cache performance 2

last time

AMAT and multi-level caches

cache tradeoffs:

- which effects hit time/hit rate/miss penalty

compulsory (or cold)/conflict/capacity misses

counting misses based on C code

arrays 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?

arrays 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 = 0; 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?

set-associative not better?

2KB direct-mapped cache, 4B array elements: array[0 to 3] and array[512 to 515] map to same set

- 2KB apart — mapping “wraps around”
- four misses for this set
- access to array[0]+array[2], then array[512]+array[514], then array[1]+array[3], then array[513]+array[515]
- same for each of other 127 sets

2KB, 2-way set associative cache: array[0 to 3] and array[256 to 259] and array[512 to 515] and array[756 to 759] map to same set

- 1KB apart — shorter time until mapping wraps around
- eight misses + same for other 63 sets

approximate miss analysis

very tedious to precisely count cache misses

- even more tedious when we take advanced cache optimizations into account

instead, approximations:

good or bad temporal/spatial locality

- good temporal locality: value stays in cache
- good spatial locality: use all parts of cache block

with nested loops: what does inner loop use?

- intuition: values used in inner loop loaded into cache once
- (that is, once each time the inner loop is run)
- …if they can all fit in the cache

approximate miss analysis

very tedious to precisely count cache misses

- even more tedious when we take advanced cache optimizations into account

instead, approximations:

good or bad temporal/spatial locality

- good temporal locality: value stays in cache
- good spatial locality: use all parts of cache block

with nested loops: what does inner loop use?

- intuition: values used in inner loop loaded into cache once
- (that is, once each time the inner loop is run)
- …if they can all fit in the cache

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?

exercise: miss estimating (1)

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

Assume: 4 array elements per block, N very large, nothing in cache at beginning.

Example: $N/4$ estimated misses for A accesses:

- A[i] should always be hit on all but first iteration of inner-most loop.
- first iter: A[i] should be hit about $3/4$s of the time (same block as A[i-1] that often)

Exericse: estimate $\#$ of misses for B, C

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]
```

convertion re: rows/columns

going to call the first index rows

$A_{i,j}$ is A row i, column j

rows are stored together

this is an arbitrary choice

5x5 array and 4-element cache blocks

array[0*5 + 0]	array[0*5 + 1]	array[0*5 + 2]	array[0*5 + 3]	array[0*5 + 4]
array[1*5 + 0]	array[1*5 + 1]	array[1*5 + 2]	array[1*5 + 3]	array[1*5 + 4]
array[2*5 + 0]	array[2*5 + 1]	array[2*5 + 2]	array[2*5 + 3]	array[2*5 + 4]
array[3*5 + 0]	array[3*5 + 1]	array[3*5 + 2]	array[3*5 + 3]	array[3*5 + 4]
array[4*5 + 0]	array[4*5 + 1]	array[4*5 + 2]	array[4*5 + 3]	array[4*5 + 4]

5x5 array and 4-element cache blocks

array[0*5 + 0]	array[0*5 + 1]	array[0*5 + 2]	array[0*5 + 3]	array[0*5 + 4]
array[1*5 + 0]	array[1*5 + 1]	array[1*5 + 2]	array[1*5 + 3]	array[1*5 + 4]
array[2*5 + 0]	array[2*5 + 1]	array[2*5 + 2]	array[2*5 + 3]	array[2*5 + 4]
array[3*5 + 0]	array[3*5 + 1]	array[3*5 + 2]	array[3*5 + 3]	array[3*5 + 4]
array[4*5 + 0]	array[4*5 + 1]	array[4*5 + 2]	array[4*5 + 3]	array[4*5 + 4]

if array starts on cache block
first cache block = first elements
all together in one row!

5x5 array and 4-element cache blocks

array[0*5 + 0]	array[0*5 + 1]	array[0*5 + 2]	array[0*5 + 3]	array[0*5 + 4]
array[1*5 + 0]	array[1*5 + 1]	array[1*5 + 2]	array[1*5 + 3]	array[1*5 + 4]
array[2*5 + 0]	array[2*5 + 1]	array[2*5 + 2]	array[2*5 + 3]	array[2*5 + 4]
array[3*5 + 0]	array[3*5 + 1]	array[3*5 + 2]	array[3*5 + 3]	array[3*5 + 4]
array[4*5 + 0]	array[4*5 + 1]	array[4*5 + 2]	array[4*5 + 3]	array[4*5 + 4]

second cache block:
1 from row 0
3 from row 1

5x5 array and 4-element cache blocks

array[0*5 + 0]	array[0*5 + 1]	array[0*5 + 2]	array[0*5 + 3]	array[0*5 + 4]
array[1*5 + 0]	array[1*5 + 1]	array[1*5 + 2]	array[1*5 + 3]	array[1*5 + 4]
array[2*5 + 0]	array[2*5 + 1]	array[2*5 + 2]	array[2*5 + 3]	array[2*5 + 4]
array[3*5 + 0]	array[3*5 + 1]	array[3*5 + 2]	array[3*5 + 3]	array[3*5 + 4]
array[4*5 + 0]	array[4*5 + 1]	array[4*5 + 2]	array[4*5 + 3]	array[4*5 + 4]

5x5 array and 4-element cache blocks

array[0*5 + 0]	array[0*5 + 1]	array[0*5 + 2]	array[0*5 + 3]	array[0*5 + 4]
array[1*5 + 0]	array[1*5 + 1]	array[1*5 + 2]	array[1*5 + 3]	array[1*5 + 4]
array[2*5 + 0]	array[2*5 + 1]	array[2*5 + 2]	array[2*5 + 3]	array[2*5 + 4]
array[3*5 + 0]	array[3*5 + 1]	array[3*5 + 2]	array[3*5 + 3]	array[3*5 + 4]
array[4*5 + 0]	array[4*5 + 1]	array[4*5 + 2]	array[4*5 + 3]	array[4*5 + 4]

generally: cache blocks contain data from 1 or 2 rows
$\rightarrow$ better performance from reusing rows

matrix multiply

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

```
/* 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)
      C[i * N + j] += A[i * N + k] * B[k * N + j];
```

matrix multiply

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

```
/* 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)
      C[i*N+j] += A[i * N + k] * B[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)
      C[i*N+j] += A[i * N + k] * B[k * N + j];
```

which is better?

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

```
/* 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)
      C[i*N+j] += A[i * N + k] * B[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)
      C[i*N+j] += A[i * N + k] * B[k * N + j];
```

exercise: Which version has better spatial/temporal locality for...
...accesses to C? ...accesses to A? ...accesses to B?

loop orders and locality

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

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

kij order: A_{ik} has temporal locality

… better than …

ijk order: A_{ik} has spatial locality

ijk order: C_{ij} has temporal locality

loop orders and locality

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

$ki\textcolor{red}{j}$ order: $C_{i\textcolor{red}{j}}$, $B_{k\textcolor{red}{j}}$ have spatial locality

kij order: A_{ik} has temporal locality

… better than …

$ij\textcolor{red}{k}$ order: $A_{i\textcolor{red}{k}}$ has spatial locality

ijk order: C_{ij} has temporal locality

matrix multiply

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

```
/* 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)
      C[i*N+j] += A[i * N + k] * B[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)
      C[i*N+j] += A[i * N + k] * B[k * N + j];
```

matrix multiply

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

```
/* 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)
      C[i*N+j] += A[i * N + k] * B[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)
      C[i*N+j] += A[i * N + k] * B[k * N + j];
```

matrix multiply

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

```
/* 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)
      C[i*N+j] += A[i * N + k] * B[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)
      C[i*N+j] += A[i * N + k] * B[k * N + j];
```

counting misses: version 1

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

if N really large

- assumption: can't get close to storing N values in cache at once

for A: about $N \div$ block size misses per k-loop

- total misses: $N^3 \div$ block size

for B: about N misses per k-loop

- total misses: N^3

for C: about $1 \div$ block size miss per k-loop

- total misses: $N^2 \div$ block size

counting misses: version 2

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

for A: about 1 misses per j-loop

total misses: N^2

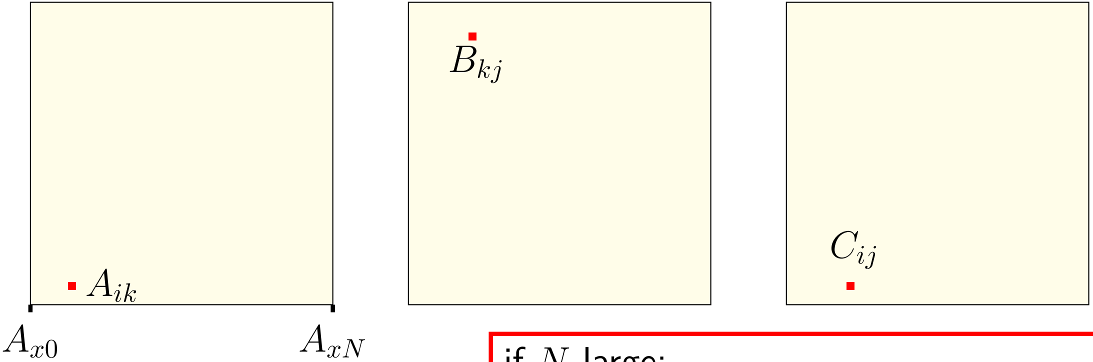
for B: about $N \div$ block size miss per j-loop

total misses: $N^3 \div$ block size

for C: about $N \div$ block size miss per j-loop

total misses: $N^3 \div$ block size

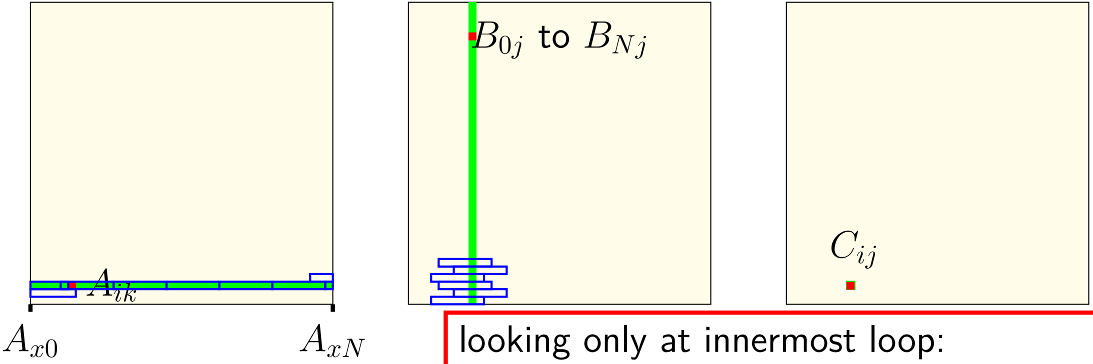
array usage: ijk order



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

if N large:
using C_{ij} many times per load into cache
using A_{ik} once per load-into-cache
(but using $A_{i,k+1}$ right after)
using B_{kj} once per load into cache

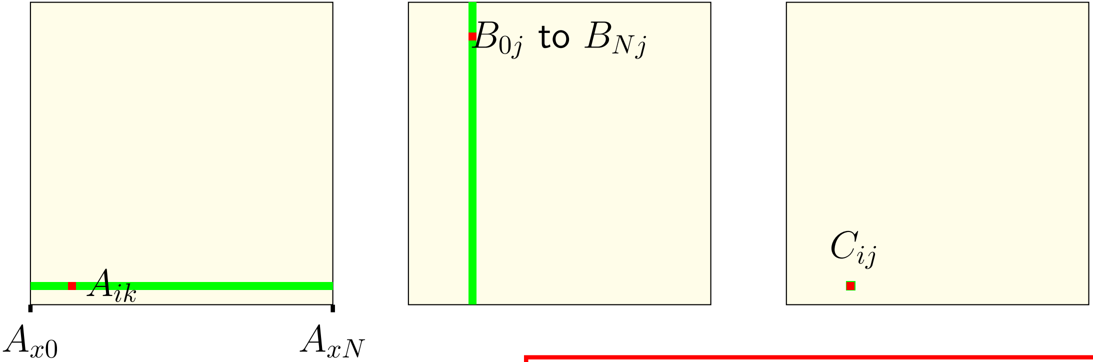
array usage: ijk order



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

looking only at innermost loop:
good spatial locality in A
(rows stored together = reuse cache blocks)
bad spatial locality in B
(use each cache block once)
no useful spatial locality in C

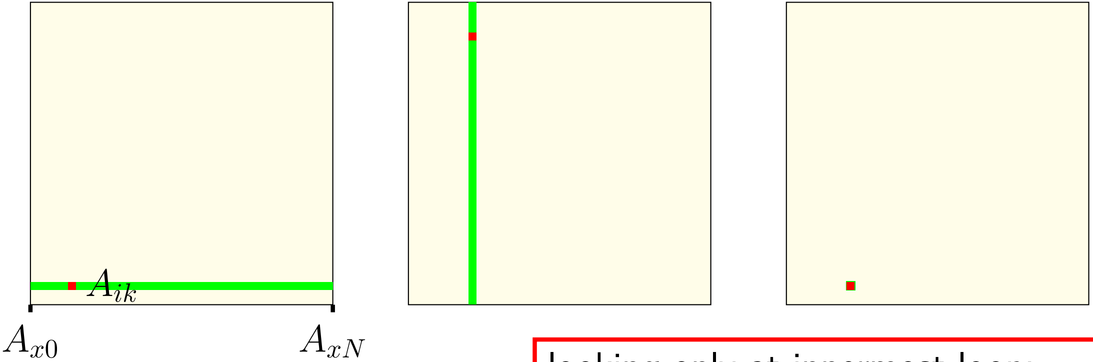
array usage: ijk order



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

looking only at innermost loop:
temporal locality in C
bad temporal locality in everything else
(everything accessed exactly once)

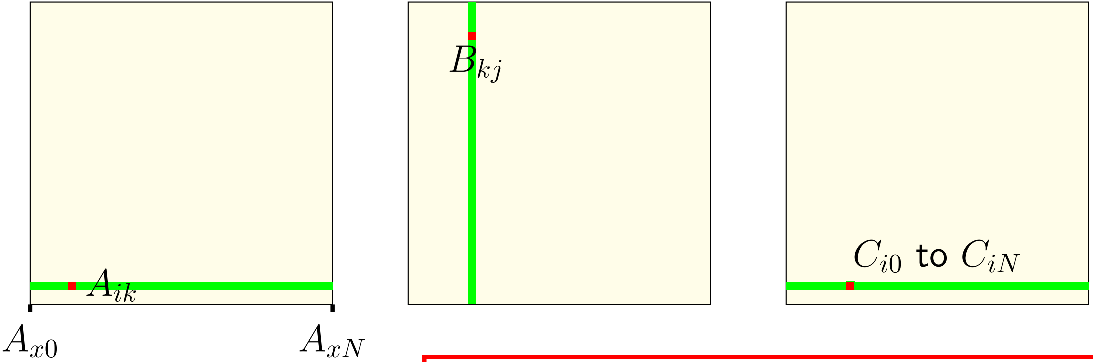
array usage: ijk order



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

looking only at innermost loop:
row of A (elements used once)
column of B (elements used once)
single element of C (used many times)

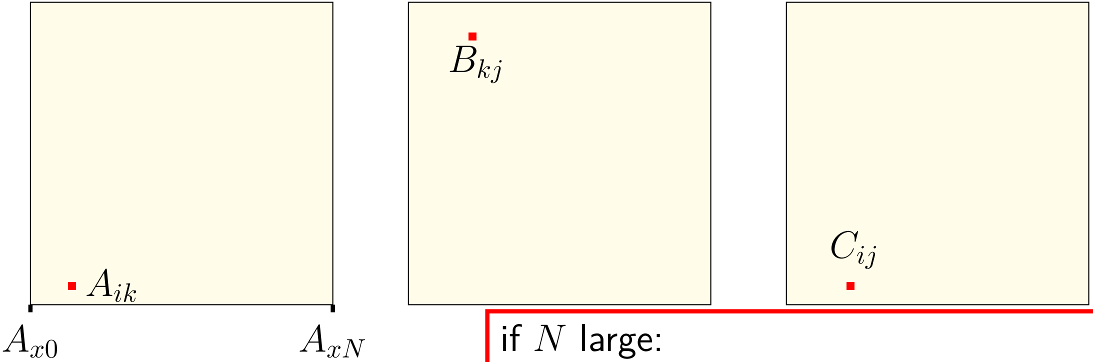
array usage: ijk order



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

looking only at two innermost loops together:
some temporal locality in A (column reused)
some temporal locality in B (row reused)
some temporal locality in C (row reused)

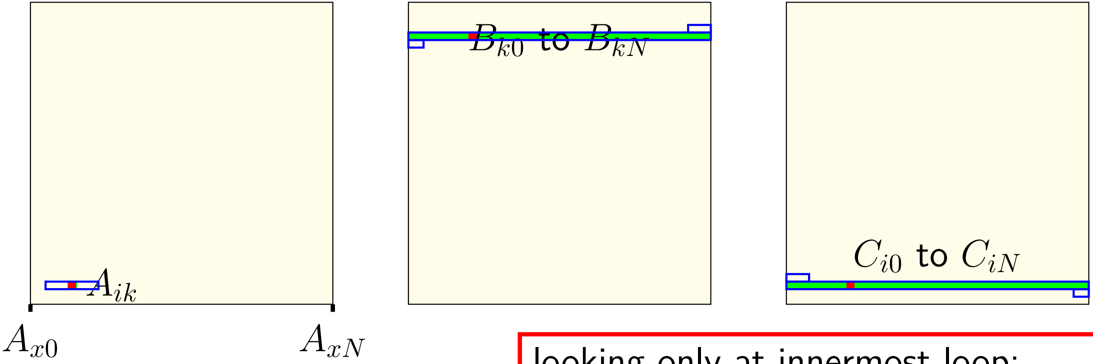
array usage: kij order



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

if N large:
using C_{ij} once per load into cache
(but using $C_{i,j+1}$ right after)
using A_{ik} many times per load-into-cache
using B_{kj} once per load into cache
(but using $B_{k,j+1}$ right after)

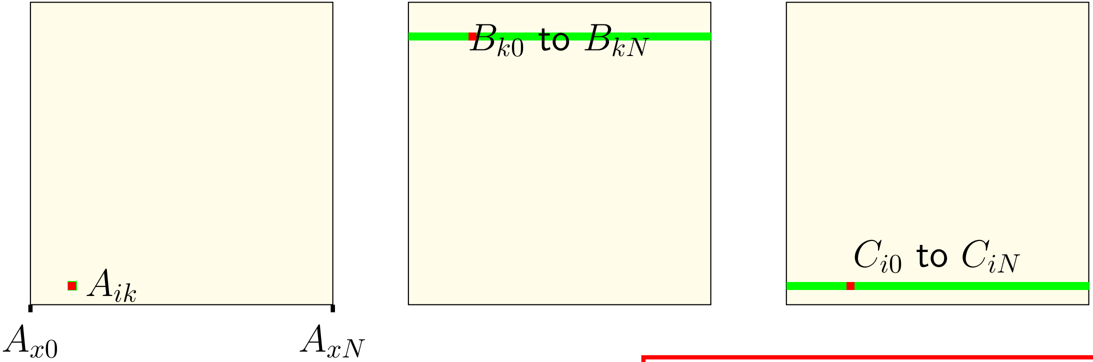
array usage: kij order



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

looking only at innermost loop:
spatial locality in B, C
(use most of loaded B, C cache blocks)
no useful spatial locality in A
(rest of A's cache block wasted)

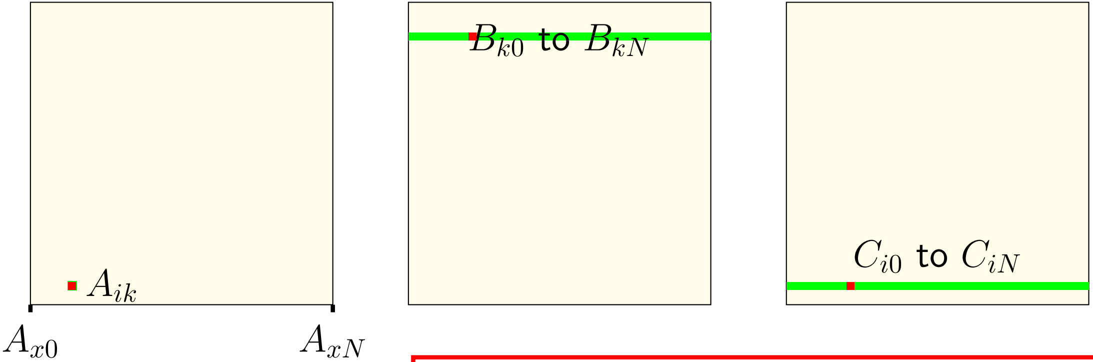
array usage: kij order



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

looking only at innermost loop:
temporal locality in A
no temporal locality in B, C
(B, C values used exactly once)

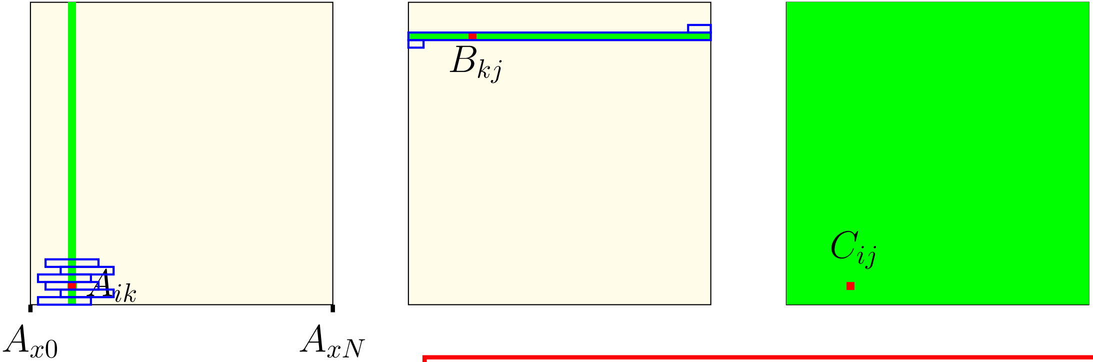
array usage: kij order



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

looking only at innermost loop:
processing one element of A (use many times)
row of B (each element used once)
column of C (each element used once)

array usage: kij order



for all k:

for all i:

for all j:

$C_{ij} += A_{ik} \times B_{kj}$

looking only at two innermost loops together:
good temporal locality in A (column reused)
good temporal locality in B (row reused)
bad temporal locality in C (nothing reused)

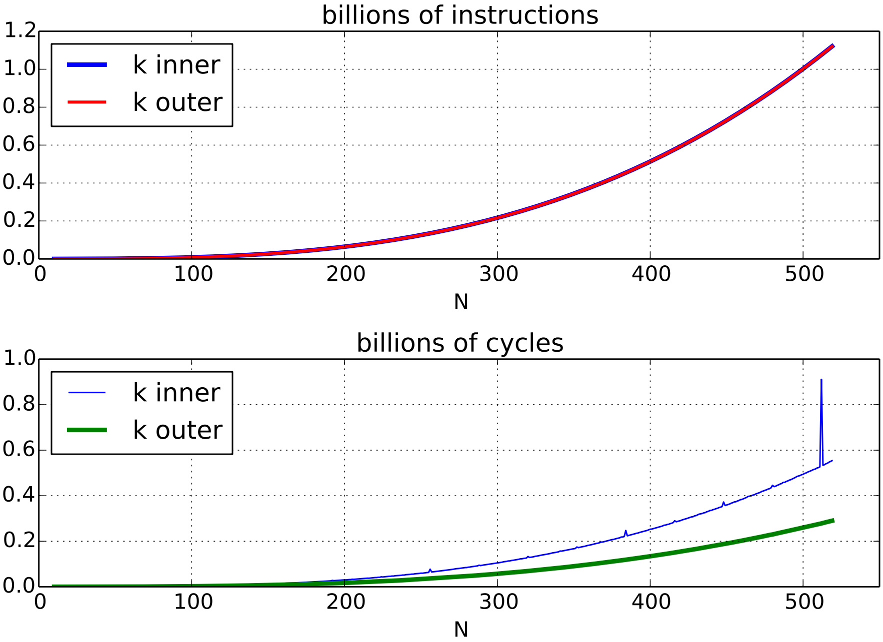
matrix multiply

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

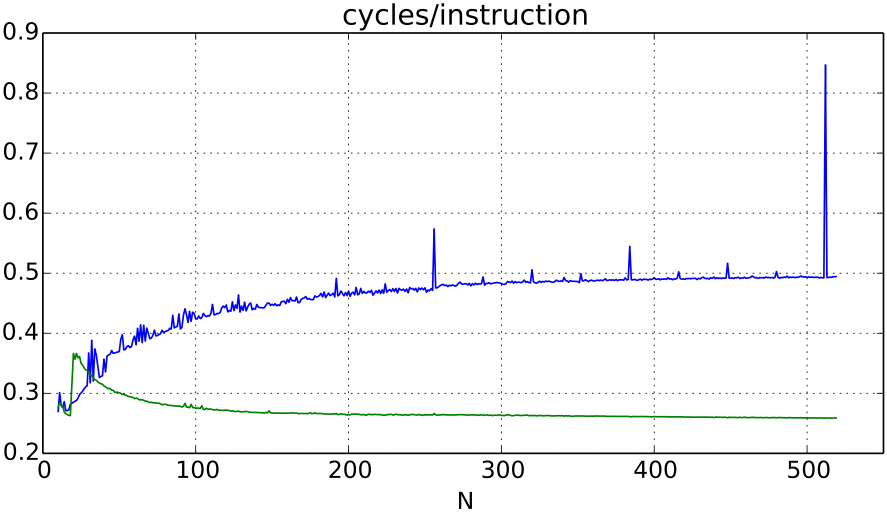
```
/* version 1: inner loop is k, middle is j*/
for (int i = 0; i < N; ++i)
  for (int j = 0; j < N; ++j)
    for (int k = 0; k < N; ++k)
      C[i*N+j] += A[i * N + k] * B[k * N + j];

/* 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)
      C[i*N+j] += A[i * N + k] * B[k * N + j];
```

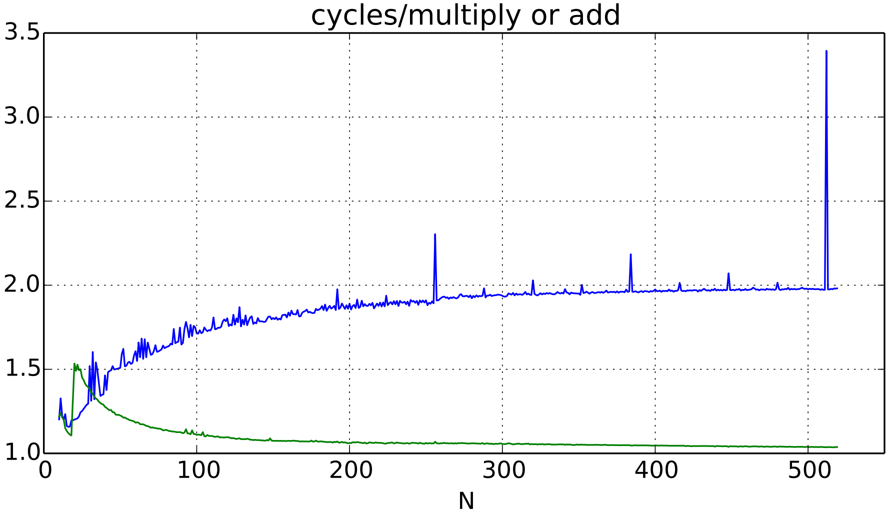
performance (with A=B)



alternate view 1: cycles/instruction



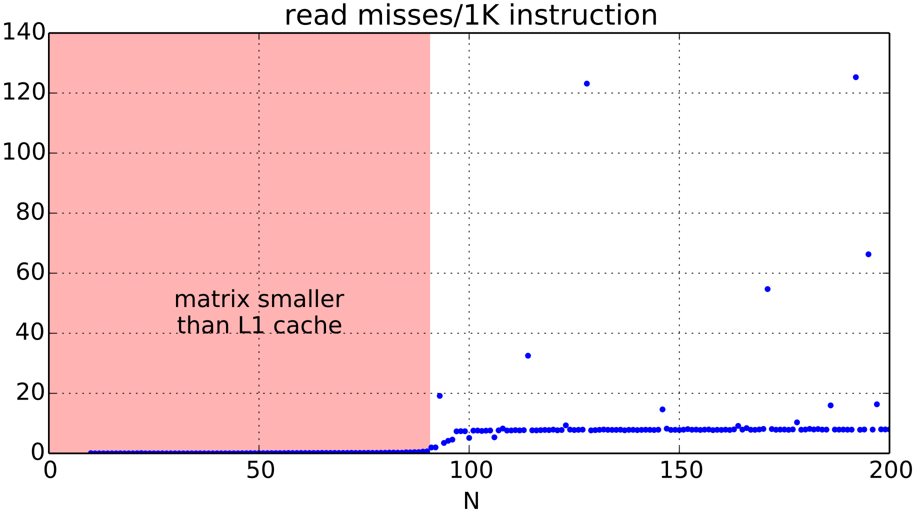
alternate view 2: cycles/operation



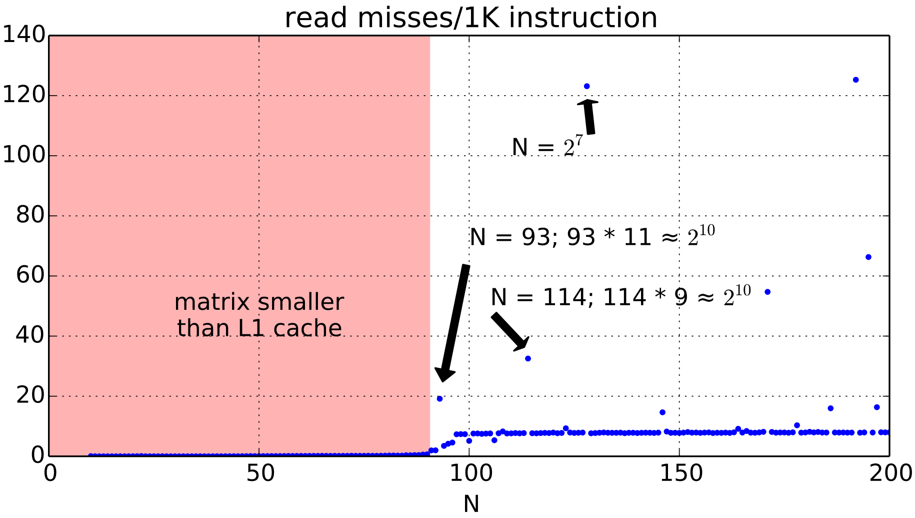
backup slides

L1 misses (with A=B)

L1 miss detail (1)



L1 miss detail (2)



addresses

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

addresses

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

test system L1 cache: 6 index bits, 6 block offset bits

conflict misses

powers of two — lower order bits unchanged

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

 1023 elements apart (4092 bytes; 63.9 cache blocks)

64 sets in L1 cache: usually maps to same set

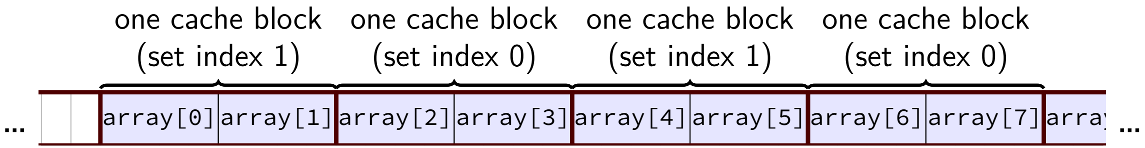
`B[k*93+(j+1)]` will not be cached (next i loop)

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

how to fix? improve spatial locality

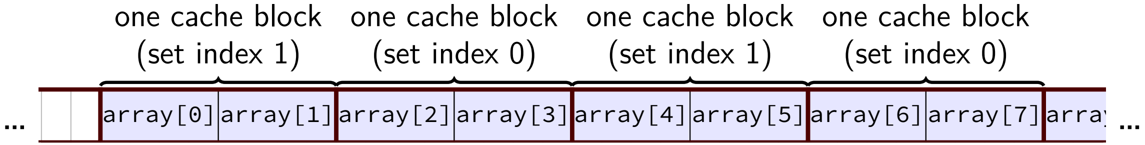
 (maybe even if it requires copying)

quiz exercise solution



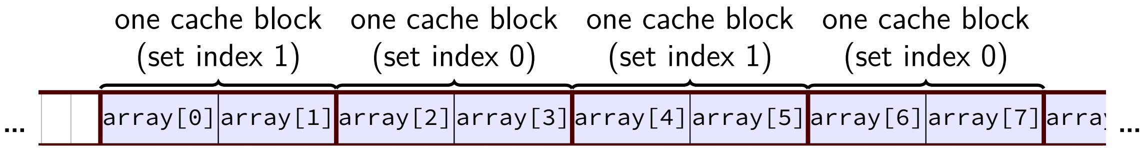
memory access	**set 0 afterwards**	**set 1 afterwards**
—	(empty)	(empty)
read `array[0]` (miss)	{array[0], array[1]}	(empty)
read `array[3]` (miss)	{array[0], array[1]}	{array[2], array[3]}
read `array[6]` (miss)	{array[0], array[1]}	{array[6], array[7]}
read `array[1]` (hit)	{array[0], array[1]}	{array[6], array[7]}
read `array[4]` (miss)	{array[4], array[5]}	{array[6], array[7]}
read `array[7]` (hit)	{array[4], array[5]}	{array[6], array[7]}
read `array[2]` (miss)	{array[4], array[5]}	{array[2], array[3]}
read `array[5]` (hit)	{array[4], array[5]}	{array[6], array[7]}
read `array[8]` (miss)	{array[8], array[9]}	{array[6], array[7]}

quiz exercise solution



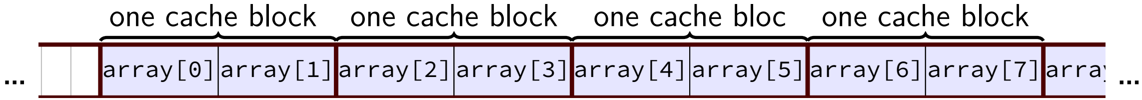
memory access	**set 0 afterwards**	**set 1 afterwards**
—	(empty)	(empty)
read array[0] (miss)	{array[0], array[1]}	(empty)
read array[3] (miss)	{array[0], array[1]}	{array[2], array[3]}
read array[6] (miss)	{array[0], array[1]}	{array[6], array[7]}
read array[1] (hit)	{array[0], array[1]}	{array[6], array[7]}
read array[4] (miss)	{array[4], array[5]}	{array[6], array[7]}
read array[7] (hit)	{array[4], array[5]}	{array[6], array[7]}
read array[2] (miss)	{array[4], array[5]}	{array[2], array[3]}
read array[5] (hit)	{array[4], array[5]}	{array[6], array[7]}
read array[8] (miss)	{array[8], array[9]}	{array[6], array[7]}

quiz exercise solution



memory access	set 0 afterwards	set 1 afterwards
—	(empty)	(empty)
read array[0] (miss)	{array[0], array[1]}	(empty)
read array[3] (miss)	{array[0], array[1]}	{array[2], array[3]}
read array[6] (miss)	{array[0], array[1]}	{array[6], array[7]}
read array[1] (hit)	{array[0], array[1]}	{array[6], array[7]}
read array[4] (miss)	{array[4], array[5]}	{array[6], array[7]}
read array[7] (hit)	{array[4], array[5]}	{array[6], array[7]}
read array[2] (miss)	{array[4], array[5]}	{array[2], array[3]}
read array[5] (hit)	{array[4], array[5]}	{array[6], array[7]}
read array[8] (miss)	{array[8], array[9]}	{array[6], array[7]}

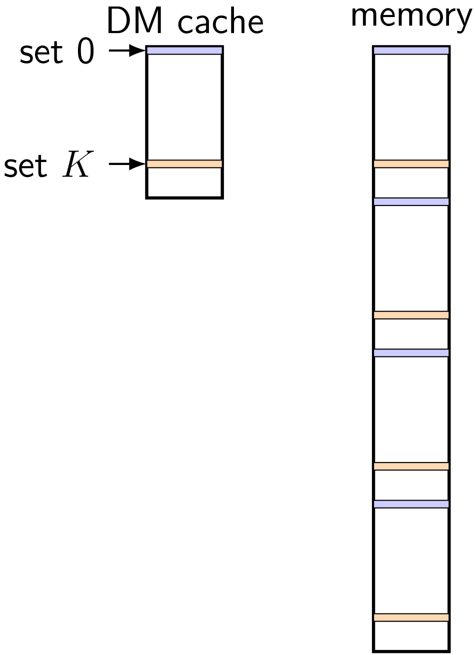
not the quiz problem



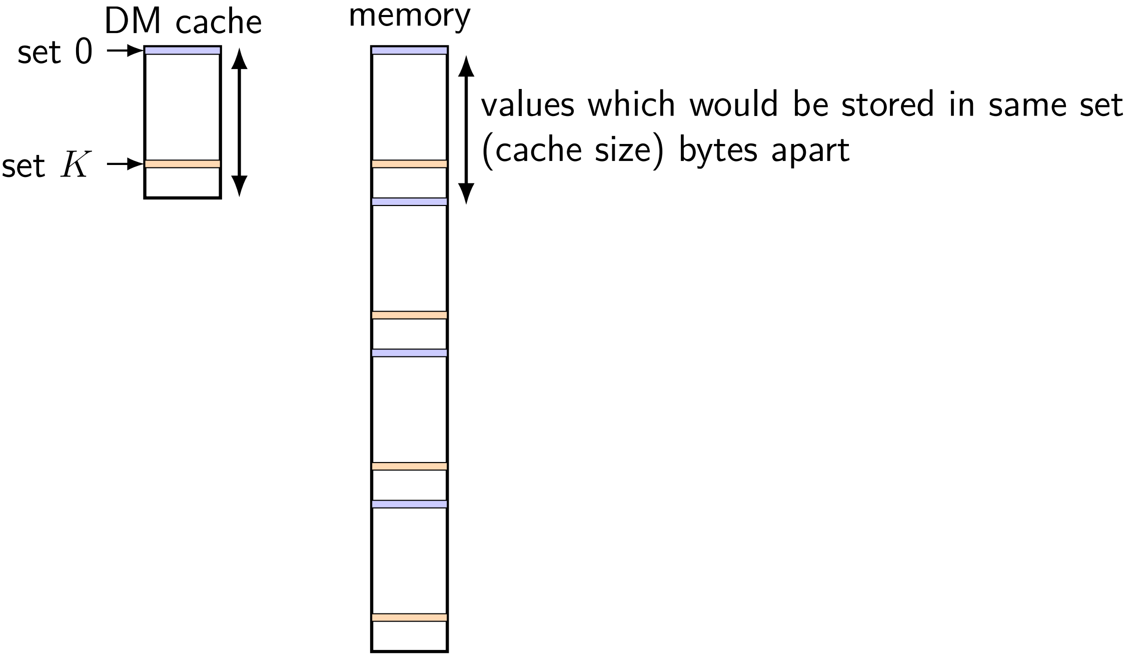
if 1-set 2-way cache instead of 2-set 1-way cache:

memory access	single set with 2-ways, LRU first
—	`---, ---`
read `array[0]` (miss)	`---, {array[0], array[1]}`
read `array[3]` (miss)	`{array[0], array[1]}, {array[2], array[3]}`
read `array[6]` (miss)	`{array[2], array[3]}, {array[6], array[7]}`
read `array[1]` (miss)	`{array[6], array[7]}, {array[0], array[1]}`
read `array[4]` (miss)	`{array[0], array[1]}, {array[3], array[4]}`
read `array[7]` (miss)	`{array[3], array[4]}, {array[6], array[7]}`
read `array[2]` (miss)	`{array[6], array[7]}, {array[2], array[3]}`
read `array[5]` (miss)	`{array[2], array[3]}, {array[5], array[6]}`
read `array[8]` (miss)	`{array[5], array[6]}, {array[8], array[9]}`

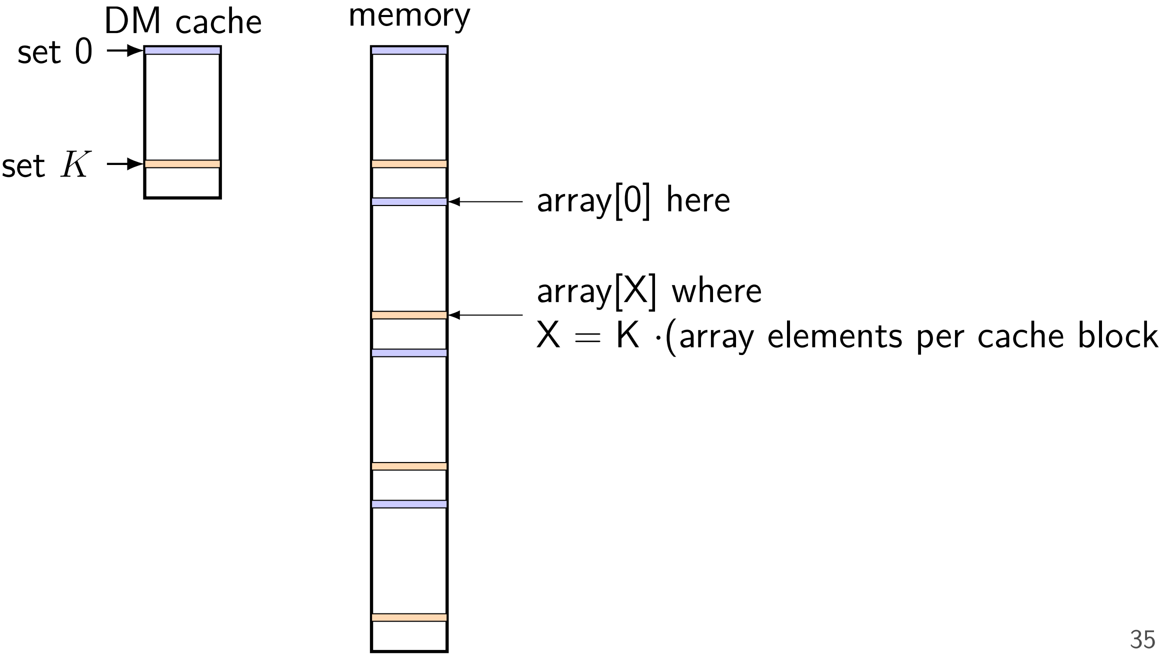
mapping of sets to memory (direct-mapped)



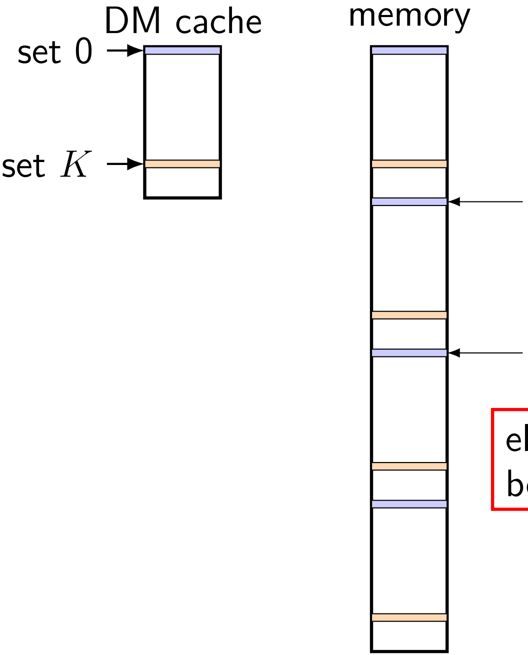
mapping of sets to memory (direct-mapped)



mapping of sets to memory (direct-mapped)



mapping of sets to memory (direct-mapped)



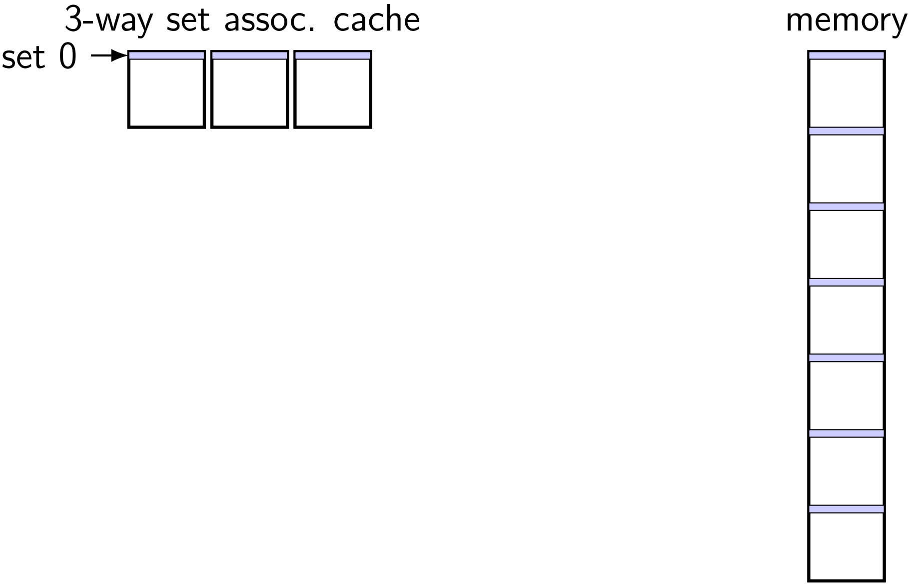
array[0] here

array[X]

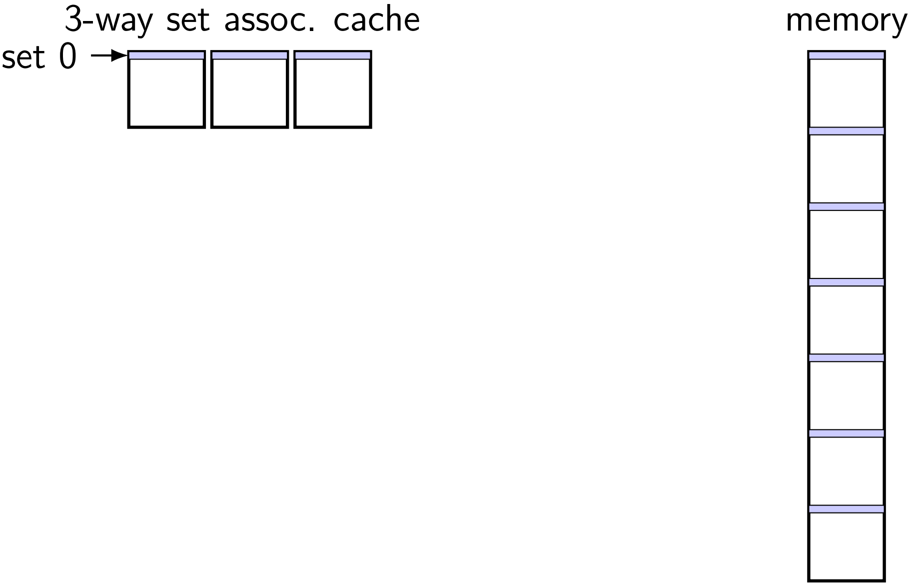
X $=$ (cache size $/$ array element size)

elements (cache size) bytes apart in array
beware conflict misses!

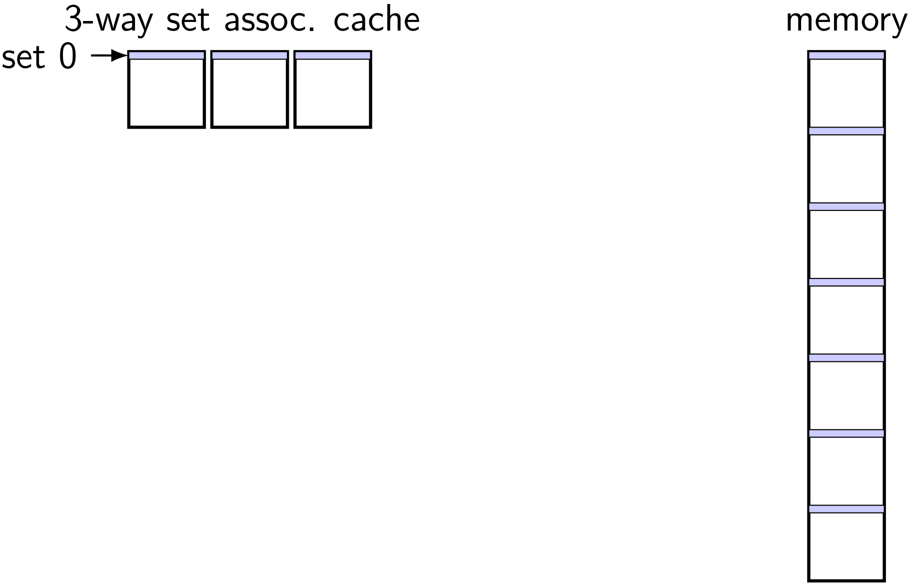
mapping of sets to memory (3-way)



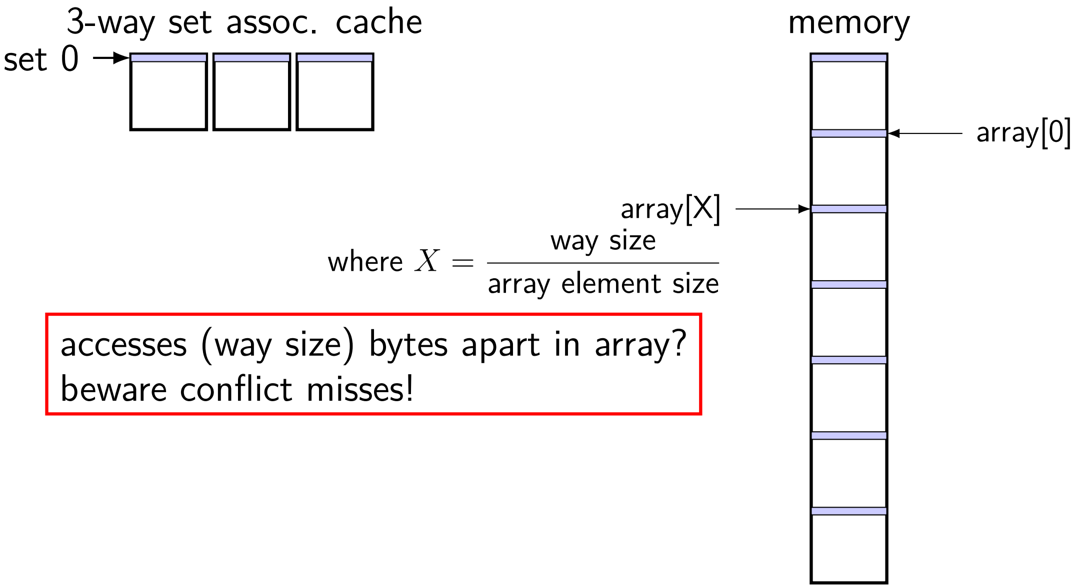
mapping of sets to memory (3-way)



mapping of sets to memory (3-way)



mapping of sets to memory (3-way)



where $X = \frac{\text{way size}}{\text{array element size}}$

accesses (way size) bytes apart in array?
beware conflict misses!

C and cache misses (4)

```
typedef struct {
    int a_value, b_value;
    int other_values[6];
} item;
item items[5];
int a_sum = 0, b_sum = 0;
for (int i = 0; i < 5; ++i)
    a_sum += items[i].a_value;
for (int i = 0; i < 5; ++i)
    b_sum += items[i].b_value;
```

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

C and cache misses (4, rewrite)

```
int array[40]
int a_sum = 0, b_sum = 0;
for (int i = 0; i < 40; i += 8)
    a_sum += array[i];
for (int i = 1; i < 40; i += 8)
    b_sum += array[i];
```

Assume everything but `array` is kept in registers (and the compiler does not do anything funny) and array starts at beginning of cache block.

How many *data cache misses* on a 2-way set associative 128B cache with 16B cache blocks and LRU replacement?

C and cache misses (4, solution pt 1)

ints 4 byte $\rightarrow$ array[0 to 3] and array[16 to 19] in same cache set

- 64B = 16 ints stored per way
- 4 sets total

accessing 0, 8, 16, 24, 32, 1, 9, 17, 25, 33

C and cache misses (4, solution pt 1)

ints 4 byte → array[0 to 3] and array[16 to 19] in same cache set

- 64B = 16 ints stored per way
- 4 sets total

accessing 0, 8, 16, 24, 32, 1, 9, 17, 25, 33

0 (set 0), 8 (set 2), 16 (set 0), 24 (set 2), 32 (set 0)

1 (set 0), 9 (set 2), 17 (set 0), 25 (set 2), 33 (set 0)

C and cache misses (4, solution pt 2)

access	set 0 after (LRU first)	result
—	—, —	
array[0]	—, array[0 to 3]	miss
array[16]	array[0 to 3], array[16 to 19]	miss
array[32]	array[16 to 19], array[32 to 35]	miss
array[1]	array[32 to 35], array[0 to 3]	miss
array[17]	array[0 to 3], array[16 to 19]	miss
array[32]	array[16 to 19], array[32 to 35]	miss

6 misses for set 0

C and cache misses (4, solution pt 3)

access	set 2 after (LRU first)	result	
—	—, —		
array[8]	—, array[8 to 11]	miss	2 misses for set 1
array[24]	array[8 to 11], array[24 to 27]	miss	
array[9]	array[8 to 11], array[24 to 27]	hit	
array[25]	array[16 to 19], array[32 to 35]	hit	

C and cache misses (3)

```
typedef struct {
    int a_value, b_value;
    int other_values[10];
} item;
item items[5];
int a_sum = 0, b_sum = 0;
for (int i = 0; i < 5; ++i)
    a_sum += items[i].a_value;
for (int i = 0; i < 5; ++i)
    b_sum += items[i].b_value;
```

observation: 12 ints in struct: only first two used

equivalent to accessing array[0], array[12], array[24], etc.

...then accessing array[1], array[13], array[25], etc.

C and cache misses (3, rewritten?)

```
int array[60];
int a_sum = 0, b_sum = 0;
for (int i = 0; i < 60; i += 12)
    a_sum += array[i];
for (int i = 1; i < 60; i += 12)
    b_sum += array[i];
```

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

How many *data cache misses* on a 128B two-way set associative cache with 16B cache blocks and LRU replacement?

observation 1: first loop has 5 misses — first accesses to blocks

observation 2: array[0] and array[1], array[12] and array[13], etc. in same cache block

C and cache misses (3, solution)

ints 4 byte → array[0 to 3] and array[16 to 19] in same cache set

- 64B = 16 ints stored per way
- 4 sets total

accessing array indices 0, 12, 24, 36, 48, 1, 13, 25, 37, 49

so access to 1, 21, 41, 61, 81 all hits:

- set 0 contains block with array[0 to 3]
- set 5 contains block with array[20 to 23]
- etc.

C and cache misses (3, solution)

ints 4 byte → array[0 to 3] and array[16 to 19] in same cache set
- 64B = 16 ints stored per way
- 4 sets total

accessing array indices 0, 12, 24, 36, 48, 1, 13, 25, 37, 49

so access to 1, 21, 41, 61, 81 all hits:
- set 0 contains block with array[0 to 3]
- set 5 contains block with array[20 to 23]
- etc.

C and cache misses (3, solution)

ints 4 byte → array[0 to 3] and array[16 to 19] in same cache set

- 64B = 16 ints stored per way
- 4 sets total

accessing array indices 0, 12, 24, 36, 48, 1, 13, 25, 37, 49

0 (set 0, array[0 to 3]), 12 (set 3), 24 (set 2), 36 (set 1), 48 (set 0)

- each set used at most twice
- no replacement needed

so access to 1, 21, 41, 61, 81 all hits:

- set 0 contains block with array[0 to 3]
- set 5 contains block with array[20 to 23]
- etc.

C and cache misses (3)

```
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)

```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 4-way set associative 2KB direct-mapped cache with 16B cache blocks?

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, …

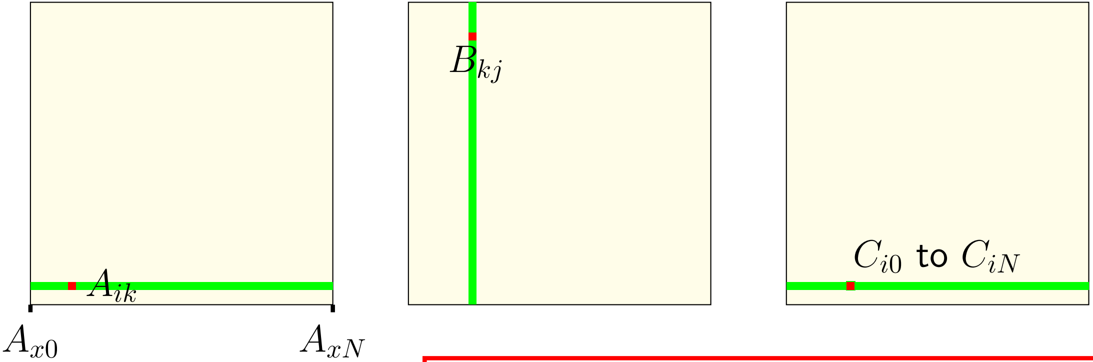
- address 16: array[4] through array[7]

…

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

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

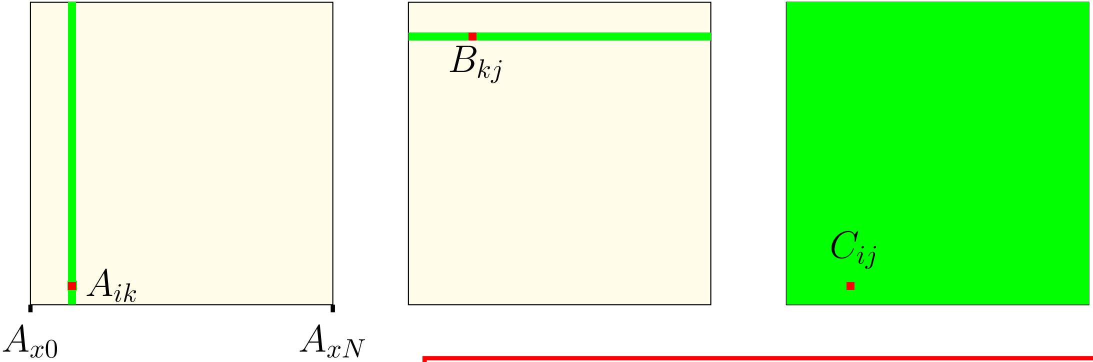
array usage: ijk order



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

looking only at two innermost loops together:
good spatial locality in A
poor spatial locality in B
good spatial locality in C

array usage: kij order



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

looking only at two innermost loops together:
poor spatial locality in A
good spatial locality in B
good spatial locality in C

simple blocking – with 3?

```
for (int kk = 0; kk < N; kk += 3)
  for (int i = 0; i < N; i += 1)
    for (int j = 0; j < N; ++j) {
      C[i*N+j] += A[i*N+kk+0] * B[(kk+0)*N+j];
      C[i*N+j] += A[i*N+kk+1] * B[(kk+1)*N+j];
      C[i*N+j] += A[i*N+kk+2] * B[(kk+2)*N+j];
    }
```

$\frac{N}{3} \cdot N$ j-loop iterations, and (assuming N large):

about 1 misses from A per j-loop iteration
  $N^2/3$ total misses (before blocking: N^2)

about $3N \div$ block size misses from B per j-loop iteration
  $N^3 \div$ block size total misses (same as before)

about $3N \div$ block size misses from C per j-loop iteration
  $N^3 \div$ block size total misses (same as before)

simple blocking – with 3?

```
for (int kk = 0; kk < N; kk += 3)
  for (int i = 0; i < N; i += 1)
    for (int j = 0; j < N; ++j) {
      C[i*N+j] += A[i*N+kk+0] * B[(kk+0)*N+j];
      C[i*N+j] += A[i*N+kk+1] * B[(kk+1)*N+j];
      C[i*N+j] += A[i*N+kk+2] * B[(kk+2)*N+j];
    }
```

$\frac{N}{3} \cdot N$ j-loop iterations, and (assuming N large):

about 1 misses from A per j-loop iteration

$N^2/3$ total misses (before blocking: N^2)

about $3N \div$ block size misses from B per j-loop iteration

$N^3 \div$ block size total misses (same as before)

about $3N \div$ block size misses from C per j-loop iteration

$N^3 \div$ block size total misses (same as before)

more than 3?

can we just keep doing this increase from 3 to some large X? ...

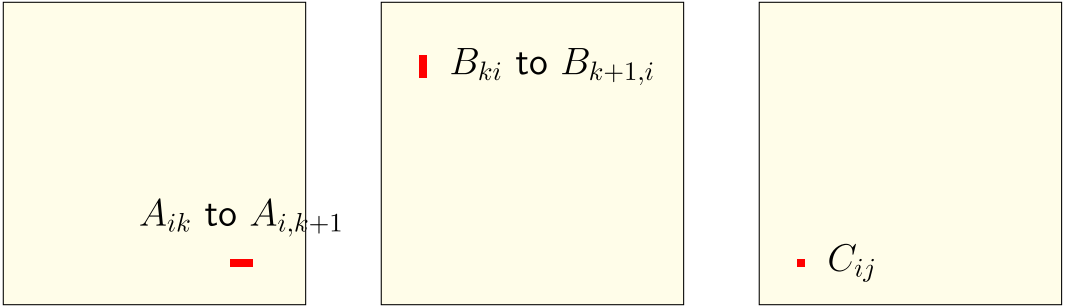
assumption: X values from A would stay in cache

- X too large — cache not big enough

assumption: X blocks from B would help with spatial locality

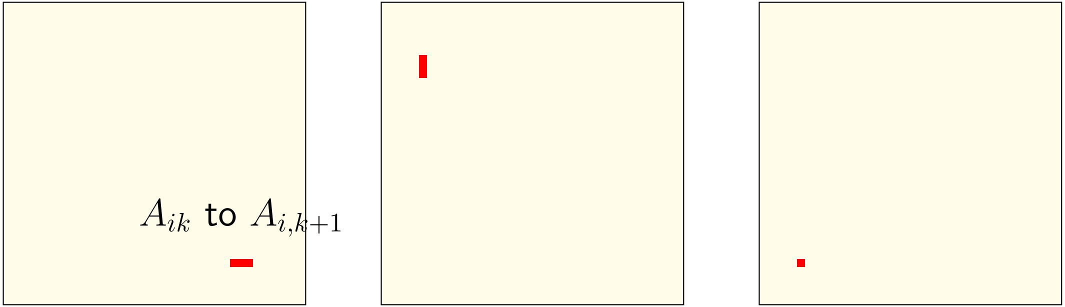
- X too large — evicted from cache before next iteration

array usage (2 k at a time)



for each kk:
    for each i:
        for each j:
            for k=kk,kk+1:
                $C_{ij} += A_{ik} \cdot B_{kj}$

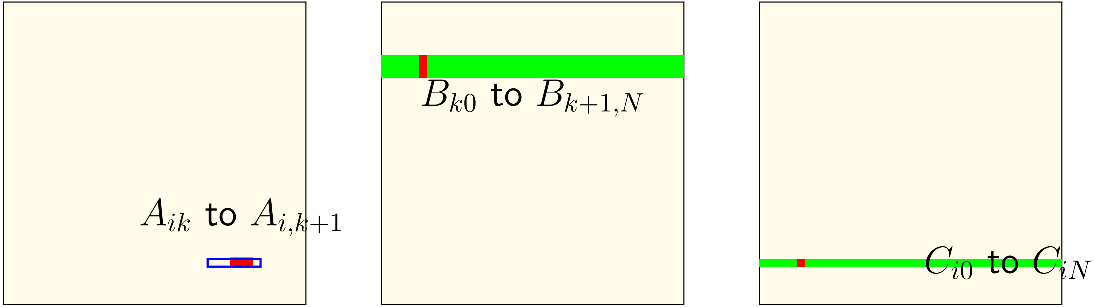
array usage (2 k at a time)



for each kk:
    for each i:
        for each j:
            for k=kk,kk+1:
                $C_{ij} {+}{=} A_{ik} \cdot B_{kj}$

within innermost loop
good spatial locality in A
bad locality in B
good temporal locality in C

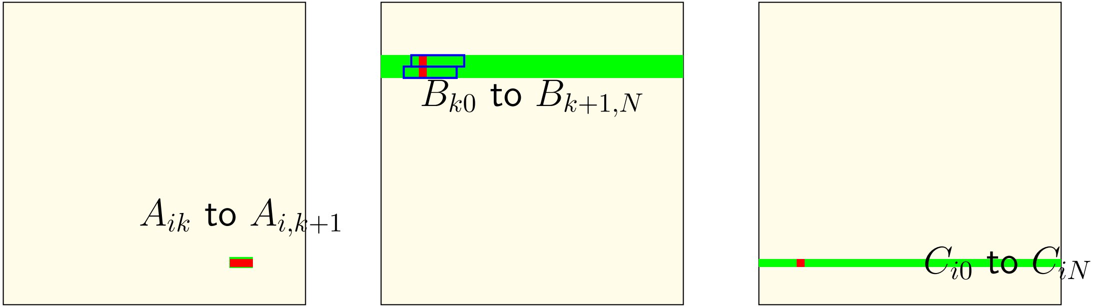
array usage (2 k at a time)



for each kk:
    for each i:
        for each j:
            for k=kk,kk+1:
                $C_{ij} += A_{ik} \cdot B_{kj}$

loop over j: better spatial locality over A than before; still good temporal locality for A

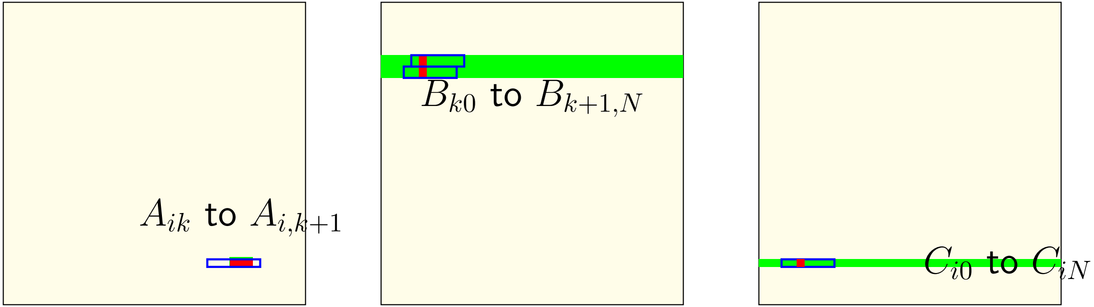
array usage (2 k at a time)



for each kk:
for each i:
for each j:
for k=kk,kk+1:
$C_{ij} += A_{ik} \cdot B_{kj}$

loop over j: spatial locality over B is worse
but probably not more misses
cache needs to keep two cache blocks
for next iter instead of one
(probably has the space left over!)

array usage (2 k at a time)



for each kk:
    for each i:
        for each j:
            for k=kk,kk+1:
                $C_{ij} += A_{ik} \cdot$

right now: only really care about
keeping 4 cache blocks in j loop

have more than 4 cache blocks?
increasing kk increment would use more of them

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