EC527 — Programming Assignment 2

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 you will explore the methods of code motion, loop unrolling, accumulation, and branch promotion. Also, extend your understanding of branch prediction and memory latency.

Reading

B&O Chapter 5.4-5.12 (you should already have read 5.1-5.3). 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 – Understanding basic CPU features: pipeline, branch prediction, and conditional moves.
SW – Optimization methods including code motion, loop unrolling, distributed accumulation, and forcing conditional moves.

Assignment

The point of this assignment is to look at different ways to make repetitive calculations faster, by looking at how loops can be improved and measuring the resulting changes in CPE (cycles per element). In order to focus just on these loop improvements, we want to be able to assume that data are stored exclusively in the highest (fastest) 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 the previous assignment?)

1. Experiment with 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:

1a. Compile (using `-O1`, but also try `-O0` once if you're curious) and run the code.

Each of the main test loops chooses the vector lengths to test by evaluating the polynomial Ax^2+Bx+C for values of x from 0 to `NUM_TESTS-1`. Adjust the coefficients `A`, `B` and `C` to make it cover a larger range of sizes; you can also increase `NUM_TESTS` to get more data points. However, make sure the largest size will fit in the level 1 cache (32K bytes). In other words, when x=`NUM_TESTS`, make sure that (Ax^2+Bx+C) × `sizeof(data_t)` is less than 32K bytes.

Find the CPEs of the seven options (`combine1` through `combine7`). Vary the data types and operations to see what changes.

Make some tables and graphs, like the examples seen in the textbook (e.g., section 5.9.1, pages 514 and 515). You *do not need to do all combinations*, just pick some you think are important. If you see differences from the result shown in the book, tell what you see and try to explain the differences.

1b. In this part you only need to generate data for data type **double** and operation **+** (addition). The existing test code does unrolling, but only by a factor of 2. Unroll by factors up to 10 using the method seen in `combine5`. Graph your CPE results as shown in Figure 5.22. The plain "non-unrolled" loop (`combine4`) can be plotted as "unroll factor = 1". 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?

1c. In this part (again) you only need to generate data for data type **double** and operation **+** (addition). Add versions of the two parallelization methods (multiple accumulators seen in `combine6`, and reassociative transformation seen in `combine7`) to the code you made for part 1b. Plot your results and give a graph of CPE versus unroll factor.

Hand in: your modified code, tables, 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()`. Use the data type **double**. Optimize the code using the methods you practiced in Part 1b and 1c: loop unrolling and parallelization.

Hand in:

- your code, which should include both the initial simplest version of dot product, as well as the best version you came up with
- a description of the optimization methods it contains (brief)
- the CPE and plots of the simple version 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 `branch1()` and `branch2()` analogous to the code (`minmax1` and `minmax2`) 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 functions `init_vector_pred` and `init_vector_unpred`. Finish coding them to "force" the data dependent behavior described in the text: the first function should fill the whole array with the same values (so the array contents are "predictable"), and the second function should fill the array with a pattern (possibly random, but at least unpredictable enough so that the branch predictor cannot handle it).

3a. Use the same coefficients `A`, `B`, and `C` that you chose for part 1 so that the vector size always fits in the fastest cache (32K bytes). Compile with the option `-O1`. It measures the speed of the two functions `branch1()` and `branch2()` using the two types of data (predictable and unpredictable) for a total of 4 combinations. Calculate the CPE for the 4 versions, and discuss the results.

3b. *OPTIONAL:* If `branch1` runs at the same speed on both types of data, it may be that the compiler is too smart. As an alternative, you can try the "Compiler Explorer" at `https://godbolt.org`

Enter the following code into the left side:

```
float max_if(double n1, double n2) {
  float rv;
  if (n1 > n2) {
      rv = n1;
  } else {
      rv = n2;
  }
  return rv;
}

float max_ce(double num1, double num2) {
  float rv;
  rv = (num1 > num2) ? num1 : num2;
  return rv;
}
```

On the right side, from a menu you can select what compiler and version to use (for example **x86-64 gcc 4.8.5**). Make sure to also put "**-O1**" in the box to the right of the chosen compiler version. Many versions of GCC generate something like this:

```
  max_if(double, double):
    comisd      xmm0, xmm1
    jbe         .L8
    cvtsd2ss    xmm0, xmm0
    ret
  .L8:
    pxor        xmm0, xmm0
    cvtsd2ss    xmm0, xmm1
    ret

  max_ce(double, double):
    maxsd       xmm0, xmm1
    cvtsd2ss    xmm0, xmm0
    ret
```

The first function needs to use a branch instruction (`jbe`, highlighted) The second function uses the "`maxsd`" instruction and does not need to compare or branch. Having a mix of types (`float` and `double`) seems to help in this case: the need to convert when making the assignment seems to confuse the compiler's optimiser just enough to make the two versions generate different code. (On the left side, change all four occurrences of "`float`" to "`double`" and see what happens!)

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 could include looking at assembly language (as explained in the optional section) or generating data-flow graphs (as shown in B&O Figures 5.18-5.20).

Part 4: QC

4a. How long did this take?
4b. Did any part take and “unreasonable” amount of time for what it is trying to accomplish?
4c. Are you missing skills needed to carry out this assignment?
4d. Are there problems with the lab?