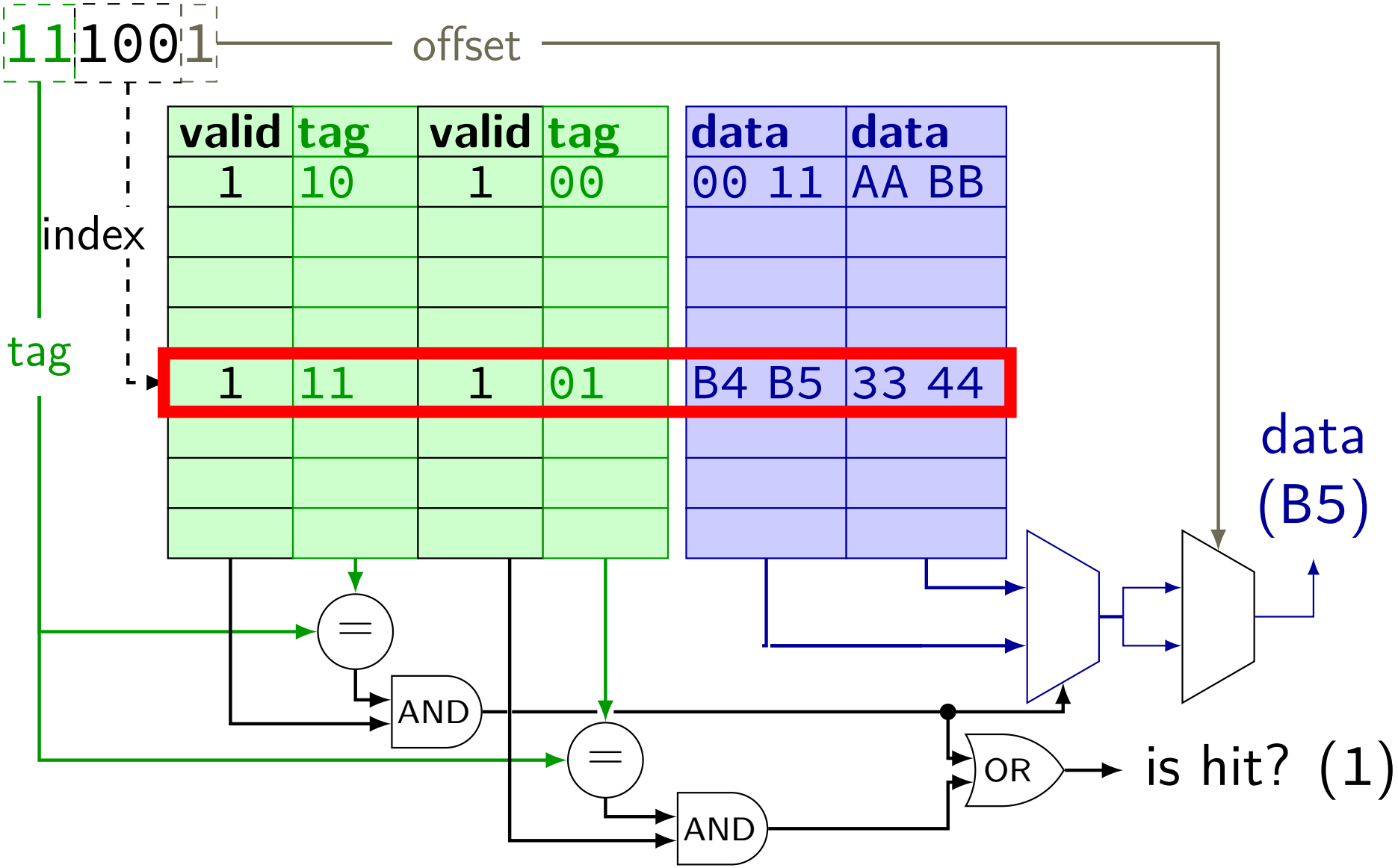
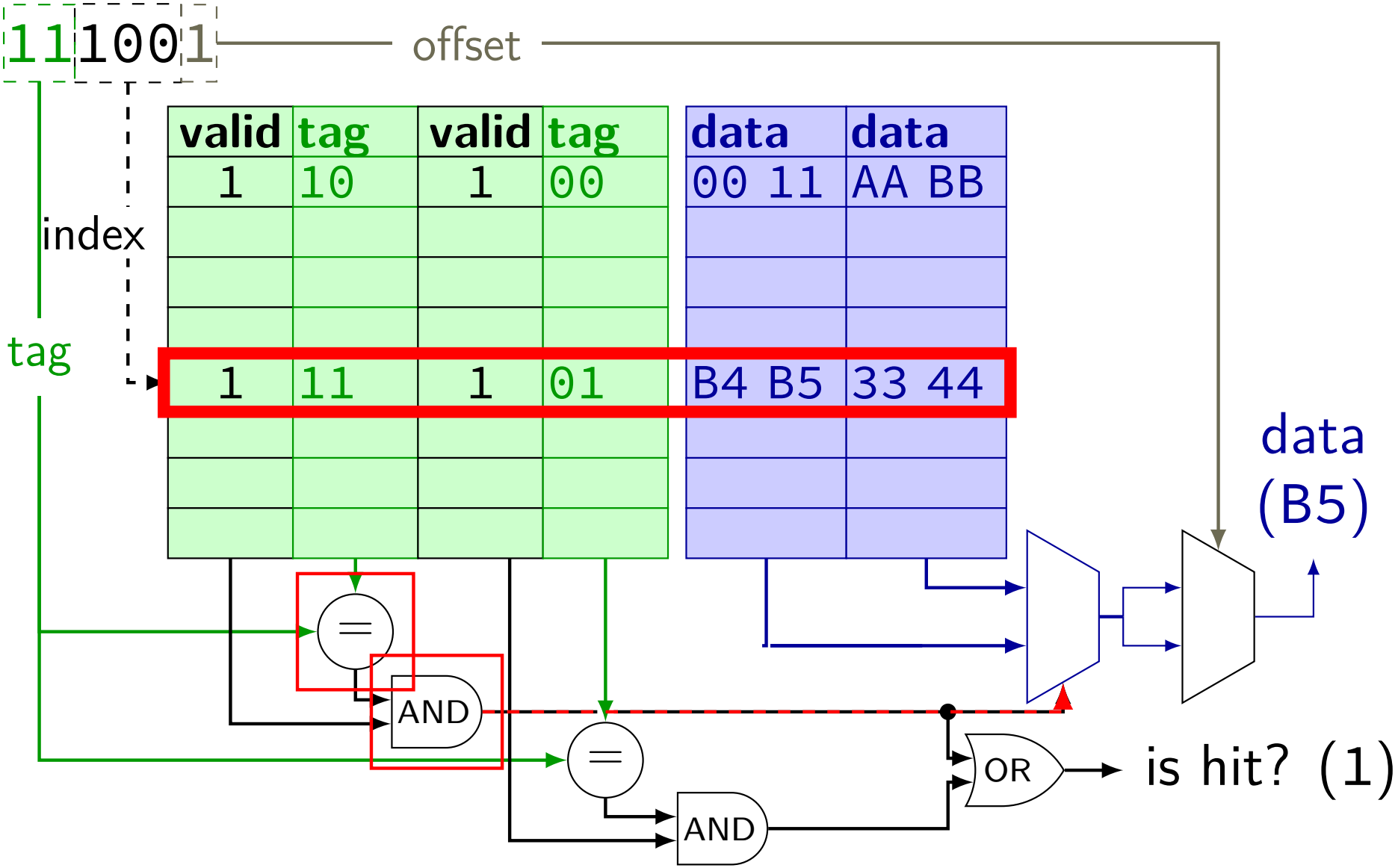
Cache Performance II

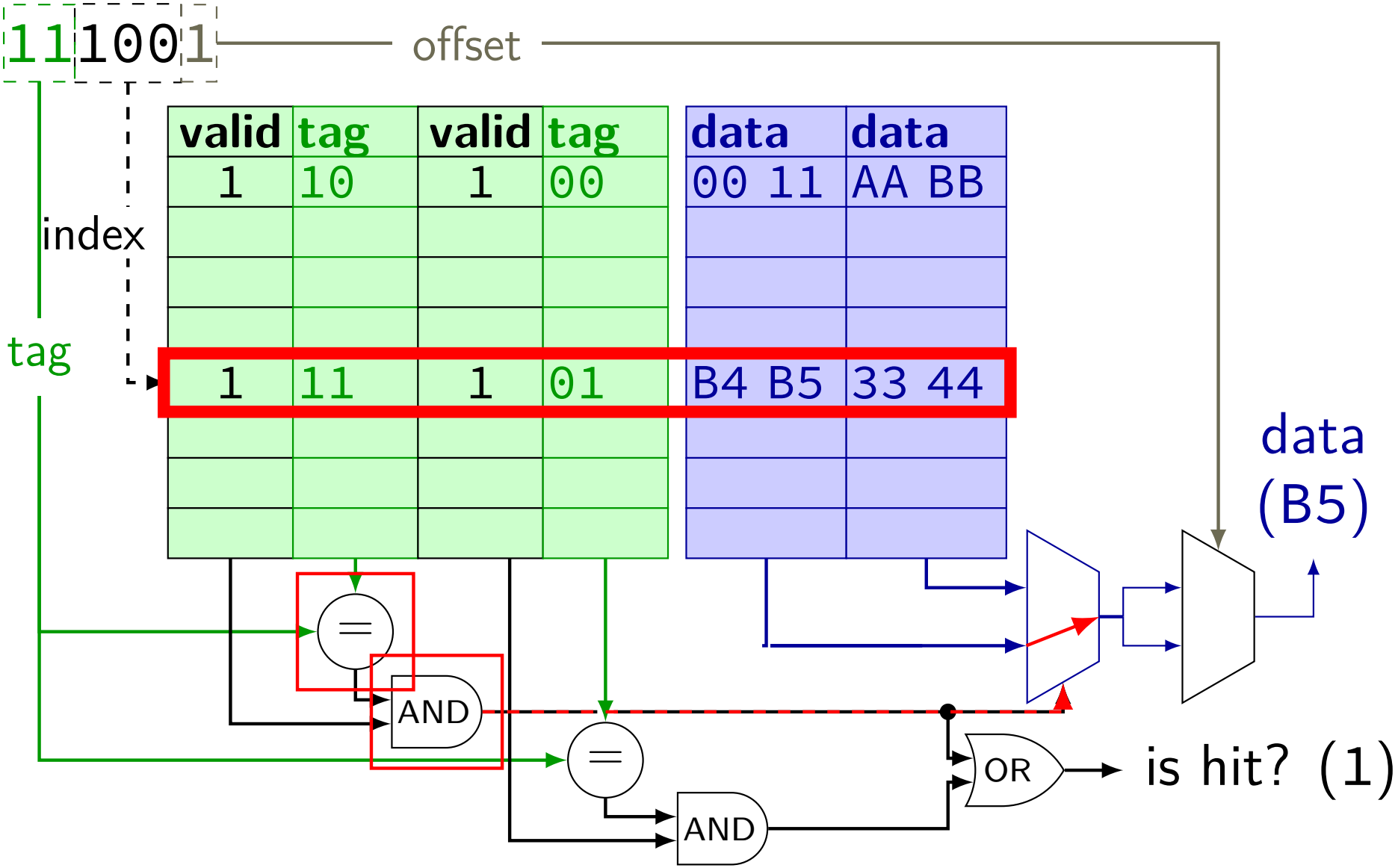
cache operation (associative)



cache operation (associative)



cache operation (associative)



Tag-Index-Offset formulas

m	memory addreses bits (Y86-64: 64)
E	number of blocks per set ("ways")
$S = 2^s$	number of sets
s	(set) index bits
$B = 2^b$	block size
b	(block) offset bits
$t = m - (s + b)$	tag bits
$C = B \times S \times E$	cache size (excluding metadata)

Tag-Index-Offset exercise

m	memory addreses bits (Y86-64: 64)
E	number of blocks per set ("ways")
$S = 2^s$	number of sets
s	(set) index bits
$B = 2^b$	block size
b	(block) offset bits
$t = m - (s + b)$	tag bits
$C = B \times S \times E$	cache size (excluding metadata)

My desktop:

L1 Data Cache: 32 KB, 8 blocks/set, 64 byte blocks
L2 Cache: 256 KB, 4 blocks/set, 64 byte blocks
L3 Cache: 8 MB, 16 blocks/set, 64 byte blocks

Divide the address `0x34567` into tag, index, offset for each cache.

T-I-O exercise: L1

quantity	value for L1
block size (given)	$B = 64$Byte
	$B = 2^b$ (b: block offset bits)

T-I-O exercise: L1

quantity	value for L1
block size (given)	$B = 64$Byte
	$B = 2^b$ (b: block offset bits)
block offset bits	$b = 6$

T-I-O exercise: L1

quantity	value for L1
block size (given)	$B = 64$Byte
	$B = 2^b$ (b: block offset bits)
block offset bits	$b = 6$
blocks/set (given)	$E = 8$
cache size (given)	$C = 32$KB $= E \times B \times S$

T-I-O exercise: L1

quantity	value for L1
block size (given)	$B = 64$Byte
	$B = 2^b$ (b: block offset bits)
block offset bits	$b = 6$
blocks/set (given)	$E = 8$
cache size (given)	$C = 32\text{KB} = E \times B \times S$
	$S = \dfrac{C}{B \times E}$ (S: number of sets)

T-I-O exercise: L1

quantity	value for L1
block size (given)	$B = 64$Byte
	$B = 2^b$ (b: block offset bits)
block offset bits	$b = 6$
blocks/set (given)	$E = 8$
cache size (given)	$C = 32\text{KB} = E \times B \times S$
	$S = \dfrac{C}{B \times E}$ (S: number of sets)
number of sets	$S = \dfrac{32\text{KB}}{64\text{Byte} \times 8} = 64$

T-I-O exercise: L1

quantity	value for L1
block size (given)	$B = 64\text{Byte}$
block offset bits	$B = 2^b$ (b: block offset bits) $b = 6$
blocks/set (given)	$E = 8$
cache size (given)	$C = 32\text{KB} = E \times B \times S$
number of sets	$S = \dfrac{C}{B \times E}$ (S: number of sets) $S = \dfrac{32\text{KB}}{64\text{Byte} \times 8} = 64$
set index bits	$S = 2^s$ (s: set index bits) $s = \log_2(64) = 6$

T-I-O results

	L1	L2	L3
sets	64	1024	8192
block offset bits	6	6	6
set index bits	6	10	13
tag bits		(the rest)	

T-I-O: splitting

	L1	L2	L3
block offset bits	6	6	6
set index bits	6	10	13
tag bits	(the rest)		

	3	4	5	6	7
`0x34567`:	`0011`	`0100`	`0101`	`0110`	`0111`

bits 0-5 (all offsets): `100111` = `0x27`

T-I-O: splitting

	L1	L2	L3
block offset bits	6	6	6
set index bits	6	10	13
tag bits	(the rest)		

	3	4	5	6	7
`0x34567`:	`0011`	`0100`	`0101`	`0110`	`0111`

bits 0-5 (all offsets): `100111` = `0x27`

T-I-O: splitting

	L1	L2	L3
block offset bits	6	6	6
set index bits	6	10	13
tag bits	(the rest)		

`0x34567`:	3	4	5	6	7
	`0011`	`0100`	`0101`	`0110`	`0111`

bits 0-5 (all offsets): `100111` = `0x27`

L1:

bits 6-11 (L1 set): `01 0101` = `0x15`
bits 12- (L1 tag): `0x34`

T-I-O: splitting

	L1	L2	L3
block offset bits	6	6	6
set index bits	6	10	13
tag bits	(the rest)		

	3	4	5	6	7
`0x34567`:	`0011`	`0100`	`0101`	`0110`	`0111`

bits 0-5 (all offsets): `100111` = `0x27`

L1:

bits 6-11 (L1 set): `01 0101` = `0x15`
bits 12- (L1 tag): `0x34`

T-I-O: splitting

	L1	L2	L3
block offset bits	6	6	6
set index bits	6	10	13
tag bits	(the rest)		

	3	4	5	6	7
`0x34567`:	`0011`	`0100`	`0101`	`0110`	`0111`

bits 0-5 (all offsets): `100111` = `0x27`

L2:

bits 6-15 (set for L2): `01 0001 0101` = `0x115`
bits 16-: `0x3`

T-I-O: splitting

	L1	L2	L3
block offset bits	6	6	6
set index bits	6	10	13
tag bits	(the rest)		

	3	4	5	6	7
`0x34567`:	`0011`	`0100`	`0101`	`0110`	`0111`

bits 0-5 (all offsets): `100111` = `0x27`

L2:

bits 6-15 (set for L2): `01 0001 0101` = `0x115`
bits 16-: `0x3`

T-I-O: splitting

	L1	L2	L3
block offset bits	6	6	6
set index bits	6	10	13
tag bits	(the rest)		

	3	4	5	6	7
`0x34567`:	`0011`	`0100`	`0101`	`0110`	`0111`

bits 0-5 (all offsets): `100111` = `0x27`

L3:

bits 6-18 (set for L3): `0 1101 0001 0101` = `0xD15`

bits 18-: `0x0`

matrix squaring

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

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

matrix squaring

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

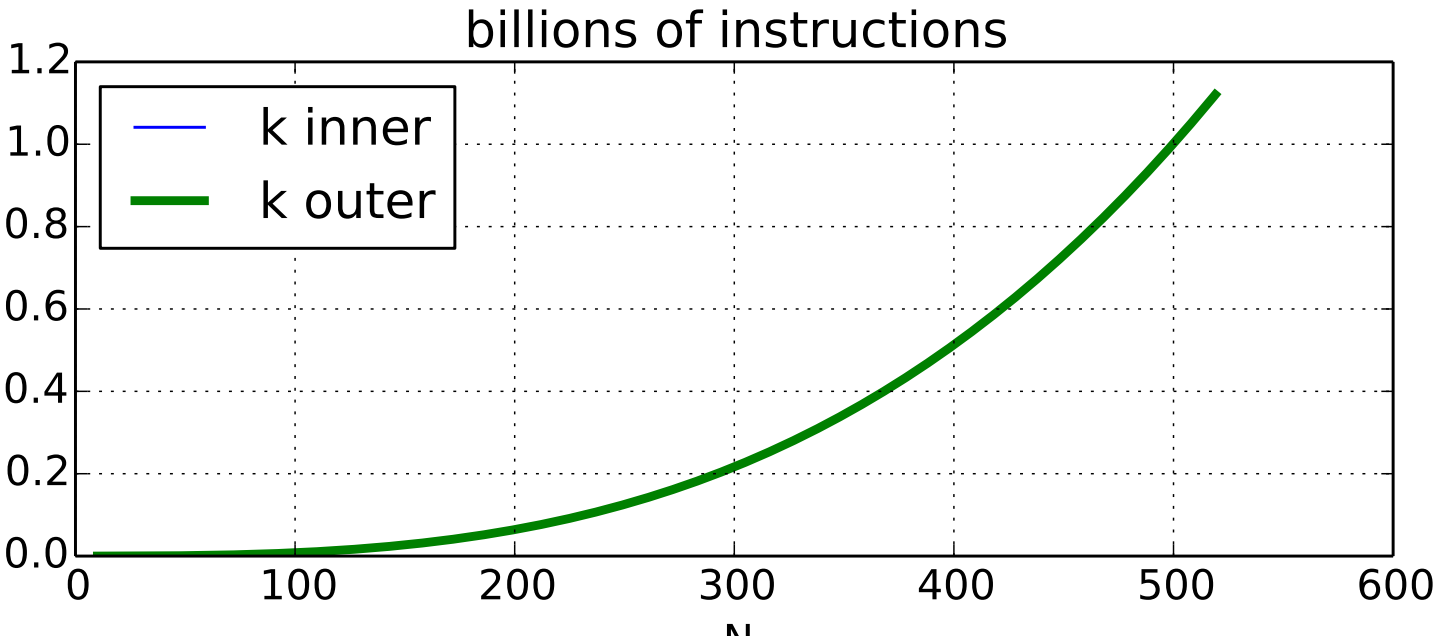
```
/* version 1: inner loop is k, middle is j*/
for (int i = 0; i < N; ++i)
  for (int j = 0; j < N; ++j)
    for (int k = 0; k < N; ++k)
      B[i*N+j] += A[i * N + k] * A[k * N + j];

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

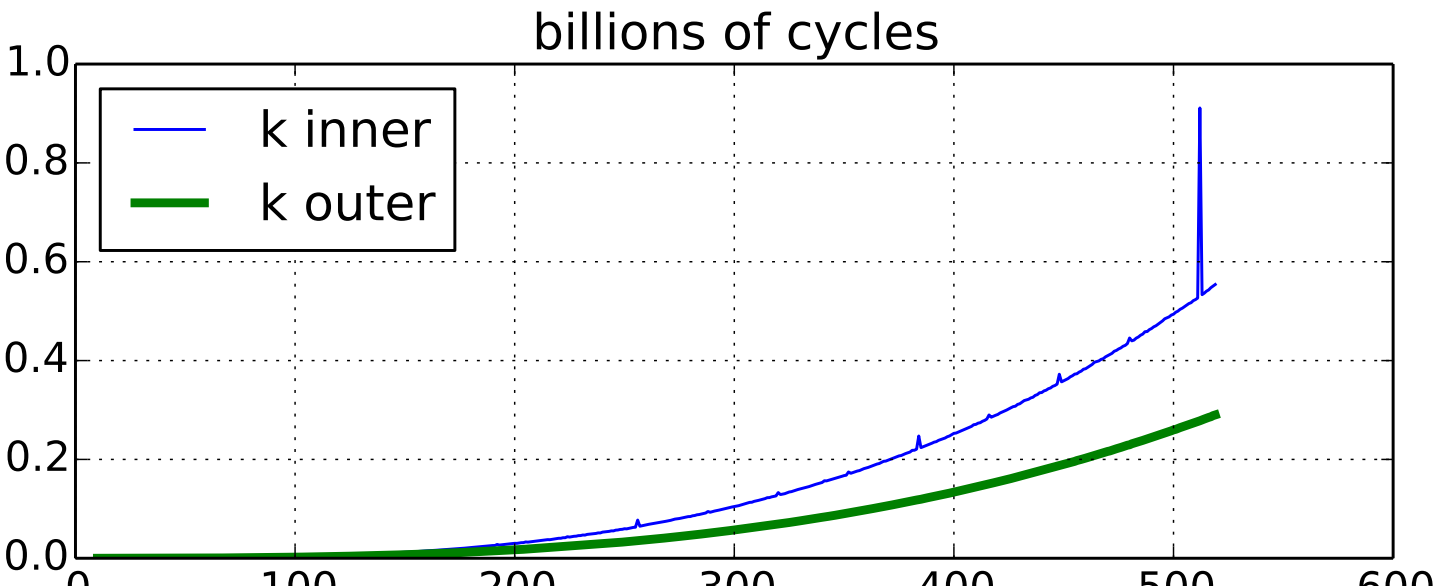
matrix squaring

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

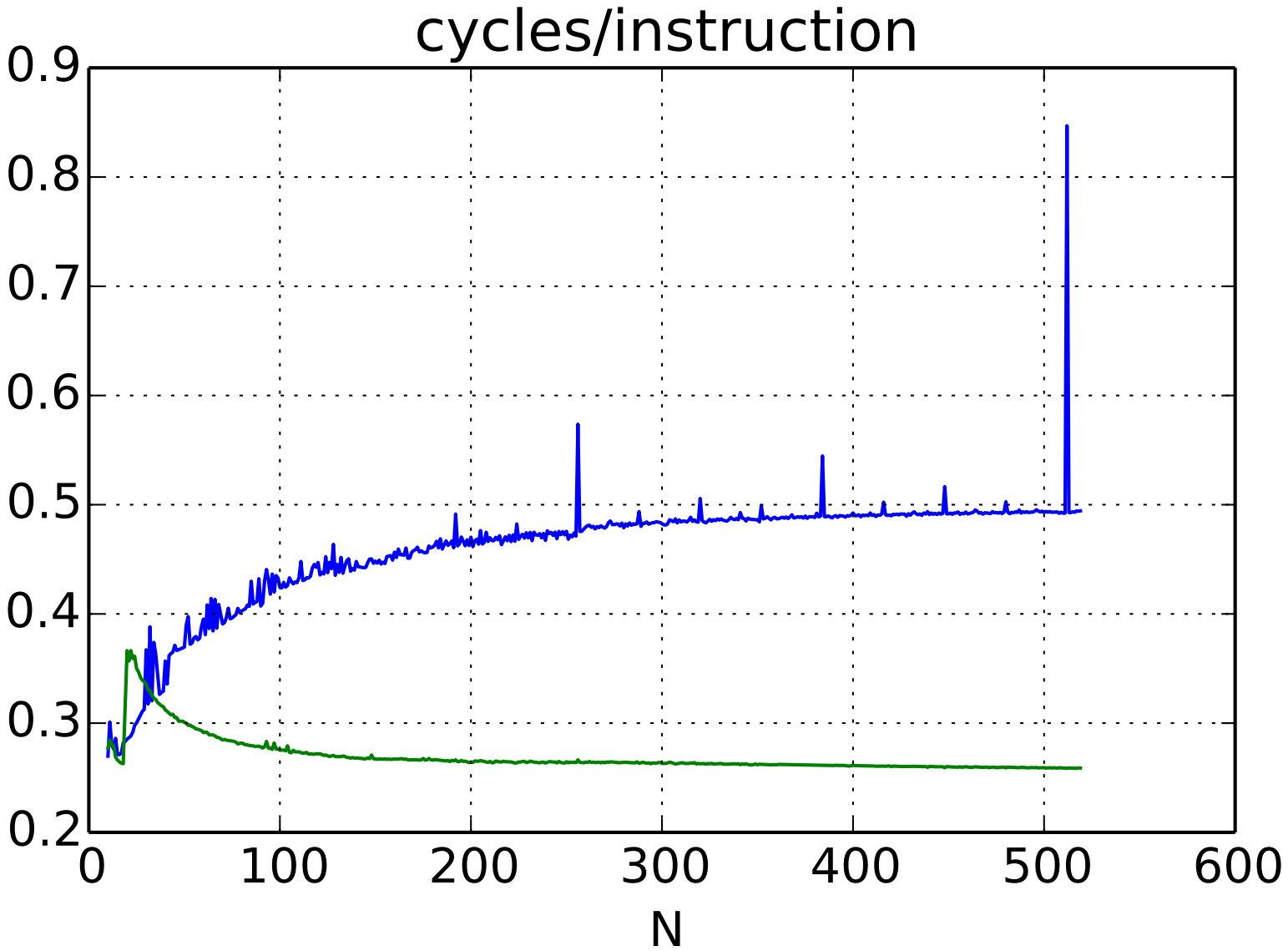
```
/* version 1: inner loop is k, middle is j*/
for (int i = 0; i < N; ++i)
  for (int j = 0; j < N; ++j)
    for (int k = 0; k < N; ++k)
      B[i*N+j] += A[i * N + k] * A[k * N + j];

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

performance

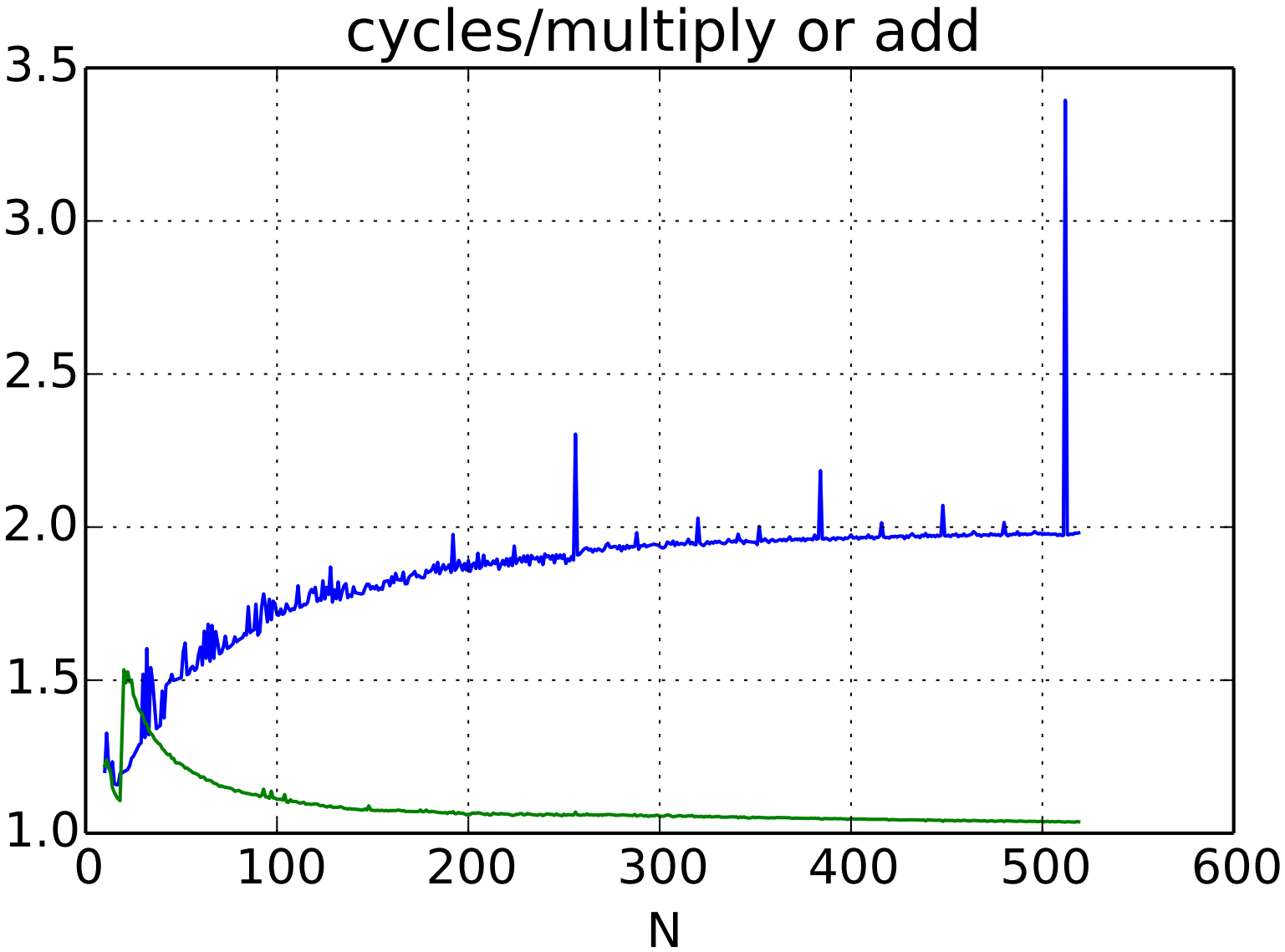




alternate view 1: cycles/instruction



alternate view 2: cycles/operation



loop orders and locality

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

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

kij order: A_{ik} has temporal locality

... better than ...

ijk order: A_{ik} has spatial locality

ijk order: B_{ij} has temporal locality

loop orders and locality

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

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

kij order: A_{ik} has temporal locality

... better than ...

ijk order: A_{ik} has spatial locality

ijk order: B_{ij} has temporal locality

matrix squaring

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

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

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

matrix squaring

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

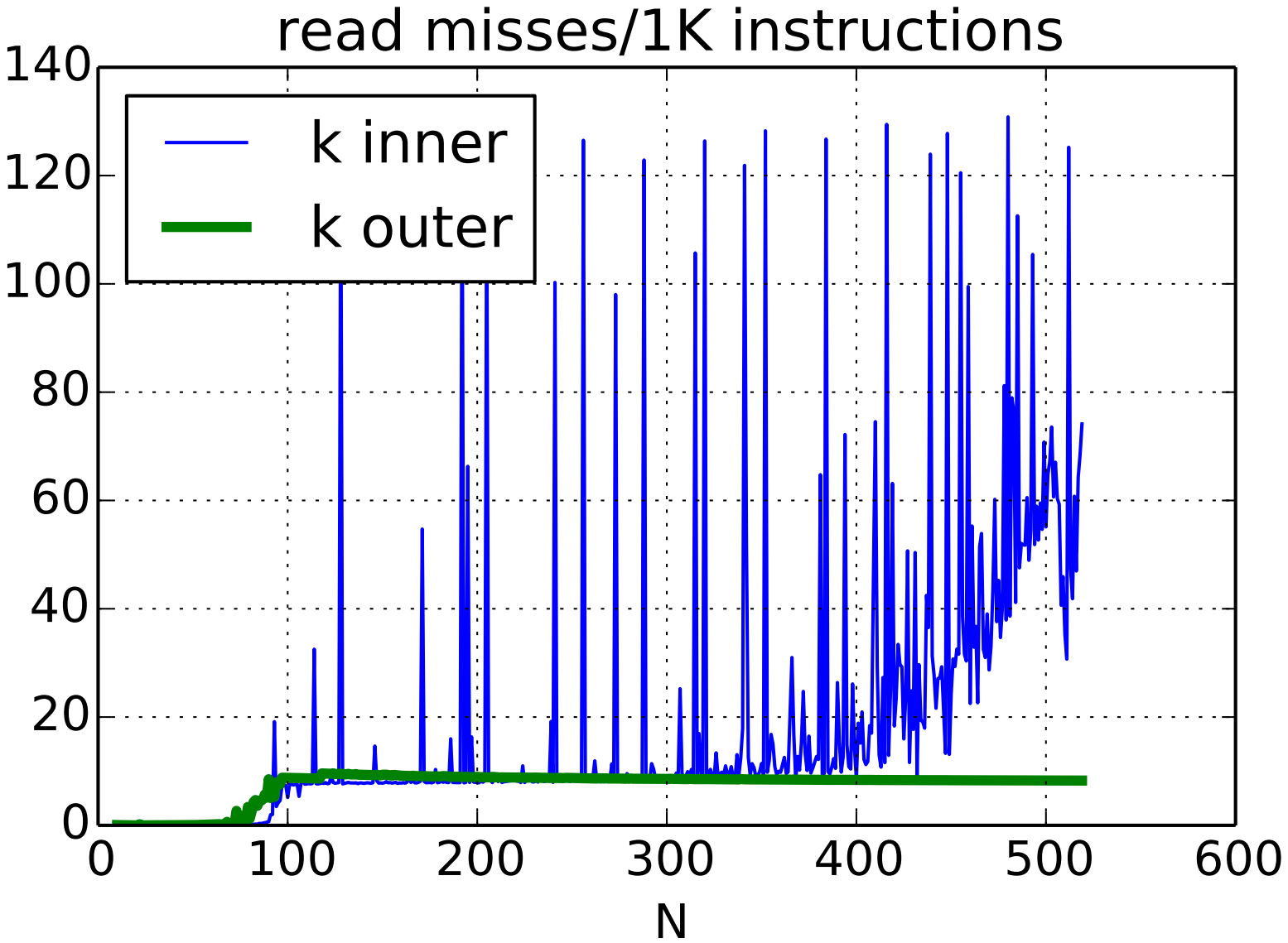
```
/* version 1: inner loop is k, middle is j*/
for (int i = 0; i < N; ++i)
  for (int j = 0; j < N; ++j)
    for (int k = 0; k < N; ++k)
      B[i*N+j] += A[i * N + k] * A[k * N + j];

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

matrix squaring

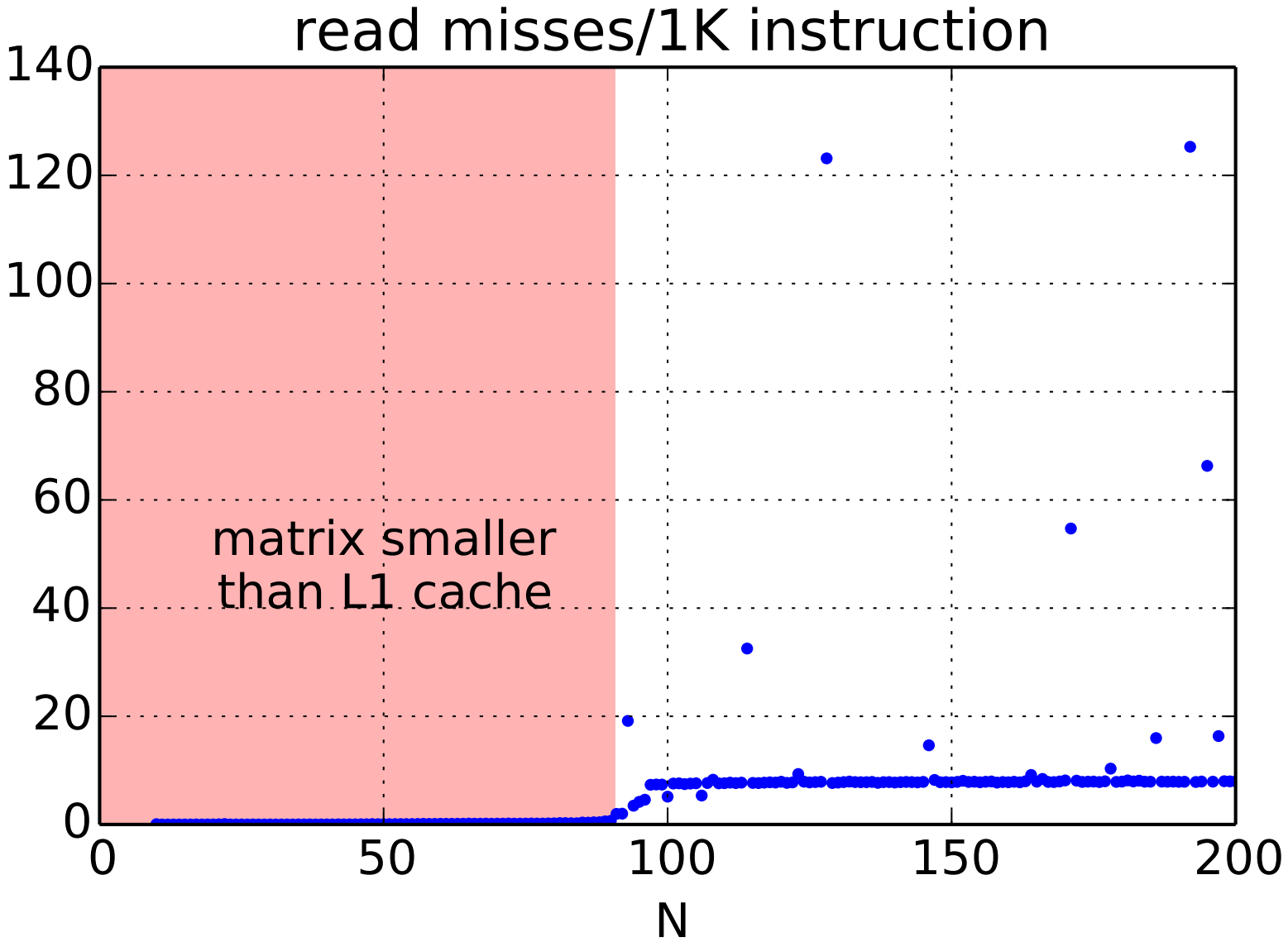
$$B_{ij} = \sum_{k=1}^{n} A_{ik} \times A_{kj}$$

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

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

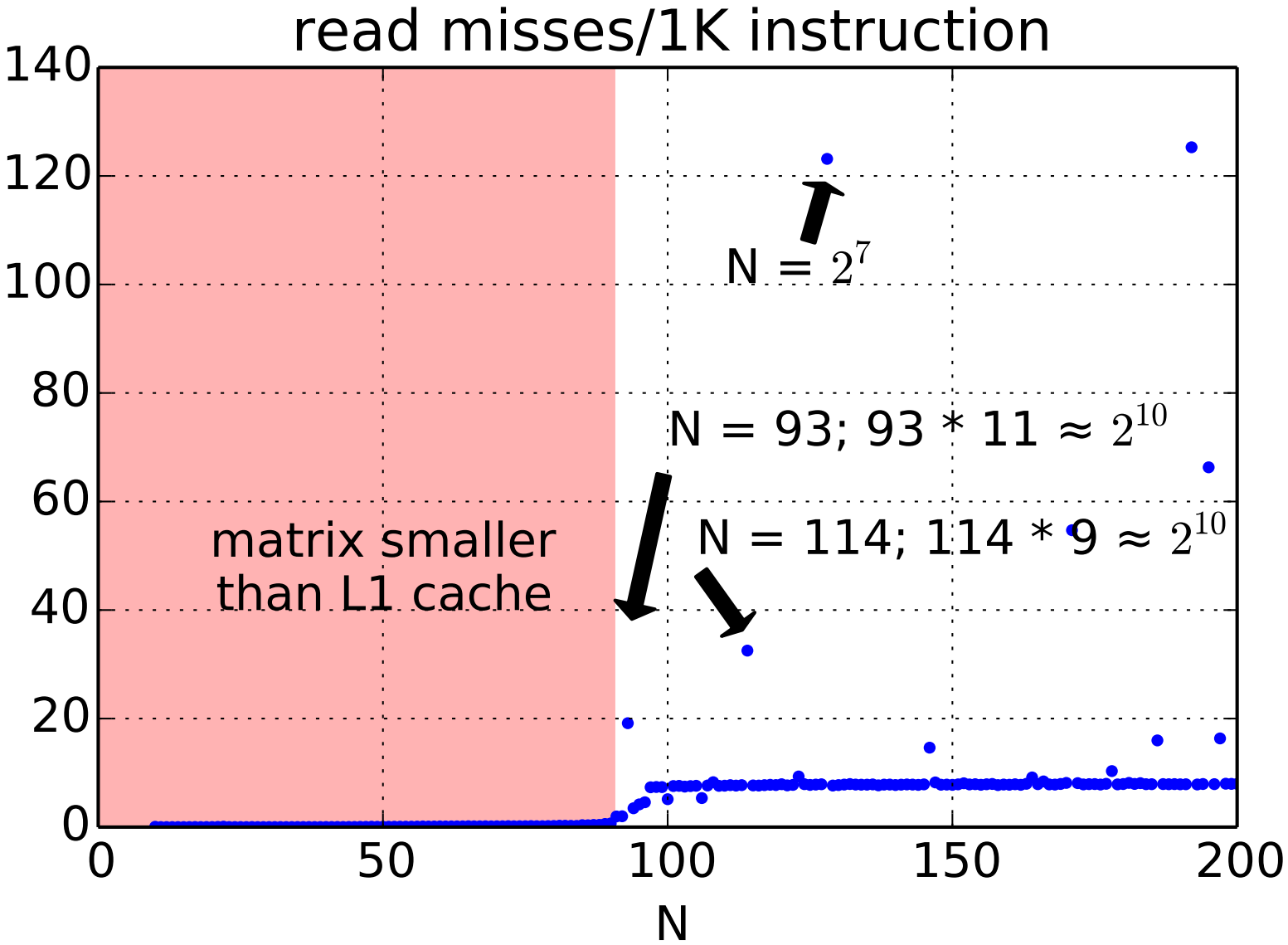
L1 misses


read misses/1K instructions
k inner
k outer
140
120
100
80
60
40
20
0
0
100
200
300
400
500
600
N

L1 miss detail (1)



L1 miss detail (2)



addresses

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

addresses

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

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

conflict misses

powers of two — lower order bits unchanged

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

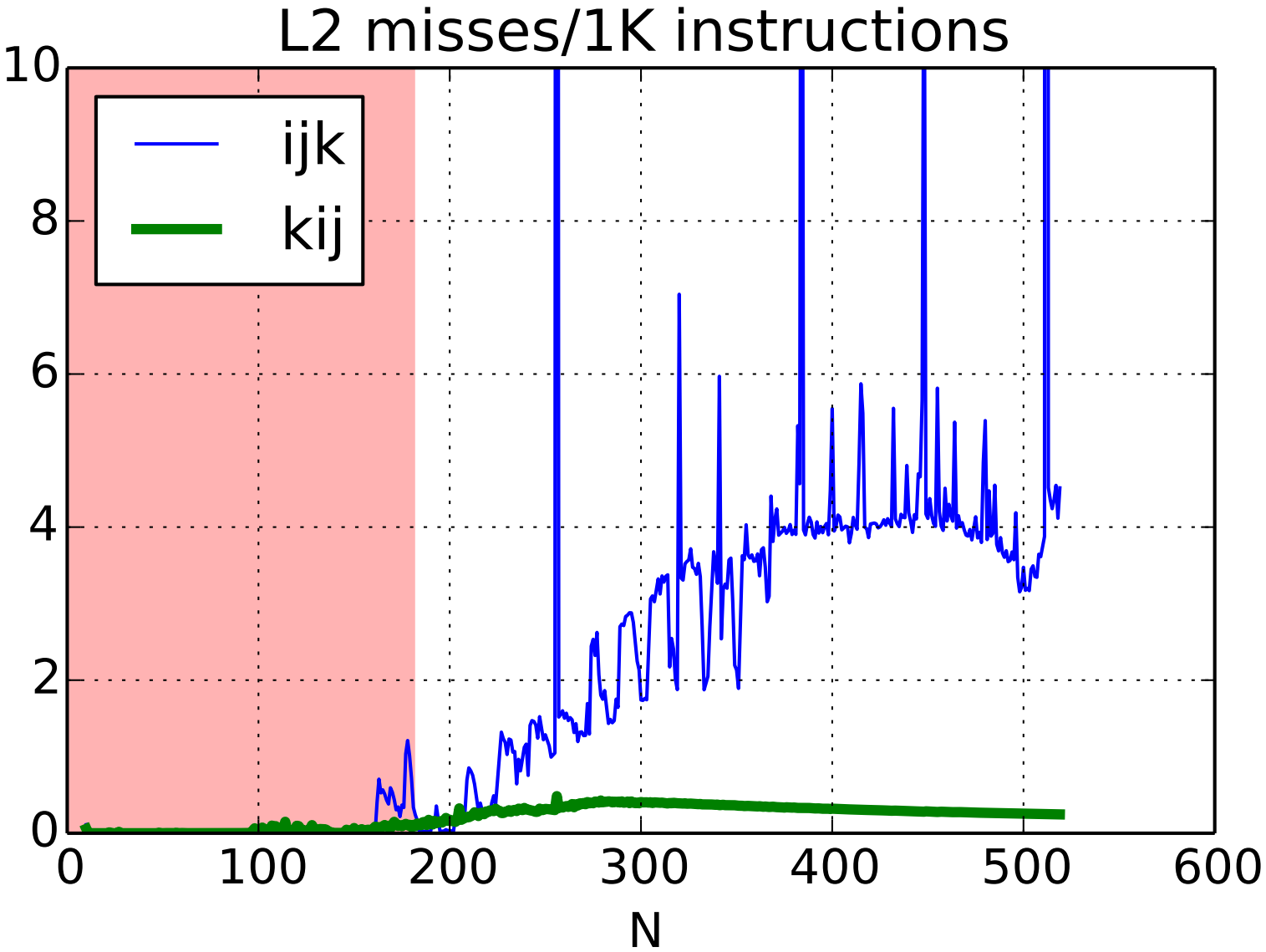
1023 elements apart (4092 bytes; 63.9 cache blocks)

64 sets in L1 cache: usually maps to same set

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

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

L2 misses


L2 misses/1K instructions
ijk
kij
0
2
4
6
8
10
0
100
200
300
400
500
600
N

systematic approach (1)

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

goal: get most out of each cache miss

if N is larger than the cache:

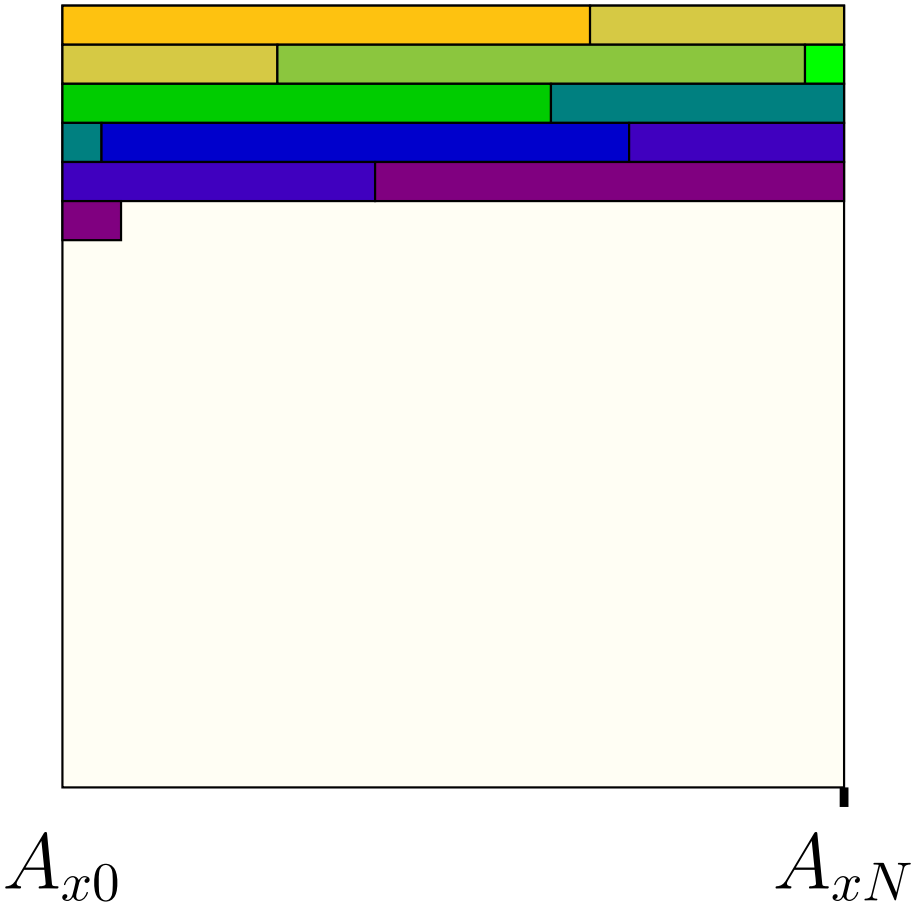
miss for B_{ij} — 1 comptuation

miss for A_{ik} — N computations

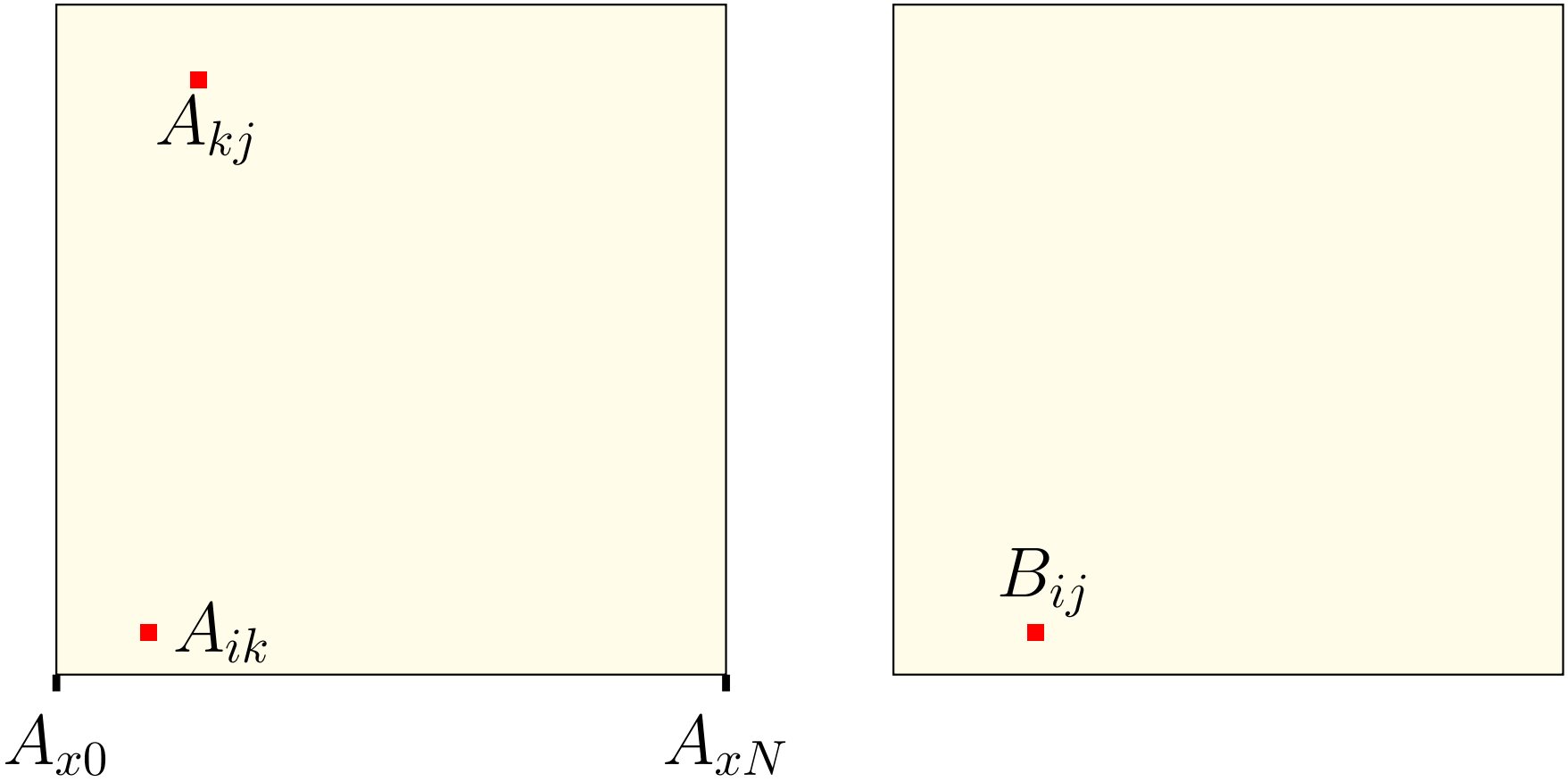
miss for A_{kj} — 1 computation

effectively caching just 1 element

‘flat’ 2D arrays and cache blocks



array usage: kij order

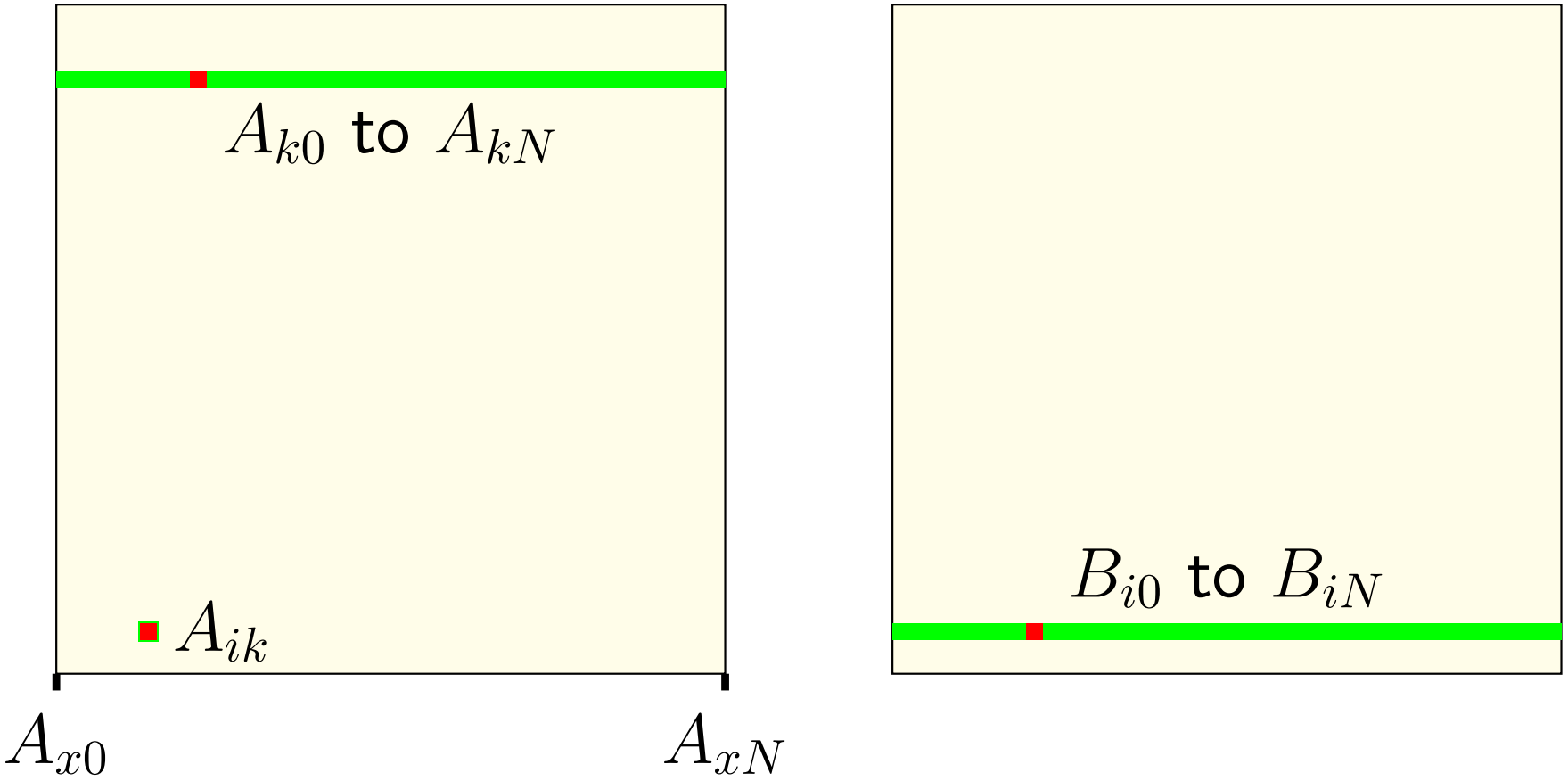


for all k: for all i: for all j: $B_{ij} += A_{ik} \times A_{kj}$

N calculations for A_{ik}

1 for A_{kj}, B_{ij}

array usage: kij order

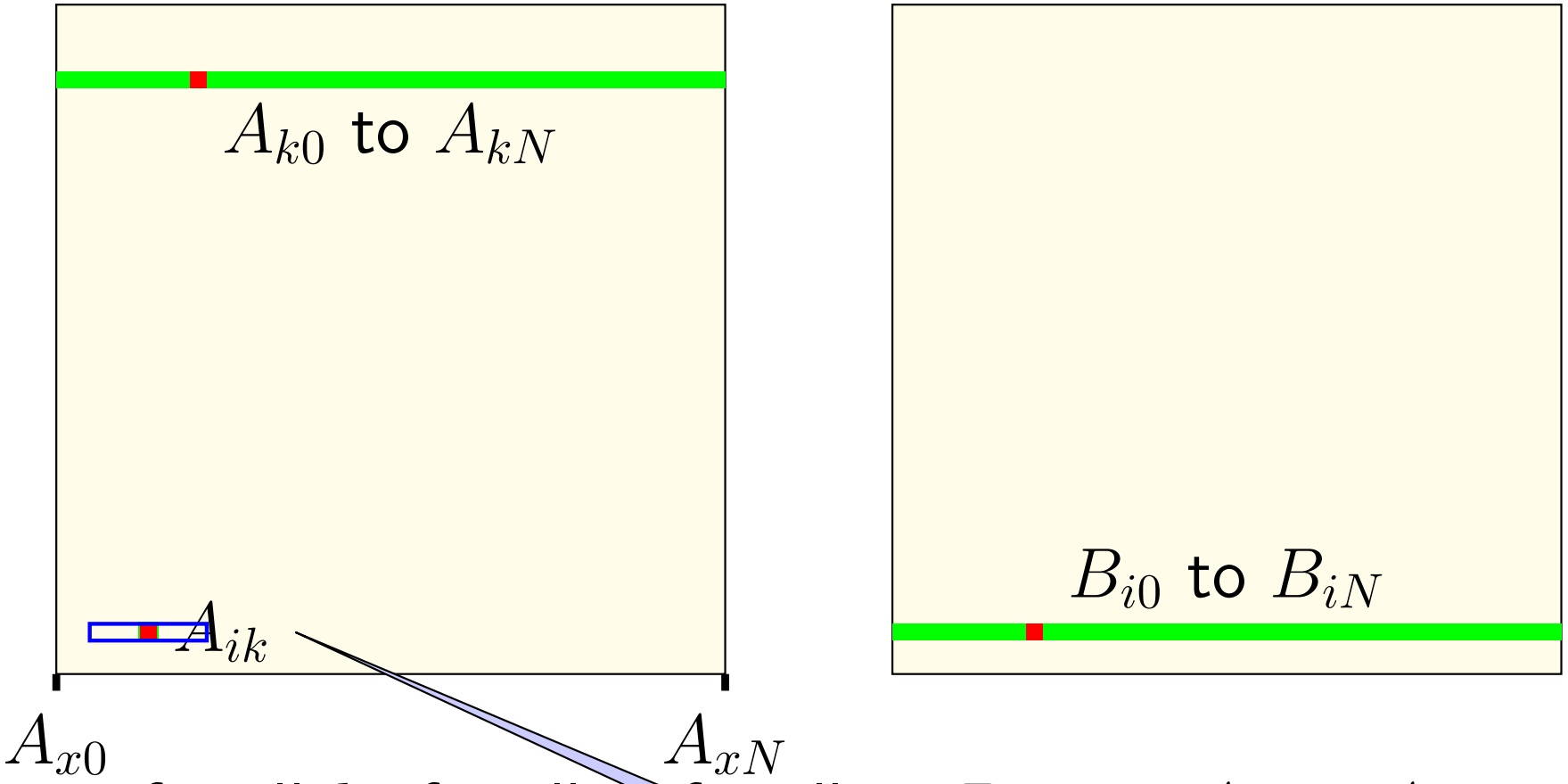


for all k: for all i: for all j: $B_{ij} += A_{ik} \times A_{kj}$

N calculations for A_{ik}

1 for A_{kj}, B_{ij}

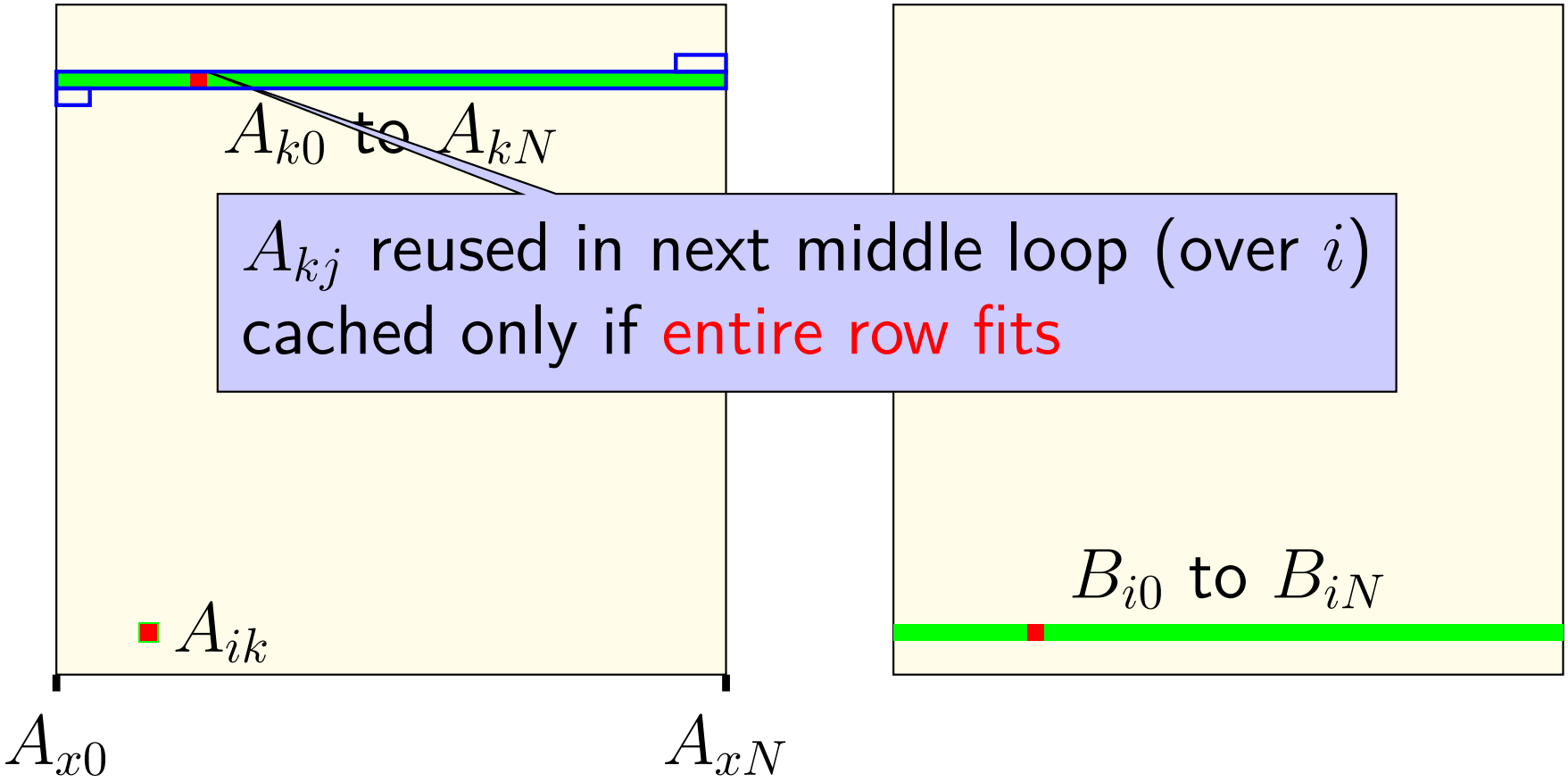
array usage: kij order



for

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

array usage: kij order

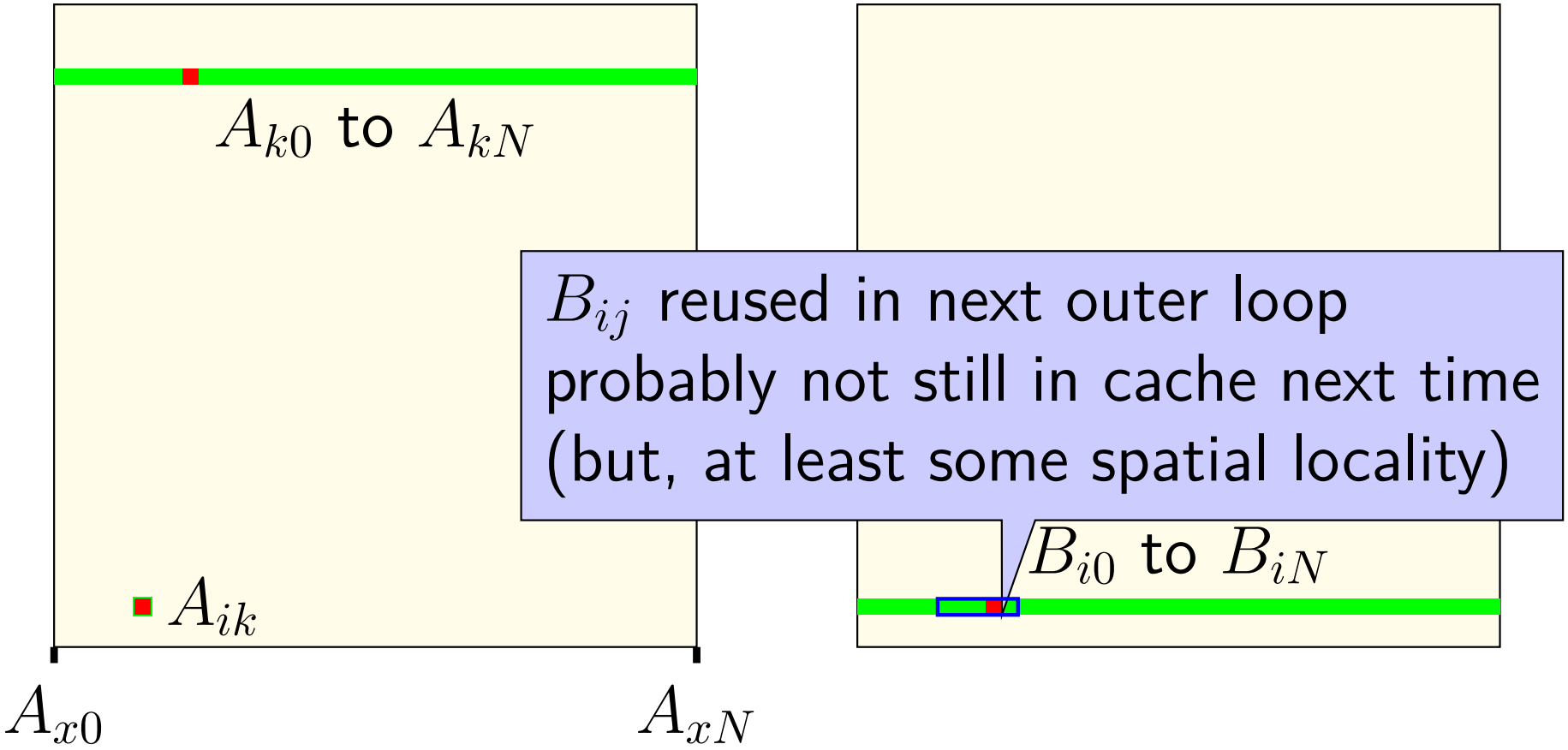


for all k: for all i: for all j: $B_{ij} += A_{ik} \times A_{kj}$

N calculations for A_{ik}

1 for A_{kj}, B_{ij}

array usage: kij order



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

N calculations for A_{ik}

1 for A_{kj}, B_{ij}

inefficiencies

if N is large enough that a row doesn't fit in cache:

keeping one block in cache
accessing one element of that block

if N is small enough that a row does fit in cache:

keeping one block in cache for A_{ik}, using one element
keeping row in cache for A_{kj}, using N times

systematic approach (2)

```
for (int k = 0; k < N; ++k) {
  for (int i = 0; i < N; ++i) {
    A_ik loaded once in this loop (N^2 times):
    for (int j = 0; j < N; ++j)
      B_ij, A_kj loaded each iteration (if N big):
      B[i*N+j] += A[i*N+k] * A[k*N+j];
```

N^3 multiplies, N^3 adds

about 1 load per operation

a transformation

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

split the loop over k — should be exactly the same

(assuming even N)

a transformation

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

split the loop over k — should be exactly the same

(assuming even N)

simple blocking

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

now reorder split loop

simple blocking

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

now reorder split loop

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": */
      B[i*N+j] += A[i*N+kk] * A[kk*N+j];
      B[i*N+j] += A[i*N+kk+1] * A[(kk+1)*N+j];
    }
  }
}
```

simple blocking – expanded

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

Temporal locality in B_{ij}s

simple blocking – expanded

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

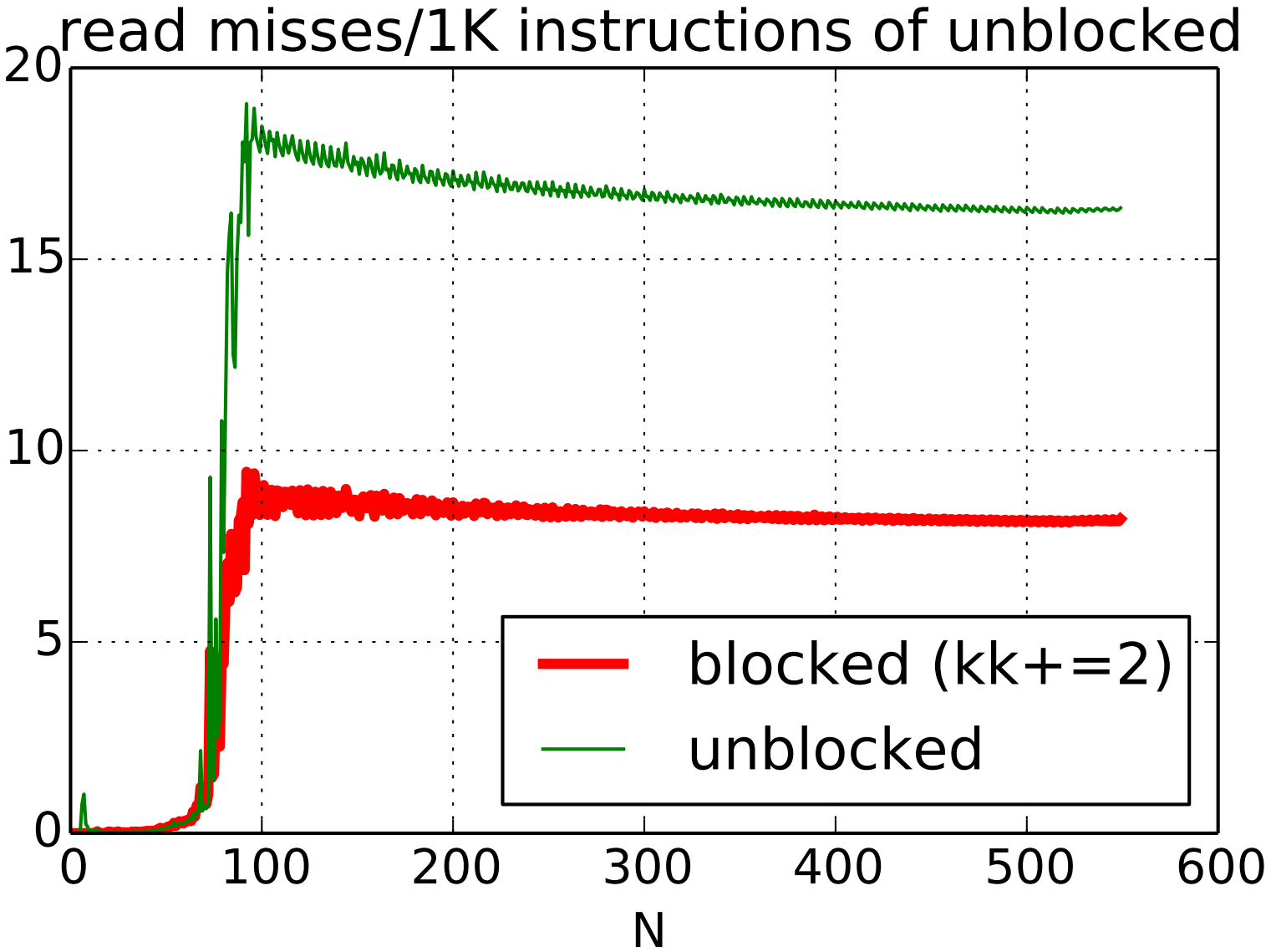
More spatial locality in A_{ik}

simple blocking – expanded

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

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

improvement in read misses



simple blocking (2)

same thing for i in addition to k?

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

simple blocking — expanded

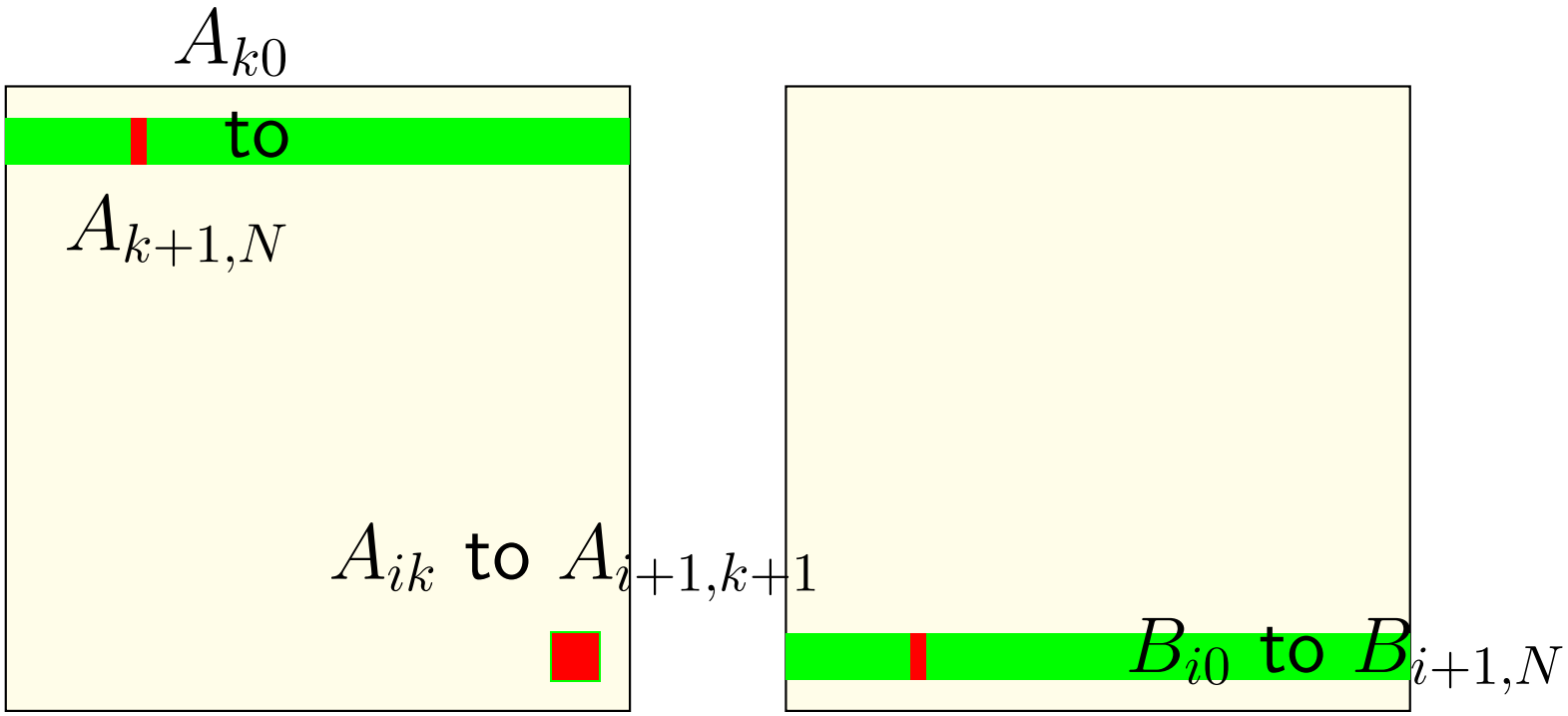
```
for (int k = 0; k < N; k += 2) {
  for (int i = 0; i < N; i += 2) {
    for (int j = 0; j < N; ++j) {
      /* process a "block": */
      B_{i+0,j}   += A_{i+0,k+0}   * A_{k+0,j}
      B_{i+0,j}   += A_{i+0,k+1}   * A_{k+1,j}
      B_{i+1,j}   += A_{i+1,k+0}   * A_{k+0,j}
      B_{i+1,j}   += A_{i+1,k+1}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

simple blocking — expanded

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

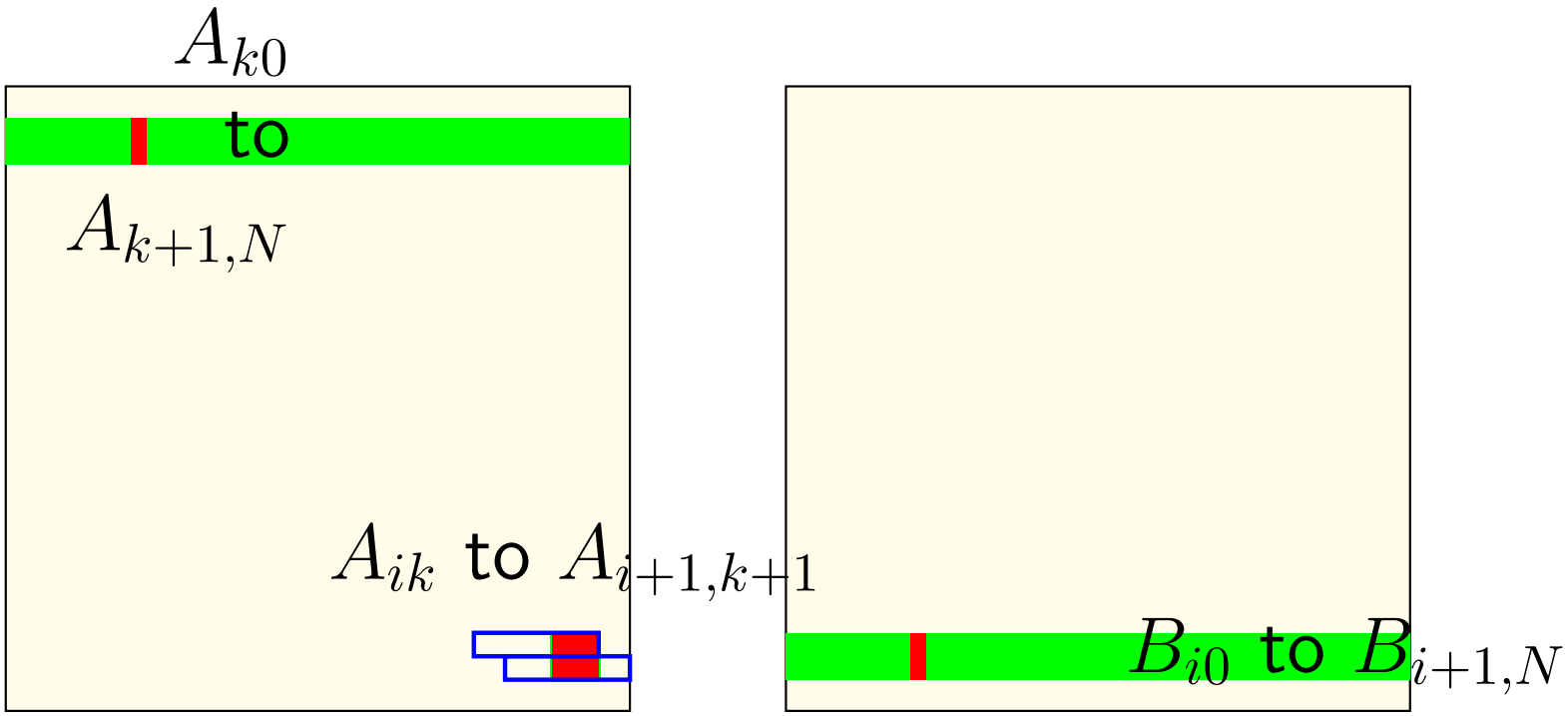
Now A_{kj} reused in inner loop — more calculations per load!

array usage (better)



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

array usage (better)



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

generalizing cache blocking

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

generalizing cache blocking

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

B_{ij} used K times for one miss

generalizing cache blocking

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

A_{ik} used $> J$ times for one miss

generalizing cache blocking

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

A_{kj} used I times for one miss

generalizing cache blocking

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

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

keeping values in cache

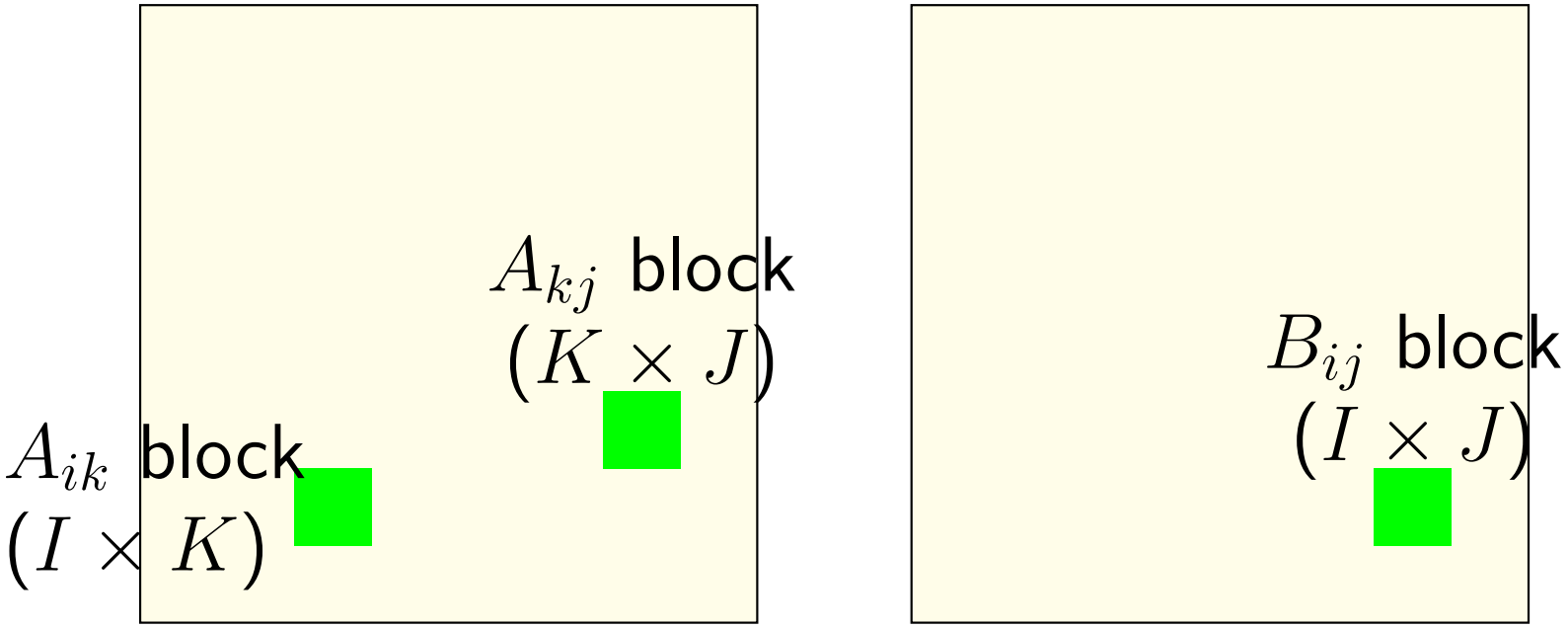
can't *explicitly* ensure values are kept in cache

...but reusing values *effectively* does this

- cache will try to keep recently used values

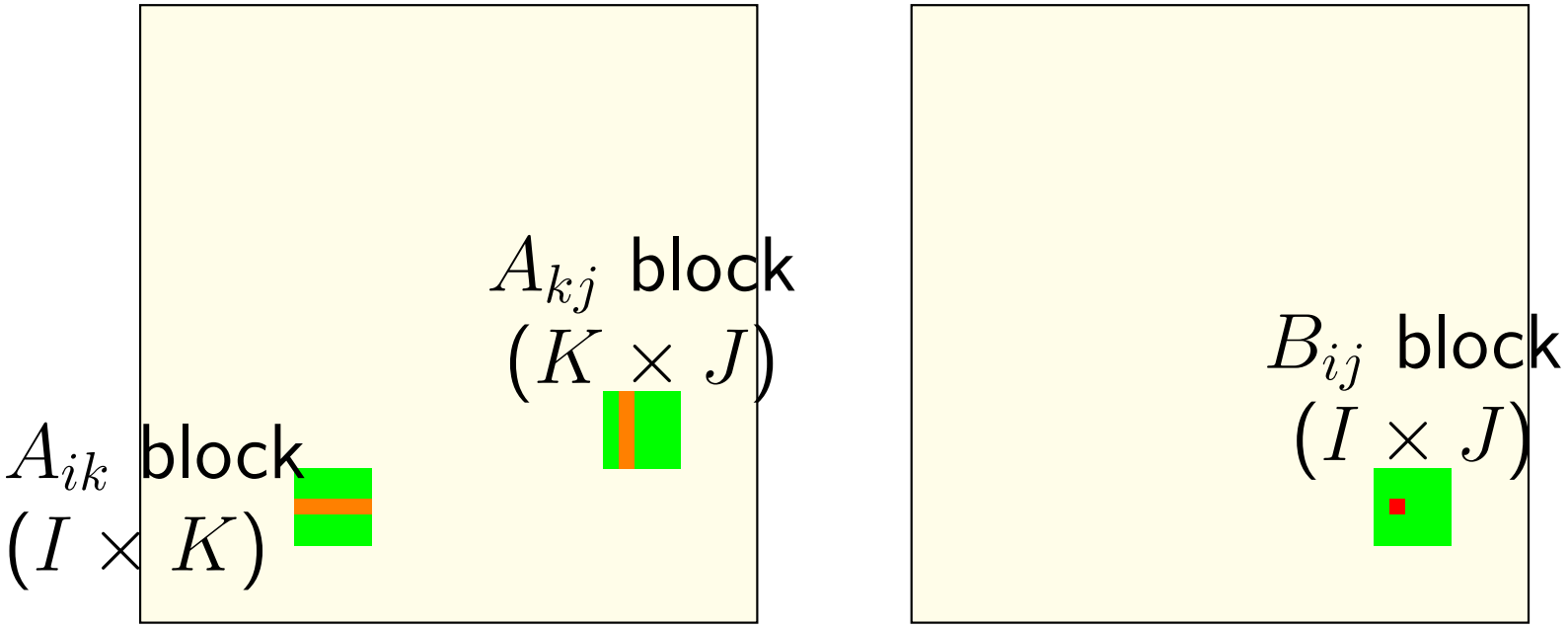
cache optimization idea: choose what's in the cache

array usage: block



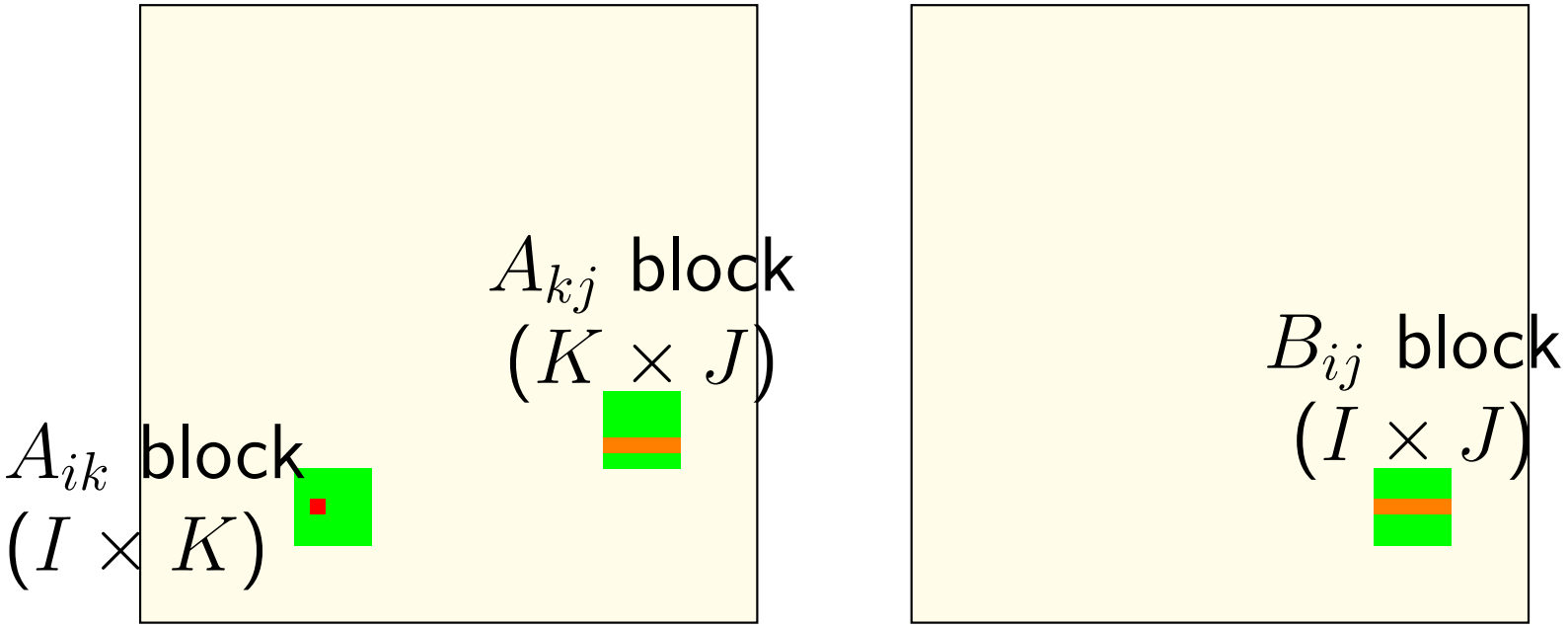
inner loop keeps "blocks" from A, B in cache

array usage: block



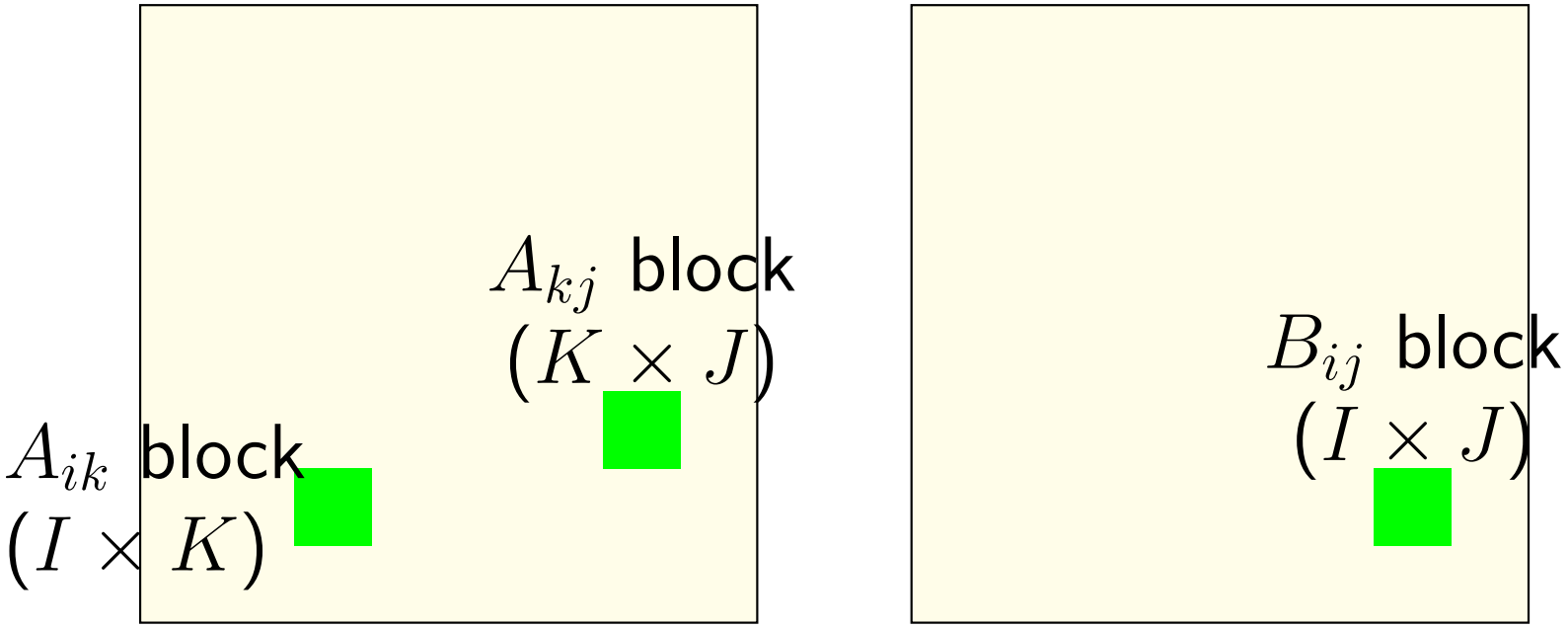
B_{ij} calculation uses strips from A
K calculations for one load (cache miss)

array usage: block



A_{ik} calculation uses strips from A, B
J calculations for one load (cache miss)

array usage: block



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

cache blocking efficiency

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

do $> J$ multiplies with each

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

do I multiplies with each

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

do K adds with each

bigger blocks — more work per load!

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

cache blocking rule of thumb

fill the most of the cache with useful data

and do as much work as possible from that

example: my desktop 32KB L1 cache

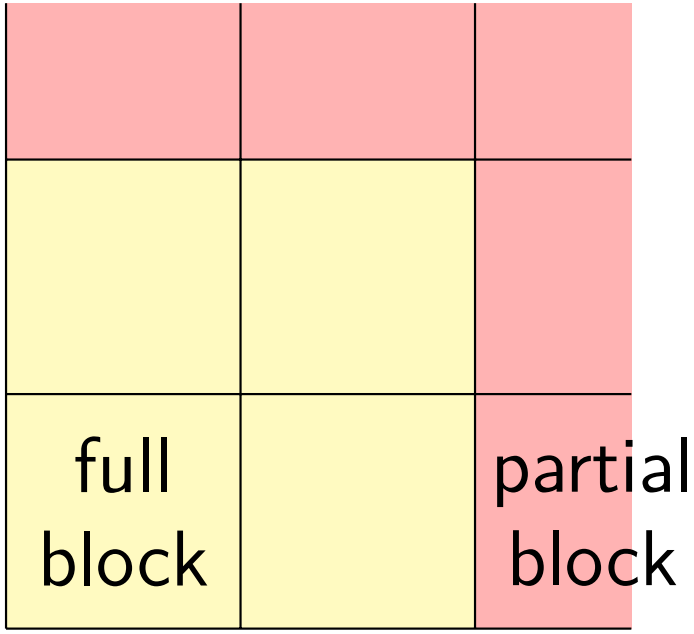
$I = J = K = 48$ uses $48^2 \times 3$ elements, or 27KB.

assumption: conflict misses aren't important

view 2: divide and conquer

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

cache blocking ugliness — fringe



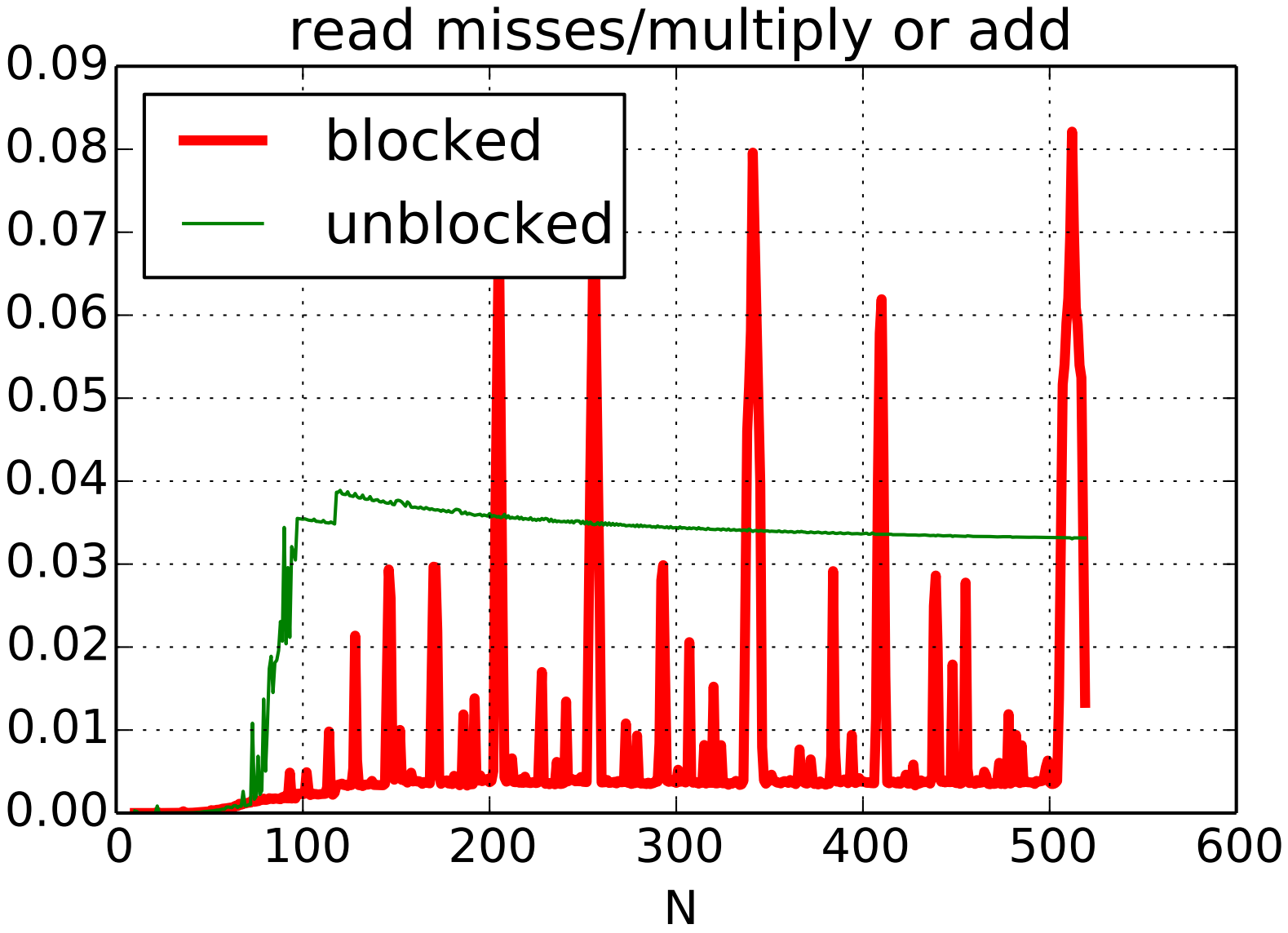
cache blocking ugliness — fringe

```
for (int kk = 0; kk < N; kk += K) {
  for (int ii = 0; ii < N; ii += I) {
    for (int jj = 0; jj < N; jj += J) {
      for (int k = kk; k < min(kk+K, N) ; ++k) {
          // ...
      }
    }
  }
}
```

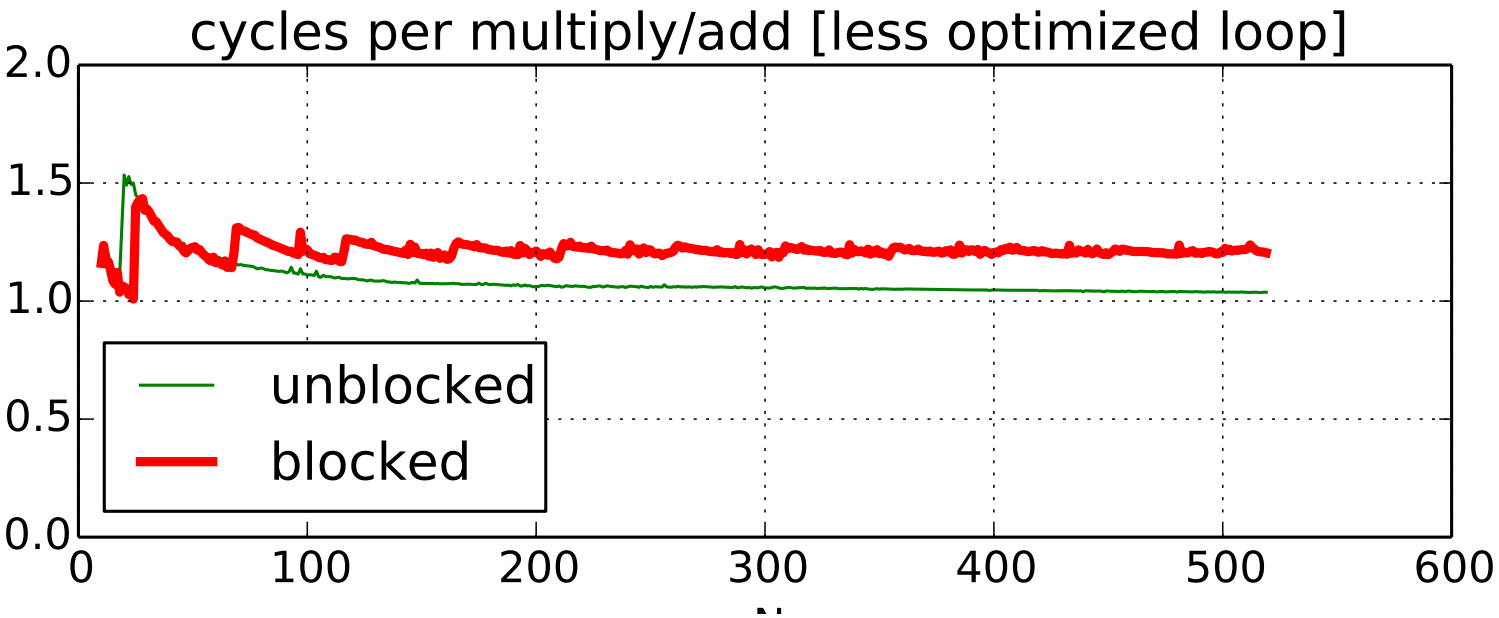
cache blocking ugliness — fringe

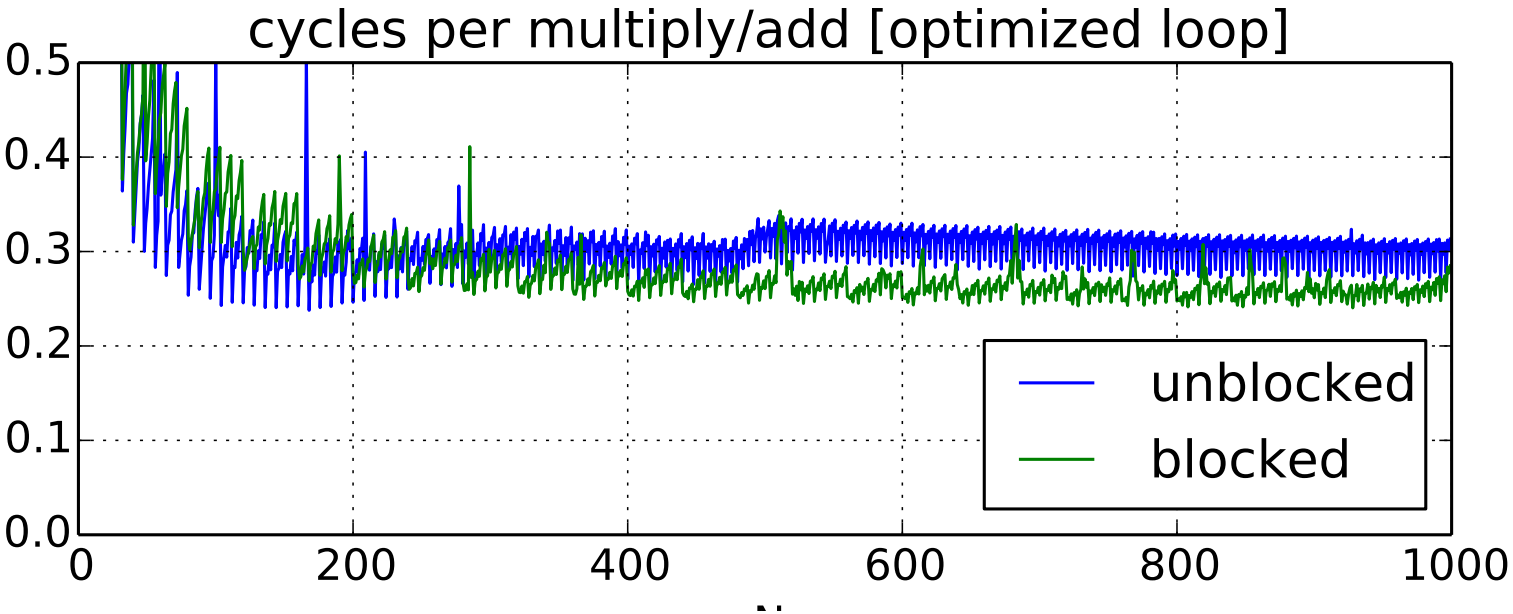
```
for (kk = 0; kk + K <= N; kk += K) {
  for (ii = 0; ii + I <= N; ii += I) {
    for (jj = 0; jj + J <= N; ii += J) {
      // ...
    }
    for (; jj < N; ++jj) {
      // handle remainder
    }
  }
  for (; ii < N; ++ii) {
    // handle remainder
  }
}
for (; kk < N; ++kk) {
  // handle remainder
}
```

cache blocking and miss rate

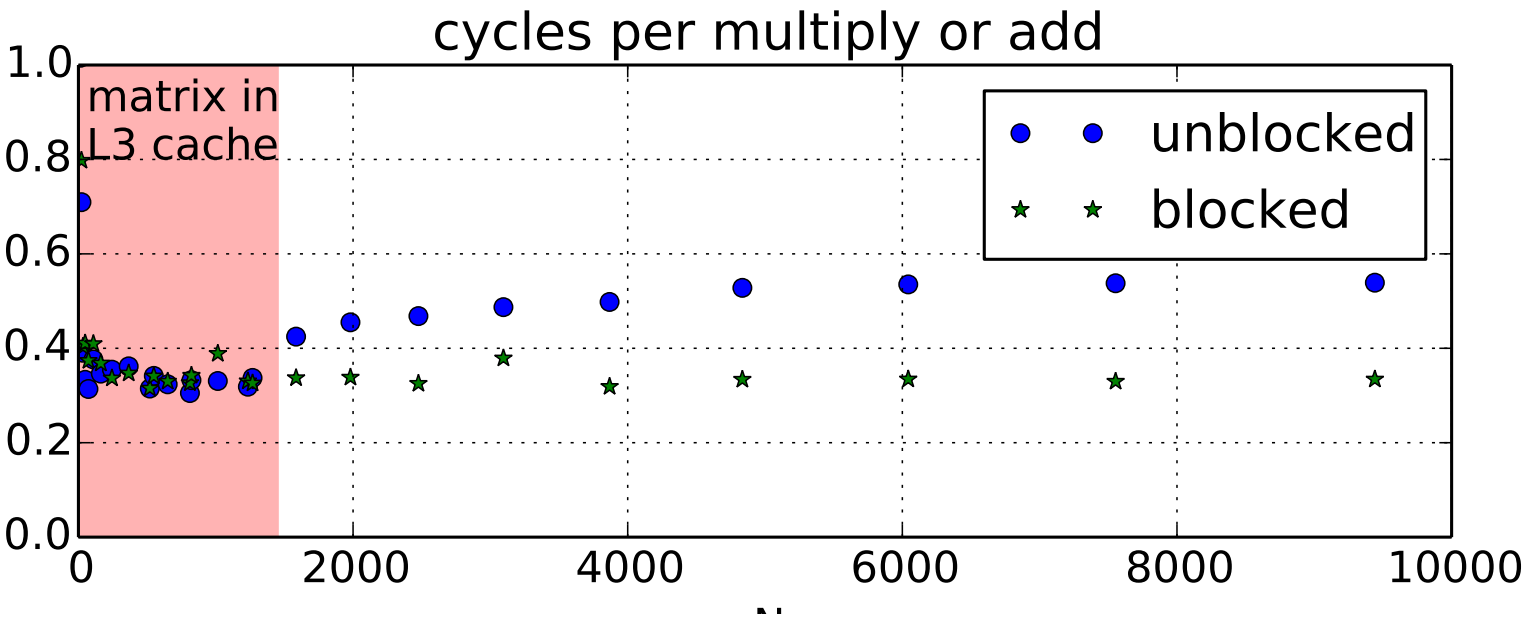


what about performance?





performance for big sizes



optimized loop???

performance difference wasn't visible at small sizes

until I optimized arithmetic in the loop

(by supplying better options to GCC)

1: loading $B_{i,j}$ through $B_{i,j+7}$ with one instruction

2: doing adds and multiplies with less instructions

3: simplifying address computations

optimized loop???

performance difference wasn't visible at small sizes

until I optimized arithmetic in the loop

(by supplying better options to GCC)

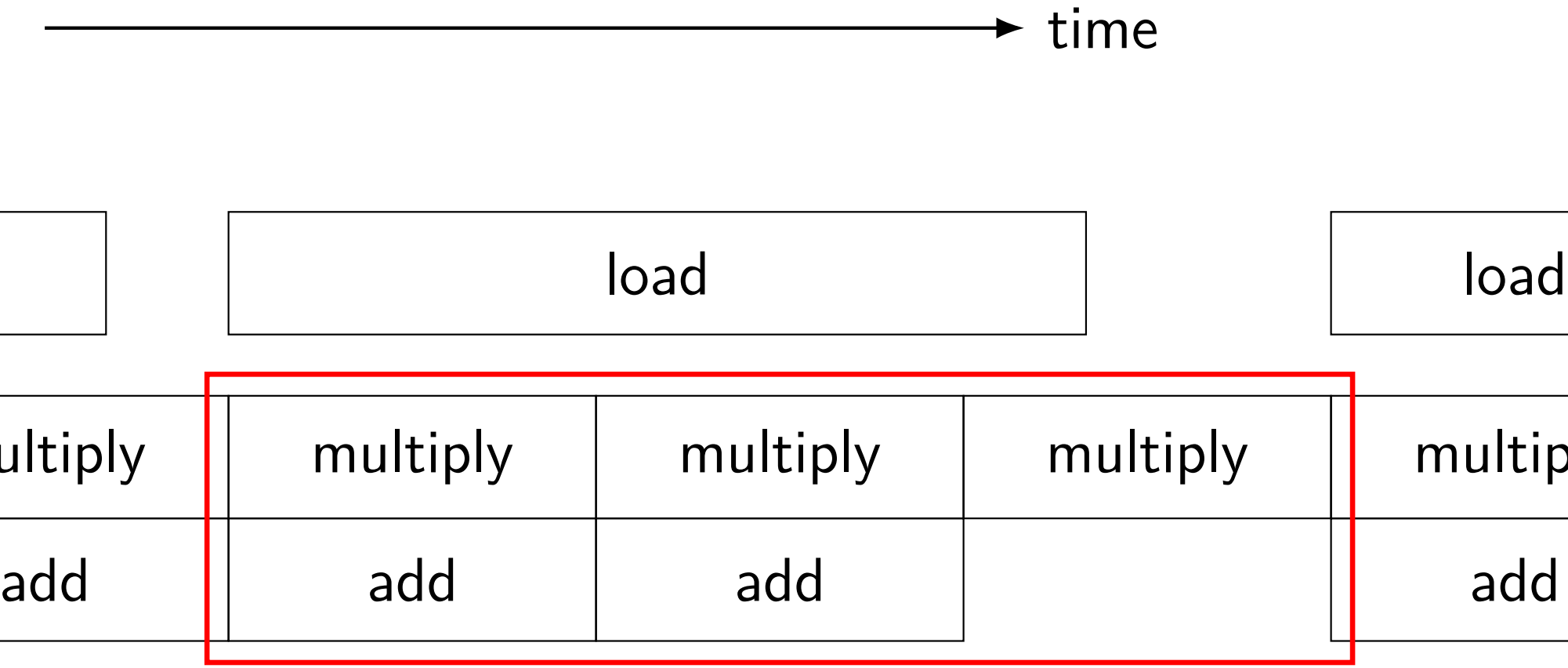
1: loading $B_{i,j}$ through $B_{i,j+7}$ with one instruction

2: doing adds and multiplies with less instructions

3: simplifying address computations

but... how can that make cache blocking better???

overlapping loads and arithmetic



speed of load might not matter if these are slower

register reuse

```
for (int k = 0; k < N; ++k)
  for (int i = 0; i < N; ++i)
    for (int j = 0; j < N; ++j)
      B[i*N+j] += A[i*N+k] * A[k*N+j];
// optimize into:
for (int k = 0; k < N; ++k)
  for (int i = 0; i < N; ++i) {
    float Aik = A[i*N+k]; // hopefully keep in reg
                          // faster than even cac
    for (int j = 0; j < N; ++j)
      B[i*N+j] += Aik * A[k*N+j];
  }
}
```

can compiler do this for us?

can compiler do register reuse?

Not easily — What if $A = B$?

```
for (int k = 0; k < N; ++k)
  for (int i = 0; i < N; ++i) {
    // want to preload A[i*N+k] here!
    for (int j = 0; j < N; ++j) {
      // but if A = B, modifying here!
      B[i*N+j] += A[i*N+k] * A[k*N+j];
    }
  }
}
```

Automatic register reuse

Compiler would need to generate overlap check:

```
if ((B > A + N * N || B < A) &&
    (B + N * N > A + N * N ||
     B + N * N < A)) {
  for (int k = 0; k < N; ++k) {
    for (int i = 0; i < N; ++i) {
      float Aik = A[i*N+k];
      for (int j = 0; j < N; ++j) {
        B[i*N+j] += Aik * A[k*N+j];
      }
    }
  }
} else { /* other version */ }
```

“register blocking”

```
for (int k = 0; k < N; ++k) {
  for (int i = 0; i < N; i += 2) {
    float Ai0k = A[(i+0)*N + k];
    float Ai1k = A[(i+1)*N + k];
    for (int j = 0; j < N; j += 2) {
      float Akj0 = A[k*N + j+0];
      float Akj1 = A[k*N + j+1];
      B[(i+0)*N + j+0] += Ai0k * Akj0;
      B[(i+1)*N + j+0] += Ai1k * Akj0;
      B[(i+0)*N + j+1] += Ai0k * Akj1;
      B[(i+1)*N + j+1] += Ai1k * Akj1;
    }
  }
}
```

cache blocking: summary

reorder calculation to reduce cache misses:

make explicit choice about what is in cache

perform calculations in cache-sized blocks

get more spatial and temporal locality

temporal locality — reuse values in many calculations

- before they are replaced in the cache

spatial locality — use adjacent values in calculations

- before cache block is replaced

avoiding conflict misses

problem — array is scattered throughout memory

observation: 32KB cache can store 32KB contiguous array

- contiguous array is split evenly among sets

solution: copy block into contiguous array

avoiding conflict misses (code)

```
process_block(ii, jj, kk) {
  float B_copy[I * J];
  /* pseudocode for loop to save space */
  for i = ii to ii + I, j = jj to jj + J:
    B_copy[i * J + j] = B[i * N + j];
  for i = ii to ii + I, j = jj to jj + J,
    B_copy[i * J + j] += A[k * N + j] * A
  for all i, j:
    B[i * N + j] = B_copy[i * J + j];
}
```

prefetching

processors detect sequential access patterns

e.g. accessing memory address 0, 8, 16, 24, ...?

processor will prefetch 32, 48, etc.

another way to take advantage of spatial locality

part of why miss rate is so low

matrix sum

```
int sum1(int matrix[4][8]) {
    int sum = 0;
    for (int i = 0; i < 4; ++i) {
        for (int j = 0; j < 8; ++j) {
            sum += matrix[i][j];
        }
    }
}
```

access pattern:

`matrix[0][0]`, `[0][1]`, `[0][2]`, …, `[1][0]` …

matrix sum: spatial locality

matrix in memory (4 bytes/row)

[0][0]	iter. 0
[0][1]	iter. 1
[0][2]	iter. 2
[0][3]	iter. 3
[0][4]	iter. 4
[0][5]	iter. 5
[0][6]	iter. 6
[0][7]	iter. 7
[1][0]	iter. 8
[1][1]	iter. 9
...	...

matrix sum: spatial locality

matrix in memory (4 bytes/row)

8-byte cache block?

[0][0]	iter. 0
[0][1]	iter. 1
[0][2]	iter. 2
[0][3]	iter. 3
[0][4]	iter. 4
[0][5]	iter. 5
[0][6]	iter. 6
[0][7]	iter. 7
[1][0]	iter. 8
[1][1]	iter. 9
...	...

matrix sum: spatial locality

matrix in memory (4 bytes/row)

8-byte cache block?

[0][0]	iter. 0	miss
[0][1]	iter. 1	hit (same block as before)
[0][2]	iter. 2	miss
[0][3]	iter. 3	hit (same block as before)
[0][4]	iter. 4	miss
[0][5]	iter. 5	hit
[0][6]	iter. 6	...
[0][7]	iter. 7	
[1][0]	iter. 8	
[1][1]	iter. 9	
...	...	

block size and spatial locality

larger blocks — exploit spatial locality

... but larger blocks means fewer blocks for same size

less good at exploiting temporal locality

alternate matrix sum

```
int sum2(int matrix[4][8]) {
    int sum = 0;
    // swapped loop order
    for (int j = 0; j < 8; ++j) {
        for (int i = 0; i < 4; ++i) {
            sum += matrix[i][j];
        }
    }
}
```

access pattern:

`matrix[0][0]`, `[1][0]`, `[2][0]`, …, `[0][1]`, …

matrix sum: bad spatial locality

matrix in memory (4 bytes/row)

[0][0]	iter. 0
[0][1]	iter. 4
[0][2]	iter. 8
[0][3]	iter. 12
[0][4]	iter. 16
[0][5]	iter. 20
[0][6]	iter. 24
[0][7]	iter. 28
[1][0]	iter. 1
[1][1]	iter. 5
...	...

matrix sum: bad spatial locality

matrix in memory (4 bytes/row)

8-byte cache block?

[0][0]	iter. 0
[0][1]	iter. 4
[0][2]	iter. 8
[0][3]	iter. 12
[0][4]	iter. 16
[0][5]	iter. 20
[0][6]	iter. 24
[0][7]	iter. 28
[1][0]	iter. 1
[1][1]	iter. 5
...	...

miss unless value not evicted for 4 iterations

cache organization and miss rate

depends on program; one example:

SPEC CPU2000 benchmarks, 64B block size

LRU replacement policies

data cache miss rates:

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

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

cache organization and miss rate

depends on program; one example:

SPEC CPU2000 benchmarks, 64B block size

LRU replacement policies

data cache miss rates:

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

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

is LRU always better?

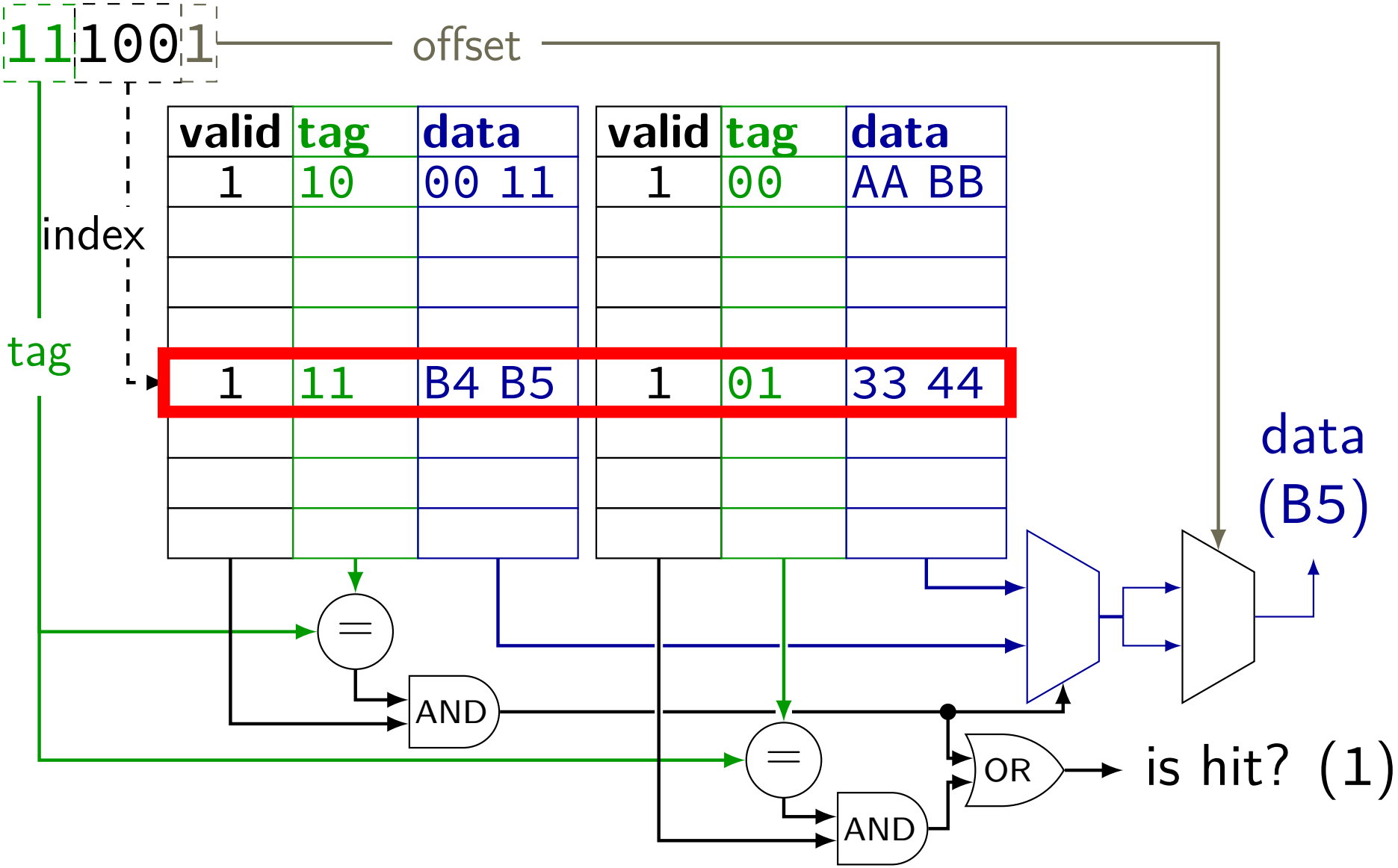
least recently used exploits temporal locality

making LRU look bad

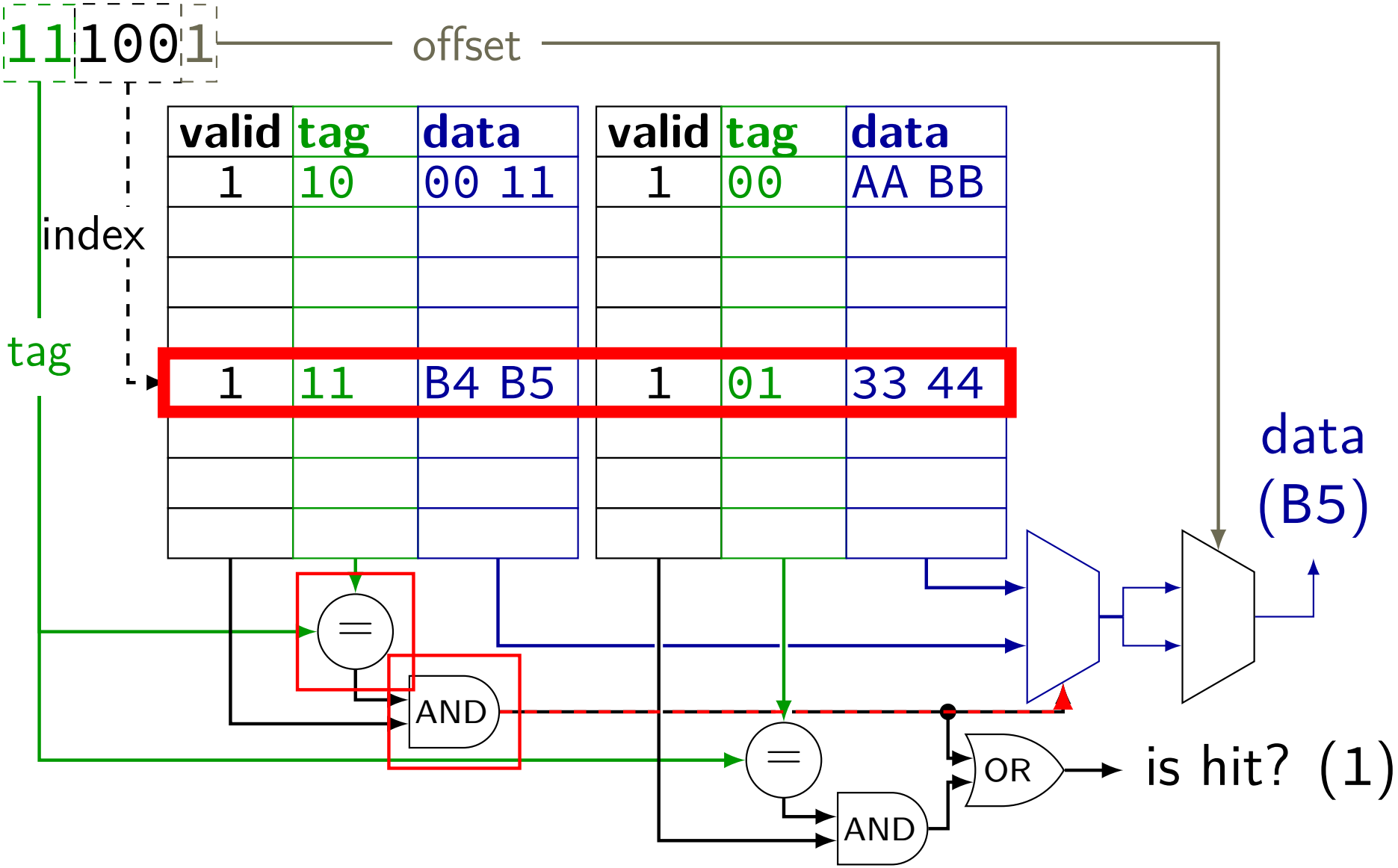
* = least recently used

	direct-mapped (2 sets)		fully-associative (1 set)	
read 0	miss:	mem[0]; —	miss:	mem[0], —*
read 1	miss:	mem[0]; mem[1]	miss:	mem[0]*, mem[1]
read 3	miss:	mem[0]; mem[3]	miss:	mem[3], mem[1]*
read 0	hit:	mem[0]; mem[3]	miss:	mem[3]*, mem[0]
read 2	miss:	mem[2]; mem[3]	miss:	mem[2], mem[0]*
read 3	hit:	mem[2]; mem[3]	miss:	mem[2]*, mem[3]
read 1	hit:	mem[2]; mem[1]	hit:	mem[1], mem[3]*
read 2	hit:	mem[2]; mem[1]	miss:	mem[1]*, mem[2]

cache operation (associative)



cache operation (associative)



cache operation (associative)