Task-1

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

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

- (1) Loop unrolling
- (2) Loop fusion
- (3) Avoiding branches
- (4) Branch prediction
- (5) Auto vectorization
- (6) Assembler code (optional)

The idea of loop unrolling is to rewrite a loop so that the loop iterations are done in groups

Task-2:

From what we can see with my optimzed code, I removed the if statement since it was not necessary to have, and added the 4-part loop unrolling:

- f_std tests took 1.069 wall seconds.
- f_opt tests took 0.209 wall seconds.
- Checking correctness: max_abs_diff = 0
- OK, result seems correct.
- ./unroll_test 2.51s user 0.03s system 88% cpu 2.855 total

Optimized is 5x faster in this example.

adding the flag -funroll-loops to the compiling did not really change a thing.

Task-3:

In this task you will try a technique called loop fusion

We get this, meaning we got a slower result!:

- Doing 400 iterations with N=232 using apply_stencil function version 1 took 0.180 wall seconds.
- Doing 400 iterations with N=232 using apply_stencil function version 2 took 0.455 wall seconds.
- Doing 400 iterations with N=232 using apply_stencil function version 3 took 0.180 wall seconds.
- Checking correctness by comparing 1 and 2: max_abs_diff = 1.443e-15
- Checking correctness by comparing 1 and 3: max_abs_diff = 0
- OK, results seem correct!
- ./stencil_test 1.53s user 0.03s system 82% cpu 1.897 total

Try changing the STENCIL_SZ value in stencil.h from 20 to 4.

- Doing 400 iterations with N=200 using apply_stencil function version 1 took 0.018 wall seconds.
- Doing 400 iterations with N=200 using apply_stencil function version 2 took 0.018 wall seconds.
- Doing 400 iterations with N=200 using apply_stencil function version 3 took 0.018 wall seconds.
- Checking correctness by comparing 1 and 2: max_abs_diff = 7.772e-16
- Checking correctness by comparing 1 and 3: max_abs_diff = 0
- OK, results seem correct!
- ./stencil_test 0.13s user 0.01s system 35% cpu 0.401 total

This decreased the time for all of the functions!

Changing STENCIL_SZ back to 20 and running with the loop fusion in V3 gave this results, indicating no much difference in speed:

- Doing 400 iterations with N=232 using apply_stencil function version 1 took 0.383 wall seconds.
- Doing 400 iterations with N=232 using apply_stencil function version 2 took 0.423 wall seconds.
- Doing 400 iterations with N=232 using apply_stencil function version 3 took 0.419 wall seconds.
- Checking correctness by comparing 1 and 2: max_abs_diff = 1.443e-15
- Checking correctness by comparing 1 and 3: max_abs_diff = 1.443e-15
- OK, results seem correct!
- ./stencil_test 2.48s user 0.02s system 91% cpu 2.718 total

Task-4:

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

After fixing the code, this is the results we get:

- f_std tests took 0.596 wall seconds.
- f_opt tests took 0.186 wall seconds.
- Checking correctness: abs_diff = 0
- OK, result seems correct.
- ./branch_test 1.52s user 0.02s system 78% cpu 1.958 total

We got a big difference in performance.

Task-5:

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.

From 0 and 1:

- f_std tests took 0.630 wall seconds.
- f_opt tests took 0.595 wall seconds.
- Checking correctness: abs_diff = 0
- OK, result seems correct.
- ./branch_test 2.28s user 0.02s system 87% cpu 2.619 total

Changing from values to 0 and 0.5:

- f_std tests took 0.568 wall seconds.
- f_opt tests took 0.598 wall seconds.
- Checking correctness: abs_diff = 0
- OK, result seems correct.
- ./branch_test 2.28s user 0.02s system 88% cpu 2.607 total

When changing the std function to the same from **Task-4:**

- f_std tests took 0.186 wall seconds.
- f_opt tests took 0.564 wall seconds.
- Checking correctness: abs_diff = 0
- OK, result seems correct.
- ./branch_test 1.51s user 0.01s system 89% cpu 1.705 total

Task-6: