HIGH PERFORMANCE PROGRAMMING
UPPSALA UNIVERSITY
SPRING 2018
LAB 6: INSTRUCTION-LEVEL PARALLELISM (ILP)

In this lab we focus on optimizations related to instruction-level parallelism (ILP). ILP allows the computer to perform several instructions at the same time, which of course can be very good for performance. However, this only works if the program contains independent instructions; if each instruction depends on the result of the previous instruction, they cannot run in parallel. Therefore, optimizing a program to make use of ILP typically means rewriting the code to get more independent instructions.

Background reading:

ILP works largely because of a feature in the CPU called *pipelining*. In a pipelined CPU, the execution of an instruction is separated into several stages. Here is one possible set of such stages:

(1) Instruction Fetch (IF) — Instruction code is retrieved from program

(2) Instruction Decode (ID) — Instruction is decoded

(3) Execute (EX) — Operations are executed

(4) Memory access (MEM) — Memory is read or written

(5) Register write back (WB) — Result is written to the output register

For our purposes here, it is not important to know exactly what the stages are and what they are doing (and the list above is anyway just an example). The important thing for us is to understand the idea of a pipeline, that the work needed to carry out each instruction is usually split into a set of stages.

The benefit of pipelining is that a different instruction can be present in each stage of the pipeline. With five stages, up to five instructions can be executed simultaneously. By the time the first instruction has finished, the second instruction is already at the WB stage, and a sixth instruction is already being fetched. The downside is that the minimal time to execute a single instruction is five clock cycles (one cycle per stage).

In a real processor nowadays, pipeline stages typically do not correspond exactly to the classical five stages above. At the height of the pipelining craze, the push for ever faster processor clocks had Intel designing CPU's with up to 31 stages! Today, pipelines are usually far shorter, with around 5-10 stages.

Date: February 1, 2018.

One reason for short modern pipelines is the problem of *hazards*, instructions that depend on each other. If one instruction uses the output of another, then they cannot follow directly after one another in the pipeline.

Example: say that our program has an "add" operation directly followed by a "move" operation. The add operation adds some numbers together and then, when it is done, places the result in a register R. The move operation is supposed to take the value in R and "move" (copy) it to a given memory address. This will not work properly if those two instructions were to be fed into the pipeline directly after eachother; the move operation would then try to get the value in the register R before the add operation had placed its result there. To make it work, the CPU would need to wait and not insert the move operation into the pipeline until it was certain the the result of the add was available.

Something that can be even more problematic for pipelining is *conditional branches*. A branch is a point in the program where execution is to be continued somewhere else, and a conditional branch is when the branch should be done or not depending on some condition. Thus, and if-statement in a C/C++ program will generally correspond to a "conditional branch" in the resulting machine code. When we have a loop that continues or stops depending on some condition, that will also give a conditional branch.

A conditional branch is very problematic for pipelining because the processor does not even know which instruction to fetch until the branch instruction is complete. Also in this case, it becomes impossible to keep the pipeline full. Some "empty space" or "no-operation" instructions must then be inserted into the pipeline instead of useful instructions. This is called a *bubble* in the pipeline. The longer the pipeline, the worse these problem become, and very long pipelines are essentially never filled by ordinary programs.

Compilers can try to generate machine code that minimizes these problems, and there are a few programming techniques that can help.

The following ILP-related issues are covered in this lab:

(1) Loop unrolling

(2) Loop fusion

(3) Avoiding branches

(4) Branch prediction

The lab also includes a part about how assembly code output can be used to help understand what the compiler is doing.

1. Main Tasks

Start by unpacking the `Lab06_ILP.tar.gz` file which contains the files needed for the tasks in this lab.

The code for each task is available in a separate directory for each task, including a makefile that you can use to compile. If you want to try different compiler optimization flags, you can do that by changing CFLAGS in the makefiles.

Task 1

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

This task is about an optimization technique called *loop unrolling*. The idea of loop unrolling is to rewrite a loop so that the loop iterations are done in groups, like this:

```
for(i = 0; i < N; i++) {
  // do something related to i
}
-->
for(i = 0; i < N; i+=4) {
  // do something related to i
  // do something related to i+1
  // do something related to i+2
  // do something related to i+3
}
```

The idea is that the code can be faster this way because the instructions for i, i+1, i+2, i+3 can be done without interruptions for checking the loop condition.

Of course, it does not have to be 4 loop iterations that are grouped, it can be any number. This number is sometimes called the *unroll factor*. Note that if our code is written as above, we are assuming that N is divisible by the unroll factor. If N is not divisible by the unroll factor we can of course still use loop unrolling but the remaining iterations need to be handled separately, for example after the unrolled loop.

Loop unrolling is a kind of optimization that can often be done by the compiler, but there are also situations where you can benefit from doing it yourself.

Now, to try how loop unrolling works, start by looking at the code in the `Task-1` directory. The file `testfuncs.c` contains two routines, `f_std` and `f_opt`, which to begin with are identical. Your task is to change `f_opt` using loop unrolling, to see if you can make it faster in that way. Try your implementation and compare the run time to the unchanged `f_std` routine. Does loop unrolling make it faster? What seems to be a good choice of the unroll factor?

Remember to check correctness of the results! Make sure that you verify correctness also for cases when the number of loop iterations (`N1` in `main.c`) is not divisible by the unroll factor. For example, when N1 is 200, try the unroll factor 3 — make sure you get correct results also in that case.

To see if the compiler can do a better job, also try changing the compiler flags in the makefile. To ask the compiler to do loop unrolling, add the option `-funroll-loops`.

You can also try changing from `-O2` to `-O3`. Do those changes make the `f_std` routine faster than your own loop-unrolled code?

Using "`man gcc`" you can read about the precise meaning of the compiler option `-funroll-loops` and compare it to the `-funroll-all-loops` option. Note that according to the man page, `-funroll-all-loops` often makes programs run more slowly.

Task 2

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

In this task, we look at an example where it is difficult for the compiler to do loop unrolling, so it can be worthwhile do it manually.

Look at the code in `testfuncs.c`. As in the previous task, `testfuncs.c` contains two routines, `f_std` and `f_opt`, which to begin with are identical. This time, the loop is complicated by the fact that there is a variable called "counter" and an if-statement checking its value, so that in some iterations the value `x` changes.

Optimize the `f_opt` code using loop unrolling. *Hint: since the "counter" is reset every 4 iterations, it is probably beneficial to unroll the loop by 4. Can you even get rid of the if-statement?*

Try changing the compiler flags as in the previous task, to ask the compiler to do loop unrolling automatically. Does this make the code faster? If not, can you understand why it may be more difficult for the compiler to optimize the code in this case?

Task 3

The code for this task is in the `Task-3` directory.

In this task you will try a technique called *loop fusion*. Loop fusion essentially works like this:

```
for(i = 0; i < N; i++) {
  // do "work A" related to i
}
for(i = 0; i < N; i++) {
  // do "work B" related to i
}
-->
for(i = 0; i < N; i++) {
  // do "work A" related to i
  // do "work B" related to i
}
```

As with loop unrolling, this gives a code where more instructions can be carried out without checking the loop condition in between.

Look at the code in `stencil.c`. There are three versions of the `apply_stencil` routine, labeled v1, v2, v3. To start with, try optimizing `apply_stencil_v2` using loop fusion. Does this make `apply_stencil_v2` faster than `apply_stencil_v1`?

Try changing the `STENCIL_SZ` value in `stencil.h` from 20 to 4. How does this affect the timings of `apply_stencil_v2` compared to `apply_stencil_v1`? Can you understand why changing `STENCIL_SZ` has this effect? *Hint: the compiler may do some optimizations automatically for very short loops, but not for longer loops.*

Extra part: The third version of the `apply_stencil` routine, `apply_stencil_v3`, is a blocked implementation. To get a scenario where blocking is more likely to help, you can change the `GridPt` structure in `stencil.h` by including the previously commented-out "`double v[5]`" line. This means that the data structure becomes larger (the `v[5]` part is not used, it just takes up space), so there are more problems with cache, so that the blocking implementation may be faster than the others. Now try improving `apply_stencil_v3` also using loop fusion, and see if the blocking makes it faster than `apply_stencil_v2`.

Task 4

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

In this task, we look at an example of how important it can be to avoid if-statements (branches) inside inner loops.

The file `testfuncs.c` contains two routines, `f_std` and `f_opt`, which to begin with are identical. Your task is to optimize `f_opt` by changing it so that fewer if-statements are used.

Hint: the if-statements are related to the "counter" variable, but we can see that the value of "counter" does not matter as soon as it has become larger than 3. Then the computed sum is not affected by changes in "counter". Can you exploit this, to make most of the computations in a loop without any if-statements, and still get the correct result?

How much faster can you make the code by trying to avoid if-statements?

Task 5

The code for this task is in the `Task-5` directory.

This task is about looking at the effects of *branch prediction*.

Some computers have advanced branch prediction capabilities. This means that the problems of having branches (if-statements) in the code can be alleviated by the hardware guessing which way a branch is going, so the execution of instructions can continue without interruption even though the branch condition has not been checked yet. Then, when the branch condition check is ready, the hardware may need to roll back some computed results in case the branch went the other way.

Branch prediction can make a program run fast in spite of the existence of branches, especially if the branches are very predictable. For example, if a particular branch condition is true 99.9% of the time, it will almost always be beneficial to continue execution as if it is true; it will only very rarely happen that the results need to be discarded because the condition was false.

To see how branch prediction works on the computer you are using, first run the code in the `Task-5` directory unmodified, and note the times. Then look at the following two if-statements in the `f_std` routine:

```
if(counter > 5 && a[i] > 0.5)
  counter = 5;
if(counter > 6 && a[i] > 0.6)
  counter = 6;
```

There, the value in `a[i]` is compared to 0.5 and 0.6, respectively. Because of the way the `a[i]` values are generated in `main.c`, they will be randomly distributed between 0 and 1, so comparisons with 0.5 and 0.6 will sometimes be true, sometimes false, quite unpredictable.

The `a[i]` values are generated by the following code lines in `main.c`:

```
for(i = 0; i < N1; i++)
  a[i] = (rand() % 1000) * 0.001;
```

To make branch prediction work better, you can try running the code with different input, changing in `main.c` so that the `a[i]` values are instead distributed between for example 0 and 0.5. Then, comparing with 0.5 or 0.6 will always give the same result, so if the hardware is doing branch prediction efficiently, the program should run faster. Does it? How much does the time change when you modify the `a[i]` values like that?

Note that branch prediction capabilities in the CPU will never completely remove the performance problems dut to branches; when possible, it is better to get rid of the branches completely. To see this, you can compare the performance you got in this task with the previous task (the code in the two tasks is the same).

Task 6

The code for this task is in the `Task-6` directory.

Here we will look at how to get assembly code output from the compiler, and how to recognize branching instructions in the assembly code output.

Us the `-S` flag to request assembly code output from the compiler, e.g. like this:

```
gcc -S -c myfunctions.c
```

The compiler will then generate a text file called `myfunctions.s` containing the assembly code. Then, by looking at the myfunctions.s file we can see exactly what the compiler has done; how our C source code has been compiled to machine instructions.

Open the generated `myfunctions.s` file and study the assembler code, and compare it to the C source code in `myfunctions.c`. Look for the function name "`ffff`" and a line saying "`ret`" (return). Between those lines in the file we find the implementation of the function.

It can be tricky to understand the assembly code, but already some very basic knowledge about it can provide us with useful information. One thing that is good to know is how to identify branches in the code. A conditional branch typically consists of a comparison (cmp) instruction followed by a "jump" instruction starting with `j` (`je`, `jne`, `jg`, `jge` etc).

For more information about branching instructions in assembly language, see for example this page:
https://www.tutorialspoint.com/assembly_programming/assembly_conditions.htm

In case of your "`ffff`" function in `myfunctions.c` that function includes a loop, so we should expect to see a conditional branch corresponding to the loop. Can you identify the loop in the assembly code?

Also try commenting out the loop and looking at the assembly code again — now there should not be any conditional branch in the function. You can also try adding another loop, either after the first loop or as a nested loop inside the first loop — are you able to see this in the assembly code?

Being able to identify loops in the assembly code allows us to check what the compiler is doing, for example if a loop has been unrolled or not.

It is also useful to be able to identify function calls. In this test code, the main function is calling the "`ffff`" function. Look at the assembly code for the main function — can you find the function call there? What does it look like?

You can add the `-fverbose-asm` flag to make the code more readable.

Extra part: look at the assembly code for some of the previous tasks, for example where the code was optimized by removing if-statements — can you see that there is a smaller number of comparisons and branches in the code with fewer if-statements?