High Performance Programming, Lecture 4

Optimization I: reducing instructions

Reminders

Assignment 1 deadline **tomorrow**!

Assignment 2 deadline next Friday.
Assignment 3 available on Monday.

Example codes in Student Portal under "Example programs":
`pointers`
`file_io`
`static_and_dynamic_mem_alloc`
`function_pointers`

– use online help (search for documentation, examples, specific questions)

Optimization overview

Program performance can be improved by:

- I: **Doing less work** $\longleftarrow$ this lecture!
- II: Waiting less for data
- III: Doing the work faster
- IV: Using less space (to fit a bigger problem)

Reading Material

This module (doing less work):

- Fog 7.2, 7.3, 7.5, 7.13, 7.34
- Fog 14.1-7

Next module (memory usage):

- Fog 7.1, Ch 9 (except 9.7)
- Pacheco 2.1, 2.2

Optimization I

How to avoid work:

- Do as little as possible inside loops
- Use faster boolean evaluations (lab 2)
- Reduce function call overhead with inlining
- Avoid denormalized floating-point numbers
- Strength reduction – use cheap operations when possible

Compiler optimizations

Always start by turning on and exploring compiler optimizations, and only then attempt to hand-optimize.

(except sometimes in our labs and lectures where we sometimes hand-optimize anyway to understand things)

Compiler optimizations

Some commonly used compiler optimizations flags:

- **-O0** Turn off all optimization (default)
- **-O1** Do some optimization
- **-O2** Optimize more
- **-O3** Optimize even more
- **-Ofast** As -O3 but goes even further, *disregarding strict standards compliance* (may reduce accuracy)
- **-Os** Optimize for size, enables all -O2 optimizations except those that often increase code size

See GCC manual for details, and many other more detailed options: `https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html`

Assembly code

We can ask the compiler to output a text file with generated assembly code, using the "`-S`" compiler option.

Although we will not be doing any assembly language programming in this course, looking at the assembly code output can help us understand what the compiler is doing.

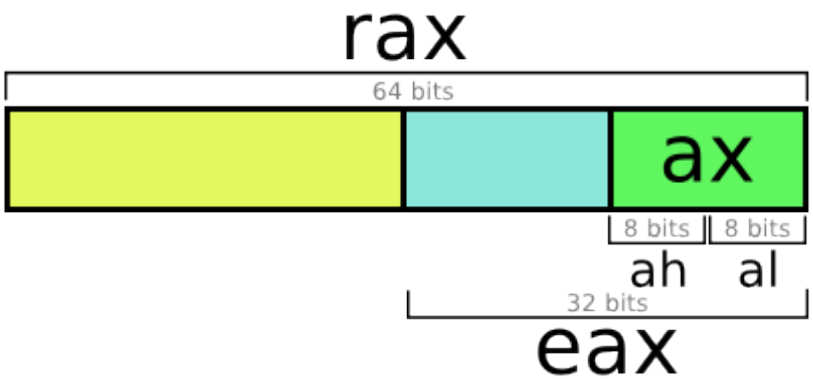
We focus on the industry-standard **x86 instruction set**, supported by both Intel and AMD processors.

A very short assembly primer

"Assembly code" for x86 is usually written in **AT&T** syntax (this is what gcc outputs), in **Intel** syntax (in x86 documentation and in Windows environments), or **NASM** (Netwide Assembler).

x86-64 has 16 64-bit integer registers (manipulate integers, including memory addresses):

rax rbx rcx rdx rsi rdi rbp rsp r8 – r15



Floating point registers

- xmm0-7 SSE floating point registers (128 bit)

AT&T Syntax versus Intel Syntax

GNU tools use the traditional AT&T syntax, which is used across many processors on Unix-like operating systems
Intel syntax typically used on DOS and Windows systems.

AT&T syntax: `movl %esp, %ebp`

In the AT&T syntax, the source is always given first, and the destination is always given second.

Intel syntax: `mov ebp, esp`

In the Intel syntax, the destination is always given first, and the source is always given second.

Examples in AT&T [Intel]

Example: load a stack variable from the address rbp-4 to register eax:

```
movl -4(%rbp), %eax      [mov eax, DWORD PTR [rbp-4]]
```

Example: load the constant number 8 to register eax:

```
movl $8, %eax      [mov eax, 8]
```

Operation suffixes in AT&T syntax determine the size of the operands:
byte, **s**hort, **w**ord, **l**ong, **q**uad.

Loops and control flow is managed with jump instructions, e.g.

```
cmpl %eax, %ebx      [ cmp ebx, eax]
je LABEL      [ je LABEL]
```

Execution will jump to LABEL if *eax == *ebx.

Examples in AT&T [Intel]

Some example arithmetic instructions:

Divide scalar double precision in xmm0 by that in xmm1, store result in xmm1:

```
divsd %xmm0, %xmm1      [ divsd %xmm1, %xmm0]
```

Add 1 to a value at the stack address rbp-8 and save it at the stack address rbp-8:

```
addl $1, -8(%rbp)   [add DWORD PTR [rbp-8], 1]
```

Examples of assembly code instructions

Examples of assembly code instructions:

- operations like mul, div, add (multiply, divide, add)
- function calls: "call"
- return from function: "ret"
- comparisons and branches: "cmp" followed by
  - je (jump if equal)
  - jne (jump if not equal)
  - jg (jump if greater than)
  - jge (jump if greater than or equal to)
  - jl (jump if less than)
  - jle (jump if less than or equal to)

Assembly code, why look at it?

- Helps us **understand what the compiler is doing**

- The compiler is your friend, good to understand it to have a friendly relationship!

How to get assembly code output?

- How to get assembly code output? $\longrightarrow$ use `-S` compiler option!

- How to make it more verbose? $\longrightarrow$ `-fverbose-asm`

- How to choose asm syntax/dialect? $\longrightarrow$ `-masm=att` or `-masm=intel` (default is `att`)

Use `https://gcc.godbolt.org` to see what compilers do!

Usefulness of assembly code output

Examples of questions that can often be answered by looking at the assembly code:

- was a function call made or was the function inlined?

- was a loop unrolled? by what unroll factor?

- was a loop completely optimized away?

- how are function input arguments passed?

Loop optimization

- Identify loop invariants (something that appears in a loop but does not change with the loop variable)
- Use constant loop bounds & integer counters
- Micro-optimizations aren't generally worthwhile (let compiler do it)
- Optimization attempts can have different effect depending on compiler and hardware

The compiler is often (not always) good at:

- identifying loop invariants
- micro-optimizations
- loop unrolling

Examples

Use `https://gcc.godbolt.org` to see what compilers do!

We consider two programs:
`string_loop.c`
`array_loop.c`

You can toggle between the "fast" version and the "slow" version by redefining the FAST preprocessor macro on line 4.

Try to compile without optimization flags: `gcc string_loop.c`. Try change compilers and add optimization options! For example, try to compile with -O3 option. Can you see the difference in the assembly between slow and fast version?

string_loop.c
(Compiler explorer link: https://gcc.godbolt.org/z/zoT7YP)

```
#if FAST

void lowercase(char* p) {
   int i;
   for (i = 100; i>0; i--)
        *(p++) |= 0x20;
}

#else

void lowercase(char* p) {
   while (*p != 0)
        *(p++) |= 0x20;
}

#endif
```

`string_loop.c`

In slow version `*p != 0` translates to 2 memory operations and 1 compare.

In fast version `for` loop translates to 1 subtract and 1 compare.

Try to use optimization flag `-O3`.

Compare compilers: gcc 7.3 and gcc 8.2.

array_loop.c
(Compiler explorer link: https://gcc.godbolt.org/z/CfGsJb)

```
#if FAST
int digits2int(int* z, int N) {
   int zi = 0;
   int *zend = z + N-1;
   while(z < zend){
      zi = 10*zi + *z;
      z++;
   }
   return zi;
}
#else
int digits2int(int* z, int N) {
   int i;
   int zi = 0;
   for(i = 0; i<N; i++) {
      zi = 10 * zi + z[i];
   }
   return zi;
}
#endif
```

`array_loop.c`
Difference:
– z[i] vs *z
– for loop requires extra addition (i++) – one loop iteration is removed by using do...while loop

Try to use `-O3` flag.

Integer representation

Two's complement

Example (Two's complement):

```
char a = 3;  // 00000011
a = -3;      // 00000011 -> 11111100 -> 11111101
unsigned char b = (unsigned char) a;
b == 253;
```

Integer representation

Two's complement

Binary	hex	unsigned	signed
1111 1111	0xff	255	-1
1111 1110	0xfe	254	-2
1111 1101	0xfd	253	-3
1000 0010	0x82	130	-126
1000 0001	0x81	129	-127
1000 0000	0x80	128	-128
0111 1111	0x7f	127	127
0111 1110	0x7e	126	126
0000 0011	0x03	3	3
0000 0010	0x02	2	2
0000 0001	0x01	1	1
0000 0000	0x00	0	0

Bounds checking (Example 14.4 (Fog))

```
const int size = 16; int i; float list[size];
...
if (i < 0 || i >= size) {
cout << "Error: Index out of range";
}
else {
list[i] += 1.0f;
}
```

The two comparisons can be replaced by a single comparison (type conversion generates no extra code at all):

```
if ((unsigned int)i >= (unsigned int)size) {
cout << "Error: Index out of range";
}
```

Floating point numbers

Normalize number is written as $1.lalala \cdot 2^{exponent}$, where 1.lalala is called mantissa. Mantissa is in range $[1, 2[$.

The value of exponent is the value of the exponent bits as an unsigned integer minus a fixed bias: 127 for single precision, 1023 for double. Mantissa is stored without leading 1.

- Single precision: 4 bytes/32 bits (range $\pm 1.18 \cdot 10^{-38}$ to $\pm 3.4 \cdot 10^{38}$)

```
    1          2------9    10------32
  sign        exponent     mantissa
```

- Double precision: 8 bytes/64 bits (range $\pm 2.23 \cdot 10^{-308}$ to $\pm 1.80 \cdot 10^{308}$)

Example:
$-13.6875_{10} = -1101.1011_2 = -1.1011011 \cdot 2^3$
$3 + 127 = 130_{10} = 10000010_2$
1 10000010 10110110000000000000000

Subnormal numbers

Also known as denormalized numbers

Subnormal numbers are numbers like $0.lalala \cdot 2^e$, where e=-127 for single and -1023 for double precision:
-5.87747210^{-39} is 0 00000000 10000000000000000000000

This gives you:

- really, *really* small numbers (smallest number: $7.0 \cdot 10^{-46}$ for single and $2.5 \cdot 10^{-324}$ for double precision)
- gradual underflow
- slow arithmetic

Some compiler optimization options (`-Ofast` for gcc) cause denormalized numbers to be flushed (rounded) to 0 instead. That avoids the performance problem (but may give an accuracy problem instead).

Machine precision

Machine precision is an upper bound on the relative error due to rounding in floating point operations. Defined as the distance between 1 and the next floating point number.

Consider:

0 01111111 00000000000000000000000
0 01111111 00000000000000000000001

Single precision: $2^{-23} \approx 1.19 \cdot 10^{-7}$
Double precision: $2^{-52} \approx 2.22 \cdot 10^{-16}$

Machine precision

```c
#include<stdio.h>
#include<float.h>

int main(){
    printf("Real precision = %e\n", FLT_EPSILON);
    printf("Double precision = %e\n", DBL_EPSILON);
}


//Real precision = 1.192093e-07
//Double precision = 2.220446e-16
```

Check example code `find_epsilon.c` in Student Portal.

Strength Reduction

Reformulate your arithmetic to make it cheaper:

- Integer +, -, <<, && faster than:
- Integer *, FP +, - faster than:
- FP * faster than:
- FP / faster than:
- Integer /, %

(Above, FP means "floating-point".)

Cost of arithmetic operations

Example code

Test code `cost_of_fp_ops.c` available under "Example programs" in Student Portal.

Performs the same number of operations for different kinds of artithmetic operations:

- Integer addition
- Floating-point addition
- Floating-point subtraction (cost same as for addition)
- Floating-point multiplication
- Floating-point division

Prints time taken for each case.

Cost of arithmetic operations

Example timings

Test code `cost_of_fp_ops.c` available in "Example programs", compiled with `-O3`.

	Computer A	Computer B	Computer C	Computer D
Int add	0.041	0.125	0.127	0.157
FP add	0.372	0.325	0.253	0.954
FP mul	0.622	0.326	0.398	0.955
FP div	2.470	3.016	3.507	4.969

A: laptop: Intel Core i7-4600U @ 2.10GHz, gcc 8.2.0
B: desktop machine: AMD A8-3820 APU, gcc 6.2.0
C: cronstedt.it.uu.se: Intel Xeon E5520 @ 2.27GHz, gcc 4.8.4
D: gullviva.it.uu.se: AMD Opteron 6274, gcc 4.4.7

How to find out processor model?

When reporting performance results/timings, important to include information about the processor model used. Saying just "Intel" or "Intel i7" is not enough, **include full model name** like "Intel Core i7-2670QM".

Different ways to find out what processor model you have:

```
cat /proc/cpuinfo
```

or:

```
lscpu
```

(Sometimes lscpu does not give all info you want, checking /proc/cpuinfo may give more info.)

inline functions

Each function call incurs an overhead cost due to copying arguments and return value data to/from the stack and/or registers.

- `inline` function
  - less overhead and is generally faster than a function call.
  - Best candidates are small functions that are called frequently from a few places, or functions called with one or more compile-time constant parameter.

Compiler often inlines some functions automatically. The `inline` keyword can be used as a hint to the compiler, but does not guarantee that the function is inlined.

Do inline functions improve performance? Yes and no.

Prefer inline function over macros.

That's all

Questions?