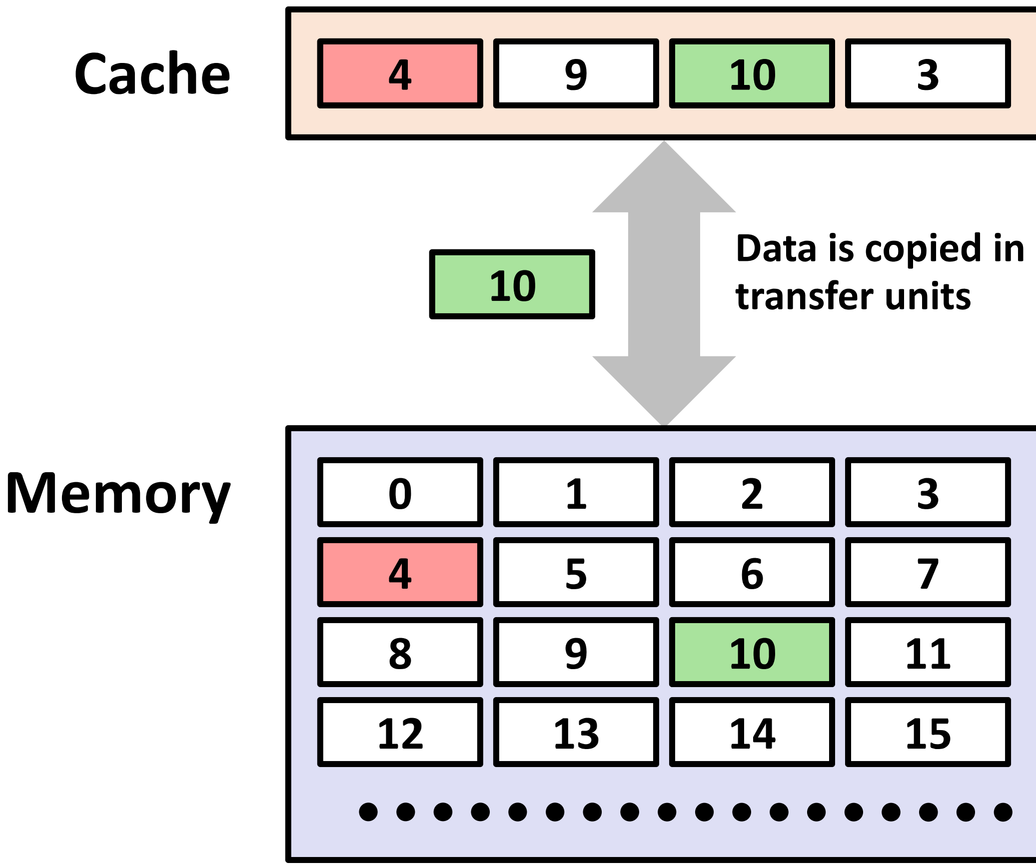
Cache Memories

Lecture, Oct. 30, 2018

General Cache Concept



Smaller, faster, more expensive memory caches a subset of the blocks

Data is copied in block-sized transfer units

Larger, slower, cheaper memory viewed as partitioned into "blocks"

C and cache misses (1)

```
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];
    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?

C 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?

3

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;
```

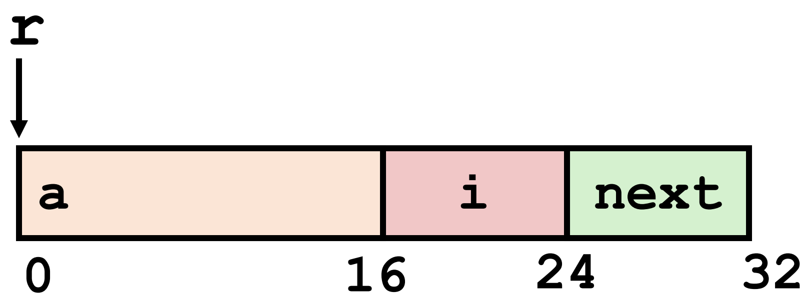
Assume everything but `items` is kept in registers (and the compiler does not do anything funny).

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

6

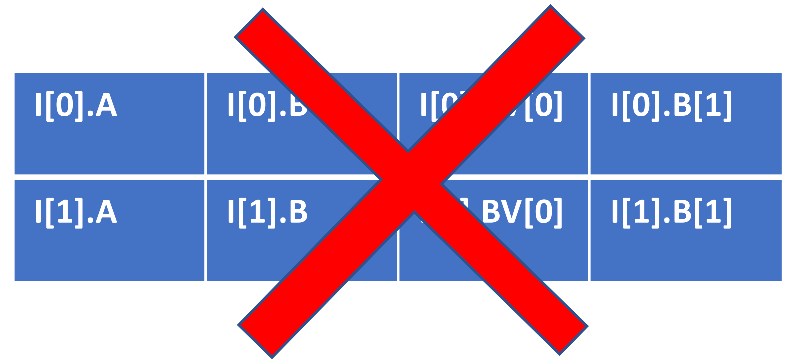
Structure Representation

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



C and cache misses (3)

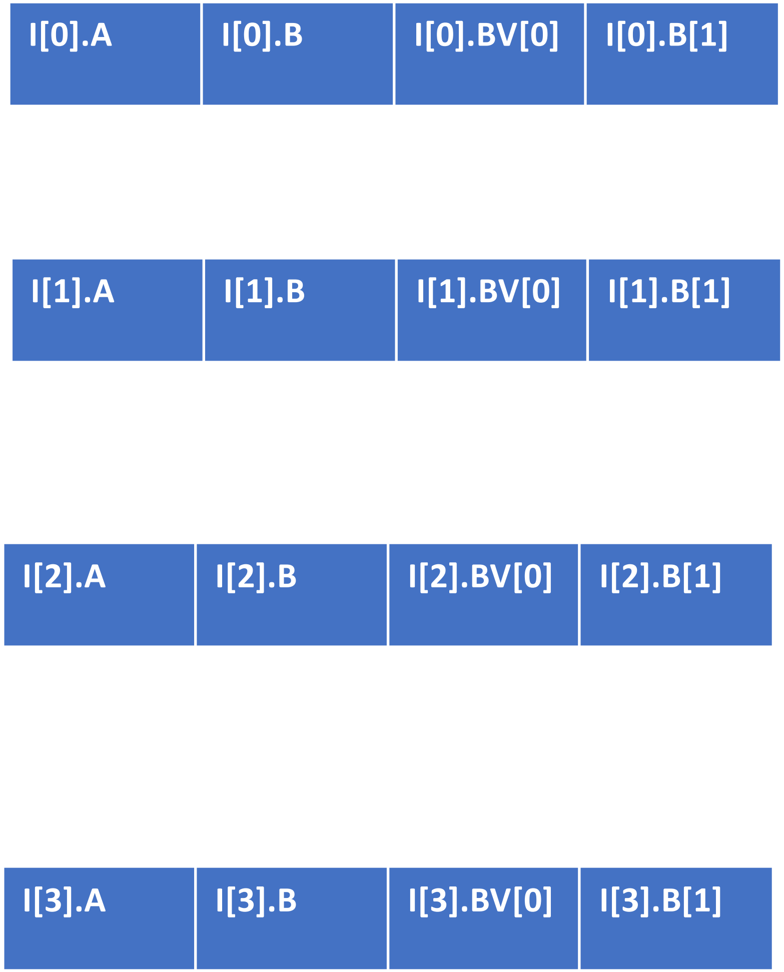
```
typedef struct {
    int a_value, b_value;
    int boring_values[126];
} item;
item items[8]; // 4 KB array
int a_sum = 0, b_sum = 0;
for (int i = 0; i < 8; ++i)
    a_sum += items[i].a_value;
for (int i = 0; i < 8; ++i)
    b_sum += items[i].b_value;
```



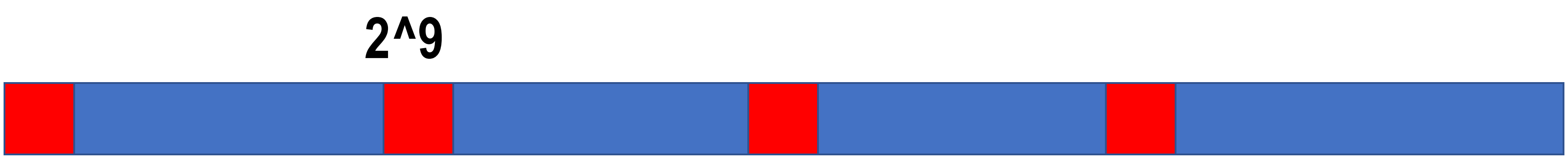
Assume everything but `items` is kept in registers (and the compiler does not do anything funny).

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

6



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



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

Inner loop analysis

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]

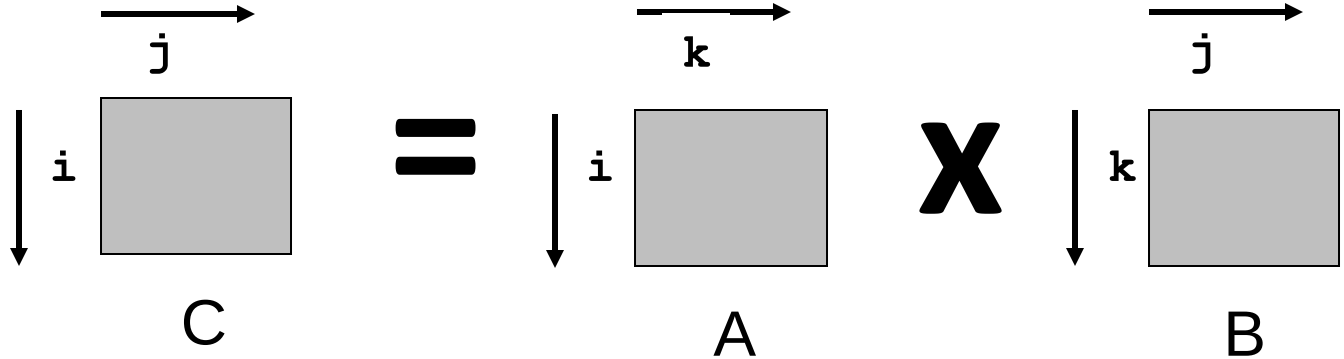
Matrix Multiplication Refresher

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix} \begin{bmatrix} 7 & 8 \\ 9 & 10 \\ 11 & 12 \end{bmatrix} = \begin{bmatrix} 58 & \\ & \end{bmatrix}$$

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

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



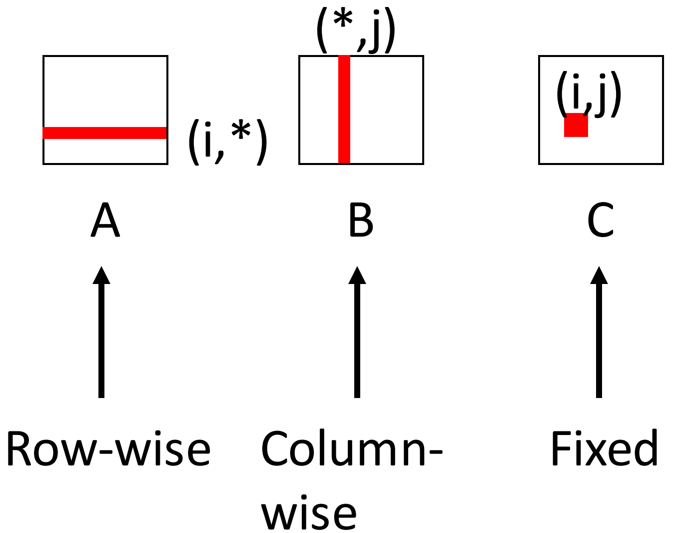
Layout of C Arrays in Memory (review)

- 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:



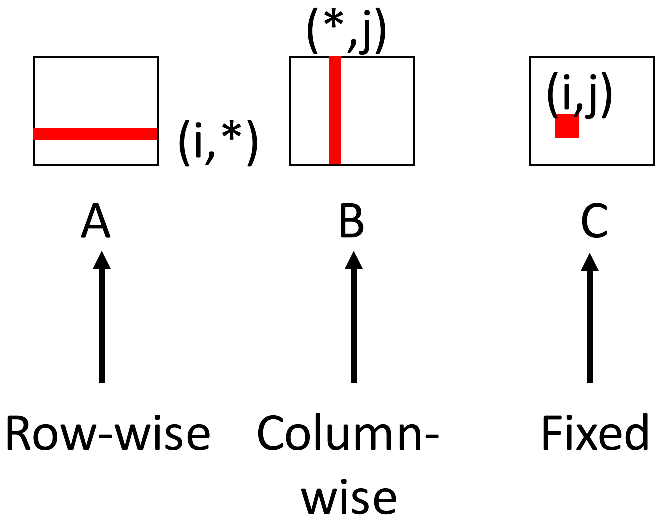
Misses per inner loop iteration:

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:

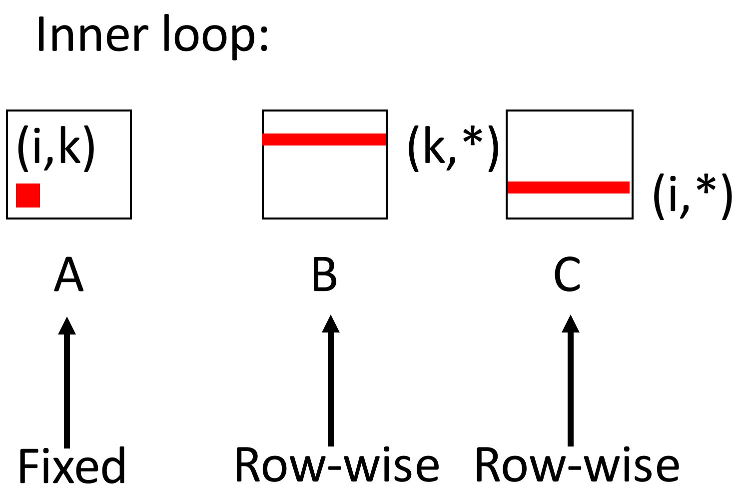


Misses per inner loop iteration:

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
```

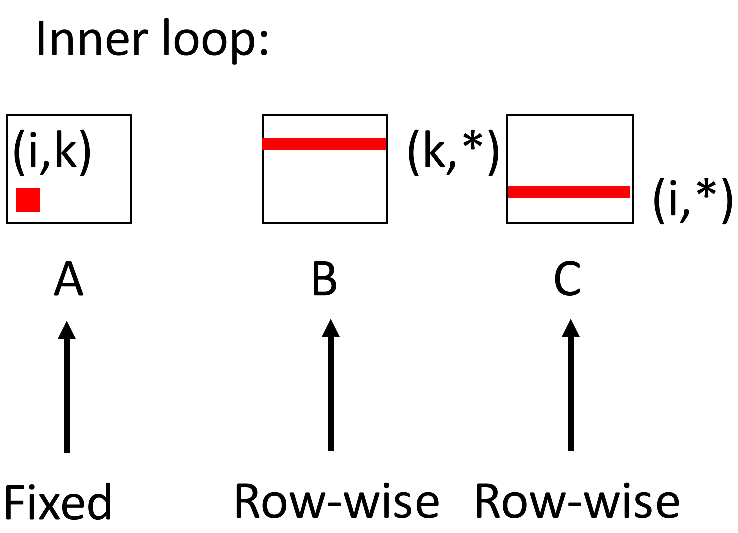


<u>Misses per inner loop iteration:</u>

<u>A</u>	<u>B</u>	<u>C</u>
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:

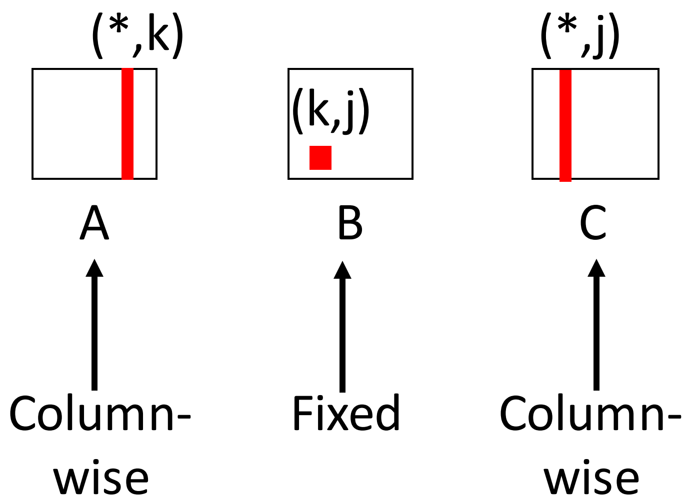
A	B	C
0.0	0.25	0.25

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:



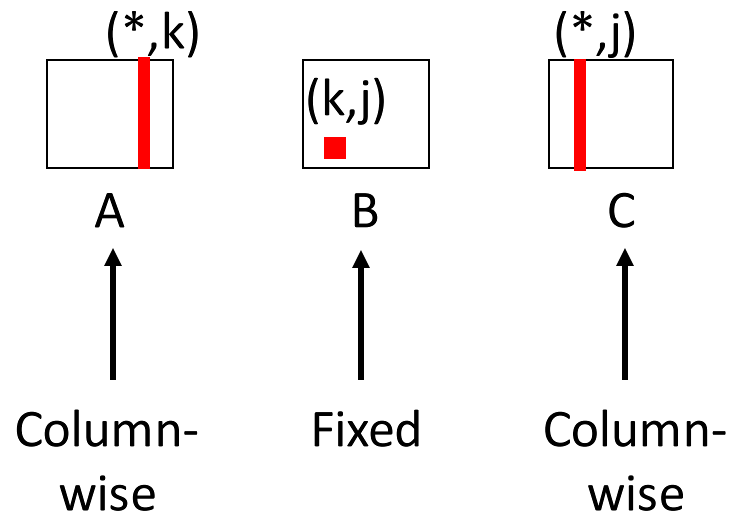
<u>Misses per inner loop iteration:</u>

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**

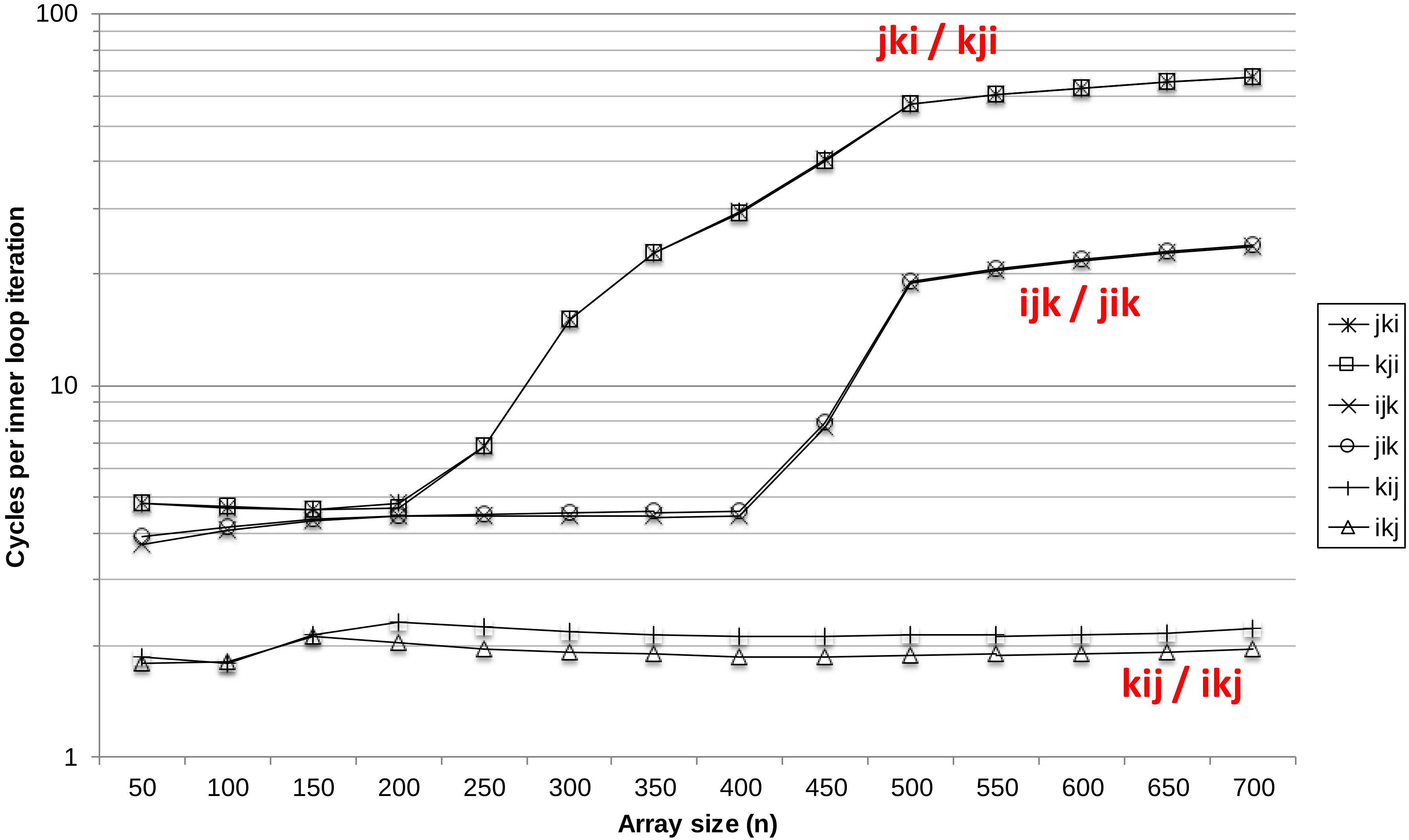
```
for (k=0; k<n; k++) {
 for (i=0; i<n; i++) {
  r = a[i][k];
  for (j=0; j<n; j++){
   c[i][j] += r * b[k][j];}
 }
}
```

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**

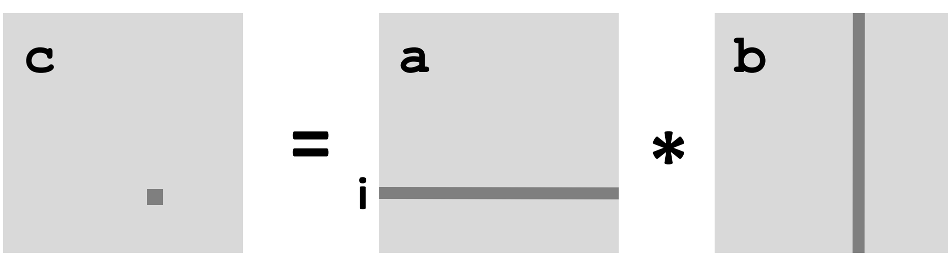
Core i7 Matrix Multiply Performance



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];
}
```

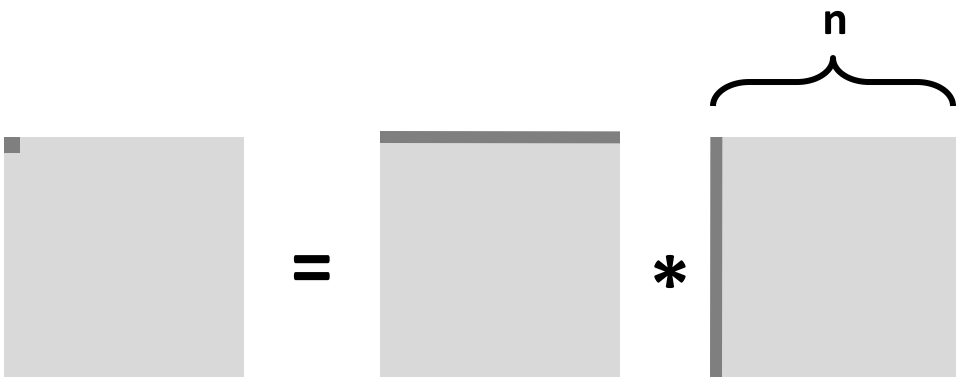


Cache Miss Analysis

- Assume:
  - Matrix elements are doubles
  - 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)

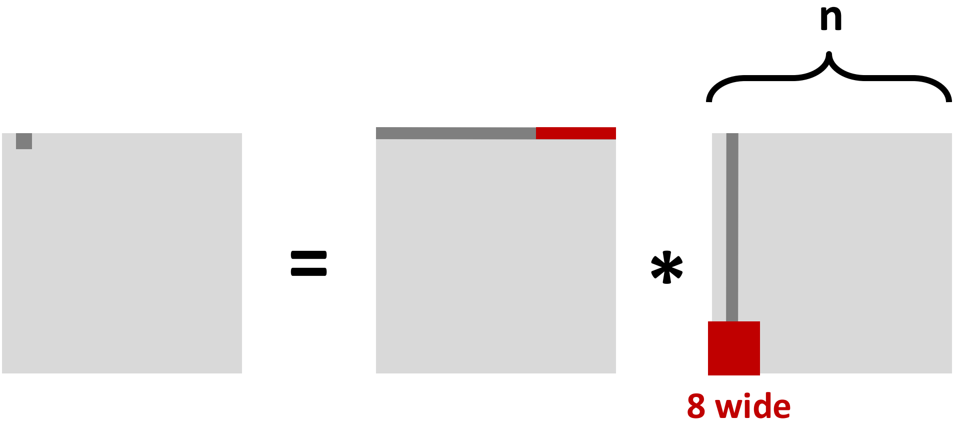


Cache Miss Analysis

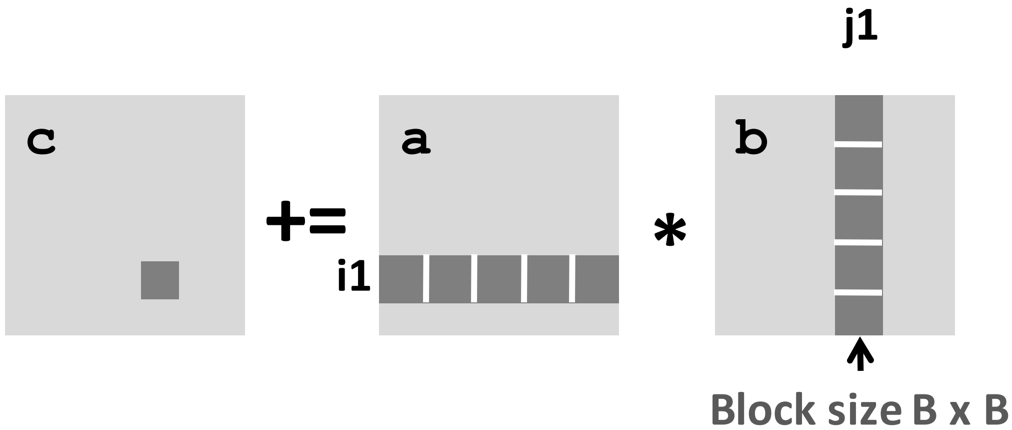
- Assume:
  - Matrix elements are doubles
  - Cache block = 8 doubles
  - Cache size C << n (much smaller than n)

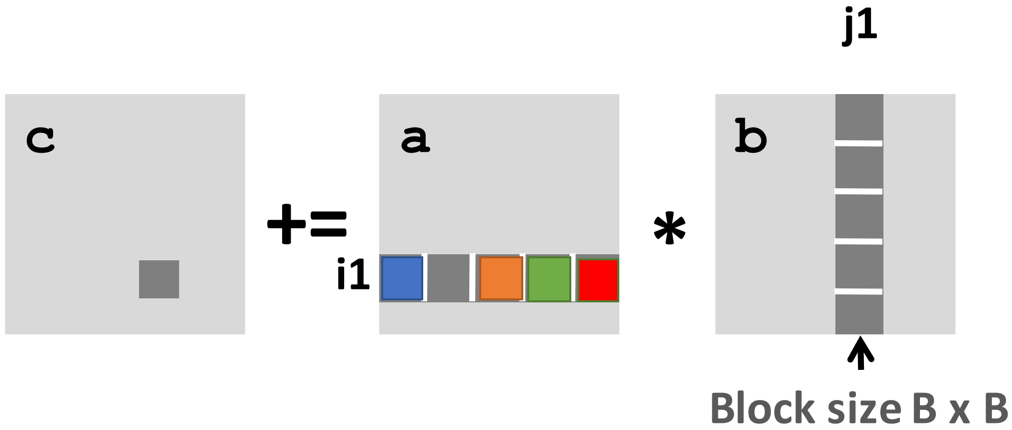
- Second iteration:
  - Again:
    n/8 + n = 9n/8 misses

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

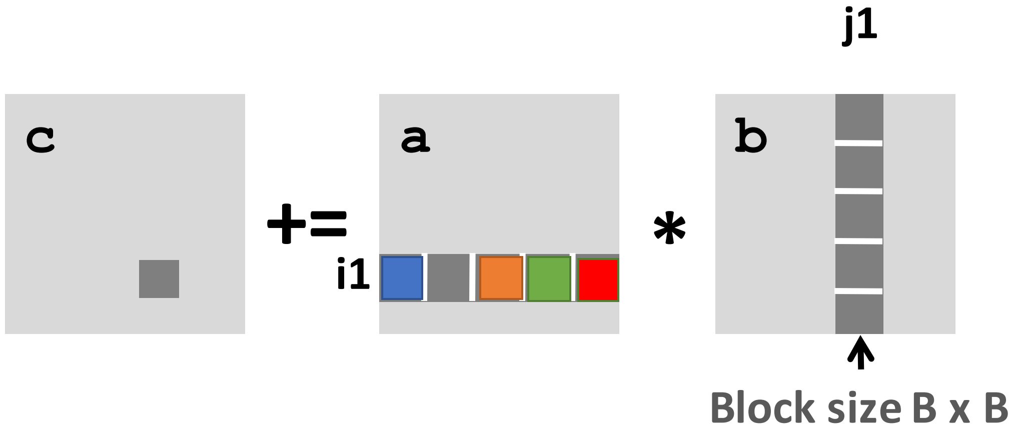


Blocked Matrix Multiplication



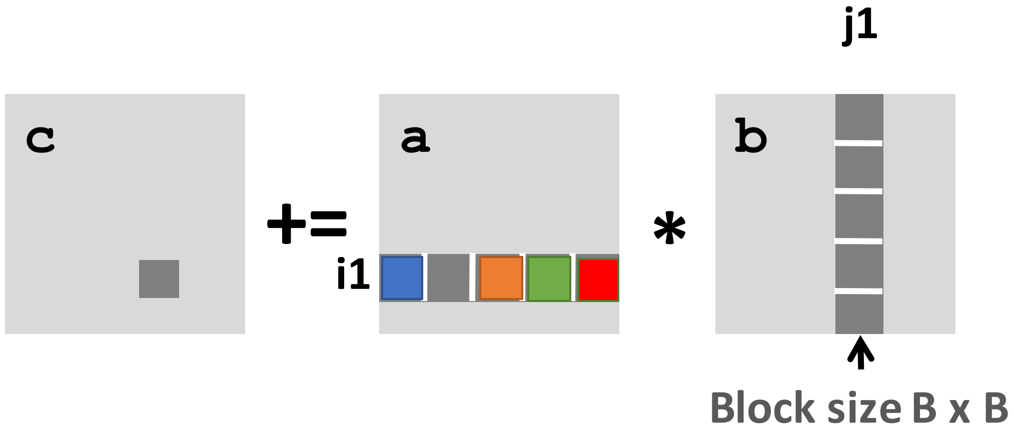


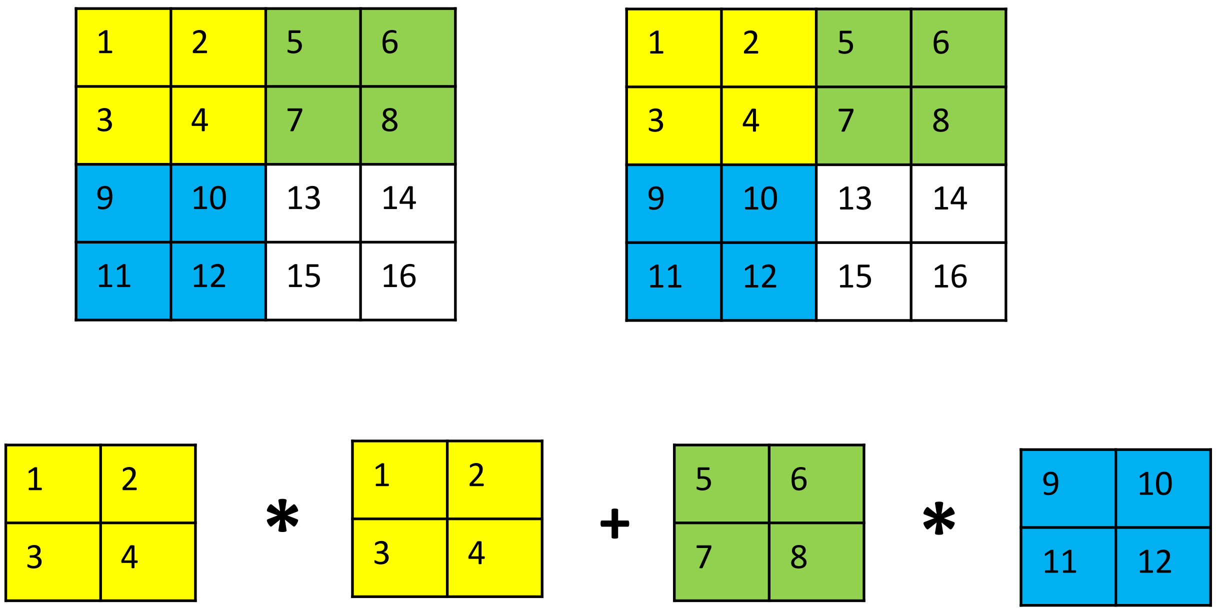
j1
c
+=
a
i1
*
b
Block size B x B

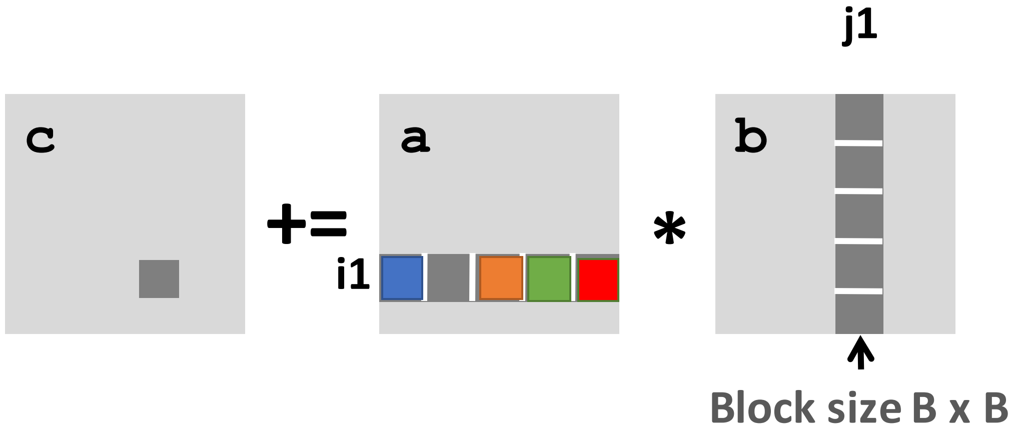


1	2	5	6
3	4	7	8
9	10	13	14
11	12	15	16

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


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

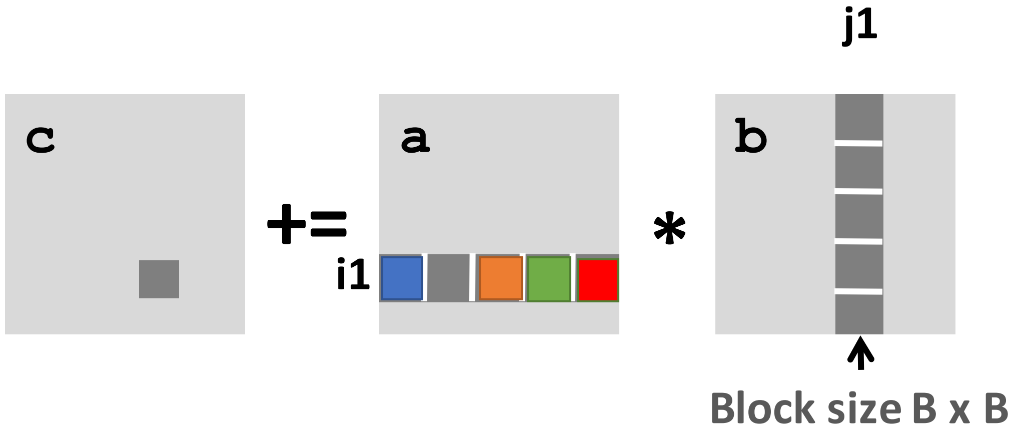

1 2 5 6
3 4 7 8
9 10 13 14
11 12 15 16
1 2 5 6
3 4 7 8
9 10 13 14
11 12 15 16
1 2
3 4
*
1 2
3 4
+
5 6
7 8
*
9 10
11 12



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

$$\begin{bmatrix} 118 & 132 \\ 166 & 188 \end{bmatrix} = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} * \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} + \begin{bmatrix} 5 & 6 \\ 7 & 8 \end{bmatrix} * \begin{bmatrix} 9 & 10 \\ 11 & 12 \end{bmatrix}$$



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

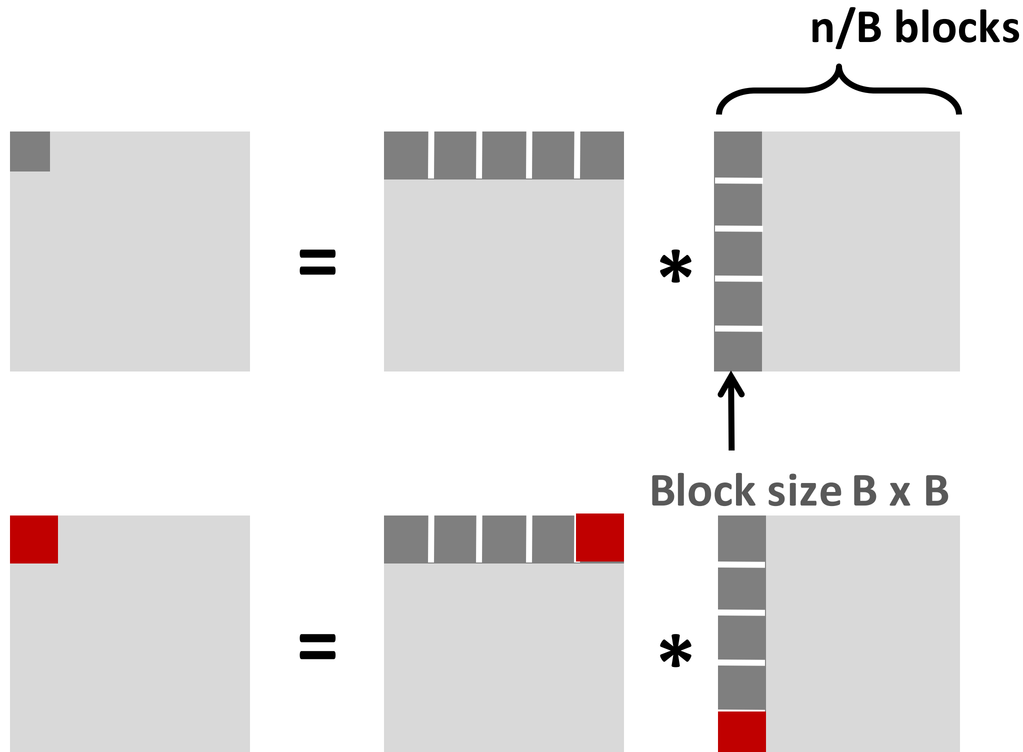
$$\begin{bmatrix} 118 & 132 \\ 166 & 188 \end{bmatrix} = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} * \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} + \begin{bmatrix} 5 & 6 \\ 7 & 8 \end{bmatrix} * \begin{bmatrix} 9 & 10 \\ 11 & 12 \end{bmatrix}$$

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)

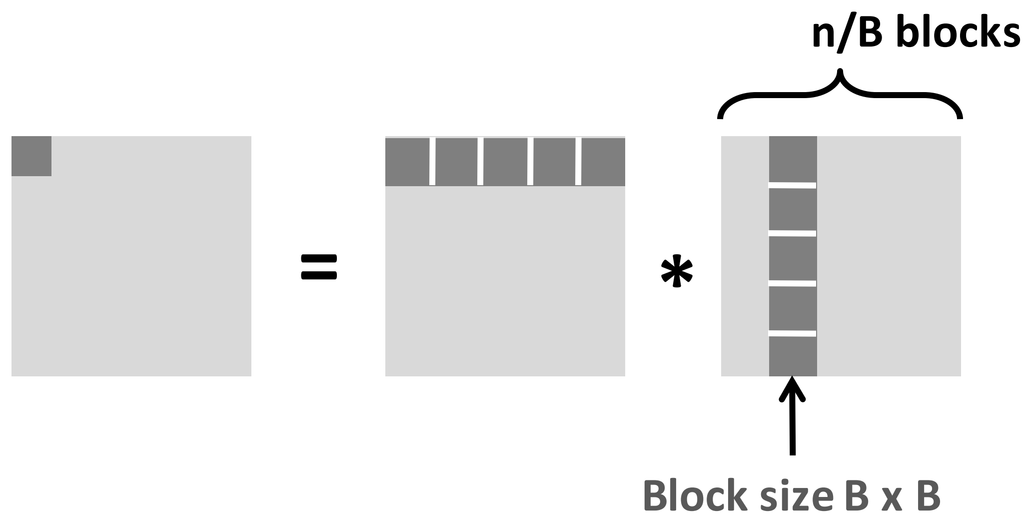


Cache Miss Analysis

- 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

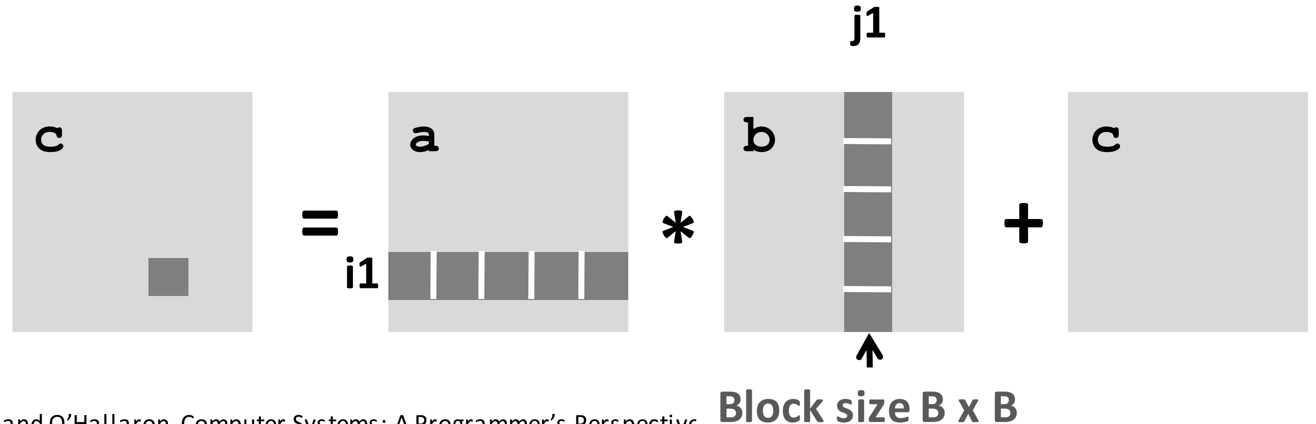
- You can write your programs to exploit this!
  - Focus on the inner loops, where bulk of computations and memory accesses occur.
  - Try to maximize spatial locality by reading data objects with sequentially with stride 1.
  - Try to maximize temporal locality by using a data object as often as possible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



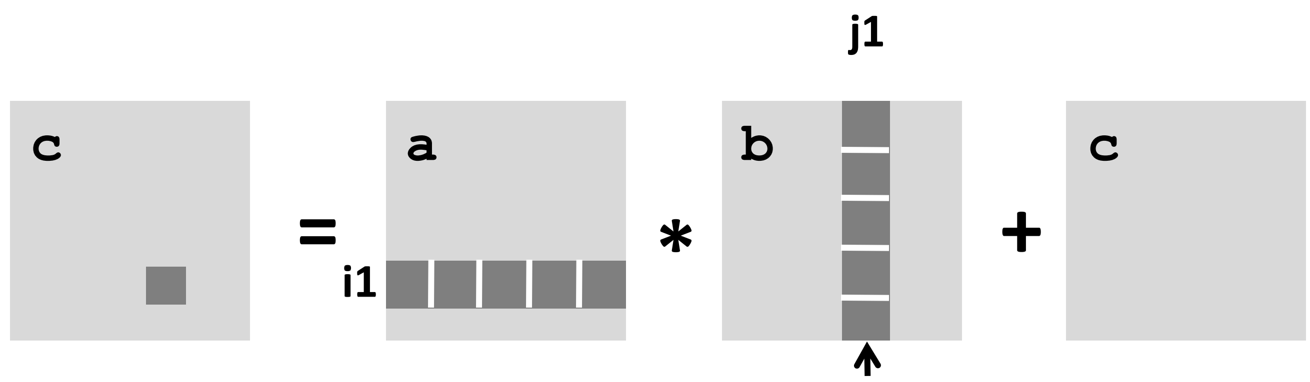
Program Optimization

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



Block size B x B

Compiler Optimizations

Optimizing Compilers

- Provide efficient mapping of program to machine
  - register allocation
  - code selection and ordering (scheduling)
  - dead code elimination
  - eliminating minor inefficiencies
- Don't (usually) improve asymptotic efficiency
  - up to programmer to select best overall algorithm
  - big-O savings are (often) more important than constant factors
    - but constant factors also matter
- Have difficulty overcoming "optimization blockers"
  - potential memory aliasing
  - potential procedure side-effects

Limitations of Optimizing Compilers

- Operate under fundamental constraint
  - Must not cause any change in program behavior
    - Except, possibly when program making use of nonstandard language features
  - Often prevents it from making optimizations that would only affect behavior under edge conditions.

- Most analysis is performed only within procedures
  - Whole-program analysis is too expensive in most cases
  - Newer versions of GCC do interprocedural analysis within individual files
    - But, not between code in different files

- Most analysis is based only on *static* information
  - Compiler has difficulty anticipating run-time inputs

- When in doubt, the compiler must be conservative

example assembly (unoptimized)

```
long sum(long *A, int N) {
    long result = 0;
    for (int i = 0; i < N; ++i)
        result += A[i];
    return result;
}
```

```
sum:    ...
the_loop:
        ...
        leaq    0(,%rax,8), %rdx// offset ← i * 8
        movq    −24(%rbp), %rax // get A from stack
        addq    %rdx, %rax      // add offset
        movq    (%rax), %rax    // get *(A+offset)
        addq    %rax, −8(%rbp)  // add to sum, on stack
        addl    $1, −12(%rbp)   // increment i
condition:
        movl    −12(%rbp), %eax
        cmpl    −28(%rbp), %eax
        jl      the_loop
```

21

example assembly (gcc 5.4 -Os)

```
long sum(long *A, int N) {
    long result = 0;
    for (int i = 0; i < N; ++i)
        result += A[i];
    return result;
}
```

```
sum:
        xorl    %edx, %edx
        xorl    %eax, %eax
the_loop:
        cmpl    %edx, %esi
        jle     done
        addq    (%rdi,%rdx,8), %rax
        incq    %rdx
        jmp     the_loop
done:
        ret
```

%edx holds i

22

example assembly (gcc 5.4 -O2)

```
long sum(long *A, int N) {
    long result = 0;
    for (int i = 0; i < N; ++i)
        result += A[i];
    return result;
}
sum:
        testl   %esi, %esi
        jle     return_zero
        leal    −1(%rsi), %eax
        leaq    8(%rdi,%rax,8), %rdx // rdx=end of A
        xorl    %eax, %eax
the_loop:
        addq    (%rdi), %rax // add to sum
        addq    $8, %rdi     // advance pointer
        cmpq    %rdx, %rdi
        jne     the_loop
        rep ret
return_zero:    ...
```

```
long *p = A;
long * end = A + N-1;
while( p!= end){
       result+ = p;
       p++;
}
```

Optimization removes i
Makes a more efficient compare
Because were are now testing for equivalence so we can use test.

Also makes the address calculation simpler

23

Some categories of optimizations compilers are good at

Generally Useful Optimizations

- Optimizations that you or the compiler should do regardless of processor / compiler

- Code Motion
  - Reduce frequency with which computation performed
    - If it will always produce same result
    - Especially moving code out of loop

```
void set_row(double *a, double *b,
   long i, long n)
{
    long j;
    for (j = 0; j < n; j++)
         a[n*i+j] = b[j];
}
```

→

```
    long j;
    int ni = n*i;
    for (j = 0; j < n; j++)
         a[ni+j] = b[j];
```

Reduction in Strength

- Replace costly operation with simpler one
- Shift, add instead of multiply or divide

```
16*x   -->    x << 4
```

  - Depends on cost of multiply or divide instruction
    - On Intel Nehalem, integer multiply requires 3 CPU cycles
    - https://www.agner.org/optimize/instruction_tables.pdf
- Recognize sequence of products

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

→

```
int ni = 0;
for (i = 0; i < n; i++) {
  for (j = 0; j < n; j++)
    a[ni + j] = b[j];
  ni += n;
}
```

We can replace multiple operation with and add

Share Common Subexpressions

- Reuse portions of expressions
- GCC will do this with –O1

```
/* Sum neighbors of i,j */
up =    val[(i-1)*n + j  ];
down =  val[(i+1)*n + j  ];
left =  val[i*n     + j-1];
right = val[i*n     + j+1];
sum = up + down + left + right;
```

```
long inj = i*n + j;
up =    val[inj - n];
down =  val[inj + n];
left =  val[inj - 1];
right = val[inj + 1];
sum = up + down + left + right;
```

3 multiplications: i*n, (i–1)*n, (i+1)*n

1 multiplication: i*n

```
leaq   1(%rsi), %rax   # i+1
leaq   -1(%rsi), %r8   # i-1
imulq  %rcx, %rsi      # i*n
imulq  %rcx, %rax      # (i+1)*n
imulq  %rcx, %r8       # (i-1)*n
addq   %rdx, %rsi      # i*n+j
addq   %rdx, %rax      # (i+1)*n+j
addq   %rdx, %r8       # (i-1)*n+j
```

```
imulq    %rcx, %rsi   # i*n
addq     %rdx, %rsi   # i*n+j
movq     %rsi, %rax   # i*n+j
subq     %rcx, %rax   # i*n+j-n
leaq     (%rsi,%rcx), %rcx # i*n+j+n
```

Share Common Subexpressions

- Reuse portions of expressions
- GCC will do this with –O1

Distribute the N

```
/* Sum neighbors of i,j */
up =    val[(i-1)*n + j  ];
down =  val[(i+1)*n + j  ];
left =  val[i*n     + j-1];
right = val[i*n     + j+1];
sum = up + down + left + right;
```

3 multiplications: i*n, (i–1)*n, (i+1)*n

```
long inj = i*n + j;
up =    val[inj - n];
down =  val[inj + n];
left =  val[inj - 1];
right = val[inj + 1];
sum = up + down + left + right;
```

1 multiplication: i*n

Write Compiler Friendly code: Times when the compilers need help

Optimization Blocker #1: Procedure Calls

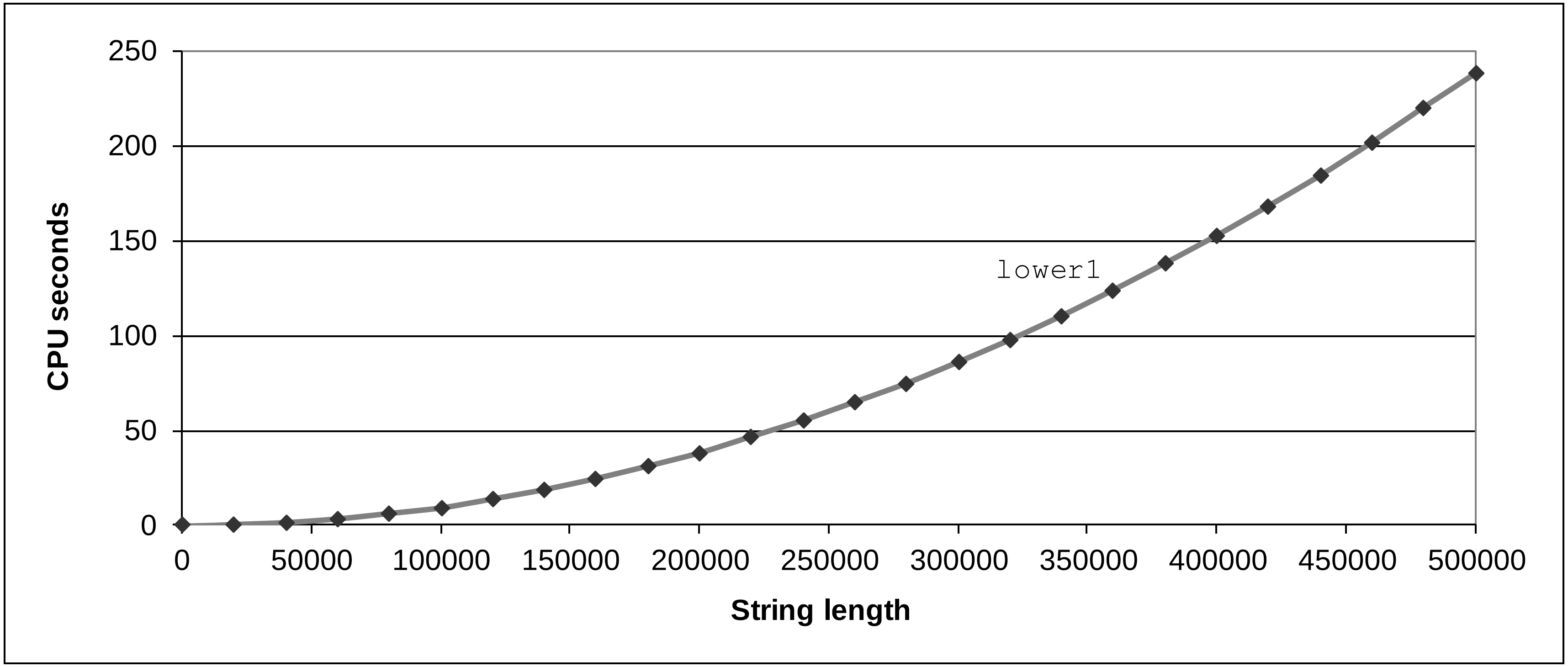
- Procedure to Convert String to Lower Case

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

A = 65
Z = 90
a = 97
z = 122

Lower Case Conversion Performance

- Time quadruples when double string length
- Quadratic performance



Convert Loop To Goto Form

```
void lower(char *s)
{
   size_t i = 0;
   if (i >= strlen(s))
     goto done;
 loop:
   if (s[i] >= 'A' && s[i] <= 'Z')
       s[i] -= ('A' - 'a');
   i++;
   if (i < strlen(s))
     goto loop;
 done:
}
```

- `strlen` executed every iteration

Calling Strlen

```
/* My version of strlen */
size_t strlen(const char *s)
{
    size_t length = 0;
    while (*s != '\0') {
       s++;
       length++;
    }
    return length;
}
```

- Strlen performance
  - Only way to determine length of string is to scan its entire length, looking for null character.
- Overall performance, string of length N
  - N calls to strlen
  - Require times N, N-1, N-2, …, 1
  - Overall $O(N^2)$ performance

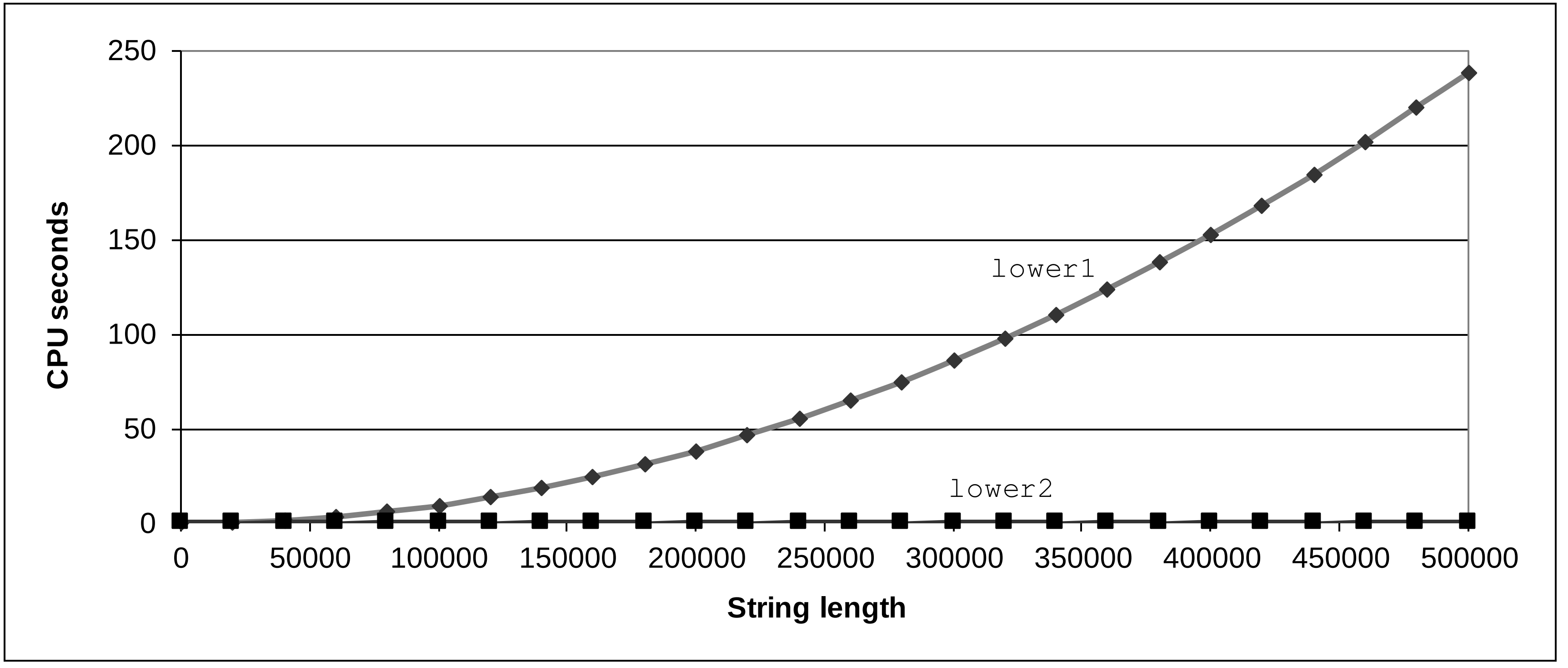
Improving Performance

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

- Move call to `strlen` outside of loop
- Since result does not change from one iteration to another
- Form of code motion

Lower Case Conversion Performance

- Time doubles when double string length
- Linear performance of lower2



Optimization Blocker: Procedure Calls

- *Why couldn't compiler move* `strlen` *out of inner loop?*
  - Procedure may have side effects
    - Alters global state each time called
  - Function may not return same value for given arguments
    - Depends on other parts of global state
    - Procedure `lower` could interact with `strlen`
- Warning:
  - Compiler treats procedure call as a black box
- Remedies:
  - Do your own code motion

```
size_t lencnt = 0;
size_t strlen(const char *s)
{
    size_t length = 0;
    while (*s != '\0') {
       s++; length++;
    }
    lencnt += length;
    return length;
}
```

loop with a function call

```
int addWithLimit(int x, int y) {
    int total = x + y;
    if (total > 10000)
        return 10000;
    else
        return total;
}
...
int sum(int *array, int n) {
    int sum = 0;
    for (int i = 0; i < n; i++)
        sum = addWithLimit(sum, array[i]);
    return sum;
}
```

3

function call assembly

```
movl (%rbx), %esi // mov array[i]
movl %eax, %edi   // mov sum
call addWithLimit
```

extra instructions executed: two moves, a call, and a ret

manual inlining

```
int sum(int *array, int n) {
    int sum = 0;
    for (int i = 0; i < n; i++) {
        sum = sum + array[i];
        if (sum > 10000)
            sum = 10000;
    }
    return sum;
}
```

compiler inlining

compilers will inline, but...

will usually avoid making code much bigger

- heuristic: inline if function is small enough
- heuristic: inline if called exactly once

will usually not inline across .o files

some compilers allow hints to say "please inline/do not inline this function"

Memory Aliasing

aliasing

```
void twiddle(long *px, long *py) {
    *px += *py;
    *px += *py;
}
```

the compiler **cannot** generate this:

```
twiddle: // BROKEN // %rsi = px, %rdi = py
        movq    (%rdi), %rax // rax ← *py
        addq    %rax, %rax   // rax ← 2 * *py
        addq    %rax, (%rsi) // *px ← 2 * *py
        ret
```

27

aliasing problem

```
void twiddle(long *px, long *py) {
    *px += *py;
    *px += *py;
    // NOT the same as *px += 2 * *py;
}
...
    long x = 1;
    twiddle(&x, &x);
    // result should be 4, not 3
```

```
twiddle: // BROKEN // %rsi = px, %rdi = py
        movq    (%rdi), %rax // rax ← *py
        addq    %rax, %rax   // rax ← 2 * *py
        addq    %rax, (%rsi) // *px ← 2 * *py
        ret
```

Another example of Aliasing

Memory Aliasing

```
/* Sum rows is of n X n matrix a
   and store in vector b  */
void sum_rows1(double *a, double *b, long n) {
    long i, j;
    for (i = 0; i < n; i++) {
        b[i] = 0;
        for (j = 0; j < n; j++)
            b[i] += a[i*n + j];
    }
}
```

- Code updates `b[i]` on every iteration
- Why couldn't compiler optimize this away?

Memory Aliasing

```
/* Sum rows is of n X n matrix a
   and store in vector b  */
void sum_rows1(double *a, double *b, long n) {
    long i, j;
    for (i = 0; i < n; i++) {
        b[i] = 0;
        for (j = 0; j < n; j++)
            b[i] += a[i*n + j];
    }
}
```

```
double A[9] =
  { 0,   1,   2,
    4,   8,  16},
   32,  64, 128};

double B[3] = A+3;

sum_rows1(A, B, 3);
```

Value of B:

```
init:   [4, 8, 16]
```

```
i = 0: [3, 8, 16]
```

```
i = 1: [3, 22, 16]
```

```
i = 2: [3, 22, 224]
```

- Code updates `b[i]` on every iteration
- Must consider possibility that these updates will affect program behavior

Memory Aliasing

```
double A[9] =
  { 0,   1,   2,
    4,   8,  16},
   32,  64, 128};

double B[3] = A+3;

sum_rows1(A, B, 3);
```

```
double A[9] =
  { 0,   1,   2,
    3,   8,  16},
   32,  64, 128};

double B[3] = A+3;

sum_rows1(A, B, 3);
```

```
double A[9] =
  { 0,   1,   2,
    3,   3,  16},
   32,  64, 128};

double B[3] = A+3;

sum_rows1(A, B, 3);
```

```
double A[9] =
  { 0,   1,   2,
    3,   6,  16},
   32,  64, 128};

double B[3] = A+3;

sum_rows1(A, B, 3);
```

```
double A[9] =
  { 0,   1,   2,
    3,   6,  22},
   32,  64, 128};

double B[3] = A+3;

sum_rows1(A, B, 3);
```

Value of B:

```
init:   [4, 8, 16]
```

```
i = 0: [3, 8, 16]
```

```
i = 1: [3, 22, 16]
```

```
i = 2: [3, 22, 224]
```

- Code updates `b[i]` on every iteration
- Must consider possibility that these updates will affect program behavior

Memory Matters

```
/* Sum rows is of n X n matrix a
   and store in vector b  */
void sum_rows1(double *a, double *b, long n) {
    long i, j;
    for (i = 0; i < n; i++) {
        b[i] = 0;
        for (j = 0; j < n; j++)
            b[i] += a[i*n + j];
    }
}
```

- Code updates `b[i]` on every iteration
- Why couldn't compiler optimize this away?

```
/* Sum rows is of n X n matrix a
   and store in vector b  */
void sum_rows1(double *a, double *b, long n) {
    long i, j;
    for (i = 0; i < n; i++) {
        sum = 0;
        for (j = 0; j < n; j++)
            sum += a[i*n + j];
        b[i] = sum
    }
}
```

Optimization Blocker: Memory Aliasing

- Aliasing
  - Two different memory references specify single location
  - Easy to have happen in C
    - Since allowed to do address arithmetic
    - Direct access to storage structures
  - Get in habit of introducing local variables
    - Accumulating within loops
    - Your way of telling compiler not to check for aliasing

Loop unrolling

loop unrolling (C)

```c
for (int i = 0; i < N; ++i)
    sum += A[i];
```

```c
int i;
for (i = 0; i + 1 < N; i += 2) {
    sum += A[i];
    sum += A[i+1];
}
// handle leftover, if needed
if (i < N)
    sum += A[i];
```

loop unrolling (ASM)

```
loop:
        cmpl    %edx, %esi
        jle     endOfLoop
        addq    (%rdi,%rdx,8), %rax
        incq    %rdx
        jmp
endOfLoop:
```

```
loop:
        cmpl    %edx, %esi
        jle     endOfLoop
        addq    (%rdi,%rdx,8), %rax
        addq    8(%rdi,%rdx,8), %rax
        addq    $2, %rdx
        jmp     loop
        // plus handle leftover?
endOfLoop:
```

more loop unrolling (C)

```
int i;
for (i = 0; i + 4 <= N; i += 4) {
    sum += A[i];
    sum += A[i+1];
    sum += A[i+2];
    sum += A[i+3];
}
// handle leftover, if needed
for (; i < N; i += 1)
    sum += A[i];
```

automatic loop unrolling

loop unrolling is easy for compilers

…but often not done or done very much

why not?

slower if small number of iterations

larger code — could exceed instruction cache space

loop unrolling performance

on my laptop with 992 elements (fits in L1 cache)

times unrolled	cycles/element	instructions/element
1	1.33	4.02
2	1.03	2.52
4	1.02	1.77
8	1.01	1.39
16	1.01	1.21
32	1.01	1.15

1.01 cycles/element — latency bound