cache performance 2

last time

write buffers

- consider write complete when placed in buffer
- perform write in background (as long as buffer not filled)

cache hierarchies

average memory access time

which helps/hurts hit time / miss penalty / miss rate

mapping C code to cache accesses

- tracking contents of cache
- importance of alignment
- checking each set individually

logistics next week

I will be out of town M-Th

my office hours will not be held

Professor Tychonievich will cover lectures

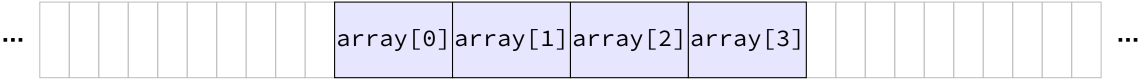
C and cache misses (warmup 1)

```
int array[4];
...
int even_sum = 0, odd_sum = 0;
even_sum += array[0];
odd_sum += array[1];
even_sum += array[2];
odd_sum += array[3];
```

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

How many data cache misses on a 1-set direct-mapped cache with 8B blocks?

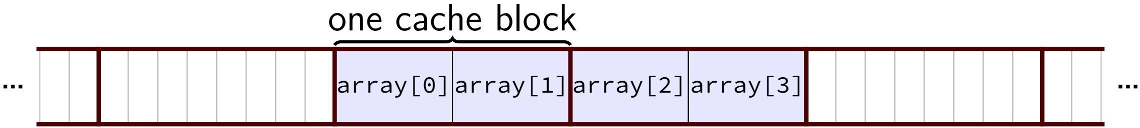
some possiblities



Q1: how do cache blocks correspond to array elements?

not enough information provided!

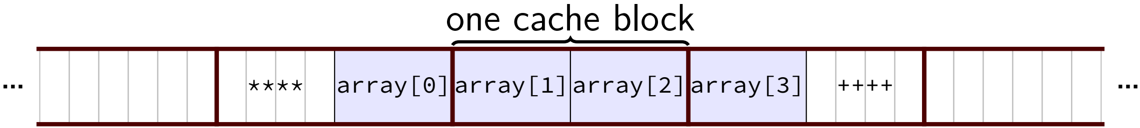
some possiblities



if array[0] starts at beginning of a cache block...
array split across two cache blocks

memory access	**cache contents afterwards**
—	`(empty)`
read `array[0]` (miss)	`{array[0], array[1]}`
read `array[1]` (hit)	`{array[0], array[1]}`
read `array[2]` (miss)	`{array[2], array[3]}`
read `array[3]` (hit)	`{array[2], array[3]}`

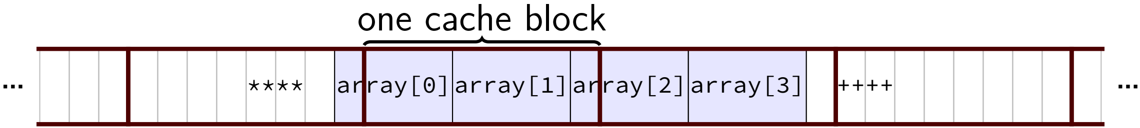
some possiblities



if array[0] starts right in the middle of a cache block
array split across three cache blocks

memory access	**cache contents afterwards**
—	(empty)
read `array[0]` (miss)	`{****, array[0]}`
read `array[1]` (miss)	`{array[1], array[2]}`
read `array[2]` (hit)	`{array[1], array[2]}`
read `array[3]` (miss)	`{array[3], ++++}`

some possiblities



if array[0] starts at an odd place in a cache block,
need to read two cache blocks to get most array elements

memory access	**cache contents afterwards**
—	(empty)
read `array[0]` byte 0 (miss)	{ ****, array[0] byte 0 }
read `array[0]` byte 1-3 (miss)	{ array[0] byte 1-3, array[2], array[3] byte 0 }
read `array[1]` (hit)	{ array[0] byte 1-3, array[2], array[3] byte 0 }
read `array[2]` byte 0 (hit)	{ array[0] byte 1-3, array[2], array[3] byte 0 }
read `array[2]` byte 1-3 (miss)	{part of array[2], array[3], ++++}
read `array[3]` (hit)	{part of array[2], array[3], ++++}

aside: alignment

compilers and malloc/new implementations usually try align values

align = make address be multiple of something

most important reason: don't cross cache block boundaries

C and cache misses (warmup 2)

```
int array[4];
int even_sum = 0, odd_sum = 0;
even_sum += array[0];
even_sum += array[2];
odd_sum += array[1];
odd_sum += array[3];
```

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

Assume array[0] at beginning of cache block.

How many data cache misses on a 1-set direct-mapped cache with 8B blocks?

C and cache misses (warmup 3)

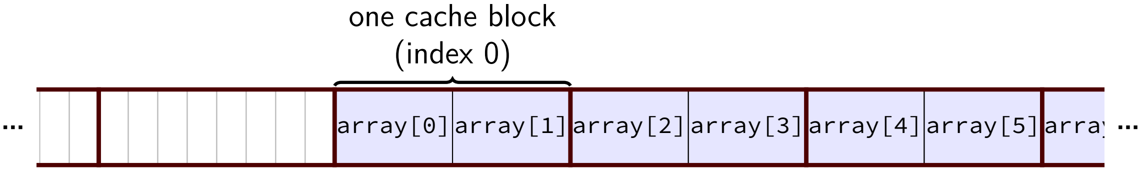
```
int array[8];
...
int even_sum = 0, odd_sum = 0;
even_sum += array[0];
odd_sum += array[1];
even_sum += array[2];
odd_sum += array[3];
even_sum += array[4];
odd_sum += array[5];
even_sum += array[6];
odd_sum += array[7];
```

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

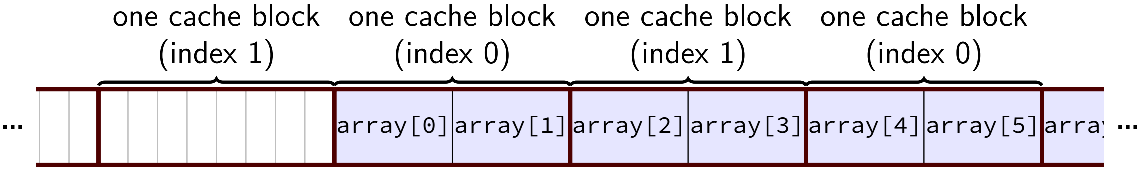
Assume array[0] at beginning of cache block.

How many data cache misses on a **2**-set direct-mapped cache with 8B blocks?

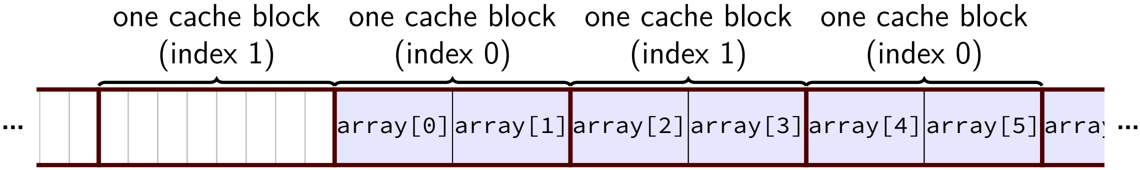
exercise solution



exercise solution

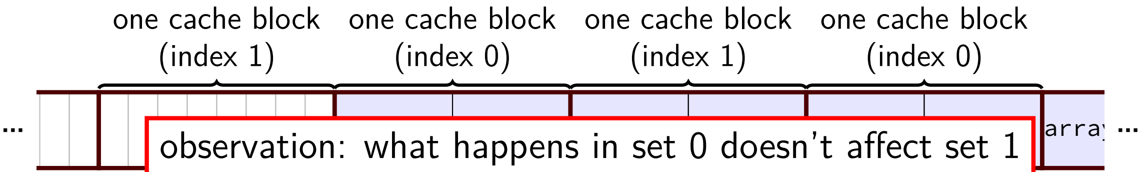


exercise solution



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

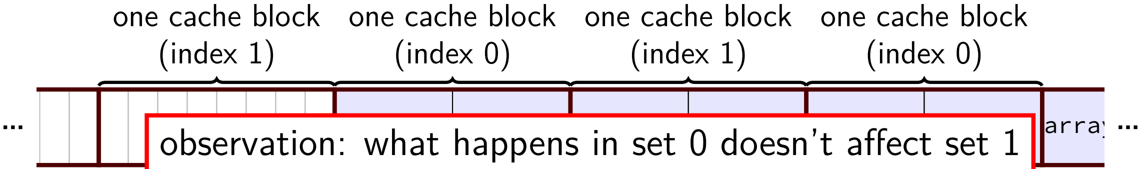
exercise solution



observation: what happens in set 0 doesn't affect set 1
when evaluating set 0 accesses,
can ignore non-set 0 accesses/content

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

exercise solution

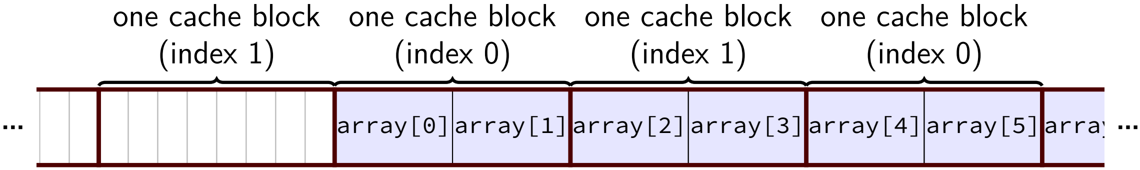


observation: what happens in set 0 doesn't affect set 1

when evaluating set 0 accesses,
can ignore non-set 0 accesses/content

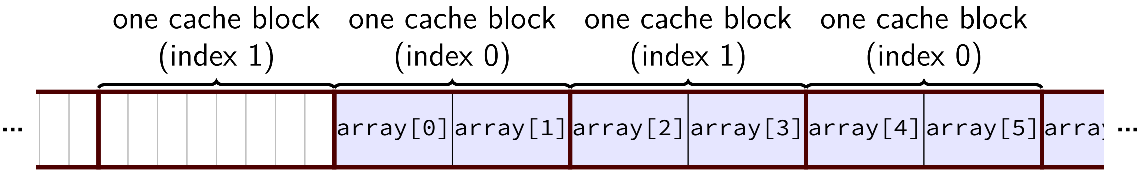
memory access	set 0 afterwards	set 1 afterwards
—	(empty)	(empty)
read `array[0]` (miss)	{array[0], array[1]}	(empty)
read `array[1]` (hit)	{array[0], array[1]}	(empty)
read `array[2]` (miss)	{array[0], array[1]}	{array[2], array[3]}
read `array[3]` (hit)	{array[0], array[1]}	{array[2], array[3]}
read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

exercise solution



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

exercise solution



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

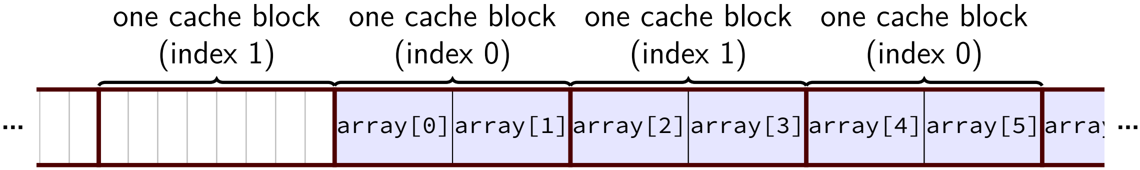
C and cache misses (warmup 4)

```
int array[8];
...
int even_sum = 0, odd_sum = 0;
even_sum += array[0];
even_sum += array[2];
even_sum += array[4];
even_sum += array[6];
odd_sum += array[1];
odd_sum += array[3];
odd_sum += array[5];
odd_sum += array[7];
```

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

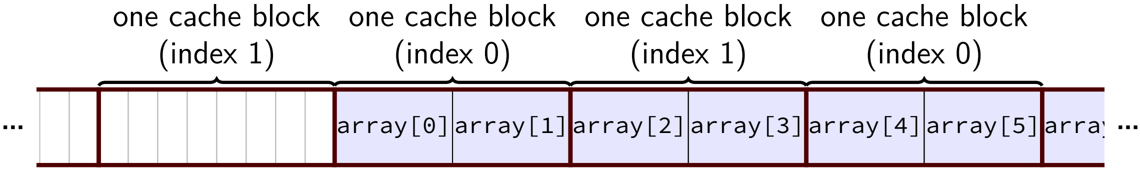
How many data cache misses on a **2**-set direct-mapped cache with 8B blocks?

exercise solution



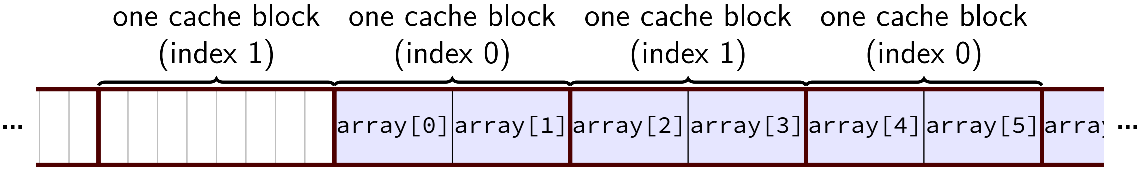
memory access	**set 0 afterwards**	**set 1 afterwards**
—	(empty)	(empty)
read `array[0]` (miss)	{array[0], array[1]}	(empty)
read `array[2]` (miss)	{array[0], array[1]}	{array[2], array[3]}
read `array[4]` (miss)	{array[4], array[5]}	{array[2], array[3]}
read `array[6]` (miss)	{array[4], array[5]}	{array[6],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

exercise solution



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

exercise solution



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

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?

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 3/4s 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];
```

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

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

kij order: A_{ik} has temporal locality

... better than ...

$ij\color{red}{k}$ order: $A_{i\color{red}{k}}$ 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];
```

which is better?

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

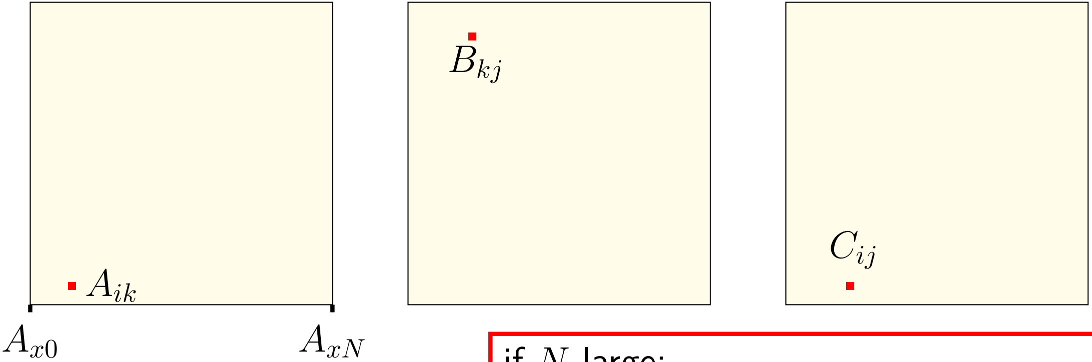
```
/* version 1: inner loop is k, middle is j*/
for (int i = 0; i < N; ++i)
  for (int j = 0; j < N; ++j)
    for (int k = 0; k < N; ++k)
      C[i*N+j] += A[i * N + k] * B[k * N + j];

/* 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?

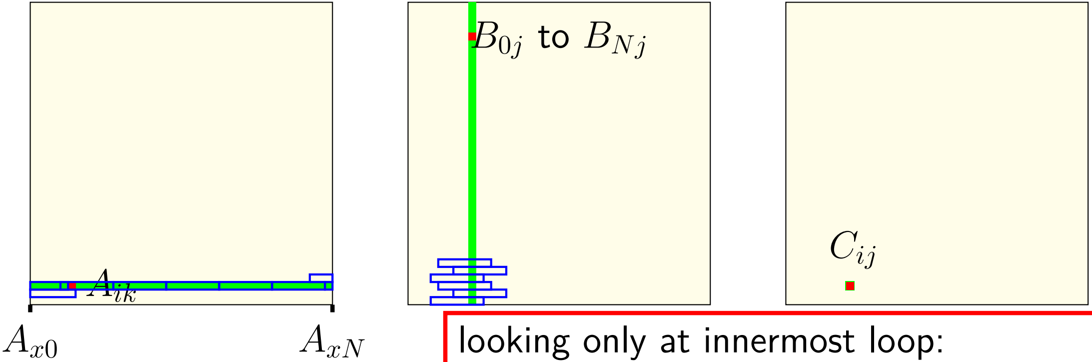
array usage: ijk order



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

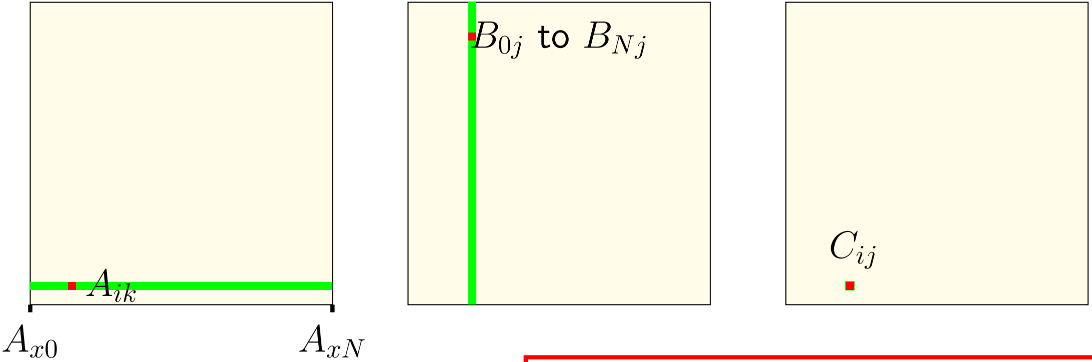
array usage: ijk order



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

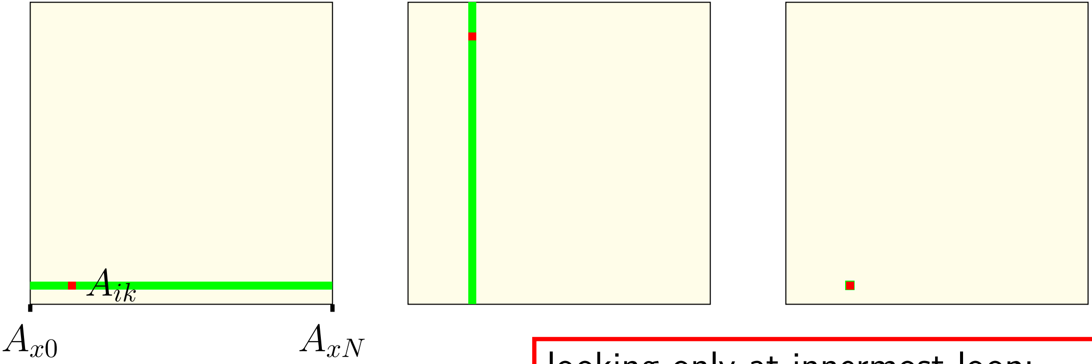
array usage: ijk order



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)

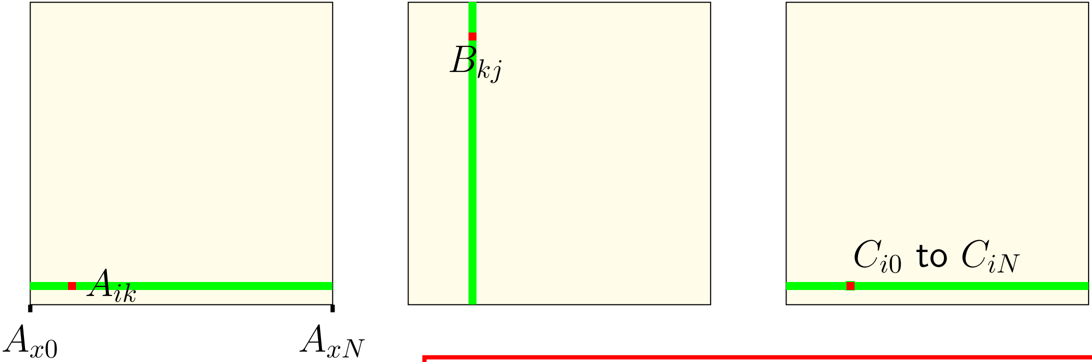
array usage: ijk order



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)

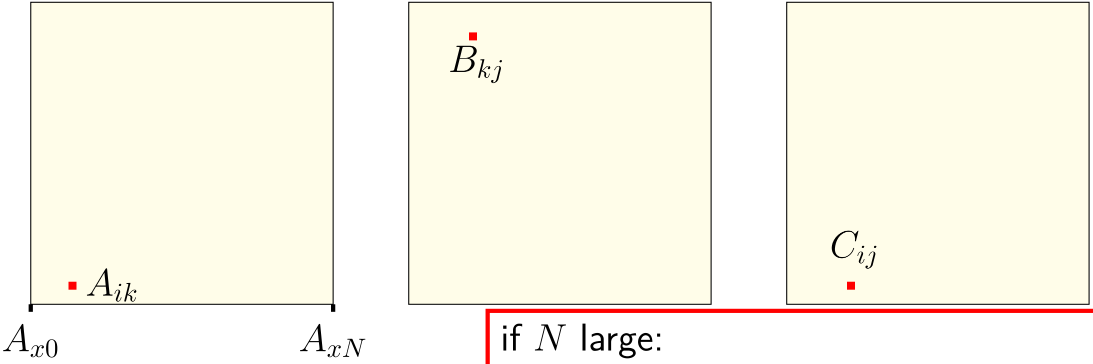
array usage: ijk order



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)

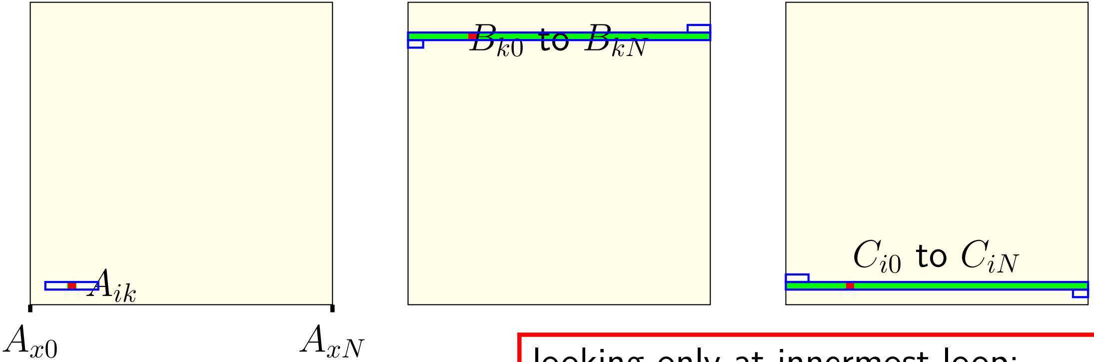
array usage: kij order



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)

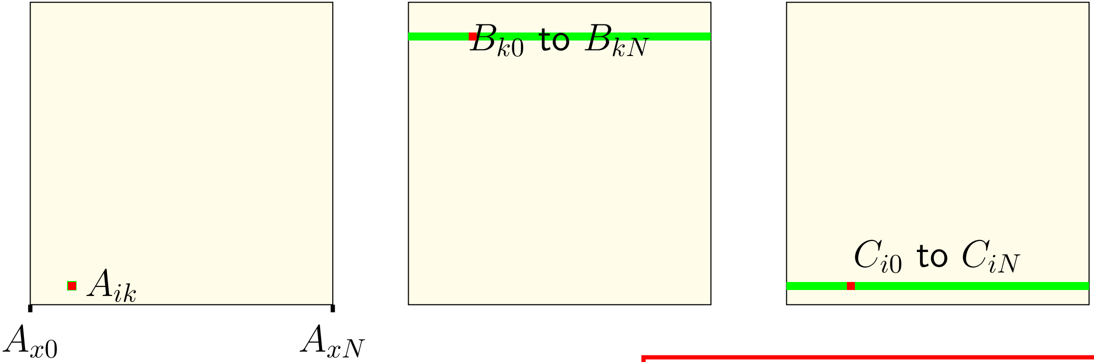
array usage: kij order



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)

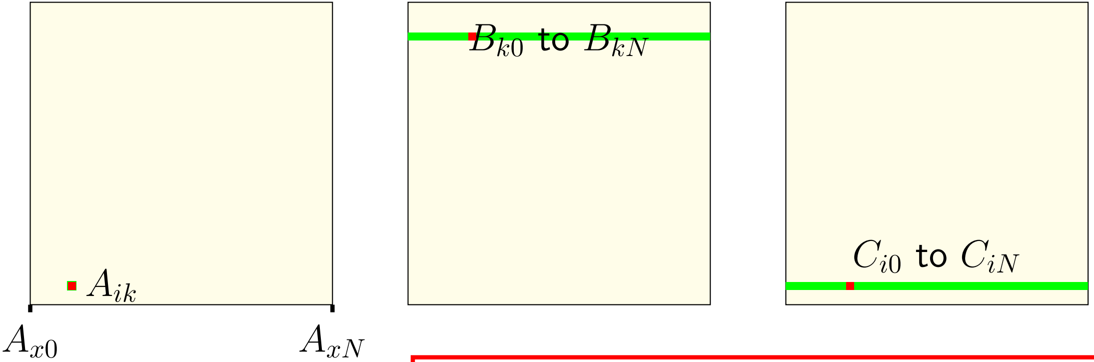
array usage: kij order



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)

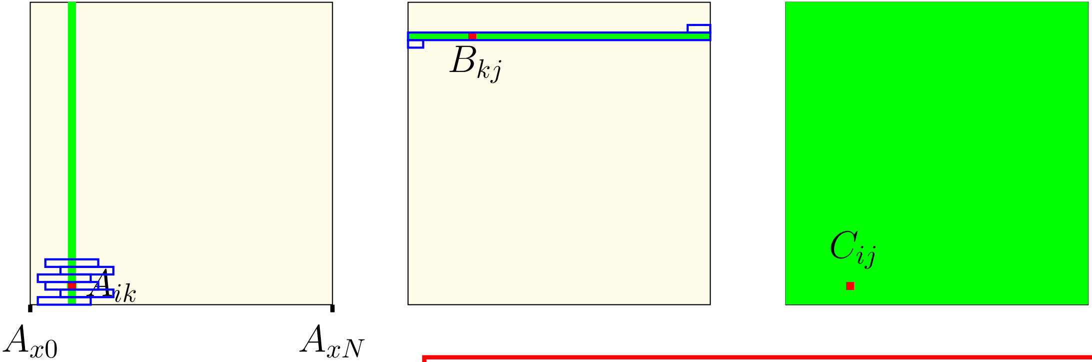
array usage: kij order



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)

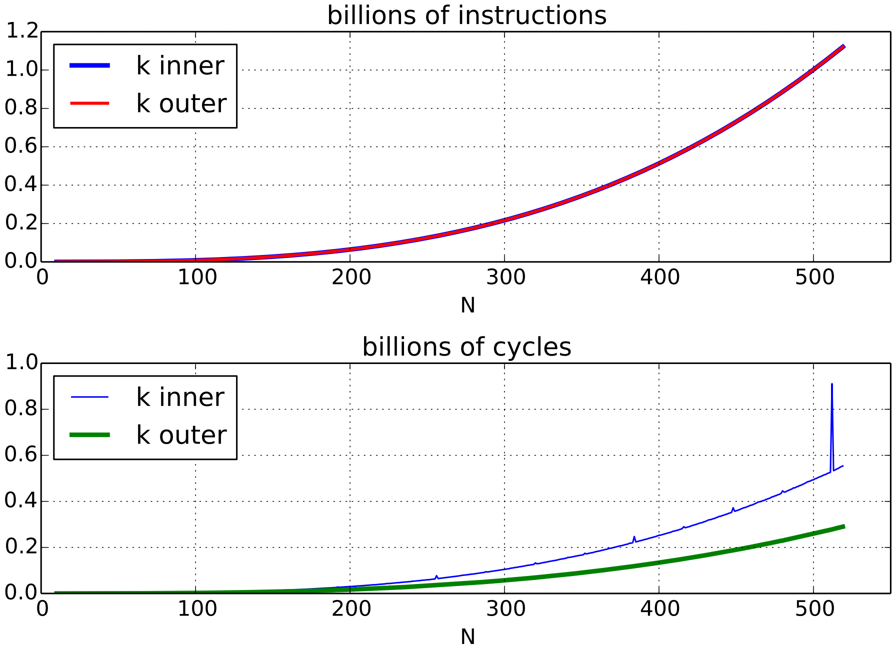
matrix multiply

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

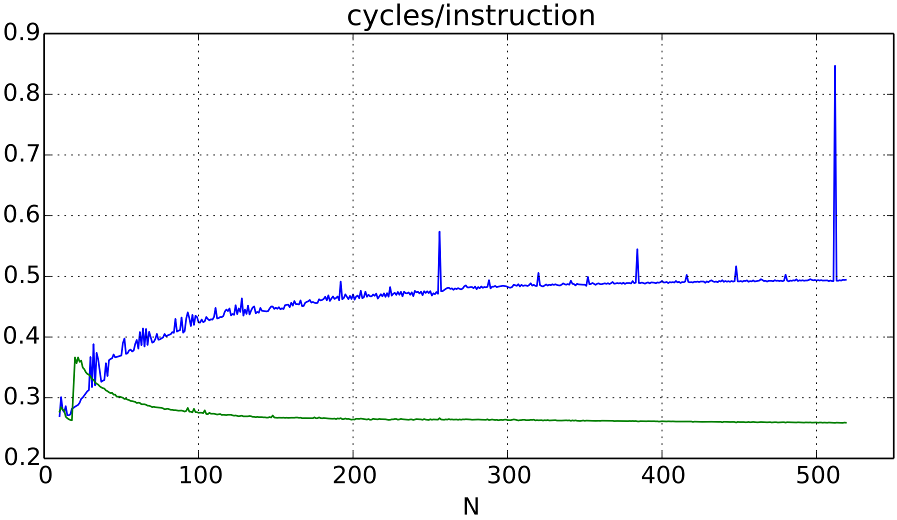
```
/* version 1: inner loop is k, middle is j*/
for (int i = 0; i < N; ++i)
  for (int j = 0; j < N; ++j)
    for (int k = 0; k < N; ++k)
      C[i*N+j] += A[i * N + k] * B[k * N + j];

/* 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];
```

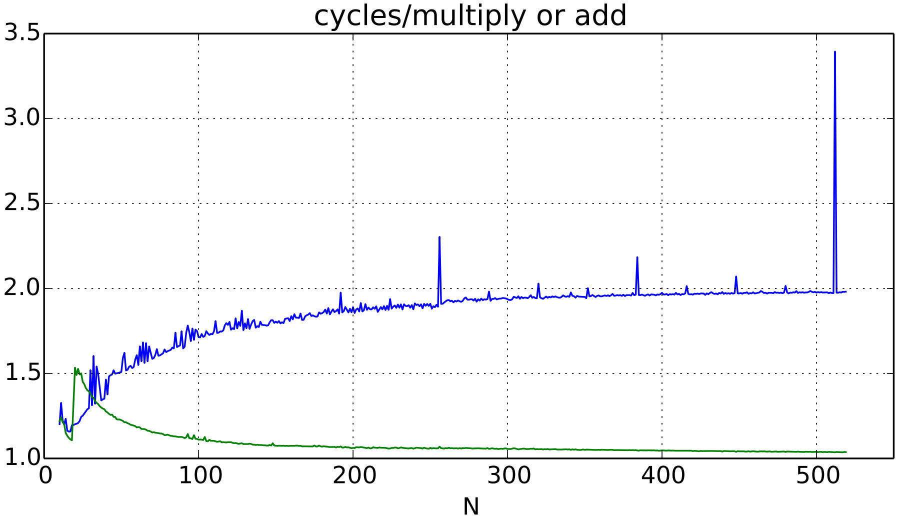
performance (with A=B)



alternate view 1: cycles/instruction



alternate view 2: cycles/operation



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

exercise: miss estimating (2)

```
for (int k = 0; k < 1000; k += 1)
    for (int i = 0; i < 1000; i += 1)
        for (int j = 0; j < 1000; j += 1)
            A[k*N+j] += B[i*N+j];
```

assuming: 4 elements per block

assuming: cache not close to big enough to hold 1K elements

estimate: *approximately* how many misses for A, B?

locality exercise (2)

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

/* version 3 */
for (int ii = 0; ii < N; ii += 32)
    for (int jj = 0; jj < N; jj += 32)
        for (int i = ii; i < ii + 32; ++i)
            for (int j = jj; j < jj + 32; ++j)
                A[i] += B[j] * C[i * N + j];
```

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

how about spatial locality?

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

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: */
      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];
    }
  }
}
```

Temporal locality in C_{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: */
      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];
    }
  }
}
```

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: */
      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];
    }
  }
}
```

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

counting misses for A (1)

```
for (int kk = 0; kk < N; kk += 2)
  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];
    }
```

access pattern for A:

A[0*N+0], A[0*N+1], A[0*N+0], A[0*N+1] ...(repeats N times)

A[1*N+0], A[0*N+1], A[0*N+0], A[1*N+1] ...(repeats N times)

...

...

counting misses for A (1)

```
for (int kk = 0; kk < N; kk += 2)
  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];
    }
```

access pattern for A:

A[0*N+0], A[0*N+1], A[0*N+0], A[0*N+1] …(repeats N times)

A[1*N+0], A[0*N+1], A[0*N+0], A[1*N+1] …(repeats N times)

…

A[(N-1)*N+0], A[(N-1)*N+1], A[(N-1)*N+0], A[(N-1)*N+1] …

A[0*N+2], A[0*N+3], A[0*N+2], A[0*N+3] …

…

counting misses for A (1)

```
for (int kk = 0; kk < N; kk += 2)
  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];
    }
```

access pattern for A:

A[0*N+0], A[0*N+1], A[0*N+0], A[0*N+1] …(repeats N times)

A[1*N+0], A[0*N+1], A[0*N+0], A[1*N+1] …(repeats N times)

…

A[(N-1)*N+0], A[(N-1)*N+1], A[(N-1)*N+0], A[(N-1)*N+1] …

A[0*N+2], A[0*N+3], A[0*N+2], A[0*N+3] …

…

counting misses for A (2)

A[0*N+0], A[0*N+1], A[0*N+0], A[0*N+1] …(repeats N times)
A[1*N+0], A[0*N+1], A[0*N+0], A[1*N+1] …(repeats N times)
…

…

counting misses for A (2)

A[0*N+0], A[0*N+1], A[0*N+0], A[0*N+1] …(repeats N times)
A[1*N+0], A[0*N+1], A[0*N+0], A[1*N+1] …(repeats N times)
…
A[(N-1)*N+0], A[(N-1)*N+1], A[(N-1)*N+0], A[(N-1)*N+1] …
A[0*N+2], A[0*N+3], A[0*N+2], A[0*N+3] …
…

likely cache misses: only first iterations of j loop

how many cache misses per iteration? usually one

- A[0*N+0] and A[0*N+1] usually in same cache block

counting misses for A (2)

A[0*N+0], A[0*N+1], A[0*N+0], A[0*N+1] ...(repeats N times)
A[1*N+0], A[0*N+1], A[0*N+0], A[1*N+1] ...(repeats N times)
...
A[(N-1)*N+0], A[(N-1)*N+1], A[(N-1)*N+0], A[(N-1)*N+1] ...
A[0*N+2], A[0*N+3], A[0*N+2], A[0*N+3] ...
...

likely cache misses: only first iterations of j loop

how many cache misses per iteration? usually one

A[0*N+0] and A[0*N+1] usually in same cache block

about $\frac{N}{2} \cdot N$ misses total

counting misses for B (1)

```
for (int kk = 0; kk < N; kk += 2)
  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];
    }
```

access pattern for B:

B[0*N+0], B[1*N+0], ...B[0*N+(N-1)], B[1*N+(N-1)]

B[2*N+0], B[3*N+0], ...B[2*N+(N-1)], B[3*N+(N-1)]

B[4*N+0], B[5*N+0], ...B[4*N+(N-1)], B[5*N+(N-1)]

...

B[0*N+0], B[1*N+0], ...B[0*N+(N-1)], B[1*N+(N-1)]

...

counting misses for B (2)

access pattern for B:

B[0*N+0], B[1*N+0], …B[0*N+(N-1)], B[1*N+(N-1)]
B[2*N+0], B[3*N+0], …B[2*N+(N-1)], B[3*N+(N-1)]
B[4*N+0], B[5*N+0], …B[4*N+(N-1)], B[5*N+(N-1)]

…

B[0*N+0], B[1*N+0], …B[0*N+(N-1)], B[1*N+(N-1)]

…

counting misses for B (2)

access pattern for B:

B[0*N+0], B[1*N+0], …B[0*N+(N-1)], B[1*N+(N-1)]
B[2*N+0], B[3*N+0], …B[2*N+(N-1)], B[3*N+(N-1)]
B[4*N+0], B[5*N+0], …B[4*N+(N-1)], B[5*N+(N-1)]

…

B[0*N+0], B[1*N+0], …B[0*N+(N-1)], B[1*N+(N-1)]

…

likely cache misses: any access, each time

counting misses for B (2)

access pattern for B:

B[0*N+0], B[1*N+0], ...B[0*N+(N-1)], B[1*N+(N-1)]
B[2*N+0], B[3*N+0], ...B[2*N+(N-1)], B[3*N+(N-1)]
B[4*N+0], B[5*N+0], ...B[4*N+(N-1)], B[5*N+(N-1)]
...
B[0*N+0], B[1*N+0], ...B[0*N+(N-1)], B[1*N+(N-1)]
...

likely cache misses: any access, each time

how many cache misses per iteration? equal to # cache blocks in 2 rows

counting misses for B (2)

access pattern for B:
B[0*N+0], B[1*N+0], ...B[0*N+(N-1)], B[1*N+(N-1)]
B[2*N+0], B[3*N+0], ...B[2*N+(N-1)], B[3*N+(N-1)]
B[4*N+0], B[5*N+0], ...B[4*N+(N-1)], B[5*N+(N-1)]
...
B[0*N+0], B[1*N+0], ...B[0*N+(N-1)], B[1*N+(N-1)]
...

likely cache misses: any access, each time

how many cache misses per iteration? equal to # cache blocks in 2 rows

about $\frac{N}{2} \cdot N \cdot \frac{2N}{\text{block size}} = N^3 \div$ block size misses

simple blocking – counting misses

```
for (int kk = 0; kk < N; kk += 2)
  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];
    }
```

$\frac{N}{2} \cdot N$ j-loop executions and (assuming N large):

about 1 misses from A per j-loop

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

about $2N \div$ block size misses from B per j-loop

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

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

  $N^3 \div (2 \cdot$ block size$)$ total misses (before: $N^3 \div$ block size)

simple blocking – counting misses

```
for (int kk = 0; kk < N; kk += 2)
  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];
    }
```

$\frac{N}{2} \cdot N$ j-loop executions and (assuming N large):

about 1 misses from A per j-loop

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

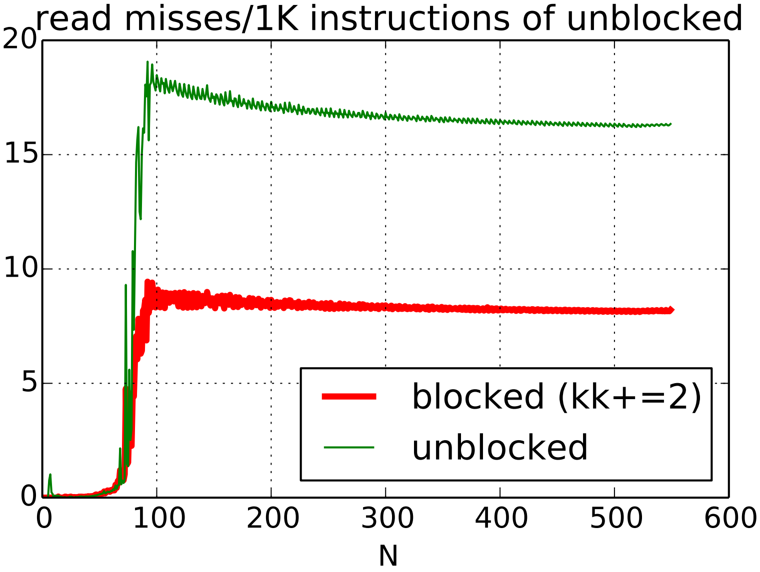
about $2N \div$ block size misses from B per j-loop

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

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

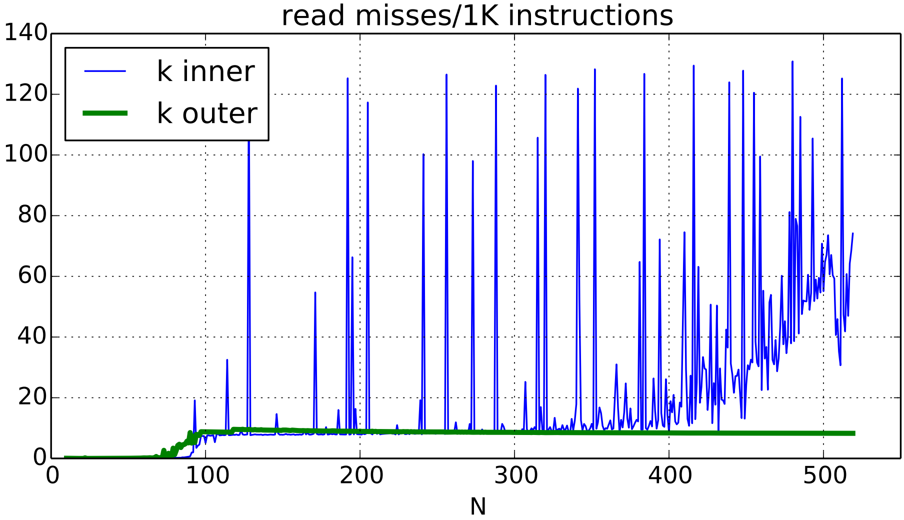
> $N^3 \div (2 \cdot$ block size$)$ total misses (before: $N^3 \div$ block size)

improvement in read misses

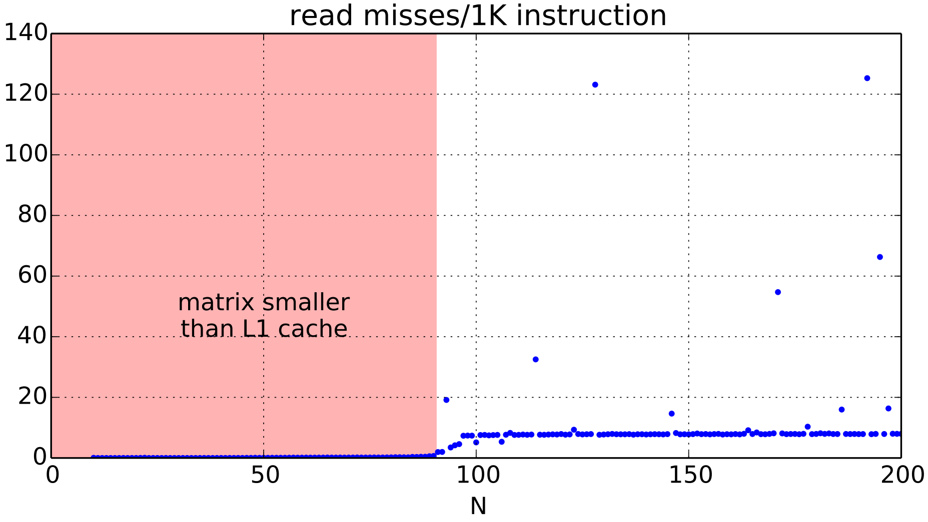


backup slides

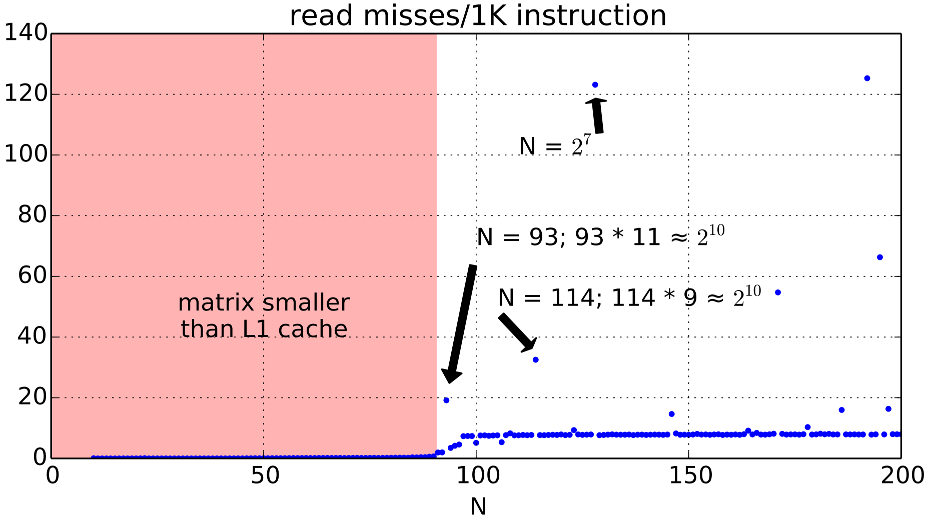
L1 misses (with A=B)



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)