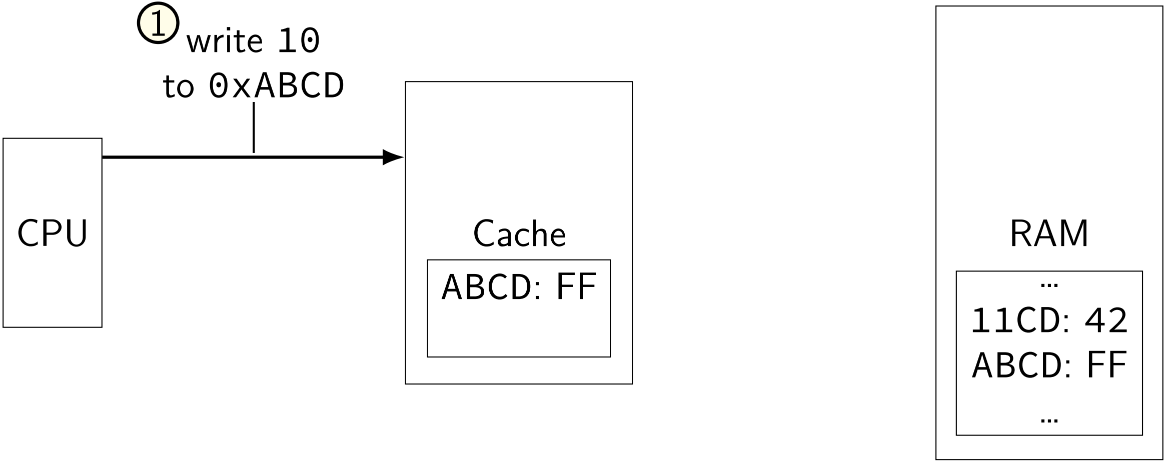
Changelog

Changes made in this version not seen in first lecture:

- 1 November 2017: “Cache optimizations”: don’t mark writeback as better miss rate; what it reduces is similar to miss rate (amount of times we go to next level), but not the same thing

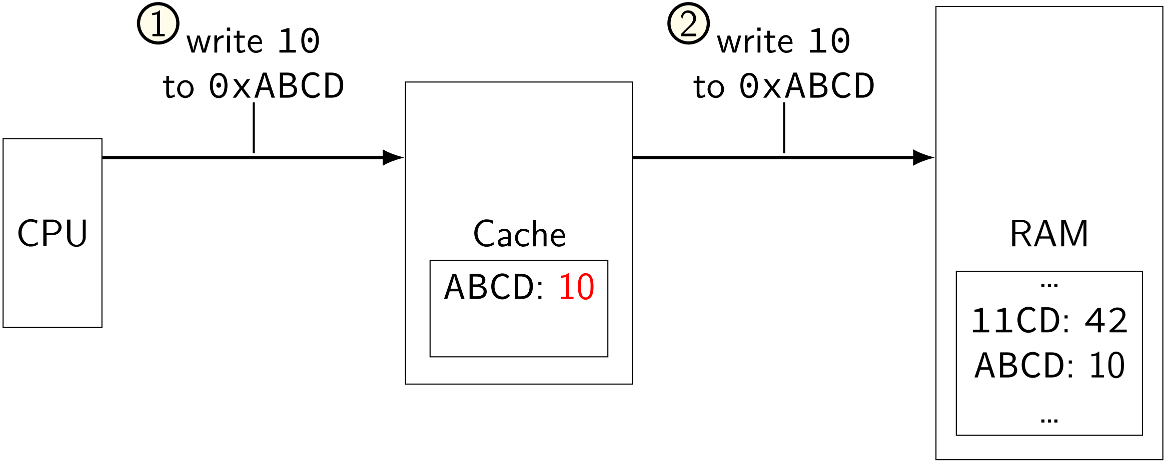
write-through v. write-back

option 1: write-through



write-through v. write-back

option 1: write-through



① write `10` to `0xABCD`

② write `10` to `0xABCD`

CPU

Cache

ABCD: 10

RAM

…

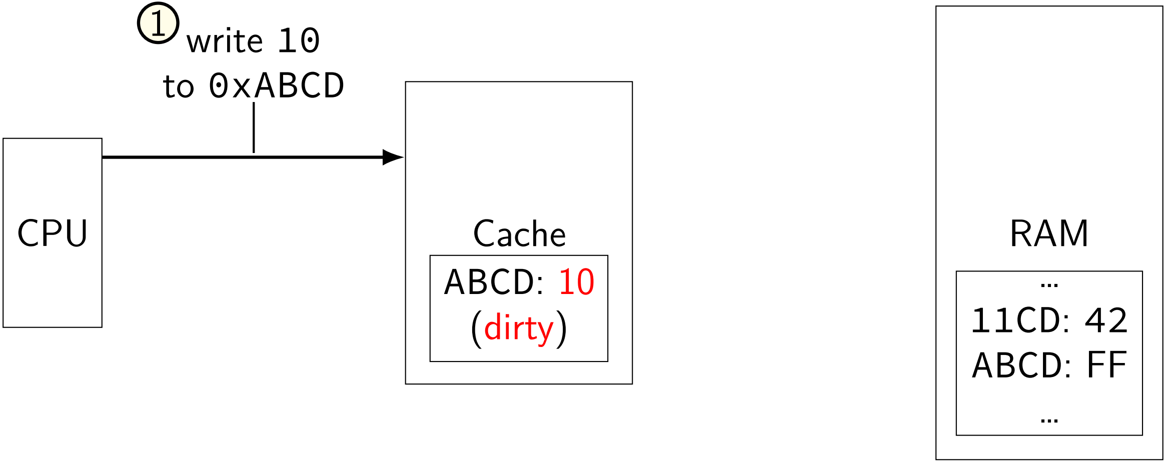
11CD: 42

ABCD: 10

…

write-through v. write-back

option 2: write-back



① write `10` to `0xABCD`

CPU

Cache

ABCD: 10 (dirty)

RAM

…

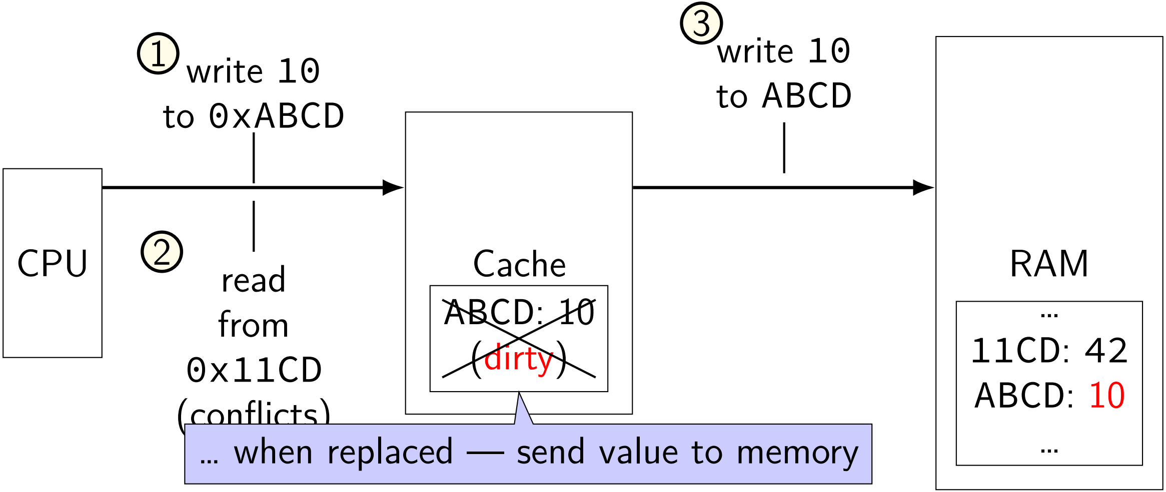
11CD: 42

ABCD: FF

…

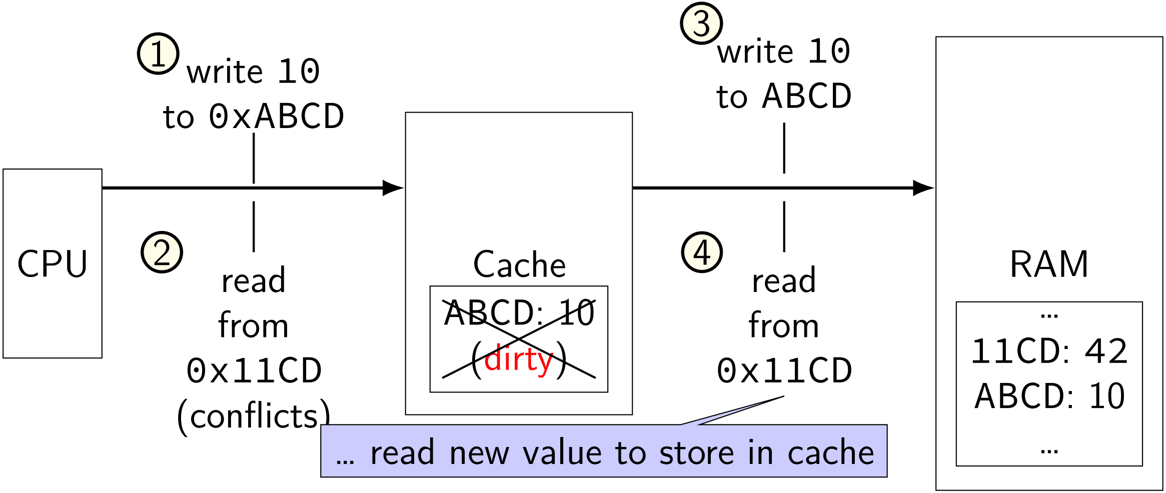
write-through v. write-back

option 2: write-back



… when replaced — send value to memory

write-through v. write-back



… read new value to store in cache

writeback policy

changed value!

2-way set associative, 4 byte blocks, 2 sets

index	valid	tag	value	dirty	valid	tag	value	dirty	LRU
0	1	`000000`	`mem[0x00]` `mem[0x01]`	0	1	`011000`	`mem[0x60]*` `mem[0x61]*`	1	1
1	1	`011000`	`mem[0x62]` `mem[0x63]`	0	0				0

1 = dirty (different than memory)
needs to be written if evicted

allocate on write?

processor writes less than whole cache block

block not yet in cache

two options:

write-allocate

- fetch rest of cache block, replace written part

write-no-allocate

- send write through to memory
- guess: not read soon?

write-allocate

2-way set associative, LRU, writeback

index	valid	tag	value	dirty	valid	tag	value	dirty	LRU
0	1	`000000`	`mem[0x00]` `mem[0x01]`	0	1	`011000`	`mem[0x60]`* `mem[0x61]`*	1	1
1	1	`011000`	`mem[0x62]` `mem[0x63]`	0	0				0

writing 0xFF into address `0x04`?
index 0, tag 000001

write-allocate

2-way set associative, LRU, writeback

index	valid	tag	value	dirty	valid	tag	value	dirty	LRU
0	1	000000	mem[0x00] mem[0x01]	0	1	011000	mem[0x60]⋆ mem[0x61]⋆	1	1
1	1	011000	mem[0x62] mem[0x63]	0	0				0

writing 0xFF into address `0x04`?

index 0, tag 000001

step 1: find least recently used block

write-allocate

2-way set associative, LRU, writeback

index	valid	tag	value	dirty	valid	tag	value	dirty	LRU
0	1	`000000`	`mem[0x00]` `mem[0x01]`	0	1	`011000`	`mem[0x60]`⋆ `mem[0x61]`⋆	~~1~~	1
1	1	`011000`	`mem[0x62]` `mem[0x63]`	0	0				0

writing 0xFF into address `0x04`?

index 0, tag 000001

step 1: find least recently used block

step 2: possibly writeback old block

write-allocate

2-way set associative, LRU, writeback

index	valid	tag	value	dirty	valid	tag	value	dirty	LRU
0	1	000000	mem[0x00] mem[0x01]	0	1	011000	0xFF mem[0x05]	1	0
1	1	011000	mem[0x62] mem[0x63]	0	0				0

writing 0xFF into address `0x04`?

index 0, tag 000001

step 1: find least recently used block

step 2: possibly writeback old block

step 3a: read in new block – to get mem[0x05]

step 3b: update LRU information

write-no-allocate

2-way set associative, LRU, writeback

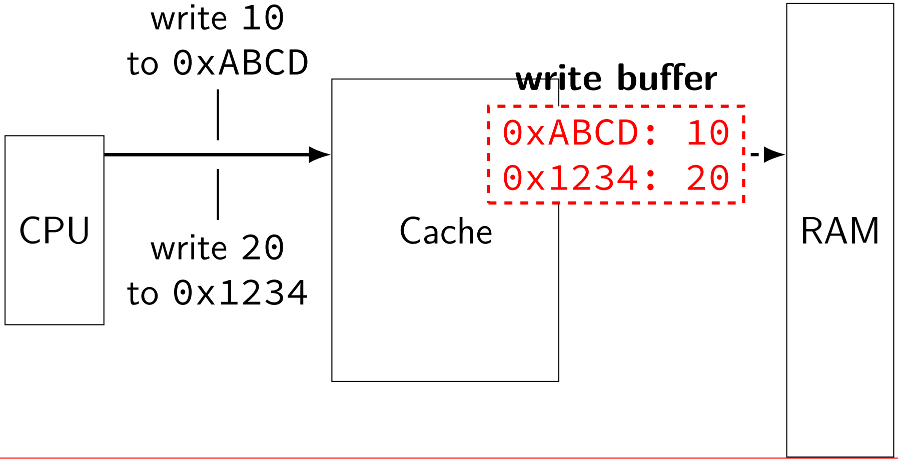
index	valid	tag	value	dirty	valid	tag	value	dirty	LRU
0	1	000000	mem[0x00] mem[0x01]	0	1	011000	mem[0x60]⋆ mem[0x61]⋆	1	1
1	1	011000	mem[0x62] mem[0x63]	0	0				0

writing 0xFF into address `0x04`?

step 1: is it in cache yet?

step 2: no, just send it to memory

fast writes



write appears to complete immediately when placed in buffer
memory can be much slower

cache organization and miss rate

depends on program; one example:

SPEC CPU2000 benchmarks, 64B block size

LRU replacement policies

data cache miss rates:

Cache size	direct-mapped	2-way	8-way	fully assoc.
1KB	8.63%	6.97%	5.63%	5.34%
2KB	5.71%	4.23%	3.30%	3.05%
4KB	3.70%	2.60%	2.03%	1.90%
16KB	1.59%	0.86%	0.56%	0.50%
64KB	0.66%	0.37%	0.10%	0.001%
128KB	0.27%	0.001%	0.0006%	0.0006%

Data: Cantin and Hill, "Cache Performance for SPEC CPU2000 Benchmarks"
`http://research.cs.wisc.edu/multifacet/misc/spec2000cache-data/`

cache organization and miss rate

depends on program; one example:

SPEC CPU2000 benchmarks, 64B block size

LRU replacement policies

data cache miss rates:

Cache size	direct-mapped	2-way	8-way	fully assoc.
1KB	8.63%	6.97%	5.63%	5.34%
2KB	5.71%	4.23%	3.30%	3.05%
4KB	3.70%	2.60%	2.03%	1.90%
16KB	1.59%	0.86%	0.56%	0.50%
64KB	0.66%	0.37%	0.10%	0.001%
128KB	0.27%	0.001%	0.0006%	0.0006%

Data: Cantin and Hill, "Cache Performance for SPEC CPU2000 Benchmarks"
`http://research.cs.wisc.edu/multifacet/misc/spec2000cache-data/`

reasoning about cache performance

hit time: time to lookup and find value in cache

  L1 cache — typically 1 cycle?

miss rate: portion of hits (value in cache)

miss penalty: extra time to get value if there's a miss

  time to access next level cache or memory

miss time: hit time + miss penalty

average memory access time

$$\text{AMAT} = \text{hit time} + \text{miss penalty} \times \text{miss rate}$$

effective speed of memory

making any cache look bad

1. access enough blocks, to fill the cache
2. access an additional block, replacing something
3. access last block replaced
4. access last block replaced
5. access last block replaced

…

but — typical real programs have locality

cache optimizations

	miss rate	hit time	miss penalty
increase cache size	better	worse	—
increase associativity	better	worse	worse?
increase block size	depends	worse	worse
add secondary cache	—	—	better
write-allocate	better	—	worse?
writeback	???	—	worse?
LRU replacement	better	?	worse?

$$\text{average time} = \text{hit time} + \text{miss rate} \times \text{miss penalty}$$

cache optimizations by miss type

	capacity	conflict	compulsory
increase cache size	fewer misses	fewer misses	—
increase associativity	—	fewer misses	—
increase block size	—	more misses	fewer misses

(assuming other listed parameters remain constant)

exercise (1)

initial cache: 64-byte blocks, 64 sets, 8 ways/set

If we leave the other parameters listed above unchanged, which will probably reduce the number of capacity misses in a typical program? (Multiple may be correct.)

A. quadrupling the block size (256-byte blocks, 64 sets, 8 ways/set)
B. quadrupling the number of sets
C. quadrupling the number of ways/set

exercise (2)

initial cache: 64-byte blocks, 8 ways/set, 64KB cache

If we leave the other parameters listed above unchanged, which will probably reduce the number of capacity misses in a typical program? (Multiple may be correct.)

A. quadrupling the block size (256-byte block, 8 ways/set, 64KB cache
B. quadrupling the number of ways/set
C. quadrupling the cache size

exercise (3)

initial cache: 64-byte blocks, 8 ways/set, 64KB cache

If we leave the other parameters listed above unchanged, which will probably reduce the number of conflict misses in a typical program? (Multiple may be correct.)

A. quadrupling the block size (256-byte block, 8 ways/set, 64KB cach
B. quadrupling the number of ways/set
C. quadrupling the cache size

C and cache misses (1)

```
int array[1024]; // 4KB array
int even_sum = 0, odd_sum = 0;
for (int i = 0; i < 1024; i += 2) {
    even_sum += array[i + 0];
    odd_sum +=  array[i + 1];
}
```

Assume everything but `array` is kept in registers (and the compiler does not do anything funny).

How many *data cache misses* on a 2KB direct-mapped cache with 16B cache blocks?

C and cache misses (2)

```
int array[1024]; // 4KB array
int even_sum = 0, odd_sum = 0;
for (int i = 0; i < 1024; i += 2)
    even_sum += array[i + 0];
for (int i = 1; i < 1024; i += 2)
    odd_sum +=  array[i + 1];
```

Assume everything but `array` is kept in registers (and the compiler does not do anything funny).

How many *data cache misses* on a 2KB direct-mapped cache with 16B cache blocks? Would a set-associtiave cache be better?

thinking about cache storage (1)

2KB direct-mapped cache with 16B blocks —

set 0: address 0 to 15, (0 to 15) + 2KB, (0 to 15) + 4KB, …

set 1: address 16 to 31, (16 to 31) + 2KB, (16 to 31) + 4KB, …

…

set 127: address 2032 to 2047, (2032 to 2047) + 2KB, …

thinking about cache storage (1)

2KB direct-mapped cache with 16B blocks —

set 0: address 0 to 15, (0 to 15) + 2KB, (0 to 15) + 4KB, …

set 1: address 16 to 31, (16 to 31) + 2KB, (16 to 31) + 4KB, …

…

set 127: address 2032 to 2047, (2032 to 2047) + 2KB, …

thinking about cache storage (1)

2KB direct-mapped cache with 16B blocks —

set 0: address 0 to 15, (0 to 15) + 2KB, (0 to 15) + 4KB, …

block at 0: array[0] through array[3]

set 1: address 16 to 31, (16 to 31) + 2KB, (16 to 31) + 4KB, …

block at 16: array[4] through array[7]

…

set 127: address 2032 to 2047, (2032 to 2047) + 2KB, …

block at 2032: array[508] through array[511]

thinking about cache storage (1)

2KB direct-mapped cache with 16B blocks —

set 0: address 0 to 15, (0 to 15) + 2KB, (0 to 15) + 4KB, …

- block at 0: array[0] through array[3]
- block at 0+2KB: array[512] through array[515]

set 1: address 16 to 31, (16 to 31) + 2KB, (16 to 31) + 4KB, …

- block at 16: array[4] through array[7]
- block at 16+2KB: array[516] through array[519]

…

set 127: address 2032 to 2047, (2032 to 2047) + 2KB, …

- block at 2032: array[508] through array[511]
- block at 2032+2KB: array[1020] through array[1023]

thinking about cache storage (2)

2KB 2-way set associative cache with 16B blocks: block addresses

---

set 0: address 0, 0 + 2KB, 0 + 4KB, …

set 1: address 16, 16 + 2KB, 16 + 4KB, …

…

set 63: address 1008, 2032 + 2KB, 2032 + 4KB …

thinking about cache storage (2)

2KB 2-way set associative cache with 16B blocks: block addresses

—

set 0: address 0, 0 + 2KB, 0 + 4KB, …

block at 0: array[0] through array[3]

set 1: address 16, 16 + 2KB, 16 + 4KB, …

address 16: array[4] through array[7]

…

set 63: address 1008, 2032 + 2KB, 2032 + 4KB …

address 1008: array[252] through array[255]

thinking about cache storage (2)

2KB 2-way set associative cache with 16B blocks: block addresses

—

set 0: address 0, 0 + 2KB, 0 + 4KB, ...
- block at 0: array[0] through array[3]
- block at 0+1KB: array[256] through array[259]
- block at 0+2KB: array[512] through array[515]
- ...

set 1: address 16, 16 + 2KB, 16 + 4KB, ...
- address 16: array[4] through array[7]

...

set 63: address 1008, 2032 + 2KB, 2032 + 4KB ...
- address 1008: array[252] through array[255]

thinking about cache storage (2)

2KB 2-way set associative cache with 16B blocks: block addresses —

set 0: address 0, 0 + 2KB, 0 + 4KB, …

block at 0: array[0] through array[3]
block at 0+1KB: array[256] through array[259]
block at 0+2KB: array[512] through array[515]
…

set 1: address 16, 16 + 2KB, 16 + 4KB, …

address 16: array[4] through array[7]

…

set 63: address 1008, 2032 + 2KB, 2032 + 4KB …

address 1008: array[252] through array[255]

C and cache misses (3)

```c
typedef struct {
    int a_value, b_value;
    int boring_values[126];
} item;
item items[8]; // 4 KB array
int a_sum = 0, b_sum = 0;
for (int i = 0; i < 8; ++i)
    a_sum += items[i].a_value;
for (int i = 0; i < 8; ++i)
    b_sum += items[i].b_value;
```

Assume everything but `items` is kept in registers (and the compiler does not do anything funny).

How many *data cache misses* on a 2KB direct-mapped cache with 16B cache blocks?

C and cache misses (3, rewritten?)

```
item array[1024]; // 4 KB array
int a_sum = 0, b_sum = 0;
for (int i = 0; i < 1024; i += 128)
    a_sum += array[i];
for (int i = 1; i < 1024; i += 128)
    b_sum += array[i];
```

C and cache misses (4)

```
typedef struct {
    int a_value, b_value;
    int boring_values[126];
} item;
item items[8]; // 4 KB array
int a_sum = 0, b_sum = 0;
for (int i = 0; i < 8; ++i)
    a_sum += items[i].a_value;
for (int i = 0; i < 8; ++i)
    b_sum += items[i].b_value;
```

Assume everything but `items` is kept in registers (and the compiler does not do anything funny).

How many *data cache misses* on a 4-way set associative 2KB direct-mapped cache with 16B cache blocks?

a note on matrix storage

A — $N \times N$ matrix

represent as array

makes dynamic sizes easier:

```
float A_2d_array[N][N];
float *A_flat = malloc(N * N);
```

```
A_flat[i * N + j] === A_2d_array[i][j]
```

matrix squaring

$$B_{ij} = \sum_{k=1}^{n} A_{ik} \times A_{kj}$$

```
/* version 1: inner loop is k, middle is j */
for (int i = 0; i < N; ++i)
  for (int j = 0; j < N; ++j)
    for (int k = 0; k < N; ++k)
      B[i * N + j] += A[i * N + k] * A[k * N + j];
```

matrix squaring

$$B_{ij} = \sum_{k=1}^{n} A_{ik} \times A_{kj}$$

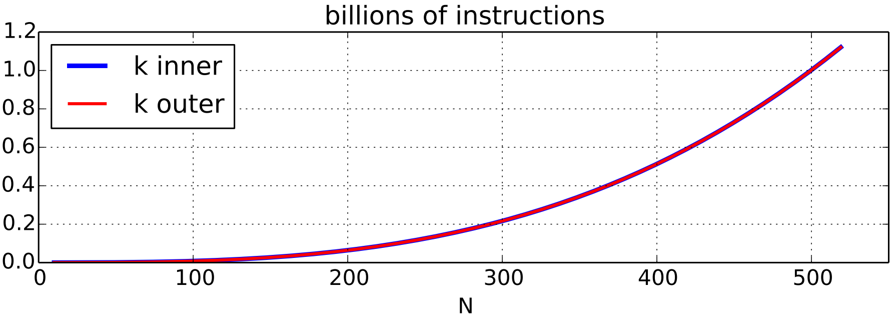
```
/* version 1: inner loop is k, middle is j*/
for (int i = 0; i < N; ++i)
  for (int j = 0; j < N; ++j)
    for (int k = 0; k < N; ++k)
      B[i*N+j] += A[i * N + k] * A[k * N + j];

/* version 2: outer loop is k, middle is i */
for (int k = 0; k < N; ++k)
  for (int i = 0; i < N; ++i)
    for (int j = 0; j < N; ++j)
      B[i*N+j] += A[i * N + k] * A[k * N + j];
```

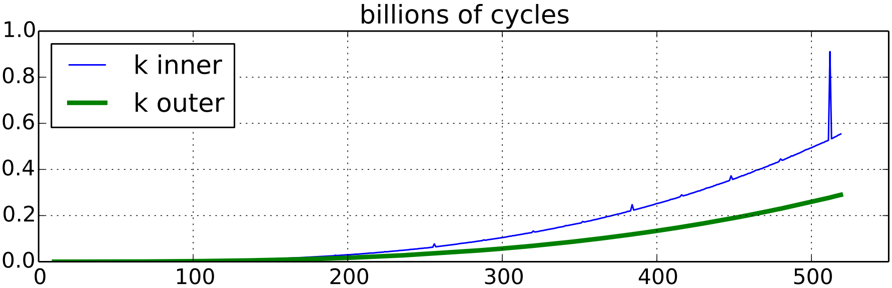
matrix squaring

$$B_{ij} = \sum_{k=1}^{n} A_{ik} \times A_{kj}$$

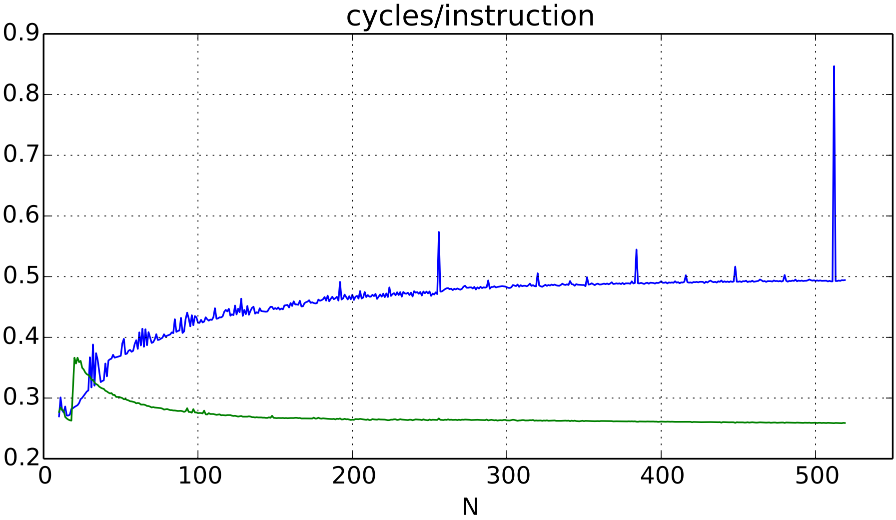
```
/* version 1: inner loop is k, middle is j*/
for (int i = 0; i < N; ++i)
  for (int j = 0; j < N; ++j)
    for (int k = 0; k < N; ++k)
      B[i*N+j] += A[i * N + k] * A[k * N + j];

/* version 2: outer loop is k, middle is i */
for (int k = 0; k < N; ++k)
  for (int i = 0; i < N; ++i)
    for (int j = 0; j < N; ++j)
      B[i*N+j] += A[i * N + k] * A[k * N + j];
```

performance

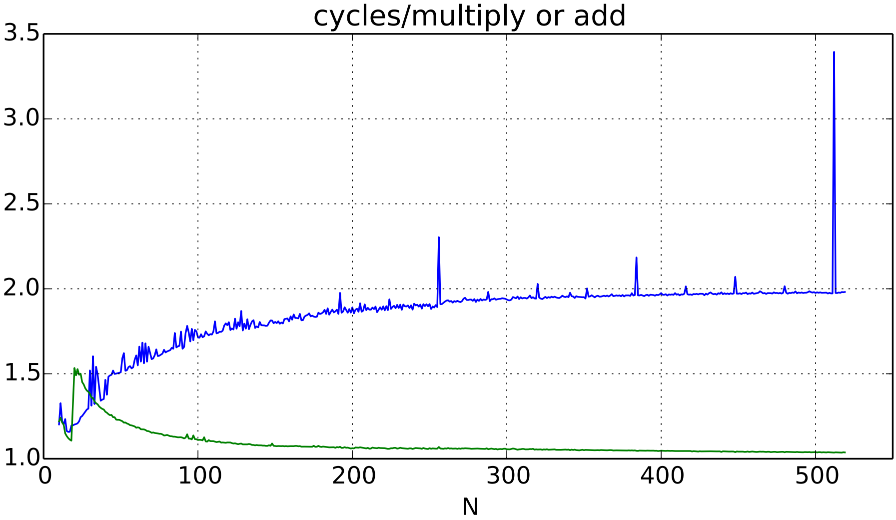

billions of instructions


billions of cycles

alternate view 1: cycles/instruction



alternate view 2: cycles/operation



loop orders and locality

loop body: $B_{ij}+=A_{ik}A_{kj}$

kij order: B_{ij}, A_{kj} have spatial locality

kij order: A_{ik} has temporal locality

… better than …

ijk order: A_{ik} has spatial locality

ijk order: B_{ij} has temporal locality

loop orders and locality

loop body: $B_{ij} + = A_{ik} A_{kj}$

kij order: B_{ij}, A_{kj} have spatial locality

kij order: A_{ik} has temporal locality

… better than …

ijk order: A_{ik} has spatial locality

ijk order: B_{ij} has temporal locality

matrix squaring

$$B_{ij} = \sum_{k=1}^{n} A_{ik} \times A_{kj}$$

```
/* version 1: inner loop is k, middle is j*/
for (int i = 0; i < N; ++i)
  for (int j = 0; j < N; ++j)
    for (int k = 0; k < N; ++k)
      B[i*N+j] += A[i * N + k] * A[k * N + j];

/* version 2: outer loop is k, middle is i */
for (int k = 0; k < N; ++k)
  for (int i = 0; i < N; ++i)
    for (int j = 0; j < N; ++j)
      B[i*N+j] += A[i * N + k] * A[k * N + j];
```

matrix squaring

$$B_{ij} = \sum_{k=1}^{n} A_{ik} \times A_{kj}$$

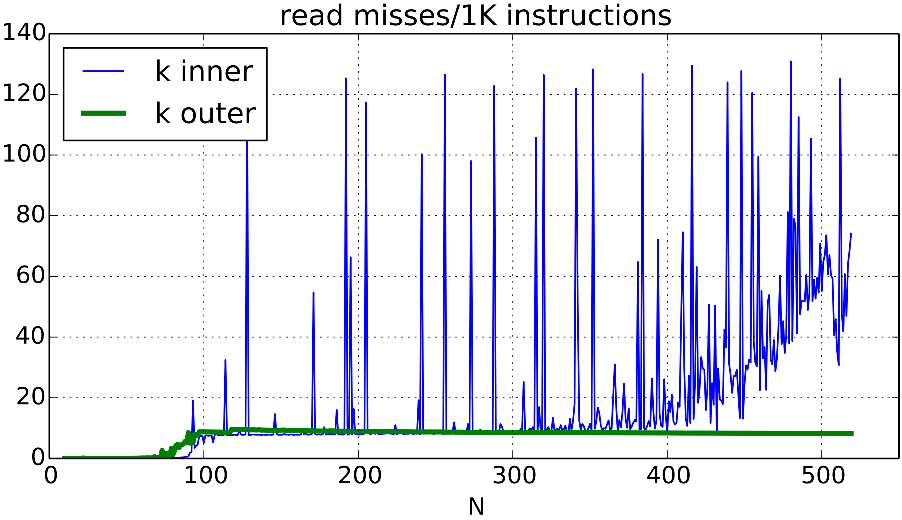
```
/* version 1: inner loop is k, middle is j*/
for (int i = 0; i < N; ++i)
  for (int j = 0; j < N; ++j)
    for (int k = 0; k < N; ++k)
      B[i*N+j] += A[i * N + k] * A[k * N + j];

/* version 2: outer loop is k, middle is i */
for (int k = 0; k < N; ++k)
  for (int i = 0; i < N; ++i)
    for (int j = 0; j < N; ++j)
      B[i*N+j] += A[i * N + k] * A[k * N + j];
```

matrix squaring

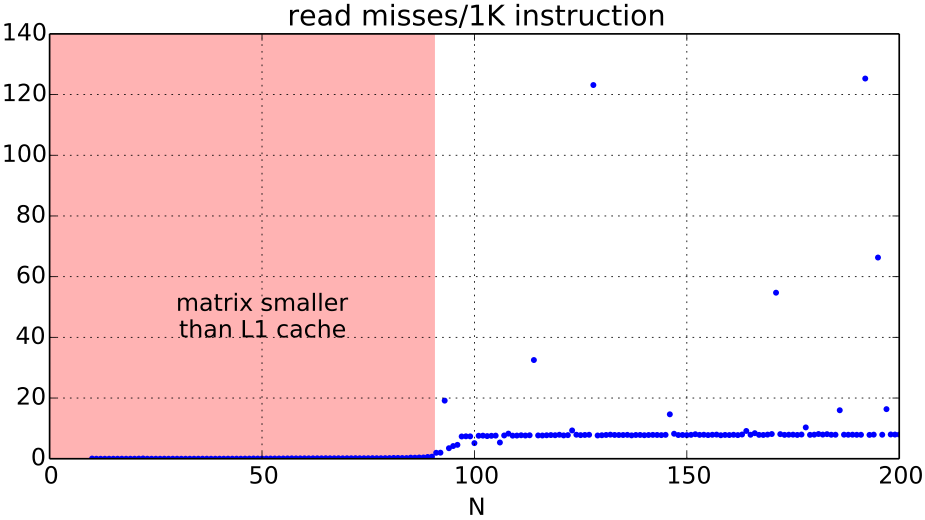
$$B_{ij} = \sum_{k=1}^{n} A_{ik} \times A_{kj}$$

```
/* version 1: inner loop is k, middle is j*/
for (int i = 0; i < N; ++i)
  for (int j = 0; j < N; ++j)
    for (int k = 0; k < N; ++k)
      B[i*N+j] += A[i * N + k] * A[k * N + j];

/* version 2: outer loop is k, middle is i */
for (int k = 0; k < N; ++k)
  for (int i = 0; i < N; ++i)
    for (int j = 0; j < N; ++j)
      B[i*N+j] += A[i * N + k] * A[k * N + j];
```

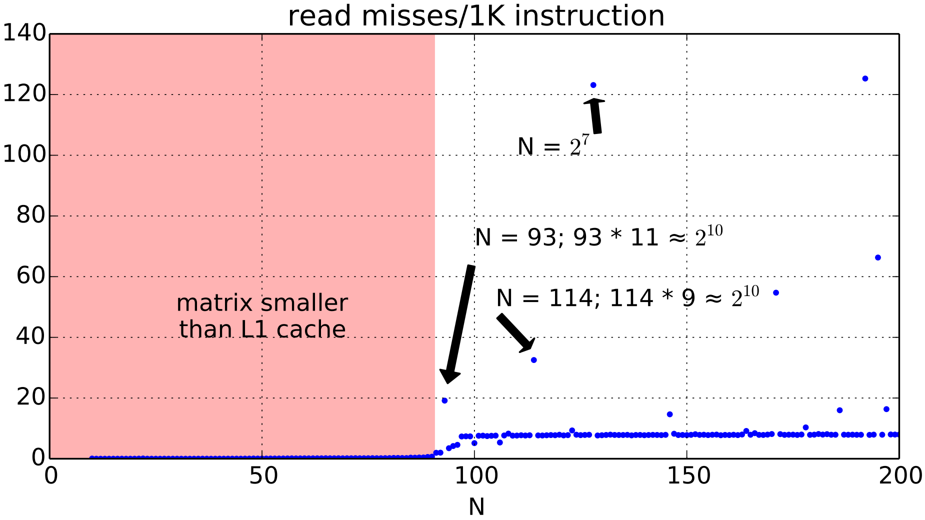
L1 misses


read misses/1K instructions
k inner
k outer
140
120
100
80
60
40
20
0
0
100
200
300
400
500
N

L1 miss detail (1)



L1 miss detail (2)



addresses

`A[k*114+j]` is at `10 0000 0000 0100`
`A[k*114+j+1]` is at `10 0000 0000 1000`
`A[(k+1)*114+j]` is at `10 0011 1001 0100`
`A[(k+2)*114+j]` is at `10 0101 0101 1100`
…
`A[(k+9)*114+j]` is at `11 0000 0000 1100`

addresses

`A[k*114+j]` is at `10` **`0000 00`**`00 0100`
`A[k*114+j+1]` is at `10` **`0000 00`**`00 1000`
`A[(k+1)*114+j]` is at `10` **`0011 10`**`01 0100`
`A[(k+2)*114+j]` is at `10` **`0101 01`**`01 1100`
…
`A[(k+9)*114+j]` is at `11` **`0000 00`**`00 1100`

recall: 6 index bits, 6 block offset bits (L1)

conflict misses

powers of two — lower order bits unchanged

`A[k*93+j]` and `A[(k+11)*93+j]`:

- 1023 elements apart (4092 bytes; 63.9 cache blocks)

64 sets in L1 cache: usually maps to same set

`A[k*93+(j+1)]` will not be cached (next i loop)

even if in same block as `A[k*93+j]`

locality exercise (1)

```
/* version 1 */
for (int i = 0; i < N; ++i)
    for (int j = 0; j < N; ++j)
        A[i] += B[j] * C[i * N + j]

/* version 2 */
for (int j = 0; j < N; ++j)
    for (int i = 0; i < N; ++i)
        A[i] += B[j] * C[i * N + j];
```

exercise: which has better temporal locality in A? in B? in C?

how about spatial locality?

systematic approach

```
for (int k = 0; k < N; ++k) {
  for (int i = 0; i < N; ++i) {
    $A_{ik}$ loaded once in this loop:
    for (int j = 0; j < N; ++j)
      $B_{ij}$, $A_{kj}$ loaded each iteration (if $N$ big):
      B[i*N+j] += A[i*N+k] * A[k*N+j];
```

N^3 multiplies, N^3 adds

values from A_{ik} loaded N^2 times

values from A_{kj} loaded N^3 times

values from B_{ij} loaded N^3 times

net: about one load into cache per operatoin

keeping values in cache

can't *explicitly* ensure values are kept in cache

…but reusing values *effectively* does this

- cache will try to keep recently used values

cache optimization ideas: choose what's in the cache

- for thinking about it: load values explicitly
- for implementing it: access only values we want loaded

a transformation

```
for (int kk = 0; kk < N; kk += 2)
  for (int k = kk; k < kk + 2; ++k)
      for (int i = 0; i < N; i += 2)
        for (int j = 0; j < N; ++j)
          B[i*N+j] += A[i*N+k] * A[k*N+j];
```

split the loop over k — should be exactly the same

(assuming even N)

a transformation

```
for (int kk = 0; kk < N; kk += 2)
  for (int k = kk; k < kk + 2; ++k)
      for (int i = 0; i < N; i += 2)
        for (int j = 0; j < N; ++j)
          B[i*N+j] += A[i*N+k] * A[k*N+j];
```

split the loop over k — should be exactly the same

(assuming even N)

simple blocking

```
for (int kk = 0; kk < N; kk += 2)
  /* was here: for (int k = kk; k < kk + 2; ++k) */
    for (int i = 0; i < N; i += 2)
      for (int j = 0; j < N; ++j)
        /* load Aik, Aik+1 into cache and process: */
        for (int k = kk; k < kk + 2; ++k)
            B[i*N+j] += A[i*N+k] * A[k*N+j];
```

now reorder split loop — same calculations

simple blocking

```
for (int kk = 0; kk < N; kk += 2)
  /* was here: for (int k = kk; k < kk + 2; ++k) */
    for (int i = 0; i < N; i += 2)
      for (int j = 0; j < N; ++j)
        /* load Aik, Aik+1 into cache and process: */
        for (int k = kk; k < kk + 2; ++k)
            B[i*N+j] += A[i*N+k] * A[k*N+j];
```

now reorder split loop — same calculations

now handle B_{ij} for $k+1$ right after B_{ij} for k

(previously: $B_{i,j+1}$ for k right after B_{ij} for k)

simple blocking

```
for (int kk = 0; kk < N; kk += 2)
  /* was here: for (int k = kk; k < kk + 2; ++k) */
    for (int i = 0; i < N; i += 2)
      for (int j = 0; j < N; ++j)
        /* load Aik, Aik+1 into cache and process: */
        for (int k = kk; k < kk + 2; ++k)
            B[i*N+j] += A[i*N+k] * A[k*N+j];
```

now reorder split loop — same calculations

now handle B_{ij} for $k+1$ right after B_{ij} for k

(previously: $B_{i,j+1}$ for k right after B_{ij} for k)

simple blocking – expanded

```
for (int kk = 0; kk < N; kk += 2) {
  for (int i = 0; i < N; i += 2) {
    for (int j = 0; j < N; ++j) {
      /* process a "block" of 2 k values: */
      B[i*N+j] += A[i*N+kk+0] * A[(kk+0)*N+j];
      B[i*N+j] += A[i*N+kk+1] * A[(kk+1)*N+j];
    }
  }
}
```

simple blocking – expanded

```
for (int kk = 0; kk < N; kk += 2) {
  for (int i = 0; i < N; i += 2) {
    for (int j = 0; j < N; ++j) {
      /* process a "block" of 2 k values: */
      B[i*N+j] += A[i*N+kk+0] * A[(kk+0)*N+j];
      B[i*N+j] += A[i*N+kk+1] * A[(kk+1)*N+j];
    }
  }
}
```

Temporal locality in B_{ij}s

simple blocking – expanded

```
for (int kk = 0; kk < N; kk += 2) {
  for (int i = 0; i < N; i += 2) {
    for (int j = 0; j < N; ++j) {
      /* process a "block" of 2 k values: */
      B[i*N+j] += A[i*N+kk+0] * A[(kk+0)*N+j];
      B[i*N+j] += A[i*N+kk+1] * A[(kk+1)*N+j];
    }
  }
}
```

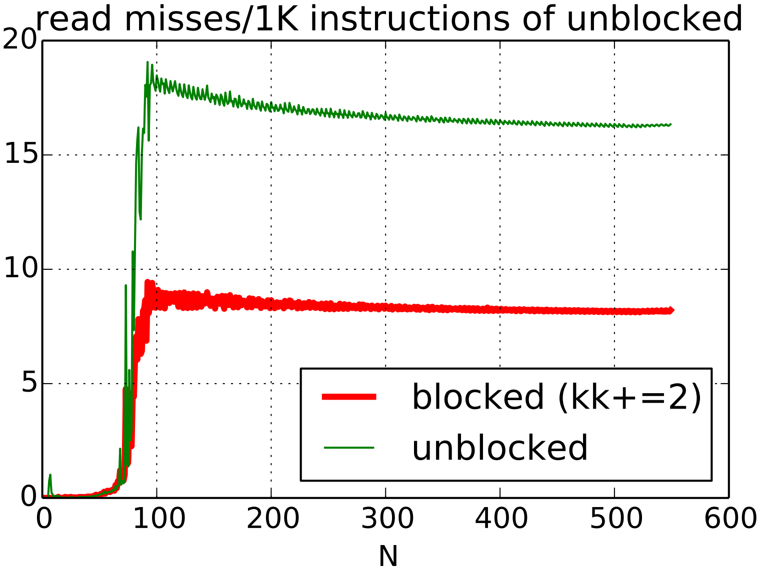
More spatial locality in A_{ik}

simple blocking – expanded

```
for (int kk = 0; kk < N; kk += 2) {
  for (int i = 0; i < N; i += 2) {
    for (int j = 0; j < N; ++j) {
      /* process a "block" of 2 k values: */
      B[i*N+j] += A[i*N+kk+0] * A[(kk+0)*N+j];
      B[i*N+j] += A[i*N+kk+1] * A[(kk+1)*N+j];
    }
  }
}
```

Still have good spatial locality in A_{kj}, B_{ij}

improvement in read misses



simple blocking (2)

same thing for i in addition to k?

```
for (int kk = 0; kk < N; kk += 2) {
  for (int ii = 0; ii < N; ii += 2) {
    for (int j = 0; j < N; ++j) {
      /* process a "block": */
      for (int k = kk; k < kk + 2; ++k)
        for (int i = 0; i < ii + 2; ++i)
            B[i*N+j] += A[i*N+k] * A[k*N+j];
    }
  }
}
```

simple blocking — expanded

```
for (int k = 0; k < N; k += 2) {
  for (int i = 0; i < N; i += 2) {
    /* load a block around Aik */
    for (int j = 0; j < N; ++j) {
      /* process a "block": */
      $B_{i+0,j}$   +=  $A_{i+0,k+0}$   *  $A_{k+0,j}$
      $B_{i+0,j}$   +=  $A_{i+0,k+1}$   *  $A_{k+1,j}$
      $B_{i+1,j}$   +=  $A_{i+1,k+0}$   *  $A_{k+0,j}$
      $B_{i+1,j}$   +=  $A_{i+1,k+1}$   *  $A_{k+1,j}$
    }
  }
}
```

simple blocking — expanded

```
for (int k = 0; k < N; k += 2) {
  for (int i = 0; i < N; i += 2) {
    /* load a block around Aik */
    for (int j = 0; j < N; ++j) {
      /* process a "block": */
      $B_{i+0,j}$ += $A_{i+0,k+0}$ * $A_{k+0,j}$
      $B_{i+0,j}$ += $A_{i+0,k+1}$ * $A_{k+1,j}$
      $B_{i+1,j}$ += $A_{i+1,k+0}$ * $A_{k+0,j}$
      $B_{i+1,j}$ += $A_{i+1,k+1}$ * $A_{k+1,j}$
    }
  }
}
```

Now A_{kj} reused in inner loop — more calculations per load!

generalizing cache blocking

```
for (int kk = 0; kk < N; kk += K) {
  for (int ii = 0; ii < N; ii += I) {
    with I by K block of A hopefully cached:
    for (int jj = 0; jj < N; jj += J) {
      with K by J block of A, I by J block of B cached:
      for i in ii to ii+I:
        for j in jj to jj+J:
          for k in kk to kk+K:
            B[i * N + j] += A[i * N + k]
                          * A[k * N + j];
```

B_{ij} used K times for one miss — N^2/K misses

A_{ik} used J times for one miss — N^2/J misses

A_{kj} used I times for one miss — N^2/I misses

catch: $IK + KJ + IJ$ elements must fit in cache

generalizing cache blocking

```
for (int kk = 0; kk < N; kk += K) {
  for (int ii = 0; ii < N; ii += I) {
    with I by K block of A hopefully cached:
    for (int jj = 0; jj < N; jj += J) {
      with K by J block of A, I by J block of B cached:
      for i in ii to ii+I:
        for j in jj to jj+J:
          for k in kk to kk+K:
            B[i * N + j] += A[i * N + k]
                          * A[k * N + j];
```

B_{ij} used K times for one miss — N^2/K misses

A_{ik} used J times for one miss — N^2/J misses

A_{kj} used I times for one miss — N^2/I misses

catch: $IK + KJ + IJ$ elements must fit in cache

generalizing cache blocking

```
for (int kk = 0; kk < N; kk += K) {
  for (int ii = 0; ii < N; ii += I) {
    with I by K block of A hopefully cached:
    for (int jj = 0; jj < N; jj += J) {
      with K by J block of A, I by J block of B cached:
      for i in ii to ii+I:
        for j in jj to jj+J:
          for k in kk to kk+K:
            B[i * N + j] += A[i * N + k]
                          * A[k * N + j];
```

B_{ij} used K times for one miss — N^2/K misses

A_{ik} used J times for one miss — N^2/J misses

A_{kj} used I times for one miss — N^2/I misses

catch: $IK + KJ + IJ$ elements must fit in cache

generalizing cache blocking

```
for (int kk = 0; kk < N; kk += K) {
  for (int ii = 0; ii < N; ii += I) {
    with I by K block of A hopefully cached:
    for (int jj = 0; jj < N; jj += J) {
      with K by J block of A, I by J block of B cached:
      for i in ii to ii+I:
        for j in jj to jj+J:
          for k in kk to kk+K:
            B[i * N + j] += A[i * N + k]
                          * A[k * N + j];
```

B_{ij} used K times for one miss — N^2/K misses

A_{ik} used J times for one miss — N^2/J misses

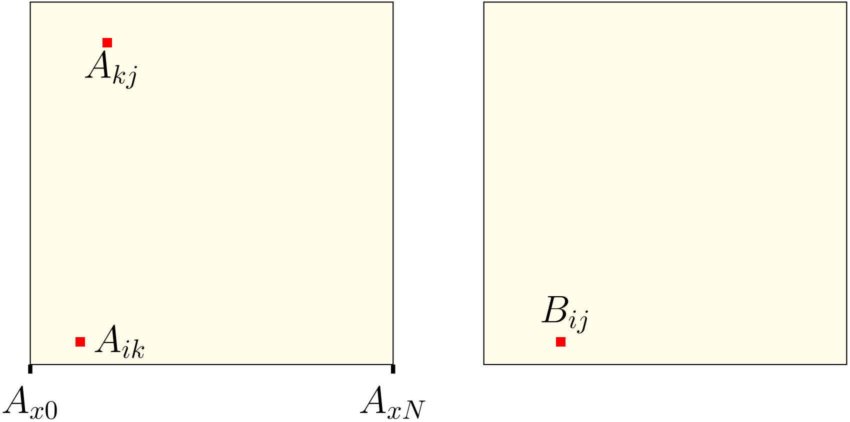
A_{kj} used I times for one miss — N^2/I misses

catch: $IK + KJ + IJ$ elements must fit in cache

view 2: divide and conquer

```
partial_square(float *A, float *B,
               int startI, int endI, ...) {
  for (int i = startI; i < endI; ++i) {
    for (int j = startJ; j < endJ; ++j) {
      ...
}
square(float *A, float *B, int N) {
  for (int ii = 0; ii < N; ii += BLOCK)
    ...
      /* segment of A, B in use fits in cache! */
      partial_square(
            A, B,
            ii, ii + BLOCK,
            jj, jj + BLOCK, ...);
}
```

array usage: kij order

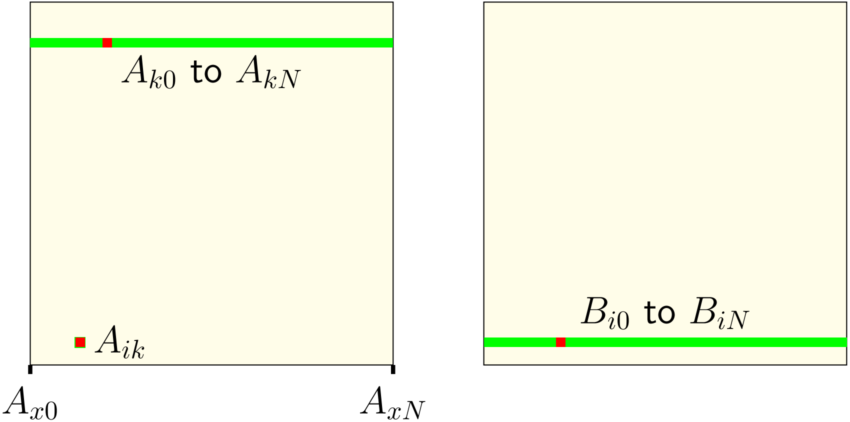


for all k: for all i: for all j: $B_{ij} += A_{ik} \times A_{kj}$

N calculations for A_{ik}

1 for A_{kj}, B_{ij}

array usage: kij order

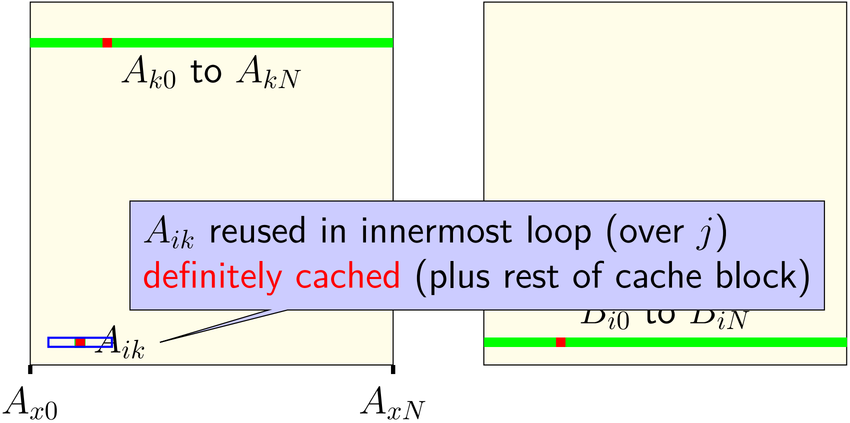


for all k: for all i: for all j: $B_{ij} += A_{ik} \times A_{kj}$

N calculations for A_{ik}

1 for A_{kj}, B_{ij}

array usage: kij order



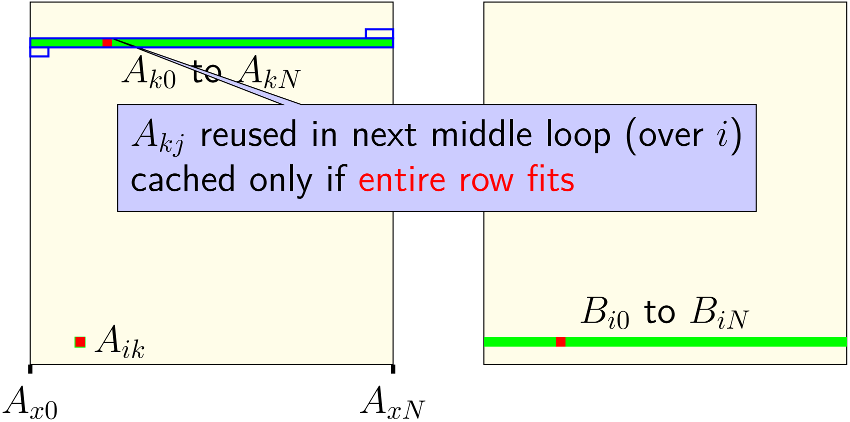
A_{ik} reused in innermost loop (over j)
definitely cached (plus rest of cache block)

for all k: for all i: for all j: $B_{ij} += A_{ik} \times A_{kj}$

N calculations for A_{ik}

1 for A_{kj}, B_{ij}

array usage: kij order



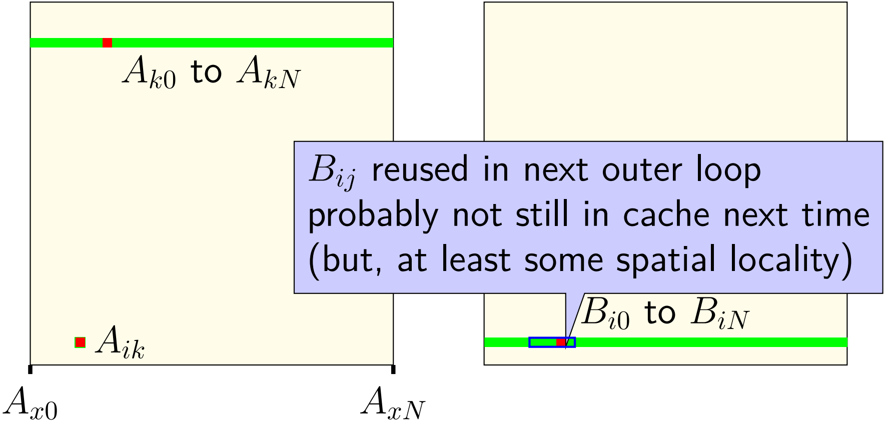
A_{kj} reused in next middle loop (over i)
cached only if entire row fits

for all k: for all i: for all j: $B_{ij} += A_{ik} \times A_{kj}$

N calculations for A_{ik}

1 for A_{kj}, B_{ij}

array usage: kij order



B_{ij} reused in next outer loop
probably not still in cache next time
(but, at least some spatial locality)

for all k: for all i: for all j: $B_{ij} += A_{ik} \times A_{kj}$

N calculations for A_{ik}

1 for A_{kj}, B_{ij}

inefficiencies

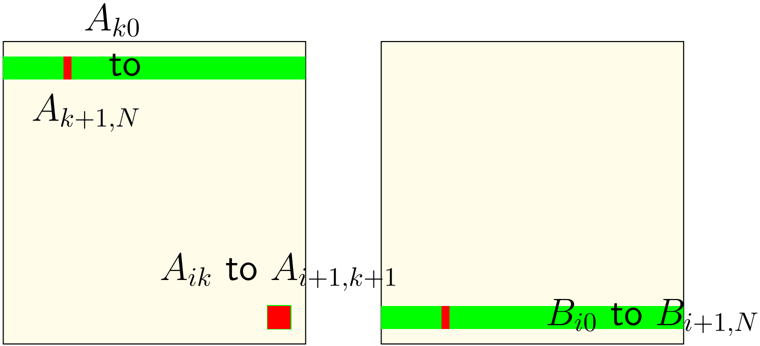
if a row doesn't fit in cache —
cache effectively holds one element

- everything else — too much other stuff between accesses

if a row does fit in cache —
cache effectively holds one row + one element

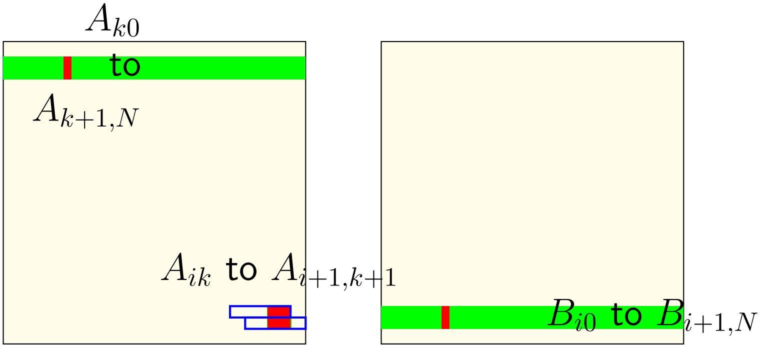
- everything else — too much other stuff between accesses

array usage (better)



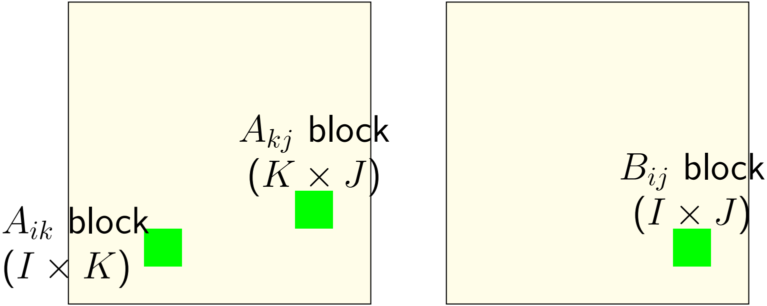
more temporal locality:
N calculations for each A_{ik}
2 calculations for each B_{ij} (for k, $k+1$)
2 calculations for each A_{kj} (for k, $k+1$)

array usage (better)



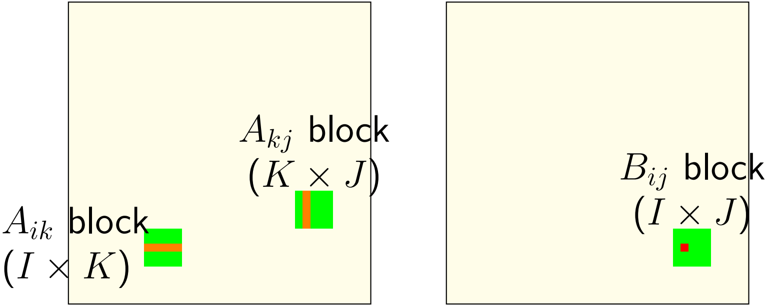
more spatial locality:
calculate on each $A_{i,k}$ and $A_{i,k+1}$ together
both in same cache block — same amount of cache loads

array usage: block



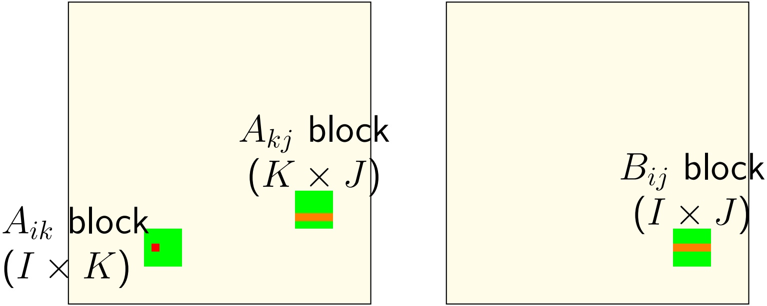
inner loop keeps “blocks” from A, B in cache

array usage: block



B_{ij} calculation uses strips from A
K calculations for one load (cache miss)

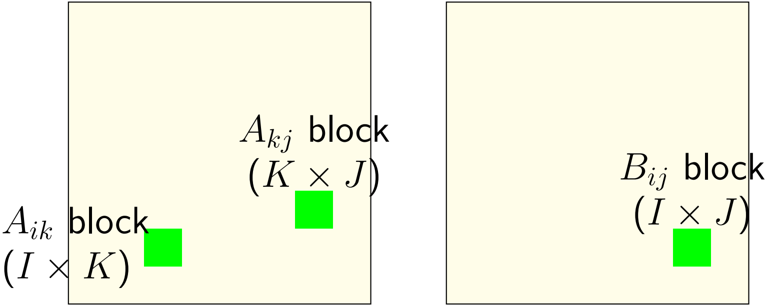
array usage: block



A_{ik} calculation uses strips from A, B

J calculations for one load (cache miss)

array usage: block



(approx.) KIJ fully cached calculations
for $KI + IJ + KJ$ loads
(assuming everything stays in cache)

cache blocking efficiency

load $I \times K$ elements of A_{ik}:

- do $> J$ multiplies with each

load $K \times J$ elements of A_{kj}:

- do I multiplies with each

load $I \times J$ elements of B_{ij}:

- do K adds with each

bigger blocks — more work per load!

catch: $IK + KJ + IJ$ elements must fit in cache

cache blocking rule of thumb

fill the most of the cache with useful data

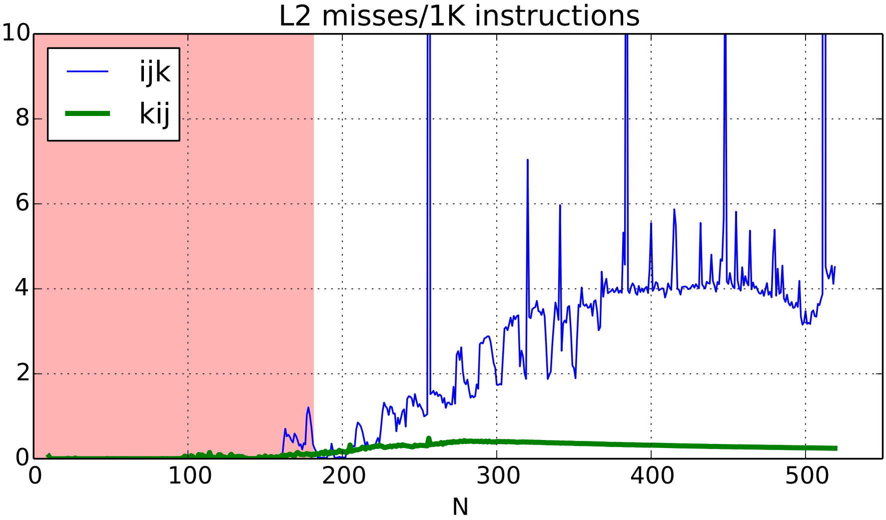
and do as much work as possible from that

example: my desktop 32KB L1 cache

$I = J = K = 48$ uses $48^2 \times 3$ elements, or 27KB.

assumption: conflict misses aren't important

L2 misses


L2 misses/1K instructions
10
8
6
4
2
0
0
100
200
300
400
500
N
ijk
kij

reasoning about loop orders

changing loop order changed locality

how do we tell which loop order will be best?

- besides running each one?

systematic approach (1)

```
for (int k = 0; k < N; ++k)
  for (int i = 0; i < N; ++i)
    for (int j = 0; j < N; ++j)
      B[i*N+j] += A[i*N+k] * A[k*N+j];
```

goal: get most out of each cache miss

if N is larger than the cache:

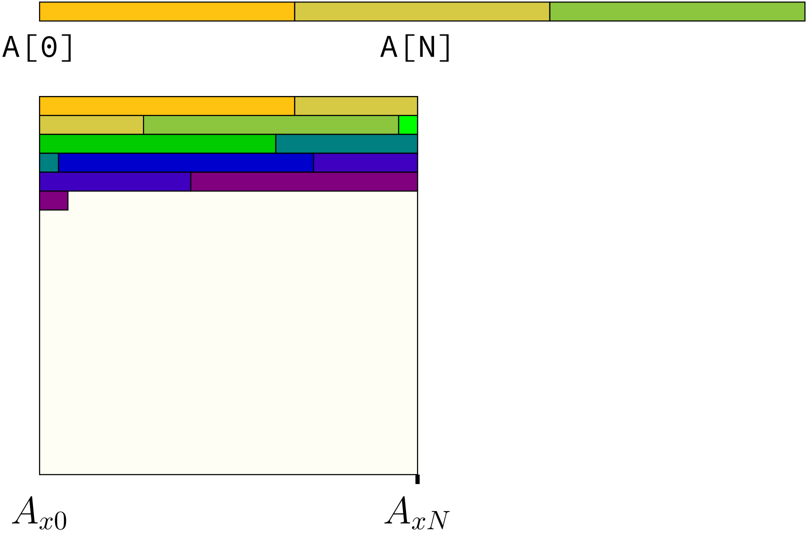
miss for B_{ij} — 1 comptuation

miss for A_{ik} — N computations

miss for A_{kj} — 1 computation

effectively caching just 1 element

‘flat’ 2D arrays and cache blocks



adding associativity

2-way set associative, 2 byte blocks, 2 sets

index	**valid**	**tag**	**value**	**valid**	**tag**	**value**
`0`	`0`			`0`		
`1`	`0`			`0`		

multiple places to put values with same index
avoid conflict misses

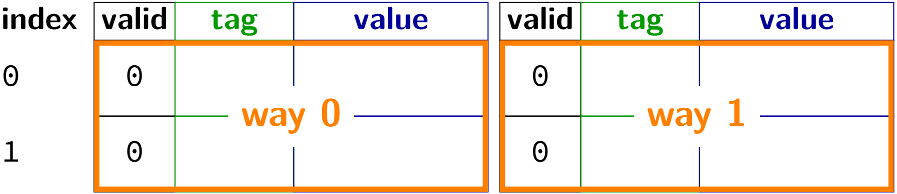
adding associativity

2-way set associative, 2 byte blocks, 2 sets

index	**valid**	**tag**	**value**	**valid**	**tag**	**value**
`0`	`0`		**set 0**	`0`		
`1`	`0`		**set 1**	`0`		

adding associativity

2-way set associative, 2 byte blocks, 2 sets



index	valid	tag	value	valid	tag	value
0	0			0		
1	0			0		

adding associativity

2-way set associative, 2 byte blocks, 2 sets

index	valid	tag	value	valid	tag	value
0	0			0		
1	0			0		

$m = 8$ bit addresses
$S = 2 = 2^s$ sets
$s = 1$ (set) index bits

$B = 2 = 2^b$ byte block size
$b = 1$ (block) offset bits
$t = m - (s + b) = 6$ tag bits

adding associativity

2-way set associative, 2 byte blocks, 2 sets

index	valid	tag	value	valid	tag	value
0	1	000000	mem[0x00] mem[0x01]	0		
1	0			0		

address (hex)	result
00000000 (00)	miss
00000001 (01)	
01100011 (63)	
01100001 (61)	
01100010 (62)	
00000000 (00)	
01100100 (64)	

tag index offset

adding associativity

2-way set associative, 2 byte blocks, 2 sets

index	valid	tag	value	valid	tag	value
0	1	000000	mem[0x00] mem[0x01]	0		
1	0			0		

address (hex)	result
00000000 (00)	miss
00000001 (01)	hit
01100011 (63)	
01100001 (61)	
01100010 (62)	
00000000 (00)	
01100100 (64)	

tag index offset

adding associativity

2-way set associative, 2 byte blocks, 2 sets

index	valid	tag	value	valid	tag	value
0	1	000000	mem[0x00] mem[0x01]	0		
1	1	011000	mem[0x62] mem[0x63]	0		

address (hex)	result
00000000 (00)	miss
00000001 (01)	hit
01100011 (63)	miss
01100001 (61)	
01100010 (62)	
00000000 (00)	
01100100 (64)	

tag index offset

adding associativity

2-way set associative, 2 byte blocks, 2 sets

index	valid	tag	value	valid	tag	value
0	1	000000	mem[0x00] mem[0x01]	1	011000	mem[0x60] mem[0x61]
1	1	011000	mem[0x62] mem[0x63]	0		

address (hex)	result
00000000 (00)	miss
00000001 (01)	hit
01100011 (63)	miss
01100001 (61)	miss
01100010 (62)	
00000000 (00)	
01100100 (64)	

tag index offset

adding associativity

2-way set associative, 2 byte blocks, 2 sets

index	valid	tag	value	valid	tag	value
0	1	000000	mem[0x00] mem[0x01]	1	011000	mem[0x60] mem[0x61]
1	1	011000	mem[0x62] mem[0x63]	0		

address (hex)	result
00000000 (00)	miss
00000001 (01)	hit
01100011 (63)	miss
01100001 (61)	miss
01100010 (62)	hit
00000000 (00)	
01100100 (64)	

tag index offset

adding associativity

2-way set associative, 2 byte blocks, 2 sets

index	valid	tag	value	valid	tag	value
0	1	000000	mem[0x00] mem[0x01]	1	011000	mem[0x60] mem[0x61]
1	1	011000	mem[0x62] mem[0x63]	0		

address (hex)	result
00000000 (00)	miss
00000001 (01)	hit
01100011 (63)	miss
01100001 (61)	miss
01100010 (62)	hit
00000000 (00)	hit
01100100 (64)	

tag index offset

adding associativity

2-way set associative, 2 byte blocks, 2 sets

index	valid	tag	value	valid	tag	value
0	1	000000	mem[0x00] mem[0x01]	1	011000	mem[0x60] mem[0x61]
1	1	011000	mem[0x62] mem[0x63]	0		

address (hex)	result
00000000 (00)	miss
00000001 (01)	hit
01100011 (63)	miss
01100001 (61)	miss
01100010 (62)	hit
00000000 (00)	hit
01100100 (64)	miss

tag index offset

needs to replace block in set 0!

adding associativity

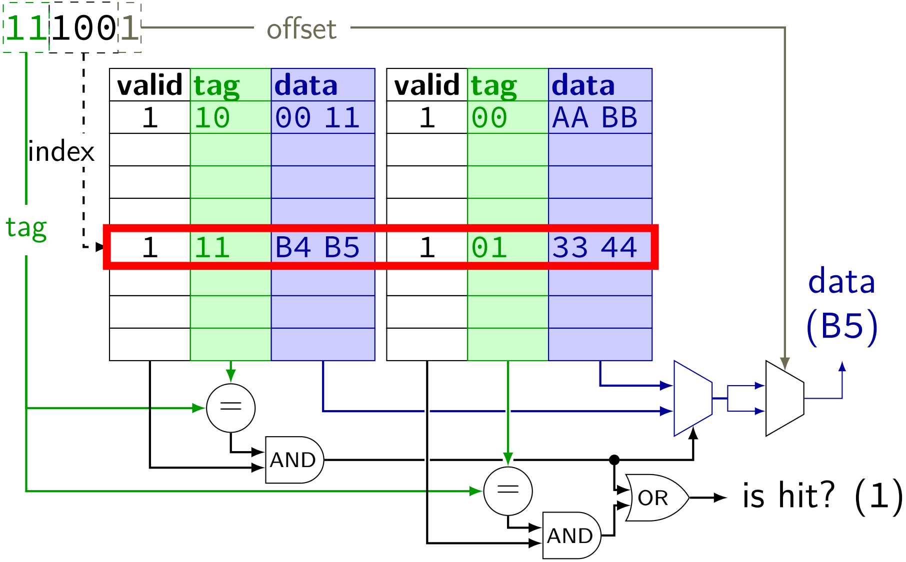
2-way set associative, 2 byte blocks, 2 sets

index	valid	tag	value	valid	tag	value
0	1	000000	mem[0x00] mem[0x01]	1	011000	mem[0x60] mem[0x61]
1	1	011000	mem[0x62] mem[0x63]	0		

address (hex)	result
00000000 (00)	miss
00000001 (01)	hit
01100011 (63)	miss
01100001 (61)	miss
01100010 (62)	hit
00000000 (00)	hit
01100100 (64)	miss

tag index offset

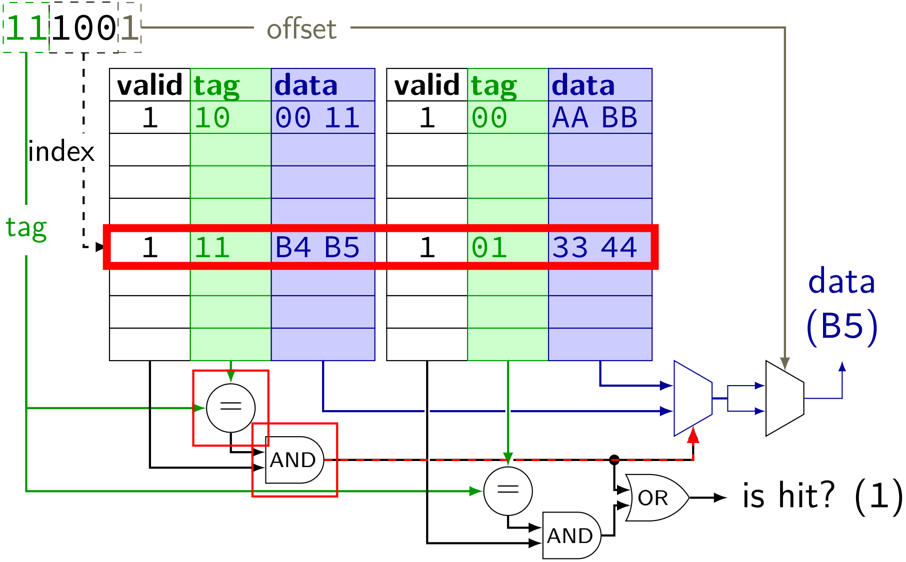
cache operation (associative)



valid	tag	data
1	10	00 11
1	11	B4 B5

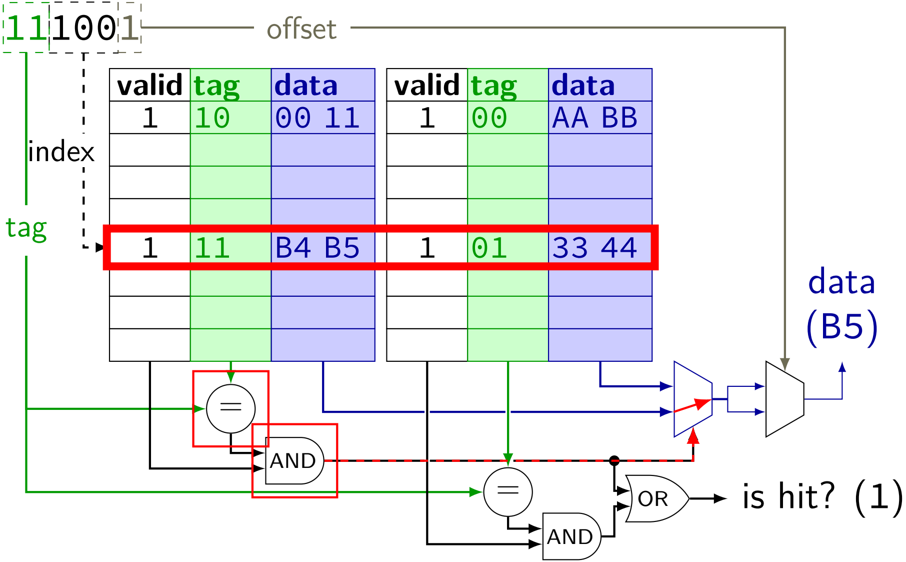
valid	tag	data
1	00	AA BB
1	01	33 44

cache operation (associative)



valid	tag	data	valid	tag	data
1	10	00 11	1	00	AA BB
1	11	B4 B5	1	01	33 44

cache operation (associative)



associative lookup possibilities

none of the blocks for the index are valid

none of the valid blocks for the index match the tag

- something else is stored there

one of the blocks for the index is valid and matches the tag