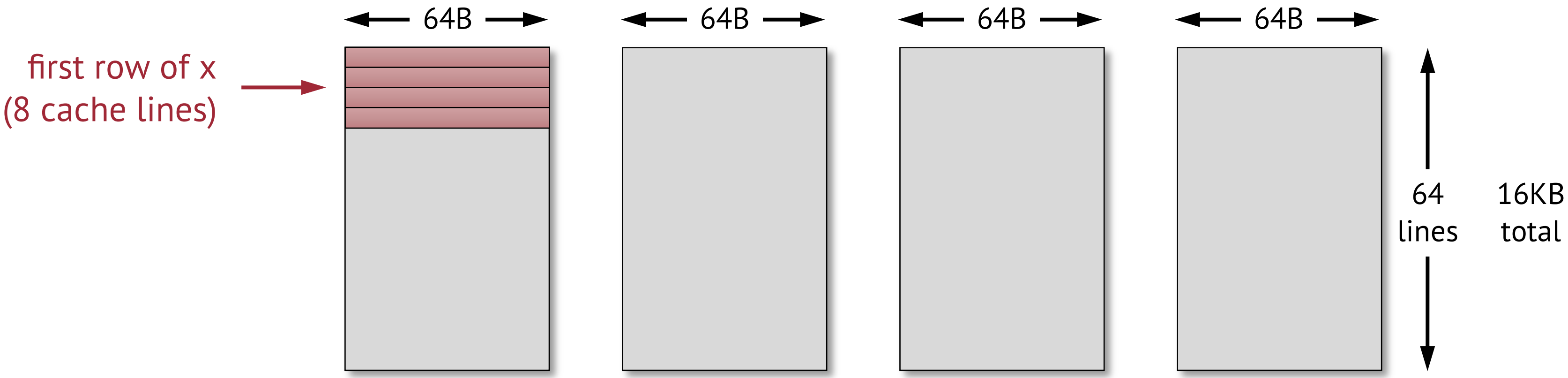
```
$ valgrind --tool=cachegrind -D1=16384,4,64 ...
                          ...
D    refs:        1,056,375  (792,821 rd   + 263,554 wr)
D1   misses:        268,094  (267,960 rd   +     134 wr)
D1   miss rate:        25.4% (   33.8%     +     0.1%  )
```

~1 in 3 reads miss???

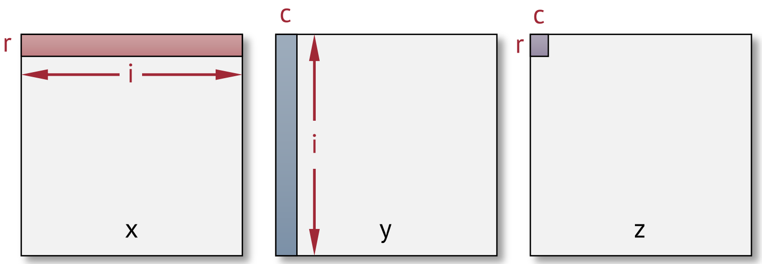
most writes hit

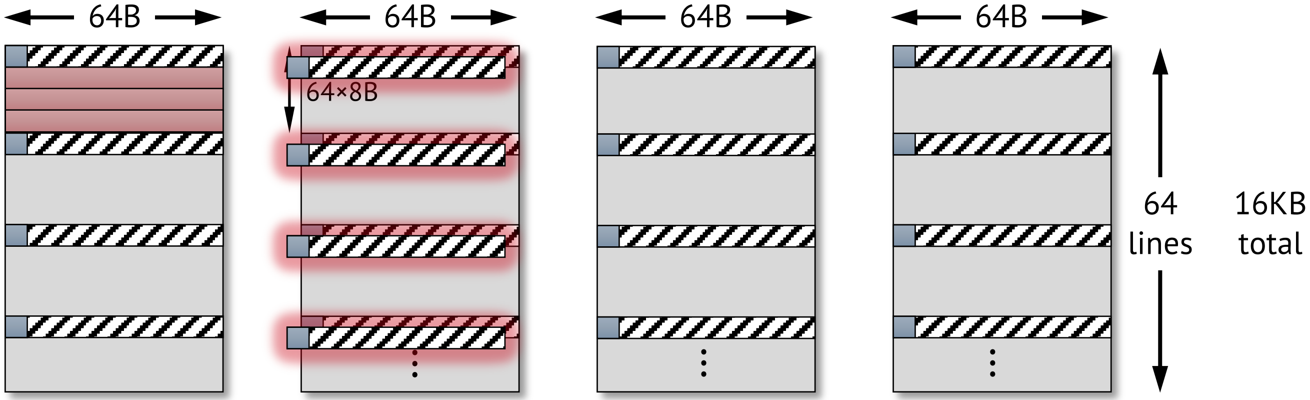
```
N = 64; // 3 × 4096-elt matrices (32KB each); 16KB cache
for (r = 0; r < N; ++r) // output row
    for (c = 0; c < N; ++c) // output col
        for (i = 0; i < N; ++i) // x col, y row
            z[N * r + c] += x[N * r + i] * y[N * i + c];
```





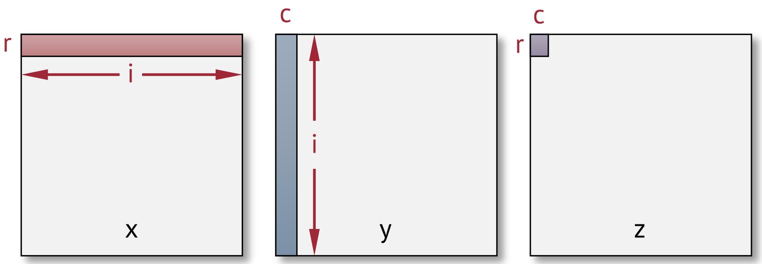
```
N = 64; // 3 × 4096-elt matrices (32KB each); 16KB cache
for (r = 0; r < N; ++r) // output row
    for (c = 0; c < N; ++c) // output col
        for (i = 0; i < N; ++i) // x col, y row
            z[N * r + c] += x[N * r + i] * y[N * i + c];
```





problem: all N^3 accesses to y miss in the cache!

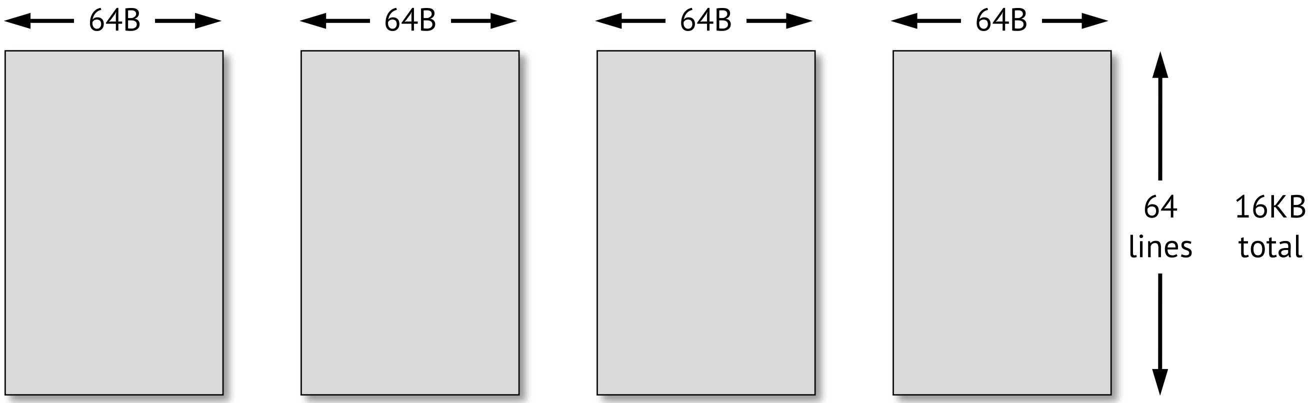
```
N = 64; // 3 × 4096-elt matrices (32KB each); 16KB cache
for (r = 0; r < N; ++r) // output row
    for (c = 0; c < N; ++c) // output col
        for (i = 0; i < N; ++i) // x col, y row
            z[N * r + c] += x[N * r + i] * y[N * i + c];
```



observation: want *i* loop to be **outside** of *c* loop

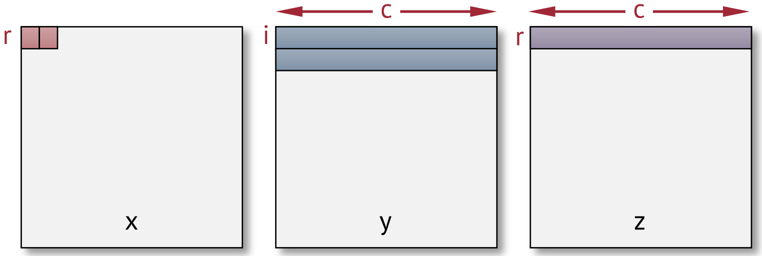
idea:

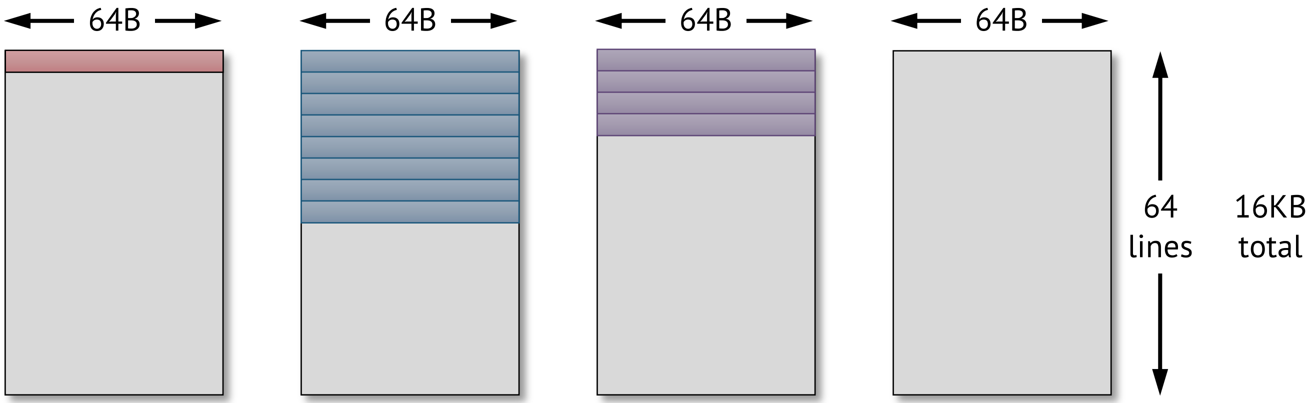
innermost loop swap



Q. does swapping maintain functional correctness?

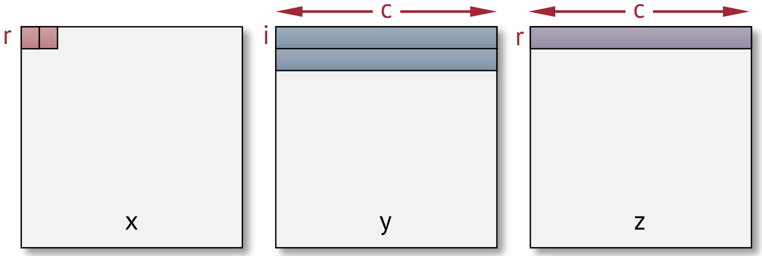
```
N = 64; // 3 × 4096-elt matrices (32KB each); 16KB cache
for (r = 0; r < N; ++r) // output row
    for (i = 0; i < N; ++i) // x col, y row
        for (c = 0; c < N; ++c) // output col
            z[N * r + c] += x[N * r + i] * y[N * i + c];
```

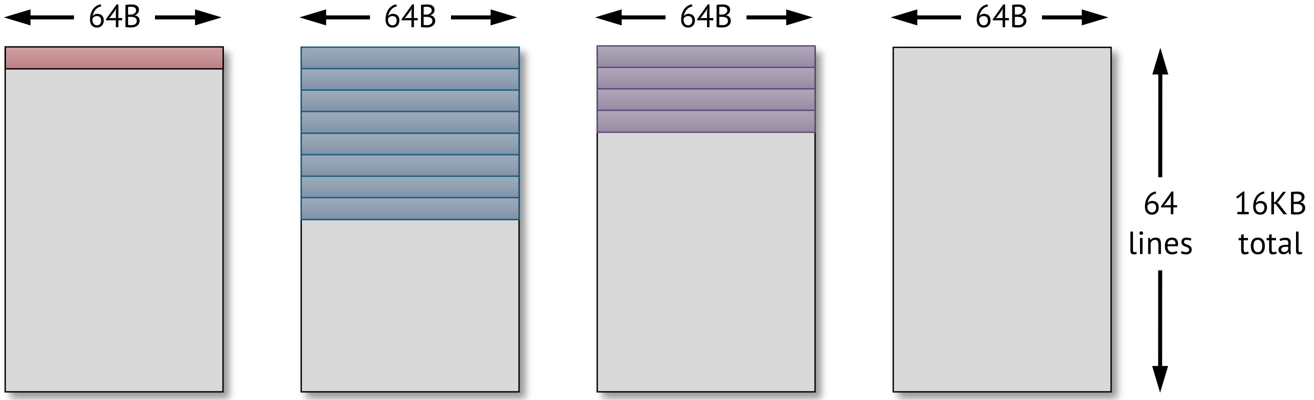




Q. what happens when i increments?

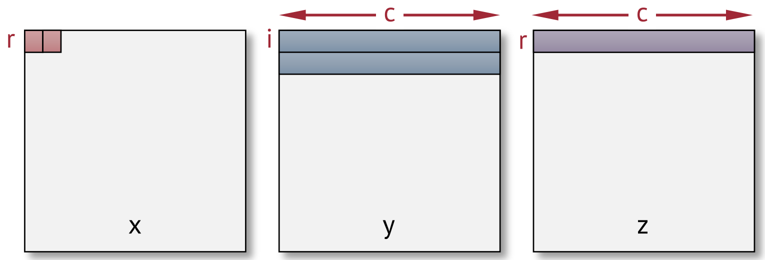
```
N = 64; // 3 × 4096-elt matrices (32KB each); 16KB cache
for (r = 0; r < N; ++r) // output row
    for (i = 0; i < N; ++i) // x col, y row
        for (c = 0; c < N; ++c) // output col
            z[N * r + c] += x[N * r + i] * y[N * i + c];
```





Q. estimated read / write miss rates for large N?

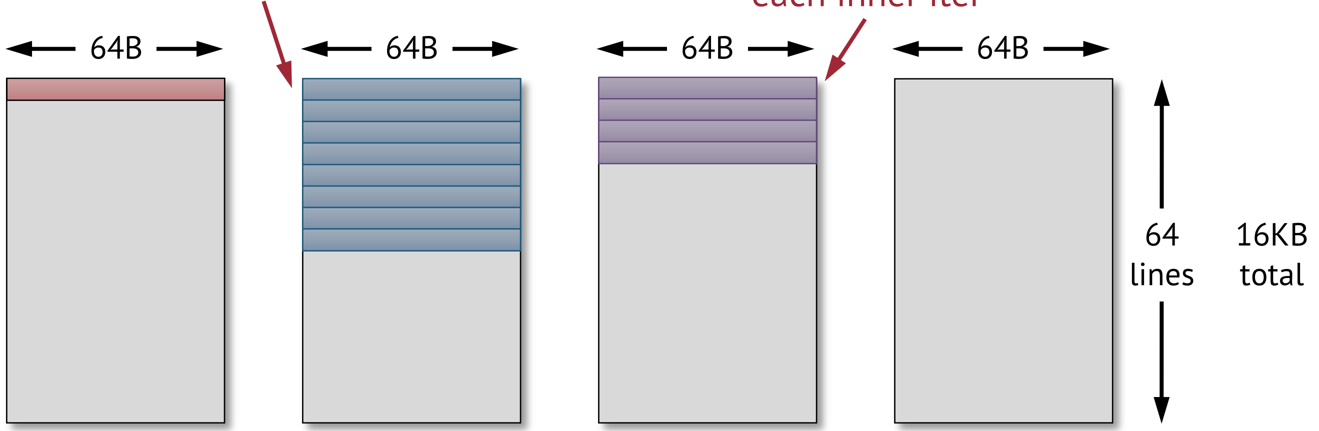
```
N = 64; // 3 × 4096-elt matrices (32KB each); 16KB cache
for (r = 0; r < N; ++r) // output row
    for (i = 0; i < N; ++i) // x col, y row
        for (c = 0; c < N; ++c) // output col
            z[N * r + c] += x[N * r + i] * y[N * i + c];
```



fully reused in each inner iter

spatial reuse only (**1 in 8** refs miss)

fully reused in each inner iter



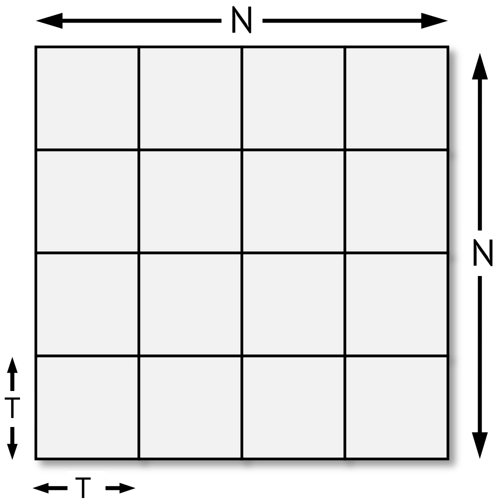
also **x** and **z** occasionally evicted by **y**

```
D1  miss rate:      3.2% (    4.3%     +     0.1%   )
```

processing in cache-sized chunks

- loop reordering → better reuse
- but total footprint still **too big for the cache**

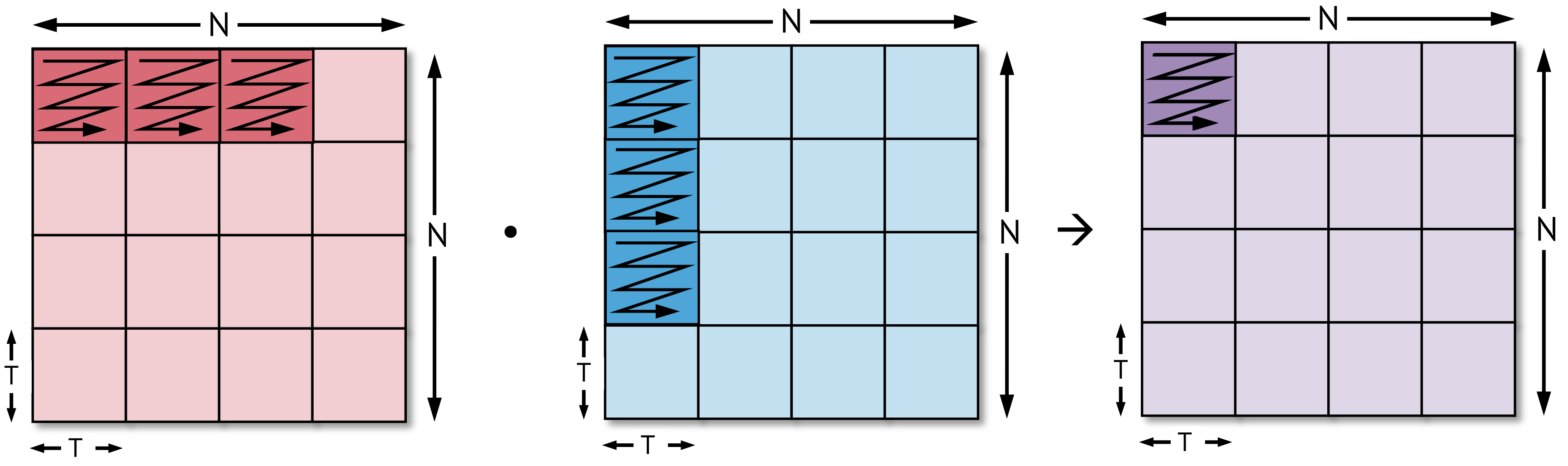
- idea: **tiling**
  - split matrix into **submatrices**
  - as big as we can where all three still **fit in the cache**
  - finish tile before moving on



```
N = 64; T = 16;
for (r0 = 0; r0 < N; r0 += T) // output row tile
    for (c0 = 0; c0 < N; c0 += T) // output col tile
        for (i0 = 0; i0 < N; i0 += T) // x col, y row tile
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
            for (r = r0; (r < r0 + T) && (r < N); ++r) // output row
                for (i = i0; (i < i0 + T) && i < N; ++i) // x col, y row
                    for (c = c0; (c < c0 + T) && (c < N); ++c) // output col
                        z[N * r + c] += x[N * r + i] * y[N * i + c];
```

loop through tiles

loop through elements inside tile



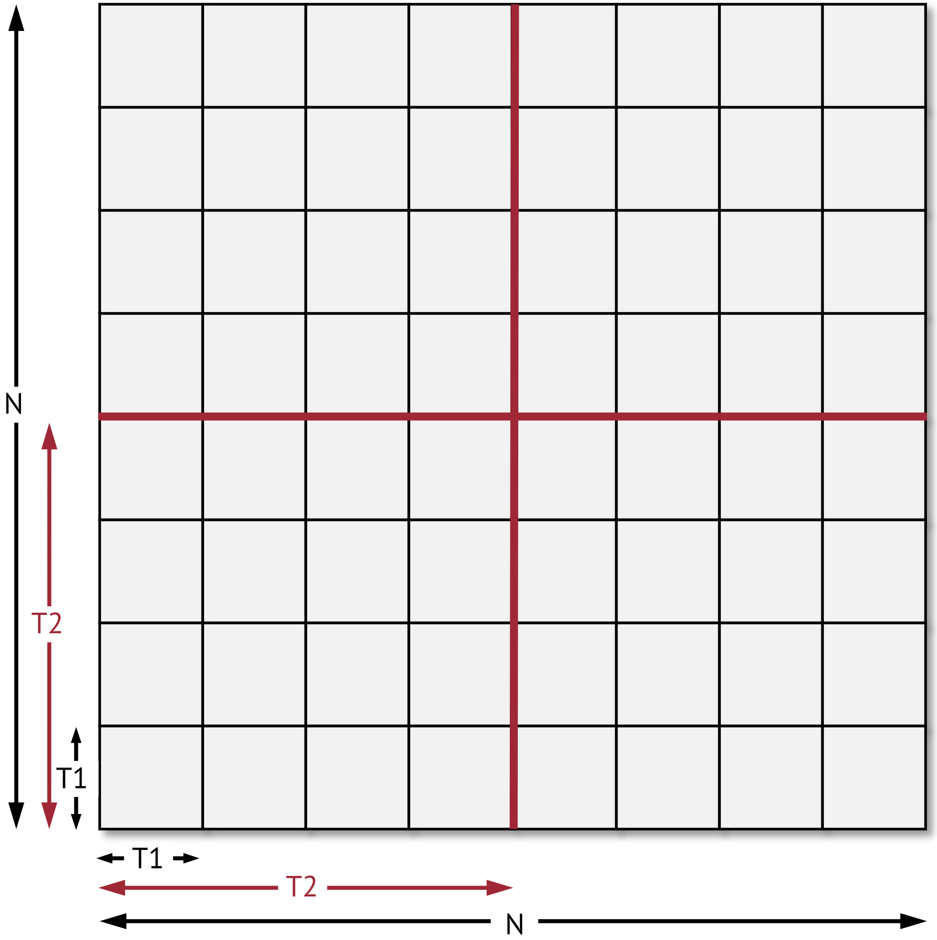
L1 + L2 cache?

- **two** levels of cache

→ **two** tiling levels

- reuse T1 in L1$ (inner tile)
- reuse T2 in L2$ (outer tile)

Q. **what about L3$?**



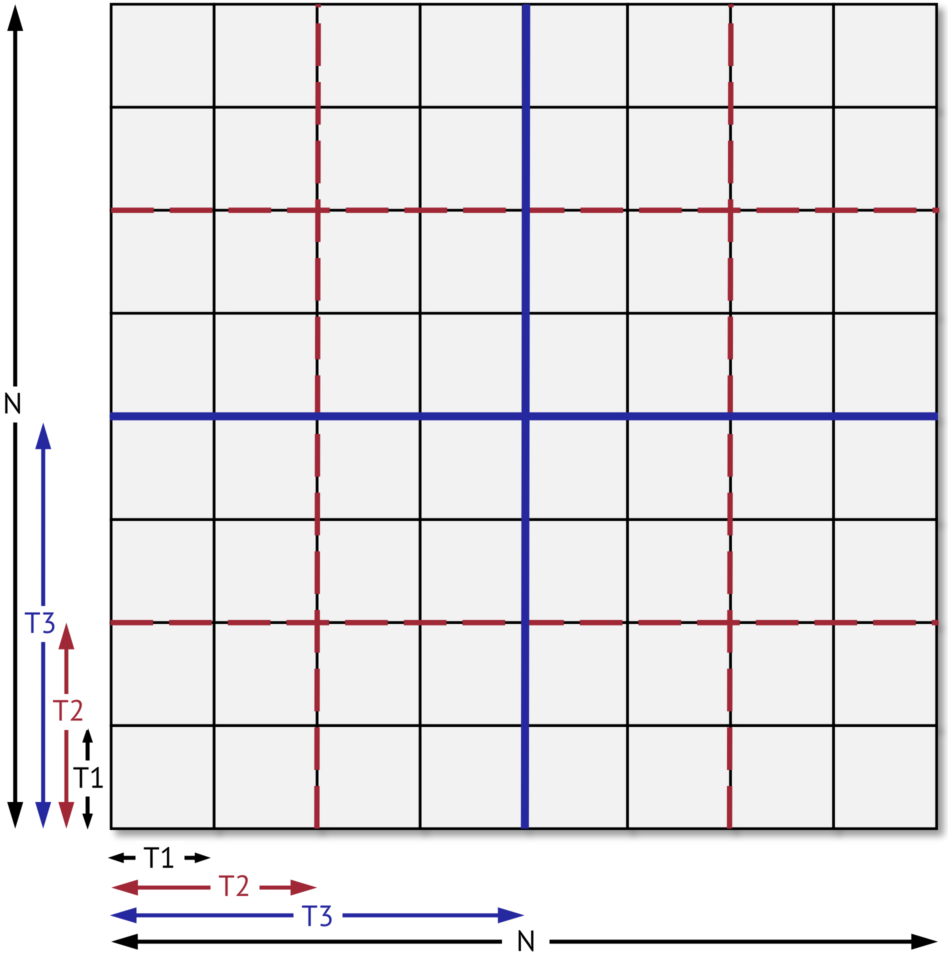
L1 + L2 cache?

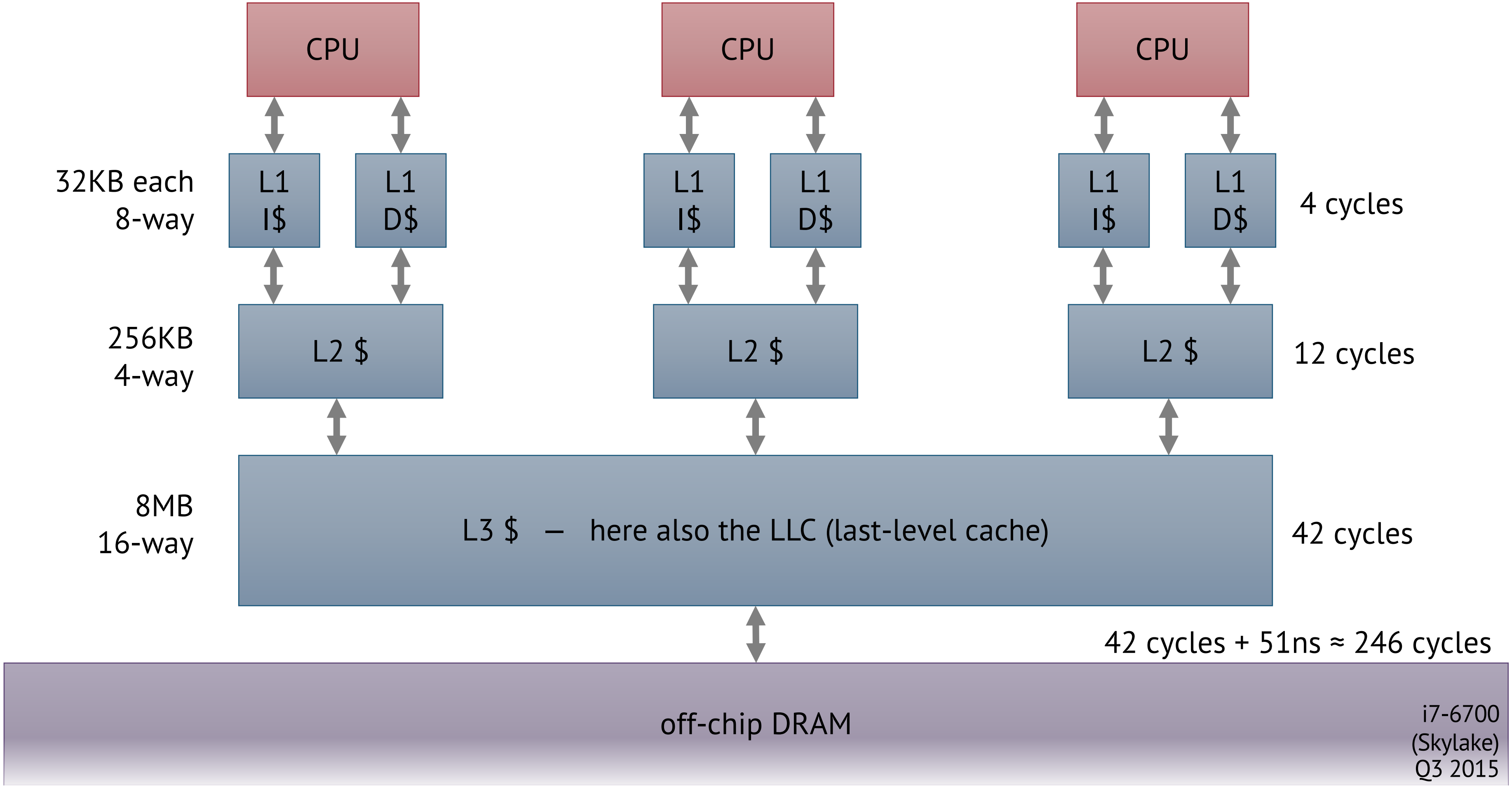
- **two** levels of cache

→ **two** tiling levels

- reuse T1 in L1$
  (inner tile)
- reuse T2 in L2$
  (outer tile)

Q. **what about L3$?**





CPU

CPU

CPU

32KB each
8-way

L1 I$

L1 D$

L1 I$

L1 D$

L1 I$

L1 D$

4 cycles

256KB
4-way

L2 $

L2 $

L2 $

12 cycles

8MB
16-way

L3 $ – here also the LLC (last-level cache)

42 cycles

42 cycles + 51ns ≈ 246 cycles

off-chip DRAM

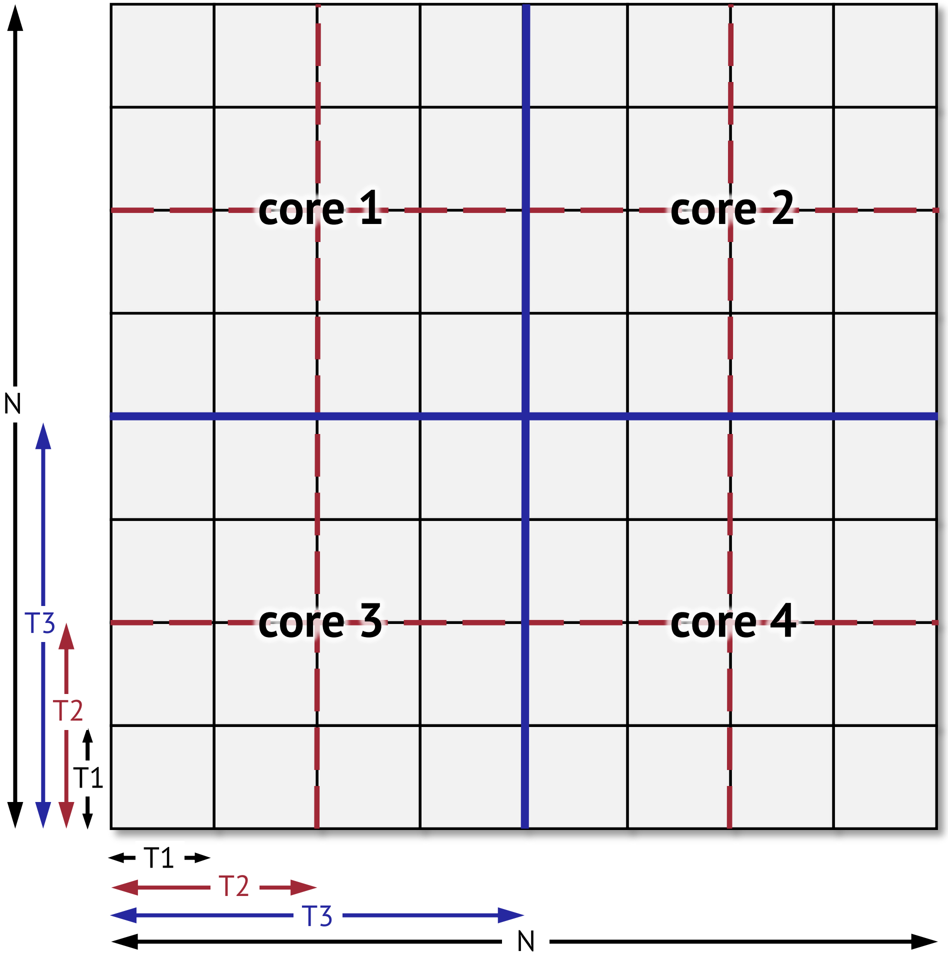
i7-6700
(Skylake)
Q3 2015

multicore reuse

- typical hierarchy
  - L1, L2 **private**
  - L3 (aka LLC) **shared**

Q. how to parallelize?

- v1: no L3 reuse T_T

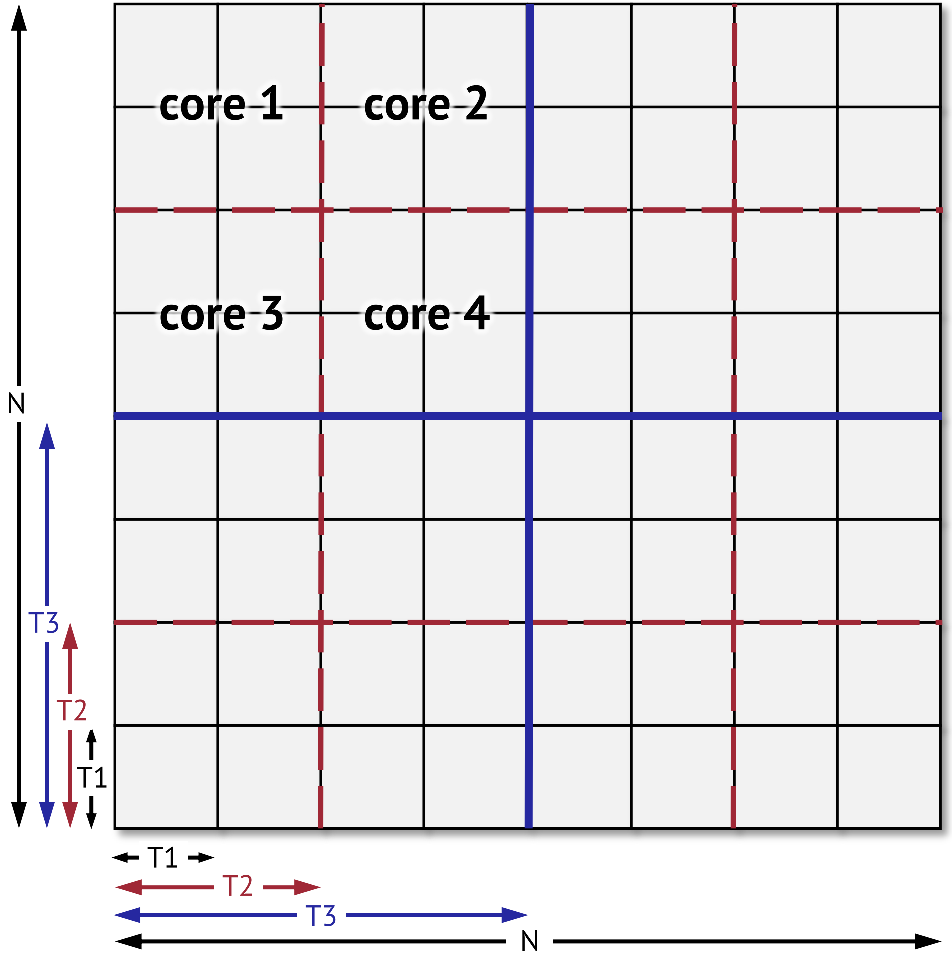


multicore reuse

- typical hierarchy
  - L1, L2 **private**
  - L3 (aka LLC) **shared**

Q. how to parallelize?

- ~~v1: no L3 reuse T_T~~
- v2: L3 reuse 👍

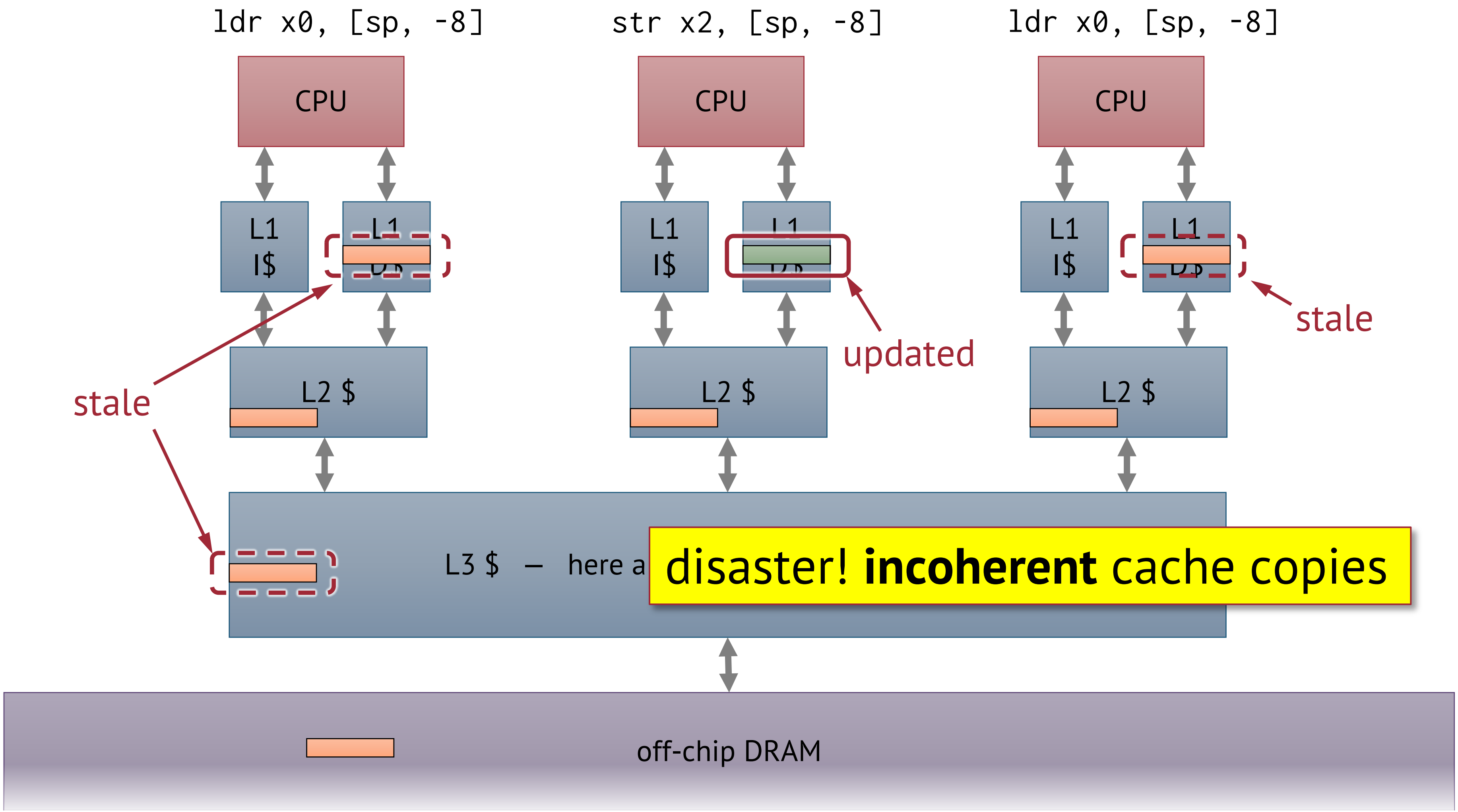


cache-aware optimization summary

- spatial locality → **change loop bounds**
  - columns inner loop, rows outer loop (if row-major)

- temporal locality → **tile matrix**
  - size tiles so that cache holds one tile from each matrix

- cache hierarchy → **nested tiling**
  - L1$ + L2$ → 2 tiling levels
  - L1$ + L2$ + L3$ → 3 tiling levels
  - on multicore, ensure **shared-level cache** reuse

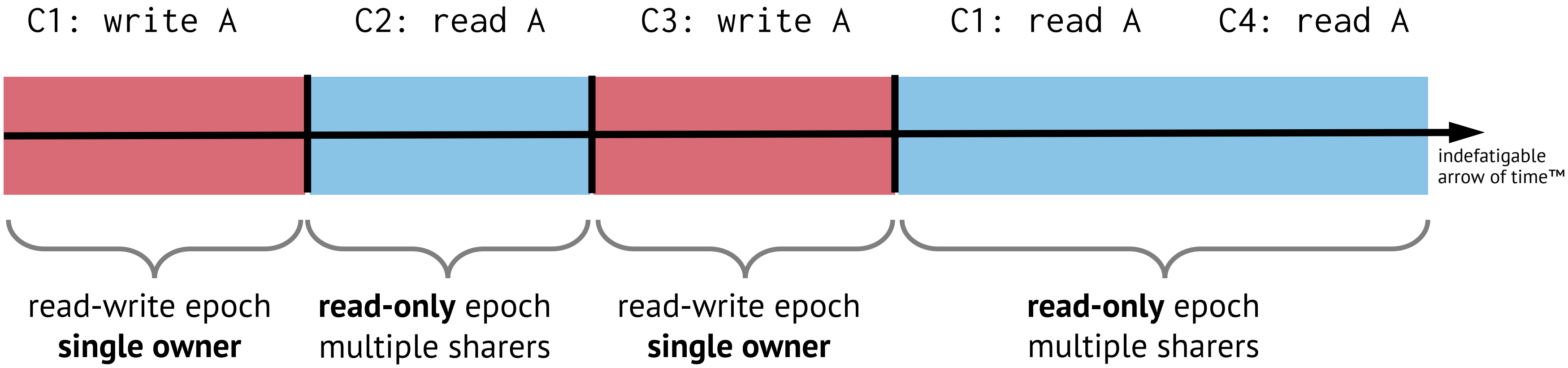


```
ldr x0, [sp, -8]        str x2, [sp, -8]        ldr x0, [sp, -8]
```

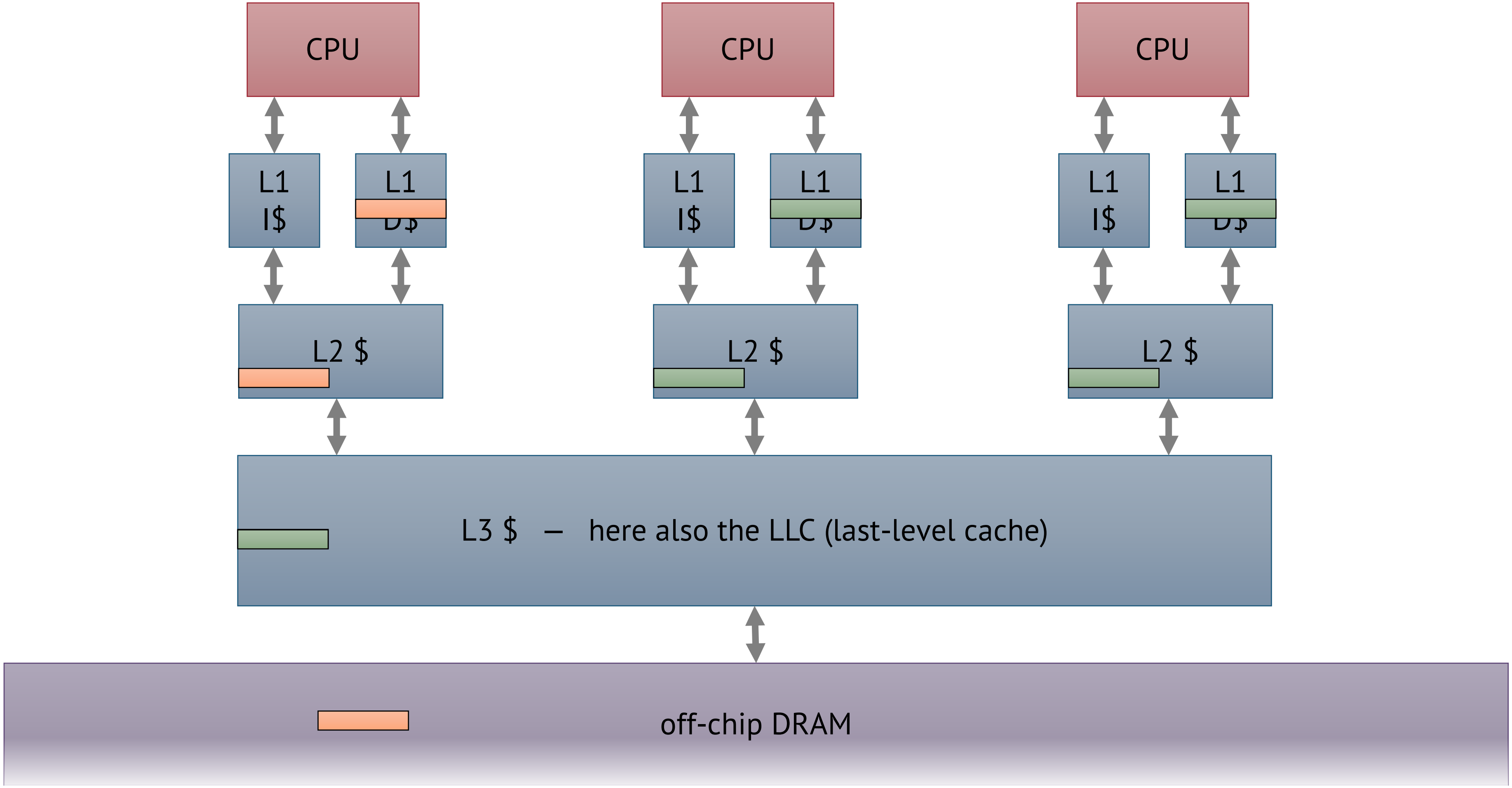
disaster! **incoherent** cache copies

idea: before writing, become the only owner

invariant: single writer, multiple readers



```
ldr x0, [sp, -8]          str x2, [sp, -8]          ldr x0, [sp, -8]
```



CPU CPU CPU

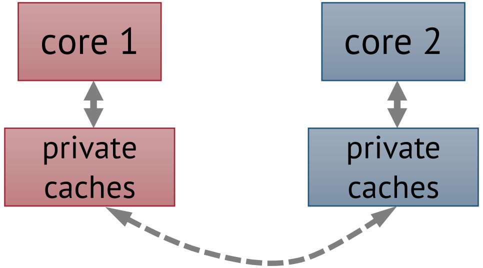
L1 I$ L1 D$ L1 I$ L1 D$ L1 I$ L1 D$

L2 $ L2 $ L2 $

L3 $ — here also the LLC (last-level cache)

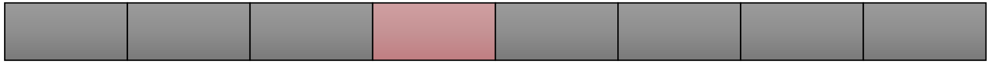
off-chip DRAM

considerations for coherent caches



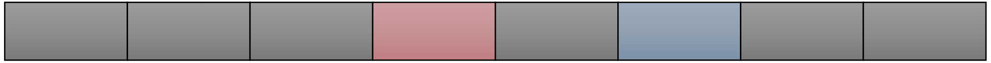
- if two cores modify the same data, it will **ping-pong** between the cores

- **true sharing**: same block, **same** bytes



  - C1 reads cache block to access word 3
  - C2 writes word 3, invalidates C1’s copy of block

- **false sharing**: same block, **different** bytes



  - C1 reads cache block to access word 3
  - C2 writes word 5, invalidates C1’s copy of block *only because it’s only same cache line!*

→ align R/W shared data on cache-line boundaries