Great Reality

There’s more to performance than asymptotic complexity

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 to measure program performance and identify bottlenecks
  ■ How to improve performance without destroying code modularity, generality, readability

Optimizing Compilers

Provide efficient mapping of program to machine
  ■ Register allocation
  ■ Code selection and ordering
  ■ 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
    ● E.g., O(N^2) sort is faster for 7 or fewer items

Have difficulty overcoming “optimization blockers”
  ■ Potential memory aliasing
  ■ Potential procedure side effects

Limitations of Optimizing Compilers

Compilers operate under fundamental constraint
  ■ Must not cause any change in program behavior under any possible condition
  ■ Often prevents optimizations that would only affect behavior in pathological situations

Behavior obvious to the programmer can be obfuscated by languages and coding styles
  ■ E.g., data ranges may be more limited than variable types suggest

Most analysis is performed only within procedures
  ■ Whole-program analysis is too expensive in most cases
  ■ (gcc does lots of interprocedural analysis—but not across files)

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

When in doubt, the compiler must be conservative
Generally Useful Optimizations

- Optimizations you 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
  - Gcc often does this for you (so check assembly)

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

for (i = 0; i < n; i++) {
    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)
{
    set_row:
    testq %rcx, %rcx # Test njle .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:njne .L3 # if !=, goto loop
    .L1: # done:
        ret
}
```

Strength Reduction

- Replace costly operation with simpler one
  - Shift, add instead of multiply or divide
    - Utility is machine-dependent
    - Depends on cost of multiply or divide instruction
      - On Intel Nehalem, integer multiply requires only 3 CPU cycles
    - Recognize sequence of products
      - Again, gcc often does it

```
for (i = 0; i < n; i++)
    for (j = 0; j < n; j++)
        a[n*i + 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 and up

```
/* 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
subq   %rcx, %rax     # i*n+j-n
leaq   (%rsi,%rcx), %rcx # i*n+j+n
```

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

```
leaq   (%rsi,%rcx), %rcx # i*n+j+n
```

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

```
for (i = 0; i < n; i++) {
    for (j = 0; j < n; j++)
        a[ni + j] = b[j];
    ni += n;
}
```
Optimization Blocker #1: Procedure Calls

Procedure to Convert String to Lower Case

```c
#include <ctype.h>
void lower(char *s) {
    size_t i;
    for (i = 0; i < strlen(s); i++)
        if (isupper(s[i]))
            s[i] = tolower(s[i]);
}
```

Extracted from many student programs

Lower-Case Conversion Performance

- Time quadruples when double string length
- Quadratic performance

Convert Loop To Goto Form

```c
void lower(char *s) {
    size_t i = 0;
    if (i >= strlen(s))
        goto done;
    loop:
    if (isupper(s[i]))
        s[i] = tolower(s[i]);
    i++;
    if (i < strlen(s))
        goto loop;
    done:
}
```

- `strlen` executed every iteration

Calling Strlen

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

- Only way to determine length of string is to scan its entirety, looking for NUL character.

Overall performance, string of length N
- N calls to strlen, each takes O(N) time
- Overall O(N^2) performance
Improving Performance

- Programmer moves call to `strlen` outside of loop
  - Since result does not change from one iteration to another
- Form of code motion
- Side comment: note lack of curly braces—why does this work?

```c
void lower(char *s) {
  size_t i;
  size_t len = strlen(s);
  for(i = 0; i < len; i++)
    if(isupper(s[i]))
      s[i] = tolower(s[i]);
}
```

Lower-Case Conversion Performance

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

![Graph showing linear performance of lower2](image)

Optimization Blocker: Procedure Calls

- Why couldn’t compiler move `strlen` out of inner loop?
  - Procedure may have side effects
    - Might alter 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 calls as a black box
- Weak optimizations near them

Remedies:
- Use inline functions
  - GCC does this with `-O1`
    - But only within single file
  - Do your own code motion

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

Memory Matters

- Code updates `b[i]` on every iteration
- Why couldn’t compiler use register and optimize this away?
Memory Aliasing

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

```
/* 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 = A+3;
```

Removing Aliasing

No need to store intermediate results
Also more likely to be what programmer wanted!

```
# sum_rows2 inner loop.
L10:
addsd (%rdi), %xmm0 # FP load + add
addq $8, %rdi cmpq %rax, %rdi
jne .L10
```

```
/* 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;
    }
}
```

Optimization Blocker: Memory Aliasing

Aliasing

- Two different memory references specify single location
- Easy to have happen in C
  - Since allowed to do address arithmetic
  - Language allows direct access to storage structures
- Get in habit of introducing local variables
  - E.g., accumulating within loops
- Your way of telling compiler not to check for aliasing

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

```
/* retrieves 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 Example: Data Type for Vectors

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

```c
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`
  - `+` and `0`
  - `*` and `1`

**Benchmark Performance**

```c
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

<table>
<thead>
<tr>
<th>Method</th>
<th>Integer</th>
<th>Double FP</th>
</tr>
</thead>
<tbody>
<tr>
<td>Operation</td>
<td>Add</td>
<td>Add</td>
</tr>
<tr>
<td>Combine1</td>
<td>22.68</td>
<td>19.98</td>
</tr>
<tr>
<td>Combine1 –O1</td>
<td>10.12</td>
<td>10.17</td>
</tr>
</tbody>
</table>

**Cycles Per Element (CPE)**

Convenient way to express performance of program that operates on vectors or lists

**Length** = \( n \)

In our case: 

\[ T = \text{CPE} \times n + \text{Overhead} \]

- CPE is slope of line

**Basic Optimizations**

Move `vec_length` out of loop

Avoid bounds check on each cycle

Accumulate in temporary
Effect of Basic Optimizations

```c
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;
}
```

<table>
<thead>
<tr>
<th>Method</th>
<th>Integer</th>
<th>Double FP</th>
</tr>
</thead>
<tbody>
<tr>
<td>Operation</td>
<td>Add</td>
<td>Mult</td>
</tr>
<tr>
<td>Combine1 -O1</td>
<td>10.12</td>
<td>10.12</td>
</tr>
<tr>
<td>Combine4</td>
<td>1.27</td>
<td>3.01</td>
</tr>
</tbody>
</table>

Eliminates sources of overhead in loop

Exploiting Instruction-Level Parallelism

We can go farther!

But 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

We’ll talk about that next time