

caching

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)

Exercise: 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]
```

conversion 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

5x5 array and 4-element cache blocks

<code>array[0*5 + 0]</code>	<code>array[0*5 + 1]</code>	<code>array[0*5 + 2]</code>	<code>array[0*5 + 3]</code>	<code>array[0*5 + 4]</code>
<code>array[1*5 + 0]</code>	<code>array[1*5 + 1]</code>	<code>array[1*5 + 2]</code>	<code>array[1*5 + 3]</code>	<code>array[1*5 + 4]</code>
<code>array[2*5 + 0]</code>	<code>array[2*5 + 1]</code>	<code>array[2*5 + 2]</code>	<code>array[2*5 + 3]</code>	<code>array[2*5 + 4]</code>
<code>array[3*5 + 0]</code>	<code>array[3*5 + 1]</code>	<code>array[3*5 + 2]</code>	<code>array[3*5 + 3]</code>	<code>array[3*5 + 4]</code>
<code>array[4*5 + 0]</code>	<code>array[4*5 + 1]</code>	<code>array[4*5 + 2]</code>	<code>array[4*5 + 3]</code>	<code>array[4*5 + 4]</code>

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

<code>array[0*5 + 0]</code>	<code>array[0*5 + 1]</code>	<code>array[0*5 + 2]</code>	<code>array[0*5 + 3]</code>	<code>array[0*5 + 4]</code>
<code>array[1*5 + 0]</code>	<code>array[1*5 + 1]</code>	<code>array[1*5 + 2]</code>	<code>array[1*5 + 3]</code>	<code>array[1*5 + 4]</code>
<code>array[2*5 + 0]</code>	<code>array[2*5 + 1]</code>	<code>array[2*5 + 2]</code>	<code>array[2*5 + 3]</code>	<code>array[2*5 + 4]</code>
<code>array[3*5 + 0]</code>	<code>array[3*5 + 1]</code>	<code>array[3*5 + 2]</code>	<code>array[3*5 + 3]</code>	<code>array[3*5 + 4]</code>
<code>array[4*5 + 0]</code>	<code>array[4*5 + 1]</code>	<code>array[4*5 + 2]</code>	<code>array[4*5 + 3]</code>	<code>array[4*5 + 4]</code>

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
→ better performance from reusing rows

matrix multiply

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

```
/* 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

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

```
/* 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

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

```
/* 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: $C_i[j] = A_i[k]B_k[j]$

$kj[i]$. *fragment fragment – index = 2. custom. myem – only* order:

$C_i[j]$. *fragment fragment – index = 2. custom. myem – only*,

$B_k[j]$. *fragment fragment – index = 2. custom. myem – only* have spatial locality

kij order: $A_i[k]$ has temporal locality

... better than ...

$ij[k]$. *fragment fragment – index = 2. custom. myem – only* order:

$A_i[k]$. *fragment fragment – index = 2. custom. myem – only* has spatial locality

ijk order: $C_i[j]$ has temporal locality

loop orders and locality

loop body: $C_i[j] = A_i[k]B_k[j]$

$kj[i]$. *fragment fragment – index = 2. custom. myem – only* order:

$C_i[j]$. *fragment fragment – index = 2. custom. myem – only*,

$B_k[j]$. *fragment fragment – index = 2. custom. myem – only* have **spatial locality**

kij order: $A_i[k]$ has **temporal locality**

... better than ...

$ij[k]$. *fragment fragment – index = 2. custom. myem – only* order:

$A_i[k]$. *fragment fragment – index = 2. custom. myem – only* has spatial locality

ijk order: $C_i[j]$ has temporal locality

matrix multiply

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

```
/* 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

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

```
/* 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

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

```
/* 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?

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

```
/* 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?

which is better?

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

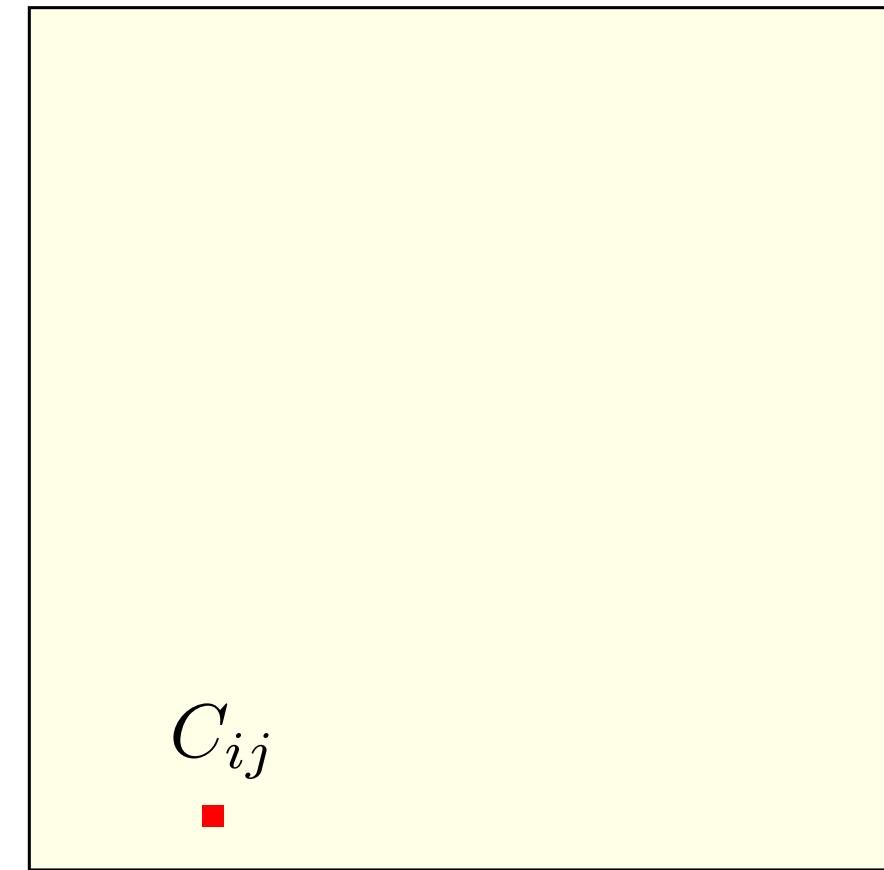
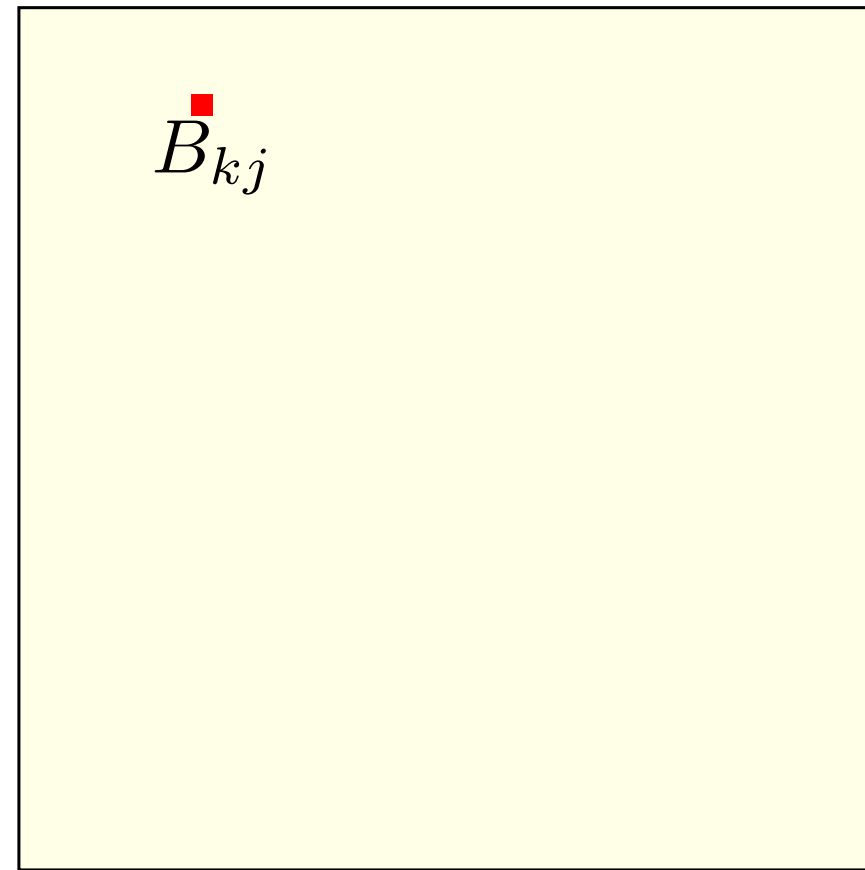
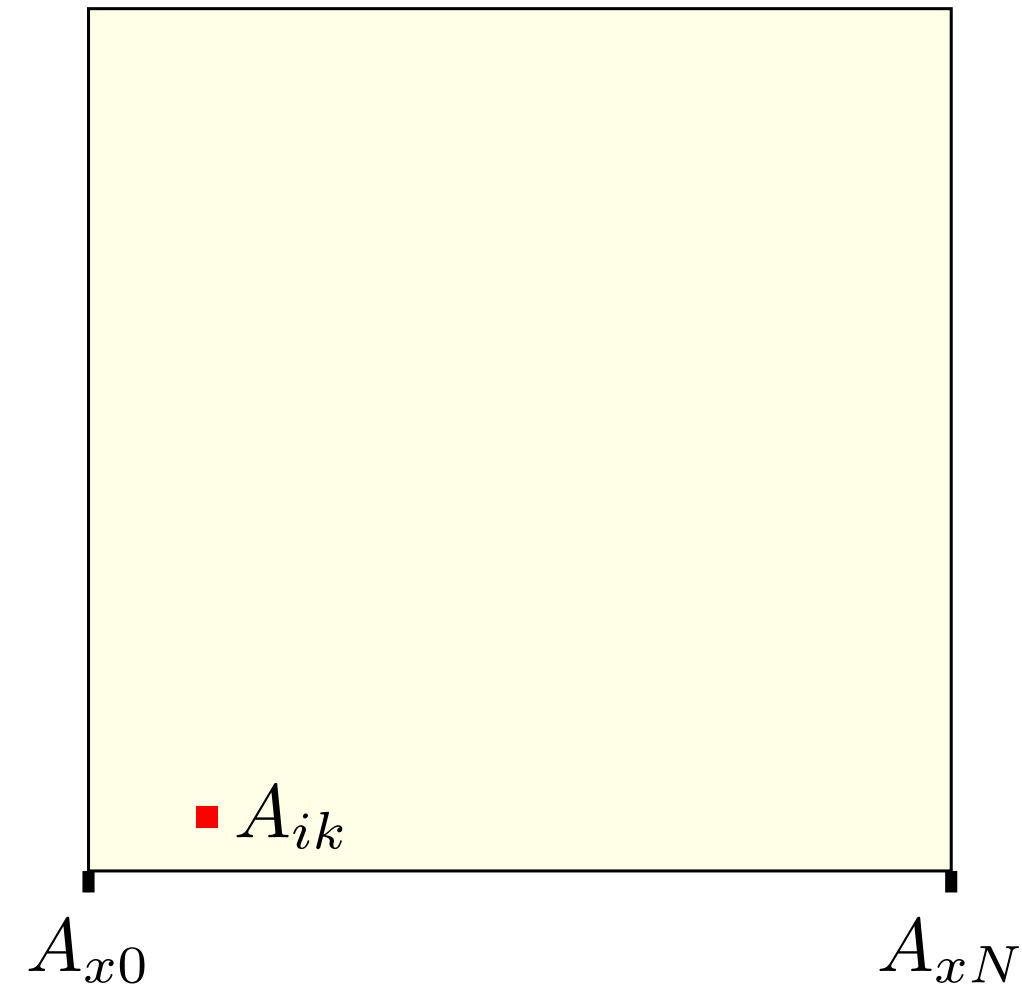
```
/* version 1: inner loop is k, middle is j*/  
for (int i = 0; i < N; ++i)  
  for (int j = 0; j < N; ++j)  
    for (int k = 0; k < N; ++k)  
      C[i*N+j] += A[i * N + k] * B[k * N + j];
```

```
/* 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

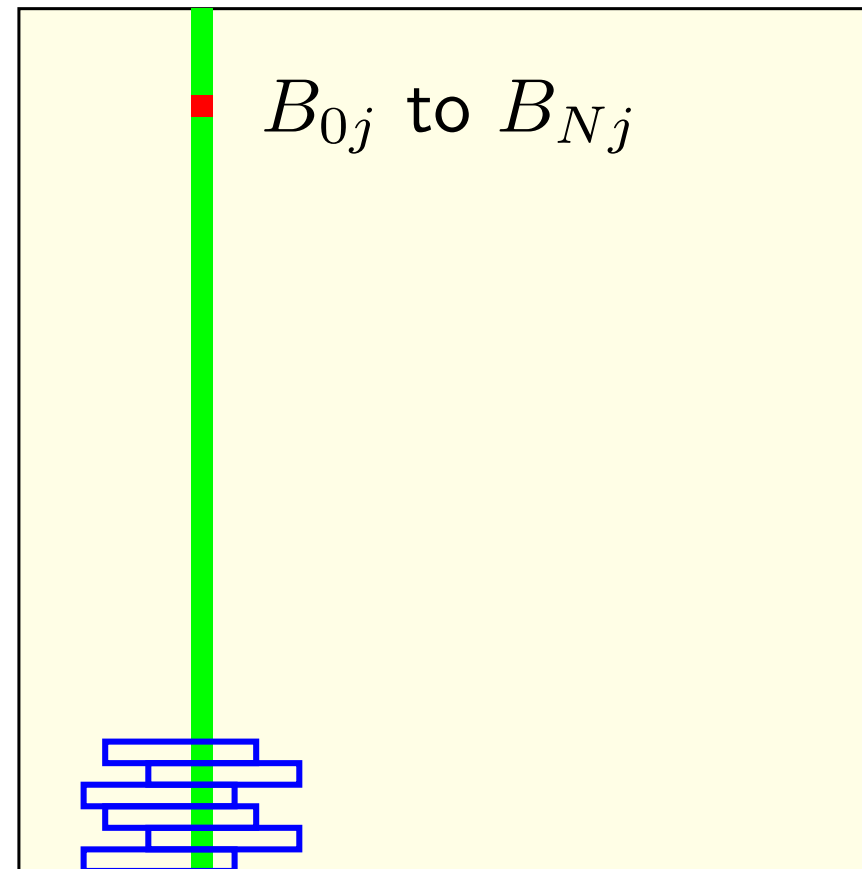
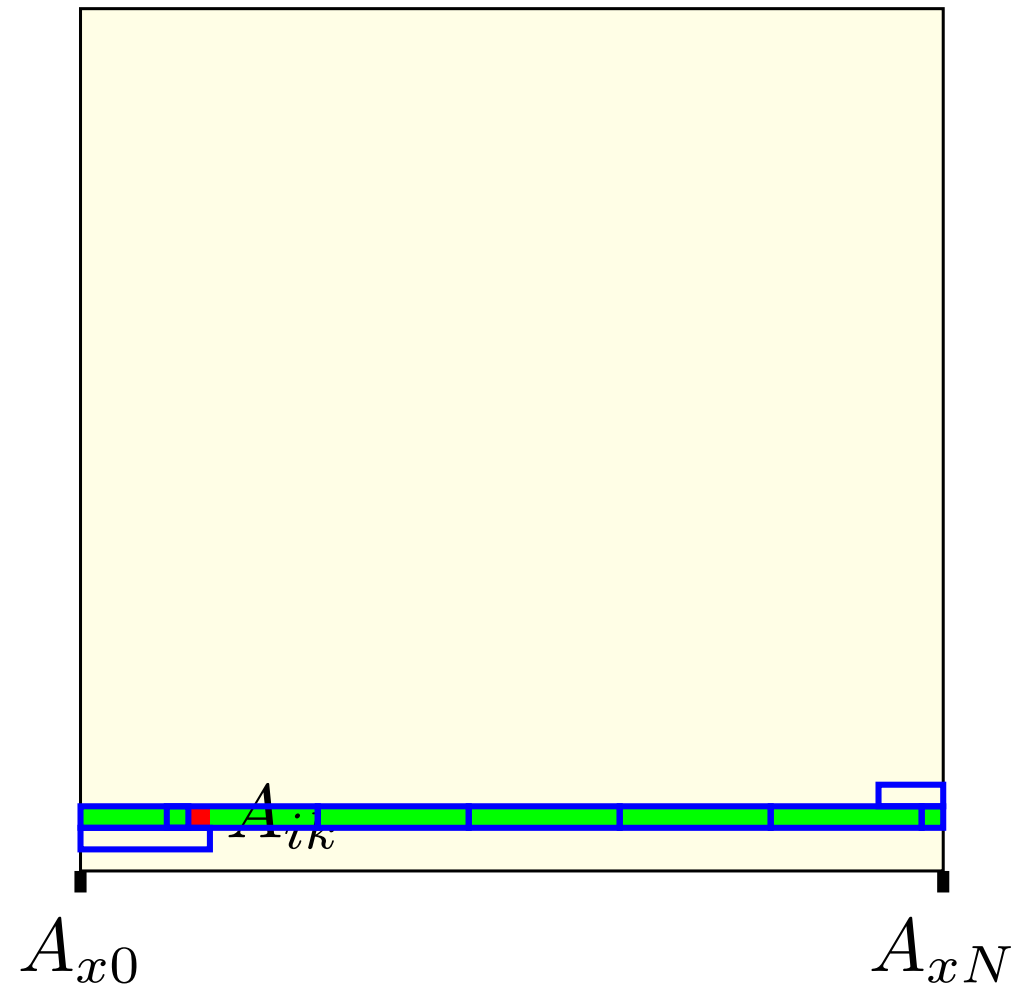
array usage: *ijk* order



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 :

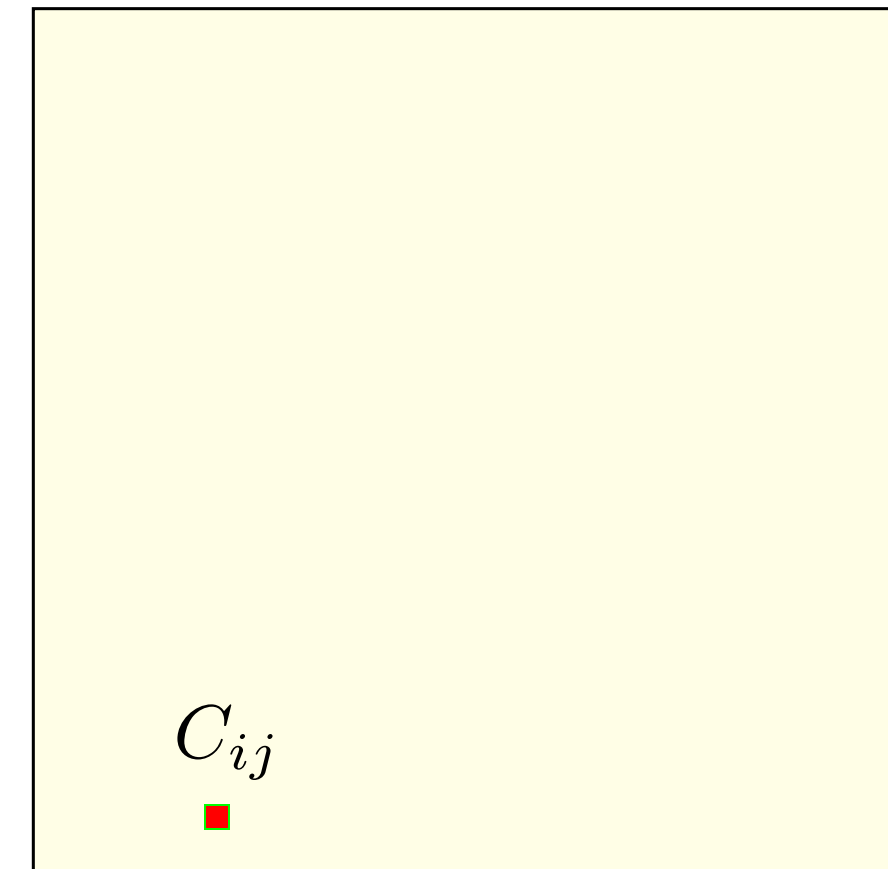
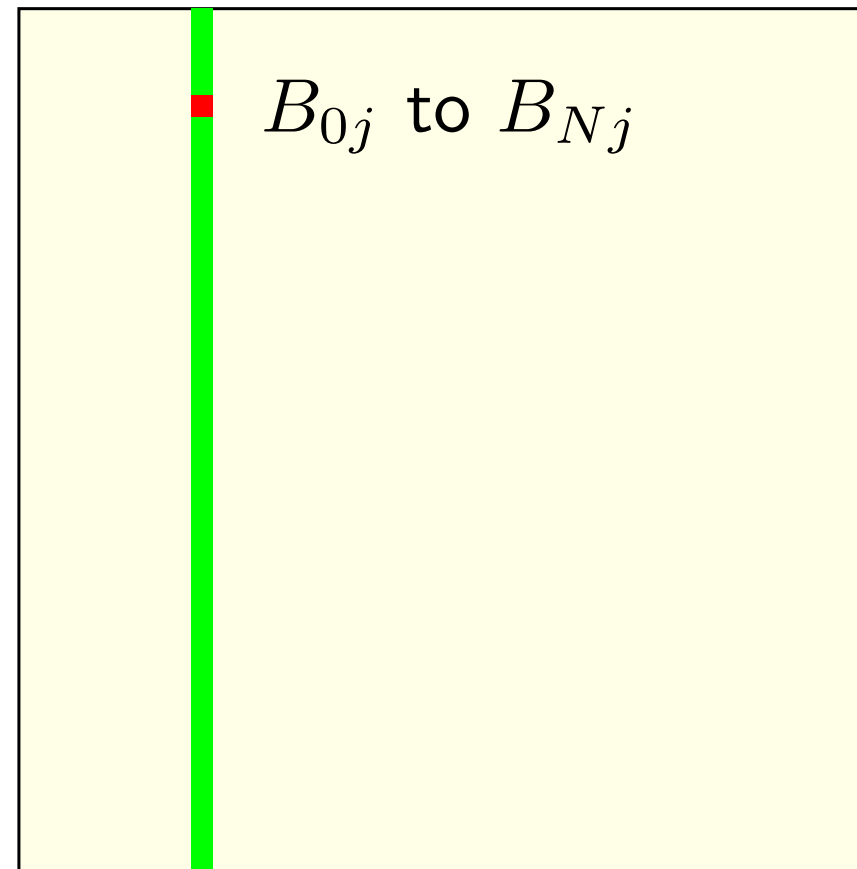
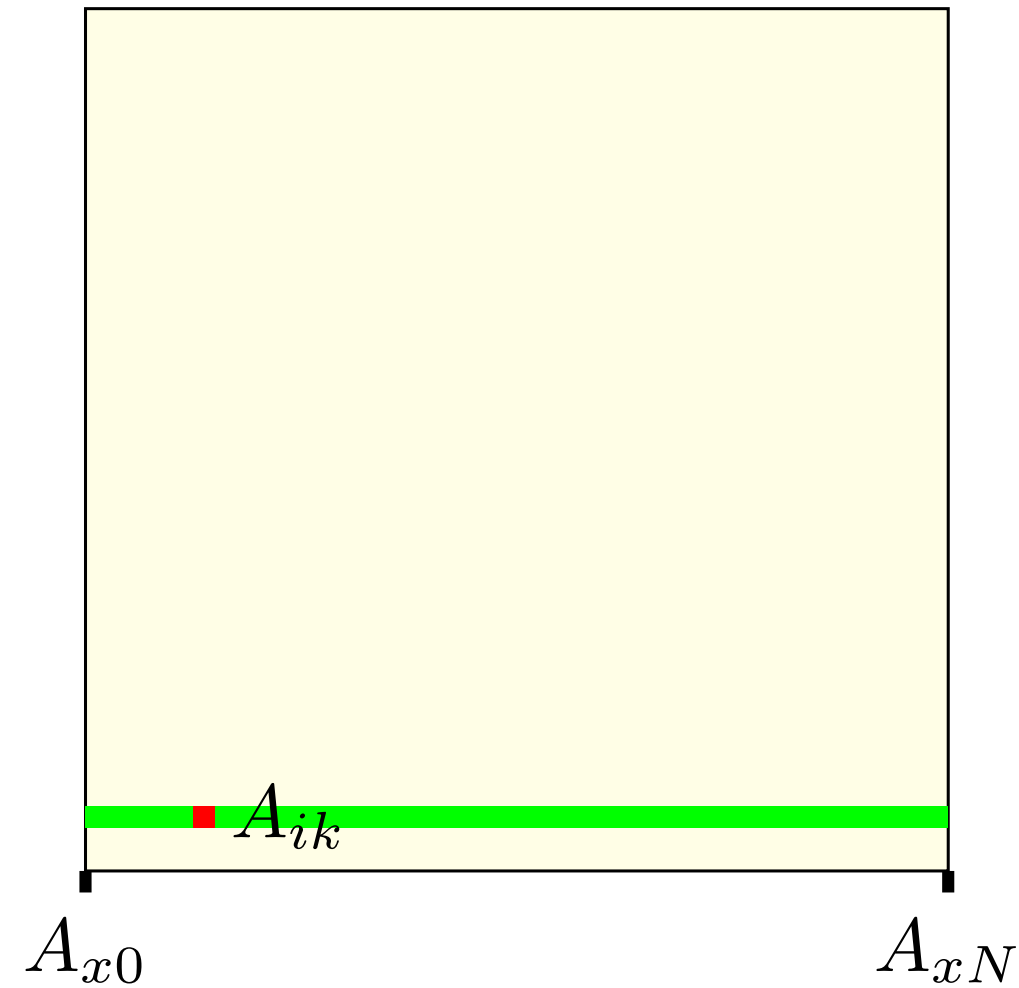
for all j :

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 :

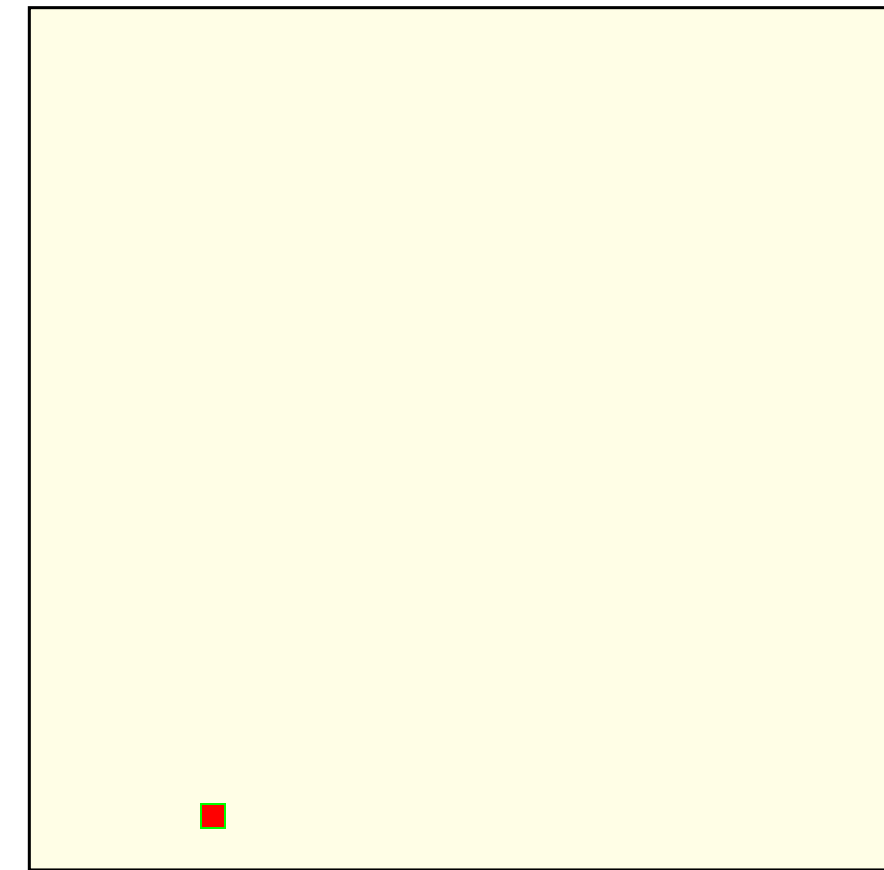
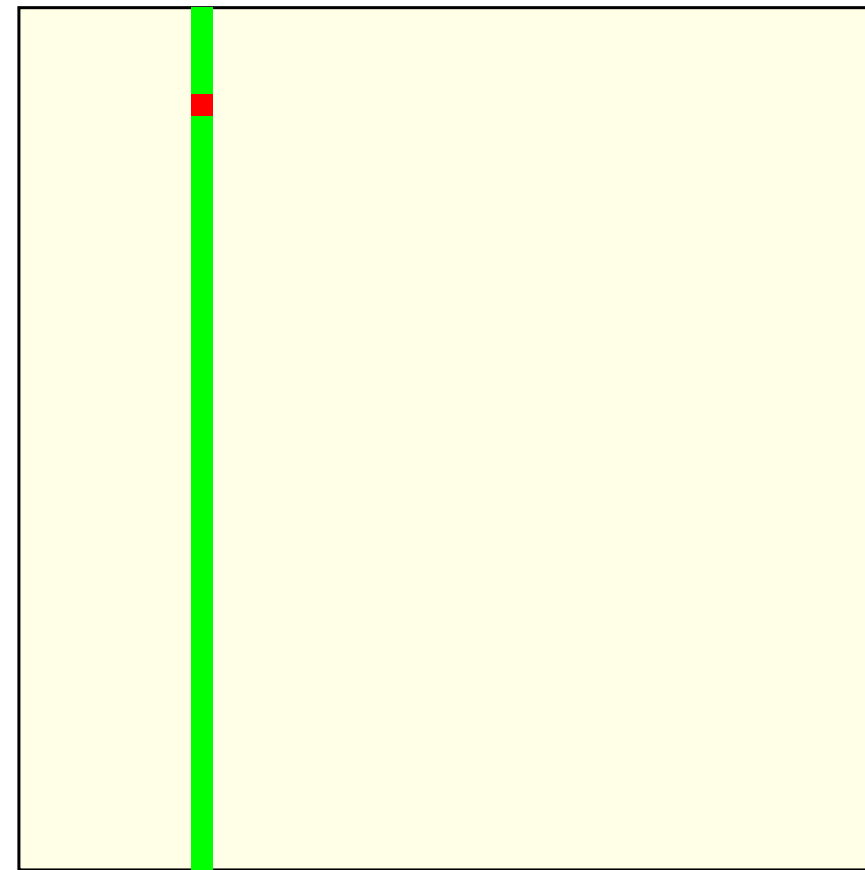
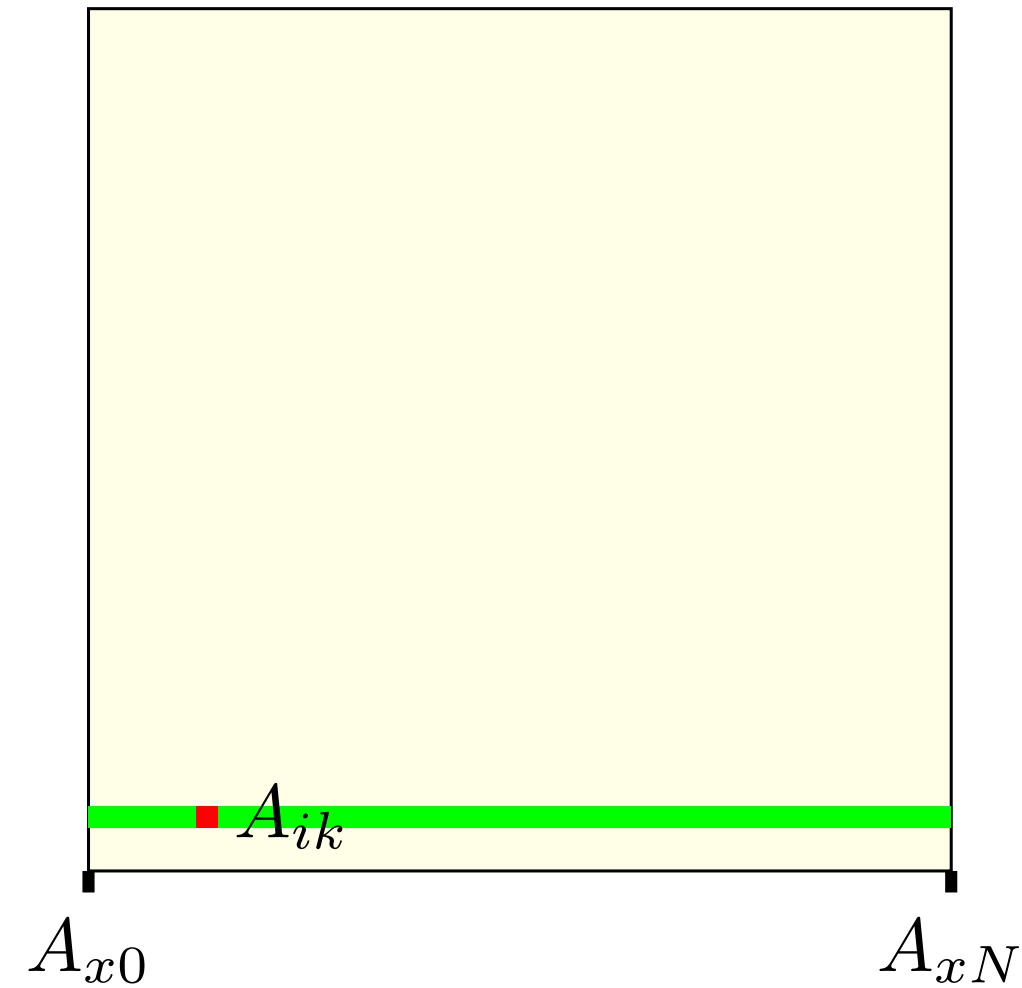
for all j :

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 :

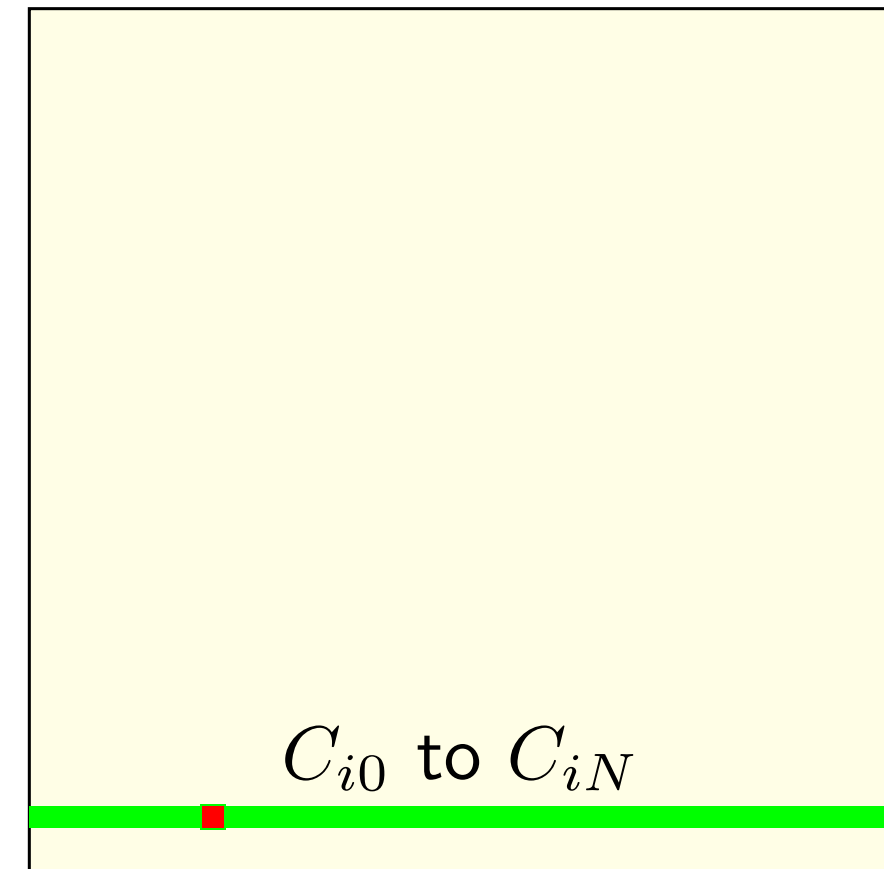
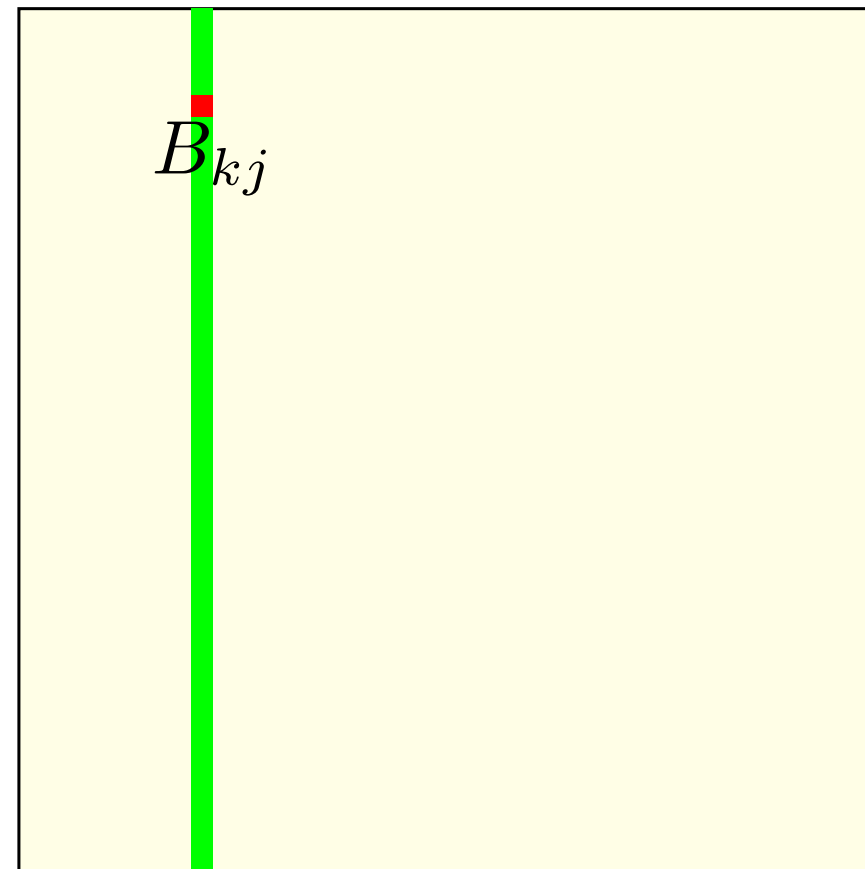
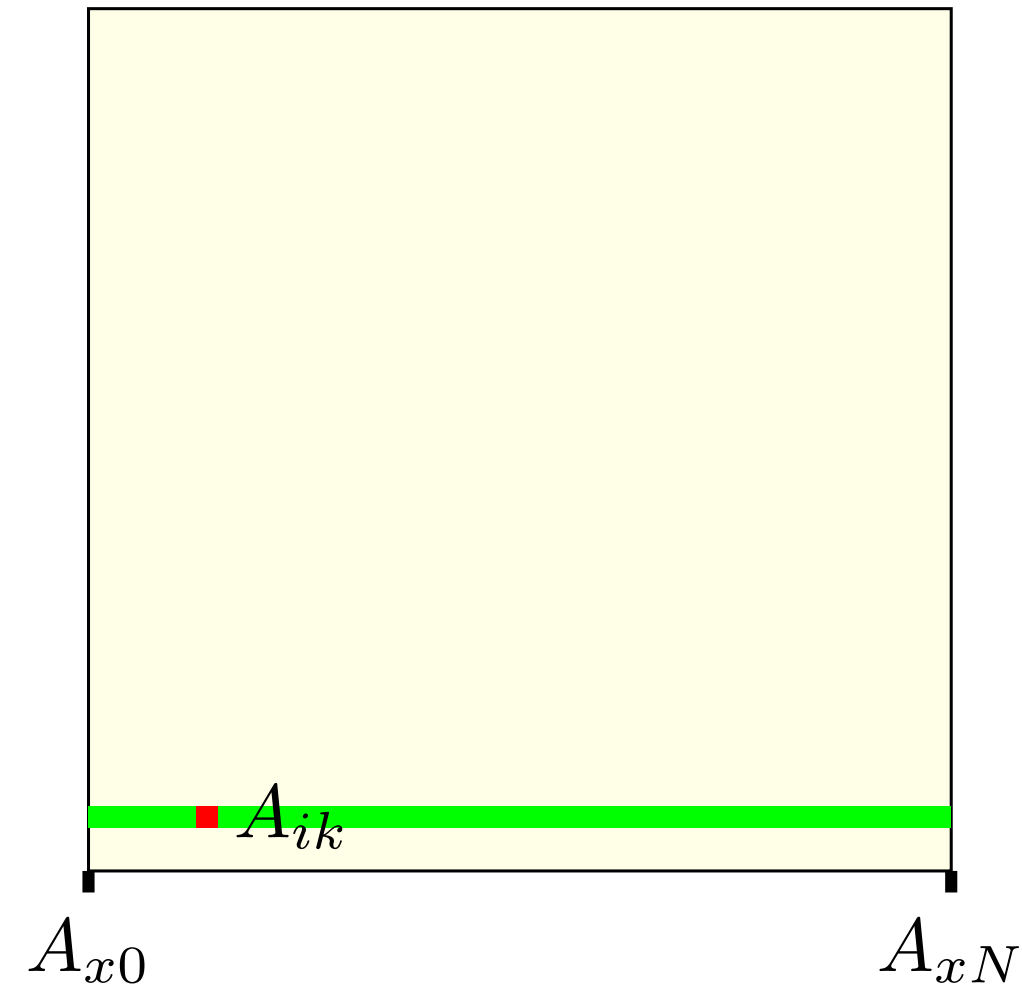
for all j :

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 :

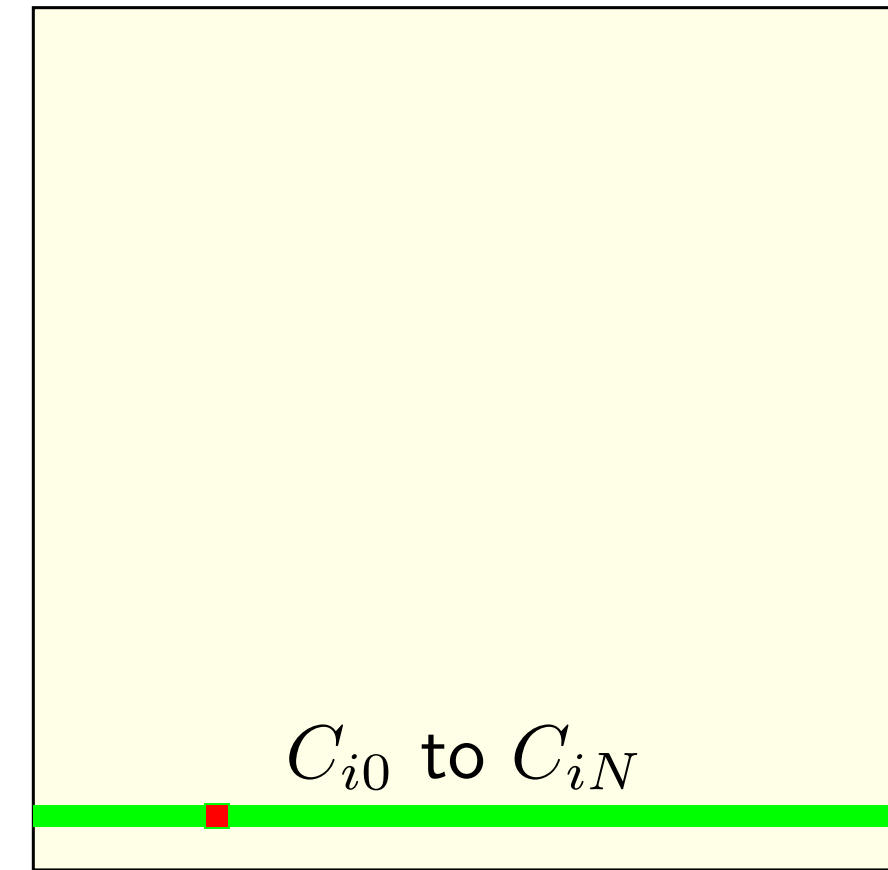
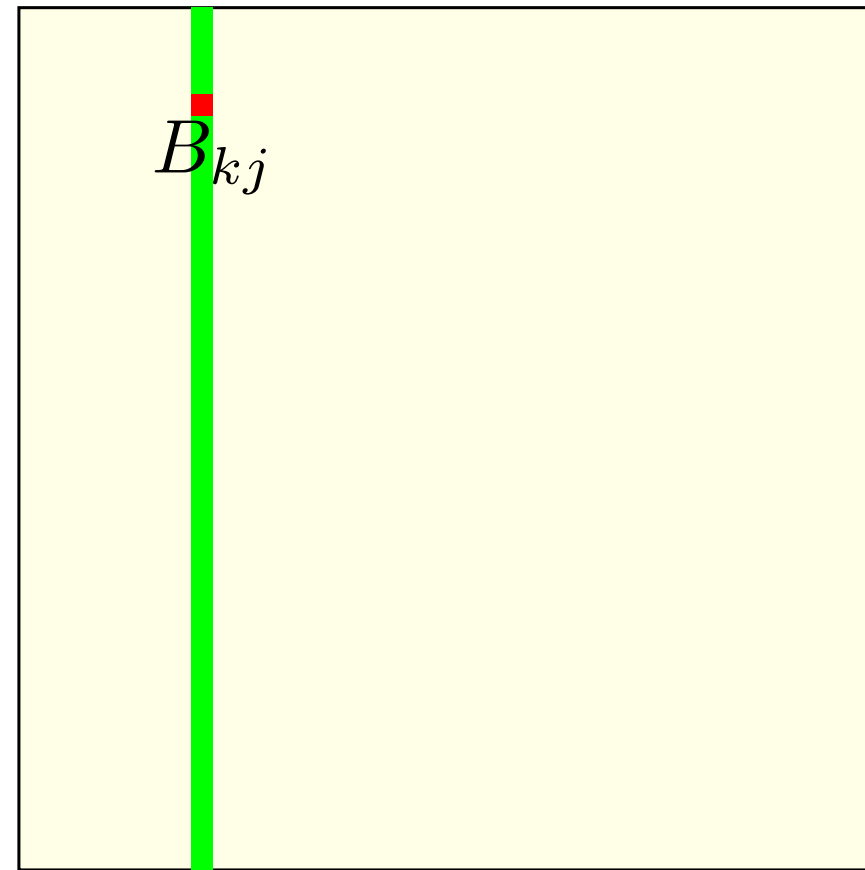
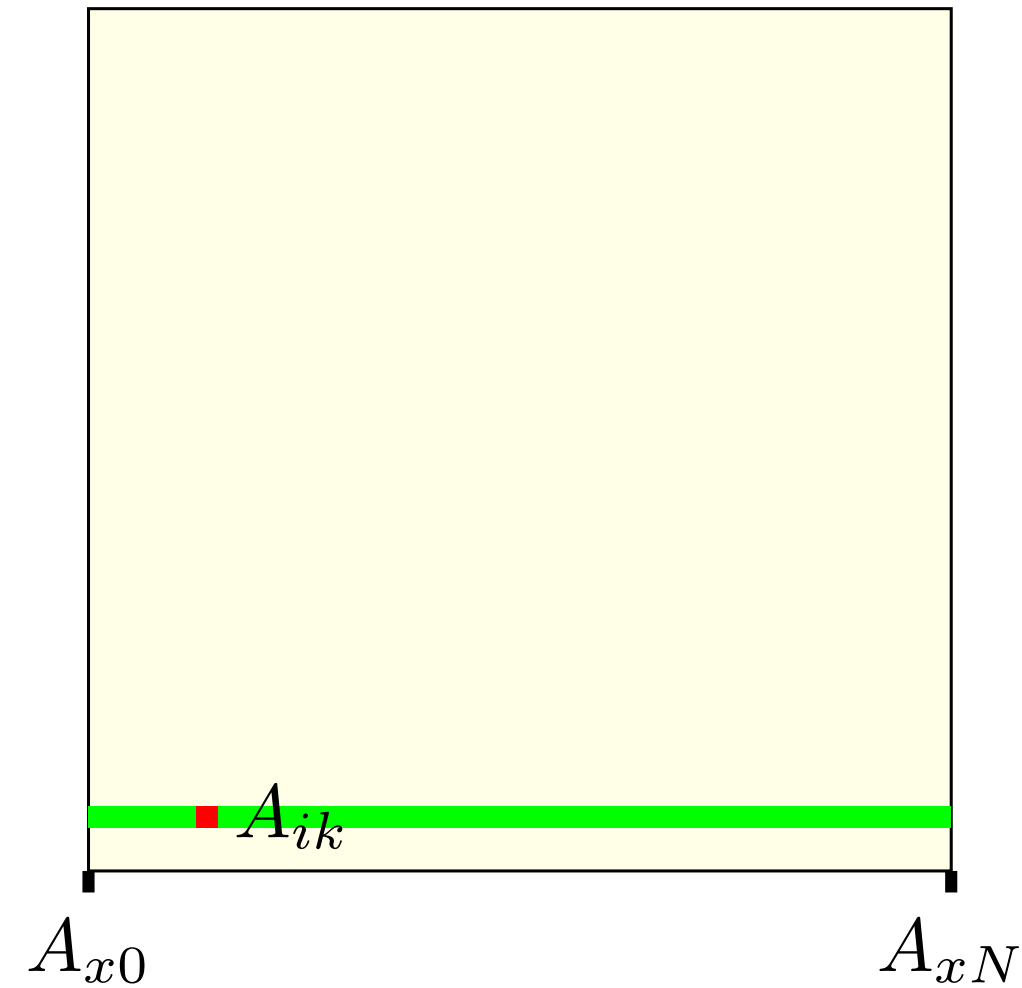
for all j :

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: *ijk* order



for all i :

for all j :

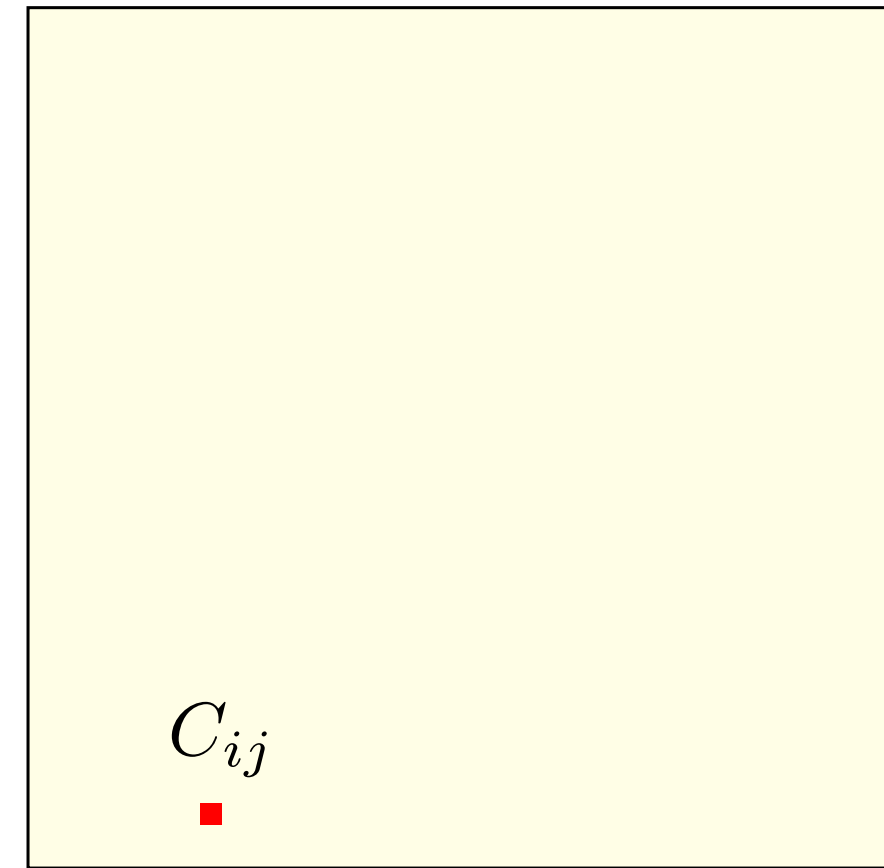
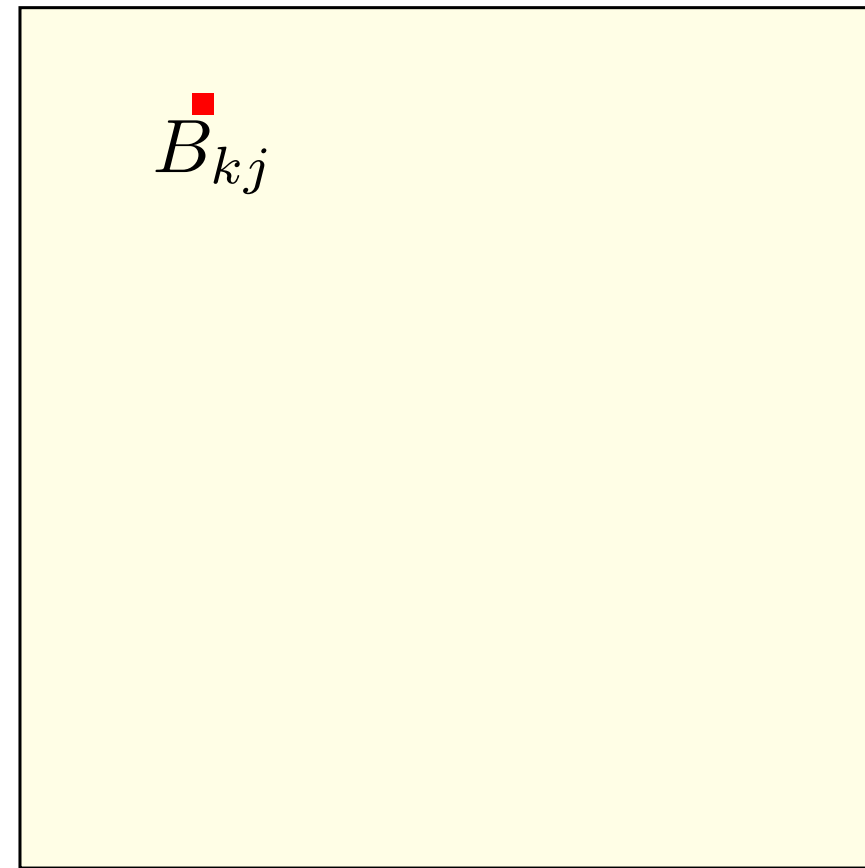
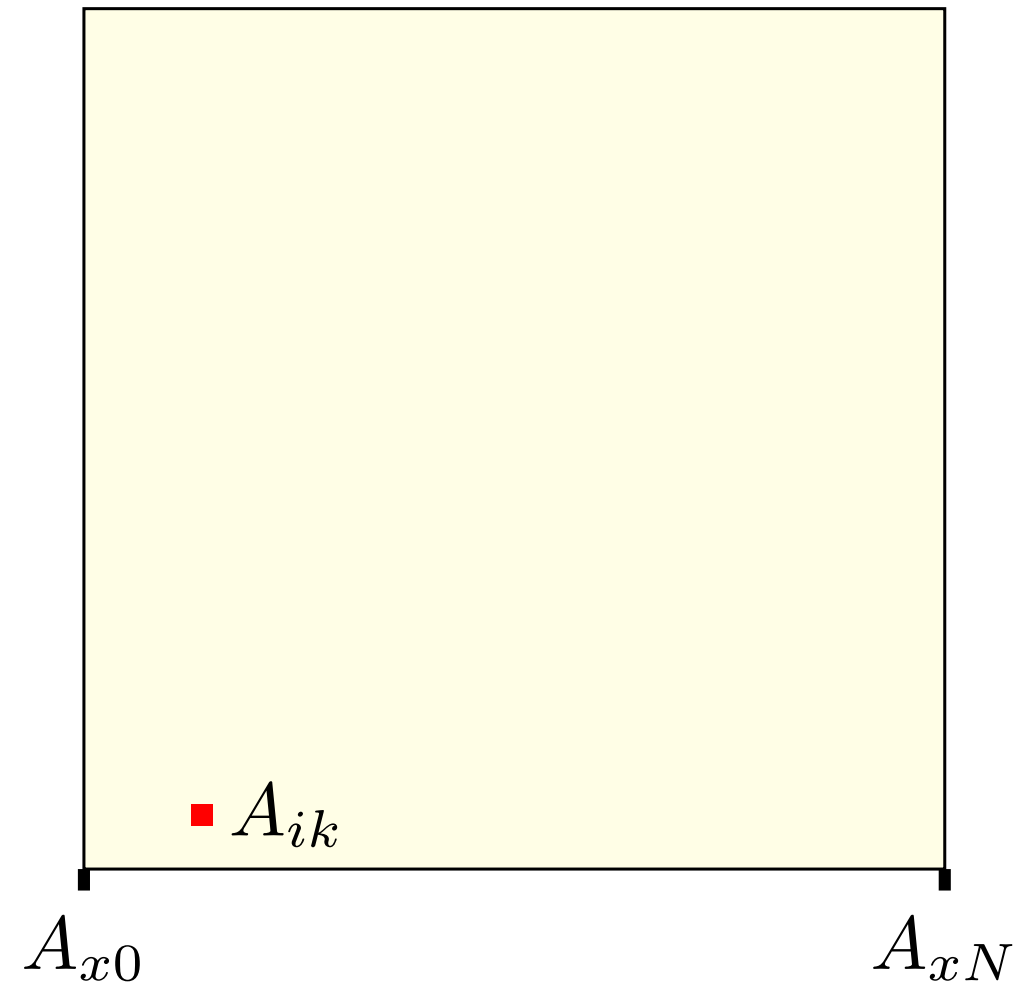
for all k :

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

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

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

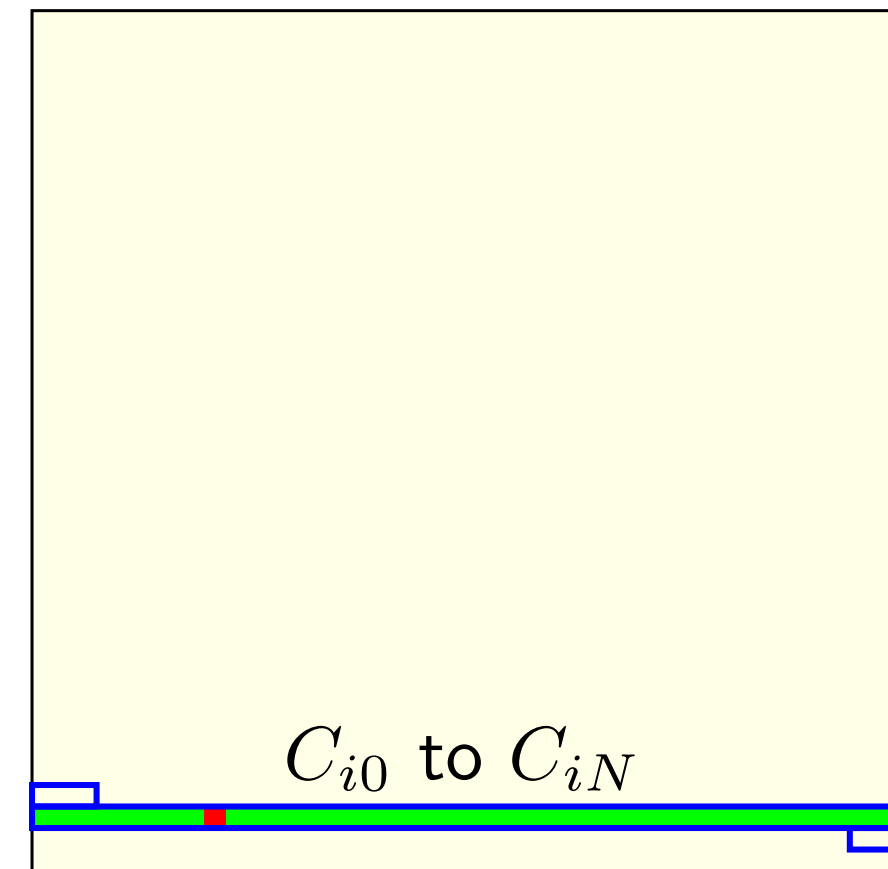
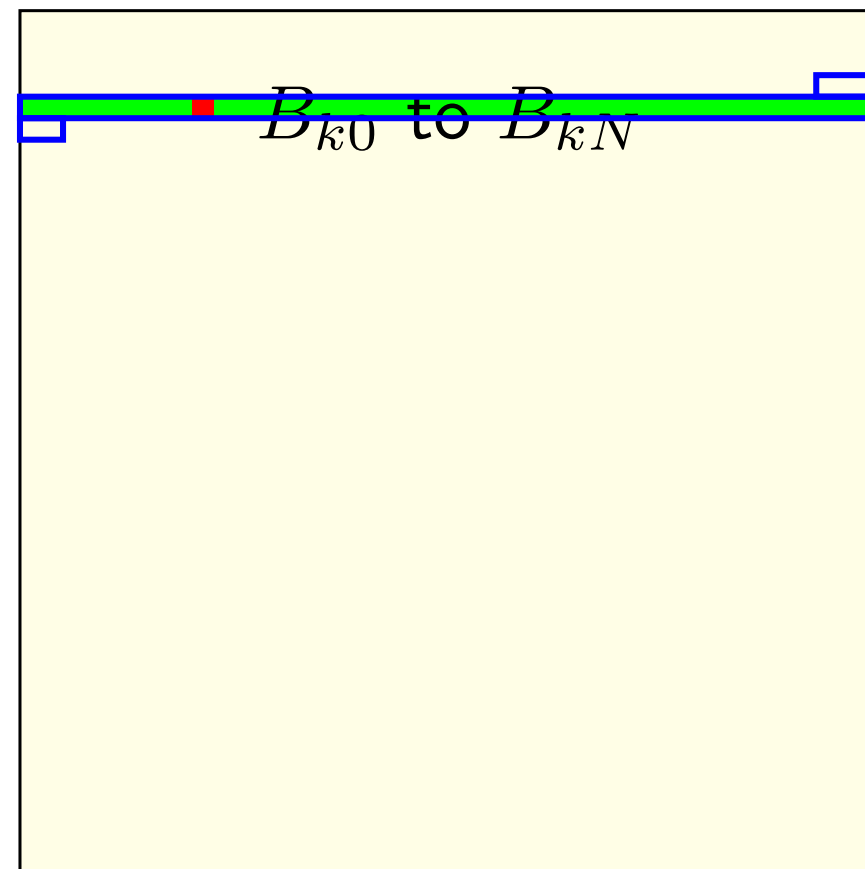
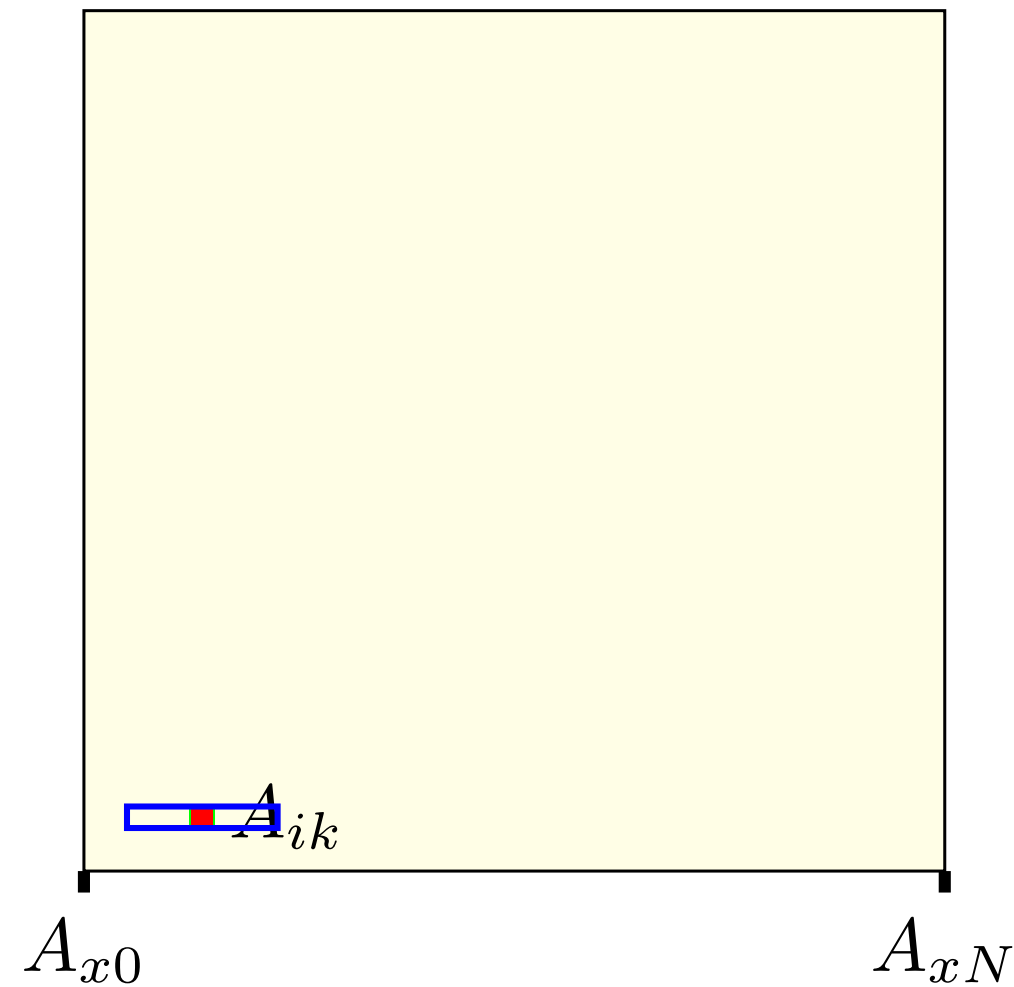
(but using $C_{i,j+1}$ right after)

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 :

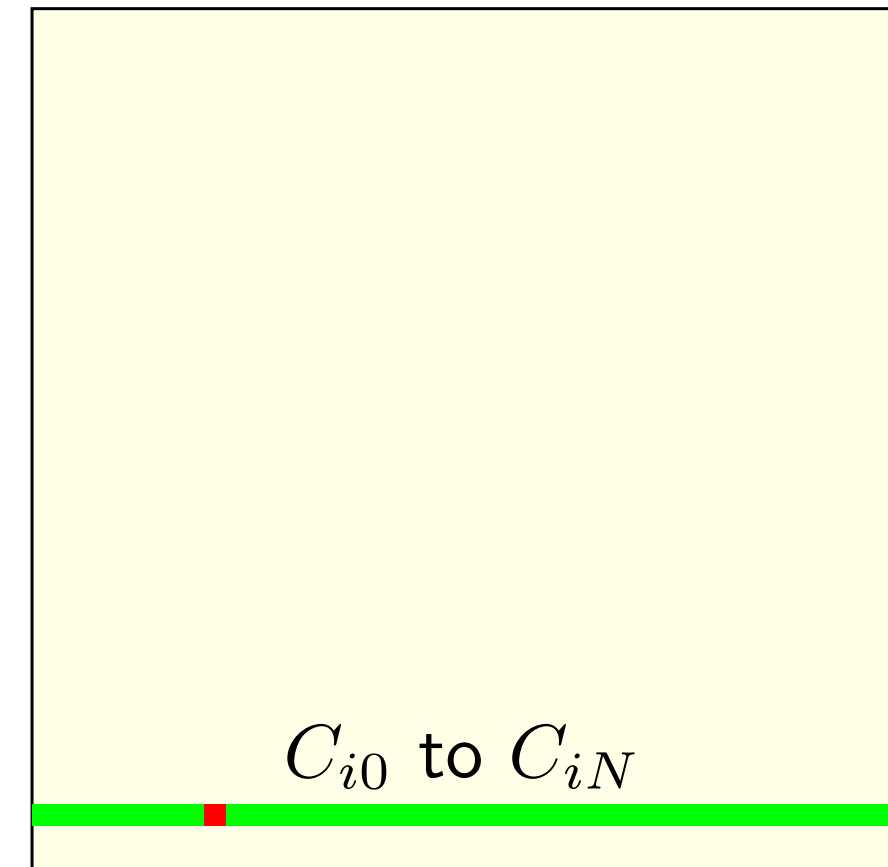
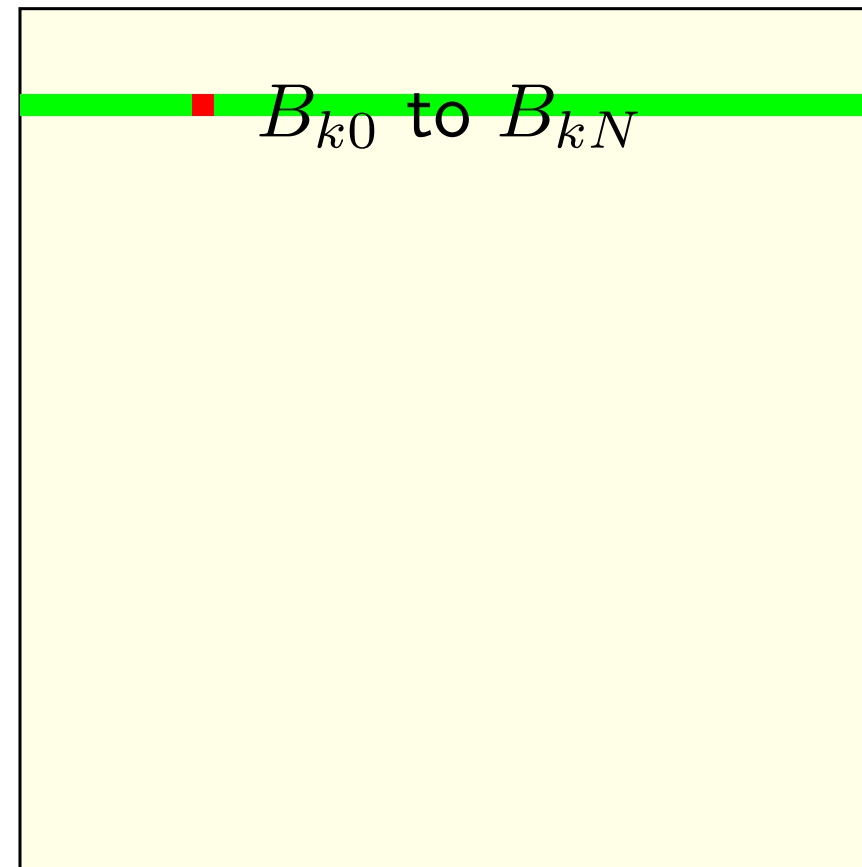
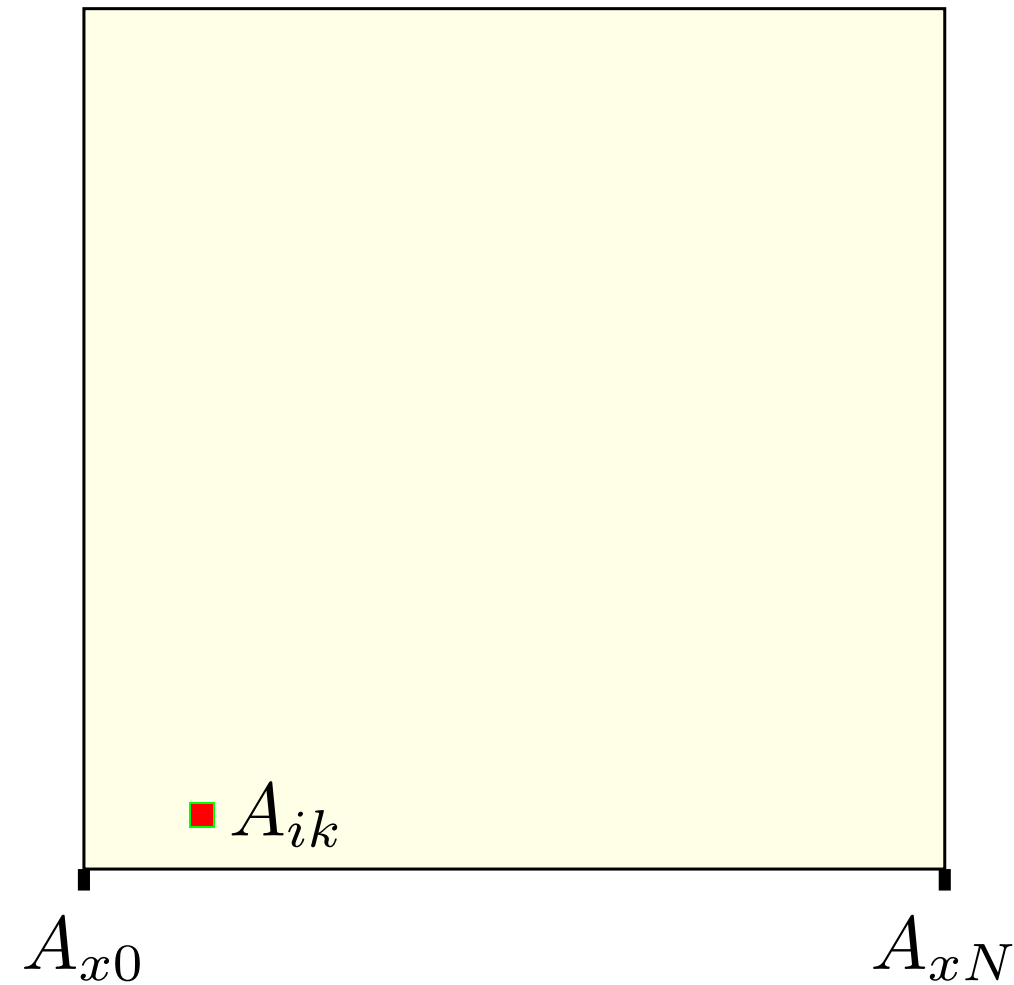
for all i :

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 :

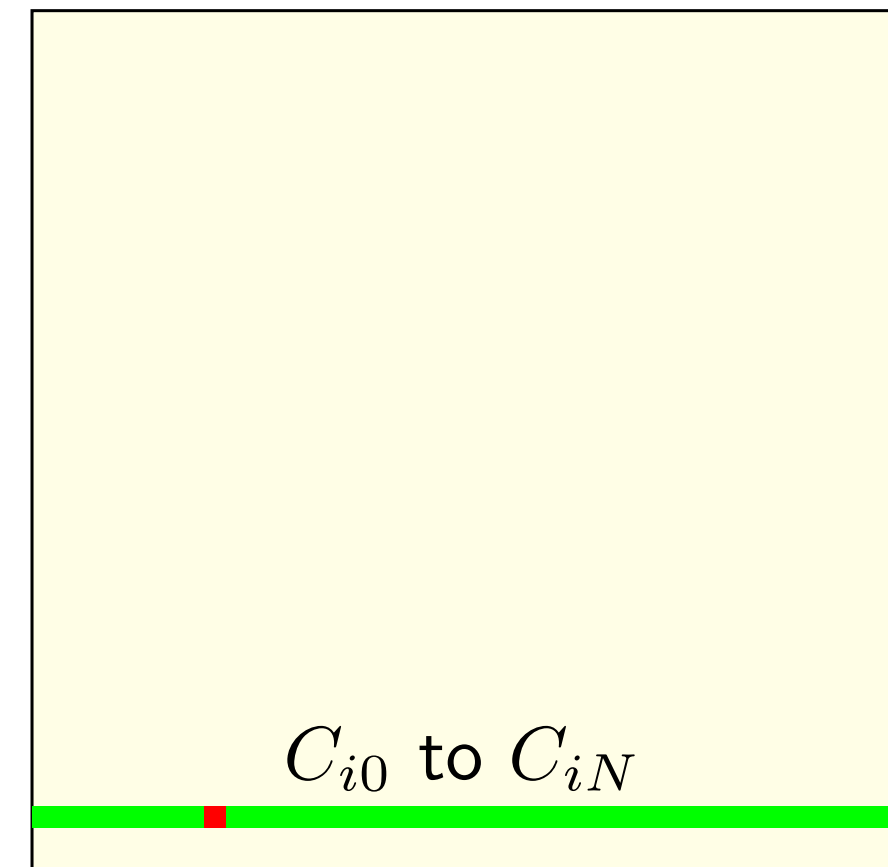
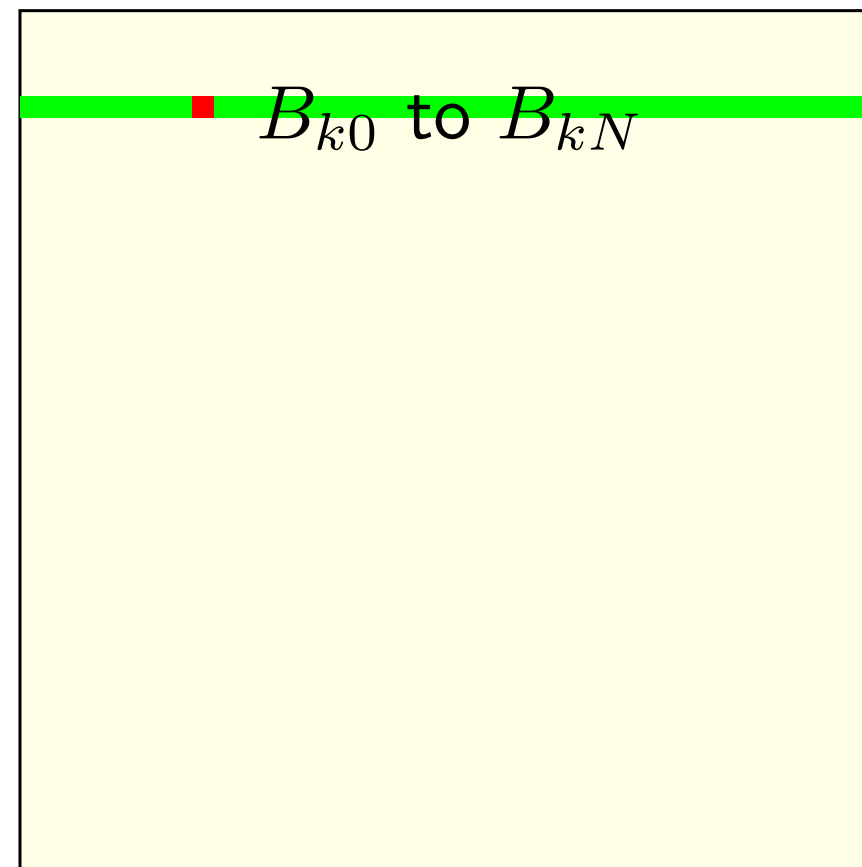
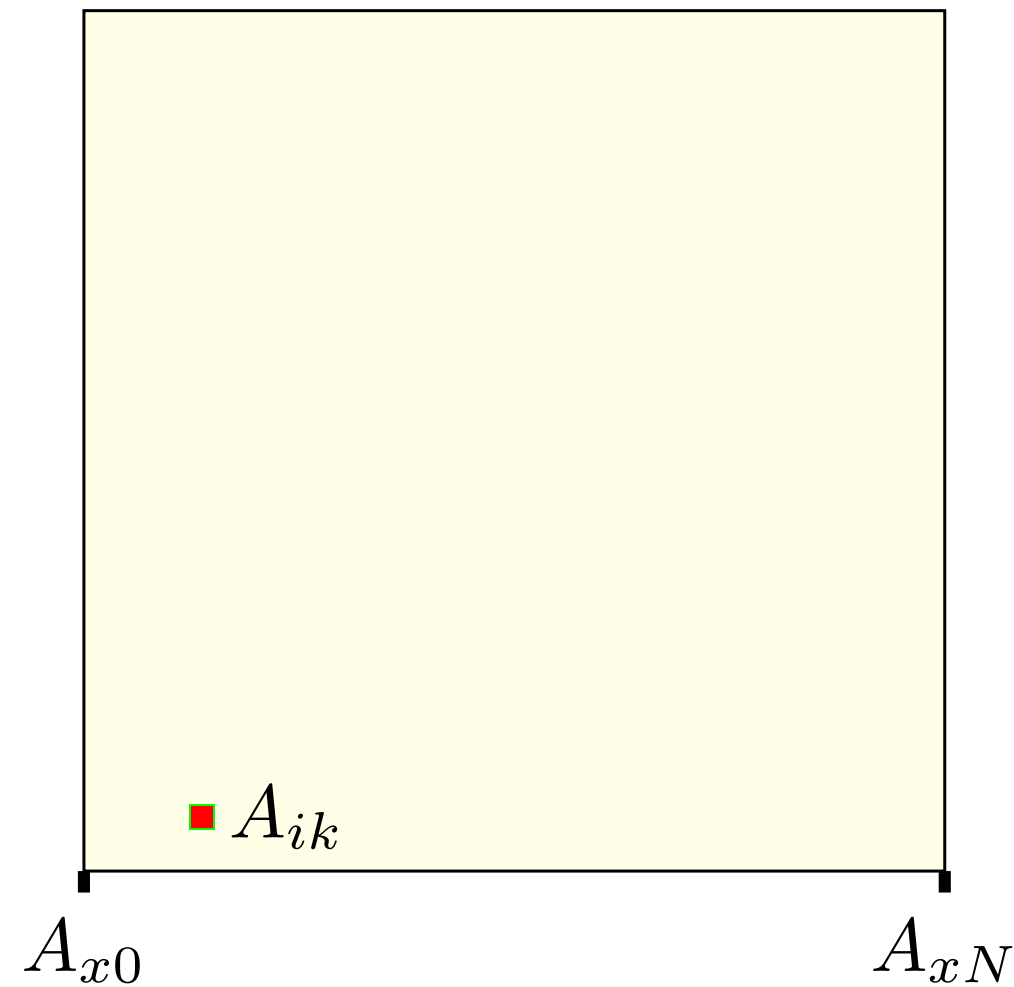
for all i :

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 :

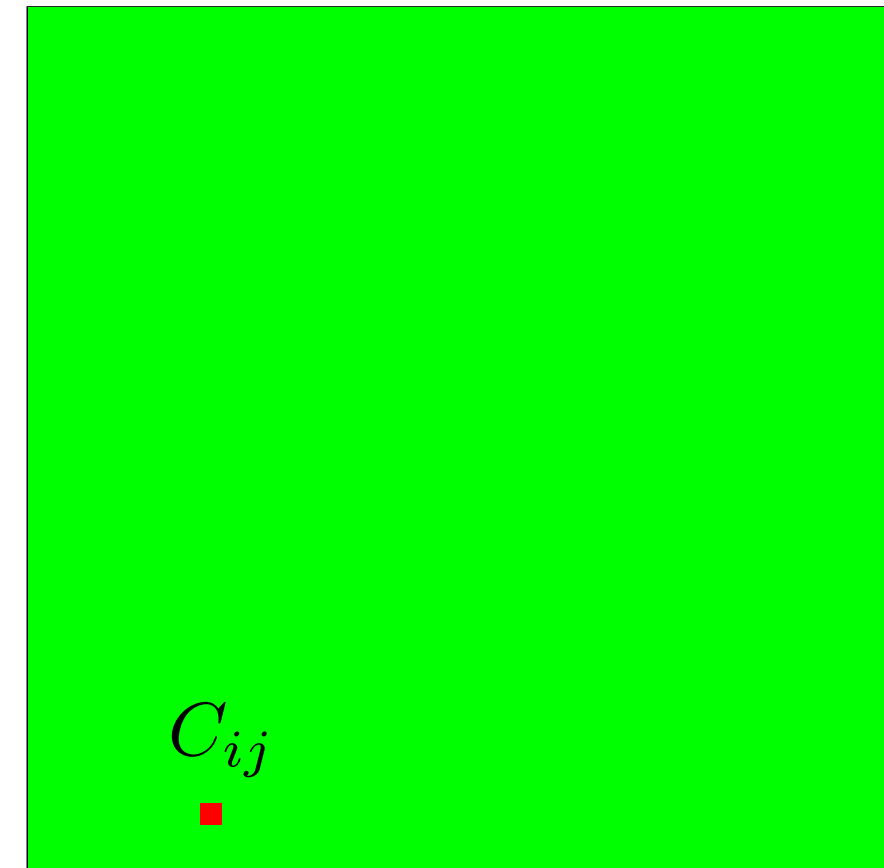
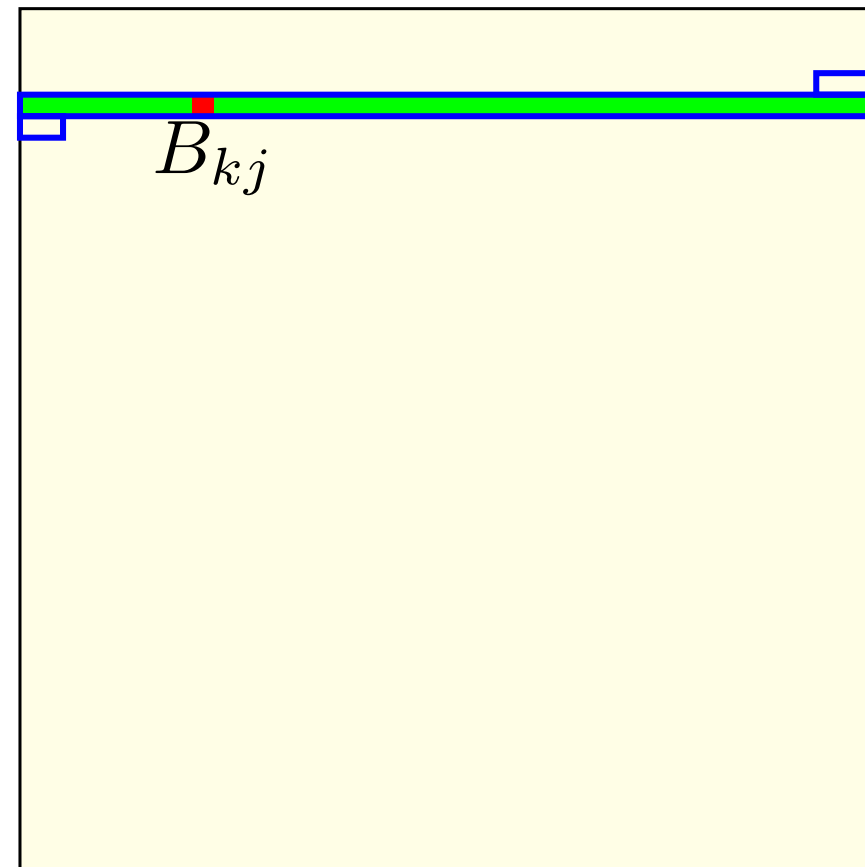
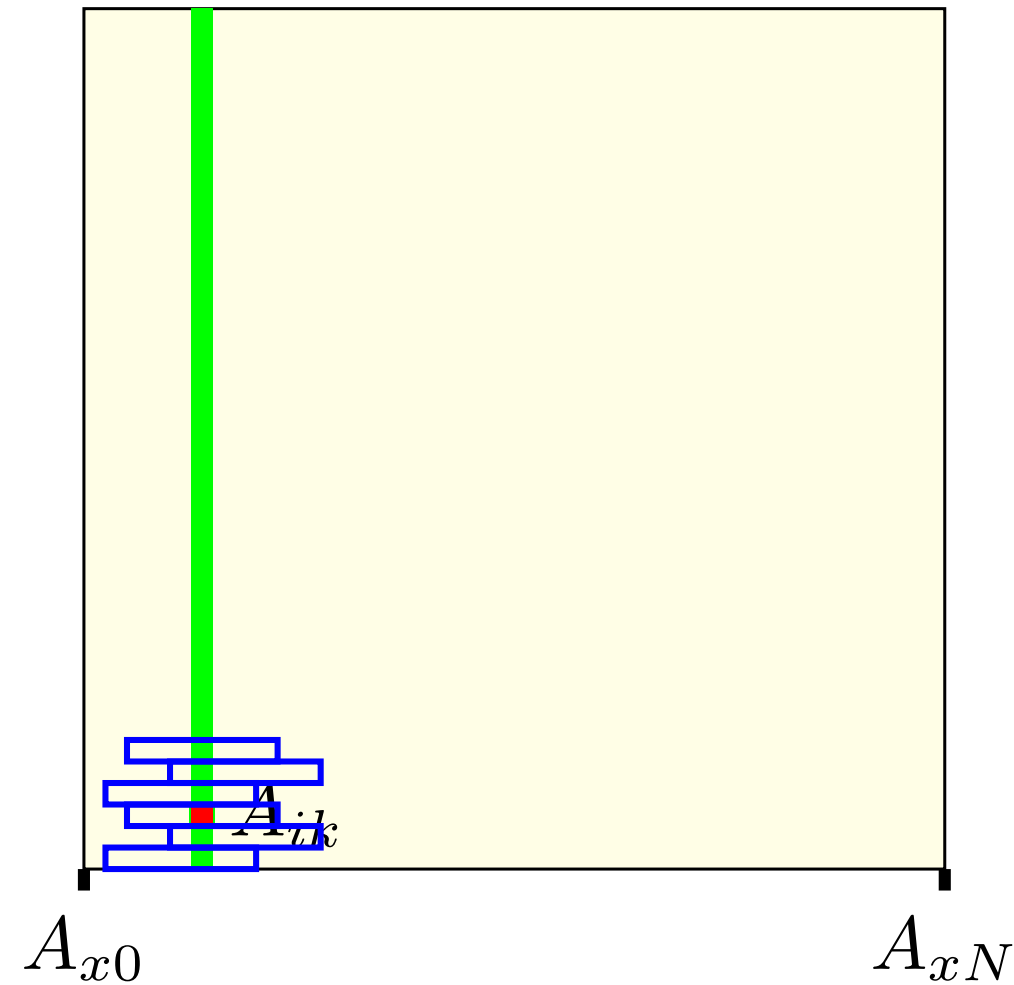
 for all i :

 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 :

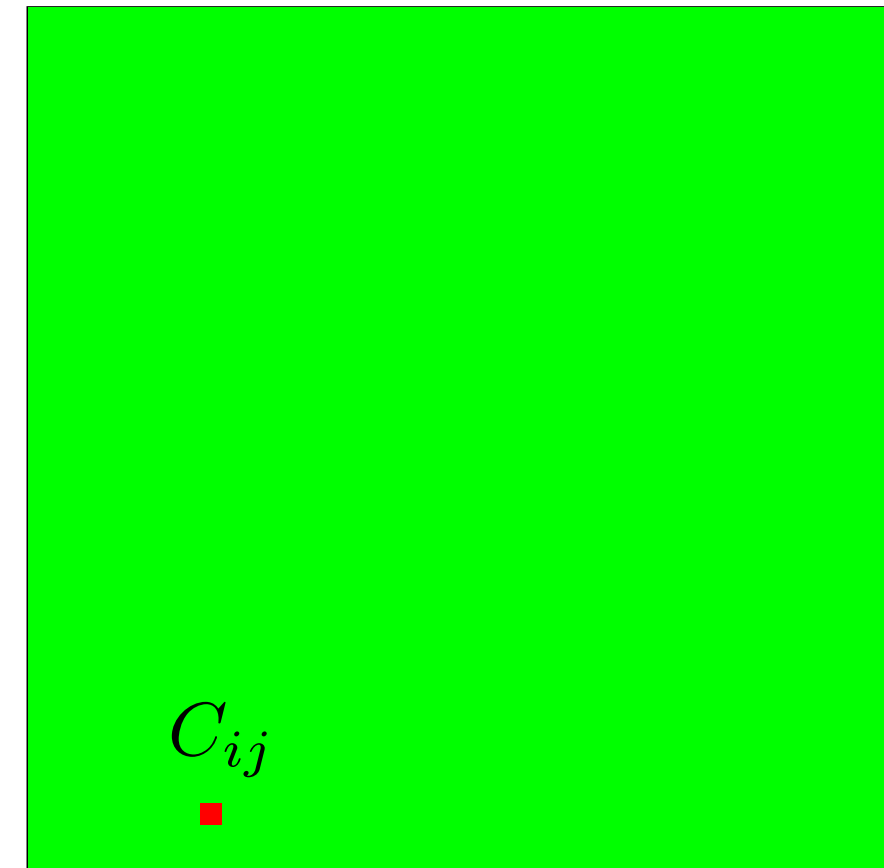
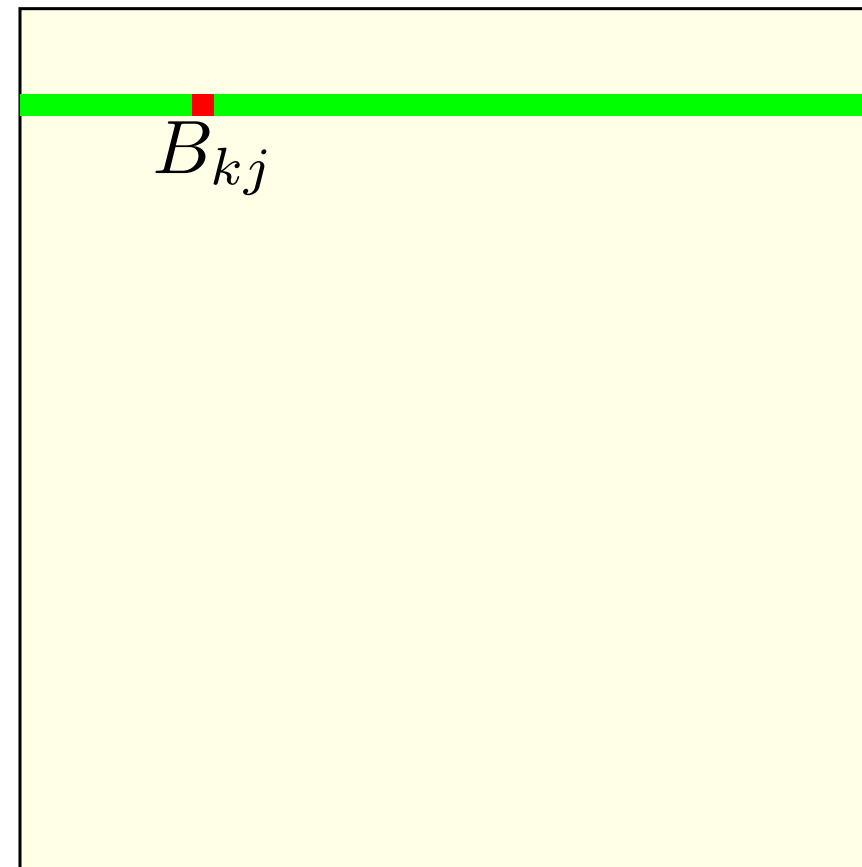
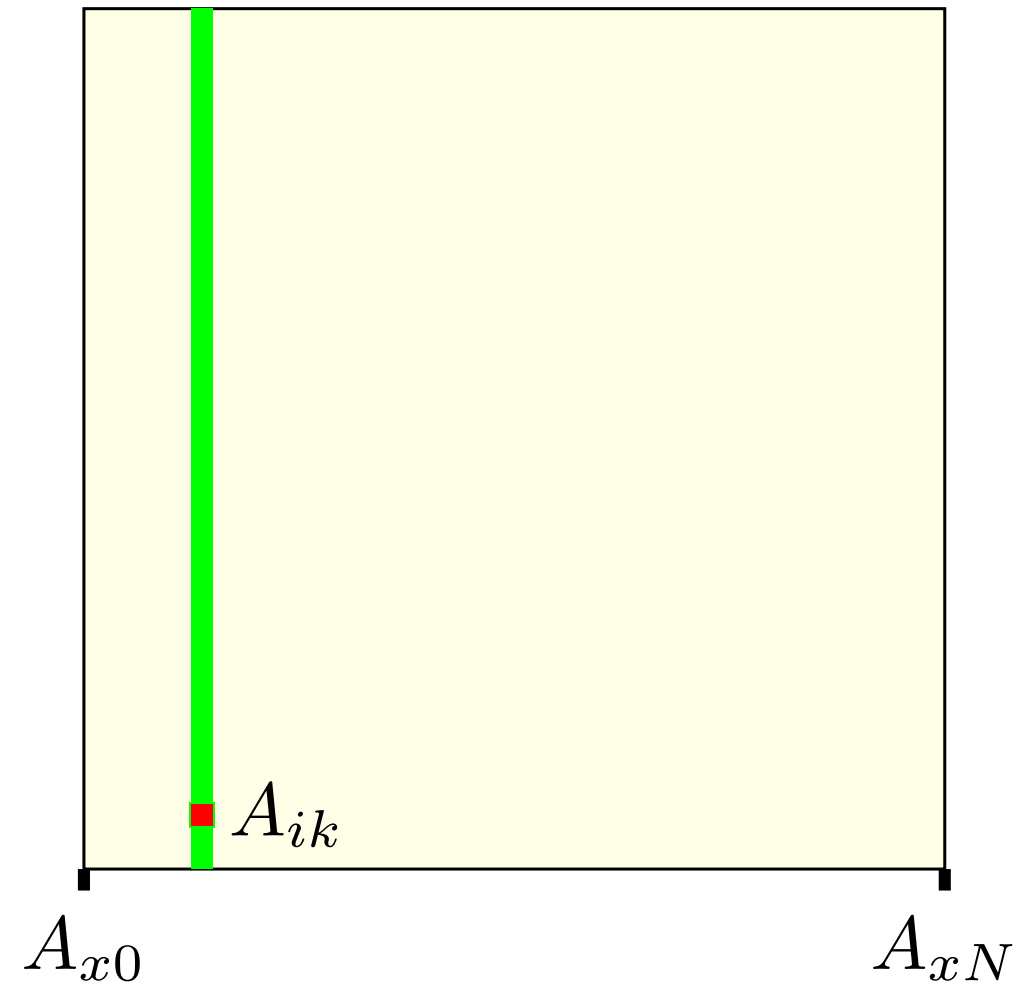
for all i :

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)

array usage: kij order



for all k :

for all i :

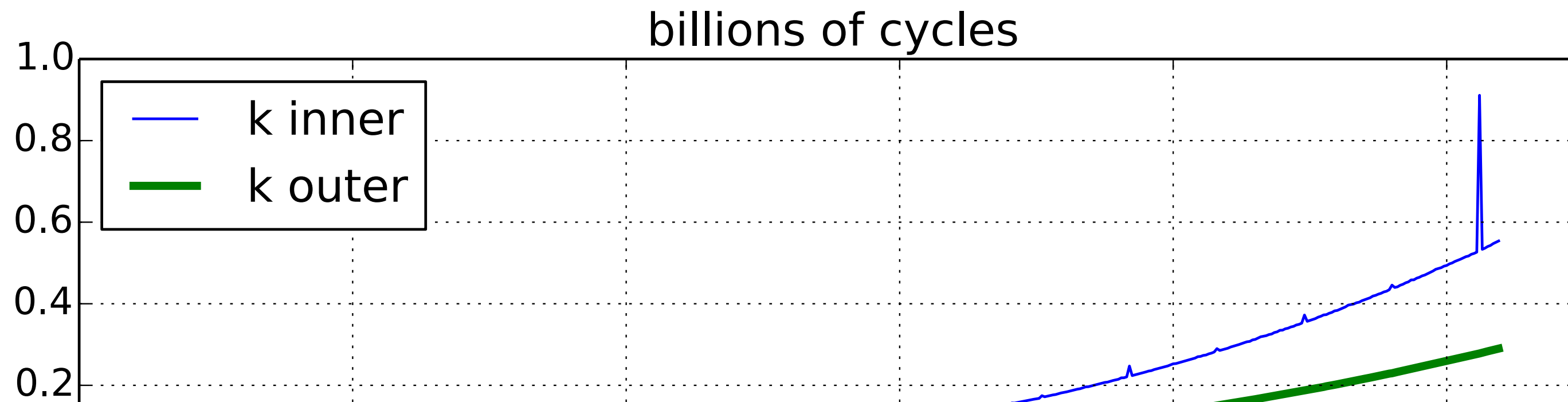
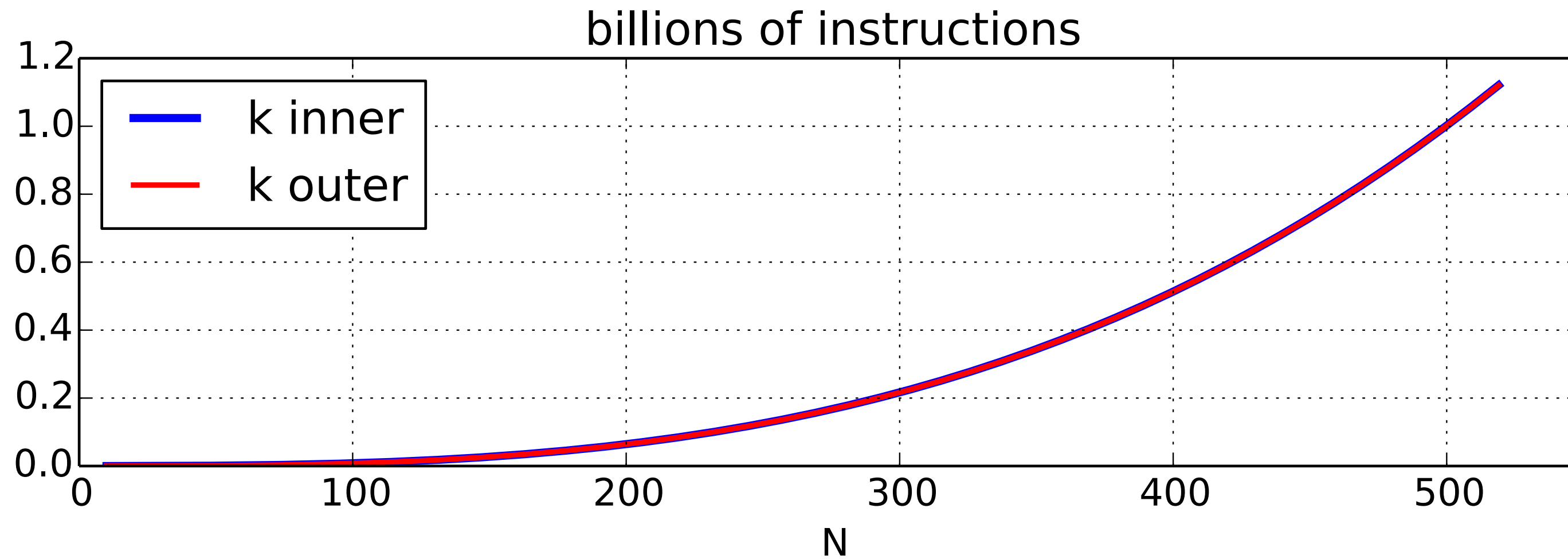
for all j :

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

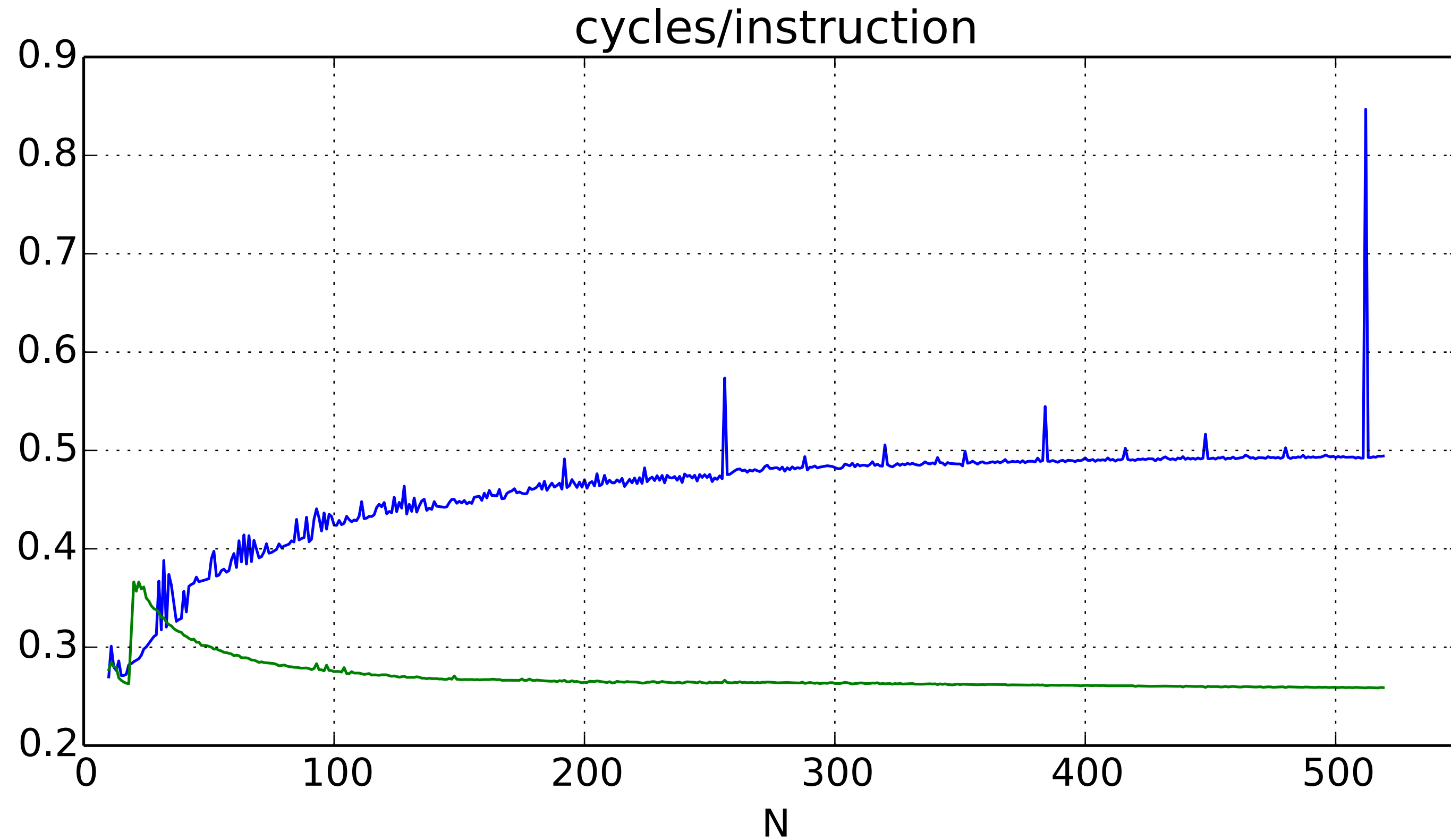
looking only at two innermost loops together:
poor spatial locality in A
good spatial locality in B
good spatial locality in C

againframe(matrixMultTwoVersions)

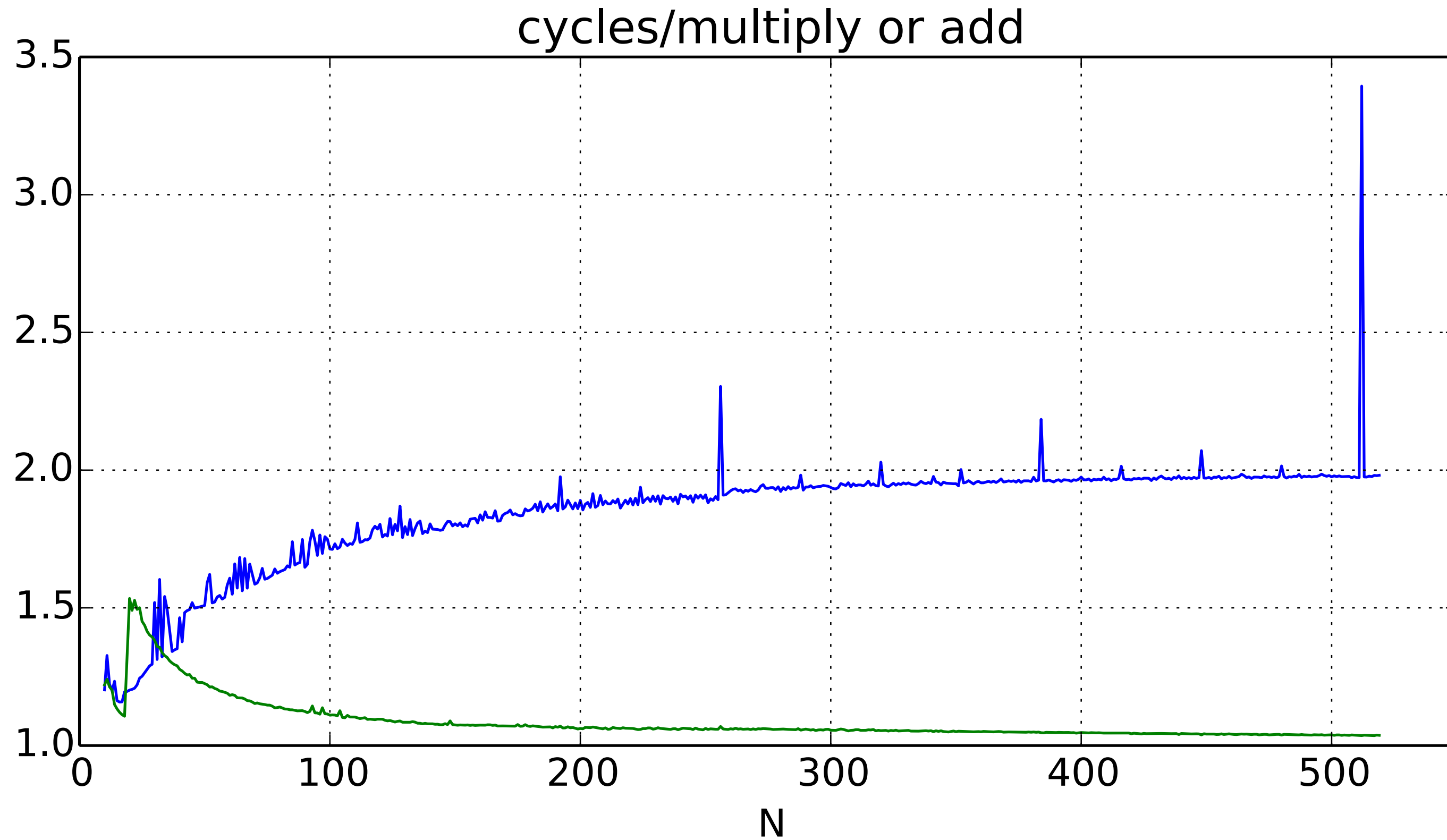
performance (with A=B)



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

textblocksize

misses per k-loop

total misses:

N^3

div

textblocksize

for B: about N misses per k-loop

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

textblocksize

miss per j-loop

total misses:

N^3

div

textblocksize

for C: about

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

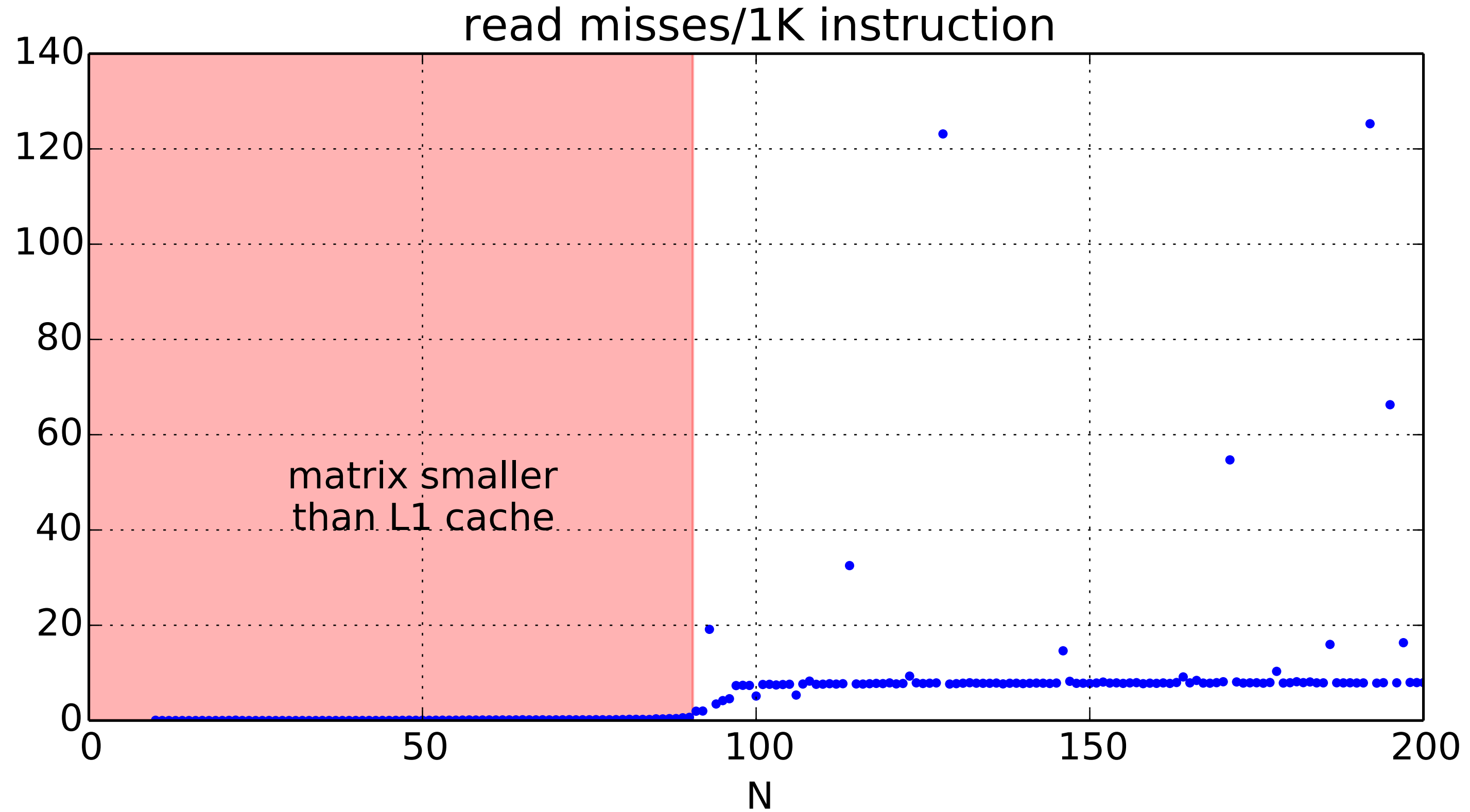
assuming: cache not close to big enough to hold 1K elements

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

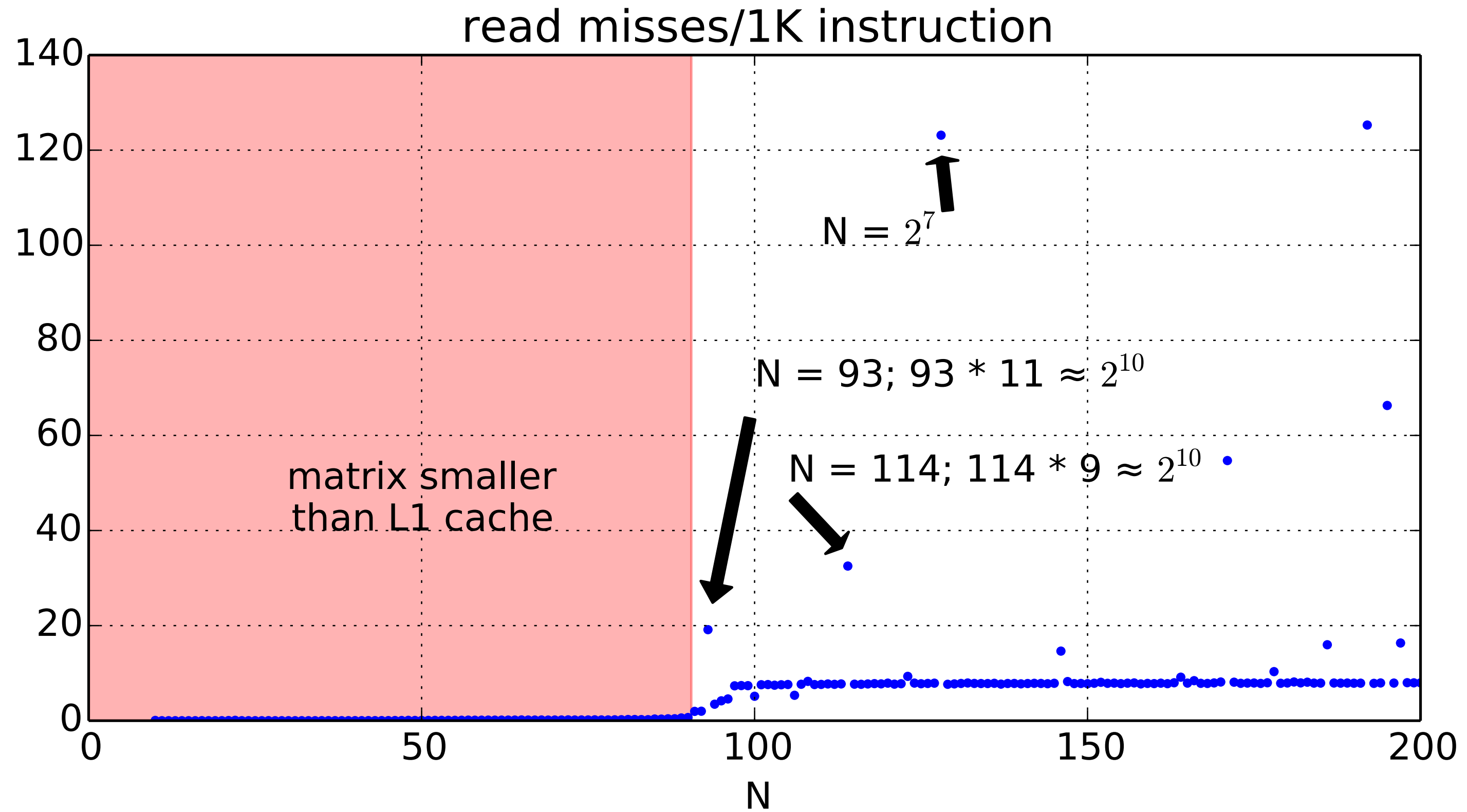
L1 misses (with A=B)



L1 miss detail (1)



L1 miss detail (2)



addresses

$B[k \cdot 114 + j]$	is at	10	0000	0000	0100
$B[k \cdot 114 + j + 1]$	is at	10	0000	0000	1000
$B[(k+1) \cdot 114 + j]$	is at	10	0011	1001	0100
$B[(k+2) \cdot 114 + j]$	is at	10	0101	0101	1100
...					
$B[(k+9) \cdot 114 + j]$	is at	11	0000	0000	1100

test system L1 cache: 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)

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 k = 0; k < N; k += 1)
  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 (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 k = 0; k < N; k += 1)
  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 (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)]{.fragment fragment-index=2 .custom .myem-only}]{.fragment fragment-index=3 .custom .myem-only}
      C\[i*N+j] += A\[i*N+k] * B\[k*N+j];
```

now *reorder* split loop – same calculations

now handle $B_{i,j}$ for $k + 1$ right after $B_{i,j}$ for k

(previously: $B_{i, j + 1}$ for k right after $B_{i,j}$ 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)]{.fragment fragment-index=2 .custom .myem-only}]{.fragment fragment-
index=3 .custom .myem-only}
      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) {
        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) {  
        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) {  
        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) {
        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];
        }
    }
}
```

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)

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

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

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

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

...

...

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

$\frac{N^2}{2}$

$\cdot N$

misses total

counting misses for A (2)

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

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

...

...

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

$\frac{N^2}{2}$

$\cdot N$

misses total

counting misses for A (2)

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

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

...

...

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

$\frac{N^2}{2}$

$\cdot N$

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:

$B[0*N+0], B[1*N+0], \dots B[0*N+(N-1)], B[1*N+(N-1)]$

$B[2*N+0], B[3*N+0], \dots B[2*N+(N-1)], B[3*N+(N-1)]$

$B[4*N+0], B[5*N+0], \dots B[4*N+(N-1)], B[5*N+(N-1)]$

...

$B[0*N+0], B[1*N+0], \dots B[0*N+(N-1)], B[1*N+(N-1)]$

...

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:

$B[0*N+0], B[1*N+0], \dots B[0*N+(N-1)], B[1*N+(N-1)]$

$B[2*N+0], B[3*N+0], \dots B[2*N+(N-1)], B[3*N+(N-1)]$

$B[4*N+0], B[5*N+0], \dots B[4*N+(N-1)], B[5*N+(N-1)]$

...

$B[0*N+0], B[1*N+0], \dots B[0*N+(N-1)], B[1*N+(N-1)]$

...

counting misses for B (2)

access pattern for B:

$B[0*N+0], B[1*N+0], \dots B[0*N+(N-1)], B[1*N+(N-1)]$

$B[2*N+0], B[3*N+0], \dots B[2*N+(N-1)], B[3*N+(N-1)]$

$B[4*N+0], B[5*N+0], \dots B[4*N+(N-1)], B[5*N+(N-1)]$

...

$B[0*N+0], B[1*N+0], \dots B[0*N+(N-1)], B[1*N+(N-1)]$

...

likely cache misses: any access, each time

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

about

$\frac{N^2}{2}$

$\cdot N$

\cdot

$\frac{2N}{\text{cache block size}} = N^3$

\cdot

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

textblocksize

misses from B per j-loop

-->

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];
    }
```

$\frac{N^2}{2}$

$\cdot N$

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

$\frac{N^2}{\text{blocksize}}$

misses from B per j-loop

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];
    }
```

$\frac{N^2}{2}$

$\cdot N$

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

$2N$

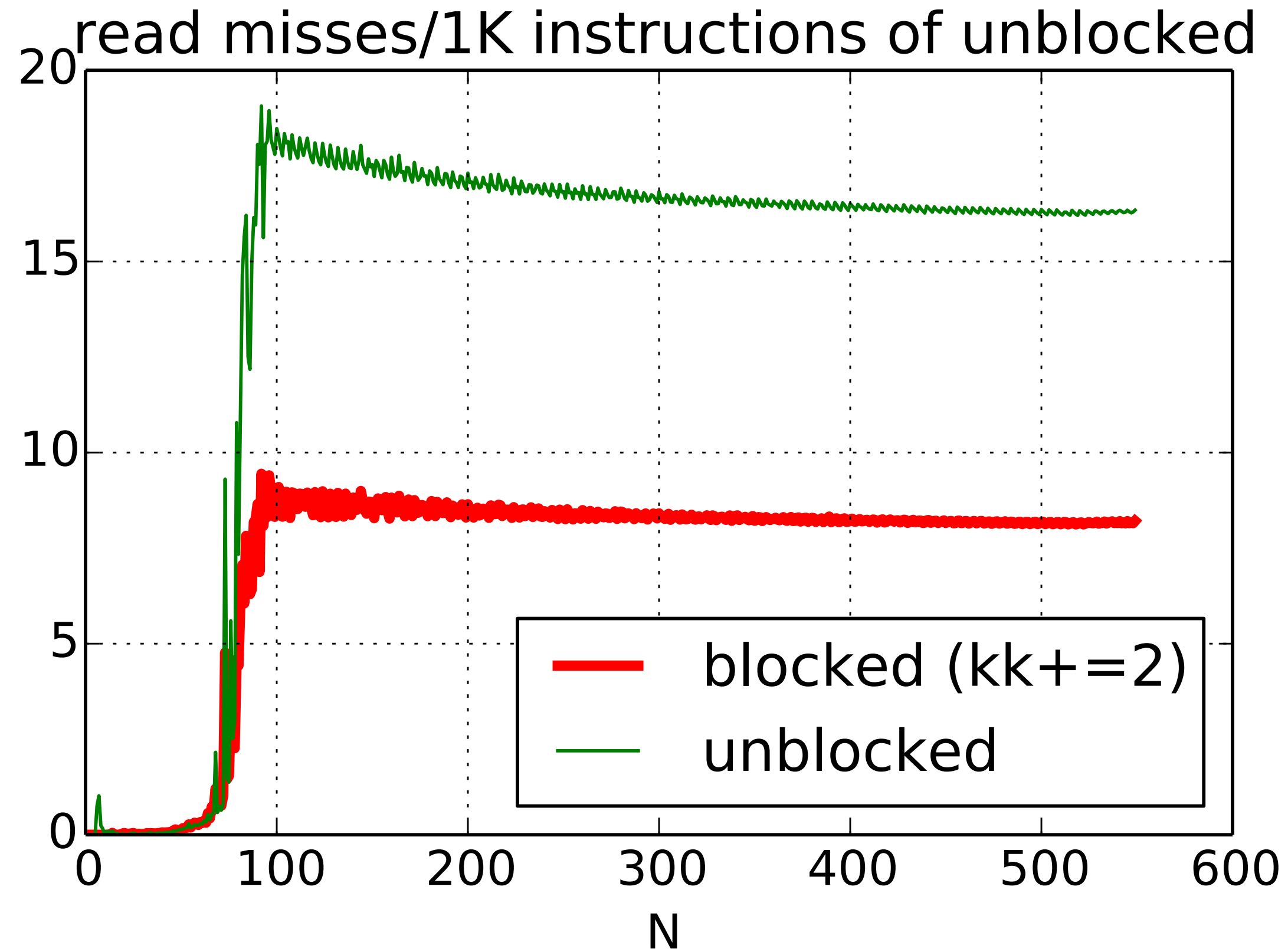
div

textblocksize

misses from B per j-loop

-->

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

simple blocking — locality

```
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": */
      @\normalsize$C_{i+0,j}$@ += @\normalsize$A_{i+0,k+0}$@ * @\normalsize\myemph{$B_{k+0,j}$}$@
      @\normalsize$C_{i+0,j}$@ += @\normalsize$A_{i+0,k+1}$@ * @\normalsize$B_{k+1,j}$@
      @\normalsize$C_{i+1,j}$@ += @\normalsize$A_{i+1,k+0}$@ * @\normalsize\myemph{$B_{k+0,j}$}$@
      @\normalsize$C_{i+1,j}$@ += @\normalsize$A_{i+1,k+1}$@ * @\normalsize$B_{k+1,j}$@
    }
  }
}
```

now: more temporal locality in B

previously: access B_{kj} , then don't use it again for a *long* time

simple blocking — counting misses for A

```
for (int k = 0; k < N; k += 2)
  for (int i = 0; i < N; i += 2)
    for (int j = 0; j < N; ++j) {
      @\normalsize$C_{i+0,j}$@ += @\normalsize$A_{i+0,k+0}$@ * @\normalsize$B_{k+0,j}$@
      @\normalsize$C_{i+0,j}$@ += @\normalsize$A_{i+0,k+1}$@ * @\normalsize$B_{k+1,j}$@
      @\normalsize$C_{i+1,j}$@ += @\normalsize$A_{i+1,k+0}$@ * @\normalsize$B_{k+0,j}$@
      @\normalsize$C_{i+1,j}$@ += @\normalsize$A_{i+1,k+1}$@ * @\normalsize$B_{k+1,j}$@
    }
}
```

$\frac{N^2}{2}$

\cdot

$\frac{N^2}{2}$

iterations of j loop

likely 2 misses per loop with A (2 cache blocks)

total misses:

$\frac{N^2}{2}$

(same as only blocking in K)

simple blocking — counting misses for B

```
for (int k = 0; k < N; k += 2)
  for (int i = 0; i < N; i += 2)
    for (int j = 0; j < N; ++j) {
      @\normalsize$C_{i+0,j}$@ += @\normalsize$A_{i+0,k+0}$@ * @\normalsize\myemph{$B_{k+0,j}$}@
      @\normalsize$C_{i+0,j}$@ += @\normalsize$A_{i+0,k+1}$@ * @\normalsize$B_{k+1,j}$@
      @\normalsize$C_{i+1,j}$@ += @\normalsize$A_{i+1,k+0}$@ * @\normalsize\myemph{$B_{k+0,j}$}@
      @\normalsize$C_{i+1,j}$@ += @\normalsize$A_{i+1,k+1}$@ * @\normalsize$B_{k+1,j}$@
    }
}
```

$\frac{N^2}{2}$

\cdot

$\frac{N^2}{2}$

iterations of j loop

likely

2

\div

textblocksize

misses per iteration with B

simple blocking — counting misses for C

```
for (int k = 0; k < N; k += 2)
  for (int i = 0; i < N; i += 2)
    for (int j = 0; j < N; ++j) {
      @\normalsize$C_{i+0,j}$@ += @\normalsize$A_{i+0,k+0}$@ * @\normalsize{$B_{k+0,j}$}$@
      @\normalsize$C_{i+0,j}$@ += @\normalsize$A_{i+0,k+1}$@ * @\normalsize$B_{k+1,j}$@
      @\normalsize$C_{i+1,j}$@ += @\normalsize$A_{i+1,k+0}$@ * @\normalsize{$B_{k+0,j}$}$@
      @\normalsize$C_{i+1,j}$@ += @\normalsize$A_{i+1,k+1}$@ * @\normalsize$B_{k+1,j}$@
    }
```

$\frac{N^2}{2}$

\cdot

$\frac{N^2}{2}$

iterations of j loop

likely

$\frac{2}{\text{textblocksize}}$

misses per iteration with C

total misses:

$\frac{N^2}{2} \cdot \frac{2}{\text{textblocksize}}$

simple blocking — counting misses (total)

```
for (int k = 0; k < N; k += 2)
  for (int i = 0; i < N; i += 2)
    for (int j = 0; j < N; ++j) {
      @\normalsize$C_{i+0,j}$@ += @\normalsize$A_{i+0,k+0}$@ * @\normalsize{$B_{k+0,j}$}$@
      @\normalsize$C_{i+0,j}$@ += @\normalsize$A_{i+0,k+1}$@ * @\normalsize$B_{k+1,j}$@
      @\normalsize$C_{i+1,j}$@ += @\normalsize$A_{i+1,k+0}$@ * @\normalsize{$B_{k+0,j}$}$@
      @\normalsize$C_{i+1,j}$@ += @\normalsize$A_{i+1,k+1}$@ * @\normalsize$B_{k+1,j}$@
    }
```

before:

A:

$$\frac{N^2}{2}$$

; B:

$$\frac{N^3}{1}$$

cdot

textblocksize

; C

$$\frac{N^3}{1}$$

cdot

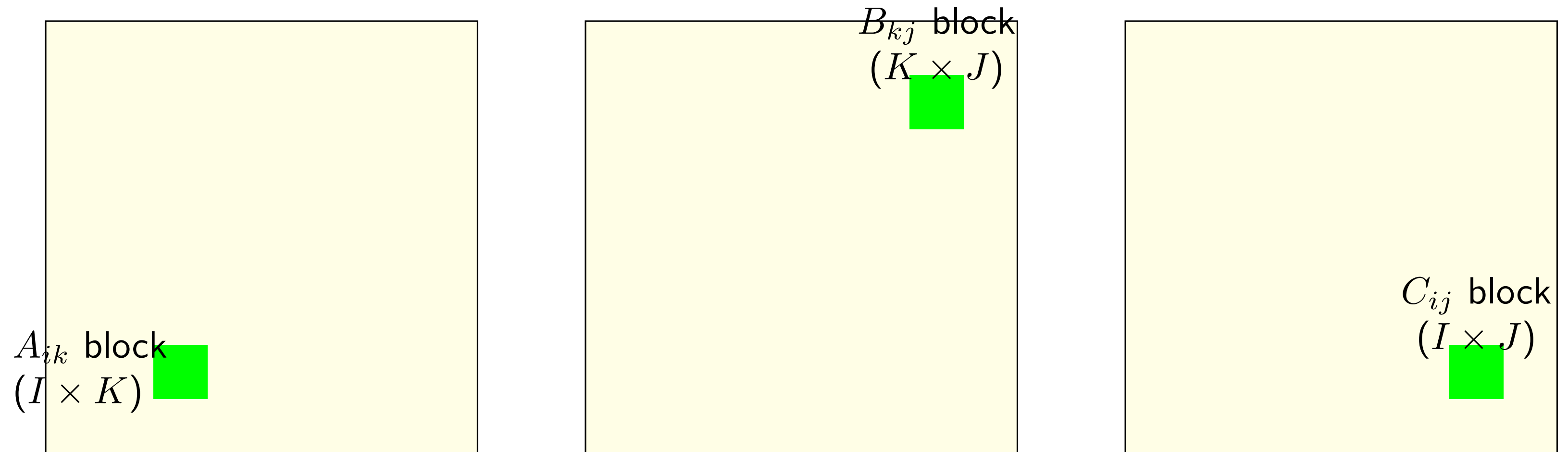
generalizing: divide and conquer

```
partial_matrixmultiply(float *A, float *B, float *C
                        int startI, int endI, ...) {
    for (int i = startI; i < endI; ++i) {
        for (int j = startJ; j < endJ; ++j) {
            for (int k = startK; k < endK; ++k) {
                ...
            }
        }
    }
}

matrix_multiply(float *A, float *B, float *C, int N) {
    for (int ii = 0; ii < N; ii += BLOCK_I)
        for (int jj = 0; jj < N; jj += BLOCK_J)
            for (int kk = 0; kk < N; kk += BLOCK_K)
                ...
                /* do everything for segment of A, B, C
                   that fits in cache! */
                partial_matmul(A, B, C,
                               ii, ii + BLOCK_I, jj, jj + BLOCK_J,
                               kk, kk + BLOCK_K)
}
}
```

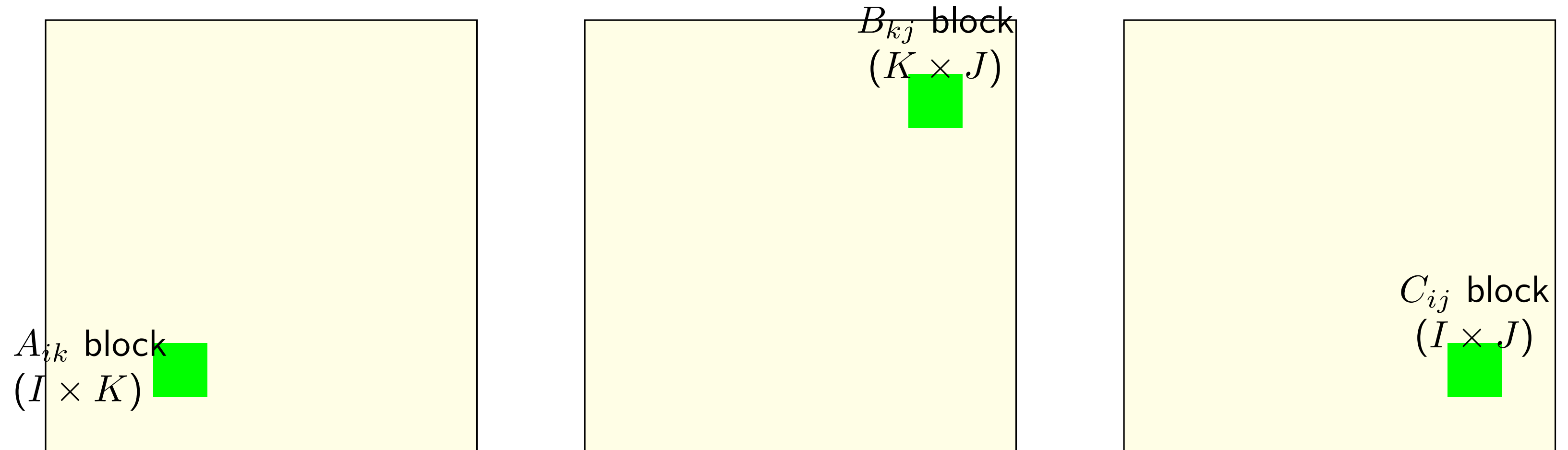
array usage: matrix block

array usage: matrix block



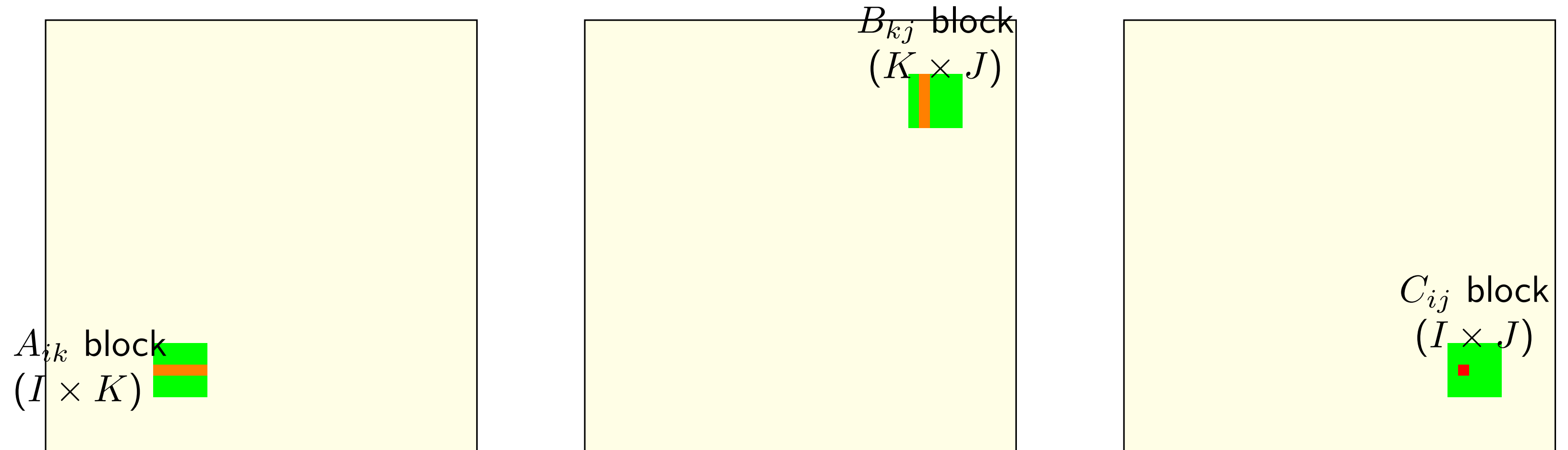
inner loops work on “matrix block” of A, B, C
rather than rows of some, little blocks of others
blocks fit into cache (b/c we choose I, K, J)
where previous rows might not

array usage: matrix block



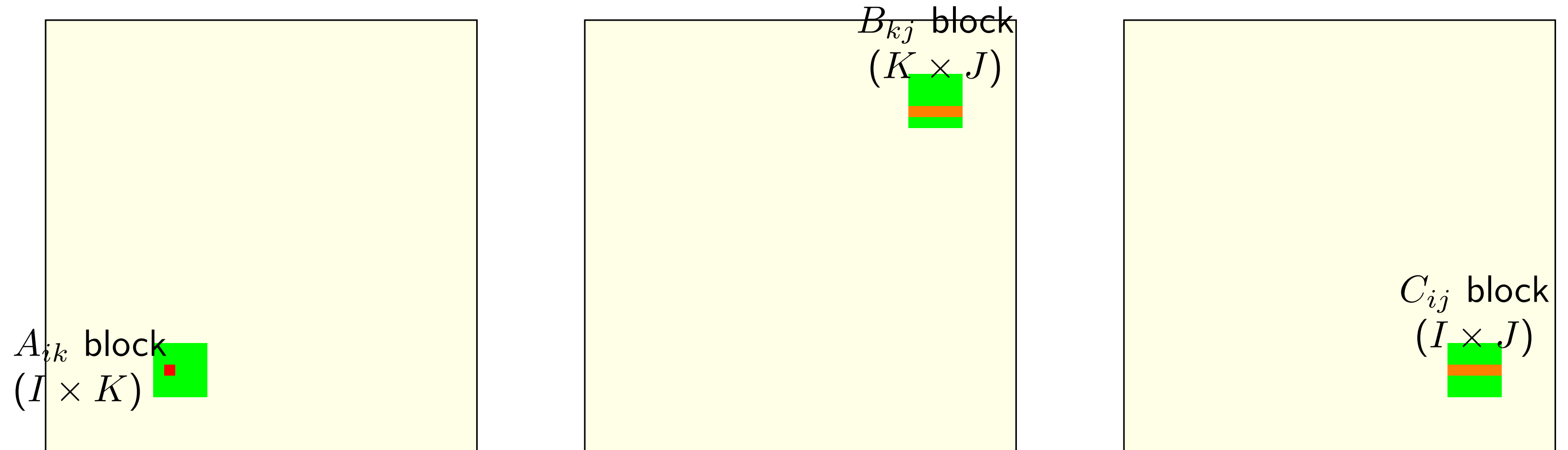
now (versus loop ordering example)
some spatial locality in A, B, and C
some temporal locality in A, B, and C

array usage: matrix block



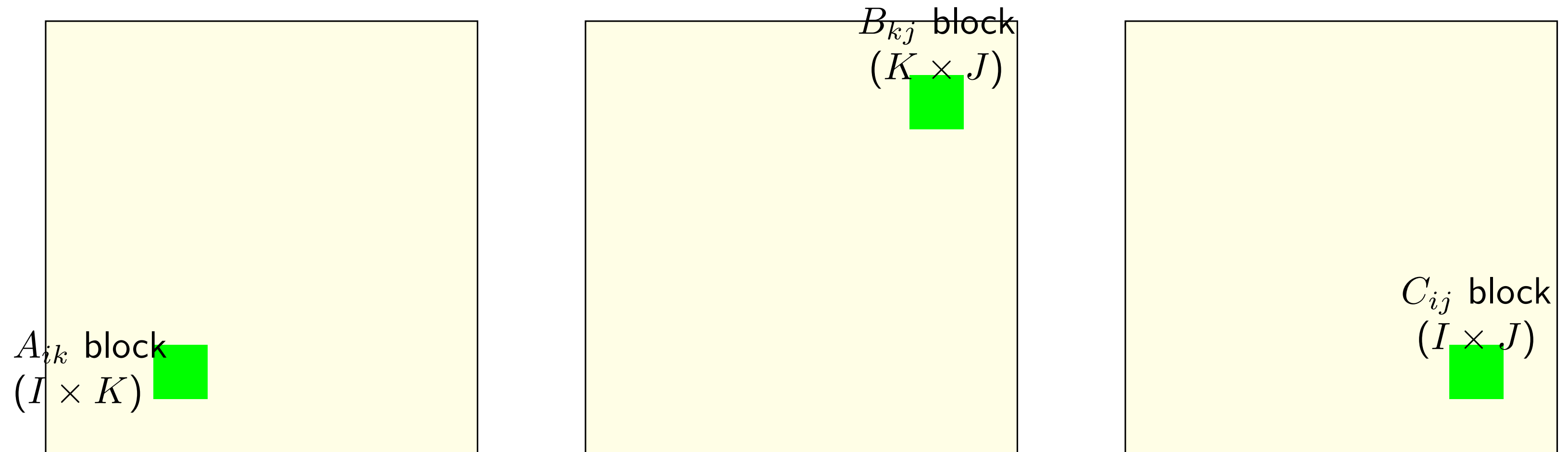
C_{ij} calculation uses strips from A , B
 K calculations for one cache miss
good temporal locality!

array usage: matrix block



A_{ik} used with entire strip of B J calculations for one cache miss
good temporal locality!

array usage: matrix block



(approx.) KIJ fully cached calculations
for $KI + IJ + KJ$ values need to be loaded per “matrix block”
(assuming everything stays in cache)

cache blocking efficiency

for each of N^3 / IJK matrix blocks:

load

I

times K

elements of $A_i k$:

approx IK

div

textblocksize

misses per matrix block

approx $N^3 / (J$

cdot

textblocksize)

misses total

load

→

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

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)  
      @$C_{ij}$, $B_{kj}$ loaded each iteration (if $N$ big):@  
      B[i*N+j] += A[i*N+k] * A[k*N+j];  
  }  
}
```

values from A_{ik} used N times per load

values from B_{kj} used 1 times per load

but good spatial locality, so cache block of B_{kj} together

values from C_{ij} used 1 times per load

but good spatial locality, so cache block of C_{ij} together

exercise: miss estimating (3)

```
for (int kk = 0; kk < 1000; kk += 10)
  for (int jj = 0; jj < 1000; jj += 10)
    for (int i = 0; i < 1000; i += 1)
      for (int j = jj; j < jj+10; j += 1)
        for (int k = kk; k < kk + 10; k += 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, but big enough to hold 500 or so

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

hint 1: part of A, B loaded in two inner-most loops only needs to be loaded once

hint 2: part of A can be reused between iterations of i loop

loop ordering compromises

loop ordering forces compromises:

```
for k: for i: for j: c[i,j] += a[i,k] * b[j,k]
```

perfect temporal locality in $a[i,k]$

bad temporal locality for $c[i,j]$, $b[j,k]$

perfect spatial locality in $c[i,j]$

bad spatial locality in $b[j,k]$, $a[i,k]$

cache blocking: work on blocks rather than rows/columns

have some temporal, spatial locality in everything

cache blocking pattern

no perfect loop order? work on rectangular matrix blocks

size amount used in inner loops based on cache size

in practice:

- test performance to determine 'size' of blocks

Backup slides

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

$\frac{N^2}{3}$ total misses (before blocking: N^2)

about

$3N$

div

$\frac{N^2}{3}$

misses from B per i-loop iteration

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

$\frac{N^2}{3}$ total misses (before blocking: N^2)

about

$3N$

div

$\frac{N^2}{3}$

misses from B per i-loop iteration

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

$\frac{N^2}{3}$ total misses (before blocking: N^2)

about

$3N$

div

$\frac{N^2}{3}$

misses from B per i-loop iteration

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

$\frac{N^2}{3}$ total misses (before blocking: N^2)

about

$3N$

div

$\frac{N^2}{3}$

misses from B per i-loop iteration

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

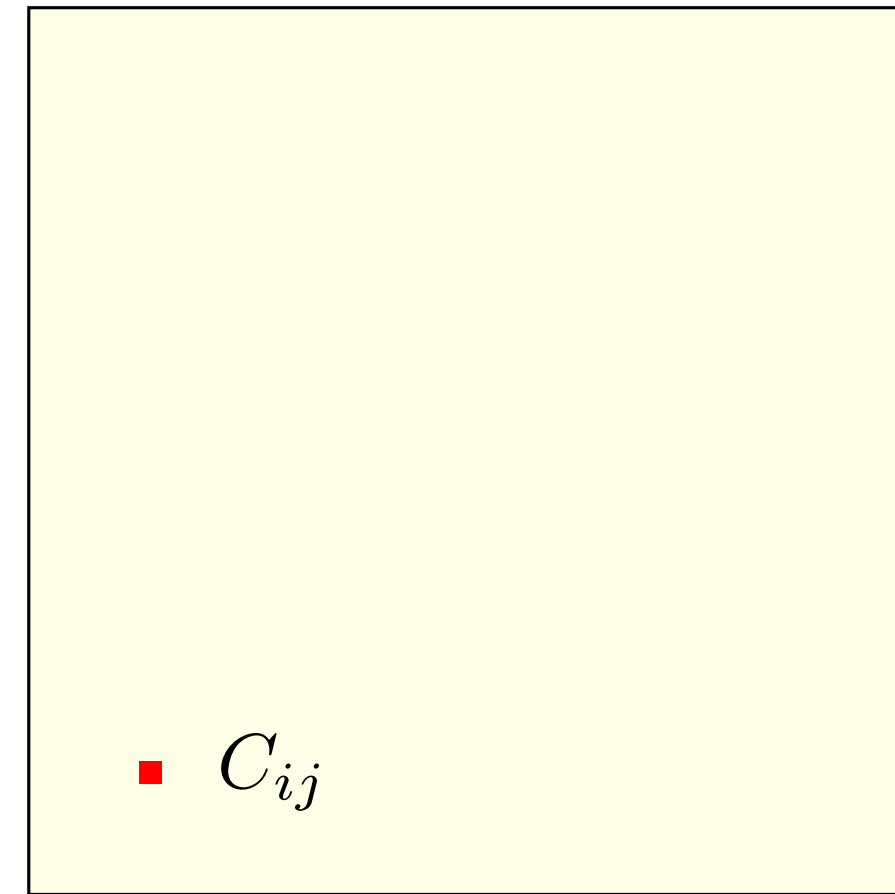
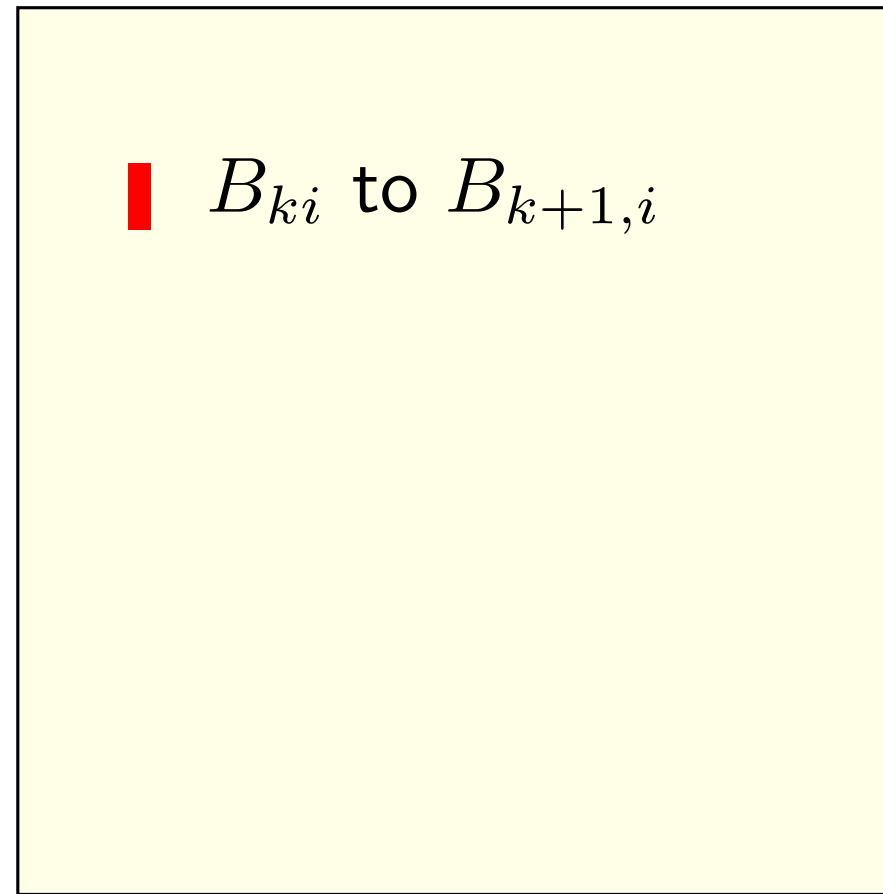
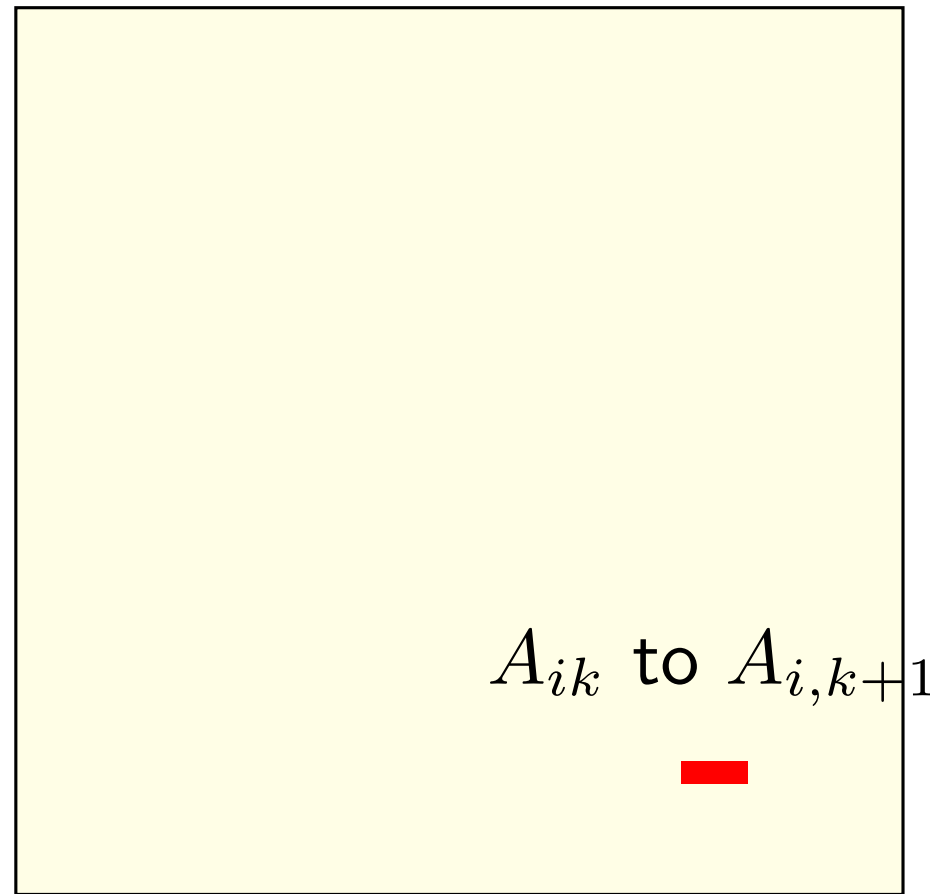
X too large – cache not big enough

assumption: X blocks from B would help with spatial locality

X too large – evicted from cache before next iteration

array usage ($2k$ at a time)

array usage (2 k at a time)



for each kk :

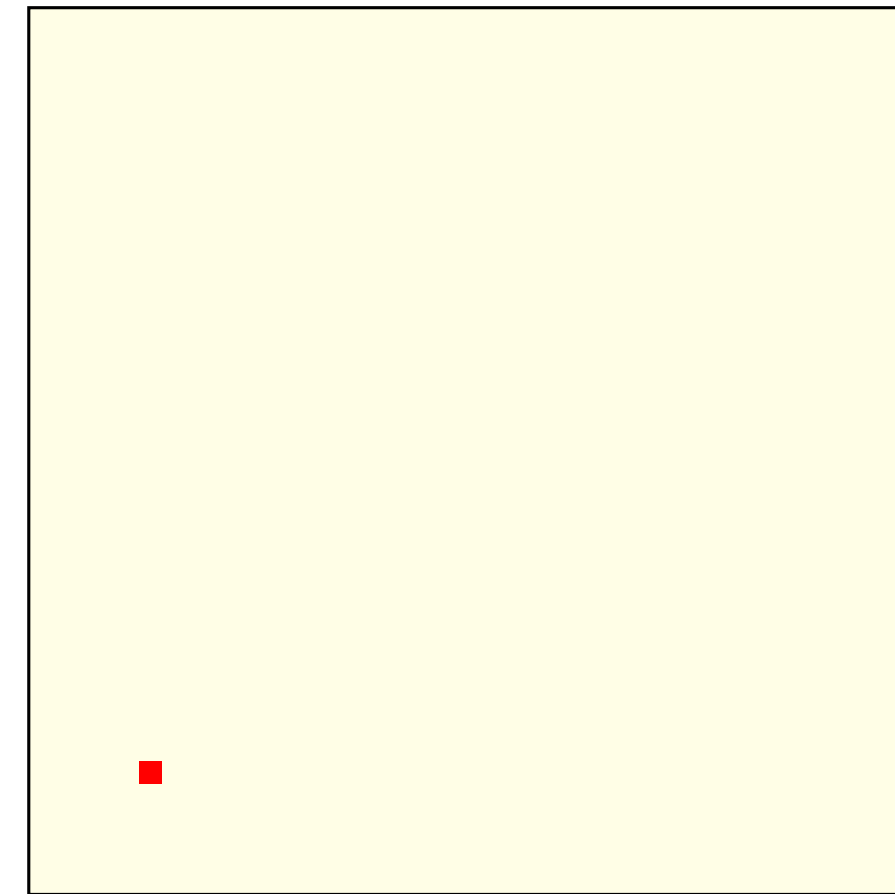
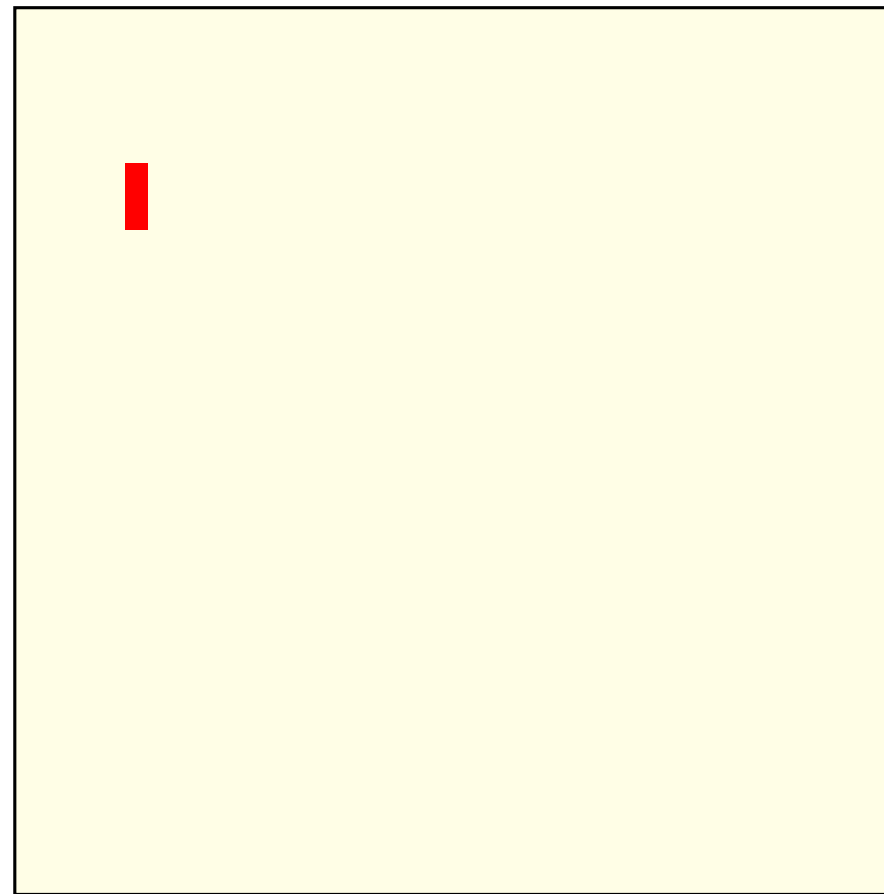
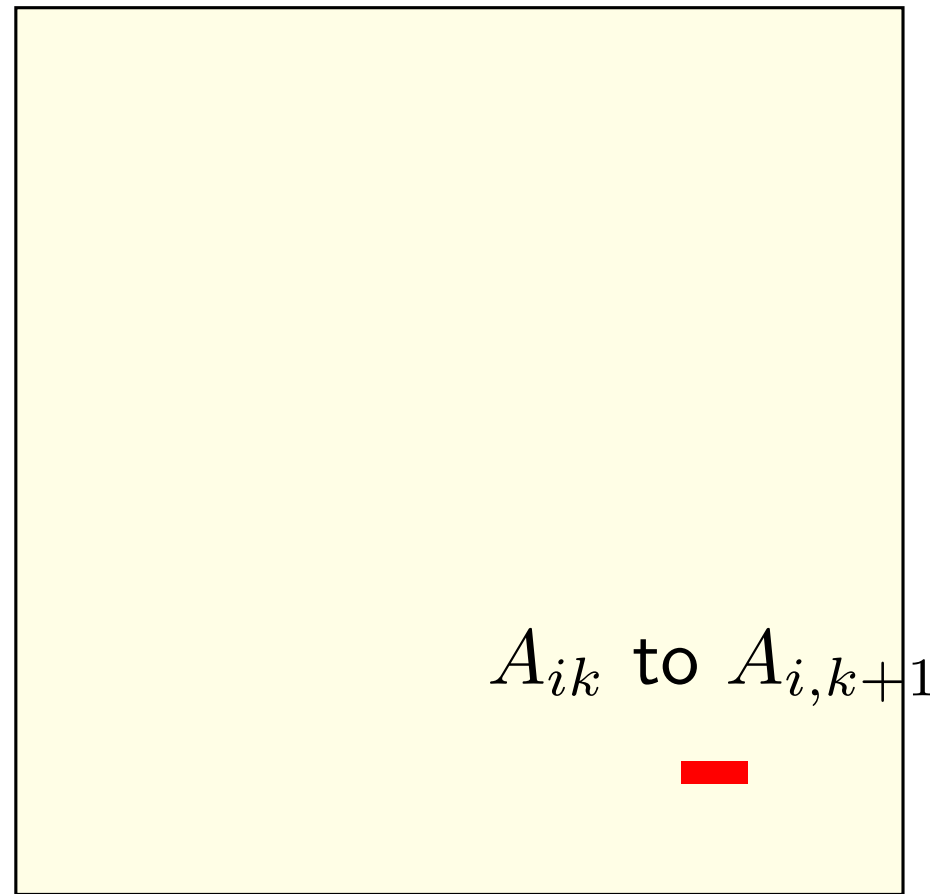
for each i :

for each j :

for $k=kk, kk+1$:

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

array usage (2 k at a time)



for each k :

for each i :

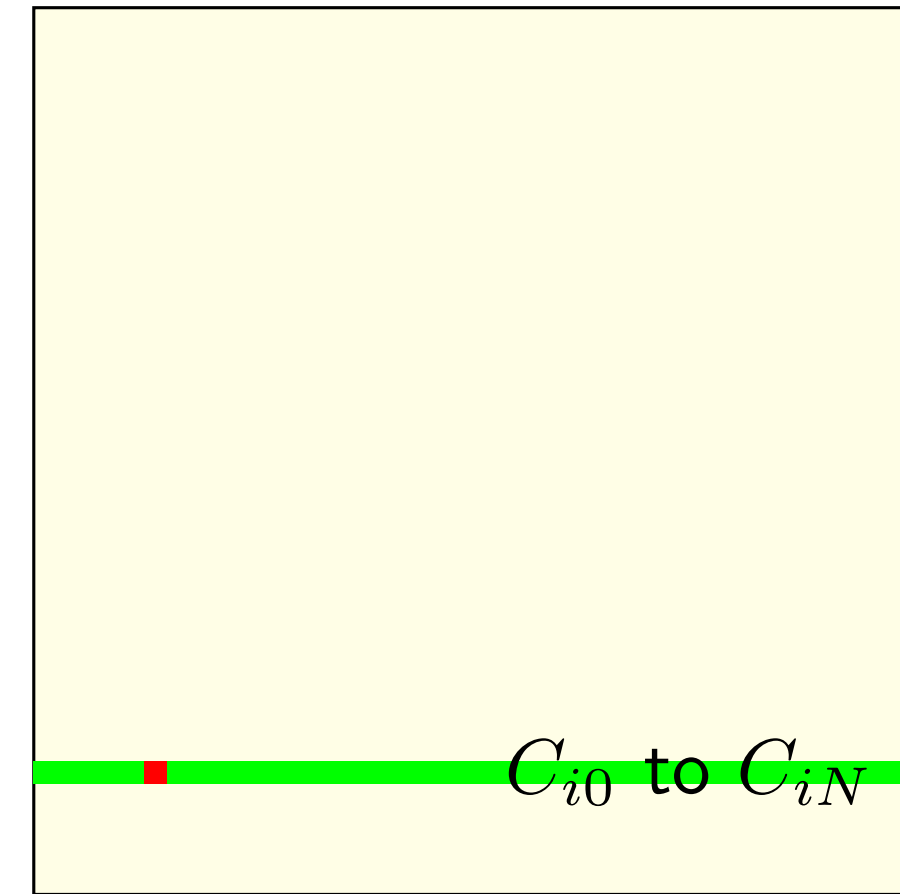
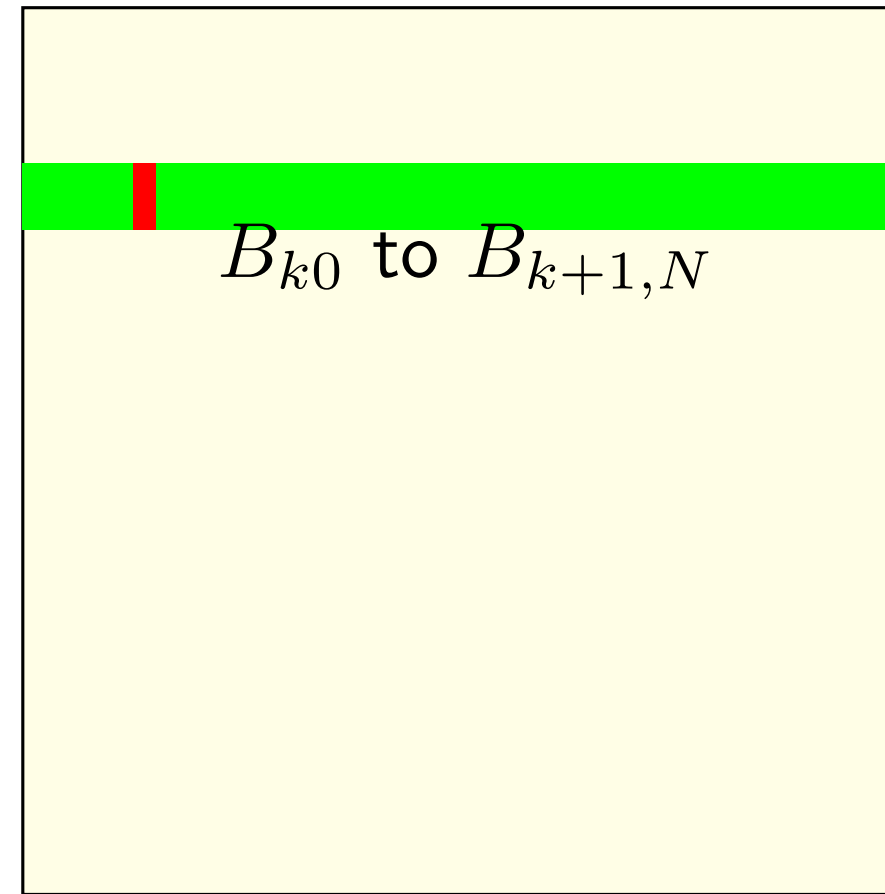
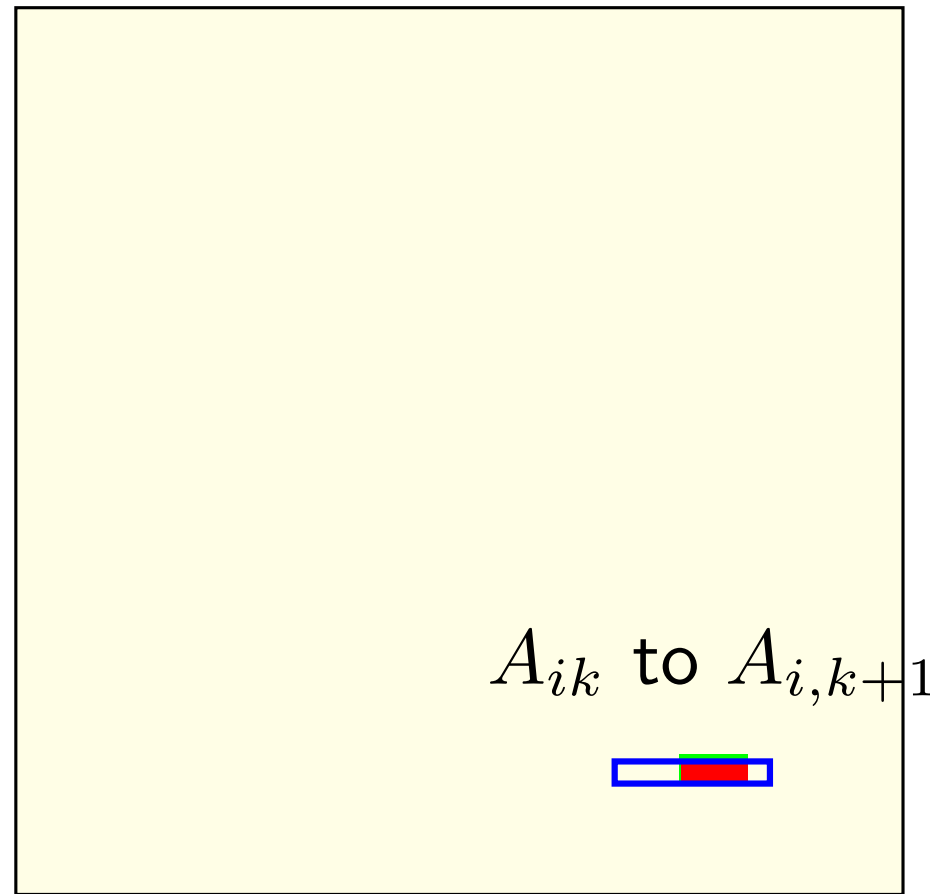
for each j :

for $k=k, k+1$:

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

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 :

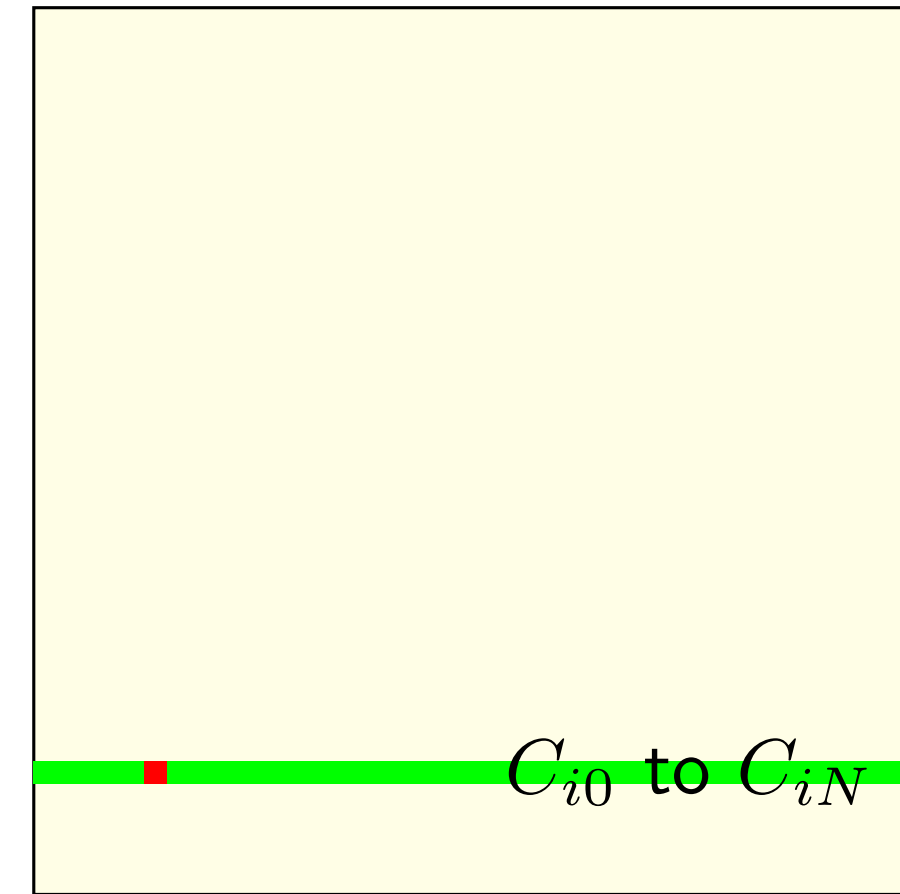
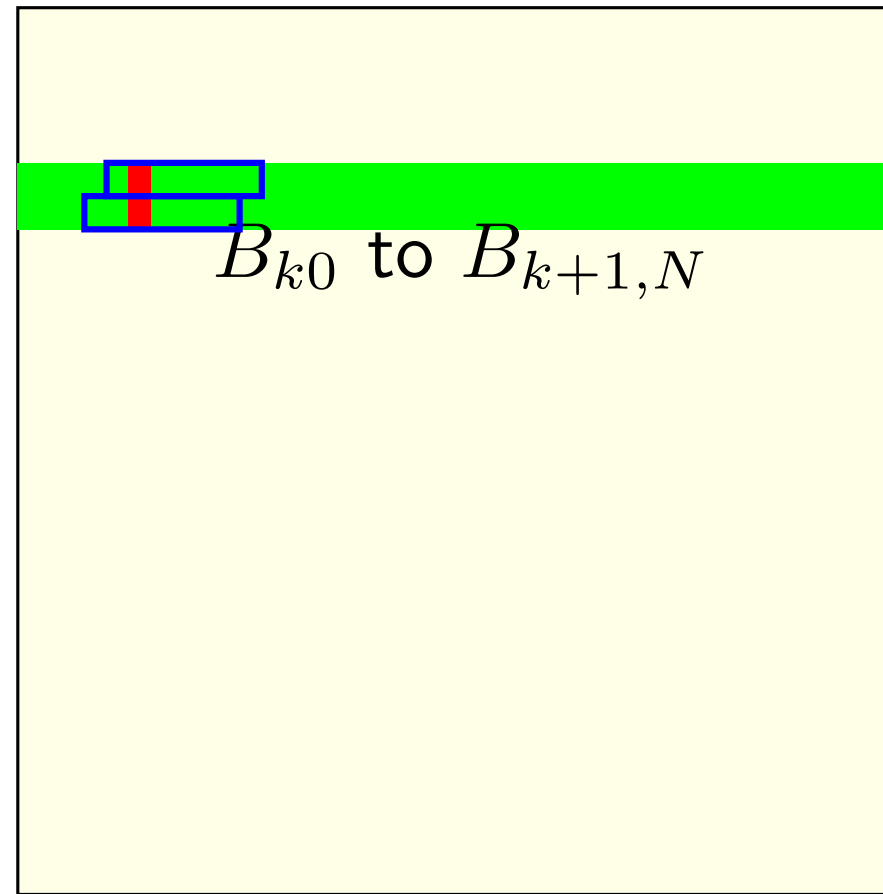
for each j :

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 :

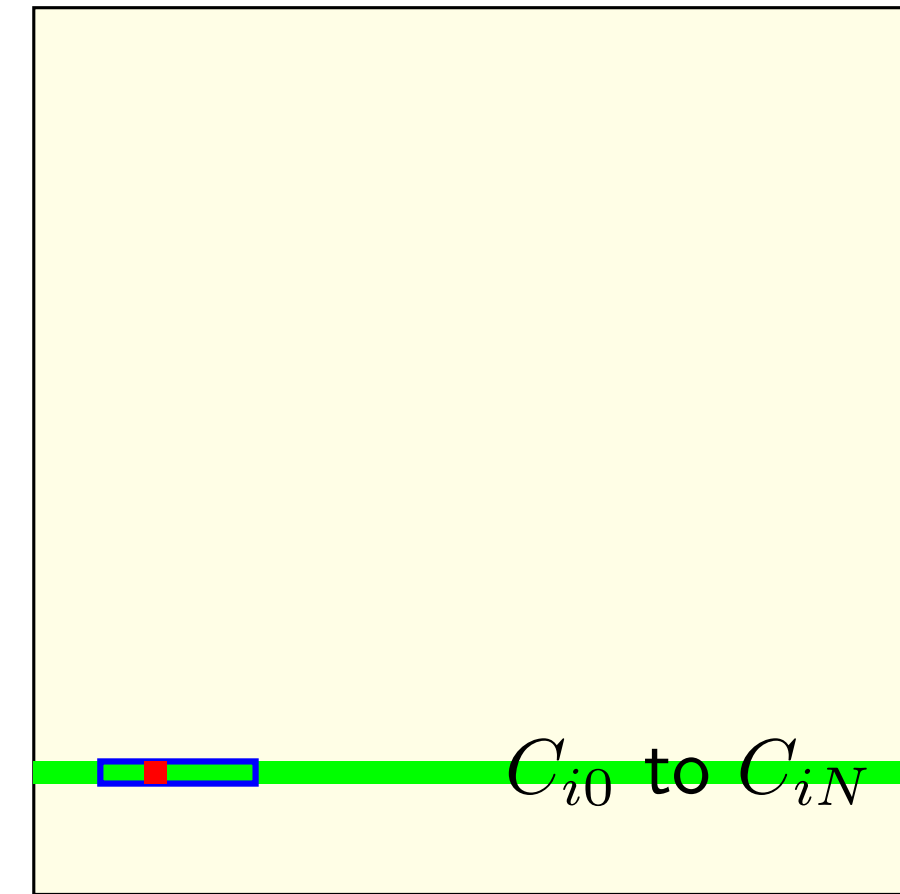
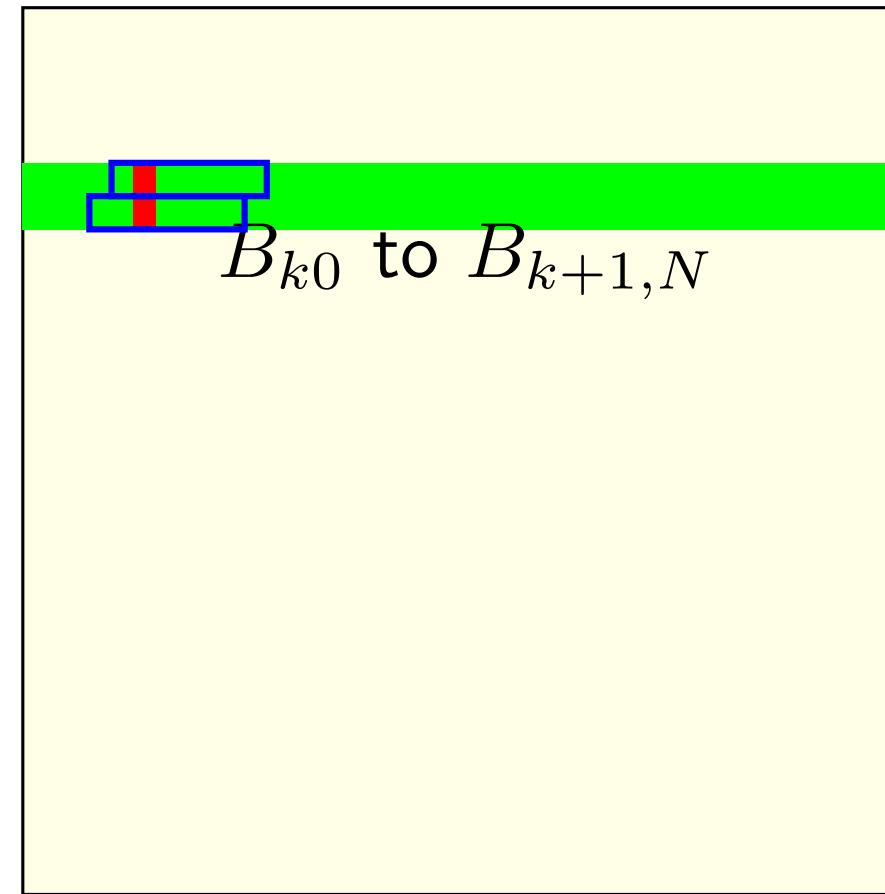
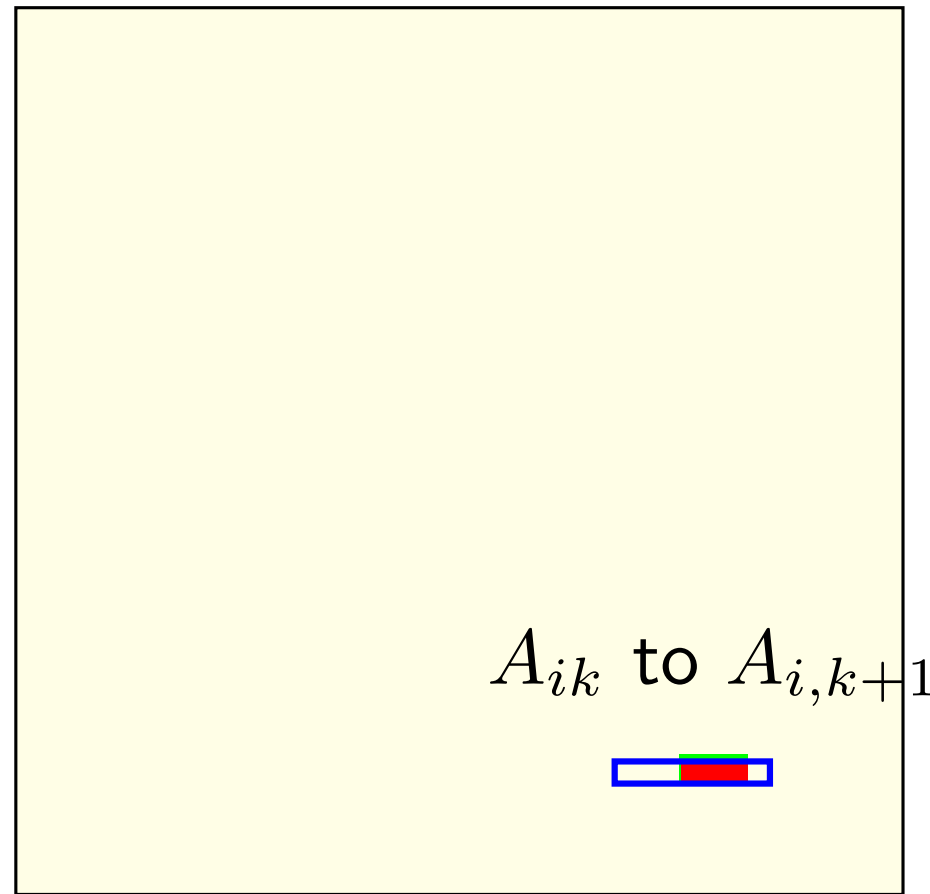
for each j :

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 $k=kk, kk+1$:

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

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