Cache Effect in Matrix Multiplication.
Tutorial for Assignment 1

See also the original slides:

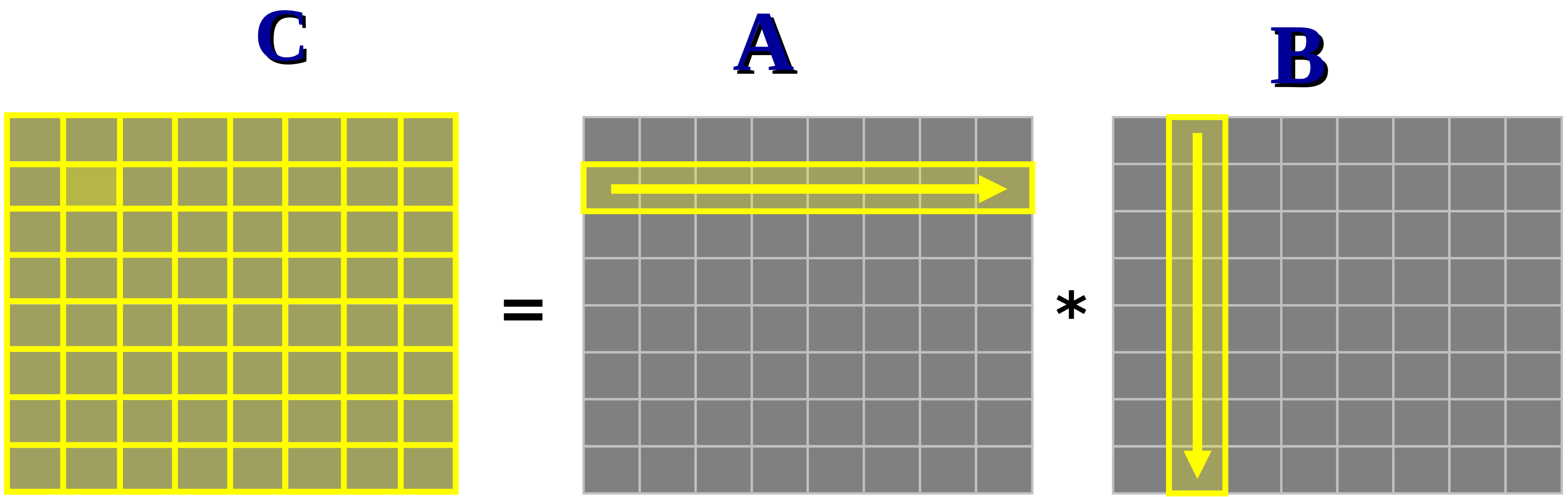
1.http://www.eecs.harvard.edu/~mdw/course/cs61/mediawiki/images/b/bb/Lectures-cacheperf.pdf

2.http://www.cs.tau.ac.il/~ozery/courses/soft-project09/presentations/assign2.ppt

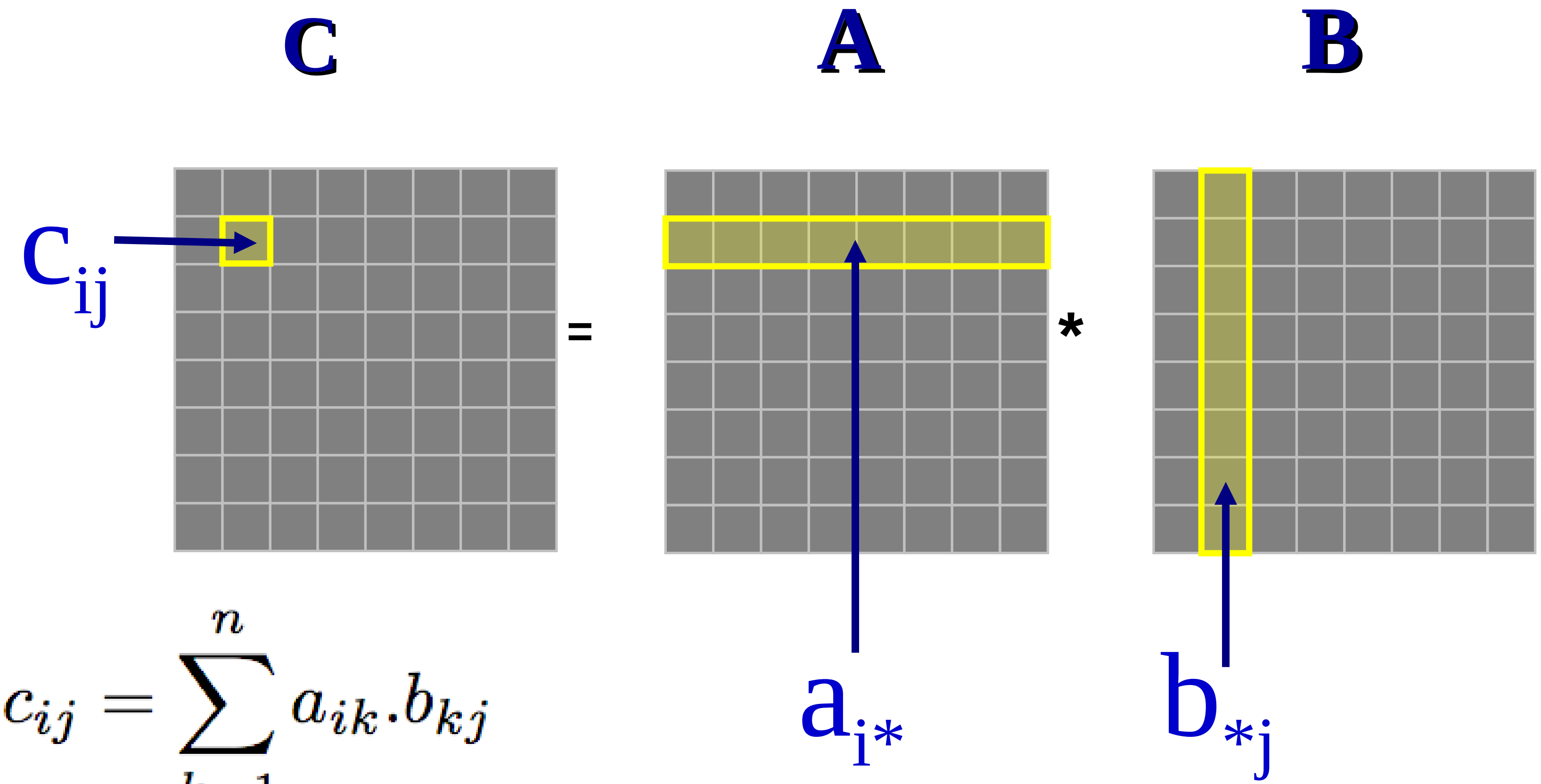
Multiplication of 2D Matrices

1. Simple algorithm
2. Time and Performance Measurement
3. Simple Code Improvements

Matrix Multiplication



Matrix Multiplication



$$c_{ij} = \sum_{k=1}^{n} a_{ik}.b_{kj}$$

The simplest algorithm

Assumption: the matrices are stored as 2-D NxN arrays

```
for (i=0;i<N;i++)
    for (j=0;j<N;j++)
      for (k=0;k<N;k++)
              c[i][j] += a[i][k] * b[k][j];
```

Advantage: code simplicity

Disadvantage: performance (?)

First Improvement

```
for (i=0;i<N;i++)
    for (j=0;j<N;j++)
       for (k=0;k<N;k++)
                c[i][j] += a[i][k] * b[k][j
];
```

c[i][j]:

- Requires address (pointer) computations
- is constant in the k-loop

First Performance Improvement

```
for (i=0;i<N;i++){
    for (j=0;j<N;j++) {
        int sum = 0;
        for (k=0;k<N;k++) {
            sum += a[i][k] * b[k][j];
        }
            c[i][j] = sum;
    }
}
```

Benchmarking

```
#include <sys/time.h>
struct timeval tv_start, tv_end;
struct timezone tz;
gettimeofday(&tv_start, &tz);
<code to benchmark>
gettimeofday(&tv_end, &tz);
double elapsed = (double) (tv_end.tv_sec-
tv_start.tv_sec) + (double) (tv_end.tv_usec-
tv_start.tv_usec) * 1.e-6;
```

Cache Memory

Cache memory (CPU cache)

- A temporary storage area where frequently accessed data can be stored for rapid access.
- Once the data is stored in the cache, future use can be made by accessing the cached copy rather than re-fetching the original data, so that the average access time is lower.

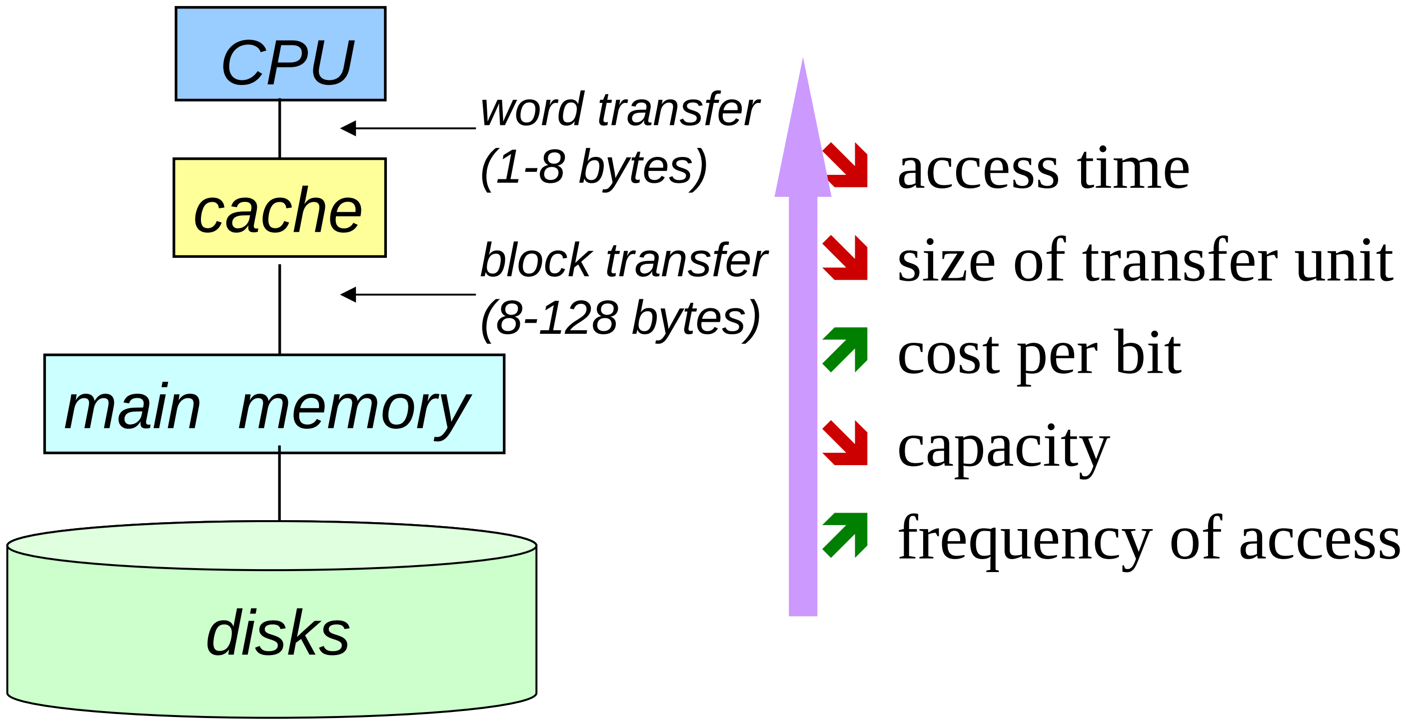
Memory Hierarchy

CPU Registers
500 Bytes
0.25 ns

Cache
16K-1M Bytes
1 ns

Main Memory
64M-2G Bytes
100ns

Disk
100 G Bytes
5 ms



- The memory cache is closer to the processor than the main memory.
- It is smaller, faster and more expensive than the main memory.
- Transfer between caches and main memory is performed in units called cache blocks/lines.

Types of Cache Misses

1. ***Compulsory misses:*** caused by first access to blocks that have never been in cache (also known as *cold-start misses*)

2. ***Capacity misses***: when cache cannot contain all the blocks needed during execution of a program. Occur because of blocks being replaced and later retrieved when accessed.

3. ***Conflict misses***: when multiple blocks compete for the same set.

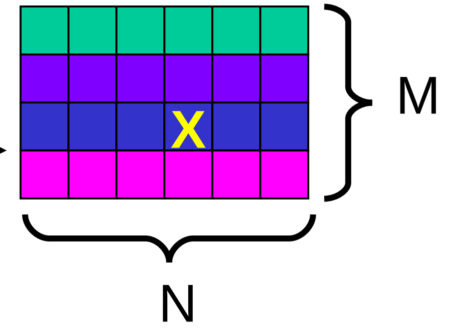
Main Cache Principles

- Temporal Locality (Locality in Time): If an item is referenced, it will tend to be referenced again soon.
- Spatial Locality (Locality in Space): If an item is referenced, close items (by address) tend to be referenced soon.
- Miss Rate : Fraction of memory references not found in cache (# misses / # references)
- Hit Rate: Time to deliver a line in the cache to the processor (includes time to determine whether the line is in the cache)

Improving Spatial Locality:

Loop Reordering for Matrices Allocated by Row

Allocation by rows

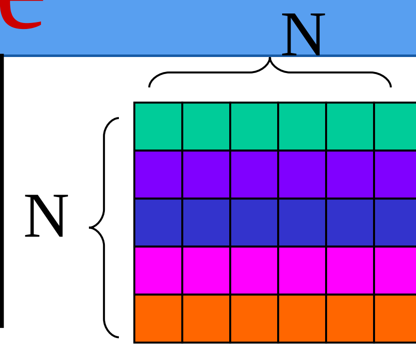


Writing Cache Friendly Code

<u>Assumptions</u>: **1)** block-size= β words (word = int)

2) N is divided by β

3) The cache cannot hold a complete row / column




```
int sumarrayrows(int a[N][N])
{
  int i, j, sum = 0;
  for (i = 0; i < N; i++)
    for (j = 0; j < N; j++)
      sum += a[i][j];
    return sum;
}
```

Accesses successive elements:
a[0][0],...,a[0][N-1],a[1][0],..

Miss rate = 1/ β

```
int sumarraycols(int a[N][N])
{
  int i, j, sum = 0;
  for (j = 0; j < N; j++)
    for (i = 0; i < N; i++)
      sum += a[i][j];
  return sum;
}
```

no spatial locality!

Accesses distant elements:
a[0][0],...,a[N-1][0],a[0][1],...

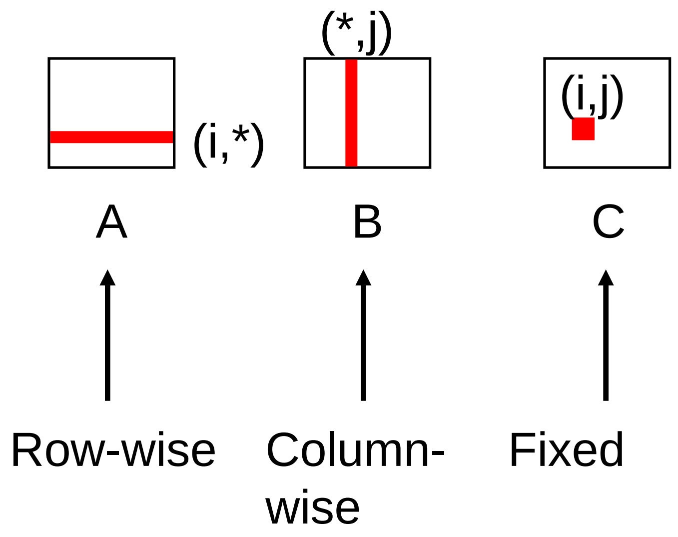
Miss rate = 1

Matrix Multiplication (ijk)

every element of A and B is accessed N times

```
/* ijk */
for (i=0; i<n; i++)  {
  for (j=0; j<n; j++) {
    int sum = 0;
    for (k=0; k<n; k++)
      sum += a[i][k] * b[k][j];
    c[i][j] = sum;
  }
}
```

Inner loop:



- Misses per Inner Loop Iteration (i=j=0):

A	B	C
$1/\beta$	1.0	1.0

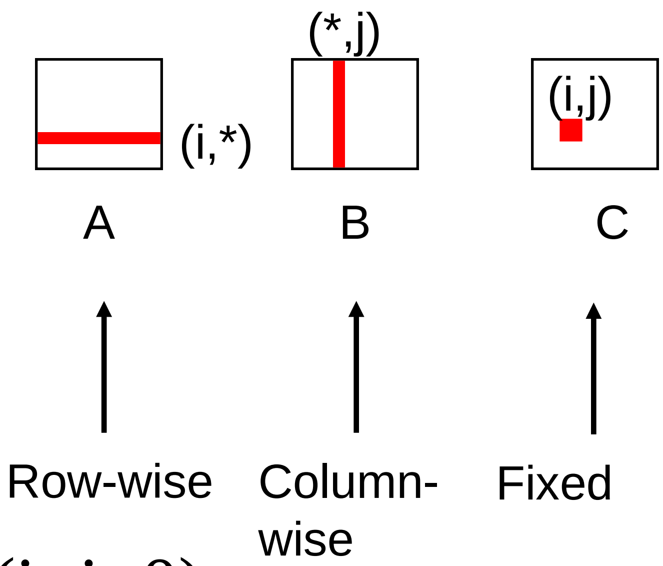
*what happens when i=0 and j=1?

Matrix Multiplication (jik)

every element of A and B is accessed N times

```
/* jik */
for (j=0; j<n; j++) {
  for (i=0; i<n; i++) {
    int sum = 0;
    for (k=0; k<n; k++)
      sum += a[i][k] * b[k][j];
    c[i][j] = sum;
  }
}
```

Inner loop:



- Misses per Inner Loop Iteration (i=j=0):

A	B	C
1/ β	1.0	1.0

*what happens when j=0 and i=1?

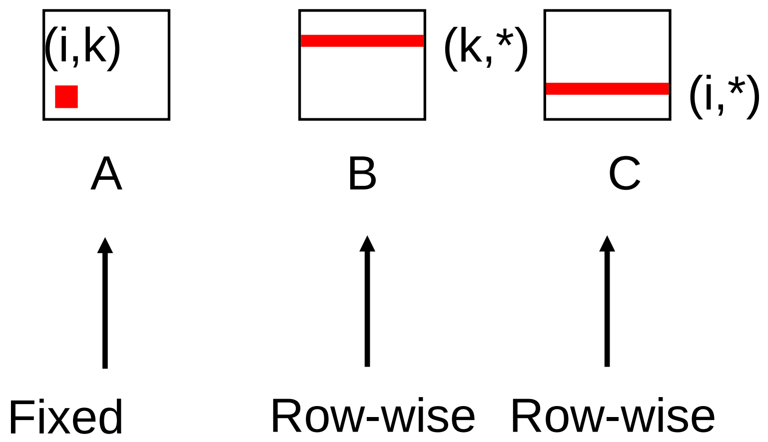
Matrix Multiplication (kij)

every element of B and C is accessed N times

```
/* kij */
for (k=0; k<n; k++) {
  for (i=0; i<n; i++) {
    int x = a[i][k];
    for (j=0; j<n; j++)
      c[i][j] += x * b[k][j];

  }
}
```

Inner loop:



- <u>Misses per Inner Loop Iteration (i=j=0):</u>

<u>A</u>	<u>B</u>	<u>C</u>
1.0	1/β	1/β

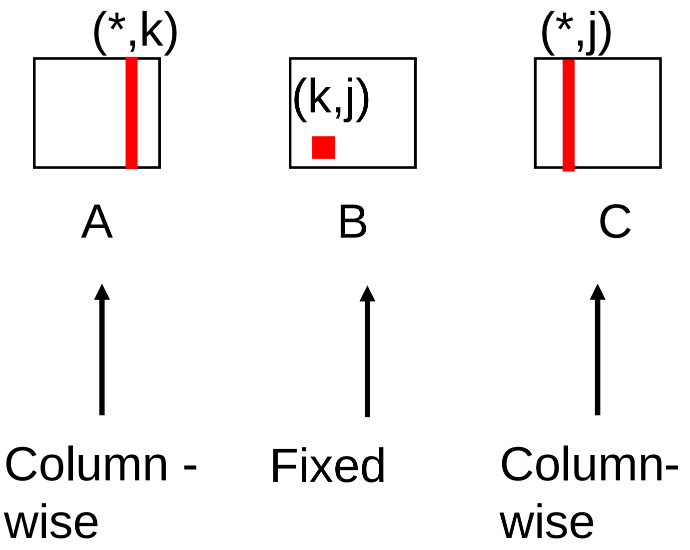
*what happens when k=0 and i=1?

Matrix Multiplication (jki)

every element of A and C is accessed N times

```
/* jki */
for (j=0; j<n; j++) {
  for (k=0; k<n; k++) {
    int x = b[k][j];
    for (i=0; i<n; i++)
      c[i][j] += a[i][k] * x;
  }
}
```

Inner loop:



- <u>Misses per Inner Loop Iteration (i=j=0):</u>

<u>A</u>	<u>B</u>	<u>C</u>
1.0	1.0	1.0

*what happens when j=0 and k=1?

Summary: misses ratios for the first iteration of the inner loop

ijk (& jik): $(\beta+1)/(2\beta) > 1/2$ kij: $1/\beta$ jki: 1

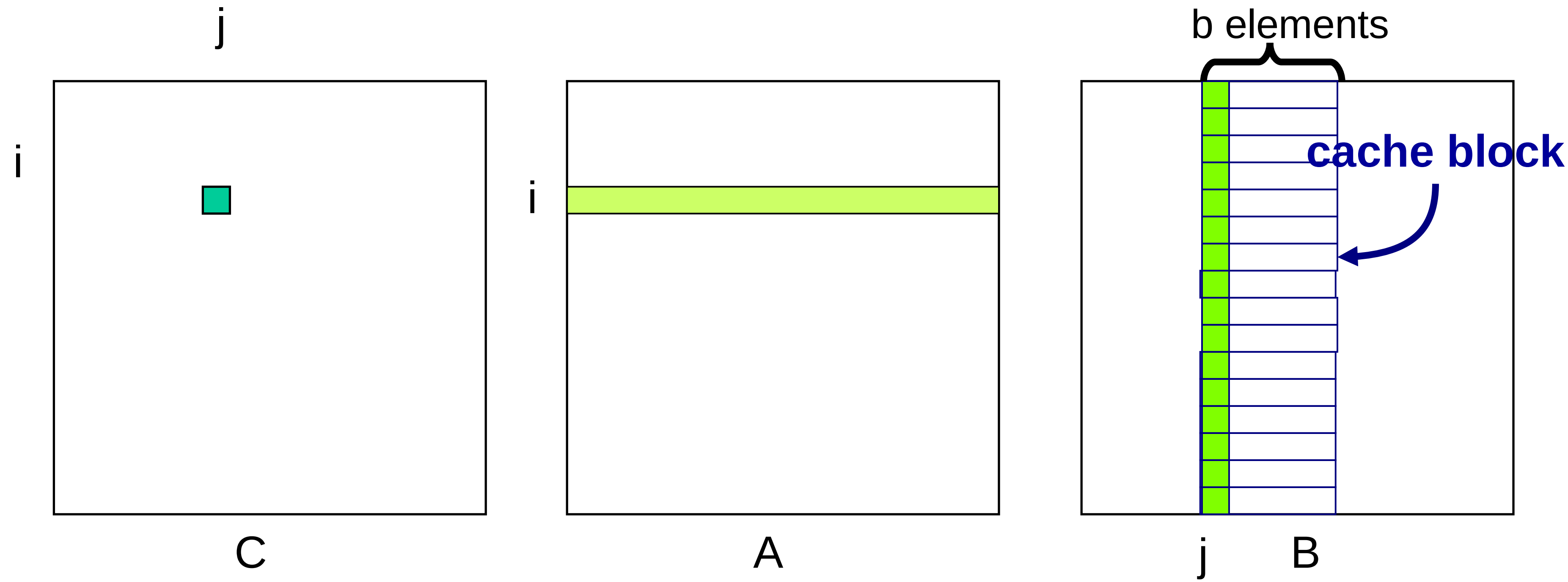
```
for (i=0; i<n; i++)  {
   for (j=0; j<n; j++) {
       int sum = 0;
       for (k=0; k<n; k++)
           sum += a[i][k] * b[k][j];
       c[i][j] = sum;
   }
}
```

```
for (k=0; k<n; k++) {
    for (i=0; i<n; i++) {
        int x = a[i][k];
        for (j=0; j<n; j++)
            c[i][j] += x * b[k][j];
    }
}
```

```
for (j=0; j<n; j++) {
    for (k=0; k<n; k++) {
        int x = b[k][j];
        for (i=0; i<n; i++)
            c[i][j] += a[i][k] * x;
    }
}
```

Improving Temporal Locality: Blocked Matrix Multiplication

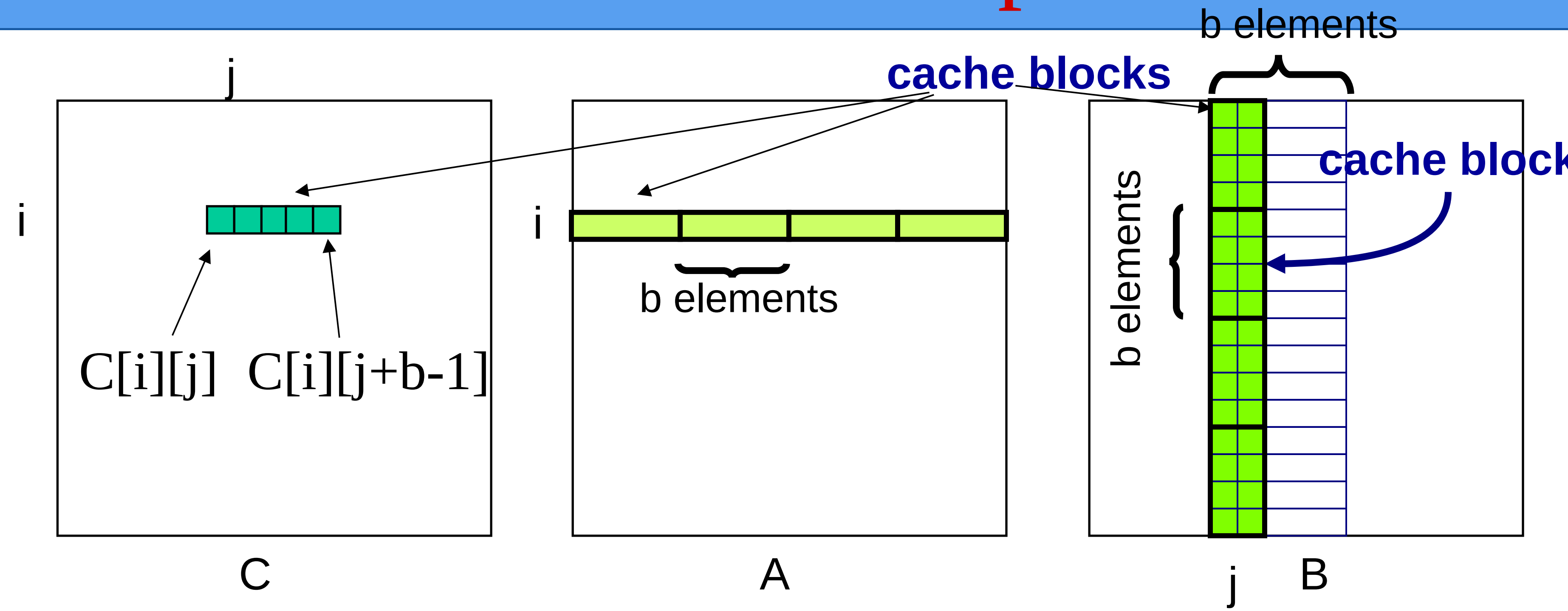
“Blocked” Matrix Multiplication



$$c_{ij} = \sum_{k=1}^{N} a_{ik} b_{kj}$$

Key idea: reuse the other elements in each cache block as much as possible

“Blocked” Matrix Multiplication

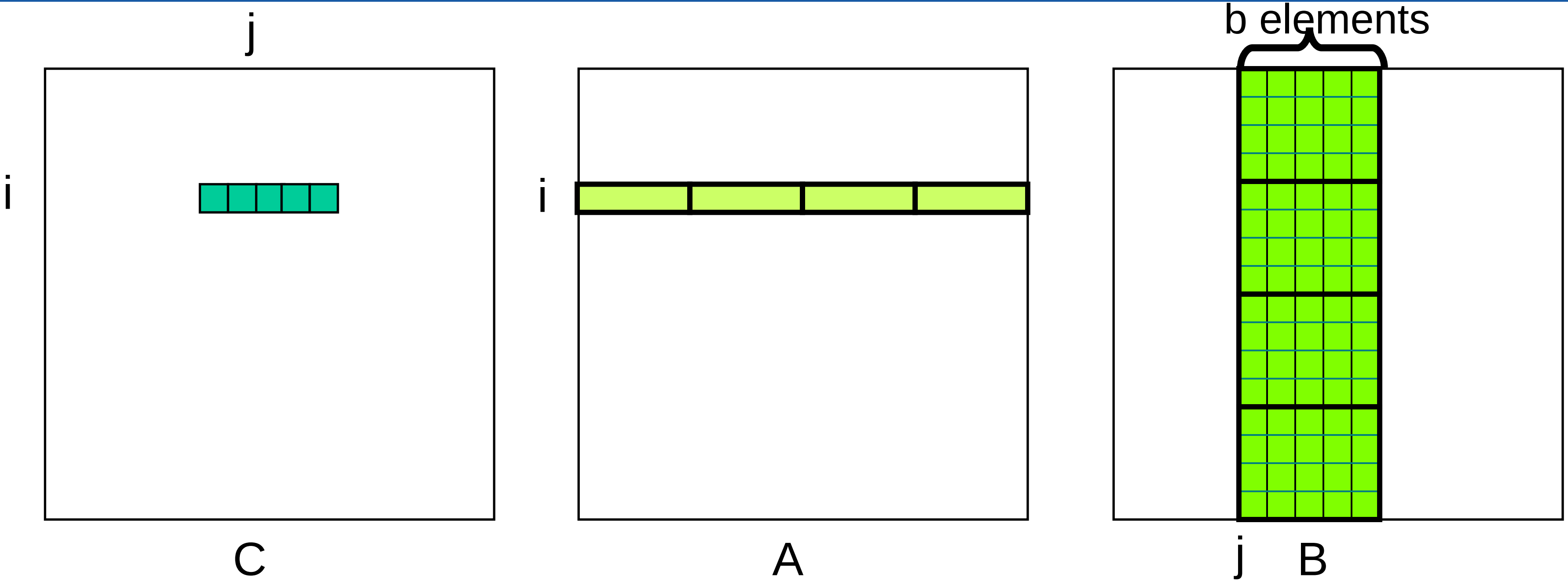


The blocks loaded for the computation of C[i][j] are appropriate for the computation of C[i,j+1]...C[i,j+ b-1]

- compute the first b terms of C[i][j],...,C[i][j+b -1]
- compute the next b terms of C[i][j],...,C[i][j+b -1]

.....

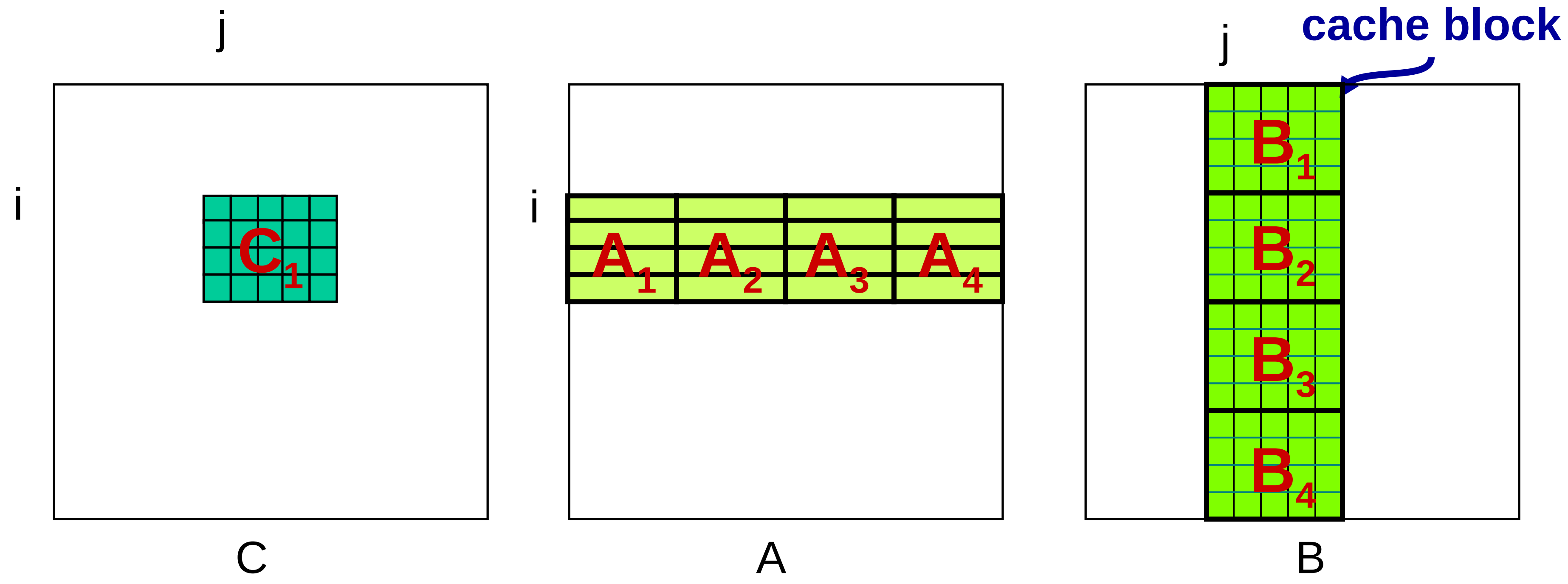
“Blocked” Matrix Multiplication



Next improvement:

Reuse the loaded blocks of B for the computation of next (b -1) subrows.

“Blocked” Matrix Multiplication



Order of the operations:

Compute the first b terms of C1 $(=A_1*B_1)$

Compute the next b terms of C1 $(=A_2*B_2)$

. . .

Compute the last b terms of C1 $(=A_4*B_4)$

“Blocked” Matrix Multiplication

C_{11}	C_{12}	C_{13}	C_{14}
C_{21}	C_{22}	C_{23}	C_{24}
C_{31}	C_{32}	C_{43}	C_{34}
C_{41}	C_{42}	C_{43}	C_{44}

A_{11}	A_{12}	A_{13}	A_{14}
A_{21}	A_{22}	A_{23}	A_{24}
A_{31}	A_{32}	A_{33}	A_{34}
A_{41}	A_{42}	A_{43}	A_{144}

B_{11}	B_{12}	B_{13}	B_{14}
B_{21}	B_{22}	B_{23}	B_{24}
B_{32}	B_{32}	B_{33}	B_{34}
B_{41}	B_{42}	B_{43}	B_{44}

N = 4 * b

$$C_{22} = A_{21}B_{12} + A_{22}B_{22} + A_{23}B_{32} + A_{24}B_{42} =$$
$$\sum_k A_{2k} * B_{k2}$$

Main Point: each multiplication operates on small “block” matrices, whose size may be chosen so that they fit in the cache.

Blocked Algorithm

- The blocked version of the i-j-k algorithm is written simply as

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

b x b matrix addition

b x b matrix multiplication

  - b = block (sub-matrix) size (Assume b divides N)
  - X[i][j] = a sub-matrix of X, defined by block row i and block column j

Maximum Block Size

- The blocking optimization works only if the blocks fit in cache.
- That is, 3 blocks of size b x b must fit in memory (for A, B, and C)
- M = size of cache (in elements/words)
- We must have: $3b^2 \approx M$, or $b \approx \sqrt{(M/3)}$
- Lower bound = $(b^2/\beta)\ (2(n/b)^3 + (n/b)^2) = (1/\beta)(2n^3/b + n^2) = \Theta\ (n^3/(b\beta)) = \Theta(n^3/(\beta\sqrt{M}))$
- Therefore, the ratio of cache misses ijk-blocked vs. ijk-unblocked: $1:\sqrt{M}$