Code optimization:
- Reducing number of operations
- Reducing memory usage
- Making your program run faster

Elias Rudberg
elias.rudberg@it.uu.se
Priorities: correctness comes first!

Priorities:
• 1: Correctness
• 2: Flexibility
• 3: Performance
Do profiling first!

- Always measure performance of different parts of the code before doing optimizations

Remember:

*Premature optimization is the root of all evil*
Always test if attempted optimizations give improvements

• Is there any real benefit? TEST!
  – If results almost unchanged, go for the simpler version
Remember to check correctness while optimizing

• Always check that you still get correct results

• Use both unit-tests and test runs that check final results for “real” test cases
Optimize on multiple levels

- Algorithms
- Data representations
- Procedures
- Loops

macro- vs. micro-optimization
How to reduce memory usage?

- Use a good algorithm
- Think about scaling, e.g. $N$ vs $N^2$
- Choose data structures carefully
- Avoid storing any redundant information
- Example: sparse matrix storage, e.g. compressed sparse row (CSR) format.
The computer

- Central Processing Unit (CPU)
- Memory
- Hard disk
Some simple advice

- Optimizations you should do regardless of processor type
  - Some of these can be performed by the compiler to varying extent
Arrays

Array Access Example:

```c
int get_digit(int* z, int dig) {
    return z[dig];
}
```

- Register reg0 contains starting address of array
- Register reg1 contains array index
- Desired digit at 4*reg1 + reg0

Memory Reference Code:

```c
# reg0 = z
# reg1 = dig
store 4*reg1 in reg2
add reg2 to reg0  # reg0 = 4*reg1 + reg0
store value at address reg0 to reg3  # Mem[4*reg1+reg0]
```
Array Loop Example

- Original Source

```c
int zd2int(int* z)
{
    int i;
    int zi = 0;
    for (i = 0; i < 5; i++) {
        zi = 10 * zi + z[i];
    }
    return zi;
}
```

- Transformed Version

```c
int zd2int(int* z)
{
    int zi = 0;
    int *zend = z + 4;
    do {
        zi = 10 * zi + *z;
        z++;
    } while(z <= zend);
    return zi;
}
```

- As generated by GCC
- Eliminate loop variable i
- Convert array code to pointer code
- Express in do-while form
- No need to test at entrance
Multidimensional arrays

- Memory is one-dimensional
- Multidimensional arrays need to be mapped onto a one-dimensional memory

- In C, we have two alternatives:
  1. Nested arrays $a[n*j+i]$;
  - One memory block with an indexing convention
  - Static or dynamic allocation
  2. Multi-level arrays $a[i][j]$;
  - Pointers to pointers
  - Static or dynamic allocation
Element Access in multidimensional array

Computation

- Element access

\[ \text{Mem[Mem[ptr+4*i]+4*j]} \]

- Must do two memory reads
  - First get pointer to row array
  - Then access element within array
struct point
{
    float x;
    float y;
    char debugname[255];
} points[10000];

• Or:

float pointxs[10000];
float pointys[10000];
char pointdebugnames[10000][255];

• Or:

float points[10000][2];
...
The Heap

- When you request memory using the standard C library functions, it will be placed in an area of the virtual address space called the heap.
- Programs that do a lot of mallocs and frees can fragment the heap.
- If you do not match each malloc() with a free(), you have created a memory leak.
Problems of Heap Allocation

• Extra bookkeeping is "stealing" CPU time and memory
  Header of 4, 8, 16 bytes per allocation

• Alignment

• Fragmentation

• Allocation and deallocation might involve iterating over a list

• Allocate larger blocks, granular deallocation might not be needed
Optimizing Compilers

- Most compilers support optimizations
- Standard optimization flag is -O
- Measure performance using different optimization flags. "highest level" of opt. not always best!
- For example, with gcc compiler, try compiler optimization flags -O2, -O3, -funroll-loops and -funroll-all-loops
- See “man gcc” for more
Limitations of Optimizing Compilers

- Behavior that may be obvious to the programmer can be obfuscated by languages and coding styles
- Most analysis is performed only within functions -- whole-program analysis is too expensive in most cases
- When in doubt, the compiler must be conservative
Avoid too many "small" function calls

Inlining:

- Saves function call overhead
- Same as preprocessor macro but with type checking
- C99 (and gcc) has an inline keyword
- (can also be controlled using flags)

```
gcc: -finline-functions

inline int get_num_rows(Matrix m) {
    return m->num_rows;
}
```
Why do function calls give overhead?

Status before function call saved on stack, to be restored after function call

Parameters passed via stack
Effects of inlining:

- The function code is appearing in multiple places
- Simple inlining is only possible if the function is defined within the same compilation unit -- Put candidates for inlining in header files
- Too much inlining can give too large code segments, can be bad for performance.

```cpp
inline int get_num_rows(Matrix m) {
    return m->num_rows;
}
```
Reduce frequency with which a computation is performed

- If it will always produce the same result
- Loop invariant
- Especially moving code out of the innermost loop
- Not only full lines of code

```
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];
}
```
Strength Reduction

- Replace a costly operation with a simpler one
- Shift, add instead of multiply or divide
- Recognize sequence of products, turn into addition
- Replace divisions by multiplication with the reciprocal (computed once)
- Replace pow() function with an algebraic expression (Compiler may not know what pow does, can modify a, i or j)
for(i=0; i<N; i++) {
    for(j=0; j<M; j++) {
        a[i][j] = (a[i+1][j]-a[i-1][j])/pow(dx,2.0);
    }
}

temp = 1.0/(dx*dx);
for(i=0; i<N; i++) {
    for(j=0; j<M; j++) {
        a[i][j] = temp*(a[i+1][j]-a[i-1][j]);
    }
}
Summary

• Priorities: correctness comes first!
• Premature optimization is root of all evil
• Do optimizations give any effect? Test!
• Choose data structures carefully
• Explore compiler optimization flags
• Avoid too many small function calls
• Move computations out of inner loops whenever possible