

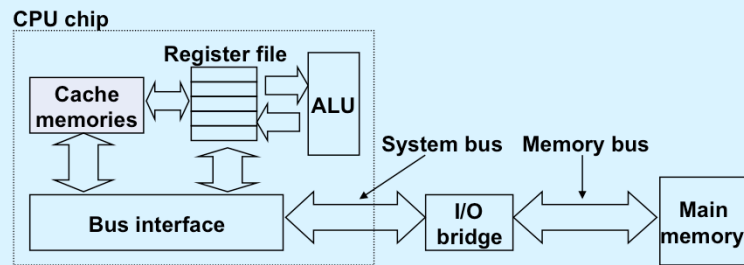
# CS 33

## Caches

Many of the slides in this lecture are either from or adapted from slides provided by the authors of the textbook “Computer Systems: A Programmer’s Perspective,” 2<sup>nd</sup> Edition and are provided from the website of Carnegie-Mellon University, course 15-213, taught by Randy Bryant and David O’Hallaron in Fall 2010. These slides are indicated “Supplied by CMU” in the notes section of the slides.

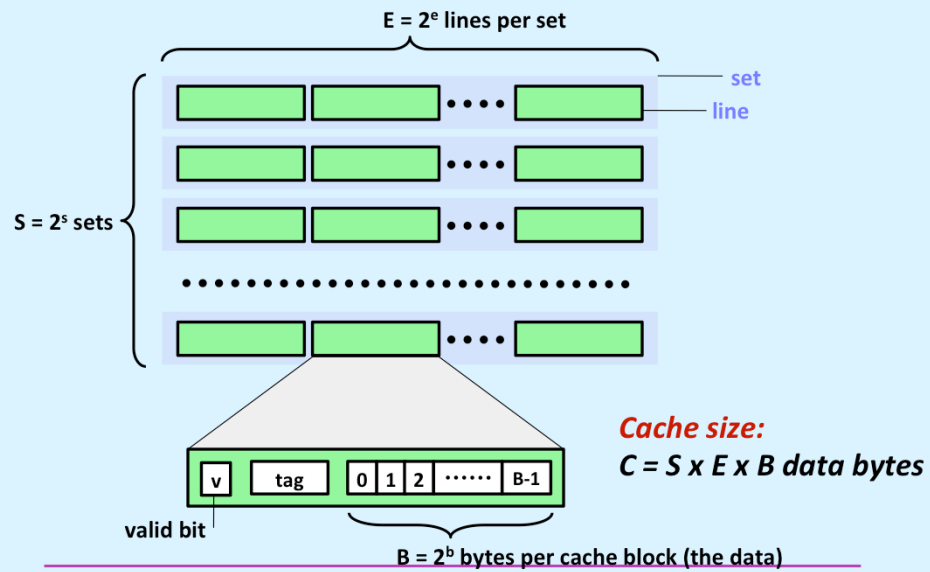
## Cache Memories

- **Cache memories** are small, fast SRAM-based memories managed automatically in hardware
  - hold frequently accessed blocks of main memory
- CPU looks first for data in caches (e.g., L1, L2, and L3), then in main memory
- Typical system structure:



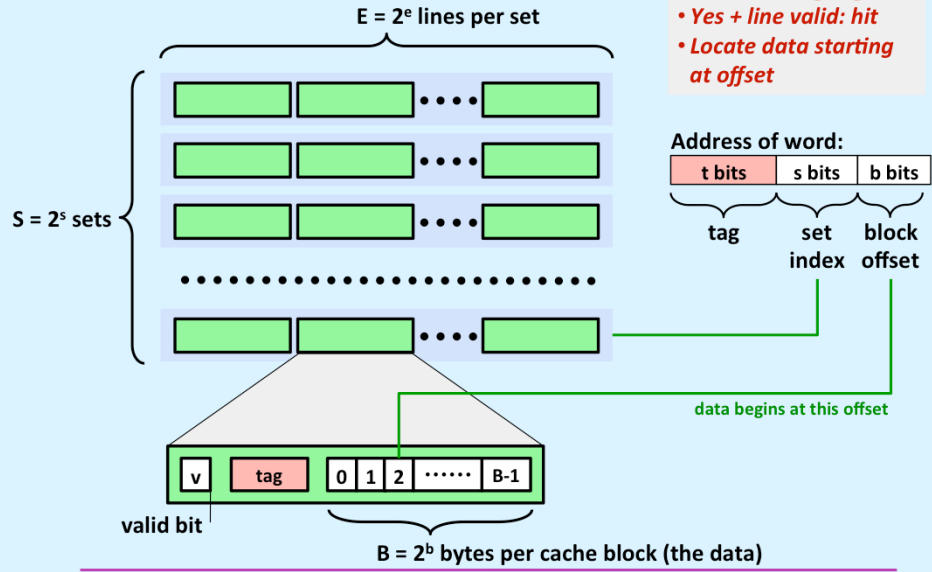
Supplied by CMU.

## General Cache Organization (S, E, B)



Supplied by CMU.

# Cache Read

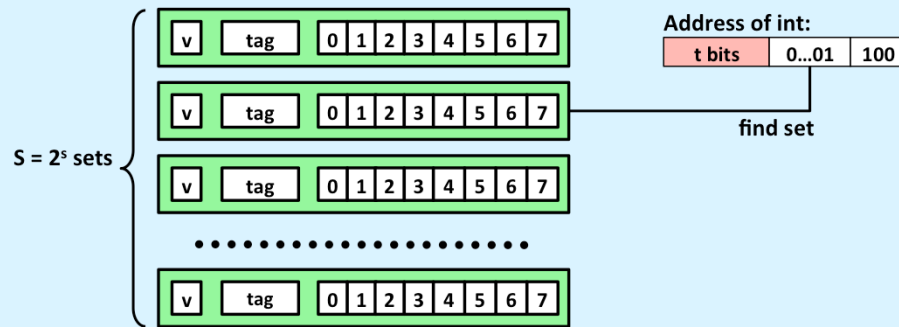


- Locate set
- Check if any line in set has matching tag
- Yes + line valid: hit
- Locate data starting at offset

Supplied by CMU.

## Example: Direct Mapped Cache (E = 1)

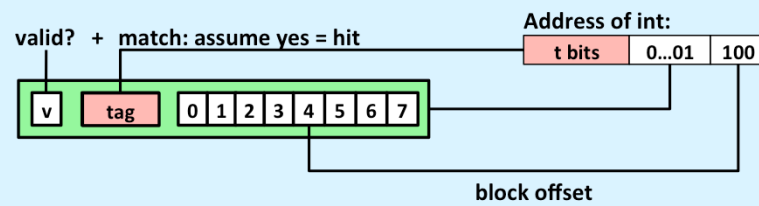
Direct mapped: one line per set  
Assume: cache block size 8 bytes



Supplied by CMU.

## Example: Direct Mapped Cache (E = 1)

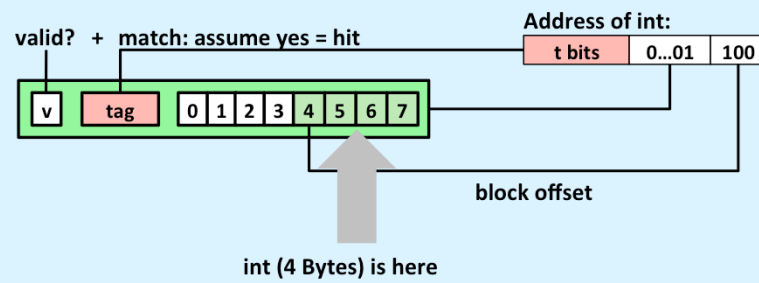
Direct mapped: one line per set  
Assume: cache block size 8 bytes



Supplied by CMU.

## Example: Direct Mapped Cache (E = 1)

Direct mapped: one line per set  
Assume: cache block size 8 bytes



**No match:** old line is evicted and replaced

Supplied by CMU.

## Direct-Mapped Cache Simulation

t=1	s=2	b=1
x	xx	x

M=16 byte addresses, B=2 bytes/block,  
S=4 sets, E=1 Blocks/set

Address trace (reads, one byte per read):

0	[0000] <sub>2</sub> ,	miss
1	[0001] <sub>2</sub> ,	hit
7	[0111] <sub>2</sub> ,	miss
8	[1000] <sub>2</sub> ,	miss
0	[0000] <sub>2</sub>	miss

	v	Tag	Block
Set 0	1	0	M[0-1]
Set 1			
Set 2			
Set 3	1	0	M[6-7]

Supplied by CMU.



## A Higher-Level Example

```
int sum_array_rows(double a[16][16])
{
    int i, j;
    double sum = 0;

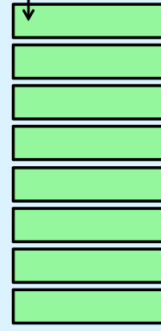
    for (i = 0; i < 16; i++)
        for (j = 0; j < 16; j++)
            sum += a[i][j];
    return sum;
}
```

```
int sum_array_cols(double a[16][16])
{
    int i, j;
    double sum = 0;

    for (j = 0; j < 16; j++)
        for (i = 0; i < 16; i++)
            sum += a[i][j];
    return sum;
}
```

*Ignore the variables sum, i, j*

assume: cold (empty) cache,  
a[0][0] goes here



32 B = 4 doubles

## A Higher-Level Example

```
int sum_array_rows(double a[16][16])
{
    int i, j;
    double sum = 0;

    for (i = 0; i < 16; i++)
        for (j = 0; j < 16; j++)
            sum += a[i][j];
    return sum;
}
```

```
int sum_array_cols(double a[16][16])
{
    int i, j;
    double sum = 0;

    for (j = 0; j < 16; j++)
        for (i = 0; i < 16; i++)
            sum += a[i][j];
    return sum;
}
```

$a_{0,0}$	$a_{0,1}$	$a_{0,2}$	$a_{0,3}$
$a_{0,4}$	$a_{0,5}$	$a_{0,6}$	$a_{0,7}$
$a_{0,8}$	$a_{0,9}$	$a_{0,10}$	$a_{0,11}$
$a_{0,12}$	$a_{0,13}$	$a_{0,14}$	$a_{0,15}$
$a_{1,0}$	$a_{1,1}$	$a_{1,2}$	$a_{1,3}$
$a_{1,4}$	$a_{1,5}$	$a_{1,6}$	$a_{1,7}$
$a_{1,8}$	$a_{1,9}$	$a_{1,10}$	$a_{1,11}$
$a_{1,12}$	$a_{1,13}$	$a_{1,14}$	$a_{1,15}$

32 B = 4 doubles

Note that the cache holds two rows of the matrix.

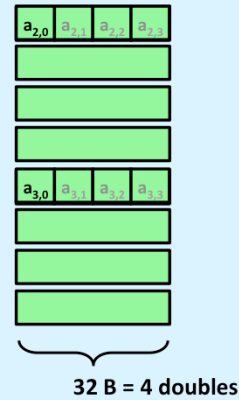
## A Higher-Level Example

```
int sum_array_rows(double a[16][16])
{
    int i, j;
    double sum = 0;

    for (i = 0; i < 16; i++)
        for (j = 0; j < 16; j++)
            sum += a[i][j];
    return sum;
}
```

```
int sum_array_cols(double a[16][16])
{
    int i, j;
    double sum = 0;

    for (j = 0; j < 16; j++)
        for (i = 0; i < 16; i++)
            sum += a[i][j];
    return sum;
}
```



For each reference to an element of the matrix, its entire row is brought into the cache, even though the rest of the row is not immediately used.

## Conflict Misses: Aligned

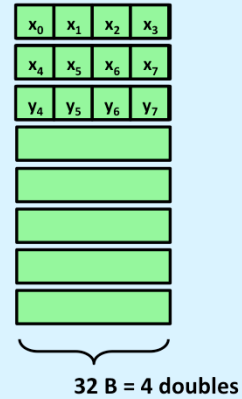
```
double dotprod(double x[8], double y[8]) {  
    double sum = 0.0;  
    int i;  
  
    for (i=0; i<8; i++)  
        sum += x[i] * y[i];  
  
    return sum;  
}
```



If arrays  $x$  and  $y$  have the same alignment, i.e., both start in the same cache set, then each access to an element of  $y$  replaces the cache line containing the corresponding element of  $x$ , and vice versa. The result is that loop is executed very slowly — each access to either array results in a conflict miss.

## Different Alignments

```
double dotprod(double x[8], double y[8]) {  
    double sum = 0.0;  
    int i;  
  
    for (i=0; i<8; i++)  
        sum += x[i] * y[i];  
  
    return sum;  
}
```

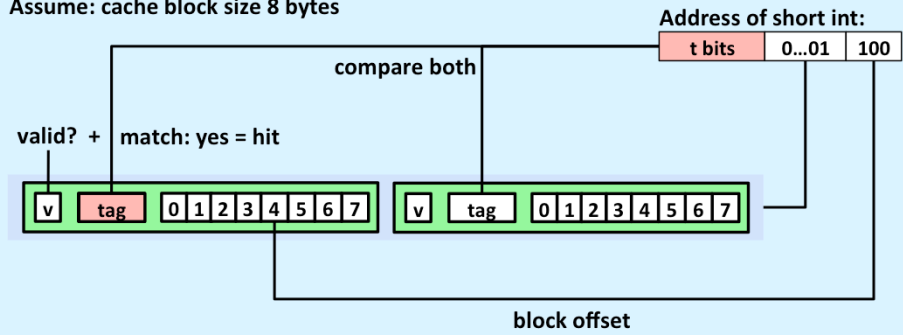


However, if the two arrays start in different cache sets, then the loop executes quickly — there is a cache miss on just every fourth access to each array.

## E-way Set-Associative Cache (Here: E = 2)

E = 2: two lines per set

Assume: cache block size 8 bytes

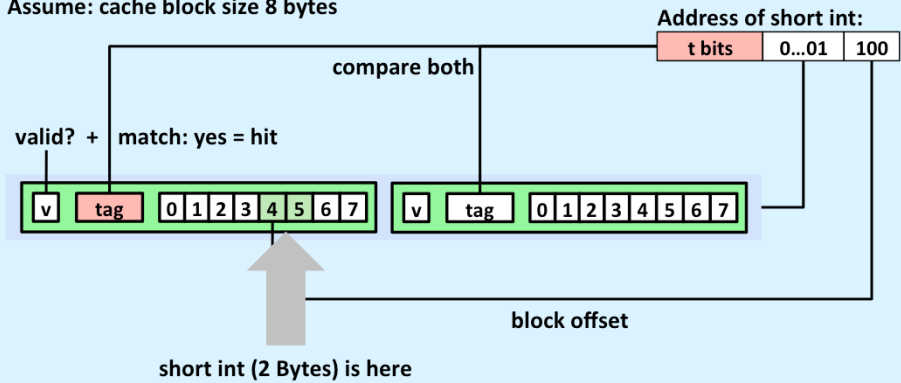


Supplied by CMU.

## E-way Set-Associative Cache (Here: E = 2)

E = 2: two lines per set

Assume: cache block size 8 bytes



### No match:

- One line in set is selected for eviction and replacement
- Replacement policies: random, least recently used (LRU), ...

Supplied by CMU.

## Quiz 1

Address of int:

100	01	100
-----	----	-----

0	v	tag=0	0	0	0	0	1	1	1	1
	v	tag=2	2	2	2	2	3	3	3	3
1	v	tag=0	4	4	4	4	5	5	5	5
	v	tag=4	6	6	6	6	7	7	7	7
2	v	tag=2	8	8	8	8	9	9	9	9
	v	tag=3	a	a	a	a	b	b	b	b
3	v	tag=4	c	c	c	c	d	d	d	d
	v	tag=a	e	e	e	e	f	f	f	f

Given the address above and the cache contents as shown, what is the value of the *int* at the given address?

- a) 1111
- b) 3333
- c) 4444
- d) 7777



## 2-Way Set-Associative Cache Simulation

t=2	s=1	b=1
xx	x	x

M=16 byte addresses, B=2 bytes/block,  
S=2 sets, E=2 blocks/set

Address trace (reads, one byte per read):

0	[0000 <sub>2</sub> ],	miss
1	[0001 <sub>2</sub> ],	hit
7	[0111 <sub>2</sub> ],	miss
8	[1000 <sub>2</sub> ],	miss
0	[0000 <sub>2</sub> ]	hit

	v	Tag	Block
Set 0	1	00	M[0-1]
	1	10	M[8-9]
Set 1	1	01	M[6-7]
	0		

Supplied by CMU.

## A Higher-Level Example

*Ignore the variables sum, i, j*

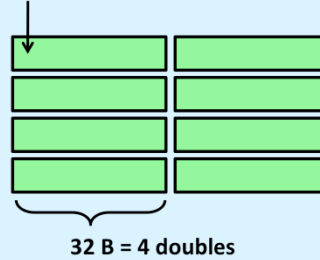
```
int sum_array_rows(double a[16][16])
{
    int i, j;
    double sum = 0;

    for (i = 0; i < 16; i++)
        for (j = 0; j < 16; j++)
            sum += a[i][j];
    return sum;
}
```

```
int sum_array_rows(double a[16][16])
{
    int i, j;
    double sum = 0;

    for (j = 0; j < 16; j++)
        for (i = 0; i < 16; i++)
            sum += a[i][j];
    return sum;
}
```

assume: cold (empty) cache,  
a[0][0] goes here



## A Higher-Level Example

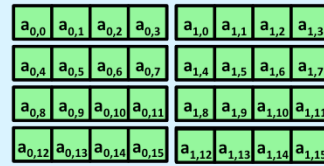
*Ignore the variables sum, i, j*

```
int sum_array_rows(double a[16][16])
{
    int i, j;
    double sum = 0;

    for (i = 0; i < 16; i++)
        for (j = 0; j < 16; j++)
            sum += a[i][j];
    return sum;
}
```

```
int sum_array_cols(double a[16][16])
{
    int i, j;
    double sum = 0;

    for (j = 0; j < 16; j++)
        for (i = 0; i < 16; i++)
            sum += a[i][j];
    return sum;
}
```



The cache still holds two rows of the matrix, but each row may go into one of two different cache lines. In the slide, the first row goes into the first lines of the cache sets, the second row goes into the second lines of the cache sets.

## A Higher-Level Example

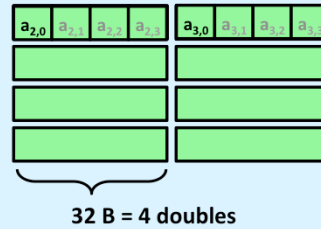
*Ignore the variables sum, i, j*

```
int sum_array_rows(double a[16][16])
{
    int i, j;
    double sum = 0;

    for (i = 0; i < 16; i++)
        for (j = 0; j < 16; j++)
            sum += a[i][j];
    return sum;
}
```

```
int sum_array_cols(double a[16][16])
{
    int i, j;
    double sum = 0;

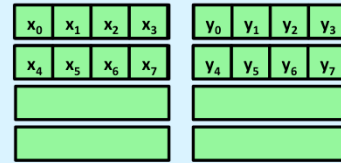
    for (j = 0; j < 16; j++)
        for (i = 0; i < 16; i++)
            sum += a[i][j];
    return sum;
}
```



There is still a cache miss on each access.

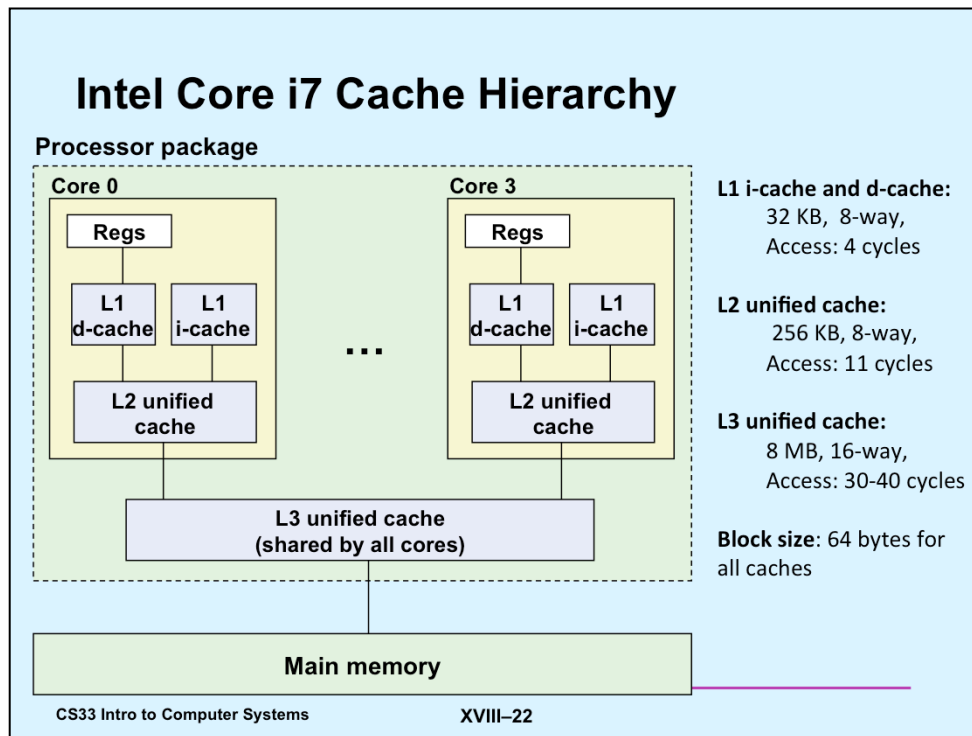
## Conflict Misses

```
double dotprod(double x[8], double y[8]) {  
    double sum = 0.0;  
    int i;  
  
    for (i=0; i<8; i++)  
        sum += x[i] * y[i];  
  
    return sum;  
}
```



32 B = 4 doubles

With a 2-way set-associative cache, our dot-product example runs quickly even if the two arrays have the same alignment.



Supplied by CMU.

The L3 cache is known as the *last-level cache* (LLC) in the Intel documentation.

## What About Writes?

- **Multiple copies of data exist:**
    - L1, L2, main memory, disk
  - **What to do on a write-hit?**
    - **write-through** (write immediately to memory)
    - **write-back** (defer write to memory until replacement of line)
      - » need a dirty bit (line different from memory or not)
  - **What to do on a write-miss?**
    - **write-allocate** (load into cache, update line in cache)
      - » good if more writes to the location follow
    - **no-write-allocate** (writes immediately to memory)
  - **Typical**
    - write-through + no-write-allocate
    - write-back + write-allocate
-

## Cache Performance Metrics

- **Miss rate**
  - fraction of memory references not found in cache  
(misses / accesses)  
 $= 1 - \text{hit rate}$
  - typical numbers (in percentages):
    - » 3-10% for L1
    - » can be quite small (e.g., < 1%) for L2, depending on size, etc.
- **Hit time**
  - time to deliver a line in the cache to the processor
    - » includes time to determine whether the line is in the cache
  - typical numbers:
    - » 1-2 clock cycles for L1
    - » 5-20 clock cycles for L2
- **Miss penalty**
  - additional time required because of a miss
    - » typically 50-200 cycles for main memory (trend: increasing!)

Supplied by CMU.



## Let's Think About Those Numbers

- Huge difference between a hit and a miss
  - could be 100x, if just L1 and main memory
- Would you believe 99% hit rate is twice as good as 97%?
  - consider:
    - cache hit time of 1 cycle
    - miss penalty of 100 cycles
  - average access time:
    - 97% hits:  $.97 * 1 \text{ cycle} + 0.03 * 100 \text{ cycles} \approx 4 \text{ cycles}$
    - 99% hits:  $.99 * 1 \text{ cycle} + 0.01 * 100 \text{ cycles} \approx 2 \text{ cycles}$
- This is why “miss rate” is used instead of “hit rate”

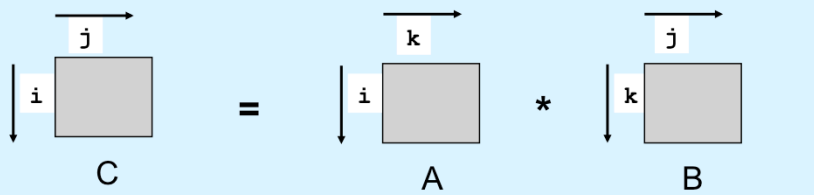
## Writing Cache-Friendly Code

- **Make the common case go fast**
  - focus on the inner loops of the core functions
- **Minimize the misses in the inner loops**
  - repeated references to variables are good (**temporal locality**)
  - stride-1 reference patterns are good (**spatial locality**)

**Key idea: our qualitative notion of locality is quantified through our understanding of cache memories**

## Miss-Rate Analysis for Matrix Multiply

- **Assume:**
  - Block size = 32B (big enough for four 64-bit words)
  - matrix dimension (N) is very large
    - » approximate  $1/N$  as 0.0
  - cache is not big enough to hold multiple rows
- **Analysis method:**
  - look at access pattern of inner loop



Supplied by CMU.

## Matrix Multiplication Example

- **Description:**
  - multiply  $N \times N$  matrices
  - $O(N^3)$  total operations
  - $N$  reads per source element
  - $N$  values summed per destination
    - » but may be able to hold in register

```
/* 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;  
    }  
}
```

*Variable sum  
held in register*

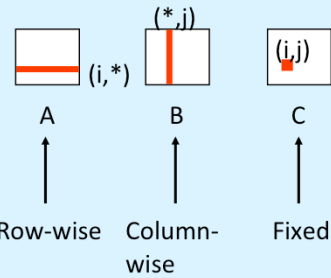
## Layout of C Arrays in Memory (review)

- **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) > 4 bytes, exploit spatial locality
    - » compulsory miss rate = 4 bytes / B
- **Stepping through rows in one column:**
  - **for** (`i = 0; i < n; i++`)  
    `sum += a[i][0];`
  - accesses distant elements
  - no spatial locality!
    - » compulsory 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;
  }
}
```

Inner loop:



Misses per inner loop iteration:

<u>A</u>	<u>B</u>	<u>C</u>
0.25	1.0	0.0

Supplied by CMU.

Assume we are multiplying arrays of doubles, thus each element is eight bytes long, and thus a cache line holds four matrix elements.

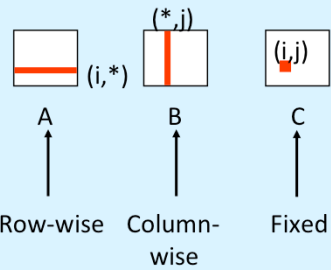
## Matrix Multiplication (jik)

```

/* 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
    }
}

```

Inner loop:



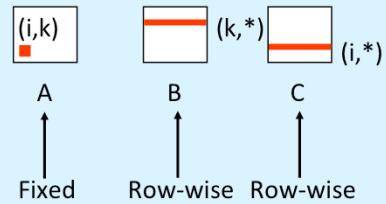
Misses per inner loop iteration:

<u>A</u>	<u>B</u>	<u>C</u>
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];  
  }  
}
```

Inner loop:



Misses per inner loop iteration:

<u>A</u>	<u>B</u>	<u>C</u>
0.0	0.25	0.25

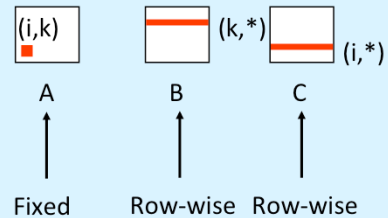
Supplied by CMU.



## 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];  
    }  
}
```

Inner loop:



Misses per inner loop iteration:

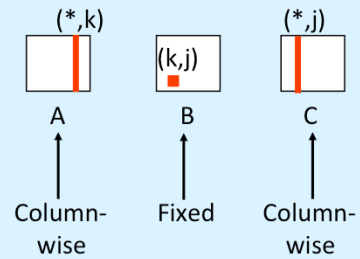
<u>A</u>	<u>B</u>	<u>C</u>
0.0	0.25	0.25

Supplied by CMU.

## 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;  
  }  
}
```

Inner loop:



Misses per inner loop iteration:

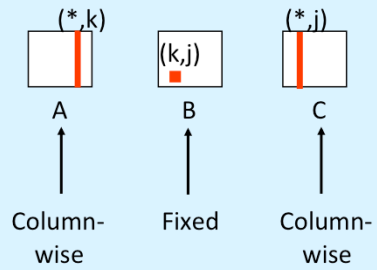
<u>A</u>	<u>B</u>	<u>C</u>
1.0	0.0	1.0

Supplied by CMU.

## 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;  
  }  
}
```

Inner loop:



Misses per inner loop iteration:

<u>A</u>	<u>B</u>	<u>C</u>
1.0	0.0	1.0

Supplied by CMU.

## 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];  
  }
```

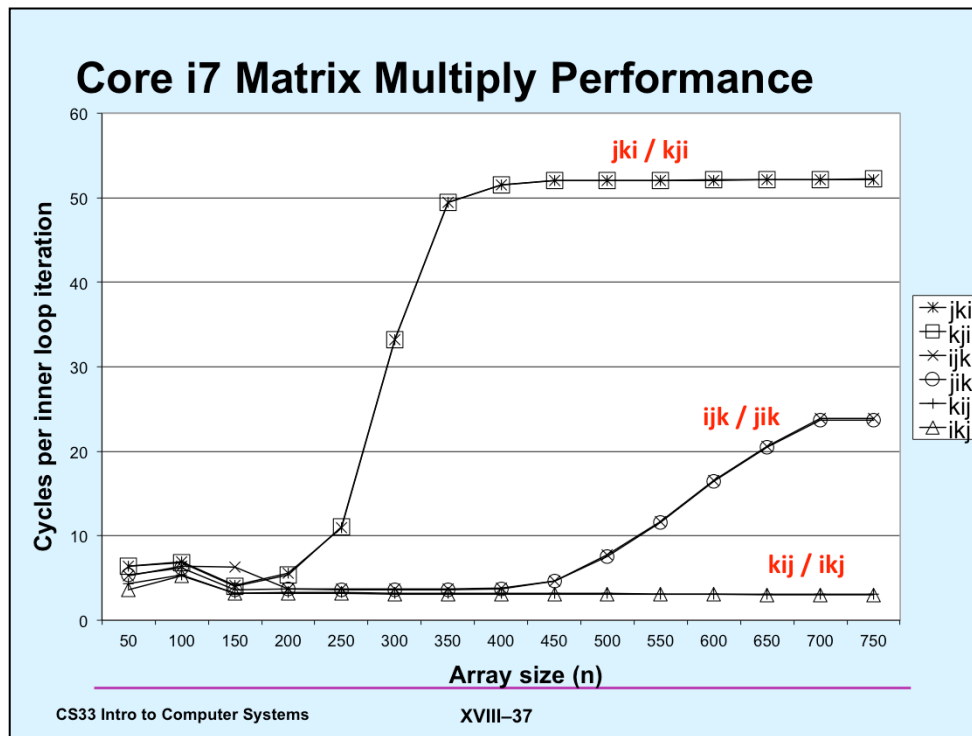
**kij (& ikj):**

- 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):**

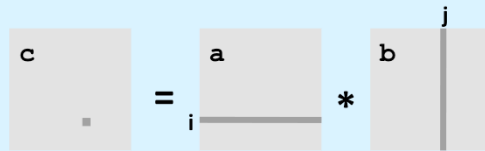
- 2 loads, 1 store
- misses/iter = **2.0**



Supplied by CMU.

## Matrix Multiplication: More Analysis

```
/* 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];  
}
```



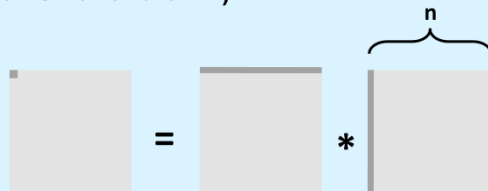
## Cache-Miss Analysis

- **Assume:**

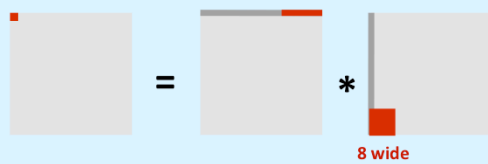
- matrix elements are doubles
- cache block = 8 doubles
- cache size  $C \ll n$  (much smaller than  $n$ )

- **First iteration:**

- $n/8 + n = 9n/8$  misses



- afterwards **in cache:**  
(schematic)



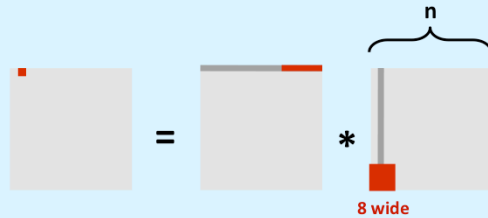
## Cache-Miss Analysis

- **Assume:**

- matrix elements are doubles
- cache block = 8 doubles
- cache size  $C \ll n$  (much smaller than  $n$ )

- **Second iteration:**

- again:  
 $n/8 + n = 9n/8$  misses



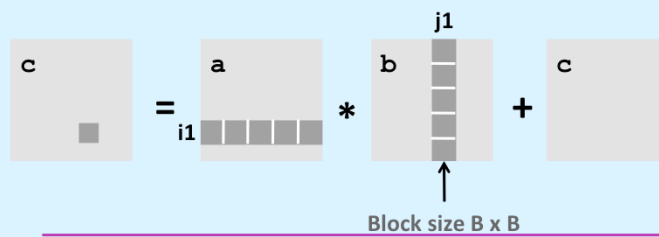
- **Total misses:**

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



## Blocked Matrix Multiplication

```
/* Multiply n x n matrices a and b */
void mmnm(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];
}
```



Supplied by CMU.

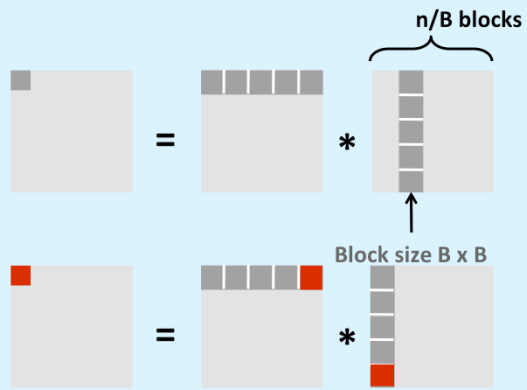
## Cache-Miss Analysis

- **Assume:**

- cache block = 8 doubles
- cache size  $C \ll n$  (much smaller than  $n$ )
- three blocks  $\blacksquare$  fit into cache:  $3B^2 < C$


- **First (block) iteration:**

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



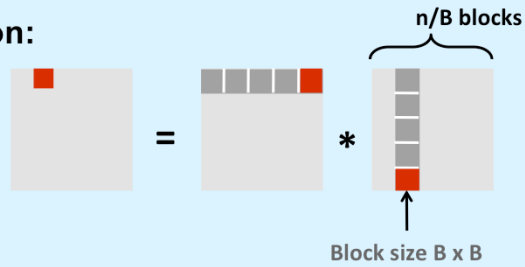
## Cache-Miss Analysis

- **Assume:**

- cache block = 8 doubles
- cache size  $C \ll n$  (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:**

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

## 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 element used  $O(n)$  times!
  - but program has to be written properly

Supplied by CMU.

## Quiz 2

**What is the smallest value of B (in 8-byte doubles) for which the cache-miss analysis works?**

- a) 1
- b) 2
- c) 4
- d) 8

## Concluding Observations

- **Programmer can optimize for cache performance**
  - how data structures are organized
  - how data are accessed
    - » nested loop structure
    - » blocking is a general technique
- **All systems favor “cache-friendly code”**
  - getting absolute optimum performance is very platform specific
    - » cache sizes, line sizes, associativities, etc.
  - can get most of the advantage with generic code
    - » keep working set reasonably small (temporal locality)
    - » use small strides (spatial locality)

Supplied by CMU.