HIGH PERFORMANCE PROGRAMMING
UPPSALA UNIVERSITY
SPRING 2022
LAB 5: SERIAL OPTIMIZATION PART 1, REDUCING INSTRUCTIONS

The aim of this lab is to demonstrate and explore ways of reducing the amount of work in a program. Many of the techniques here can be done automatically when the compiler optimizes code, but sometimes it helps to do things by hand. However, hand-optimizing can also confuse the compiler and result in *worse* performance: never optimize without a good reason and always check performance and correctness afterwards! In fact, you may find that some of the exercises in this lab produce results that run against your intuition and expectation.

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.)

This is only the beginning, however, because GCC will by default assume very little about the processor architecture and even the instruction set. 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`, which automatically selects the architecture to match your system. 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`.

Date: February 7, 2022.

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.

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.

Of course, our goal should normally be to achieve the best performance we can using both compiler optimizations and manual code changes, so when making a change in our code the most important question is if that change improves performance when we have compiler optimizations turned on. However, to better understand what is going on and what the compiler is doing, it can be helpful to compare results with and without compiler optimizations.

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

The lab also includes parts about timing measurements and about the standard storage format for floating-point numbers.

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

1. Main Tasks

Start by downloading and unpacking the `Lab05_Instructions.tar.gz` file which contains the files needed for the tasks in this lab.

Task 0

The code for this task is in the `Task-0` directory.

In the instructions for this task we assume that you have compiled each program in such a way that the executable is named like the C source file without the `.c` extension, e.g.:

```
gcc -o regularcode regularcode.c
```

but of course you can choose to call your executable files whatever you want.

An easy way of measuring time is to use the `time` command. For example, after you compile the executable `regularcode`, run “`time ./regularcode`”. After the program completes, `time` will present three timing measurements. 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.

The `Task-0` directory contains four different codes with different behaviors. Look at each code to see what it is doing. Then compile and run each of them, using the `time` command to measure timings. 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, and `threadedcode` performs some computation using two threads. Check how the different timings given by the `time` command behave for these different cases. Do they behave differently? Can you understand why?

Note: `threadedcode` uses pthreads, so you will need to link with `-lpthread` (the pthreads library) in order to build that code. You will learn more about threaded programs later in the course.

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

The code for this task is in the `Task-1` directory.

In general, loops are optimized by doing as little as possible in the body of a loop. This seems obvious, but sometimes requires more thought.

A "loop invariant" is something that appears in a loop but does not change with the loop variable; for example, the `i*N` expression in the "slow" part of the code in `loop_invariants.c` can be seen as a loop invariant with respect to the loop over `j`, since its value does not depend on `j`.

The work function in `loop_invariants.c` exists in two versions. You can toggle between the "fast" version and the "slow" version by redefining the `FAST` preprocessor macro on line 4, setting `FAST` to 1 or 0 to enable the "fast" or the "slow" version, respectively. Read the codes and form an expectation of the performance difference.

Measure and compare the speed of the two versions. Does the actual speed difference match your expectation?

Try without compiler optimizations, then with compiler optimizations. Does the "fast" version of the code perform better than the "slow" version when compiler optimization is used?

Extra part: Compile to assembly (use -S and -fverbose-asm flags), examine the .s-files, and explain what the compiler optimization does.

In loops where the loop iterator and comparison operations are significant compared to the work inside the loop, careful formulation of the loop construct itself can improve performance.

The program in `string_loop.c` is written twice. You can toggle between the "fast" version and the "slow" version by redefining the `FAST` macro on line 4. Read the codes and form an expectation of the performance difference. Measure and compare the speed of the two versions. Does the actual speed difference match your expectation?

The work function in `array_loop.c` is exists in two versions. You can toggle between the "fast" version and the "slow" version by redefining the `FAST` macro on line 4. Read the codes and form an expectation of the performance difference. Measure and compare the speed of the two versions. Does the actual speed difference match your expectation?

Task 2

The code for this task is in the `Task-2` directory.

In the previous task, we looked at improvements in the loop construction itself. Here, we'll improve the comparison operation. This is applicable to if-statements inside the body of a loop as well as loop comparisons.

The most basic optimization of a comparison is called the *boolean short circuit.* In C (and Java, among other languages), the evaluation of a boolean expression is terminated as soon as the outcome is known.

- `A && B`: If A is false, then the statement is false regardless of B.
- `A || B`: If A is true, then the statement is true regardless of B.

In both cases, B is not evaluated at all. This is useful for instance if B is an expensive function, but also when it's necessary to avoid causing the side-effects of evaluating B.

For example:

```
if(p != NULL && *p > 3.9) {
  /* Do something.. */
}
```

ensures that `p` is not dereferenced when it points to NULL and would cause a "segmentation fault" error.

In `short-circuit.c`, you'll find a code that repeatedly generates some random boolean values and sets a variable according to certain rules. Improve the formulation of the if-statements so they are more efficient.

Sometimes, program correctness demands that you perform bounds checking to ensure proper access to the array. Proper array bounds are usually between 0 and a positive, reasonably sized integer. The program in `bounds.c` contains a loop over an array and a naively formulated bounds check. Use casting to unsigned integers to remove the logical OR operation from the check. See Section 14.2 in the Fog book *Optimizing software in C++*. You can toggle between the "fast" version and the "slow" version by redefining the `FAST` macro on line 4. Measure and compare the speed of the two versions. Does the actual speed difference match your expectation?

Task 3

Floating-point numbers

Floating-point numbers are stored in a computer in the following way:

$$(-1)^s \times M \times 2^E$$

where s is a sign bit, M is the mantissa or significand, and E is the exponent. The number of bits in each part are shown in Table 1.1.

Type	Sign	Exponent	Mantissa	Decimal digits
Single precision (`float`)	1	8	23	7
Double precision (`double`)	1	11	52	16

TABLE 1.1. Binary representation of floating-point numbers. Note that the number of decimal digits shown is approximate, it is really the number of bits in the mantissa/significand that determines the precision.

The mantissa is stored using that number of bits (23 or 52), so there are $2^{23} = 8388608$ and $2^{52} = 4503599627370496$ different possible values of the mantissa for single and double precision, respectively. The mantissa bits are interpreted as a fraction so that the factor M in the formula above becomes a number between 1 and 2 (except for denormalized numbers, see below).

The value of the exponent is the value of the exponent bits as an unsigned integer *minus* a fixed bias. The bias for single precision is 127, for double precision 1023.

The range of the exponent is $[-126, 127]$ for single precision and $[-1022, 1023]$ for double precision. This determines the normal range of numbers that can be stored.

Note that the range $[-126, 127]$ for single precision is slightly smaller than the possible range for a 8-bit integer. This is because a few values are reserved for special purposes, like encoding the special numbers "infinity" and "not-a-number" (NaN). The same goes for double precision, where the range $[-1022, 1023]$ is smaller than the possible range for a 11-bit integer.

When a computation results in a number requiring a larger exponent than the maximum, the result is converted to + or - infinity. If the result is undefined for other reasons, it is represented as "not-a-number" (NaN).

The largest number that can be stored in the normal way, without being converted to "infinity", corresponds to the case when the exponent has its maximum value and the mantissa is also as large as it can be, corresponding to a factor M of nearly 2. Therefore, the largest numbers that can be stored in the normal way are slightly smaller than 2^{128} and 2^{1024} for single and double precision, respectively.

Write a small program that tests what happens when a number becomes too small or too large: initialize a `float` or `double` variable to some value and then use a loop where you increase the number by e.g. a factor of 100 each time, and use printf to print the number each time. At what point does it become "inf"? Does this match your expectation based on the storage format described above? Also test producing a "not-a-number" (NaN) result by e.g. calling `sqrt()` for a negative number, and printing the result. What happens if you continue to do operations on a "inf" or "nan" number, e.g. if you add something to it, what is the result?

One way of checking the relative precision of floating-point numbers is to try adding a very small number to 1 and check if the result is greater than 1. Write a small program doing such a test: keep a small number in a variable e.g. `epsilon` and in a loop make `epsilon` smaller and smaller, for example by multipluing it by 0.5 each loop iteration. In each loop iteration, do a little test where you compute `epsilon + 1` and store that result in another variable, and then check if it is greater than 1. For very small numbers you should find that the result becomes exactly 1; the difference is so small that it cannot be represented with the limited number of bits used. For how small numbers do you get a result different from 1? Does this match the "Decimal digits" info given in Table 1.1?

Denormalized (subnormal) floating-point numbers

For very small numbers, all of the exponent bits are 0. When this is the case, the number is called a *subnormal* or *denormalized* number. Full precision is achieved when all the bits in the mantissa store significant information. For numbers smaller than the largest subnormal number, the leading bits of the mantissa will be zero. This causes loss of precision.

There is also a performance consequence. Calculations on subnormal numbers can be significantly slower than ordinary calculations.

In the two codes found in the `Task-3` directory, a single-precision number is created and used for calculations. Use timing to determine the performance penalty due to calculations with denormalized numbers.

The performance of computations on denormalized numbers can be different for different CPU models; if you do not see a significant difference on your computer, try on the lab computers or by logging in using ssh to e.g. trygger.it.uu.se and run the test there.

Certain compiler optimization options cause subnormal numbers to be flushed to zero, thereby avoiding the performance problem but with a risk of getting wrong results in case calculations on such very small numbers are important for the final result. Compare the compiler optimization flags `-O2`, `-O3`, and `-Ofast` to see how they affect calculations with subnormal numbers. Do they improve performance? What happens with the result?

The only difference between the codes `norms.c` and `denorms.c` is the starting value of the `tiny` variable on line 7. Try making that value even smaller, try e.g. `1e-50`. What happens with the printed values when it gets small enough? What happens with the performance? Can you understand why this happens?

Task 4

The code for this task is in the `Task-4` directory.

In this task, we're going to cover several methods of reducing the computational load of some commonly seen tasks.

Copying arrays or initializing array values to zero is a common practice. The `string.h` library provides functions that perform these operations in one go. The program in `memset.c` shows how `memset` and `memmove` can be used. Read the code and form an expectation of the performance difference. Measure and compare the speed of the two versions. Does the actual speed difference match your expectation?

If a function is going to be called with only a limited set of inputs, consider implementing the function as a lookup table. The program in `lookup.c` shows how this can be done. Read the code and form an expectation of the performance difference. Measure and compare the speed of the two versions. Does the actual speed difference match your expectation? What sort of functions are candidates for this kind of optimization?

Note that although in the lookup table example in `lookup.c` the table was small and could be statically allocated and filled with hard-coded values, it is also possible to use larger tables that are allocated and initialized when the program runs. If the same values are computed many times during program execution, saving precomputed values in a table is often a good idea.

The computational cost of arithmetic operations varies a lot. Choosing the cheapest arithmetic formulation can improve performance significantly. This technique is known as *strength reduction*.

The program in `strength_reduction.c` contains a loop with a number of arithmetic operations. Reformulate them to improve the speed of the code (see list of hints below). Remember to check that the computed values are still the same.

The program in `math_functions.c` contains a loop calling a math function. See comment in the code for examples of other ways of getting the same result. Try out the different approaches. Which way is fastest? Can you understand why?

Here are some hints:

- The cheapest operations are integer + and -, and bit-level operations like `>>`, `&`, and `&&` .
- The bitwise shift operators `>>` and `<<` are equivalent to integer division or multiplication by a power of 2, respectively. Note, it should only be used for positive integers as all bits are shifted, the sign bit included.

- Integer division by a constant is faster than with a variable.
- Integer division is faster if unsigned.
- Floating-point multiply is much faster than floating-point division.
- Arithmetic operations are faster than function calls. Don't use `pow()` if you can avoid it.
- Math functions exist in different variants for different precision, e.g. `sqrt()` and `sqrtf()`. A higher precision result is usually more expensive to compute.

Extra part: write a program that calculates the relative speed of arithmetic operations. Use timing functions from `time.h` to measure the time. See the example code `timings.c` for an example of how the timing functions can be used.

Task 5

Calling a function incurs a certain cost. Execution flow must jump to a different address in the code, and return. This jump alone can take up to 4 cycles and can reduce the efficiency of the instruction cache. A new stack frame is set up, parameters are stored (on the stack in 32-bit mode which takes even more time), and the registers from the previous frame must be saved and restored.

It is therefore a good idea to limit function calls in the critical part of the code. One way of doing that while maintaining code quality is with *function inlining* with the `inline` keyword.

When the compiler inlines a function, it replaces the function call with the function code, effectively removing the associated costs. Inlining is especially important for functions called from the innermost loop of a program, but can also be an effective optimization tool when used to turn a *frame function* into a *leaf function.* A frame function is a function that calls at least one other function. A leaf function is a function that doesn't call any other function. A leaf function is simpler than a frame function because the stack unwinding information can be left out if exceptions can be ruled out or if there is nothing to clean up in case of an exception. A frame function can be turned into a leaf function by inlining all the functions that it calls.

One downside of function inlining is that the compiler has to make a non-inlined copy of the inlined function, because of the possibility that another compilation unit (.c file) contains a call to the function. This is often dead code, which can impact instruction caching and executable size. Adding the `static` keyword to the function definition tells the compiler that the function can only be called from the same compilation unit. The linker option `-ffunction-sections` allows the linker to exclude unreferenced functions from the executable.

Note that the `inline` keyword does not *force* the compiler to inline the function. If the compiler decides it's a bad idea (e.g. because the function is called from too many places or is too big), then the function is not inlined. Conversely, if it determines that there is a benefit the compiler may inline functions that are not marked with the `inline` keyword.

Construct a program (or set of programs) that demonstrates the potential benefit of function inlining. Note that you have to compile with a sufficient optimization level for inlining to occur.