Compiler Optimisation Exercise

Adrian Jackson

1st July 2019

1 Introduction

In these exercise we are going to investigate the compiler performance on the ARCHER nodes. We will use the Intel compiler. To get started with the exercises copy the following file:

/home/z01/shared/CompilerOpt.tar

You should be able unpack it with the command `tar xvf CompilerOpt.tar`

We are going to use the Intel compiler for these exercises so make sure you have it loaded. You can check with the command `module list` which will show you the modules you currently have loaded. You should have the `PrgEnv-intel` module loaded, if you don't unload the current programming environment (`module unload PrgEnv-cray`) and load the Intel one (`module load PrgEnv-intel`).

2 Exercise 1: Basic optimisation

The code for this exercise is in `/CompilerOpt/*/Opt1`. The main computation is in the loop in routine fred.

Note that for this and the following two exercises, you are asked to modify the code several times: make sure you keep a copy of each (correct) version, including the original one. Subroutine in-lining should be disabled to ensure the timing calls continue to work correctly.

1. Compile the code with no optimisation (use `-O0 -fno-inline-functions`), and record the performance.
2. Now optimise the code by hand. Record each stage with the optimisation technique used. How much performance gain can you achieve?
3. Finally, compile both the original and your optimised version with

   `-O3 -fno-inline-functions.`

How do the performances compare with your version, and why? Use the `-S` option to generate the assembly code for the various versions.

3 Exercise 2: Loop unrolling

The code for this exercise is in `/CompilerOpt/*/Opt2`. The main computation is in the loop contained in routine sum.

1. Compile the code with `-O3 -unroll=0 -no-vec,` which does not invoke the compiler's loop un-rolling, but allows other optimisations, and record the performance. We disable vectorisation because this also effectively unrolls the loop.
2. Unroll the loop by hand by a factor of 2, remembering to add a clean-up loop.
3. Record the performance. Now generate versions with larger unroll factors: what is the optimum factor?
4. Finally, recompile the original code with `-fast -no-vec` to observe the compiler's own optimisation. Use `-S` to generate assembly code and find out the unroll factor used by the compiler.

4 Exercise 3: Cache optimisation

The code for this exercise is in `/CompilerOpt/*/Opt3`. The main computation is in the loops contained in routine matmul, which forms the product of two matrices.

1. Compile the code with `-O3` and record the performance. For Fortran you will also `need -qno-opt-matmul`
2. Use loop interchange/permutation to improve the cache behaviour. Which loop ordering gives the best performance?
3. Now try tiling all three loops, using the same blocksize for each loop. Experiment to find the optimal blocksize.
4. What happens if you use `-fast` instead of `-O3`?