Exercise 05

VU Performance-oriented Computing, Summer Semester 2024

Calvin Hoy

2024-04-24

(A) Basic Optimisation Levels

I consolidated all the necessary code (including necessary modifications) in the `test_cases/` directory.

The Bash script `bench_levels.sh` builds all these test cases using different `-O` levels, then utilises `benchmark.sh` which I created for exercise sheet 2 in order to measure the performance for each.

The following plots were created using `gnuplot` (see `plot_levels.sh`). The left axis shows execution time in seconds for `wall`, `user` and `system`, while the right axis shows memory use in kilobytes. All results were obtained on LCC3 over 3-10 runs, and raw data may be found in the `results_levels/` directory.

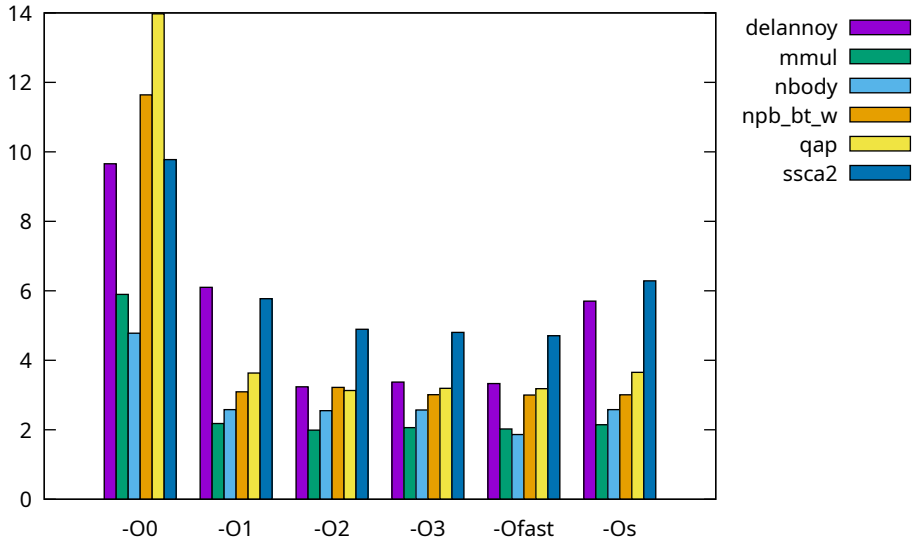

Figure 1: Wall time with different optimisation levels

We can see that for all test cases, `-O3` is faster than `-O2`, `-O2` faster than `-O1`, and `-O1` is much faster than `-O0`. `delannoy 13` profits more from `-O2` than most test cases, in particular compared with `ssca2 15`. `-Ofast` only produces minor gains, though it has a measurable impact on `nbody`. The `-Os` build meanwhile produces code roughly on par with `-O1` in performance, although it is somewhat slower for `ssca2 15`.

System time is close or equal to zero for all tested programs, and user time is basically the same as wall time, thus they are not shown here.

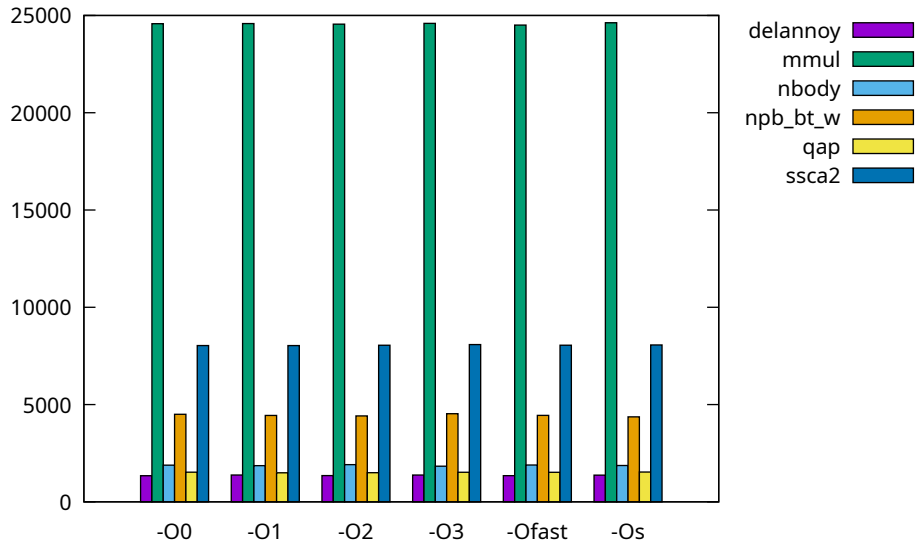

Figure 2: Memory use with different optimisation levels

Memory use is, for all intents and purposes, identical between all optimisation levels.

(B) Individual Compiler Optimisations

Using `difftastic` on the output of `gcc -Q --help=optimizers -O2` and `gcc -Q --help=optimizers -O3`, I found that the following flags are enabled/changed with `-O3` over `-O2`:

```
-fgcse-after-reload
-fipa-cp-clone
-floop-interchange
-floop-unroll-and-jam
-fpeel-loops
-fpredictive-commoning
-fsplit-loops
-fsplit-paths
-ftree-loop-distribution
-ftree-partial-pre
-funroll-completely-grow-size   # invalid for C
-funswitch-loops
-fvect-cost-model=dynamic       # -O2: =very-cheap
-fversion-loops-for-strides
```

To test the performance impact of each of these compile flags, I created the script `bench_o2o3.sh`. For each valid C flag, the script creates and dispatches a job script on LCC3, which then builds all test cases using the flag and examines the performance using `benchmark.sh`.

The following plots were also created using `gnuplot` (see `plot_o2o3.sh`). I only examined the wall time in seconds for this exercise. Again, all results were obtained on LCC3 over 3-10 runs, and raw data may be found in the `results_o2o3/` directory.

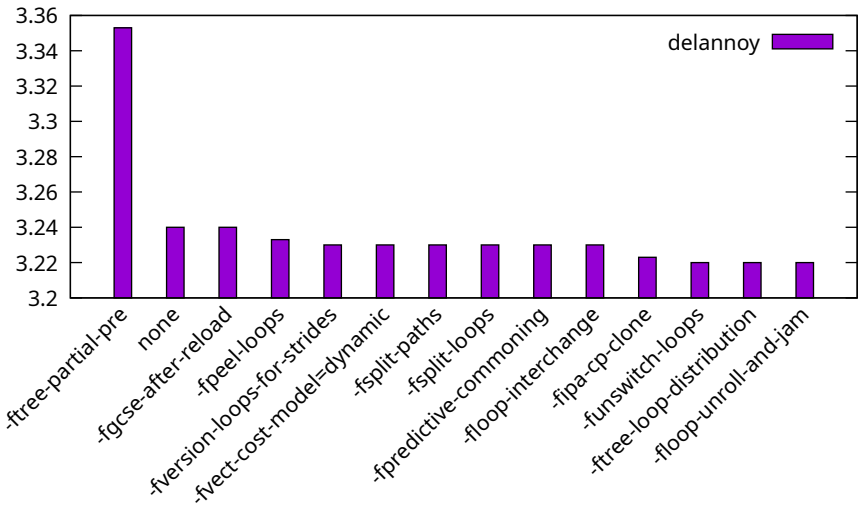

Figure 3: Wall time for `delannoy 13` using flags on top of `-O2`

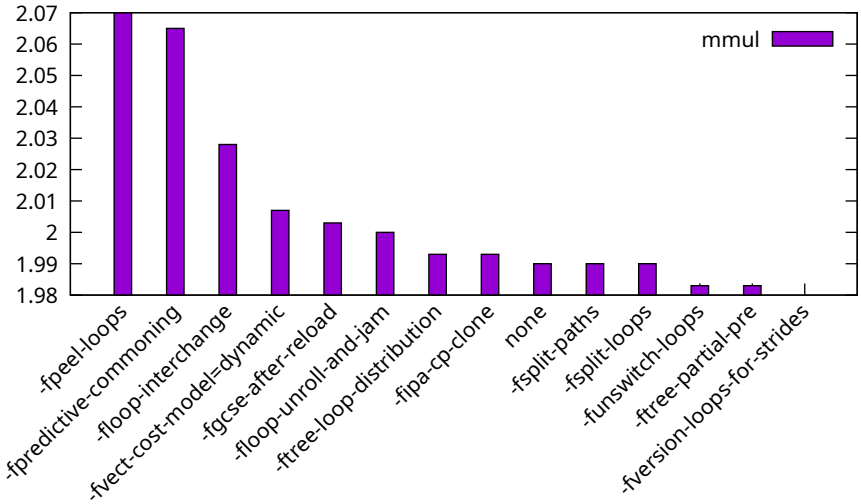

Figure 4: Wall time for `mmul` using flags on top of `-O2`

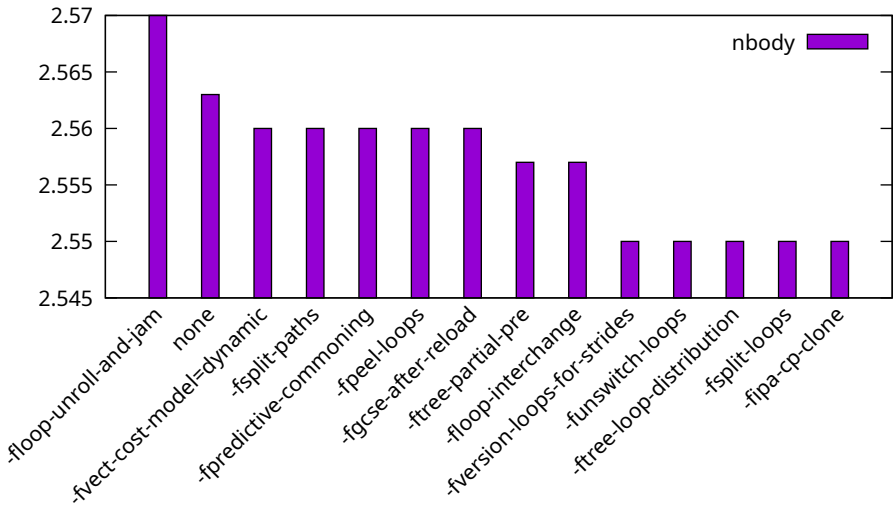

Figure 5: Wall time for `nbody` using flags on top of `-O2`

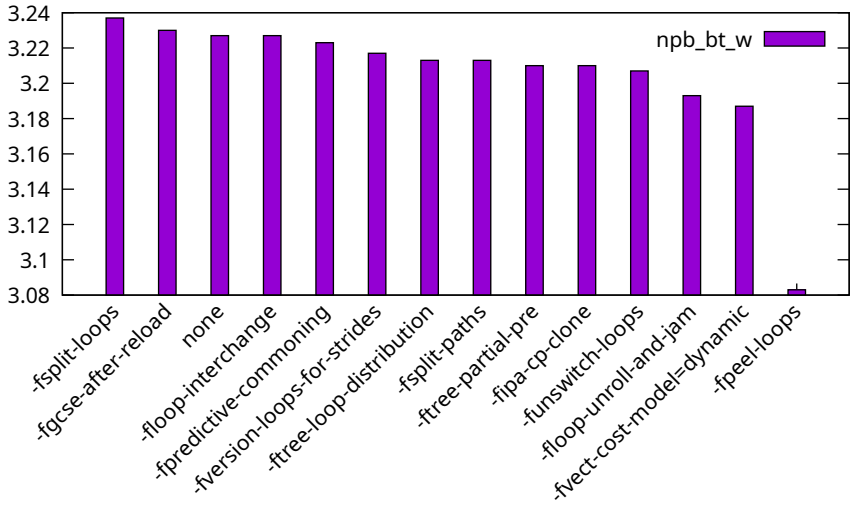

Figure 6: Wall time for `npb_bt_w` using flags on top of `-O2`

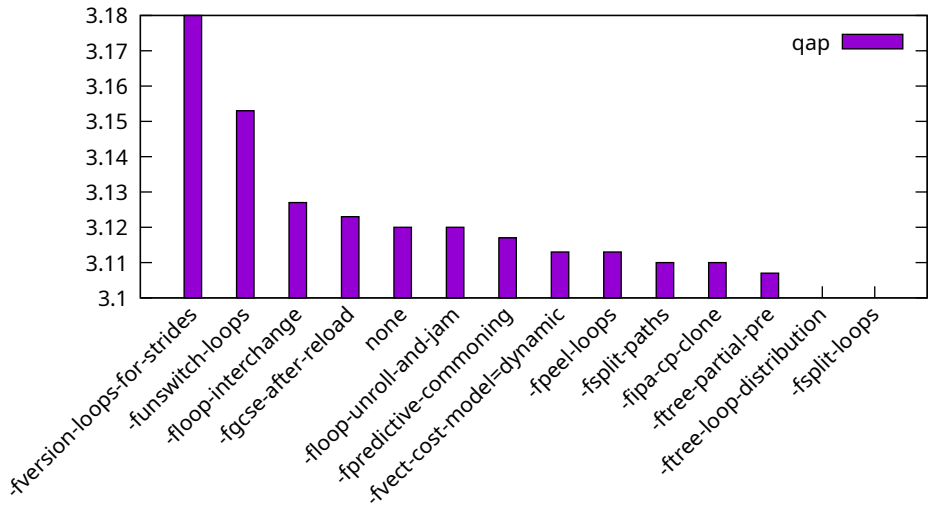

Figure 7: Wall time for `qap chr15c.dat` using flags on top of `-O2`

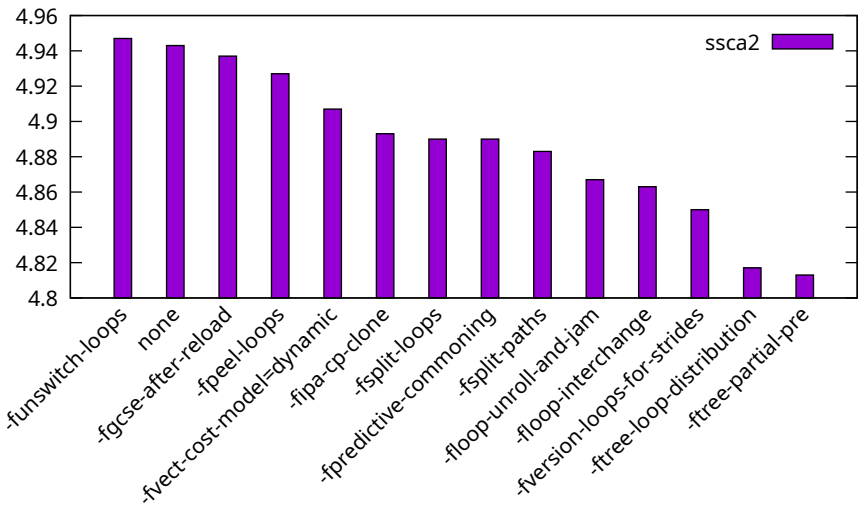

Figure 8: Wall time for `ssca2 15` using flags on top of `-O2`

Discussion

Going by median relative performance (see `o2o3_evaluate_flags.py`), the "best" flags are:

- `-ftree-loop-distribution` (0.994377)
- `-fipa-cp-clone` (0.994840)
- `-funswitch-loops` (0.995705)

The "worst" flags meanwhile are:

- `-fgcse-after-reload` (1.000465)
- `-floop-interchange` (0.998829)
- `-fpredictive-commoning` (0.998795)

Interestingly, the flag which yielded by far the best performance improvement for `npb_bt_w` (`-fpeel-loops`) – also caused the worst performance regression for `mmul`.

`-ftree-loop-distribution`

Verbatim from `man 1 gcc`:

This flag can improve cache performance on big loop bodies and allow further loop optimizations, like parallelization or vectorization, to take place. For example, the loop

```
DO I = 1, N
  A(I) = B(I) + C
  D(I) = E(I) * F
ENDDO
```

is transformed to

```
DO I = 1, N
  A(I) = B(I) + C
ENDDO
DO I = 1, N
  D(I) = E(I) * F
ENDDO
```

`-fipa-cp-clone`

When using this flag, the compiler may create multiple copies of functions "to make interprocedural constant propagation stronger." I assume this to mean that, when there are constants being passed to a function, and those constants differ in the places where the function is called, those constants are moved into (copies of) the function itself.

As a consequence of code duplication, this flag can have a significant impact on code size.

`-funswitch-loops`

This flag moves branches contained within loops to their outside, creating multiple "unswitched" copies of the affected loop which are executed based on the branch condition – iff. the branch condition does not change within the loop body. For example, the loop

```
DO I = 1, N
  IF A < 5
    [code path 1]
  ELSE
    [code path 2]
  ENDIF
ENDDO
```

is transformed to

```
IF A < 5
  DO I = 1, N
    [code path 1]
  ENDDO
ELSE
  DO I = 1, N
    [code path 2]
  ENDDO
ENDIF
```