5602 Assignment 2

Kyle Parfrey — kyle@maths.tcd.ie

November 2020

General directions

In this assignment we're going to explore vectorisation using two simple loop structures and our two favourite compilers, gcc and icc. We'll be making minor adjustments to the provided C file, vector_assignment.c, and passing different compiler flags. Don't forget to use the most recent version of gcc (version 9.2), not chuck's default version; you can load this by first running `module load cports`, then `module load gcc`.

For each numbered question below, indicate precisely (a) how you modified the code, and (b) how you invoked the compilers, either by specifying the single-line command you used or how you adjusted your makefile. (You don't need to describe how you returned anything in the code to its original form.) If you used a makefile, include this too.

Each of the questions require you to run the code, for both loop types, for one or both compilers. Produce a single table to organise all the timing results. You choose a loop type by passing "1" or "2" as the first command-line argument (have a look at the `switch` on line 59). You can measure performance for each loop type separately using the `time` command (let's take the "user" time, and round to three or four significant figures). For example, to time the second function, `loops_v2`, you would run

```
chaplinc@chuck vectors]$ time ./intelbinary 2
```

if you've named your binary `intelbinary`. The code returns a result to stderr, which is distinct for the two loop functions.

Detailed plan

For each numbered part (except No. 4), time both loops_v1 and loops_v2. You should compile with both gcc and icc for all parts (except No. 2). Place all timing measurements into a master table. Don't forget to note any modifications to the code and the compiler call (or makefile) for each, if applicable.

1. Compile with default options (no optimisation or architecture flags) for both gcc and icc.
2. Are the two compilers producing exactly the same result for the two loop functions? If not, what's the fractional discrepancy, and what do you think might be causing this? Try adding the flag `-fp-model precise` (yes, that is a space!) to your icc invocation. What effect does this have on speed and the returned result? If the returned results are now the same you should leave this flag in place when calling icc.
3. Compile with gcc and icc, using `-O3` and the correct architecture or vector-extension flag for chuck.
4. Produce vectorisation reports with each compiler. Are the inner loops inside the two functions (lines 13 and 28 in the original C file) being vectorised? What about the loop in main (line 49)?
5. What changes when you add `#pragma omp simd` to the two tight inner loops (lines 13, 28)? Does this affect the vectorisation reports or the timing? You can leave these pragmas in place from now on. (Hint: don't forget any necessary compiler flags!)

6. With the pragmas in place, try compiling with icc without the `-fp-model precise` flag. What effect does this have on performance and the returned floating-point result? If the icc binary now gives the same returned result as gcc, you can leave out the `fp-model` flag from now on.

7. So far we've been using 64-bit (double precision) floating point. Replace this with 32-bit floating point everywhere. Is there a significant speed-up? Is this what you would have expected, and why? (Reminder: in C, explicit floating-point numbers written directly into the code (known as literals) are doubles by default. A *single-precision* number is written as, e.g., `0.027f`, with a trailing "f".)

8. What would the timing have been if you had forgotten to switch the floating-point literals to 32-bit numbers?

9. Switch back to 64-bit floating point everywhere. Now align the dynamically allocated memory, to the byte boundary appropriate for chuck. Check the timing both with the standard pragmas you've been using, and then with an extended pragma which includes a clause asserting that the memory is aligned. How does icc's vectorisation report change when the additional clauses are added?

10. Replace the dynamically allocated memory with statically allocated (stack) arrays, also aligned to the relevant byte boundary. How does the performance compare?

11. Switch back to the original memory setup (unaligned, dynamically allocated). Remove the literals (the 0.027s) from inside the two tight loops. What effect does this have on the timing? If there are any differences, can you use this to guess the cause of some of the performance differences between the two compilers?