Performance Realities

- 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 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
Limitations of Optimizing Compilers

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

- **Behavior that may be 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

- **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 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

```c
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];
}
```
Compiler-Generated Code Motion

```c
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];
}
```

```asm
set_row:
    testq    %rcx, %rcx  # Test n
    jle      .L4         # If 0, goto done
    movq     %rcx, %rax  # rax = n
    imulq    %rdx, %rax  # rax *= i
    leaq (,%rdi,%rax,8), %rdx # rowp = A + n*i*8
    movl $0, %r8d       # j = 0
    .L3:                 # loop:
        movq (%rsi,%r8,8), %rax # t = b[j]
        movq %rax, (%rdx) # *rowp = t
        addq $1, %r8        # j++
        addq $8, %rdx       # rowp++
        cmpq %r8, %rcx      # Compare n:j
        jg .L3               # If >, goto loop
    .L4:                 # done:
        rep ; ret
```

long j;
long ni = n*i;
double *rowp = a+ni;
for (j = 0; j < n; j++)
    *rowp++ = b[j];
**Reduction in Strength**

- Replace costly operation with simpler one
- Shift, add instead of multiply or divide
  \[16 \times x \quad \rightarrow \quad x \ll 4\]
  - Utility machine dependent
  - Depends on cost of multiply or divide instruction
    - On Intel Nehalem, integer multiply requires 3 CPU cycles
- Recognize sequence of products

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

```c
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
- Compilers often not very sophisticated in exploiting arithmetic properties

/* 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

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
Optimization Blocker #1: Procedure Calls

- Procedure to Convert String to Lower Case

```c
void lower(char *s)
{
    int 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

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

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

![Graph showing CPU seconds vs string length for lower and lower2 conversion times.](image-url)
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
  - Weak optimizations near them

- **Remedies:**
  - Use of `inline` functions
    - GCC does this with `-O2`
    - See web aside ASM:OPT
  - Do your own code motion
```c
int select(int x, int y, int i) {
    int data[2] = { x, y };
    if (i >= 0 && i < 2)
        return data[i];
    else
        return 0;
}

int test_select() {
    return select(5, 6, 1);
}
```

**Function test_select, optimized -O1**

```assembly
test_select:
    pushl %ebp
    movl %esp, %ebp
    subl $12, %esp       # Allocate 12 bytes on stack
    movl $1, 8(%esp)    # Set 1 as 3rd argument
    movl $6, 4(%esp)    # Set 6 as 2nd argument
    movl $5, (%esp)     # Set 5 as 1st argument
    call select         # Call select(5,6,1)
    leave
    ret
```
1 int select(int x, int y, int i) {
 2   int data[2] = { x, y };;
 3   if (i >= 0 && i < 2)
 4     return data[i];
 5   else
 6     return 0;
 7 }

Function select, optimized -O3
x at %ebp+8, y at %ebp+12, i at %ebp+16
1 int test_select() {
 2   return select(5, 6, 1);
 3 }

1 test_select:
2   pushl %ebp
3   movl $6, %eax           Set result = 6
4   movl %esp, %ebp
5   popl %ebp
6   ret
Inlining

- **Good**
  - Can be much faster

- **Bad**
  - Harder to debug
  - Code size can increase
  - Hurts modularity