ENEL464 - Optimisation

1 Introduction

When writing programs, we use compilers to convert source code into executables. This process converts each line of code in an equivalent statement in assembly. The basic compilation process will generate a (usually) correct program, however, it will contain lots of superfluous instructions that aren't really needed to get the correct output. The clever people who develop our C compilers include an extra step to the compilation process called *optmisation*. This can be enabled and used with different levels to try to simplify the output machine code such that it runs faster (sometimes in the order of 5–10× faster!).

2 GCC

For GCC specifically, optimisations are controlled using the -O flag:

```
gcc -O3 -o poisson poisson.c
```

There are several options:

-O0	optimisations disabled (default)
-O1	compiler tries to reduce code size and execution time, without performing optimisations that take a greate deal of compilation time.
-O2	perform nearly all supported optimisations that do not involve a space-speed tradeoff.
-O3	enable more optimisations (but maintain standards compliance).
-Ofast	disregard strict standards compliance in addition to -O3 (i.e. fast math).
-Os	-O2 except for optimisations that increase code size.
-Og	optmise for debugging experience. Equivalent to -O1 except for optimisations that make it harder to debug.

3 Compiler Explorer

There is a fantastic tool called *Compiler Explorer* (`https://godbolt.org/`) which lets you compile code from a variety of languages (including C/C++) and can show the generated assembly code for platforms like x86-64, ARM, and so on.

Try compiling the following code with GCC for x86-64:

```
#include <cstdio>

inline int foo(int bar) {
    int sum = 1;
    for (int i = 2; i < bar; i++) {
        sum *= i;
    }

    return sum;
}


void bar() {
    printf("%i", foo(5));
}
```

In this example, the *foo* function simply calculates the product of all the numbers up to the input (i.e. $1 \times 2 \times 3 \times \ldots$). The generated code looks like:

```
foo(int):
        push    rbp
        mov     rbp, rsp
        mov     DWORD PTR [rbp-20], edi
        mov     DWORD PTR [rbp-4], 1
        mov     DWORD PTR [rbp-8], 2
        jmp     .L2
.L3:
        mov     eax, DWORD PTR [rbp-4]
        imul    eax, DWORD PTR [rbp-8]
        mov     DWORD PTR [rbp-4], eax
        add     DWORD PTR [rbp-8], 1
.L2:
        mov     eax, DWORD PTR [rbp-8]
        cmp     eax, DWORD PTR [rbp-20]
        jl      .L3
        mov     eax, DWORD PTR [rbp-4]
        pop     rbp
        ret
.LC0:
        .string "%i"
bar():
        push    rbp
        mov     rbp, rsp
        mov     edi, 5
        call    foo(int)
        mov     esi, eax
        mov     edi, OFFSET FLAT:.LC0
        mov     eax, 0
        call    printf
        nop
        pop     rbp
        ret
```

Quite a bit going on there for a small function!

If we enable *-O1* (add it to the compiler flags), we can see the compiler detects that the call to *foo* is a compile time constant. This lets the compiler do the math in advance and simply store the result in our program:

```
.LC0:
        .string "%i"
bar():
        sub     rsp, 8
        mov     esi, 24
        mov     edi, OFFSET FLAT:.LC0
        mov     eax, 0
        call    printf
        add     rsp, 8
        ret
```

If we then enable *-O3*, we can see the compiler strips out a few extraneous instructions that aren't needed for this program giving us a very small (and fast) program to run:

```
.LC0:
    .string "%i"
bar():
    mov     esi, 24
    mov     edi, OFFSET FLAT:.LC0
    xor     eax, eax
    jmp     printf
```

4 Next steps

You should compare the runtime of your solution in the different optimsation modes. Does *-Ofast* make a difference compared to *-O3*? Which optimisation modes make no change to performance and why?