Code Optimization

Kai Zhang
Fudan University
zhangk@fudan.edu.cn

from https://csapp.cs.cmu.edu/3e/home.html

Code Optimization

- **Overview**
- **Generally Useful Optimizations**
  - Code motion/precomputation
  - Strength reduction
  - Sharing of common subexpressions
  - Example: Bubblesort
- **Optimization Blockers**
  - Procedure calls
  - Memory aliasing
- **Exploiting Instruction-Level Parallelism**
- **Dealing with Conditionals**

Performance Realities

- ***There's more to performance than asymptotic complexity (big O)***
- **Constant factors matter too!**
  - Easily see 10:1 performance range depending on how code is written
  - Must optimize at multiple levels:
    - algorithm, data representations, procedures, and loops

- **Must understand system to optimize performance**
  - How programs are compiled and executed
  - How modern processors + memory systems operate
  - How to measure program performance and identify bottlenecks
  - How to improve performance without destroying code modularity and generality

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

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 the 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;
    long ni = n*i;
    double *rowp = a+ni;
    for (j = 0; j < n; j++)
        *rowp++ = 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   -->    x << 4
  ```

  - Utility is machine dependent
  - Depends on cost of multiply or divide instruction
    - On Intel Nehalem, integer multiply requires 3 CPU cycles
- 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;
}
```

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

- The limitation of GCC

```
double a = 1e100;
double b = 1e100;
double c = 1.0;
double d = a - b + c;
double e = a + c - b;
printf("result is %lf, %lf\n", d, e);
```

```
// Result
result is 1.000000, 0.000000
```

- The reason is that floating point operations are not perfectly exact, and the order of the evaluation of an expression might matter.
- -ffast-math does not work on my computer

Optimization Example: Bubblesort

- **Bubblesort program that sorts an array `A` that is allocated in static storage:**
  - an element of `A` requires four bytes of a byte-addressed machine
  - elements of `A` are numbered 1 through `n` (`n` is a variable)
  - `A[j]` is in location `&A+4*(j-1)`

```
for (i = n-1; i >= 1; i--) {
    for (j = 1; j <= i; j++)
        if (A[j] > A[j+1]) {
            temp = A[j];
            A[j] = A[j+1];
            A[j+1] = temp;
        }
}
```

Translated (Pseudo) Code

```
      i := n-1
L5:   if i<1 goto L1
      j := 1
L4:   if j>i goto L2
      t1 := j-1
      t2 := 4*t1
      t3 := A[t2]     // A[j]
      t4 := j+1
      t5 := t4-1
      t6 := 4*t5
      t7 := A[t6]     // A[j+1]
      if t3<=t7 goto L3
```

```
      t8  := j-1
      t9  := 4*t8
      temp  := A[t9]    // temp:=A[j]
      t10  := j+1
      t11:= t10-1
      t12 := 4*t11
      t13  := A[t12]    // A[j+1]
      t14  := j-1
      t15  := 4*t14
      A[t15]  := t13    // A[j]:=A[j+1]
      t16  := j+1
      t17  := t16-1
      t18  := 4*t17
      A[t18]:=temp      // A[j+1]:=temp
L3:   j := j+1
      goto L4
L2:   i := i-1
      goto L5
L1:
```

```
for (i = n-1; i >= 1; i--) {
  for (j = 1; j <= i; j++)
    if (A[j] > A[j+1]) {
      temp = A[j];
      A[j] = A[j+1];
      A[j+1] = temp;
    }
}
```

Instructions

29 in outer loop

25 in inner loop

Redundancy in Address Calculation

```
     i := n-1
L5:  if i<1 goto L1
     j := 1
L4:  if j>i goto L2
     t1 := j-1
     t2 := 4*t1
     t3 := A[t2]      // A[j]
     t4 := j+1
     t5 := t4-1
     t6 := 4*t5
     t7 := A[t6]      // A[j+1]
     if t3<=t7 goto L3
```

```
     t8 :=j-1
     t9 := 4*t8
     temp := A[t9]    // temp:=A[j]
     t10 := j+1
     t11:= t10-1
     t12 := 4*t11
     t13 := A[t12]    // A[j+1]
     t14 := j-1
     t15 := 4*t14
     A[t15] := t13    // A[j]:=A[j+1]
     t16 := j+1
     t17 := t16-1
     t18 := 4*t17
     A[t18]:=temp     // A[j+1]:=temp
L3:  j := j+1
     goto L4
L2:  i := i-1
     goto L5
L1:
```

Redundancy Removed

```
     i  := n-1
L5:  if i<1 goto L1
     j  := 1
L4:  if j>i goto L2
     t1  := j-1
     t2  := 4*t1
     t3  := A[t2]      // A[j]
     t6  := 4*j
     t7  := A[t6]      // A[j+1]
     if t3<=t7 goto L3
```

```
     t8  :=j-1
     t9  := 4*t8
     temp  := A[t9]    // temp:=A[j]
     t12  := 4*j
     t13  := A[t12]    // A[j+1]
     A[t9]:= t13       // A[j]:=A[j+1]
     A[t12]:=temp      // A[j+1]:=temp
L3:  j  := j+1
     goto L4
L2:  i  := i-1
     goto L5
L1:
```

Instructions

20 in outer loop

16 in inner loop

More Redundancy

```
      i := n-1
L5:   if i<1 goto L1
      j := 1
L4:   if j>i goto L2
      t1 := j-1
      t2 := 4*t1
      t3 := A[t2]        // A[j]
      t6 := 4*j
      t7 := A[t6]        // A[j+1]
      if t3<=t7 goto L3
```

```
      t8 :=j-1
      t9 := 4*t8
      temp := A[t9]   // temp:=A[j]
      t12 := 4*j
      t13 := A[t12]   // A[j+1]
      A[t9]:= t13     // A[j]:=A[j+1]
      A[t12]:=temp    // A[j+1]:=temp
L3:   j := j+1
      goto L4
L2:   i := i-1
      goto L5
L1:
```

Redundancy Removed

```
    i := n-1
L5: if i<1 goto L1
    j := 1
L4: if j>i goto L2
    t1 := j-1
    t2 := 4*t1
    t3 := A[t2]      // old_A[j]
    t6 := 4*j
    t7 := A[t6]      // A[j+1]
    if t3<=t7 goto L3
    A[t2] := t7      // A[j]:=A[j+1]
    A[t6] := t3      // A[j+1]:=old_A[j]
L3: j := j+1
    goto L4
L2: i := i-1
    goto L5
L1:
```

Instructions

15 in outer loop

11 in inner loop

Redundancy in Loops

```
     i := n-1
L5:  if i<1 goto L1
     j := 1
L4:  if j>i goto L2
     t1 := j-1
     t2 := 4*t1
     t3 := A[t2]      // A[j]
     t6 := 4*j
     t7 := A[t6]      // A[j+1]
     if t3<=t7 goto L3
     A[t2] := t7
     A[t6] := t3
L3:  j := j+1
     goto L4
L2:  i := i-1
     goto L5
L1:
```

Multiply -> Plus

```
     i  := n-1
L5:  if i<1 goto L1
     j  := 1
L4:  if j>i goto L2
     t1  := j-1
     t2  := 4*t1
     t3  := A[t2]      // A[j]
     t6  := 4*j
     t7  := A[t6]      // A[j+1]
     if t3<=t7 goto L3
     A[t2]  := t7
     A[t6]  := t3
L3:  j  := j+1
     goto L4
L2:  i  := i-1
     goto L5
L1:
```

```
     i  := n-1
L5:  if i<1 goto L1
     t2  := 0
     t6  := 4
     t19  := 4*i
L4:  if t6>t19 goto L2
     t3  := A[t2]
     t7  := A[t6]
     if t3<=t7 goto L3
     A[t2]  := t7
     A[t6]  := t3
L3:  t2  := t2+4
     t6  := t6+4
     goto L4
L2:  i  := i-1
     goto L5
L1:
```

Final Pseudo Code

```
     i := n-1
L5:  if i<1 goto L1
     t2 := 0
     t6 := 4
     t19 := i << 2
L4:  if t6>t19 goto L2
     t3 := A[t2]
     t7 := A[t6]
     if t3<=t7 goto L3
     A[t2] := t7
     A[t6] := t3
L3:  t2 := t2+4
     t6 := t6+4
     goto L4
L2:  i := i-1
     goto L5
L1:
```

Instruction Count
Before Optimizations
29 in outer loop
25 in inner loop

Instruction Count
After Optimizations
15 in outer loop
9 in inner loop

- These were **Machine-Independent Optimizations**.
- Will be followed by **Machine-Dependent Optimizations**, including allocating temporaries to registers, converting to assembly code

Code Optimization

- **Overview**
- **Generally Useful Optimizations**
  - Code motion/precomputation
  - Strength reduction
  - Sharing of common subexpressions
  - Example: Bubblesort
- **Optimization Blockers**
  - Procedure calls
  - Memory aliasing
- **Exploiting Instruction-Level Parallelism**
- **Dealing with Conditionals**

Limitations of Optimizing Compilers

- **Operate under fundamental constraint**
  - Must not cause any change in program behavior
  - Often prevents it from making optimizations that would only affect behavior under pathological conditions

```
void twiddle1(long *x, long *y) {
        *x += *y;
        *x += *y;
}
```

```
void twiddle1(long *x, long *y) {
        *x += 2 * *y
}
```

when x == y, returns: 4x, 3x

- **Most analysis is performed only within procedures**
  - Whole-program analysis is too expensive in most cases
  - Newer versions of GCC do inter-procedural 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**

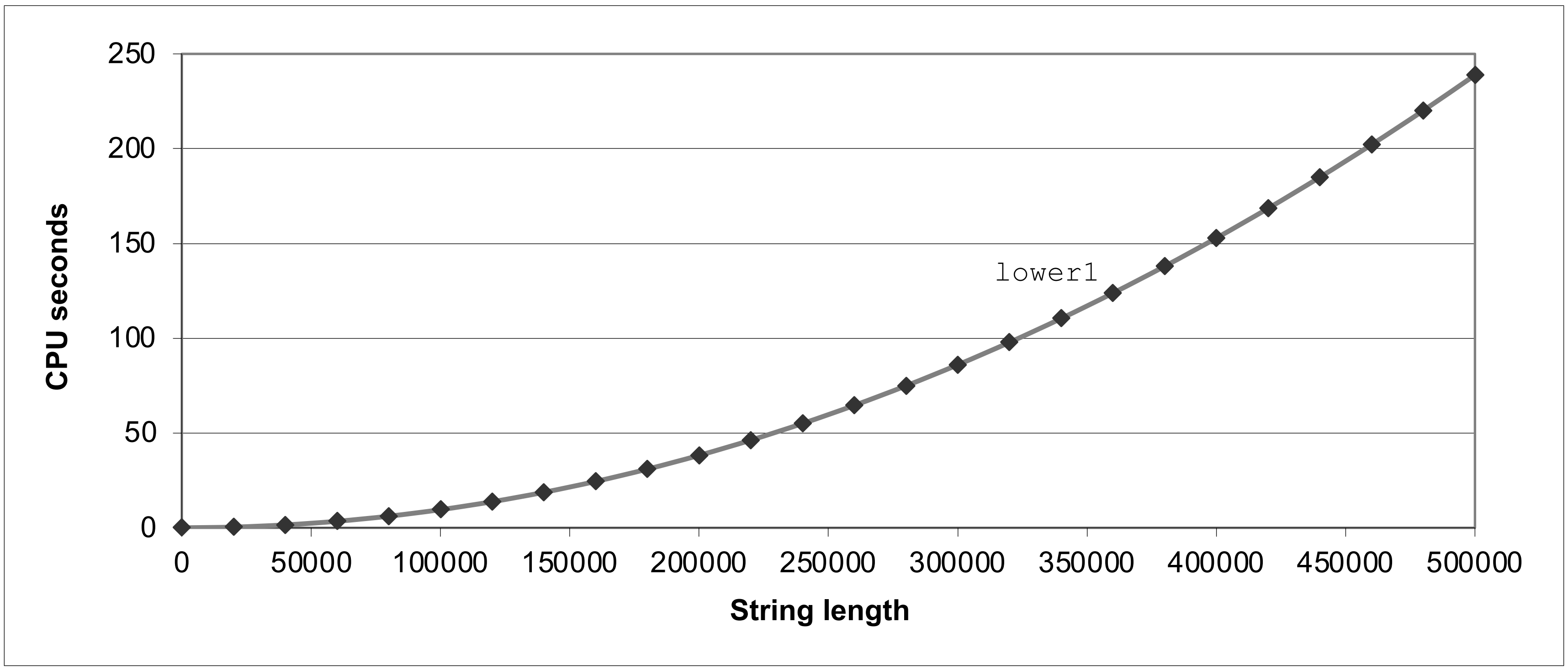
Optimization Blocker #1: Procedure Calls

- **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');
}
```

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

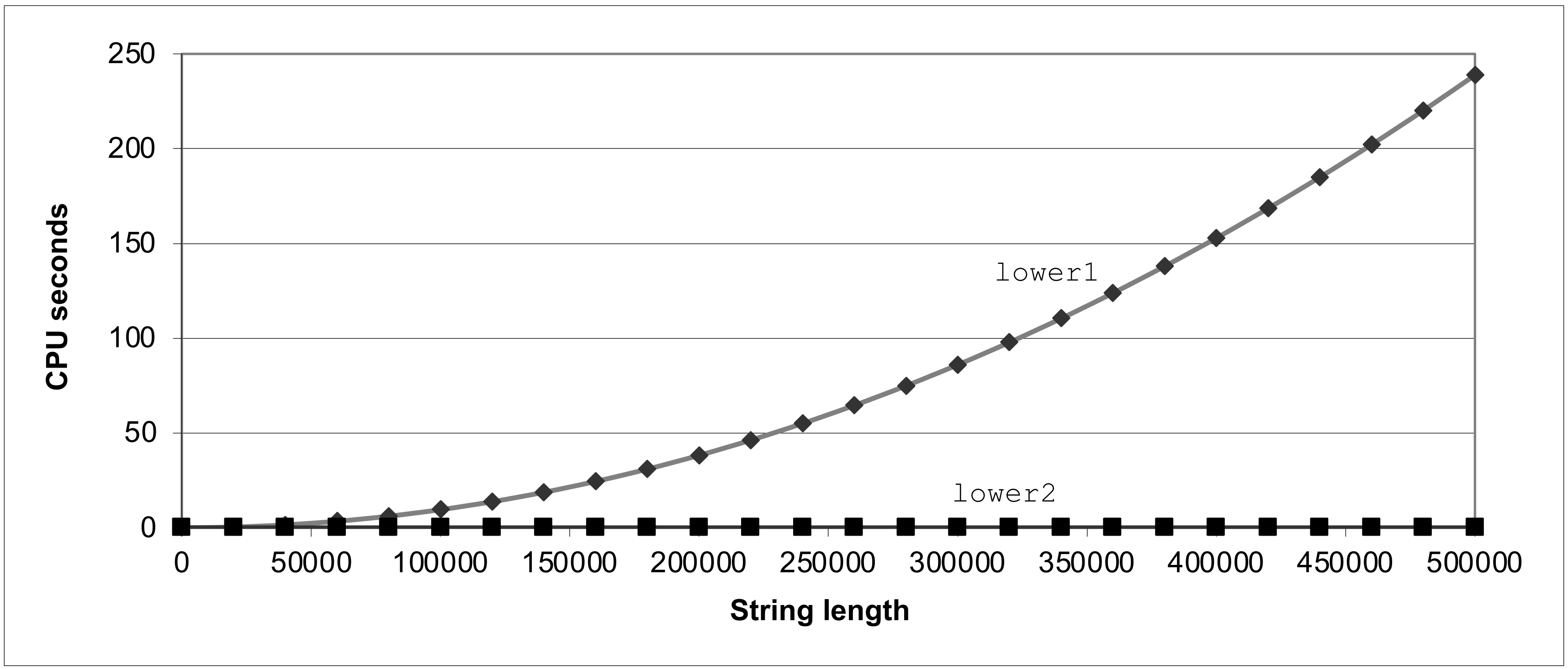
Improving Performance

```
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
- Legal 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 may treat procedure call as a black box
  - Weak optimizations near them
- **Remedies:**
  - Compiler: use of inline functions or macros
    - GCC does this with –O1
      - Within single file
  - Do your own code motion

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];
    }
}
```

```
# 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
```

- Code updates `b[i]` on every iteration
- Why couldn't compiler optimize this away, i.e., use a temp for the add and a final b[i] = temp in the end of the loop?

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];
    }
}
```

- Performance optimization?

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);
```

```
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]
```

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

Removing Aliasing

```
/* Sum rows is 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
.L10:
        addsd   (%rdi), %xmm0     # FP load + add
        addq    $8, %rdi
        cmpq    %rax, %rdi
        jne     .L10
```

- No need to store intermediate results in b[i]

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

Quiz

```
void foo1(int *array, int *size, int *value) {
    for(int i = 0; i < *size; ++i) {
        array[i] = 2 * *value;
    }
}
```

Expect that the compiler could load *value once outside the loop and then set every element in the array to that value very quickly?

```
void foo2(int *array, int size, int value) {
    for(int i = 0; i < size; ++i) {
        array[i] = 2 * value;
    }
}
```

```
foo1:
    .cfi_startproc
    cmpl    $0, (%rsi)
    jle .LBB0_3
    xorl    %eax, %eax
    .align  16, 0x90
.LBB0_2:
    movl    (%rdx), %ecx //load *value
    addl    %ecx, %ecx   // 2* *value
    movl    %ecx, (%rdi,%rax,4)
    incq    %rax
    cmpl    (%rsi), %eax
    jl  .LBB0_2
.LBB0_3:
    ret
    .size   foo, .Ltmp1-foo
    .cfi_endproc
.Leh_func_end0:
```

```
foo2:
    .cfi_startproc
    testl   %esi, %esi
    jle .LBB0_3
    addl    %edx, %edx  // 2 * value
    .align  16, 0x90
.LBB0_2:
    movl    %edx, (%rdi) //array[i]
    addq    $4, %rdi
    decl    %esi
    jne .LBB0_2
.LBB0_3:
    ret
    .size   foo, .Ltmp1-foo
    .cfi_endproc
.Leh_func_end0:
```

Code Optimization

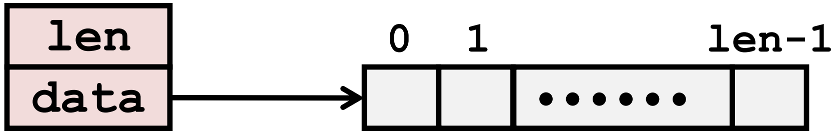
- **Overview**
- **Generally Useful Optimizations**
  - Code motion/precomputation
  - Strength reduction
  - Sharing of common subexpressions
  - Example: Bubblesort
- **Optimization Blockers**
  - Procedure calls
  - Memory aliasing
- **Exploiting Instruction-Level Parallelism**
- **Dealing with Conditionals**

Exploiting Instruction-Level Parallelism

- **Need general understanding of modern processor design**
  - Hardware can execute multiple instructions in parallel
- **Performance limited by data dependencies**
- **Simple transformations can yield dramatic performance improvement**
  - Compilers often cannot make these transformations
  - Lack of associativity and distributivity in floating-point arithmetic

Benchmark Example: Data Type for Vectors

```
/* data structure for vectors */
typedef struct{
    size_t len;
    data_t *data;
} vec;
```



- **Data Types**
  - Use different declarations for **data_t**
  - **int**
  - **long**
  - **float**
  - **double**

```
/* retrieve vector element and store
at val */

int get_vec_element
  (*vec v, size_t idx, data_t *val) {
    if (idx >= v->len)
        return 0;
    *val = v->data[idx];
    return 1;
}
```

Benchmark Computation

```
void combine1(vec_ptr v, data_t *dest)
{
    long int i;
    *dest = IDENT;
    for (i = 0; i < vec_length(v); i++) {
      data_t val;
      get_vec_element(v, i, &val);
      *dest = *dest OP val;
    }
}
```

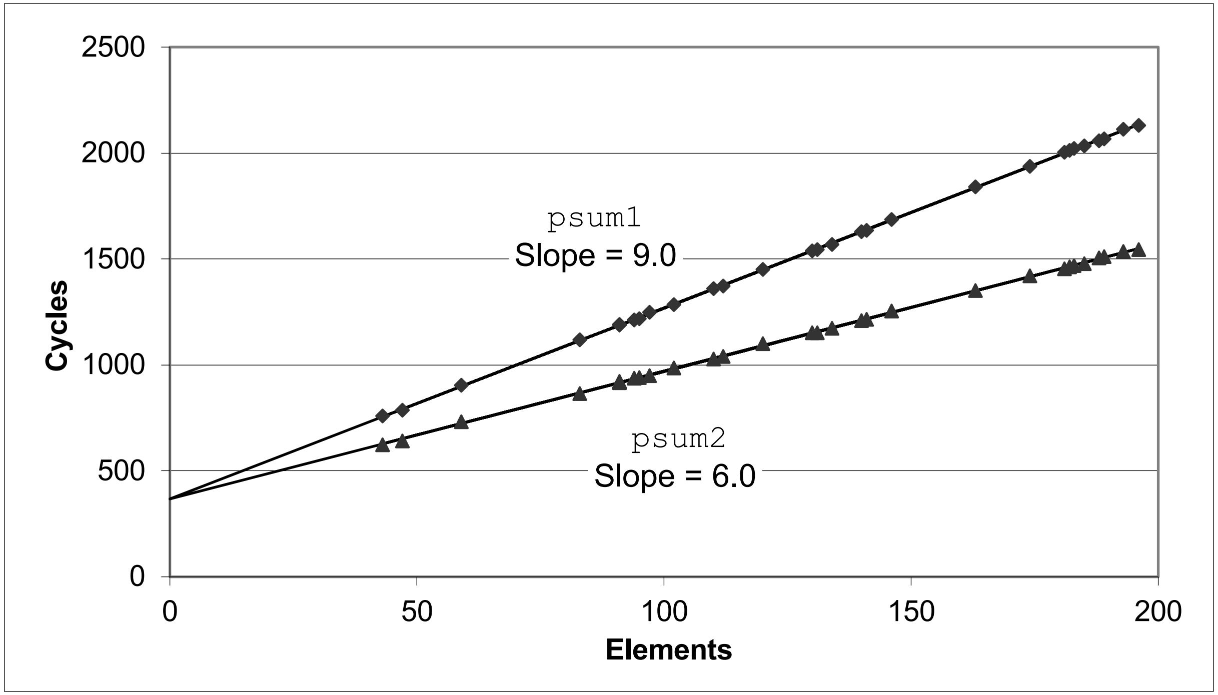
Compute sum or product of vector elements

- Data Types
  - Use different declarations for `data_t`
  - `int`
  - `long`
  - `float`
  - `double`
- Operations
  - Use different definitions of `OP` and `IDENT`
  - `+ / 0`
  - `* / 1`

Cycles Per Element (CPE)

- **Convenient way to express performance of program that operates on vectors or lists**
- **n = Length**
- **In our case: CPE = cycles per OP**
- **T = CPE * n + Overhead**
  - CPE is slope of line

How to draw this graph?



Benchmark Performance

```
void combine1(vec_ptr v, data_t *dest)
{
    long int i;
    *dest = IDENT;
    for (i = 0; i < vec_length(v); i++) {
       data_t val;
       get_vec_element(v, i, &val);
       *dest = *dest OP val;
    }
}
```

Compute sum or product of vector elements

Method	Integer		Double FP	
Operation	Add	Mult	Add	Mult
Combine1 unoptimized	22.68	20.02	19.98	20.18
Combine1 –O1	10.12	10.12	10.17	11.14
Combine1 –O3	4.5	4.5	6	7.8

Results in CPE (cycles per element)

Basic Optimizations

```
void combine4(vec_ptr v, data_t *dest)
{
  long i;
  long length = vec_length(v);
  data_t *d = get_vec_start(v);
  data_t t = IDENT;
  for (i = 0; i < length; i++)
    t = t OP d[i];
  *dest = t;
}
```

- **Move vec_length out of loop**
- **Avoid bounds check on each cycle**
- **Accumulate in temporary**

Effect of Basic Optimizations

```
void combine4(vec_ptr v, data_t *dest)
{
  long i;
  long length = vec_length(v);
  data_t *d = get_vec_start(v);
  data_t t = IDENT;
  for (i = 0; i < length; i++)
    t = t OP d[i];
  *dest = t;
}
```

Method	Integer		Double FP	
Operation	Add	Mult	Add	Mult
Combine1 –O1	10.12	10.12	10.17	11.14
Combine1 –O3	4.5	4.5	6	7.8
Combine4	1.27	3.01	3.01	5.01

- **Eliminates sources of overhead in loop**