

# COMP201

## Computer Systems & Programming

### Lecture #24 – Optimization



**KOÇ**  
**UNIVERSITY**

Aykut Erdem // Koç University // Fall 2021

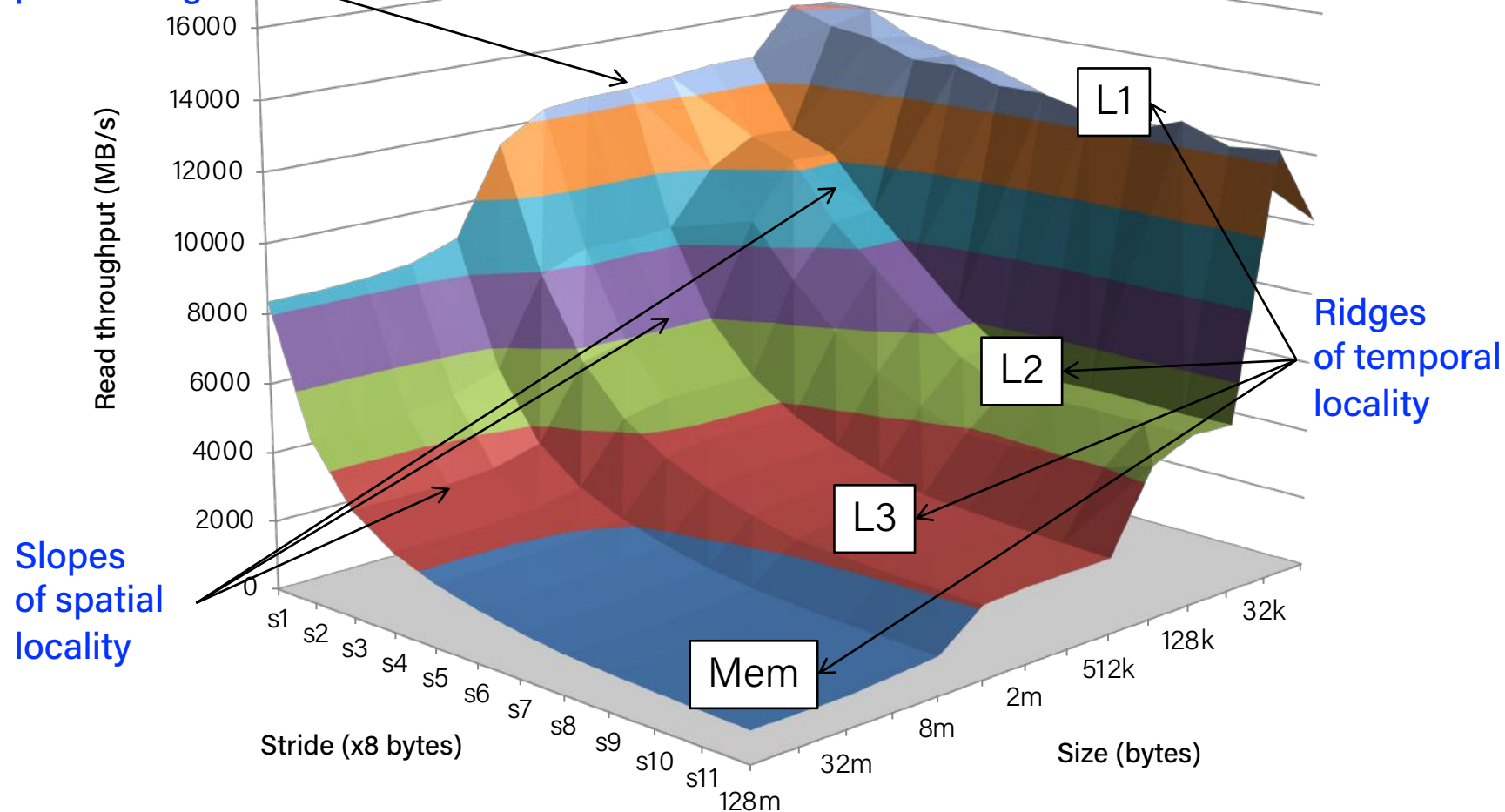
# Recap

- Cache memory organization and operation
- Memory mountation



# Recap: The Memory Mountain

Aggressive  
prefetching



Core i7 Haswell  
2.1 GHz  
32 KB L1 d-cache  
256 KB L2 cache  
8 MB L3 cache  
64 B block size

# Learning Goals

- Understand how we can optimize our code to improve efficiency and speed
- Learn about the optimizations GCC can perform

# Plan for Today

- Writing cache-friendly code
- Optimization

**Disclaimer:** Slides for this lecture were borrowed from  
—Nick Troccoli's Stanford CS107 class

# Plan for Today

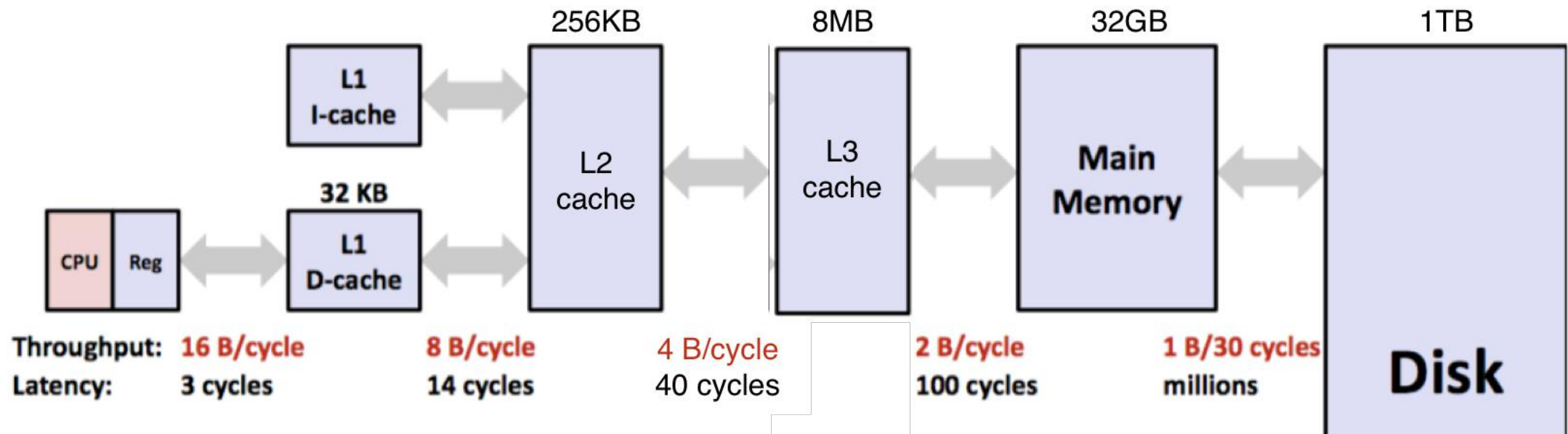
- Writing cache-friendly code
- Optimization

# Lecture Plan

- Writing cache-friendly code
  - Rearranging loops to improve spatial locality
  - Using blocking to improve temporal locality
- Optimization

# Caching

- Processor speed is not the only bottleneck in program performance – memory access is perhaps even more of a bottleneck!
- Memory exists in levels and goes from *really fast* (registers) to *really slow* (disk).
- As data is more frequently used, it ends up in faster and faster memory.





# Caching

All caching depends on locality.

## **Temporal locality**

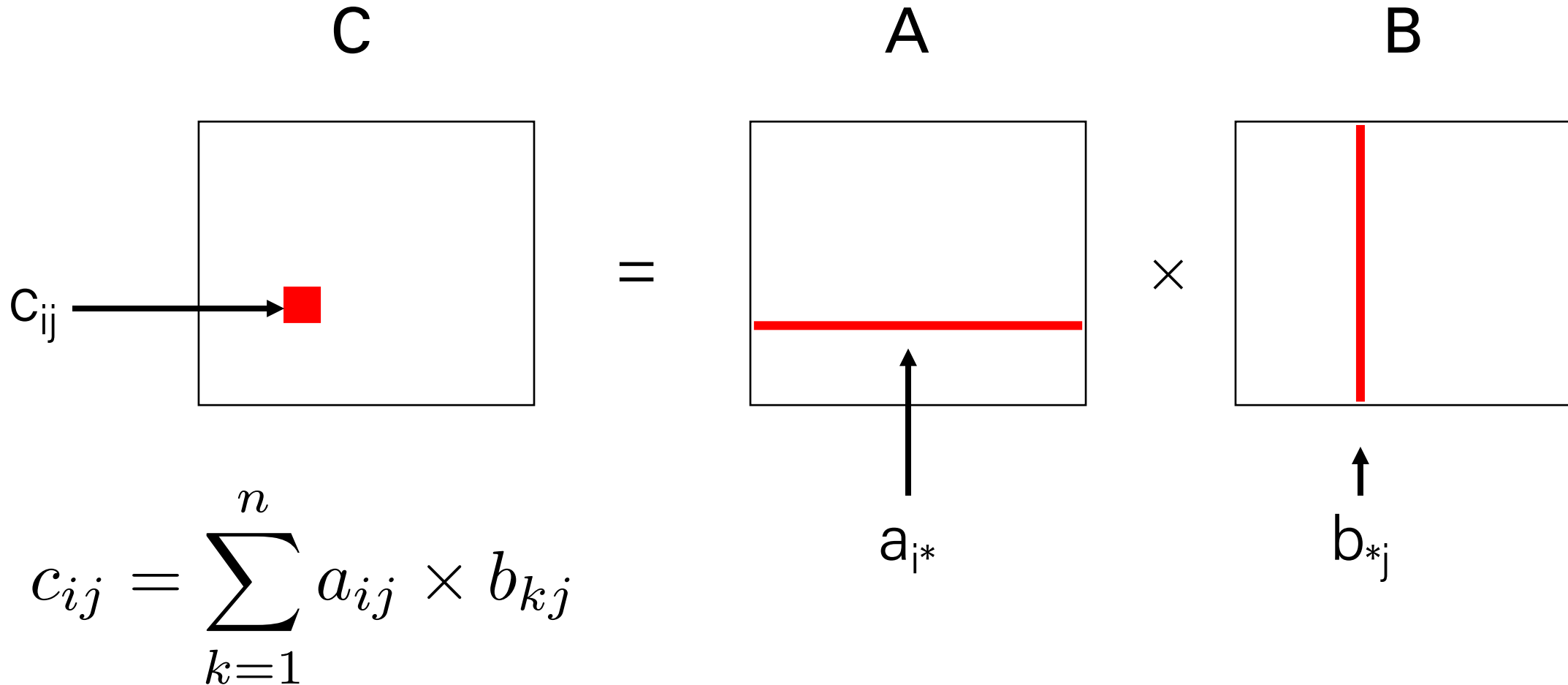
- Repeat access to the same data tends to be co-located in TIME
- Intuitively: things I have used recently, I am likely to use again soon

## **Spatial locality**

- Related data tends to be co-located in SPACE
- Intuitively: data that is near a used item is more likely to also be accessed

# Example: Matrix Multiplication

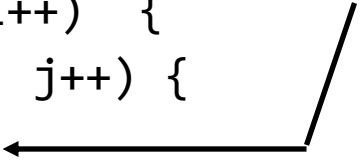
# Matrix Multiplication Example



# Matrix Multiplication Example

- Description:

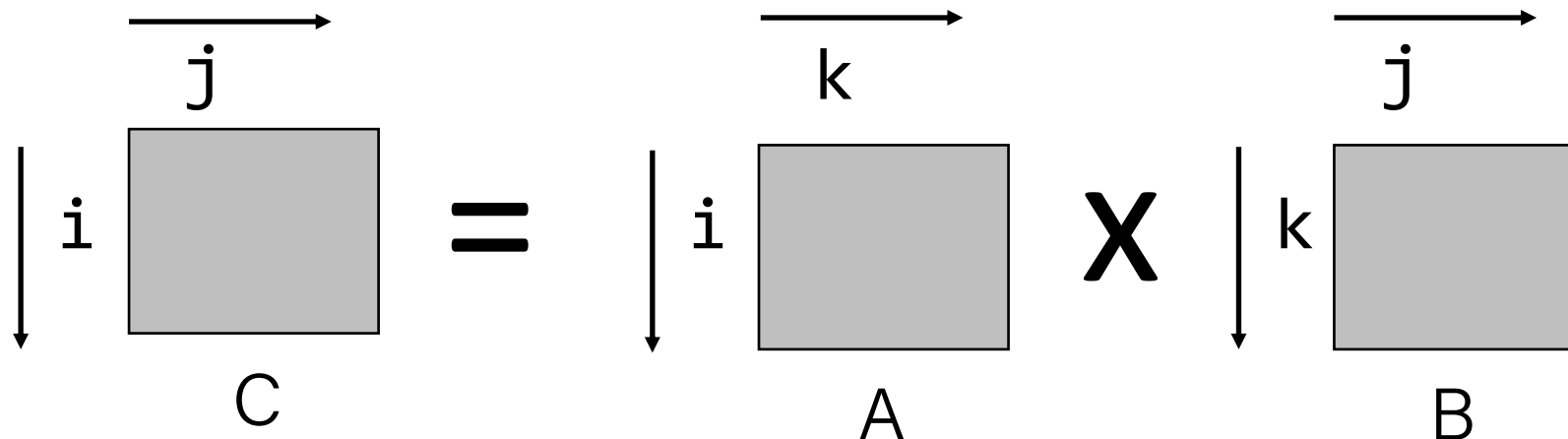
- Multiply  $N \times N$  matrices
- Matrix elements are doubles (8 bytes)
- $O(N^3)$  total operations
- $N$  reads per source element
- $N$  values summed per destination
  - but may be able to hold in register

```
/* 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;  
    }  
}
```

matmult/mm.c

# Miss Rate Analysis for Matrix Multiply

- Assume
  - Block size = 32B (big enough for four doubles)
  - Matrix dimension (N) is very large
    - Approximate  $1/N$  as 0.0
  - 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<sub>ij</sub>) bytes, exploit spatial locality
  - miss rate = sizeof(a<sub>ij</sub>) / 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%)



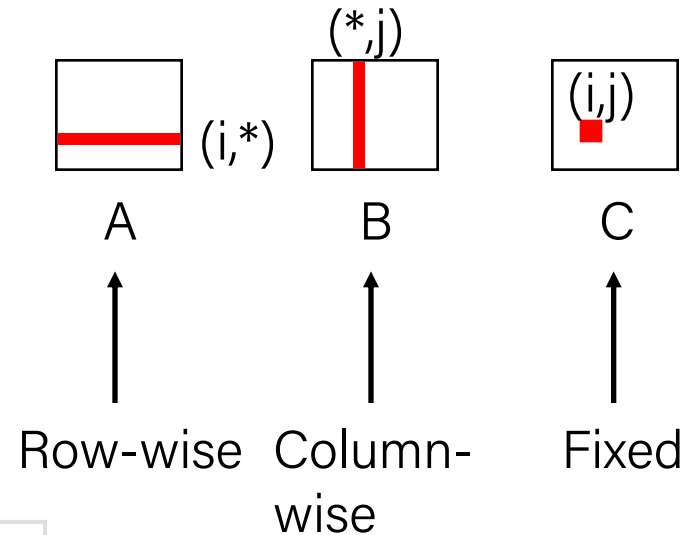
# Matrix Multiplication (ijk)

```
/* 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;
    }
}
```

Misses per inner loop iteration:

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

Inner loop:

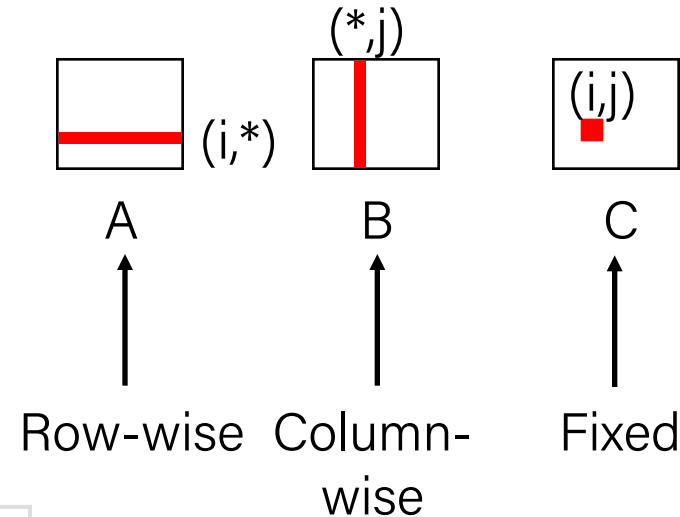


matmult/mm.c

# 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:



matmult/mm.c

Misses per inner loop iteration:

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

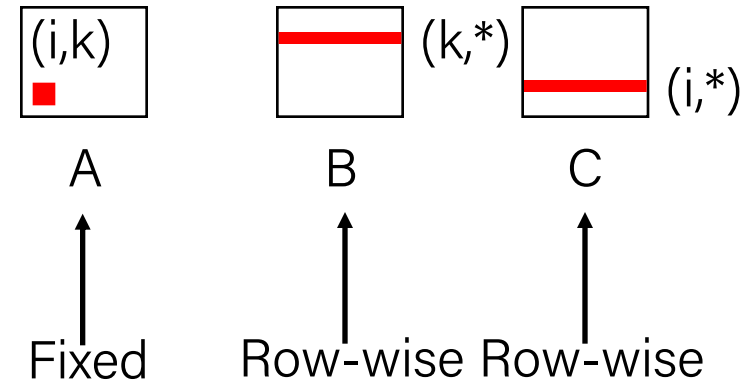
# Matrix Multiplication (kij)

```
/* 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];
    }
}
```

Misses per inner loop iteration:

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

Inner loop:



matmult/mm.c

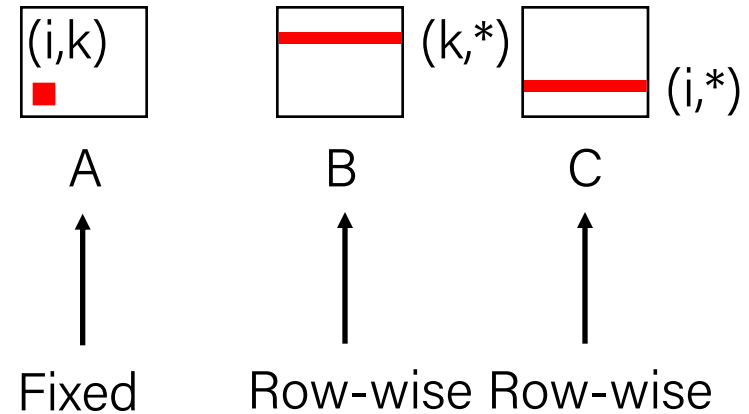
# Matrix Multiplication (ikj)

```
/* 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];  
    }  
}
```

Misses per inner loop iteration:

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

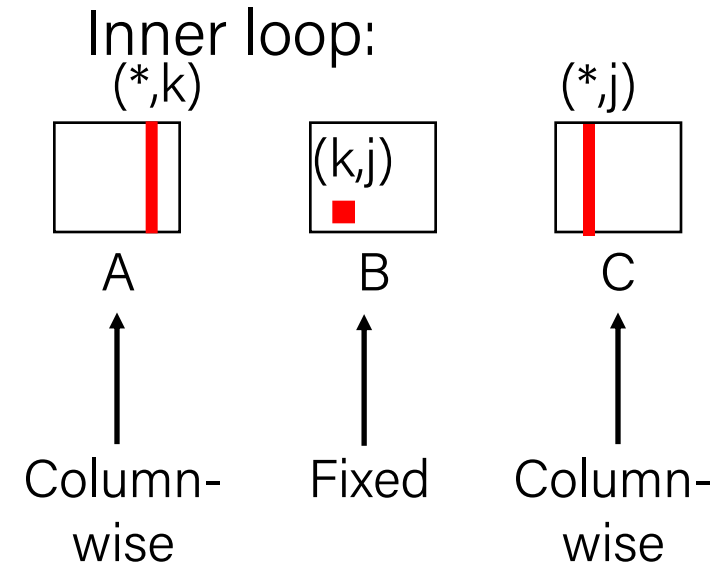
Inner loop:



matmult/mm.c

# 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

Misses per inner loop iteration:

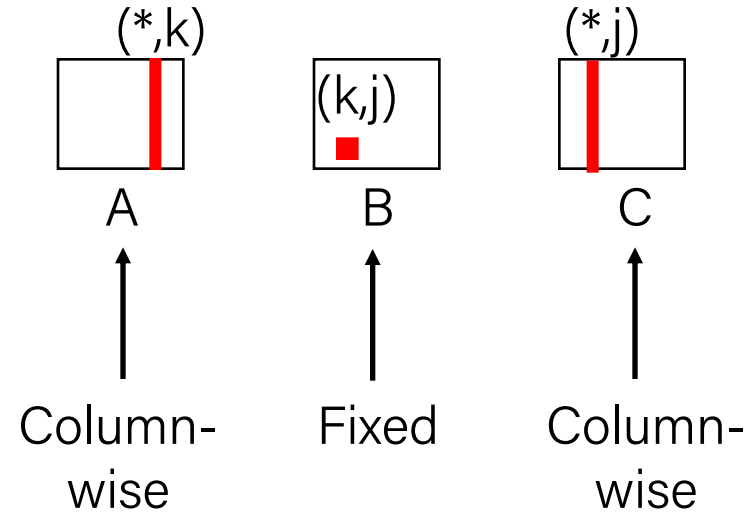
<u>A</u>	<u>B</u>	<u>C</u>
1.0	0.0	1.0

# 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:

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

```
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];  
    }  
}
```

```
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;  
    }  
}
```

ijk (& jik):

- 2 loads, 0 stores
- misses/iter = 1.25

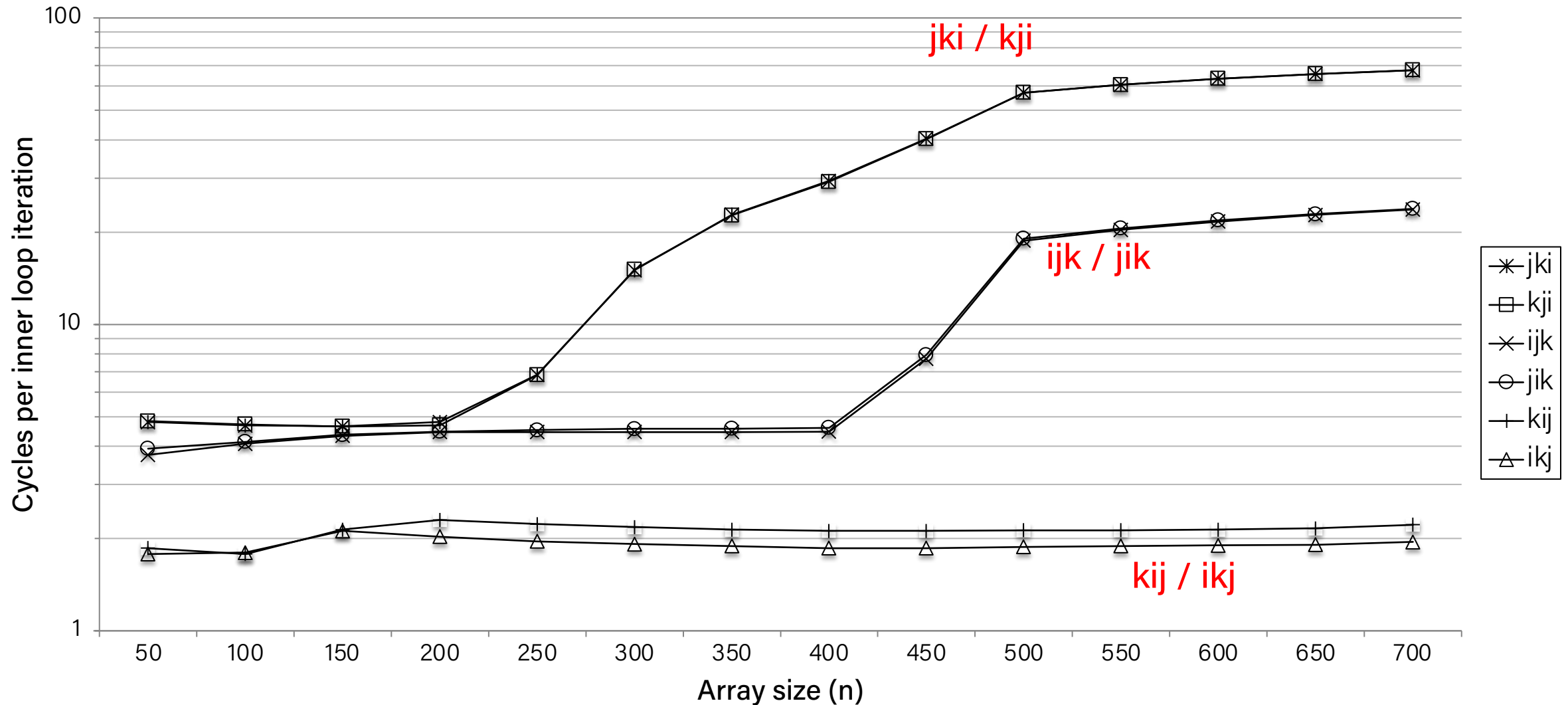
kij (& ikj):

- 2 loads, 1 store
- misses/iter = 0.5

jki (& kji):

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

# Core i7 Matrix Multiply Performance



# Lecture Plan

- Writing cache-friendly code
  - Rearranging loops to improve spatial locality
  - Using blocking to improve temporal locality
- Optimization

# 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];  
}
```

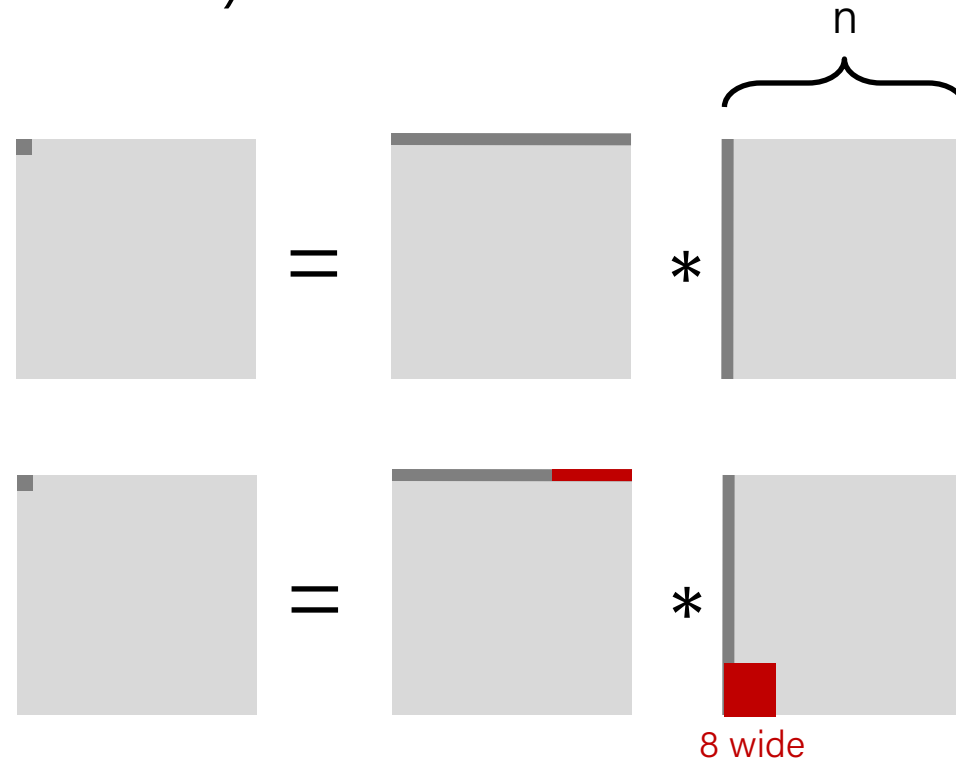


# Cache Miss Analysis

- 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



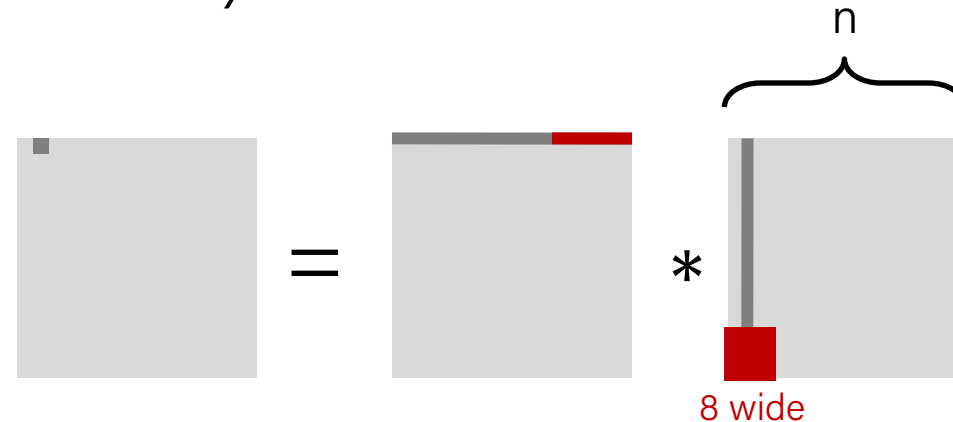
- Afterwards **in cache:**  
(schematic)

# 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:**

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

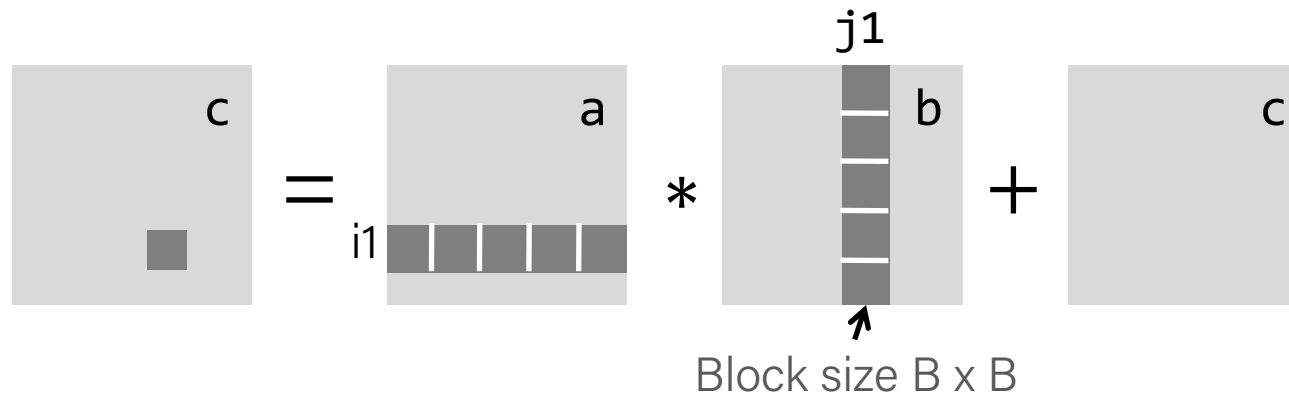


# Blocked Matrix Multiplication


```
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

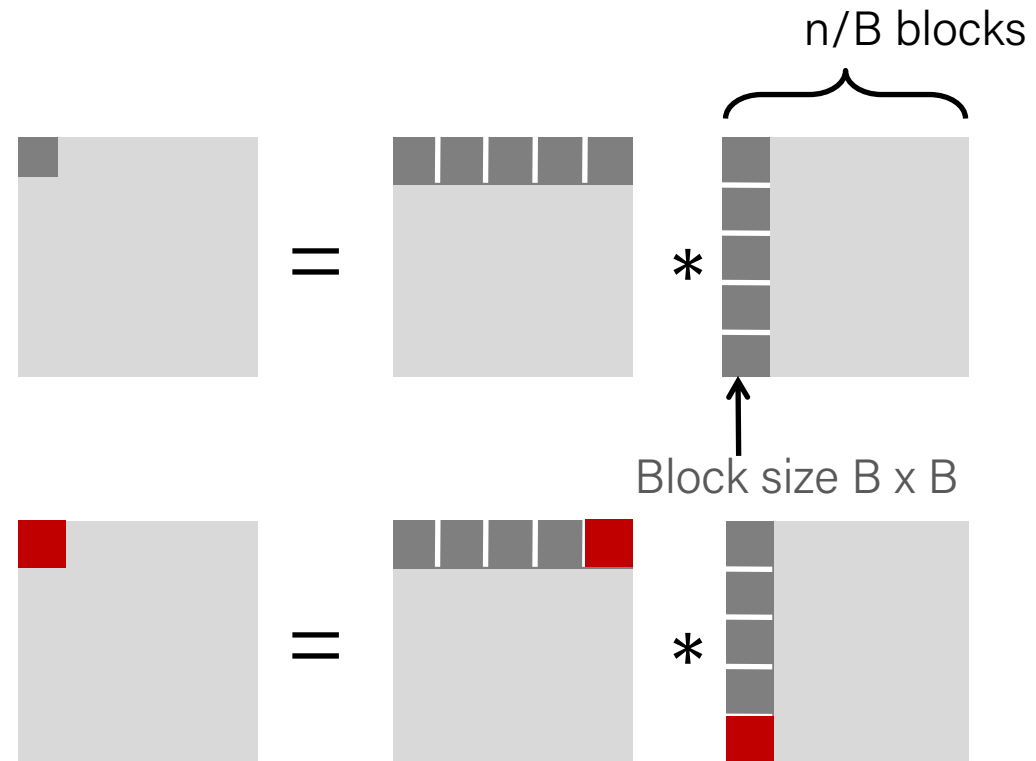


# Cache Miss Analysis


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

- **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

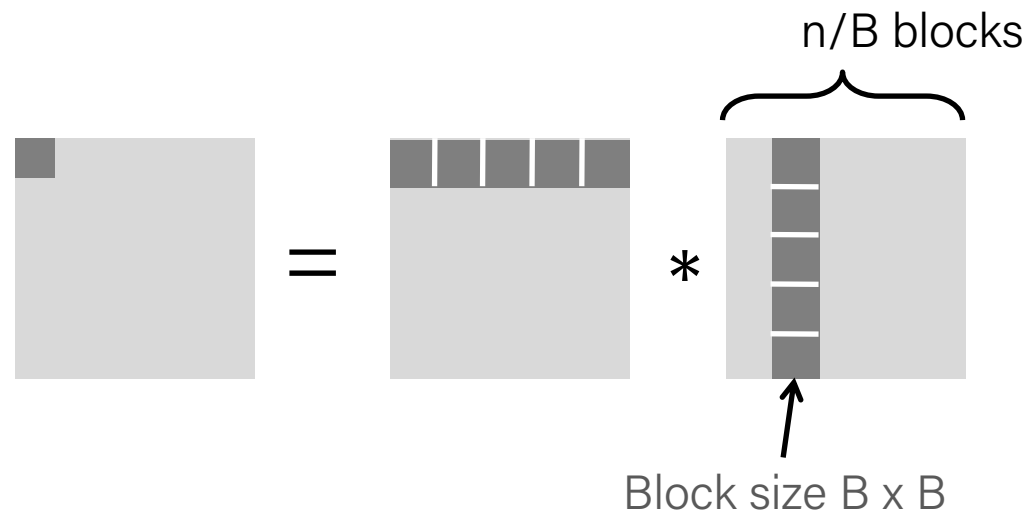
- Assume:
  - Cache block = 8 doubles
  - Cache size  $C \ll n$  (much smaller than  $n$ )
  - Three blocks  fit into cache:  $3B^2 < C$

- **Second (block) iteration:**

- Same as first iteration
  - $2n/B * B^2/8 = nB/4$

- Total misses:

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

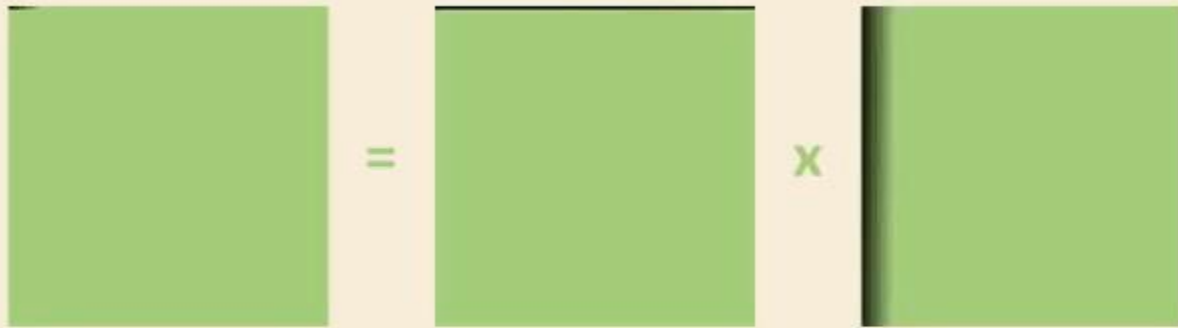


# Blocking Summary

- **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

# Naïve vs. Blocked Matrix Multiplication

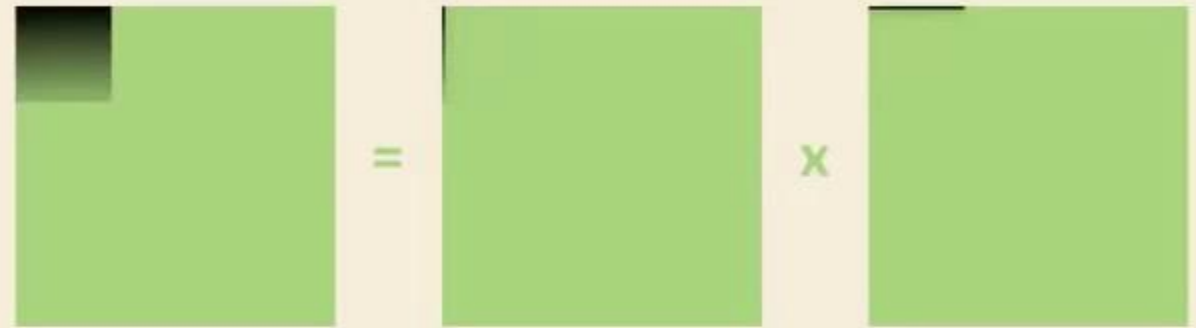
## Naïve Multiplication



Cache misses: 333

$\approx 1,020,000$  cache misses

## Blocked Multiplication



Cache misses: 333

$\approx 90,000$  cache misses

# Recap

- 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.

**Next time:** Debugging and Design



# Lecture Plan

- Writing cache-friendly codes
- Optimization
  - What is optimization?
  - GCC Optimization
  - Limitations of GCC Optimization
  - Caching revisited

# Optimization

- Optimization is the task of making your program faster or more efficient with space or time. Later you will learn about explorations of efficiency with Big-O notation!
- *Targeted, intentional* optimizations to alleviate bottlenecks can result in big gains. But it's important to only work to optimize where necessary.

# Optimization

Most of what you need to do with optimization can be summarized by:

- 1) If doing something seldom and only on small inputs, do whatever is simplest to code, understand, and debug
- 2) If doing things thing a lot, or on big inputs, make the primary algorithm's Big-O cost reasonable
- 3) Let gcc do its magic from there**
- 4) Optimize explicitly as a last resort

# GCC Optimization

- Today, we'll be comparing two levels of optimization in the gcc compiler:
  - `gcc -O0` // mostly just literal translation of C
  - `gcc -O2` // enable nearly all reasonable optimizations
  - (we use `-Og`, like `-O0` but with less needless use of the stack)
- There are other custom and more aggressive levels of optimization, e.g.:
  - `-O3` //more aggressive than `-O2`, trade size for speed
  - `-Os` //optimize for size
  - `-Ofast` //disregard standards compliance (!!)
- Exhaustive list of gcc optimization-related flags:
  - <https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html>

# Example: Matrix Multiplication

Here's a standard matrix multiply, a triply-nested for loop:

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

```
./mult          // -O0 (no optimization)  
matrix multiply 25^2: cycles    0.43M  
matrix multiply 50^2: cycles    3.02M  
matrix multiply 100^2: cycles   24.82M
```

```
./mult_opt      // -O2 (with optimization)  
matrix multiply 25^2: cycles    0.13M (opt)  
matrix multiply 50^2: cycles    0.66M (opt)  
matrix multiply 100^2: cycles   5.55M (opt)
```

# GCC Optimizations

- Constant Folding
- Common Sub-expression Elimination
- Dead Code
- Strength Reduction
- Code Motion
- Tail Recursion
- Loop Unrolling
- The Force

# GCC Optimizations

- Constant Folding
- Common Sub-expression Elimination
- Dead Code
- Strength Reduction
- Code Motion
- Tail Recursion
- Loop Unrolling
- ~~The Force~~

(may be not 🧐)

# GCC Optimizations

Optimizations may target one or more of:

- Static instruction count
- Dynamic instruction count
- Cycle count / execution time



# GCC Optimizations

- **Constant Folding**
- Common Sub-expression Elimination
- Dead Code
- Strength Reduction
- Code Motion
- Tail Recursion
- Loop Unrolling

# Constant Folding

**Constant Folding** pre-calculates constants at compile-time where possible.

```
int seconds = 60 * 60 * 24 * n_days;
```

What is the consequence of this for you as a programmer?

What should you do differently or the same knowing that compilers can do this for you?



# Constant Folding

```
int fold(int param) {  
    char arr[5];  
    int a = 0x107;  
    int b = a * sizeof(arr);  
    int c = sqrt(2.0);  
    return a * param + (a + 0x15 / c + strlen("Hello") * b - 0x37) / 4;  
}
```

# Constant Folding: Before (-O0)

0000000000400626 <fold>:

400626:	55	push	%rbp
400627:	53	push	%rbx
400628:	48 83 ec 08	sub	\$0x8,%rsp
40062c:	89 fd	mov	%edi,%ebp
40062e:	f2 0f 10 05 da 00 00	movsd	0xda(%rip),%xmm0
400635:	00		
400636:	e8 d5 fe ff ff	callq	400510 <sqrt@plt>
40063b:	f2 0f 2c c8	cvttsd2si	%xmm0,%ecx
40063f:	69 ed 07 01 00 00	imul	\$0x107,%ebp,%ebp
400645:	b8 15 00 00 00	mov	\$0x15,%eax
40064a:	99	cld	
40064b:	f7 f9	idiv	%ecx
40064d:	8d 98 07 01 00 00	lea	0x107(%rax),%ebx
400653:	bf 04 07 40 00	mov	\$0x400704,%edi
400658:	e8 93 fe ff ff	callq	4004f0 <strlen@plt>
40065d:	48 69 c0 23 05 00 00	imul	\$0x523,%rax,%rax
400664:	48 63 db	movslq	%ebx,%rbx
400667:	48 8d 44 18 c9	lea	-0x37(%rax,%rbx,1),%rax
40066c:	48 c1 e8 02	shr	\$0x2,%rax
400670:	01 e8	add	%ebp,%eax
400672:	48 83 c4 08	add	\$0x8,%rsp
400676:	5b	pop	%rbx
400677:	5d	pop	%rbp
400678:	c3	retq	

# Constant Folding: After (-O2)

00000000004004f0 <fold>:

4004f0: 69 c7 07 01 00 00

4004f6: 05 a5 06 00 00

4004fb: c3

4004fc: 0f 1f 40 00

imul \$0x107,%edi,%eax

add \$0x6a5,%eax

retq

nopl 0x0(%rax)

# GCC Optimizations

- Constant Folding
- **Common Sub-expression Elimination**
- Dead Code
- Strength Reduction
- Code Motion
- Tail Recursion
- Loop Unrolling

# Common Sub-Expression Elimination

**Common Sub-Expression Elimination** prevents the recalculation of the same thing many times by doing it once and saving the result.

```
int a = (param2 + 0x107);  
int b = param1 * (param2 + 0x107) + a;  
return a * (param2 + 0x107) + b * (param2 + 0x107);
```

# Common Sub-Expression Elimination

**Common Sub-Expression Elimination** prevents the recalculation of the same thing many times by doing it once and saving the result.

This optimization is done even at -O0!

```
int a = (param2 + 0x107);  
int b = param1 * (param2 + 0x107) + a;  
return a * (param2 + 0x107) + b * (param2 + 0x107);
```

00000000004004f0 <subexp>:

4004f0:	81 c6 07 01 00 00	add	\$0x107,%esi
4004f6:	0f af fe	imul	%esi,%edi
4004f9:	8d 04 77	lea	(%rdi,%rsi,2),%eax
4004fc:	0f af c6	imul	%esi,%eax
4004ff:	c3	retq	



# GCC Optimizations

- Constant Folding
- Common Sub-expression Elimination
- **Dead Code**
- Strength Reduction
- Code Motion
- Tail Recursion
- Loop Unrolling

# Dead Code

**Dead code elimination** removes code that doesn't serve a purpose:

```
if (param1 < param2 && param1 > param2) {  
    printf("This test can never be true!\n");  
}
```

```
// Empty for loop  
for (int i = 0; i < 1000; i++);
```

```
// If/else that does the same operation in both cases  
if (param1 == param2) {  
    param1++;  
} else {  
    param1++;  
}
```

```
// If/else that more trickily does the same operation in both cases  
if (param1 == 0) {  
    return 0;  
} else {  
    return param1;  
}
```

# Dead Code: Before (-O0)

00000000004004d6 <dead\_code>:

4004d6: b8 00 00 00 00

4004db: eb 03

4004dd: 83 c0 01

4004e0: 3d e7 03 00 00

4004e5: 7e f6

4004e7: 39 f7

4004e9: 75 05

4004eb: 8d 47 01

4004ee: eb 03

4004f0: 8d 47 01

4004f3: f3 c3

mov \$0x0,%eax

jmp 4004e0 <dead\_code+0xa>

add \$0x1,%eax

cmp \$0x3e7,%eax

jle 4004dd <dead\_code+0x7>

cmp %esi,%edi

jne 4004f0 <dead\_code+0x1a>

lea 0x1(%rdi),%eax

jmp 4004f3 <dead\_code+0x1d>

lea 0x1(%rdi),%eax

repz retq

# Dead Code: After (-O2)

00000000004004f0 <dead\_code>:

4004f0:	8d 47 01	lea	0x1(%rdi),%eax
4004f3:	c3	retq	
4004f4:	66 2e 0f 1f 84 00 00	nopw	%cs:0x0(%rax,%rax,1)
4004fb:	00 00 00		
4004fe:	66 90	xchg	%ax,%ax

# GCC Optimizations

- Constant Folding
- Common Sub-expression Elimination
- Dead Code
- **Strength Reduction**
- Code Motion
- Tail Recursion
- Loop Unrolling

# Strength Reduction

**Strength reduction** changes divide to multiply, multiply to add/shift, and mod to AND to avoid using instructions that cost many cycles (multiply and divide).

```
int a = param2 * 32;  
int b = a * 7;  
int c = b / 3;  
int d = param2 % 2;  
  
for (int i = 0; i <= param2; i++) {  
    c += param1[i] + 0x107 * i;  
}  
return c + d;
```

# Strength Reduction: After (-O3)

```
unsigned udiv19(unsigned arg) {  
    return arg / 19;  
}
```

```
udiv19(unsigned int):  
    mov     eax, edi  
    mov     edx, 2938661835  
    imul    rax, rdx  
    shr     rax, 32  
    sub     edi, eax  
    shr     edi  
    add     eax, edi  
    shr     eax, 4  
    ret
```

<https://godbolt.org/z/Wq8ra3>

What really happens here?



$$a \cdot \frac{1}{19} \approx \frac{a \cdot \frac{2938661835}{2^{32}} + \frac{a - a \cdot \frac{2938661835}{2^{32}}}{2^1}}{2^4}$$

$$a \cdot \frac{1}{19} \approx (a \cdot 2938661835 \cdot 2^{-32} + (a - a \cdot 2938661835 \cdot 2^{-32}) \cdot 2^{-1}) \cdot 2^{-4}$$

$$a \cdot \frac{1}{19} \approx a \cdot \frac{7233629131}{137438953472}$$

# GCC Optimizations

- Constant Folding
- Common Sub-expression Elimination
- Dead Code
- Strength Reduction
- **Code Motion**
- Tail Recursion
- Loop Unrolling



# Code Motion

**Code motion** moves code outside of a loop if possible.

```
for (int i = 0; i < n; i++) {  
    sum += arr[i] + foo * (bar + 3);  
}
```

Common subexpression elimination deals with expressions that appear multiple times in the code. Here, the expression appears once, but is calculated each loop iteration.

# GCC Optimizations

- Constant Folding
- Common Sub-expression Elimination
- Dead Code
- Strength Reduction
- Code Motion
- **Tail Recursion**
- Loop Unrolling

# Tail Recursion

**Tail recursion** is an example of where GCC can identify recursive patterns that can be more efficiently implemented iteratively.

```
long factorial(int n) {  
    if (n <= 1) {  
        return 1;  
    }  
    else return n * factorial(n - 1);  
}
```

# GCC Optimizations

- Constant Folding
- Common Sub-expression Elimination
- Dead Code
- Strength Reduction
- Code Motion
- Tail Recursion
- **Loop Unrolling**

# Loop Unrolling

**Loop Unrolling:** Do **n** loop iterations' worth of work per actual loop iteration, so we save ourselves from doing the loop overhead (test and jump) every time, and instead incur overhead only every n-th time.

```
for (int i = 0; i <= n - 4; i += 4) {  
    sum += arr[i];  
    sum += arr[i + 1];  
    sum += arr[i + 2];  
    sum += arr[i + 3];  
} // after the loop handle any leftovers
```

# Limitations of GCC Optimization

GCC can't optimize everything! You ultimately may know more than GCC does.

```
int char_sum(char *s) {  
    int sum = 0;  
    for (size_t i = 0; i < strlen(s); i++) {  
        sum += s[i];  
    }  
    return sum;  
}
```

What is the bottleneck? **strlen called for every character**  
What can GCC do? **code motion – pull strlen out of loop**

# Limitations of GCC Optimization

GCC can't optimize everything! You ultimately may know more than GCC does.

```
void lower1(char *s) {  
    for (size_t i = 0; i < strlen(s); i++) {  
        if (s[i] >= 'A' && s[i] <= 'Z') {  
            s[i] -= ('A' - 'a');  
        }  
    }  
}
```

What is the bottleneck?

What can GCC do?

**strlen called for every character**

**nothing! s is changing, so GCC doesn't know if length is constant across iterations. But we know its length doesn't change.**

# Optimizing Your Code

- Explore various optimizations you can make to your code to reduce instruction count and runtime.
  - More efficient Big-O for your algorithms
  - Explore other ways to reduce instruction count
    - Look for hotspots using `callgrind`
    - Optimize using `-O2`
    - And more...



# Optimizing Your Code

- Explore various optimizations you can make to your code to reduce instruction count and runtime.
  - More efficient Big-O for your algorithms
  - Explore other ways to reduce instruction count
    - Look for hotspots using `callgrind`
    - Optimize using `-O2`
    - And more...

# Compiler Optimizations

*Why not always just compile with -O2?*

- Difficult to debug optimized executables – only optimize when complete
- Optimizations may not *always* improve your program. The compiler does its best, but may not work, or slow things down, etc. Experiment to see what works best!

*Why should we bother saving repeated calculations in variables if the compiler has common subexpression elimination?*

- The compiler may not always be able to optimize every instance. Plus, it can help reduce redundancy!

# Recap

- Writing cache-friendly code
- Optimization

**Next time:** Debugging and design