EC527 -- Programming Assignment 2
Due Tuesday, February 17th

Objectives

- Learn about and practice using methods of optimizing basic blocks accounting for basic characteristics of (i) the compiler and (ii) the microarchitecture. These include dealing with "optimization blockers," FP pipeline latency, and conditional moves.
- In particular the methods of code motion, loop unrolling, accumulation, and branch promotion. Also, understanding branch prediction and memory latency.

Reading

B&O Chapter 5.4-5.12. In fact Lab 2 is very much an exercise in working through B&O Chapter 5.

Prerequisites (to be covered in class or through examples in on-line documentation)

HW – Basic CPU models including pipelined CPUs, branch prediction, and conditional moves.
SW – Optimization methods including code motion, loop unrolling, distributed accumulation, and forcing conditional moves.

--
Assignment
--

In this assignment we assume that data are stored exclusively in the highest levels of the memory hierarchy. You can satisfy this requirement trivially by keeping your vectors small. You are invited to experiment with ranges, but your CPE graphs should be linear. (Why should the graphs be linear as opposed to Assignment 1?)

1. Experiment w/ basic optimization methods as presented in B&O 5.4-5.10

Reference code: test_combine1-7.c
Read the appropriate parts of B&O and the reference code. Be sure you understand how everything works before you get started. To do:
a. Compile and run the code. Capture and graph the data. Find the CPEs of the various options. Then vary the data types and operations and repeat. Tabulate these results as is done throughout the Chapter (e.g., on page 514). Indicate differences from the result shown in the book, if any, and try to explain them.
b. In this part you only need to generate data for data type **double** and operation **multiply**. The existing test code only has a single level of unrolling (2). Unroll by factors up to 10 and graph your CPE results as shown in Figure 5.22. You don't need to do all of the intermediate values (3-10), but you should see whether performance gets worse with unrolling factors greater than 6. Why might this happen?
c. In this part (again) you only need to generate data for data type **double** and operation **multiply**. Add versions of the two parallelization methods (multiple accumulators and reassociative transformation) to the unrolled code. Plot your results and give a graph of CPE versus unroll factor.
Hand in: your modified code, the tables, the graphs, and explanations.

2. Apply basic methods to dot product

Create code: test_dot.c
To do: Create a program (test_dot.c) that performs a dot product. It is probably convenient to base it on one function in test_combine.c, in particular combine4. You only need the data type **double** using **multiply**.
Optimize the code using the methods you practiced in Part 1b,c: loop unrolling and parallelization.
Hand in:

- your code, which should have both the original dot product and the best version of your code
- a description of the optimization methods it contains (brief)
- the CPE and plots of the original unoptimized code and your best version(s).

3. Force and evaluate conditional moves
Reference code: test_branch.c
A somewhat surprising result of CPU optimizations is that performance can be data dependent. On B&O pp. 529-530 there is a description of such a code optimization: Writing code for implementation with conditional moves. test_branch.c has two code samples similar to the code in the book. Note that these two versions may have different performance depending on the data.
To do: In test_branch.c, there are partially finished versions of init_vector (1 and 2). Finish coding them to "force" the data dependent behavior described in the text. Test the two versions on various data sets, generate the performance, and evaluate the results.
Hand in:

- your modified code, together with a description of the data that you generated (how you initialized the vectors) and the code you wrote to generate them,
- the CPEs of the two code versions with respect to the different data sets (and the graphs you used to generate the CPEs)
- explain the CPEs. This may include looking at the assembly language and/or generating data-flow graphs (as shown in B&O Figures 5.18-5.20).

4. Optimization on a slightly more complex application
Create code: test_eval.c
To do:
Problem 5.20 from B&O. This problem is open ended and you will be graded on the quality of both your results and of your description. Start with the code given in the book for direct evaluation -- you do not need to do Horner's Rule. Then apply optimizations from previous sections.
Hand in:

- A description of the methods you tried and how they worked. What was your overall best?
- Your code
- The plots and CPEs of the original versions of direct evaluation and evaluation by Horner's rule, as well of your best results
- For your best version of each, give a justification for its performance. This may include looking at the assembly code and/or generating graphical representations of your code as is done in Figures 5.18-5.20.

Part 5: QC
How long did this take?
Did any part take and "unreasonable" amount of time for what it is trying to accomplish?
Are you missing skills needed to carry out this assignment?
Are there problems with the lab?