Lecture 15
Compiler Optimizations

CS213 – Intro to Computer Systems

Branden Ghena – Fall 2023

Slides adapted from:
Bryant, O'Hallaron (CMU), Garcia, Weaver (UC Berkeley)

Northwestern

Administrivia

- Attack Lab due today!
  - Most of you are finished with it though ❤
- SETI Lab is out and ready to be worked on
  - Today is the last of the material that will be helpful towards it
  - Be careful with this one:
    - Lots of C code to understand and write
    - Spans Thanksgiving, so you have less time to work on it than you think
- Homework 4 should be out later today
- Reminder: midterm 2 on Wednesday of exam week
  - Covers material from the second half of class

Today's Goals

- Discuss the role of a compiler
- Explore basic optimizations at both the local and global levels
- Understand limitations of optimizations
- Describe how GCC can be configured to use these optimizations

Outline

- **Compilers and Optimizations**
- Local Optimizations
- Global Optimizations
- Obstacles to Optimization
- GNU C Compiler (GCC)

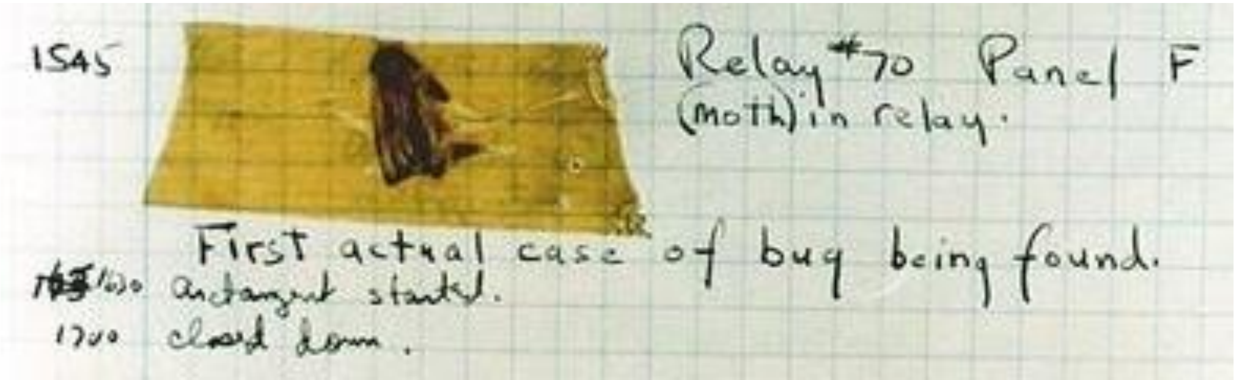
How do we get code to run on a machine?

- CPU only understands "machine code"
  - All other languages must either be interpreted or compiled

- The very bad old days: write hexadecimal instructions by hand
  - This was back in the 1940s and the days of vacuum tubes
  - Hook up wires and switches to form data input

Rear Admiral Grace Hopper

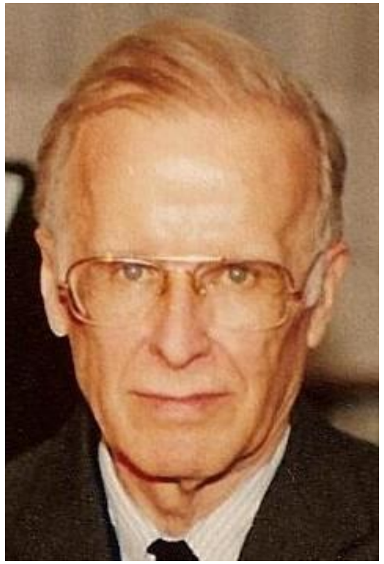
- Popularized term “debugging”
  - After finding a literal moth in their computer



- Invented first compiler in 1951
  - “I decided data processors ought to be able to write their programs in English, and the computers would translate them into machine code”

Other Compilers Champions

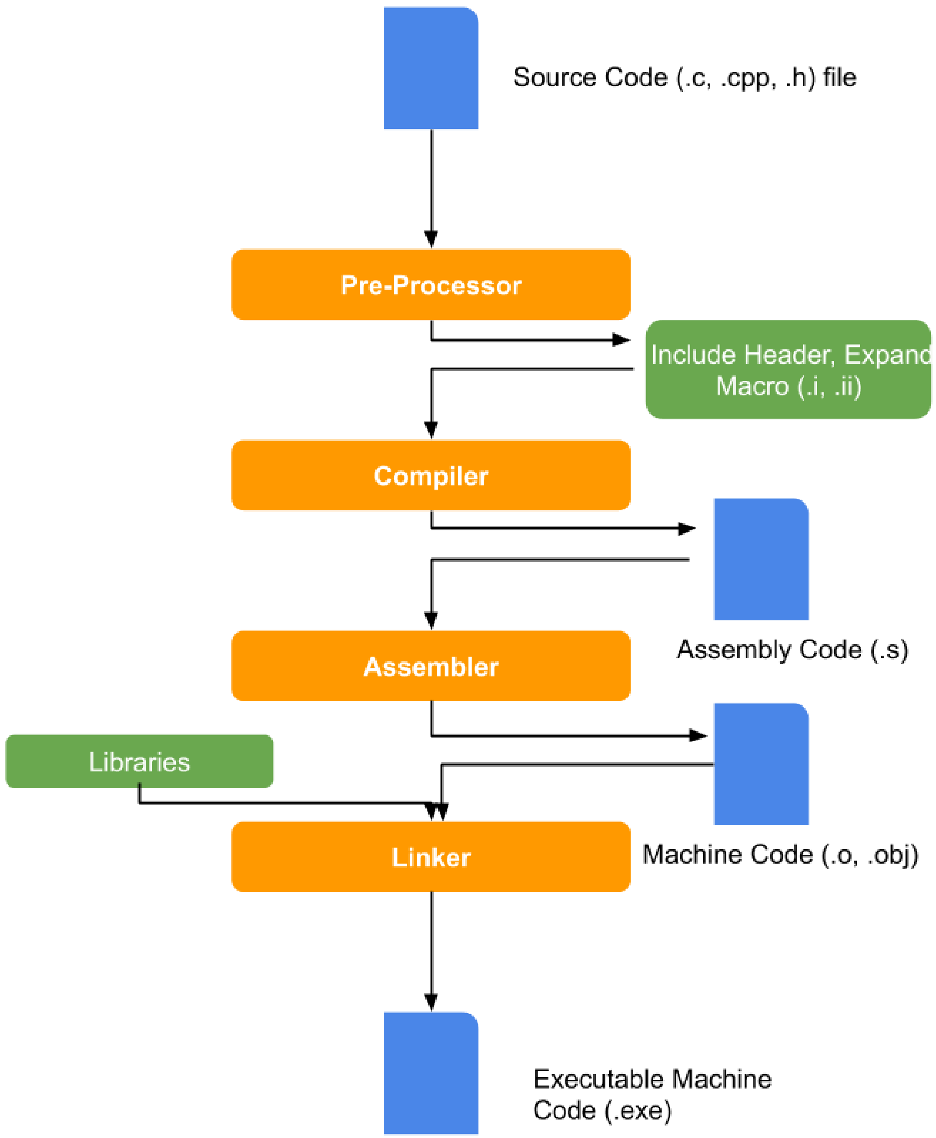
- John Backus
  - Developed FORTRAN in 1957
- "Much of my work has come from being lazy. I didn't like writing programs, and so, when I was working on the IBM 701, I started work on a programming system to make it easier to write programs"



- Fran Allen
  - Pioneer of compiler optimization techniques
  - Wrote a 1966 paper introducing control flow graphs, which are central to compiler theory
- First woman to win the Turing Award

C compilation steps

1. Pre-processor
   - Text insertion of macros and #includes
2. Compiler
   - Transform C source into assembly
   - Also perform optimizations along the way
3. Assembler
   - Transform assembly into machine code
4. Linker
   - Place code at real addresses and fixup



Optimizations

- An **optimization** is a code transformation with the goal of making a program faster
  - Can be done manually, by a programmer
  - Or can be done automatically, by a compiler
  - MUST maintain correctness

- Some optimizations are processor-dependent
  - They take advantage of unique processor capabilities
  - Example: right shift instead of divide by powers of two

- Some optimizations are processor-independent
  - They make programs faster regardless of processor
  - Example: removing redundant code

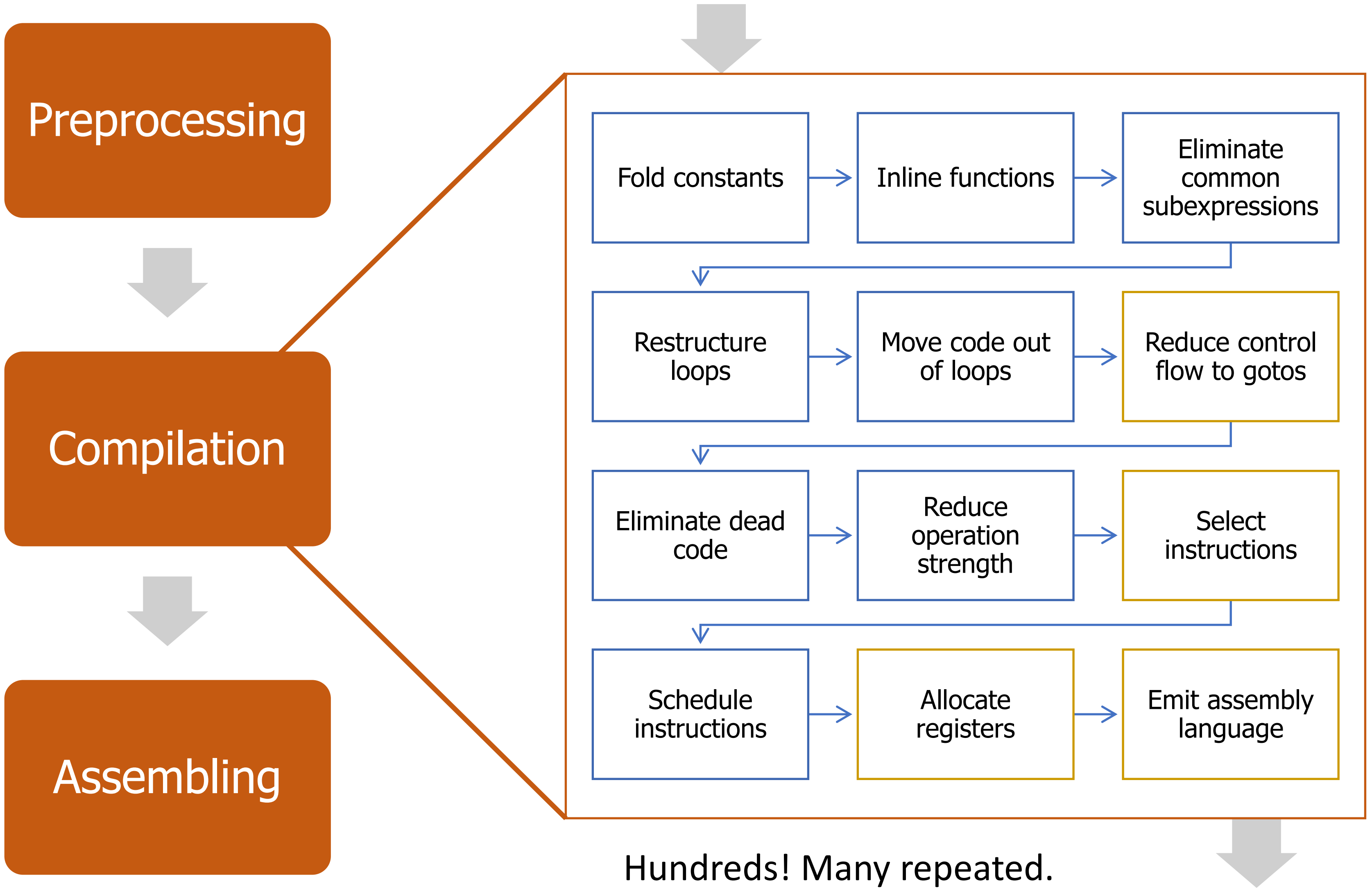
General goals of compiler optimization

- Minimize number of instructions
  - Don’t do calculations more than once
  - Don’t do unnecessary calculations at all
  - Avoid slow instructions

- Avoid waiting for memory
  - Keep everything in registers whenever possible
  - Access memory in cache-friendly patterns

- Avoid branching
  - Branches are slow for all modern processor architectures
  - Don’t make unnecessary decisions
  - Make it easier for the CPU to predict branches whenever possible

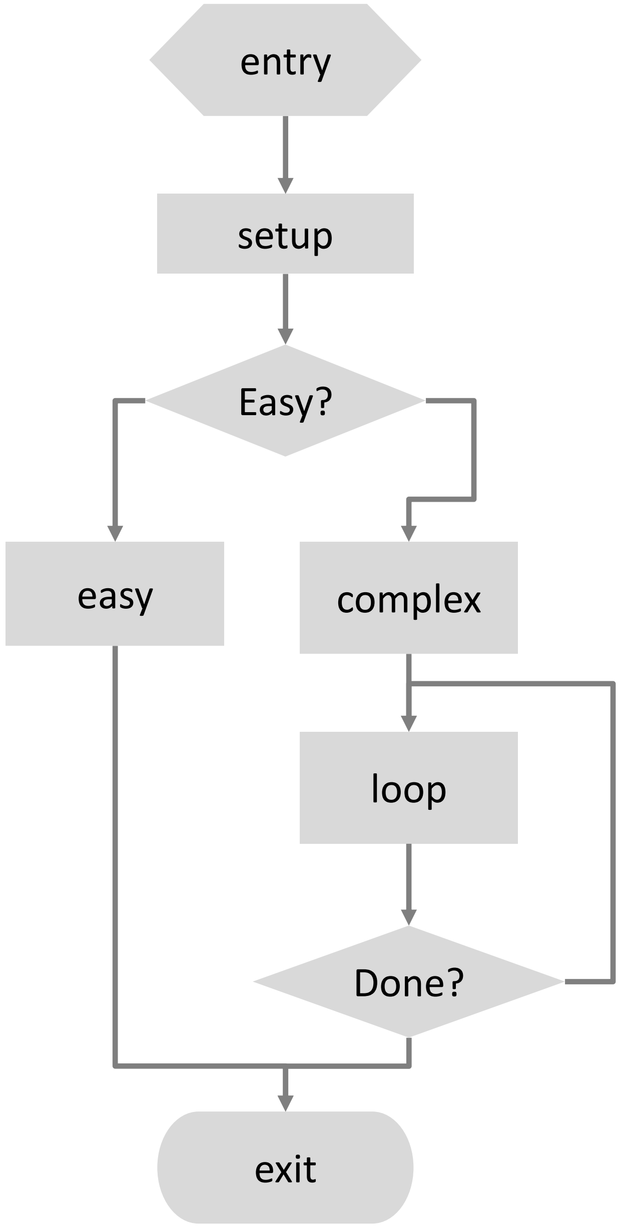
Compilation is a pipeline



Hundreds! Many repeated.

Two categories of optimizations

- Local optimizations
  - Work within a single basic block
    (chunks of code with no gotos or labels)
  - Examples: combining constants, eliminating dead code
- Global optimizations
  - Work across the "control flow graph" of an entire function
  - Examples: loop transformations
- Optimizations are often limited to function boundaries



Outline

- Compilers and Optimizations
- **Local Optimizations**
- Global Optimizations
- Obstacles to Optimization
- GNU C Compiler (GCC)

Constant Folding

- Do arithmetic in the compiler

```
long mask = 0xFF << 8;  →  long mask = 0xFF00;
```

- Any expression with constant inputs can be folded
  - Might even be able to remove library calls…

```
size_t namelen = strlen("Harry Bovik");

→  size_t namelen = 11;
```

Strength reduction

- Replace expensive operations with cheaper ones

```
long a = b * 5;
```

```
→  long a = (b << 2) + b;
```

- Multiplication and division are the usual targets
- Multiplication is often hiding in memory access expressions
  - Example: array indexing

Dead code elimination

- Don’t emit code that will never be executed

~~`if (0) { puts("Kilroy was here"); }`~~
~~`if (1) {`~~ `puts("Only bozos on this bus");` ~~`}`~~

- Don’t emit code whose result is overwritten

~~`x = 23;`~~
`x = 42;`

- These may look silly, but...
  - Can be produced by other optimizations
  - Assignments to x might be far apart

Common Subexpression Elimination

- Factor out repeated calculations or memory accesses
  - Only do them once
  - Makes code closer to the assembly representation too

```
norm[i] = v[i].x*v[i].x + v[i].y*v[i].y;
```

→

```
    elt = &v[i];
    x = elt->x;
    y = elt->y;
    norm[i] = x*x + y*y;
```

Break + Question

```
int a = 5;
int x = 2*a;
int y = x+6;
int t = x * y;
if (t < 0) {
  printf("Message 1\n");
} else {
  printf("Message 2\n");
}
```

- Optimize the code snippet as much as possible

Break + Question

```
int a = 5;
int x = 2*a;
int y = x+6;
int t = x * y;
if (t < 0) {
  printf(“Message 1\n”);
} else {
  printf(“Message 2\n”);
}
```

- Optimize the code snippet as much as possible
- Result:
  `printf(“Message 2\n”);`
- `t` is always 160
  - Fold constants
- 160 is never less than 0
  - Remove dead code

Outline

- Compilers and Optimizations
- Local Optimizations
- **Global Optimizations**
- Obstacles to Optimization
- GNU C Compiler (GCC)

Inlining

- Copy body of a function into its caller(s)
  - Can create opportunities for many other optimizations
  - Can make code much bigger and therefore slower (if larger than cache!)

```
int pred(int x) {
    if (x == 0)
        return 0;
    else
        return x - 1;
}

int func(int y) {
    return pred(y)
         + pred(0)
         + pred(y+1);
}
```

```
int func(int y) {
  int tmp;
  if (y == 0) tmp = 0; else tmp = y - 1;
  if (0 == 0) tmp += 0; else tmp += 0 - 1;
  if (y+1 == 0) tmp += 0; else tmp += (y + 1) - 1;
  return tmp;
}
```

Inlining

- Copy body of a function into its caller(s)
  - Can create opportunities for many other optimizations
  - Can make code much bigger and therefore slower (if larger than cache!)

```
int pred(int x) {
    if (x == 0)
        return 0;
    else
        return x - 1;
}

int func(int y) {
    return pred(y)
        + pred(0)
        + pred(y+1);
}
```

```
int func(int y) {
  int tmp;
  if (y == 0) tmp = 0; else tmp = y - 1;
  if (0 == 0) tmp += 0; else tmp += 0 - 1;
  if (y+1 == 0) tmp += 0; else tmp += (y + 1) - 1;
  return tmp;
}
```

Always true

Does nothing

Can constant fold

Inlining

- Copy body of a function into its caller(s)
  - Can create opportunities for many other optimizations
  - Can make code much bigger and therefore slower (if larger than cache!)

```
int func(int y) {
  int tmp;
  if (y == 0) tmp = 0; else tmp = y - 1;
  if (0 == 0) tmp += 0; else tmp += 0 - 1;
  if (y+1 == 0) tmp += 0; else tmp += (y + 1) - 1;
  return tmp;
}
```

```
int func(int y) {
  int tmp = 0;
  if (y != 0) tmp = y - 1;
  if (y != -1) tmp += y;
  return tmp;
}
```

End result is MUCH simpler!

Code Motion

- Move calculations out of a loop
  - Only valid if every iteration would produce same result

```
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];
}
```

Loop Transformations

Rearrange entire loop nests for maximum efficiency

```
/* Two stages of some calculation */
void compute(double *a, double *b, long n) {
  for (long i = 0; i < n; i++)
    for (long j = 0, j < n; j++)
      a[j*n + i] = atan2(i, j);

  for (long i = 0; i < n; i++)
    for (long j = 0, j < n; j++)
      b[i*n + j] = a[i*n + j] + (i >= 1 && j >= 1)
                                ? a[(i-1)*n + (j-1)]
                                : 0;
}
```

Loop Transformations

Loop interchange: do iterations in cache-friendly order

```
/* Two stages of some calculation */
void compute(double *a, double *b, long n) {
  for (long i = 0; i < n; i++)
    for (long j = 0, j < n; j++)
      a[i*n + j] = atan2(j, i);

  for (long i = 0; i < n; i++)
    for (long j = 0, j < n; j++)
      b[i*n + j] = a[i*n + j] + (i >= 1 && j >= 1)
                                ? a[(i-1)*n + (j-1)]
                                : 0;
}
```

Loop Transformations

Loop fusion: combine adjacent loops with the same limits

```
/* Two stages of some calculation */
void compute(double *a, double *b, long n) {
  for (long i = 0; i < n; i++) {
    for (long j = 0, j < n; j++) {
      a[i*n + j] = atan2(j, i);

  for (long i = 0; i < n; i++)
    for (long j = 0, j < n; j++)
      b[i*n + j] = a[i*n + j] + (i >= 1 && j >= 1)
                                ? a[(i-1)*n + (j-1)]
                                : 0;

    }
  }
}
```

Loop Transformations

Induction variable elimination: replace loop indices with algebra

```
/* Two stages of some calculation */
void compute(double *a, double *b, long n) {
  for (long i = 0; i < n*n; i++) {
    for (long j = 0, j < n; j++) {
      a[i] = atan2(i%n, i/n);



      b[i]      = a[i] + (i >= n && i%n >= 1)
                         ? a[i – n - 1]
                         : 0;

    }
  }
}
```

Loop Transformations

Top is the original code

Bottom is the transformed version

Note: still $O(n^2)$ complexity!

But the constant factor is much smaller than before

```
/* Two stages of some calculation */
void compute(double *a, double *b, long n) {
  for (long i = 0; i < n; i++)
    for (long j = 0, j < n; j++)
      a[j*n + i] = atan2(i, j);

  for (long i = 0; i < n; i++)
    for (long j = 0, j < n; j++)
      b[i*n + j] = a[i*n + j] + (i >= 1 && j >= 1)
                         ? a[(i-1)*n + (j-1)] : 0;
}
```

```
/* Two stages of some calculation */
void compute(double *a, double *b, long n) {
  for (long i = 0; i < n*n; i++) {
      a[i] = atan2(i%n, i/n);
      b[i] = a[i] + (i >= n && i%n >= 1)
                         ? a[i – n - 1] : 0;
  }
}
```

Break + Quiz

- **Optimize the following code:** (hint: could be MUCH smaller)

```
long multi_loop(long orig_value) {
    long new_value = 0;
    for (int i=0; i<4; i++) {
        for (int j=0; j<8; j++) {
            new_value += 1;
        }
        new_value += orig_value;
    }
    return new_value;
}
```

Break + Quiz

- **Optimize the following code:** (hint: could be MUCH smaller)

```
long multi_loop(long orig_value) {
    long new_value = 0;
    for (int i=0; i<4; i++) {
        for (int j=0; j<8; j++) {
            new_value += 1;
        }
        new_value += orig_value;
    }
    return new_value;
}
```

```
long multi_loop(long orig_value) {
    return 4*orig_value + 32;
}
```

Outline

- Compilers and Optimizations
- Local Optimizations
- Global Optimizations
- **Obstacles to Optimization**
- GNU C Compiler (GCC)

Limits to compiler optimization

- Generally cannot improve algorithmic complexity
  - Only constant factors, but those can be worth 10x or more…

- MUST NOT cause any change in program behavior
  - Programmer may not care about "edge case" behavior, but compiler does not know that
  - Exception: language may declare some changes acceptable (**UNDEFINED BEHAVIOR**)

- Often only analyze one function at a time
  - Whole-program analysis ("LTO") expensive but gaining popularity
  - Exception: *inlining* merges many functions into one

- Tricky to anticipate run-time inputs
  - Profile-guided optimization can help with common case, but…
  - "Worst case" performance can be just as important as "normal"

Optimization Challenges

1. **Memory aliasing**

2. Function calls

3. Non-associative arithmetic

4. Larger cache optimizations

Memory Aliasing

- Code updates `b[i]` on every iteration

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

`b[i]` *should* just be placed in a register and only a single memory write should occur

Memory Aliasing

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

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

```
# sum_rows1 inner loop
.L4:
   movsd    (%rsi,%rax,8), %xmm0   # FP load
   addsd    (%rdi), %xmm0          # FP add
   movsd    %xmm0, (%rsi,%rax,8)   # FP store
   addq     $8, %rdi
   cmpq     %rcx, %rdi
   jne      .L4
```

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];
    }
}
```

Compiler MUST consider that memory aliasing could occur

- Unless it can *prove* it is impossible

A and B overlap in memory?

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

sum_rows1(A, &(A[3]), 3);
```

```
double A[9] =
  { 0,   1,   2,
    3,  22, 224,
   32,  64, 128};
```

Value of B:

```
init:   [4, 8, 16]
i = 0:  [3, 8, 16]
i = 1:  [3, 22, 16]
i = 2:  [3, 22, 224]
```

Avoiding aliasing penalties: with local variable

- Use a local variable for intermediate results

```
/* Sum rows of n X n matrix a and store in vector b. */
void sum_rows2(double *a, double *b, long n) {
    long i, j;
    for (i = 0; i < n; i++) {
        double val = 0;
        for (j = 0; j < n; j++)
            val += a[i*n + j];
        b[i] = val;
    }
}
```

```
# sum_rows2 inner loop
.Loop:
        addsd   (%rdi), %xmm0    # FP load + add
        addq    $8, %rdi
        cmpq    %rax, %rdi
        jne     .Loop
```

Avoiding aliasing penalties: aliasing still occurs

- Still changes A if aliased because that's what the code specifies

```
/* Sum rows of n X n matrix a and store in vector b. */
void sum_rows2(double *a, double *b, long n) {
    long i, j;
    for (i = 0; i < n; i++) {
        double val = 0;
        for (j = 0; j < n; j++)
            val += a[i*n + j];
        b[i] = val;
    }
}
```

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

sum_rows1(A, &(A[3]), 3);
```

```
double A[9] =
  { 0,   1,   2,
    3,  27, 224,
   32,  64, 128};
```

Value of B:

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

i = 0:  [3, 8, 16]

i = 1:  [3, 27, 16]

i = 2:  [3, 27, 224]
```

Avoiding aliasing penalties: with `restrict` keyword

- Use `restrict` keyword to tell compiler that `a` and `b` cannot alias

```
/* Sum rows of n X n matrix a and store in vector b. */
void sum_rows3(double *restrict a, double *restrict 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];
    }
}
```

```
# sum_rows2 inner loop
.Loop:
        addsd   (%rdi), %xmm0    # FP load + add
        addq    $8, %rdi
        cmpq    %rax, %rdi
        jne     .Loop
```

Avoiding aliasing penalties: with different language

- Use a different language altogether
  - For example, in Fortran array arguments are assumed not to alias

```
subroutine sum_rows4(a, b, n)
    implicit none
    integer, parameter :: dp = kind(1.d0)
    real(kind=dp), dimension(:), intent(in) :: a
    real(kind=dp), dimension(:), intent(out) :: b
    integer, intent(in) :: n
    integer :: i, j
    do i = 1,n
        b(i) = 0
        do j = 1,n
            b(i) = b(i) + a(i*n + j)
        end
    end
end
```

```
# sum_rows2 inner loop
.Loop:
        addsd   (%rdi), %xmm0    # FP load + add
        addq    $8, %rdi
        cmpq    %rax, %rdi
        jne     .Loop
```

Optimization Challenges

1. Memory aliasing

2. **Function calls**

3. Non-associative arithmetic

4. Larger cache optimizations

Function calls are opaque

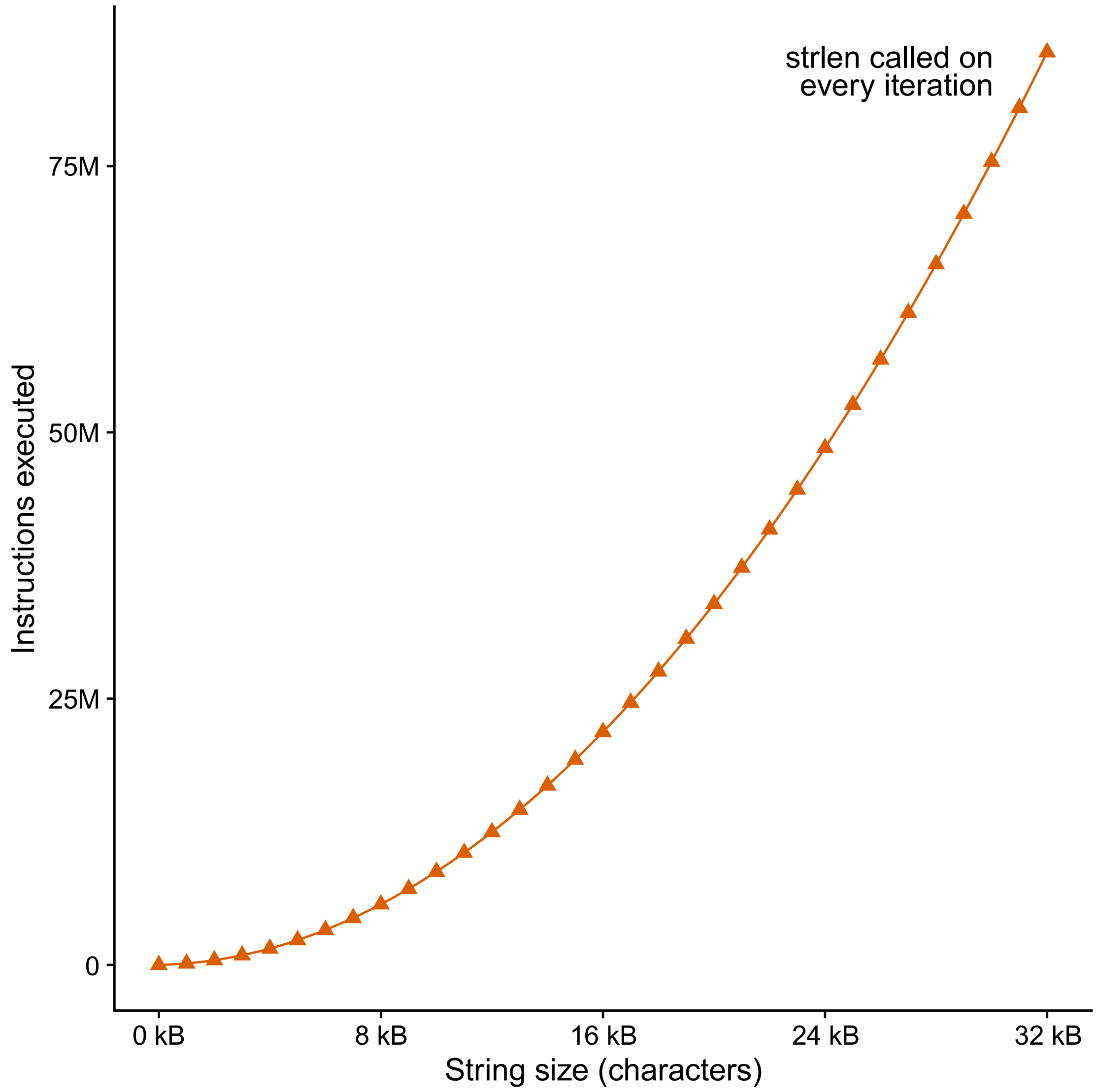
- Compiler examines one function at a time
  - Some exceptions for code in a single file
- Must assume a function call could do anything
- Cannot usually
  - Move function calls
  - Change number of times a function is called
  - Cache data from memory in registers across function calls

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

- O(n) execution time
- Return value depends on:
  - value of *s*
  - contents of memory at address *s*
    - Only cares about whether individual bytes are zero
    - Does not modify memory
- Compiler might know *some* of that (but probably not)

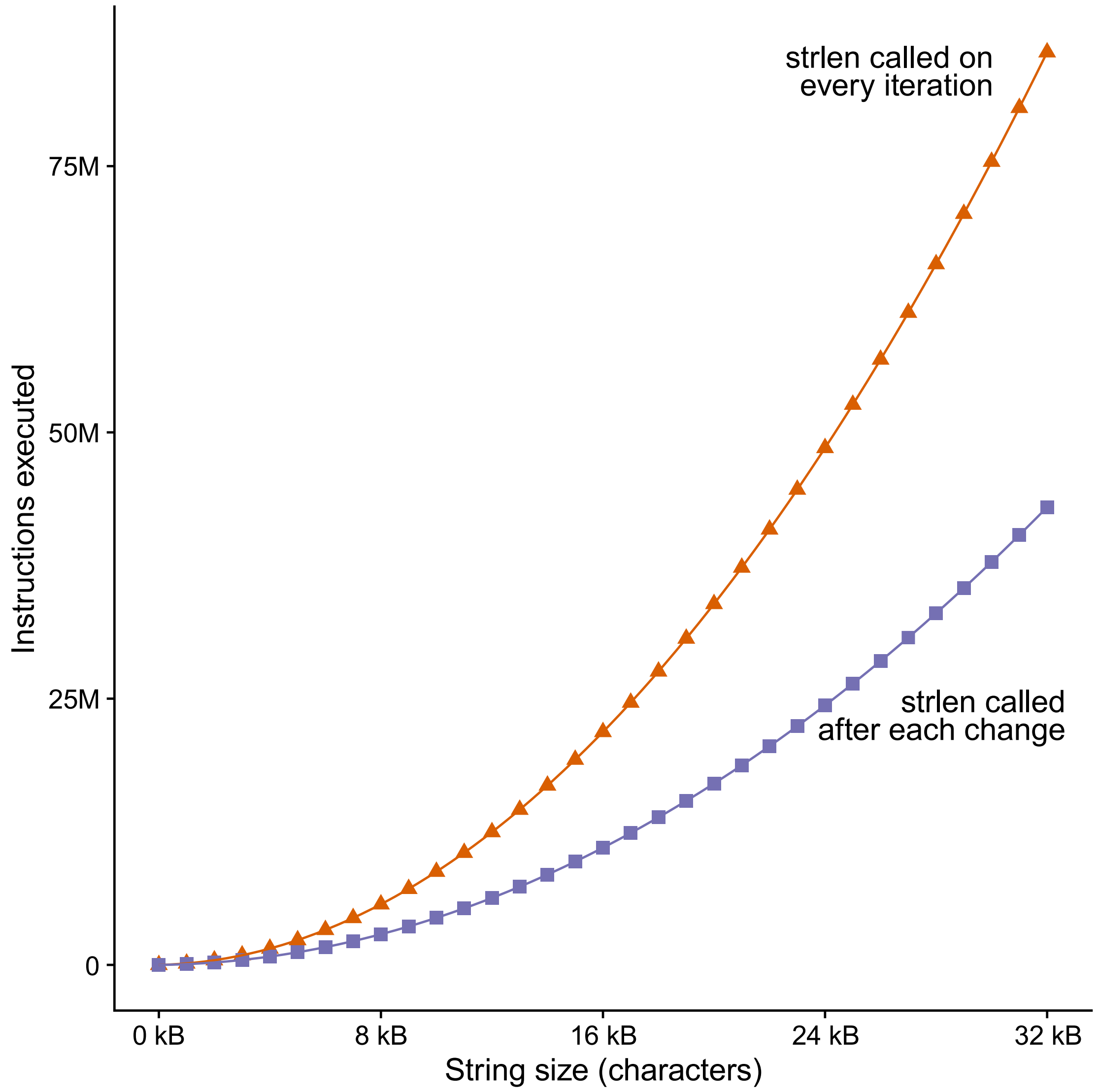
Can’t move functions out of loops

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



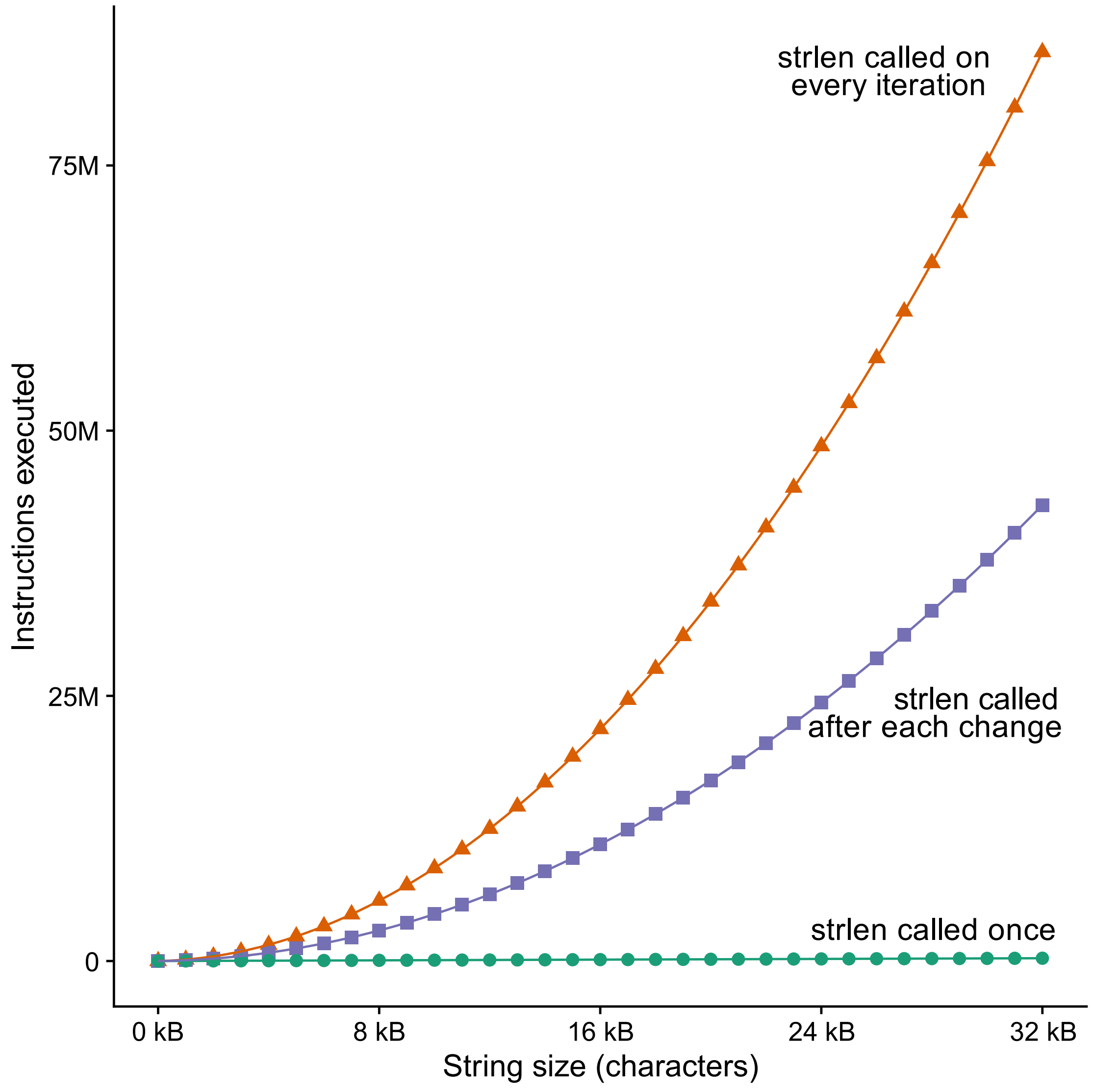
Can’t move functions out of loops

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



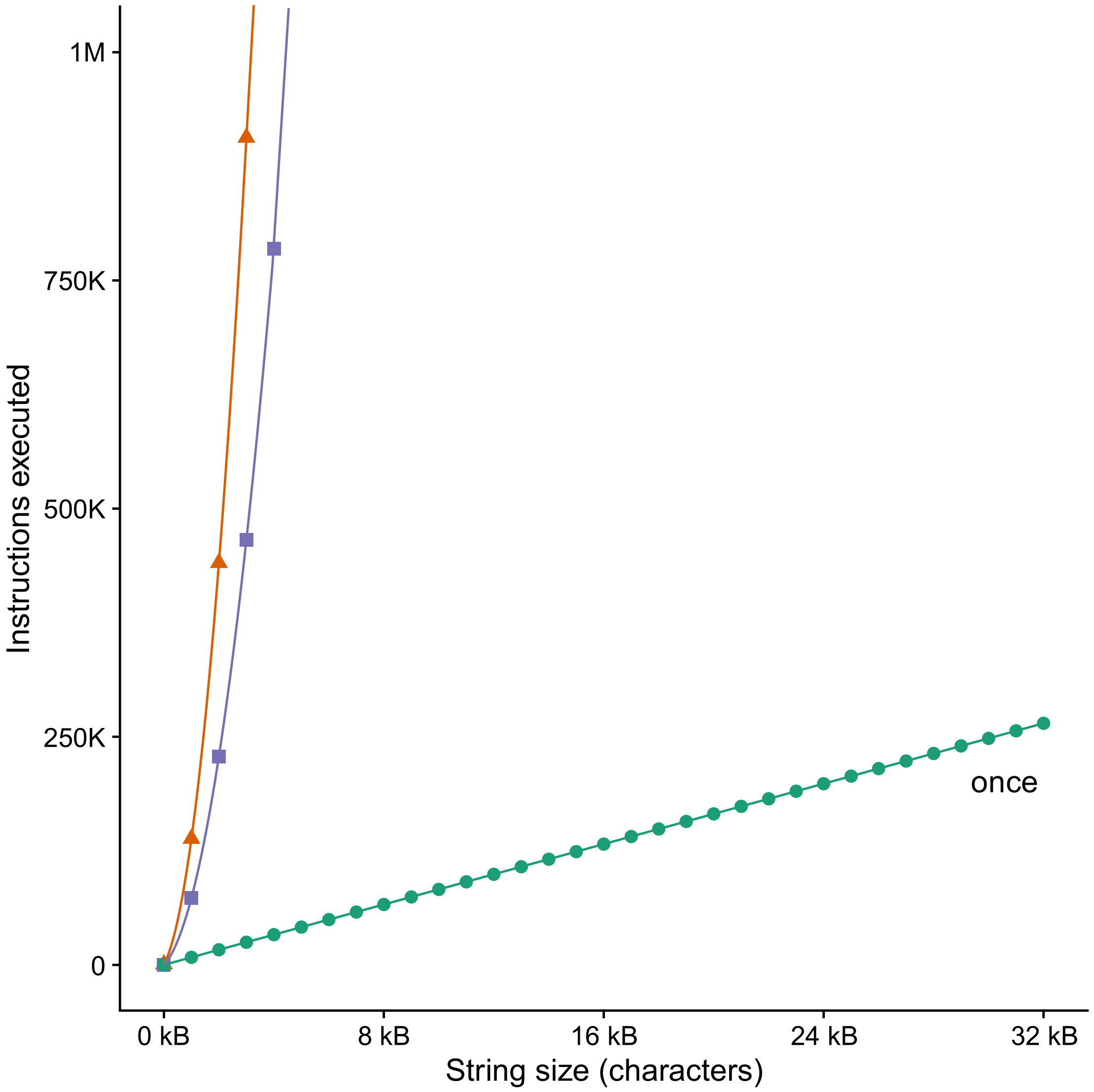
Can’t move functions out of loops

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



Can't move functions out of loops

- Even calling `strlen()` once is a linear function, it's just that the others are *terrible*
  - Zoom in here shows that
- Putting `strlen()` in the loop is a super common CS211 mistake
  - Although we let it slide



Optimization Challenges

1. Memory aliasing
2. Function calls
3. **Non-associative arithmetic**
4. Larger cache optimizations

Non-associative arithmetic

- When is $(\boldsymbol{a} \odot \boldsymbol{b}) \odot \boldsymbol{c}$ not equal to $\boldsymbol{a} \odot (\boldsymbol{b} \odot \boldsymbol{c})$?
  - *Floating-point numbers*

- Example: $\boldsymbol{a} = 1.0$, $\boldsymbol{b} = 1.5 \times 10^{38}$, $\boldsymbol{c} = -1.5 \times 10^{38}$ (single precision IEEE fp)

$$a + b = 1.5 \times 10^{38} \qquad (a + b) + c = 0$$

$$b + c = 0 \qquad a + (b + c) = 1$$

- Blocks any optimization that changes order of floating point operations

Optimization Challenges

1. Memory aliasing

2. Function calls

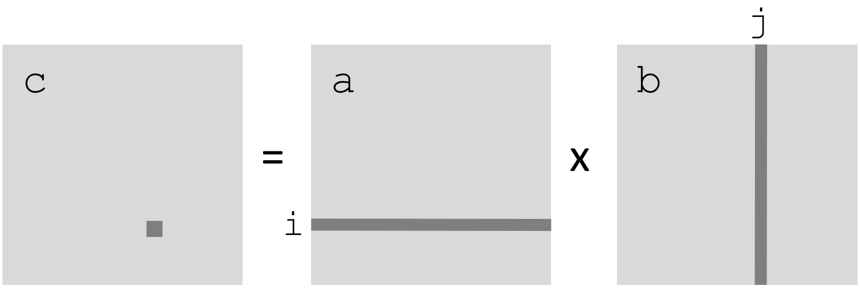
3. Non-associative arithmetic

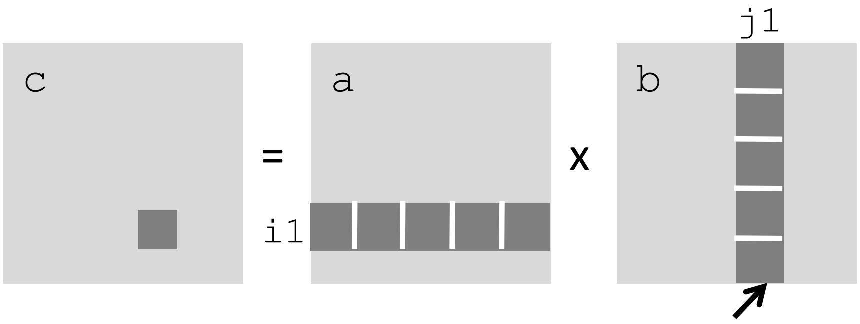
4. **Larger cache optimizations**

Larger cache optimizations

```
void mmm(double *a, double *b,
         double *c, int n) {
  memset(c, 0, n*n*sizeof(double));

  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];
}
```

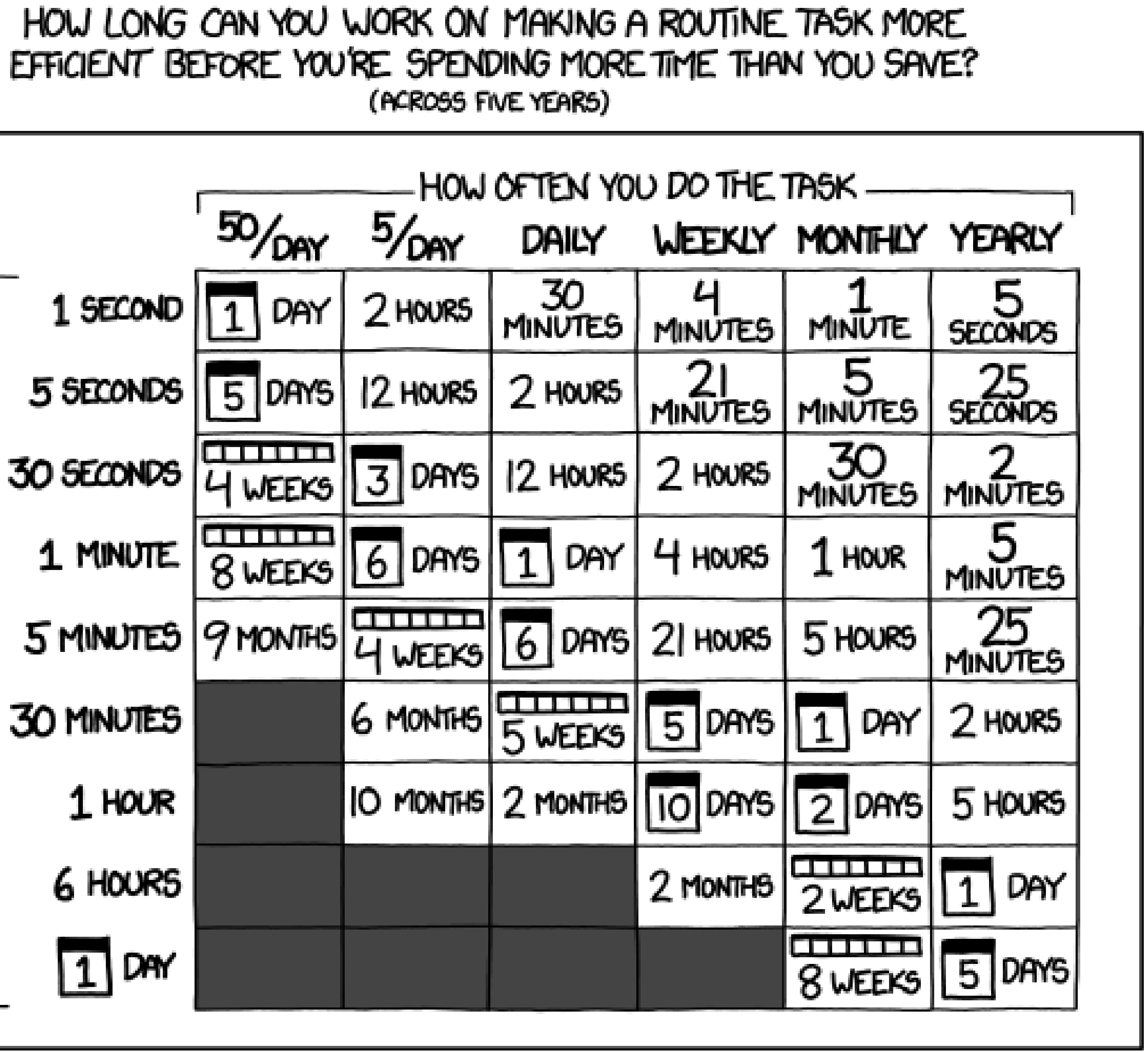
```
void mmm(double *a, double *b,
         double *c, int n) {
  memset(c, 0, n*n*sizeof(double));

  int i, j, k, i1, j1, k1;
  for (i = 0; i < n; i+=B)
    for (j = 0; j < n; j+=B)
      for (k = 0; k < n; k+=B)
        for (i1 = i; i1 < i+B; i1++)
          for (j1 = j; j1 < j+B; j1++)
            for (k1 = k; k1 < k+B; k1++)
             c[i1*n+j1] += a[i1*n + k1]
                         * b[k1*n + j1];
}
```





Compiler cannot do this transformation automatically

Break + Relevant xkcd



https://xkcd.com/1205/

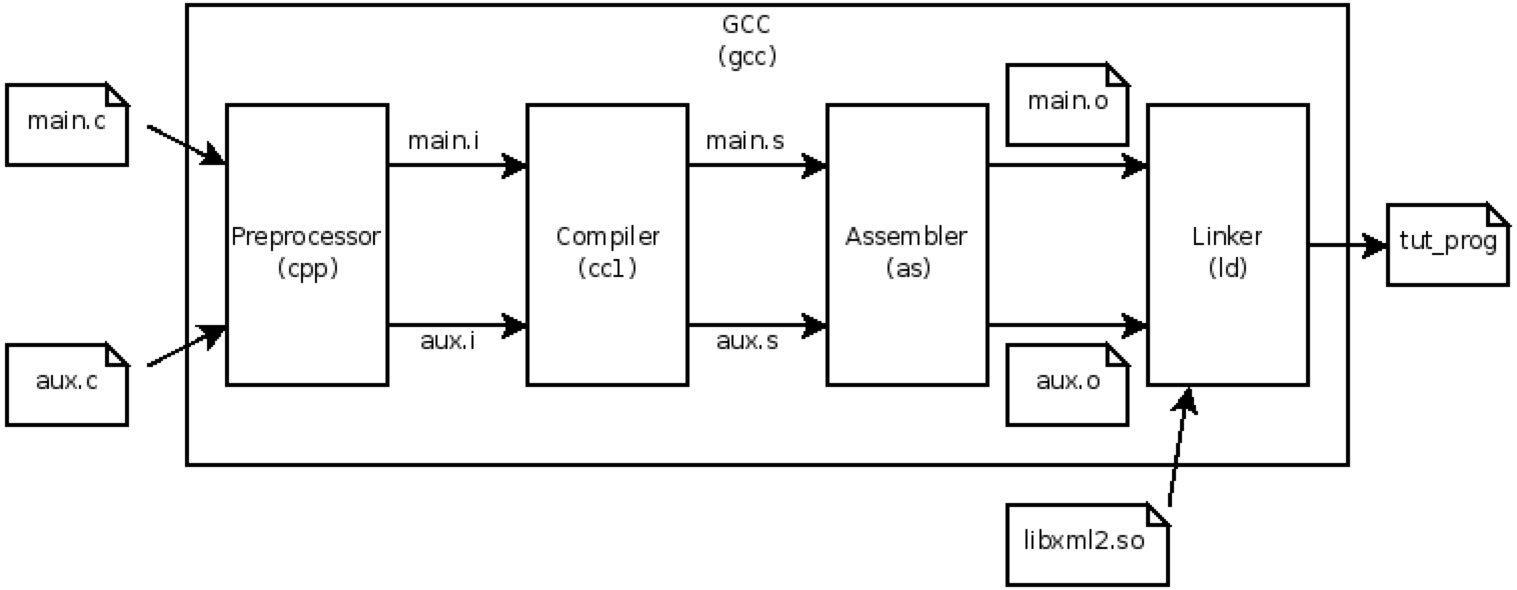
Outline

- Compilers and Optimizations
- Local Optimizations
- Global Optimizations
- Obstacles to Optimization
- **GNU C Compiler (GCC)**

GNU C Compiler (GCC)

- Very widely used compiler
  - Created in 1987
  - Originally just supported C, but now supports several languages
    - C, C++, Objective-C, Fortran, Ada, D, Go

- Collection of tools that perform the compilation steps



Enabling optimizations

- Flag given to gcc chooses optimization levels
  - `-O#` where `#` is one of `{0, 1, 2, 3, s}` (and a few custom others)
  - (that flag is a capital Oh for Optimization, not a zero)

- `-O0` is the default (oh zero)
  - Almost all optimizations are disabled
  - Code compiles more quickly!
  - Code does what you expect

More advanced optimizations

- Each level up from there is just a collection of optimizations

- -O1

```
-fauto-inc-dec
-fbranch-count-reg
-fcombine-stack-adjustments
-fcompare-elim
-fcprop-registers
-fdce
-fdefer-pop
-fdelayed-branch
-fdse
-fforward-propagate
-fguess-branch-probability
...
```

Explanation of optimizations:
https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html

Optimizations examples in godbolt

- Go to Godbolt!

Architecture-dependent optimizations

- By default, GCC knows which ISA you are compiling for
  - x86-64

- GCC does *not* know the specific processor you're compiling for
  - So it can make architecture-dependent choices
  - But it cannot make processor-dependent optimizations

- `-march=`*`cpu-type`*
  - Informs GCC of the specific processor you're on
  - Make sure you tell it the correct processor!
    - The wrong one might lead to code that crashes

Optimizations in SETI Lab

- Enable optimizations to start with
  - This should be enough to get you to 100%
  - Assuming you’ve got the concurrency part correct

- To achieve extra credit
  - Look into more advanced flags and what they do

  - Consider what optimizations you could perform on the code that the compiler cannot
    - Note: must focus these on the loops that are doing the most work

Be sure to apply optimizations to everything!

- Common SETI Lab bug: only apply optimizations to p_band_scan.c
  - In reality, much of the work is performed in the functions it calls to do signal processing

- Be sure to `make clean` and then recompile *everything* after enabling or changing optimizations

Compilers courses

- Is this lecture content interesting to you?
  - There is a LOT more depth here
  - Certainly more advanced optimizations
  - Also the idea of how does a compiler parse and understand your code

- Courses to consider:
  - CS322 – Compiler Construction
  - CS323 – Code Analysis and Transformation

Outline

- Compilers and Optimizations
- Local Optimizations
- Global Optimizations
- Obstacles to Optimization
- GNU C Compiler (GCC)