O is the letter “O”

As you already know, the compiler is the best optimisation tool you have. The main optimization flag is -O or (equivalently) -O1. With this setting, the compiler tries to reduce code size and execution time without performing any optimizations that take a great deal of compilation time.

With -O2, every optimization is turned on that does not result in a bigger executable. This option exists because the size of the executable affects the efficiency of the instruction cache.

Option -O3 turns on additional optimization options that may increase the executable size or can take a very long time to compile.

Option -Os is like -O2 but includes additional options that can shrink the executable size.

The -Ofast option is like -O3, but disregards strict IEEE standards for floating point math (it turns on -ffast-math).

(On some systems, -O4 performs optimization at link time, enabling optimization across compilation units.)

By specifying the CPU architecture with the -march option, you let GCC use a wider, architecture-dependent instruction set. You can compile to any one of a number of target architectures, but you usually want to use -march=native

The -mtune option tells GCC to just tune the compilation to suit a particular machine, while still using a smaller and more generic instruction set.

-march implies the same tuning steps as -mtune

Important: if you specify -march, the binary output will be less portable to other machines. If you want to use an executable or library on another machine, use -mtune instead

FOR THE LAB EXERCISE:
Throughout this lab and in future labs, compare the effects of your optimizations with and without compiler optimizations. Unless the lab instructions suggest otherwise, try at least the -O2 and -O3 compiler optimization flags

The strategies covered in this lab are grouped as follows:

- (1) Loop optimization
- (2) Faster boolean evaluation
- (3) Avoiding denormalized floating-point values
- (4) Strength reduction – use cheap operations
- (5) Function inlining

Reminder: as noted above, in this lab and in future labs, compare the effects of your optimizations with and without compiler optimizations.

Task-0:

Expressions from time command through terminal. Ex: time ./regularcode.

- The first one, real time, is the actual real time (also called wall clock time) taken. The one in the middle, user time, is the CPU time spent executing user code. Normally for a single-threaded program mostly doing computations the user time should be close to the real time. The third value, sys time, is the CPU time spent doing system calls. This should in most cases be small, but if your code is for example doing a lot of small malloc() and free() calls the sys time may become significant

Different timing for each program:
regularcode: 1.0000499600361 ./regularcode 1.74s user 0.03s system 86% cpu 2.027 total

mallocycode: 1.0000059952043 ./mallocycode 0.92s user 0.18s system 79% cpu 1.377 total

sleepycode: 1.0000059952043 ./sleepycode 1.85s user 0.02s system 26% cpu 7.117 total

threadedcode:

- main thread result : 1.0000499600361
- other thread result: 1.0000499600361
- ./threadedcode 3.54s user 0.03s system 198% cpu 1.792 total (using -lpthread to build it)

The code regularcode is simply a serial (single-threaded) program doing some computation, so for that we expect the user time should be close to the real time.

The code sleepycode also calls sleep() a few times

mallocycode performs lots of dynamic memory allocations.

threadedcode performs some computation using two threads.

Using -O2 compiler optimization flags:

- regularcode:
  - 1.0000499600361: ./regularcode 0.64s user 0.01s system 65% cpu 0.996 total
- mallocycode
  - 1.0000059952043: ./mallocycode 0.84s user 0.17s system 84% cpu 1.202 total
- sleepycode
  - 1.0000499600361: ./sleepycode 0.76s user 0.02s system 12% cpu 5.971 total

- threadedcode
    - main thread result : 1.0000499600361
    - other thread result: 1.0000499600361
    - ./threadedcode 1.32s user 0.01s system 156% cpu 0.850 total

Using -O3 compiler optimization flags:

- regularcode:
    - 1.0000499600361: ./regularcode 0.65s user 0.01s system 80% cpu 0.820 total
- mallocycode
    - 1.0000059952043: ./mallocycode 0.84s user 0.14s system 85% cpu 1.147 total
    -
- sleepycode
    - 1.0000499600361: ./sleepycode 0.74s user 0.01s system 12% cpu 5.939 total
- threadedcode
    - main thread result : 1.0000499600361
    - other thread result: 1.0000499600361
    - ./threadedcode 1.32s user 0.01s system 158% cpu 0.840 total

Do they behave differently? Can you understand why?

- We can tell that the real time, the actual real time is mostly the same for each one of the programs.
- The user time which is the time spent on executing user code was roughly the same for regularcode and sleepycode.
- sys time, is the CPU time spent doing system calls which was mostly the same for sleepycode.c, regularcode.c and threadedcode.c, but for mallocycode there was a big difference. This could be due to the program having to do a lot of small dynamic memory allocations.

Other tasks:

For the rest of this lab, focus mainly on the middle timing, user time, which is often the most informative. System “noise” e.g. disturbances due to other processes running on the same computer can greatly affect measurements — it is a good idea to run at least five times and record the lowest time.

Task-1:

In general, loops are optimized by doing as little as possible in the body of a loop.

A “loop invariant” is something that appears in a loop but does not change with the loop variable;

Read the codes and form an expectation of the performance difference with Fast and Slow:

- s

Without -O2 and O3 on invariant_loop():
Slow:

- Done. sum = 649785664285041.375
- ./loop_invariants 0.67s user 0.11s system 80% cpu 0.974 total

Fast:

- Done. sum = 649785664285041.375
- ./loop_invariants 0.63s user 0.12s system 81% cpu 0.927 total

With -O2:
Slow:

- Done. sum = 649785664285041.375
- ./loop_invariants 0.15s user 0.12s system 59% cpu 0.456 total

Fast:

- Done. sum = 649785664285041.375
- ./loop_invariants 0.15s user 0.11s system 60% cpu 0.431 total

With -O3:
Slow:

- Done. sum = 649785664285041.375
- ./loop_invariants 0.14s user 0.11s system 59% cpu 0.428 total

Fast:

- Done. sum = 649785664285041.375
- ./loop_invariants 0.14s user 0.08s system 57% cpu 0.387 total

Does the “fast” version of the code perform better than the “slow” version when compiler optimization is used?

- Answer: Yes