

# **CSC 252: Computer Organization**

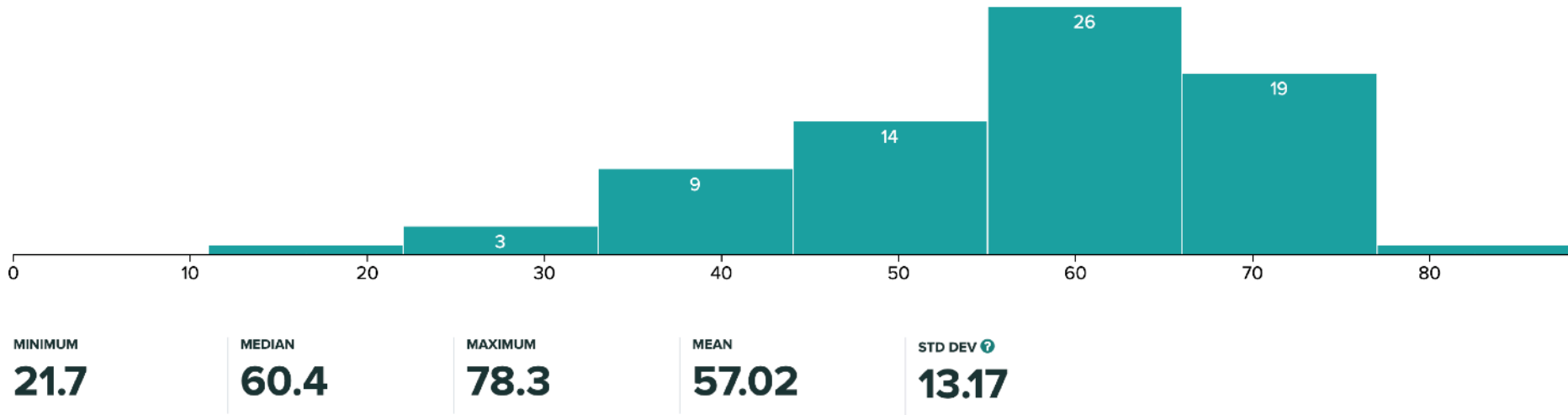
## **Spring 2022: Lecture 15**

Instructor: Yuhao Zhu

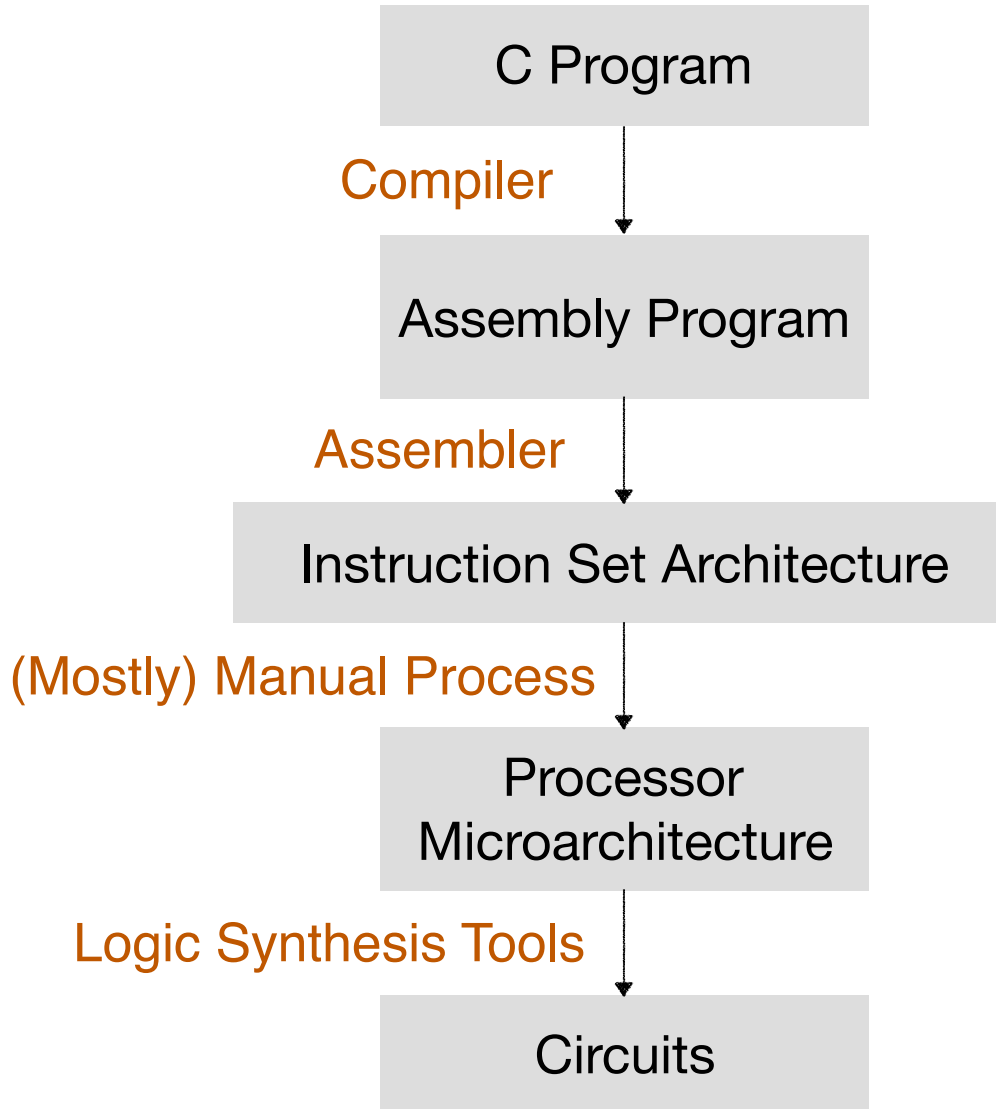
Department of Computer Science  
University of Rochester

# Announcements

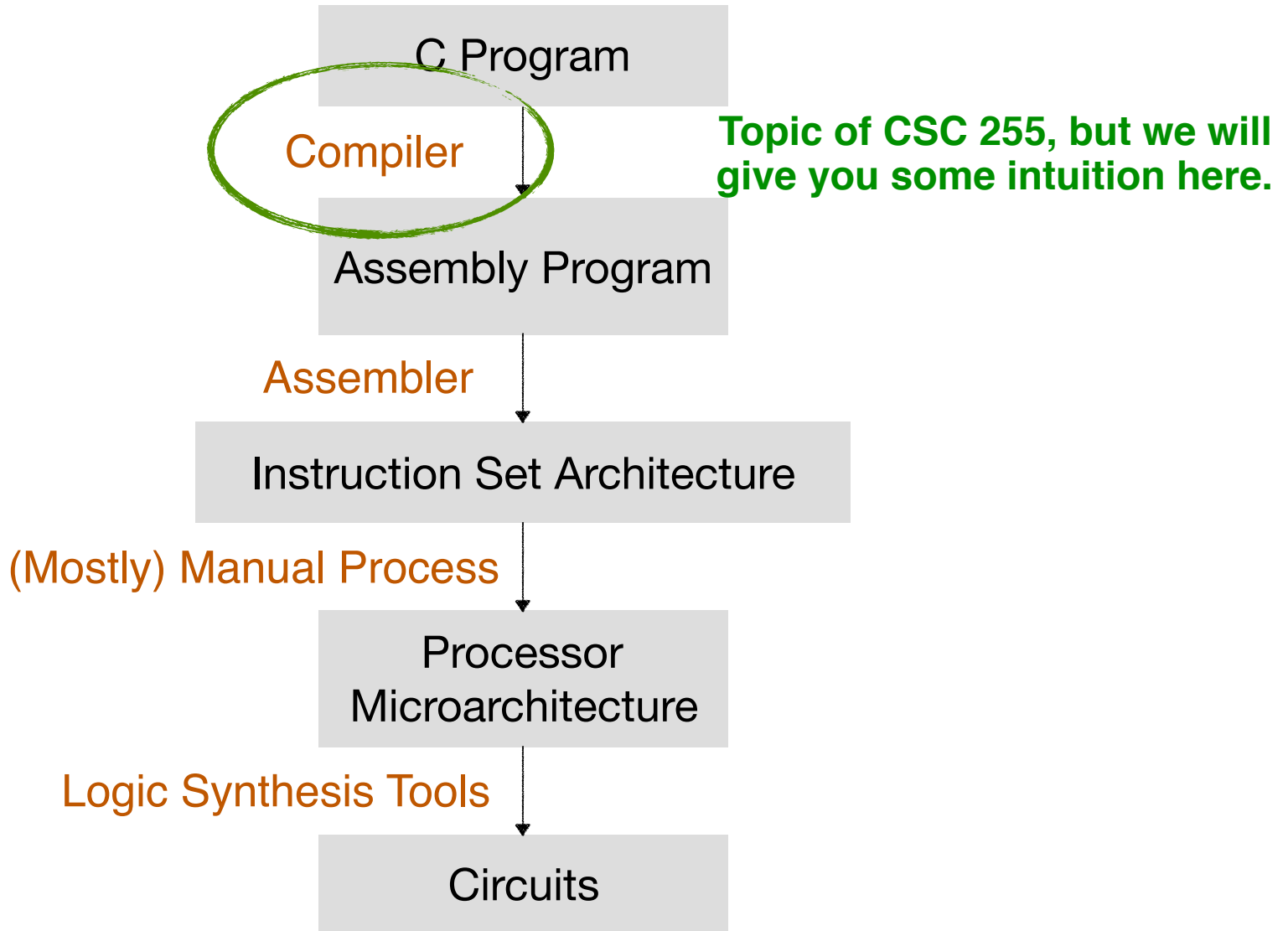
- If you have doubts, calm down and talk to the TAs.
- Will release Lab 3 grades soon.



# So far in 252...



# So far in 252...



# Code Optimization Overview

- Three entities can optimize the program: programmer, compiler, and hardware
- The best thing to speed up a program is to pick a good algorithm. Compilers/hardware can't do that in general.
  - Quicksort:  $O(n \log n) = K * n * \log(n)$
  - Bubblesort:  $O(n^2) = K * n^2$
- Algorithm choice decides overall complexity (big O), compiler/hardware decides the constant factor in the big O notation
- Compiler and hardware implementations decide the K.
- Programmers can write code that makes it easier to compiler and hardware to improve performance.

# Optimizing Code Transformation

- Hardware/Microarchitecture Independent Optimizations
  - Code motion/precomputation
  - Strength reduction
  - Sharing of common subexpressions
- Optimization Blockers
  - Procedure calls
  - Memory aliasing
- Exploit Hardware Microarchitecture

# Generally Useful Optimizations

- Optimizations that you or the compiler should do regardless of processor
- 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];
```

# Compiler-Generated Code Motion (-O1)

```
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];
```



```
set_row:  
    testq    %rcx, %rcx                # Test n  
    jle      .L1                      # If 0, goto done  
    imulq    %rcx, %rdx                # ni = n*i  
    leaq     (%rdi,%rdx,8), %rdx        # rowp = A + ni*8  
    movl     $0, %eax                 # j = 0  
.L3:                                     # loop:  
    movsd    (%rsi,%rax,8), %xmm0      # t = b[j]  
    movsd    %xmm0, (%rdx,%rax,8)      # M[A+ni*8 + j*8] = t  
    addq     $1, %rax                 # j++  
    cmpq     %rcx, %rax               # j:n  
    jne      .L3                      # if !=, goto loop  
.L1:                                     # done:  
    rep ; ret
```



# Reduction in Strength

- Replace costly operation with simpler one
- Shift, add instead of multiply or divide
  - $16 * x \quad \rightarrow \quad x \ll 4$
  - Depends on cost of multiply or divide instruction
  - On Intel Nehalem, integer multiply requires 3 CPU cycles. Division takes even more cycles. Shift can generally be done in 1 cycle.
- Use the `leaq` instruction

# Reduction in Strength

- Replace costly operation with simpler one
- Shift, add instead of multiply or divide
  - $16 * x \quad \rightarrow \quad x \ll 4$
  - Depends on cost of multiply or divide instruction
  - On Intel Nehalem, integer multiply requires 3 CPU cycles. Division takes even more cycles. Shift can generally be done in 1 cycle.
- Use the `leaq` instruction

```
long m12(long x)
{
    return x*12;
}
```

```
leaq (%rdi,%rdi,2), %rax # t <- x+x*2
salq $2, %rax           # return t<<2
```

# Common Subexpression Elimination

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

3 multiplications:  $i*n$ ,  $(i-1)*n$ ,  $(i+1)*n$

```
/* 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;
```



```
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
```

1 multiplication:  $i*n$

```
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;
```



```
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
```

# Today: Optimizing Code Transformation

- Hardware/Microarchitecture Independent Optimizations
  - Code motion/precomputation
  - Strength reduction
  - Sharing of common subexpressions
- Optimization Blockers
  - Procedure calls
  - Memory aliasing
- Exploit Hardware Microarchitecture

# 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');
}
```

# Calling Strlen

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

- **Strlen performance**
  - Has to scan the entire length of a string, looking for null character.
  - $O(N)$  complexity
- **Overall performance**
  - $N$  calls to strlen
  - Overall  $O(N^2)$  performance

# Improving Performance

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

```
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');
}
```

# Optimization Blocker: Procedure Calls

```
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');
}

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

Why couldn't compiler move `strlen` out of loop?

- Procedure may have side effects, e.g., alters global state each time called
- Function may not return same value for given arguments



# Optimization Blocker: Procedure Calls

- Most compilers treat procedure call as a black box
  - Assume the worst case, weak optimizations near them
  - There are interprocedural optimizations (IPO), but they are expensive
  - Sometimes the compiler doesn't have access to source code of other functions because they are object files in a library. Link-time optimizations (LTO) comes into play, but are expensive as well.

# Optimization Blocker: Procedure Calls

- Most compilers treat procedure call as a black box
  - Assume the worst case, weak optimizations near them
  - There are interprocedural optimizations (IPO), but they are expensive
  - Sometimes the compiler doesn't have access to source code of other functions because they are object files in a library. Link-time optimizations (LTO) comes into play, but are expensive as well.
- Remedies:
  - Use of inline functions
  - Do your own code motion

# Optimization Blocker: Procedure Calls

- Most compilers treat procedure call as a black box
  - Assume the worst case, weak optimizations near them
  - There are interprocedural optimizations (IPO), but they are expensive
  - Sometimes the compiler doesn't have access to source code of other functions because they are object files in a library. Link-time optimizations (LTO) comes into play, but are expensive as well.
- Remedies:
  - Use of inline functions
  - Do your own code motion

```
inline void swap(int *m, int *n) {  
    int tmp = *m;  
    *m = *n;  
    *n = tmp;  
}  
  
void foo () {  
    swap(&x, &y);  
}
```



```
void foo () {  
    int tmp = x;  
    x = y;  
    y = tmp;  
}
```

# Optimization Blocker #2: Memory Aliasing

```
/* Sum rows 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];
    }
}
```

Value of a:

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

Value of b:

```
init:  [x, x, x]
```

# Optimization Blocker #2: Memory Aliasing

```
/* Sum rows 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];
    }
}
```

Value of a:

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

Value of b:

```
init: [x, x, x]
```

```
i = 0: [3, x, x]
```

# Optimization Blocker #2: Memory Aliasing

```
/* Sum rows 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];
    }
}
```

Value of a:

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

Value of b:

init: [x, x, x]

i = 0: [3, x, x]

i = 1: [3, 28, x]

# Optimization Blocker #2: Memory Aliasing

```
/* Sum rows 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];
    }
}
```

Value of a:

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

Value of b:

init: [x, x, x]

i = 0: [3, x, x]

i = 1: [3, 28, x]

i = 2: [3, 28, 224]

# A Potential Optimization

```
/* Sum rows 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];
    }
}
```

Every iteration updates  
memory location `b[i]`.  
Memory accesses are  
slow, so...



# A Potential Optimization

```
/* Sum rows 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];
    }
}
```

Every iteration updates  
memory location `b[i]`.  
Memory accesses are  
slow, so...



```
double val = 0;
for (j = 0; j < n; j++)
    val += a[i*n + j];
b[i] = val;
```

Every iteration updates `val`,  
which could stay in register.  
Update memory only once.

# A Potential Optimization

```
/* Sum rows 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];
    }
}
```

Every iteration updates  
memory location `b[i]`.  
Memory accesses are  
slow, so...



```
double val = 0;
for (j = 0; j < n; j++)
    val += a[i*n + j];
b[i] = val;
```

Every iteration updates `val`,  
which could stay in register.  
Update memory only once.

Why can't a compiler perform this optimization?

# Memory Aliasing

```
/* Sum rows 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];
    }
}
```

Value of a:

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

double b[3] = a+3;

sum_rows1(a, b, 3);
```

b

Value of b:

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

# Memory Aliasing

```
/* Sum rows 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];
    }
}
```

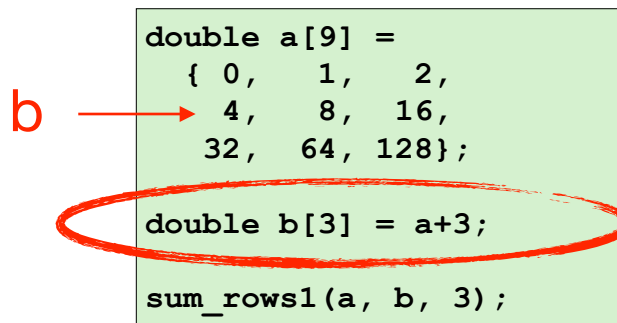
## Value of a:

**b** →

```
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]

# Memory Aliasing

```
/* Sum rows 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];
    }
}
```

Value of a:

**b** →

```
double a[9] =
{ 0, 1, 2,
  3, 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]

# Memory Aliasing

```
/* Sum rows 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];
    }
}
```

Value of a:

**b** →

```
double a[9] =
{ 0, 1, 2,
  3, 0, 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]

# Memory Aliasing

```
/* Sum rows 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];
    }
}
```

## Value of a:

**b** →

```
double a[9] =
{ 0, 1, 2,
  3, 3, 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]

# Memory Aliasing

```
/* Sum rows 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];
    }
}
```

Value of a:

**b** →

```
double a[9] =
{ 0, 1, 2,
  3, 6, 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]



# Memory Aliasing

```
/* Sum rows 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];
    }
}
```

Value of a:

**b** →

```
double a[9] =
{ 0, 1, 2,
  3, 22, 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]

# Memory Aliasing

```
/* Sum rows 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];
    }
}
```

## Value of a:

**b** →

```
double a[9] =
{ 0, 1, 2,
  3, 22, 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]

# Memory Aliasing

```
/* Sum rows 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];
    }
}
```

## Value of a:

**b** →

```
double a[9] =
{ 0, 1, 2,
  3, 22, 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]

# Optimization Blocker: Memory Aliasing

- Aliasing
  - Two different memory references (array elements or pointers) specify the same memory location
  - Easy to have in C
    - Since C allows address/pointer 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

# Today: Optimizing Code Transformation

- Hardware/Microarchitecture Independent Optimizations
  - Code motion/precomputation
  - Strength reduction
  - Sharing of common subexpressions
- Optimization Blockers
  - Procedure calls
  - Memory aliasing
- Exploit Hardware Microarchitecture

# Exploiting Instruction-Level Parallelism (ILP)

- Hardware can execute multiple instructions in parallel
  - Pipeline is a classic technique. Multiple instructions are being executed at the same time
- Performance limited by control/data dependencies
- Simple transformations can yield dramatic performance improvement
  - Compilers often cannot make these transformations
  - Lack of associativity and distributivity in floating-point arithmetic

# Baseline Code

```
for (i = 0; i < length; i++) {  
    t = t * d[i];  
    *dest = t;  
}
```

.L519:

```
imulq (%rax,%rdx,4), %ecx  
addq  $1, %rdx      # i++  
cmpq  %rdx, %rbp    # Compare length:i  
jg     .L519        # If >, goto Loop
```

← Real work

← Overhead

# Loop Unrolling (2x1)

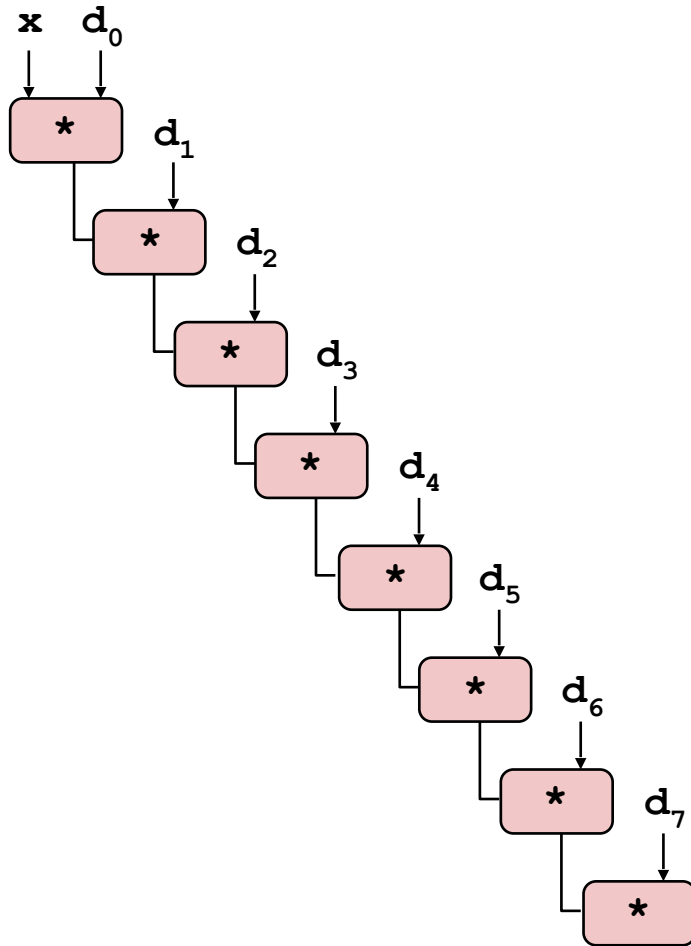
```
long limit = length-1;
long i;
/* Combine 2 elements at a time */
for (i = 0; i < limit; i+=2) {
    x = (x * d[i]) * d[i+1];
}

/* Finish any remaining elements */
for (; i < length; i++) {
    x = x * d[i];
}
*dest = x;
```

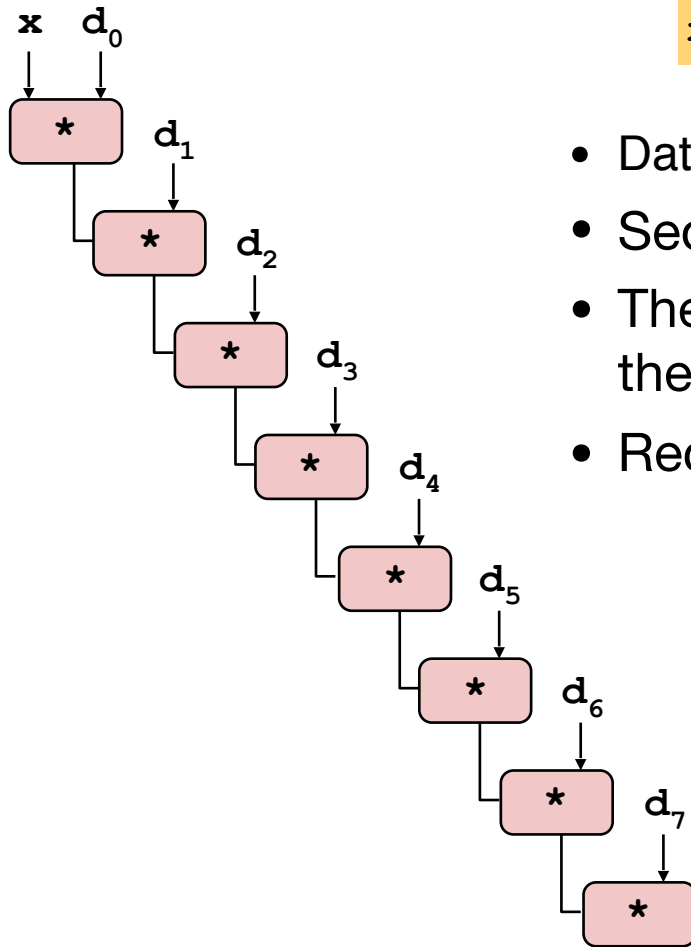
- Perform 2x more useful work per iteration
- Reduce loop overhead (comp, jmp, index dec, etc.)
- What's the trade-off here?



# DFG of This Implementation



# DFG of This Implementation



```
x = (x OP d[i]) OP d[i+1];
```

- Data dependency graph
- Sequential dependence
- The performance of the code is dictated by the the latency of OP
- Recall the read-after-write dependency

# Loop Unrolling with Separate Accumulators

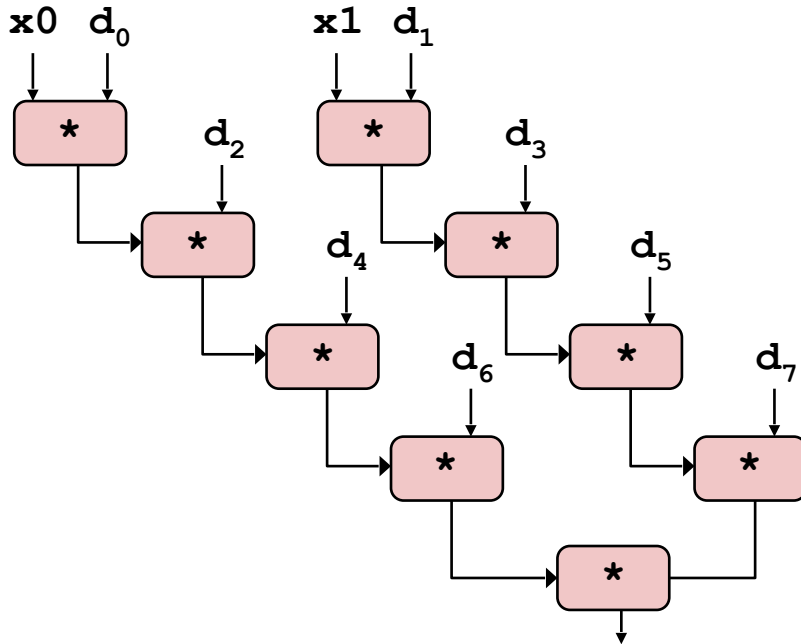
```
long limit = length-1;
long i;
/* Combine 2 elements at a time */
for (i = 0; i < limit; i+=2) {
    x0 = x0 * d[i];
    x1 = x1 * d[i+1];
}

/* Finish any remaining elements */
for (; i < length; i++) {
    x0 = x0 * d[i];
}
*dest = x0 * x1;
```

# Data-Flow Graph (DFG)

```
x0 = x0 * d[i];  
x1 = x1 * d[i+1];
```

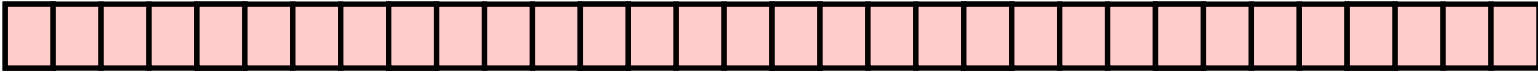
- What changed:
  - Two independent “streams” of operations
  - Reduce data dependency



# Aside: Vector Registers and Instructions

A single 32-byte register; can be used differently

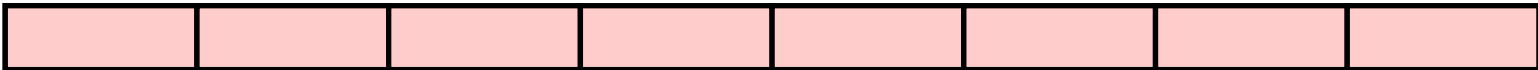
32 single-byte integers



16 2-byte integers



8 4-byte integers



8 single-precision floats



4 double-precision floats



1 single-precision floats



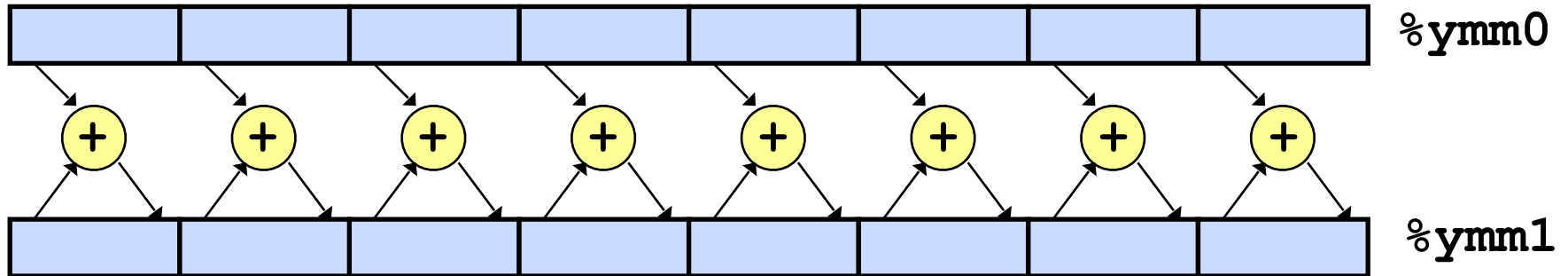
1 double-precision floats



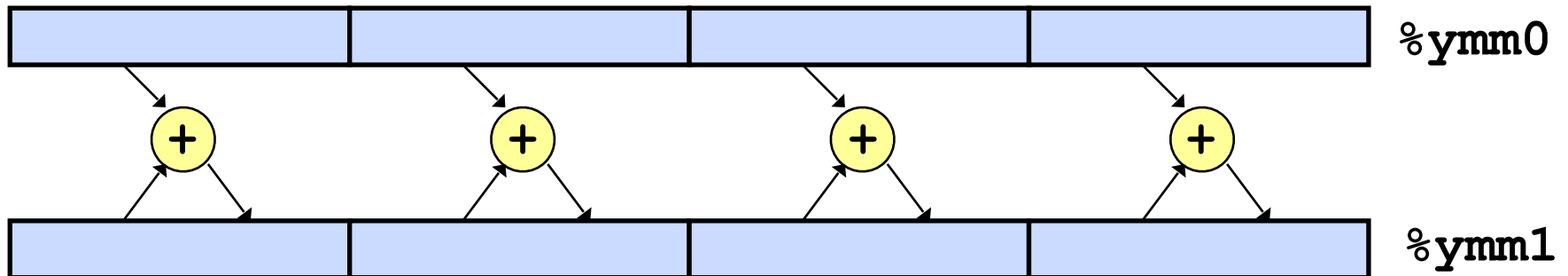
# Aside: SIMD Operations

Single Instruction Multiple Data

`vaddsd %ymm0, %ymm1, %ymm1`



`vaddpd %ymm0, %ymm1, %ymm1`



**Can manually write assembly code that uses SIMD/vector instructions;  
some compilers will automatically vectorize your code. Hard problem!**

# Aside: Profile-Guided Optimization

```
float foo(int x, int y)
{
    return pow(x, y) * 100 / log(x) * sqrt(y);
}
```

# Aside: Profile-Guided Optimization

- As a programmer, if you know what  $x$  and  $y$  will be, say 5, you could direct return the results 23769.8 without having to the computation

```
float foo(int x, int y)
{
    return pow(x, y) * 100 / log(x) * sqrt(y);
}
```



# Aside: Profile-Guided Optimization

- As a programmer, if you know what  $x$  and  $y$  will be, say 5, you could direct return the results 23769.8 without having to the computation
- Compiler would have no idea

```
float foo(int x, int y)
{
    return pow(x, y) * 100 / log(x) * sqrt(y);
}
```

# Aside: Profile-Guided Optimization

- As a programmer, if you know what  $x$  and  $y$  will be, say 5, you could direct return the results 23769.8 without having to the computation
- Compiler would have no idea
- Except...Profile-guided optimizations:

```
float foo(int x, int y)
{
    return pow(x, y) * 100 / log(x) * sqrt(y);
}
```

# Aside: Profile-Guided Optimization

- As a programmer, if you know what  $x$  and  $y$  will be, say 5, you could direct return the results 23769.8 without having to the computation
- Compiler would have no idea
- Except...Profile-guided optimizations:
  - Run the code multiple times using some sample inputs, and observe the values of  $x$  and  $y$  (statistically).

```
float foo(int x, int y)
{
    return pow(x, y) * 100 / log(x) * sqrt(y);
}
```

# Aside: Profile-Guided Optimization

- As a programmer, if you know what  $x$  and  $y$  will be, say 5, you could direct return the results 23769.8 without having to the computation
- Compiler would have no idea
- Except...Profile-guided optimizations:
  - Run the code multiple times using some sample inputs, and observe the values of  $x$  and  $y$  (statistically).
  - If let's say 99% of the time,  $x = 2$  and  $y = 5$ , what could the compiler do then?

```
float foo(int x, int y)
{
    return pow(x, y) * 100 / log(x) * sqrt(y);
}
```

# Aside: Profile-Guided Optimization

- As a programmer, if you know what  $x$  and  $y$  will be, say 5, you could direct return the results 23769.8 without having to the computation
- Compiler would have no idea
- Except...Profile-guided optimizations:
  - Run the code multiple times using some sample inputs, and observe the values of  $x$  and  $y$  (statistically).
  - If let's say 99% of the time,  $x = 2$  and  $y = 5$ , what could the compiler do then?

```
float foo(int x, int y)
{
    return pow(x, y) * 100 / log(x) * sqrt(y);
}
```

```
float foo(int x, int y)
{
    if (x == 2 && y == 5) return 23769.8;
    else return pow(x, y) * 100 / log(x) * sqrt(y);
}
```

# Code Optimization Summary

- From a programmer's perspective:
  - What you know: the functionality/intention of your code; the inputs to the program; all the code in the program
  - What you might not know: the hardware details.
- From a compiler's perspective:
  - What you know: all the code in the program; (maybe) the hardware details.
  - What you might not know: the inputs to the program; the intention of the code
- From the hardware's perspective:
  - What you know: the hardware details; some part of the code
  - What you might not know: the inputs to the program; the intention of the code
- The different perspectives indicate that different entities have different responsibilities, limitations, and advantages in optimizing the code