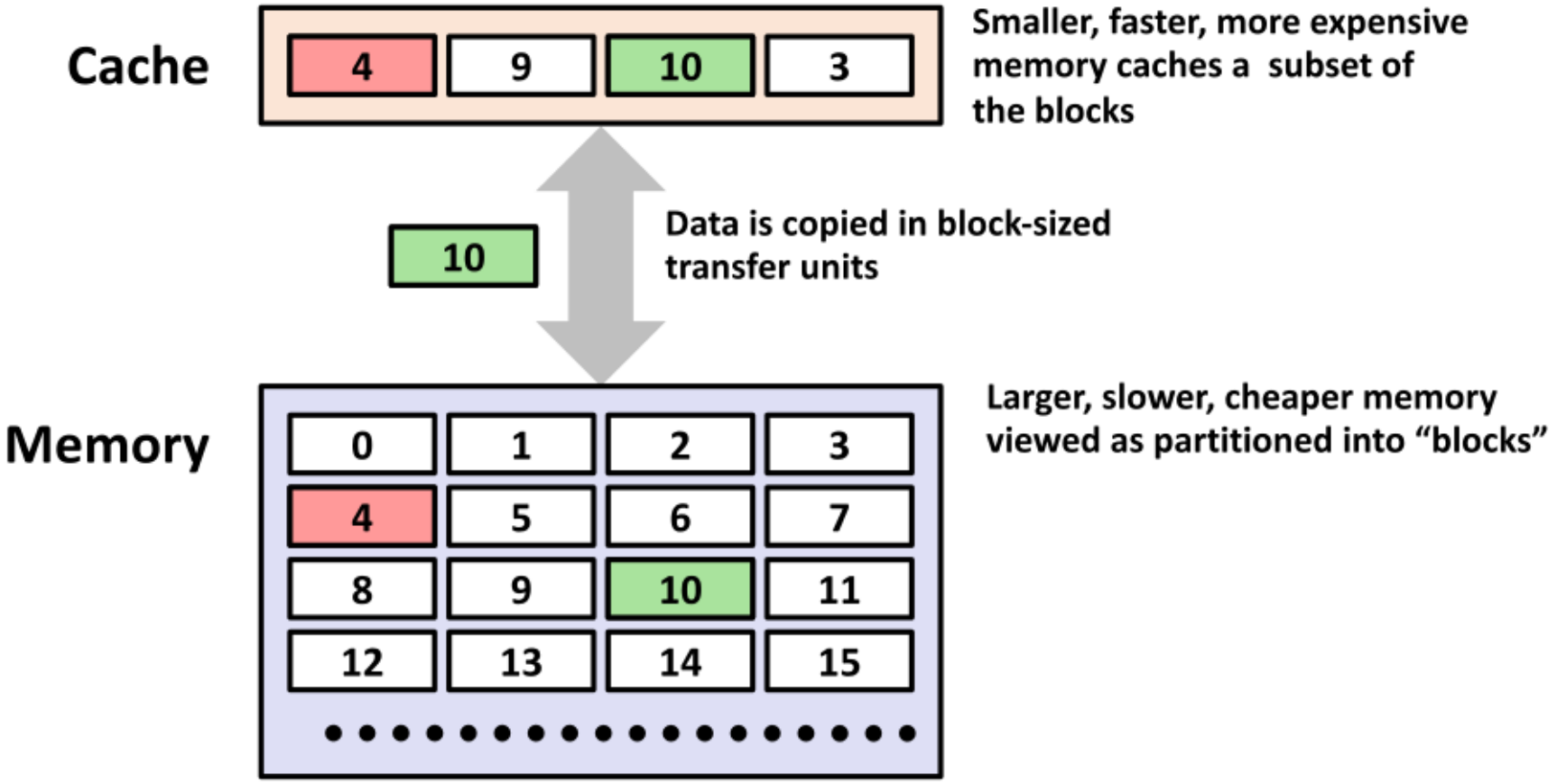
Cache Performance

General Cache Concept



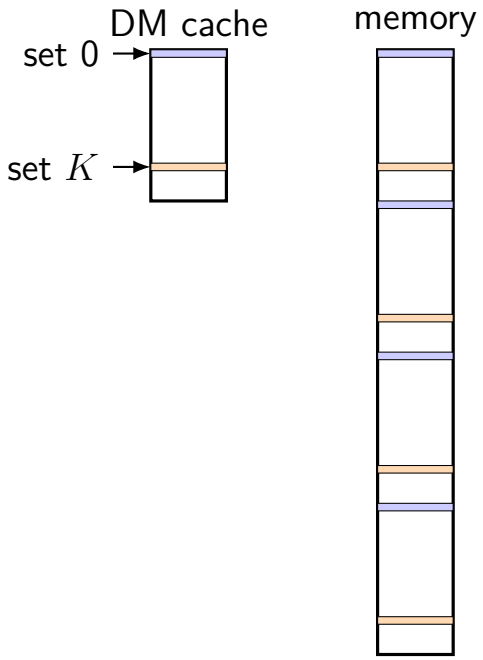
arrays and cache misses (2)

```
int array[1024]; // 4KB array
int even_sum = 0, odd_sum = 0;
for (int i = 0; i < 1024; i += 2)
    even_sum += array[i + 0];
for (int i = 0; i < 1024; i += 2)
    odd_sum +=  array[i + 1];
```

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

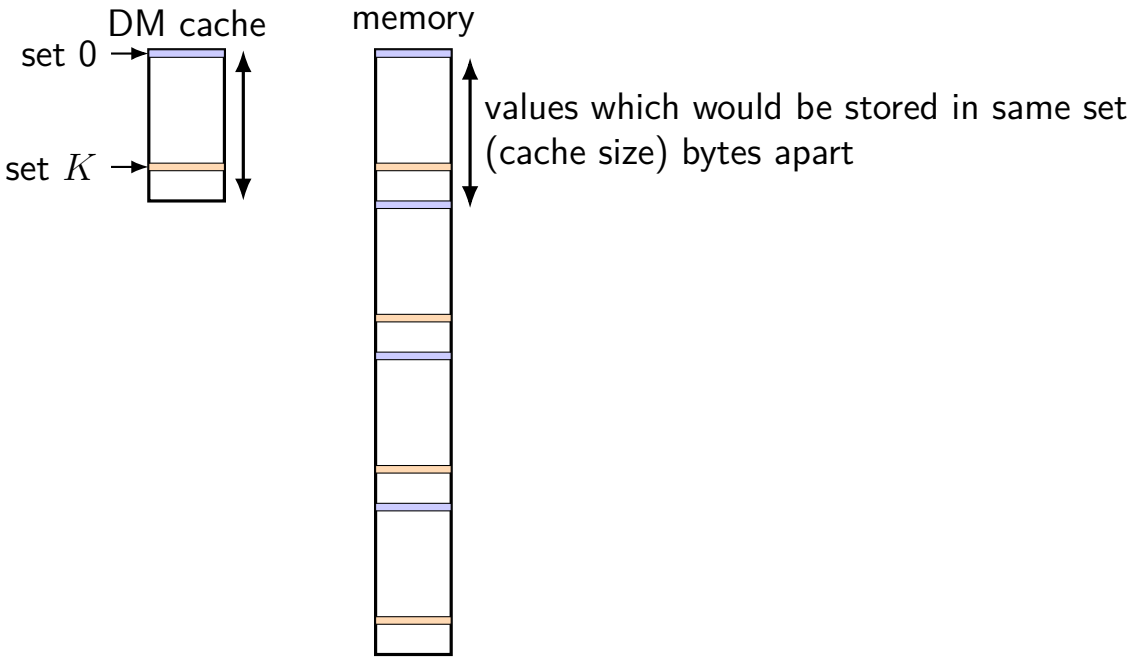
How many *data cache misses* on a 2KB direct-mapped cache with 16B cache blocks? Would a set-associtiave cache be better?

mapping of sets to memory (direct-mapped)



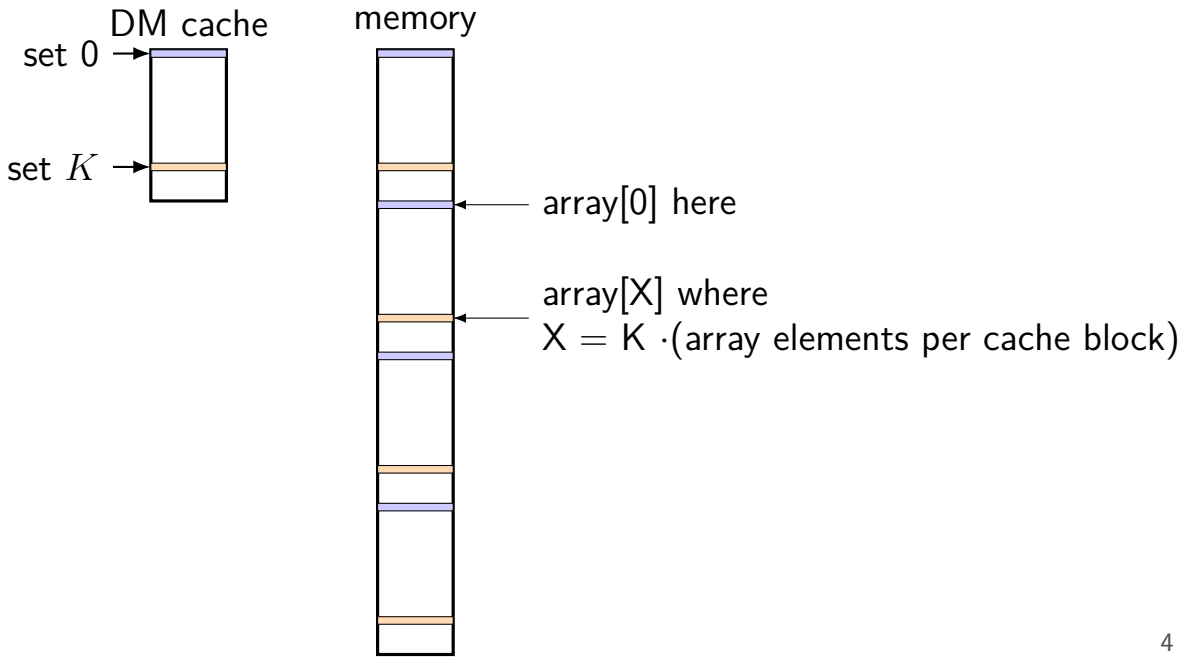
4

mapping of sets to memory (direct-mapped)



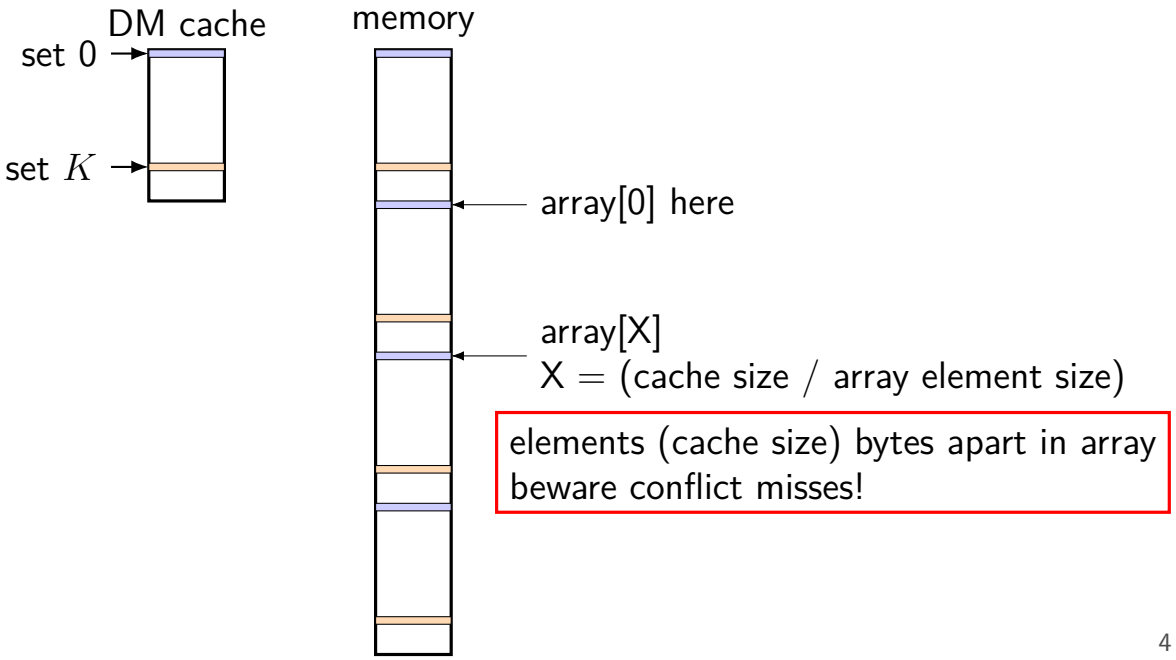
4

mapping of sets to memory (direct-mapped)



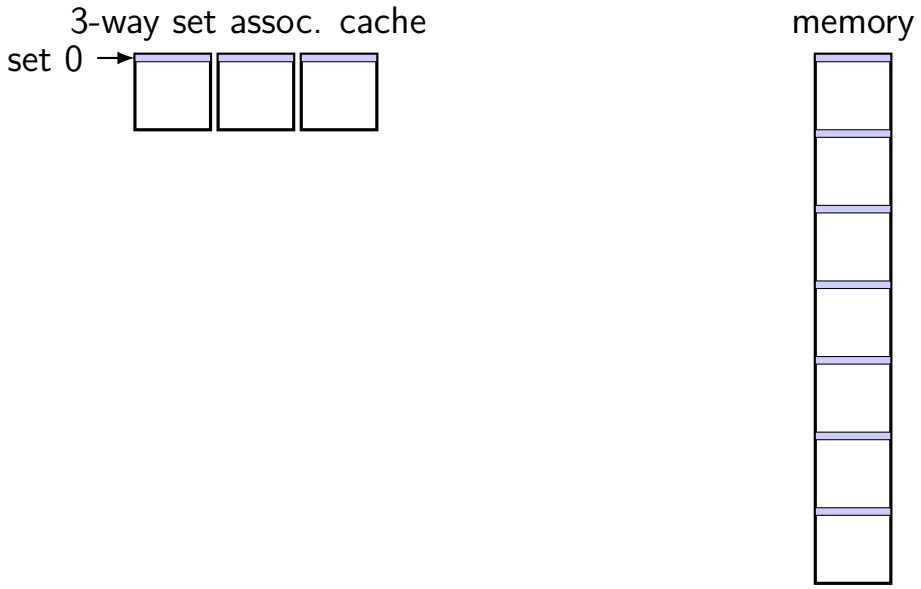
4

mapping of sets to memory (direct-mapped)



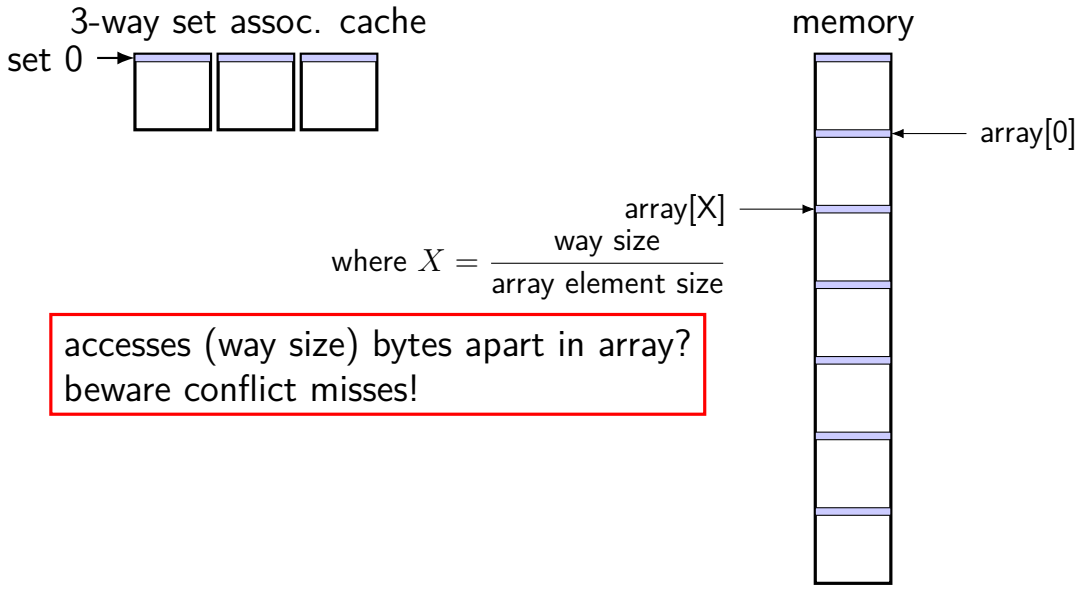
4

mapping of sets to memory (3-way)



5

mapping of sets to memory (3-way)



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

5

C and cache misses (3)

```
typedef struct {
    int a_value, b_value;
    int boring_values[126];
} item;
item items[8]; // 4 KB array
int a_sum = 0, b_sum = 0;
for (int i = 0; i < 8; ++i)
    a_sum += items[i].a_value;
for (int i = 0; i < 8; ++i)
    b_sum += items[i].b_value;
```

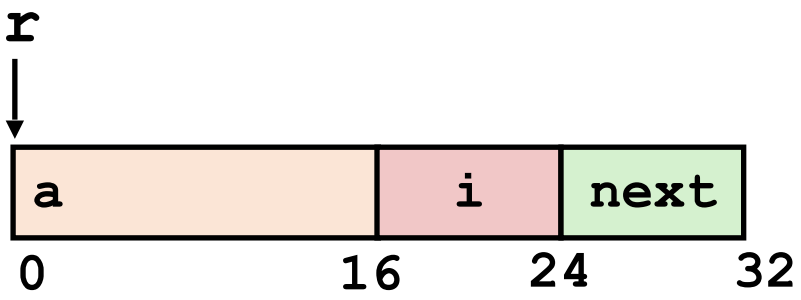
Assume everything but `items` is kept in registers (and the compiler does not do anything funny).

How many *data cache misses* on a 2KB direct-mapped cache with 16B cache blocks?

6

Structure Representation

```
struct rec {
    int a[4];
    size_t i;
    struct rec *next;
};
```



C and cache misses (3)

```
typedef struct {
    int a_value, b_value;
    int boring_values[126];
} item;
item items[8]; // 4 KB array
int a_sum = 0, b_sum = 0;
for (int i = 0; i < 8; ++i)
    a_sum += items[i].a_value;
for (int i = 0; i < 8; ++i)
    b_sum += items[i].b_value;
```

I[0].A	I[0].B	I[0].BV[0]	I[0].B[1]
I[1].A	I[1].B	I[1].BV[0]	I[1].B[1]

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

How many *data cache misses* on a 2KB direct-mapped cache with 16B cache blocks?

6

I[0].A	I[0].B	I[0].BV[0]	I[0].B[1]

I[1].A	I[1].B	I[1].BV[0]	I[1].B[1]

I[2].A	I[2].B	I[2].BV[0]	I[2].B[1]

I[3].A	I[3].B	I[3].BV[0]	I[3].B[1]

Each block associated the first half of the array has a unique spot in memory

2^9

Cache Optimization Techniques

```
for (j = 0; j < 3: j = j+1){
          for( i = 0; i < 3; i = i + 1){
                    x[i][j] = 2*x[i][j];
          }
}
```

```
for (i = 0; i < 3: i = i+1){
          for( j = 0; j < 3; j = j + 1){
                    x[i][j] = 2*x[i][j];
          }
}
```

Inner loop analysis

These two loops compute the same result

Array in row major order

X[0][0]	X[0][1]	X[0][2]
X[1][0]	X[1][1]	X[1][2]
X[2][0]	X[2][1]	X[2][2]

0x0 – 0x3	0x4 - 0x7	0x8-0x11	0x12-0x15	0x16 - 0x19	0x20-0x23

X[0][0]	X[0][1]	X[0][2]	X[1][0]	X[1][1]	X[1][2]	X[2][0]	X[2][1]	X[2][2]

Cache Optimization Techniques

```
for (j = 0; j < 3: j = j+1){
          for( i = 0; i < 3; i = i + 1){
                    x[i][j] = 2*x[i][j];
          }
}
```

```
for (i = 0; i < 3: i = i+1){
          for( j = 0; j < 3; j = j + 1){
                    x[i][j] = 2*x[i][j];
          }
}
```

These two loops compute the same result

Array in row major order

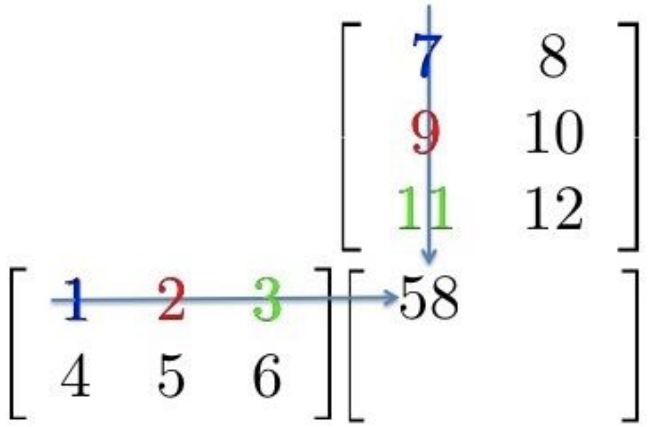
X[0][0]	X[0][1]	X[0][2]
X[1][0]	X[1][1]	X[1][2]
X[2][0]	X[2][1]	X[2][2]

```
int *x = malloc(N*N);
for (i = 0; i < 3: i = i+1){
          for( j = 0; j < 3; j = j + 1){
                    x[i*N +j] = 2*x[i*N + j];
          }
}
```

0x0 – 0x3	0x4 - 0x7	0x8-0x11	0x12-0x15	0x16 - 0x19	0x20-0x23

X[0][0]	X[0][1]	X[0][2]	X[1][0]	X[1][1]	X[1][2]	X[2][0]	X[2][1]	X[2][2]

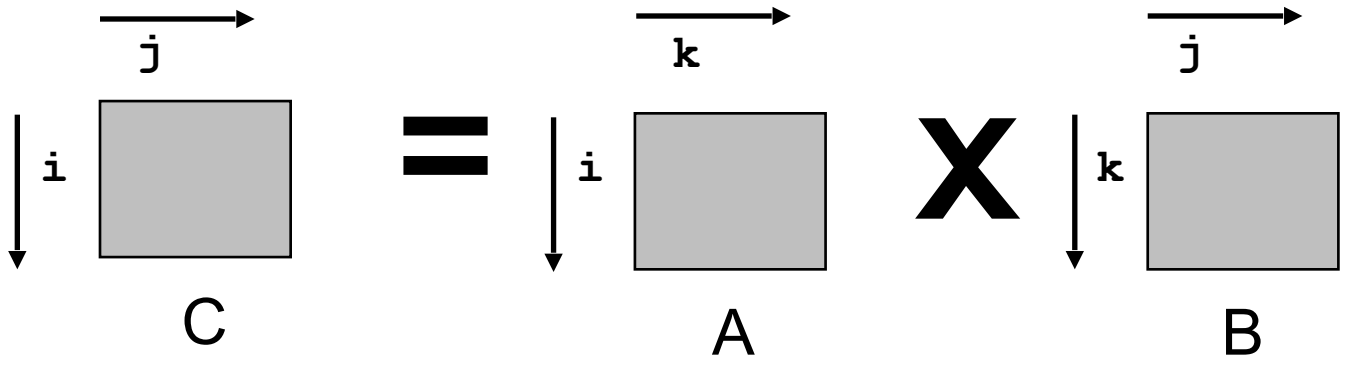
Matrix Multiplication Refresher



$$1 \cdot 7 + 2 \cdot 9 + 3 \cdot 11 = 58$$

Miss Rate Analysis for Matrix Multiply

- Assume:
  - Block size = 32B (big enough for four doubles)
  - Matrix dimension (N) is very large
  - Cache is not even big enough to hold multiple rows
- Analysis Method:
  - Look at access pattern of inner loop



Layout of C Arrays in Memory (review)

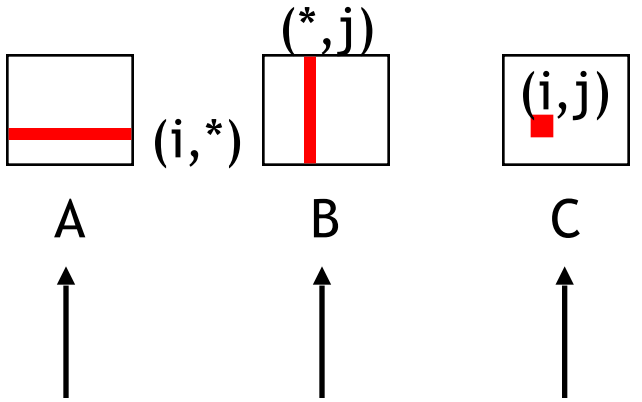
- C arrays allocated in row-major order
  - each row in contiguous memory locations
- Stepping through columns in one row:
  - `for (i = 0; i < N; i++)`
    `sum += a[0][i];`
  - accesses successive elements
  - if block size (B) > sizeof(a_{ij}) bytes, exploit spatial locality
    - miss rate = sizeof(a_{ij}) / B
- Stepping through rows in one column:
  - `for (i = 0; i < n; i++)`
    `sum += a[i][0];`
  - accesses distant elements
  - no spatial locality!
    - miss rate = 1 (i.e. 100%)

Matrix Multiplication (ijk)

```
/* ijk */
for (i=0; i<n; i++)  {
  for (j=0; j<n; j++) {
    sum = 0.0;
    for (k=0; k<n; k++)
      sum += a[i][k] * b[k][j];
    c[i][j] = sum;
  }
}
```

matmult/mm.c

Inner loop:



Misses per inner loop iteration:

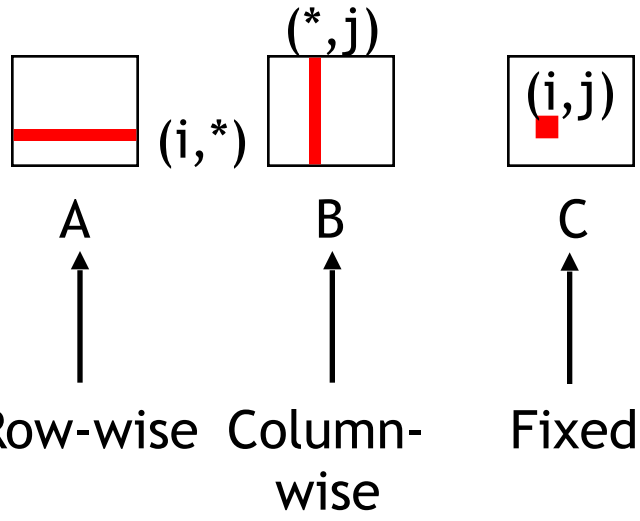
A	B	C
0.25	1.0.	0.0

Matrix Multiplication (jik)

```
/* jik */
for (j=0; j<n; j++) {
  for (i=0; i<n; i++) {
    sum = 0.0;
    for (k=0; k<n; k++)
      sum += a[i][k] * b[k][j];
    c[i][j] = sum
  }
}
```

matmult/mm.c

Inner loop:



Misses per inner loop iteration:

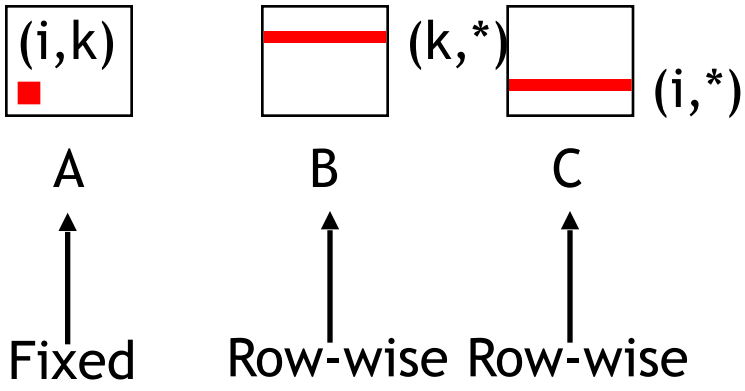
A	B	C
0.25	1.0	0.0

Matrix Multiplication (kij)

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

matmult/mm.c

Inner loop:



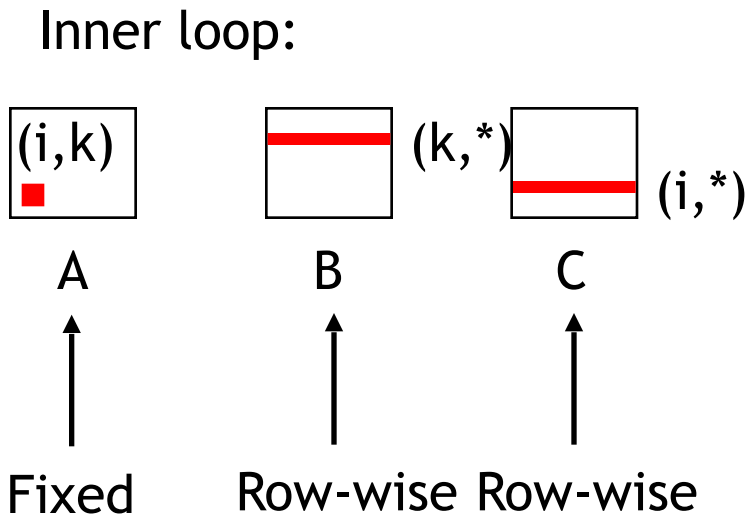
Misses per inner loop iteration:

A	B	C
0.0	0.25	0.25

Matrix Multiplication (ikj)

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

matmult/mm.c



Misses per inner loop iteration:

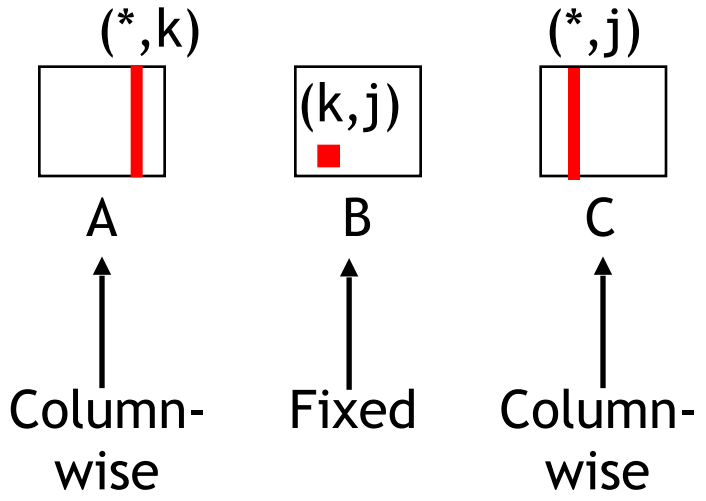
A	B	C
0.0	0.25	0.25

Matrix Multiplication (jki)

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

matmult/mm.c

Inner loop:



Misses per inner loop iteration:

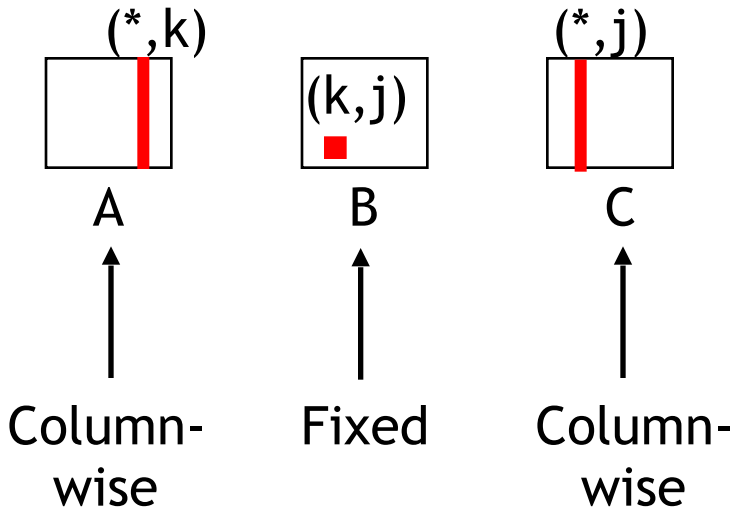
A	B	C
1.0	0.0	1.0

Matrix Multiplication (kji)

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

matmult/mm.c

Inner loop:



Misses per inner loop iteration:

A	B	C
1.0	0.0	1.0

Summary of Matrix Multiplication

```
for (i=0; i<n; i++) {
  for (j=0; j<n; j++) {
   sum = 0.0;
   for (k=0; k<n; k++) {
     sum += a[i][k] * b[k][j];}
   c[i][j] = sum;
 }
}
```

ijk (& jik):
- 2 loads, 0 stores
- misses/iter = **1.25**

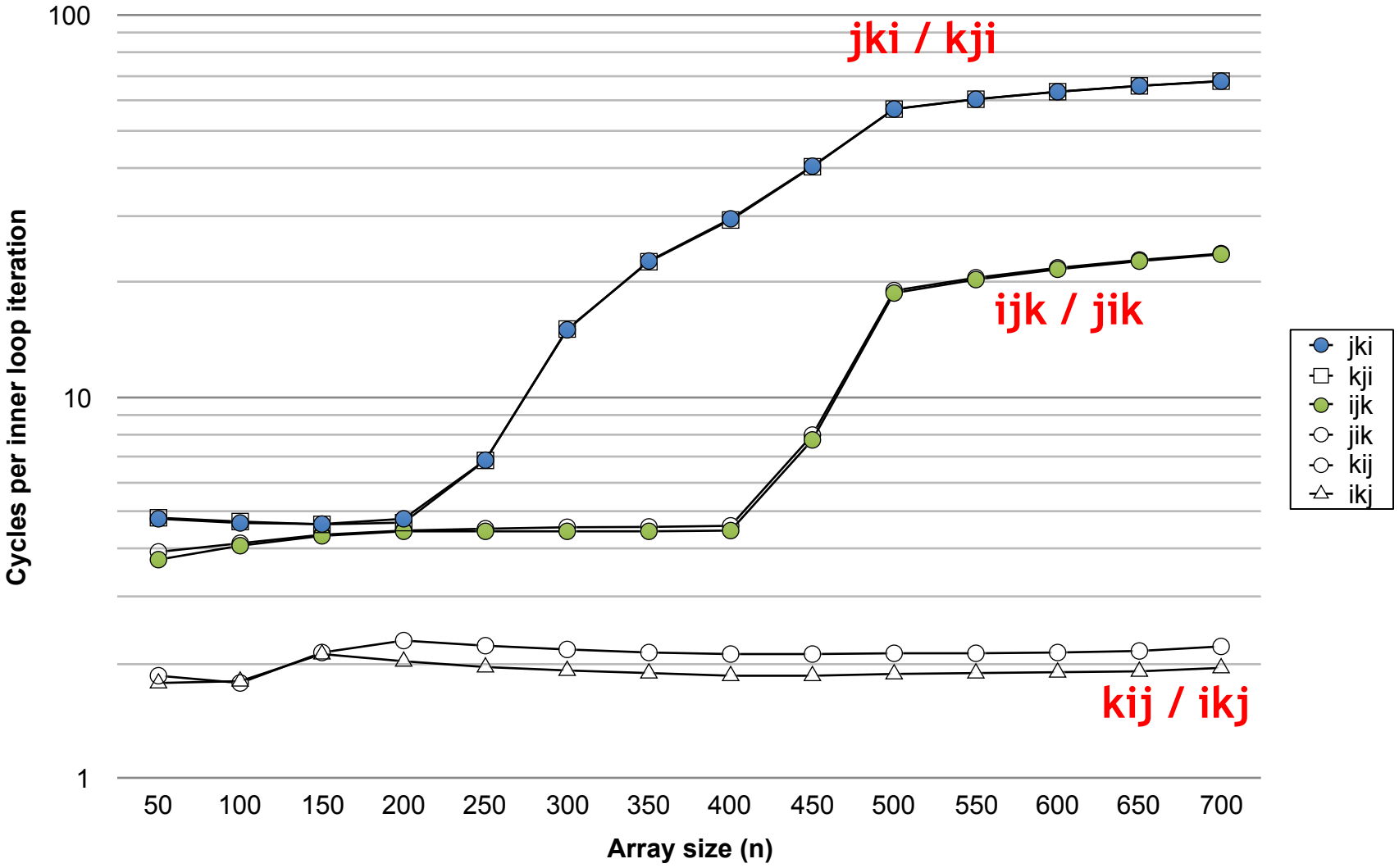
```
for (k=0; k<n; k++) {
 for (i=0; i<n; i++) {
  r = a[i][k];
  for (j=0; j<n; j++){
   c[i][j] += r * b[k][j];}
 }
}
```

kij (& ikj):
- 2 loads, 1 store
- misses/iter = **0.5**

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

jki (& kji):
- 2 loads, 1 store
- misses/iter = **2.0**

Core i7 Matrix Multiply Performance



Today

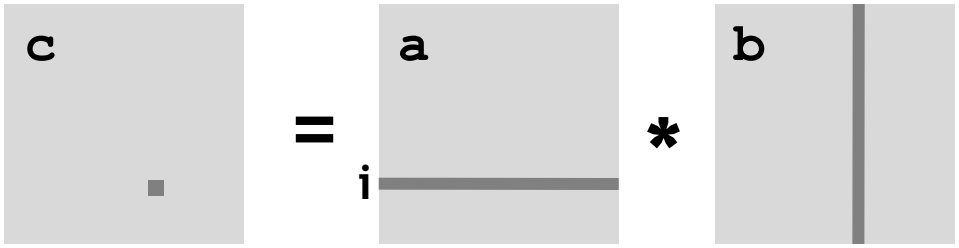
- Cache organization and operation
- Performance impact of caches
  - The memory mountain
  - Rearranging loops to improve spatial locality
  - Using blocking to improve temporal locality

Example: Matrix Multiplication

```
c = (double *) calloc(sizeof(double), n*n);

/* Multiply n x n matrices a and b  */
void mmm(double *a, double *b, double *c, int n) {
    int i, j, k;
    for (i = 0; i < n; i++)
        for (j = 0; j < n; j++)
            for (k = 0; k < n; k++)
                c[i*n + j] += a[i*n + k] * b[k*n + j];
}
```

j

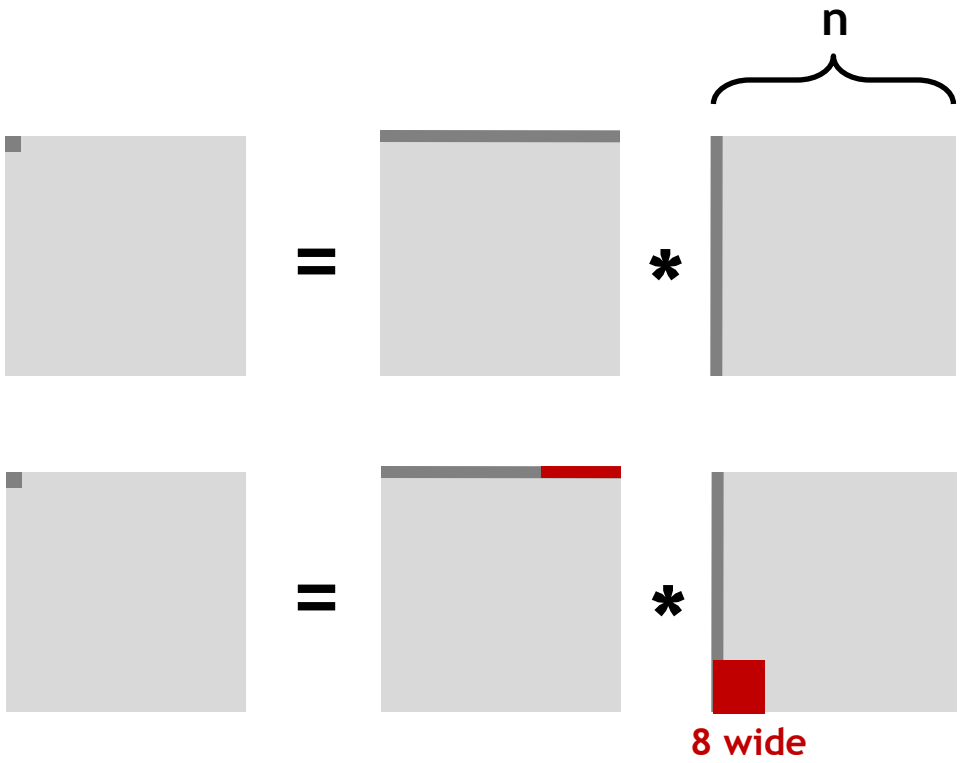


Cache Miss Analysis

- Assume:
  - Matrix elements are doubles
  - Assume the matrix is square
  - Cache block = 8 doubles
  - Cache size C << n (much smaller than n)

- First iteration:
  - n/8 + n = 9n/8 misses

  - Afterwards in cache: (schematic)

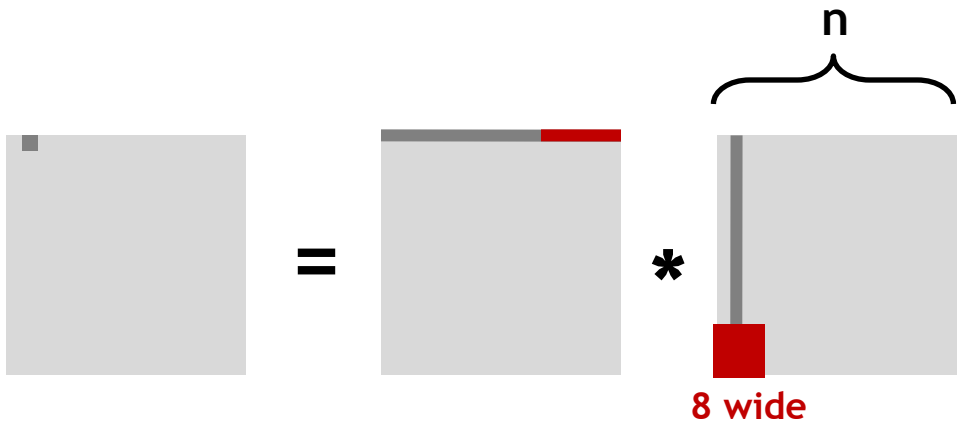


Cache Miss Analysis

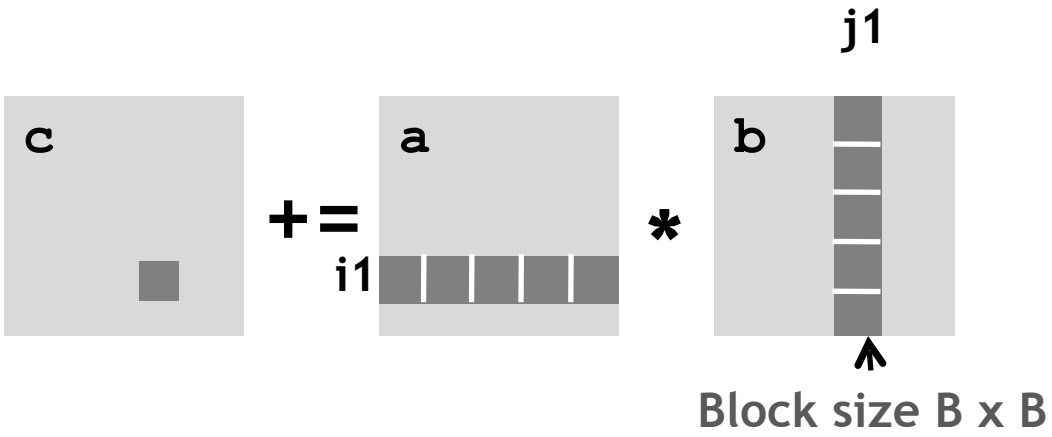
- Assume:
  - Matrix elements are doubles
  - Cache block = 8 doubles
  - Cache size C << n (much smaller than n)

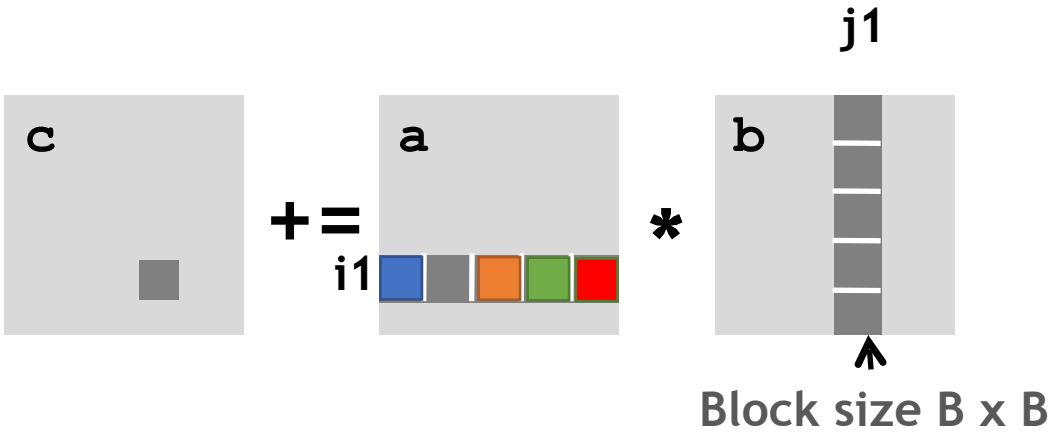
- Second iteration:
  - Again:
    n/8 + n = 9n/8 misses

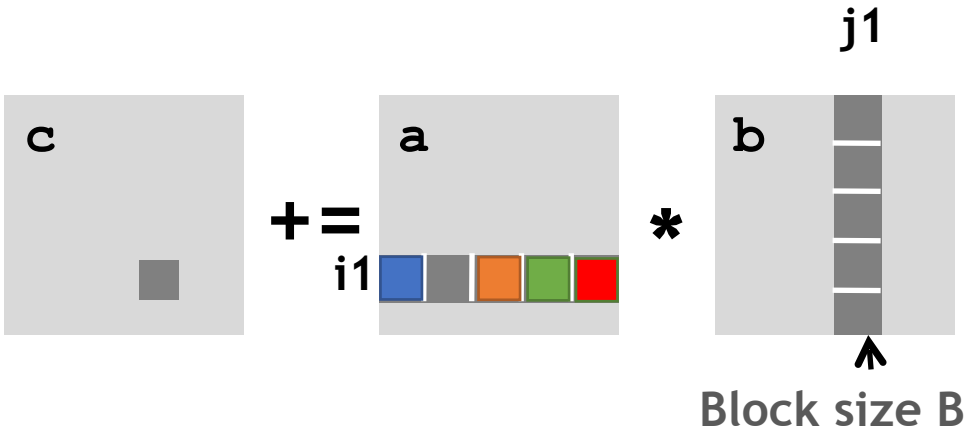
- Total misses:
  - $9n/8 * n^2 = (9/8) * n^3$



Blocked Matrix Multiplication

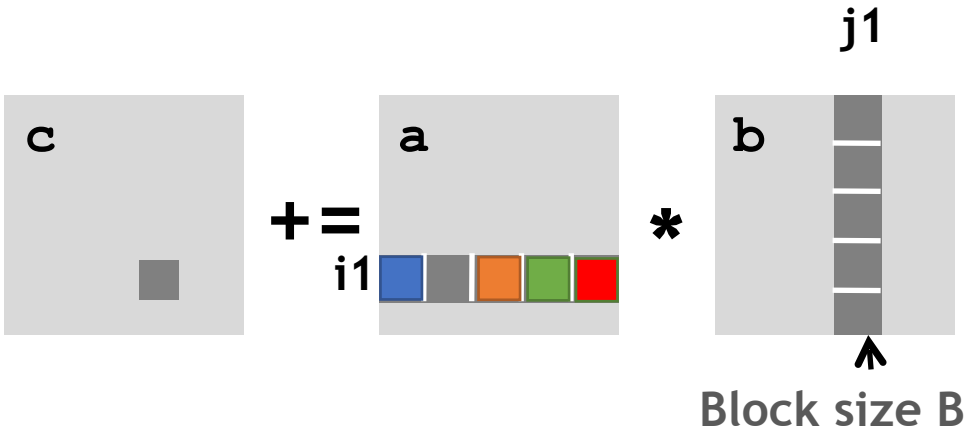



j1
c
a
b
+=
i1
*
Block size B x B



1	2	5	6
3	4	7	8
9	10	13	14
11	12	15	16

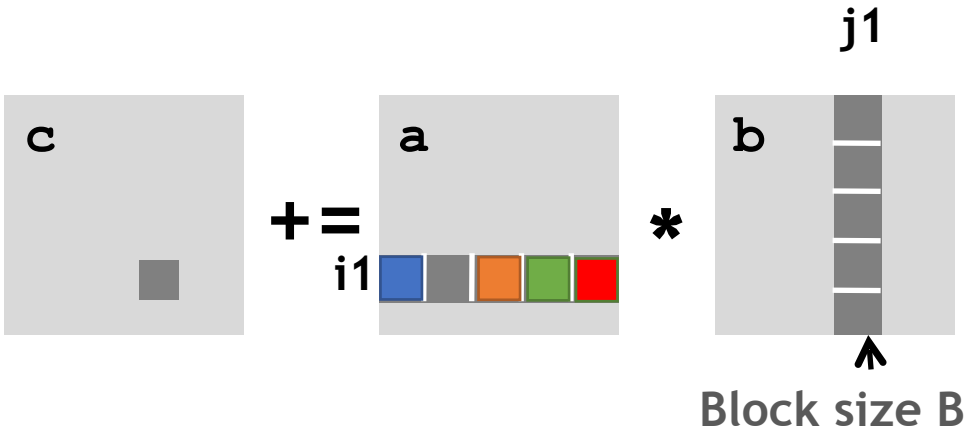
1	2	5	6
3	4	7	8
9	10	13	14
11	12	15	16



1	2	5	6
3	4	7	8
9	10	13	14
11	12	15	16

1	2	5	6
3	4	7	8
9	10	13	14
11	12	15	16

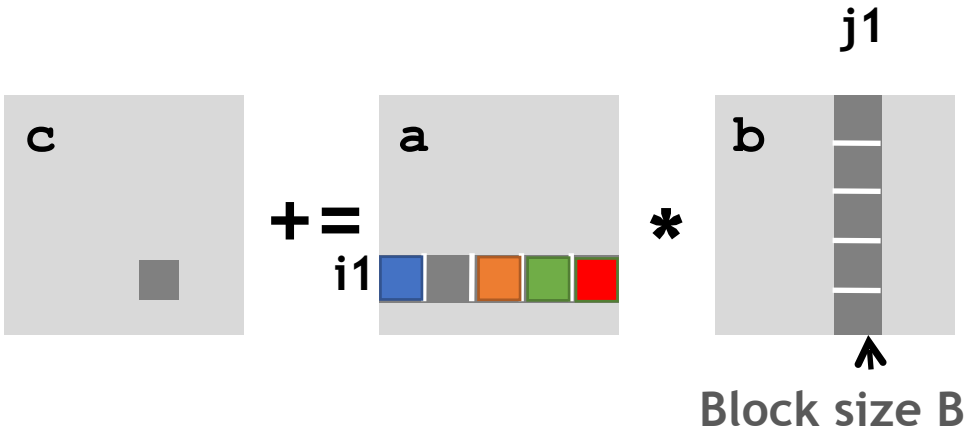
1	2	*	1	2	+	5	6	*	9	10
3	4		3	4		7	8		11	12



1	2	5	6
3	4	7	8
9	10	13	14
11	12	15	16

1	2	5	6
3	4	7	8
9	10	13	14
11	12	15	16

118	132	=	1	2	*	1	2	+	5	6	*	9	10
166	188		3	4		3	4		7	8		11	12



118	132		
166	188		

1	2	5	6
3	4	7	8
9	10	13	14
11	12	15	16

1	2	5	6
3	4	7	8
9	10	13	14
11	12	15	16

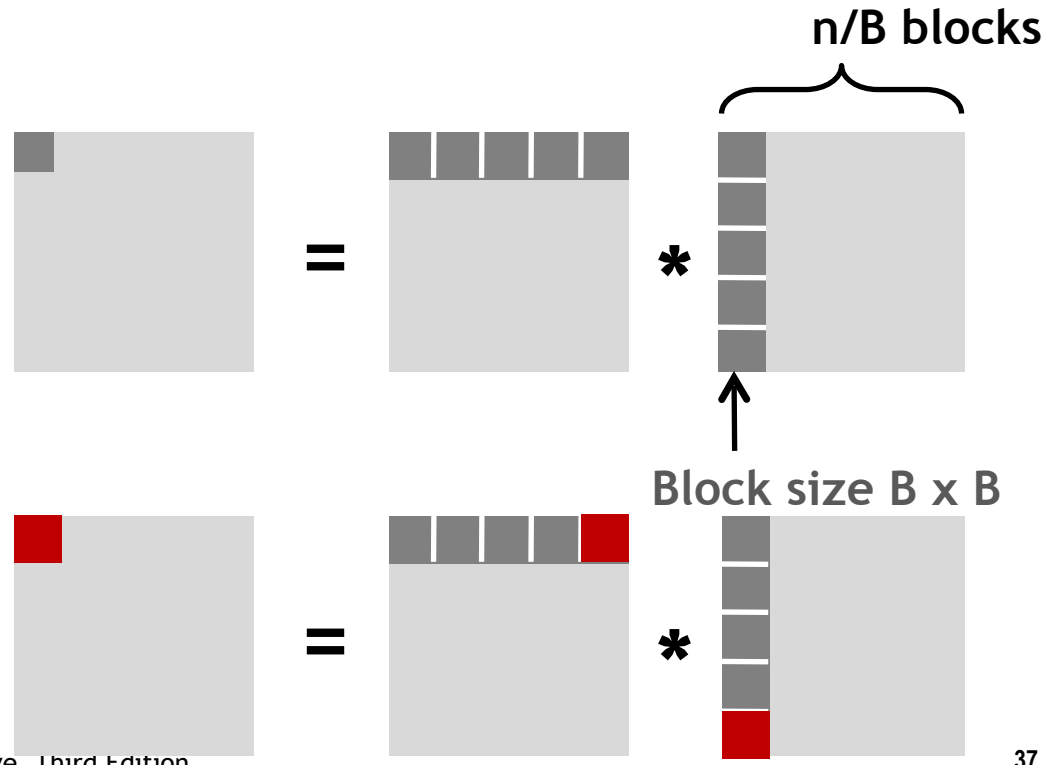
118	132	=	1	2	*	1	2	+	5	6	*	9	10
166	188		3	4		3	4		7	8		11	12

Cache Miss Analysis

- Assume:
  - Square Matrix
  - Cache block = 8 doubles
  - Cache size C << n (much smaller than n)
  - Three blocks fit into cache: $3B^2 < C$ (Where B^2 is the size of B x B block)

- First (block) iteration:
  - $B^2/8$ misses for each block
  - $2n/B * B^2/8 = nB/4$ (omitting matrix c)

  - Afterwards in cache (schematic)

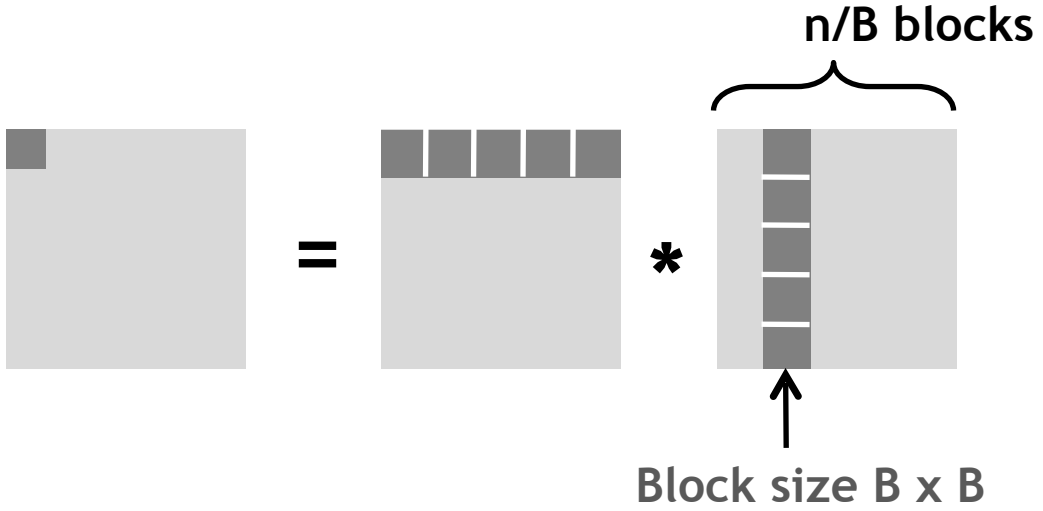


Cache Miss Analysis

- Assume:
  - Cache block = 8 doubles
  - Cache size C << n (much smaller than n)
  - Three blocks fit into cache: $3B^2 < C$

- Second (block) iteration:
  - Same as first iteration
  - $2n/B * B^2/8 = nB/4$

- Total misses:
  - $nB/4 * (n/B)^2 = n^3/(4B)$



Blocking Summary

- No blocking: (9/8) * n^3
- Blocking: 1/(4B) * n^3

- Suggest largest possible block size B, but limit $3B^2 < C$!

- Reason for dramatic difference:
  - Matrix multiplication has inherent temporal locality:
    - Input data: $3n^2$, computation $2n^3$
    - Every array elements used O(n) times!
  - But program has to be written properly

Cache Summary

- Cache memories can have significant performance impact

- You can write your programs to exploit this!
  - Focus on the inner loops, where bulk of computations and memory accesses occur.
  - Try to maximize spatial locality by reading data objects with sequentially with stride 1.
  - Try to maximize temporal locality by using a data object as often as possible once it's read from memory.

Blocked Matrix Multiplication

```
c = (double *) calloc(sizeof(double), n*n);

/* Multiply n x n matrices a and b  */
void mmm(double *a, double *b, double *c, int n) {
    int i, j, k;
    for (i = 0; i < n; i+=B)
        for (j = 0; j < n; j+=B)
            for (k = 0; k < n; k+=B)
                /* B x B mini matrix multiplications */
                for (i1 = i; i1 < i+B; i++)
                    for (j1 = j; j1 < j+B; j++)
                        for (k1 = k; k1 < k+B; k++)
                            c[i1*n+j1] += a[i1*n + k1]*b[k1*n + j1];
}
```

matmult/bmm.c