ECE 565
Fall 2018

Homework Assignment #1

1. (30 *points*) **Code Optimization – "Beat the Compiler".** For this problem, use the separate C program source file attached under the Assignments section of the course website (loop_performance.c). You will attempt to perform loop transformations to see if you can improve the execution time of the code beyond what is provided by the compiler. The code of interest is contained in the do_loops() function. The program prints out both a validation result (so that you can ensure your loop transformations maintain correctness), and a run-time for the do_loops() function in milliseconds. Build and run the program as follows:

```
gcc –O2 –o loop_performance loop_performance.c
./loop_performance 100000000
```

Create a writeup to summarize the following steps:

(a) Compile and run the unmodified base code, and record performance. You should experiment with GCC optimization level -O2 and -O3. Note that you may want to run these experiments in a non-virtualized system (i.e. not on a virtual machine) to obtain good, stable performance measurements.
(b) Describe the type of machine and environment you are running on, including: 1) processor architecture (e.g. x86, ARM, etc.), 2) CPU frequency, 3) OS type, 4) whether in a VM or a standalone system.
(c) Study the code in the do_loops() function. Try to identify loop optimizations that may improve performance. Implement and study the performance of these optimizations. Record performance observations for each individual transformation that you study. Also, include the final code you arrive at for the do_loops() function in your writeup. Remember, as you work through your loop transformations, you will likely want to use the "objdump -d a.out" command to view the assembly code produced for the do_loops() function.
(d) Discuss whether you can beat the compiler (and your thoughts on why or why not). In this case, beating the compiler would mean that your hand-tuned code compiled with -O2 or -O3 gets better performance than the original code compiled with -O2 or -O3.

2. (20 *points*) **Dependence Analysis.** For the code shown below:

```
...
for (i=1; i<=N; i++) {
   for (j=1; j<=i; j++) {  // note the index range!
         S1: a[i][j] = b[i][j] + c[i][j] * a[i+1][j-1];
         S2: b[i][j] = a[i-1][j-1] * c[i-1][j];
         S3: c[i+1][j] = a[i][j];
         S4: d[i][j] = d[i-1][j+1];
   }
}
...
```

(a) Draw its Iteration-space Traversal Graph (ITG)
(b) List all the dependences. Clearly indicate whether each dependence is loop-independent vs. loop-carried. Clearly indicate the type of each dependence (i.e. true, anti, or output).
(c) Draw its Loop-carried Dependence Graph (LDG)

3. (20 *points*) **Function In-lining and Performance.** For this question, you will use another provided source code file (func_inlining.cpp). You will study the performance and the differences in assembly code for a small function which is called many times. This function simply adds two integers and returns the result. It is used to add two arrays together. The code can be compiled and executed as follows:

```
g++ -O3 -o func_inlining func_inlining.cpp
./func_inlining 100000000
```

To study the effects of function in-lining, you will use a function attribute to instruct the compiler to either always inline or never inline the `add()` function. You can read about such attributes and their syntax at the following resource:
https://gcc.gnu.org/onlinedocs/gcc-4.1.2/gcc/Function-Attributes.html

You should:
(a) Run and record performance both with and without inlining of the `add()` function.
(b) Show assembly code snippets to illustrate how inlining affects the resulting compiled code within the array addition loop. Screenshots or pasted code snippets are a good way to show this (and again, recall the 'objdump -d' command). And as a hint, you can use the timer calls to gettimeofday to help locate the code of interest as these timer calls surround the array addition loop.
(c) Discuss whether your measured performance results match your expectations. Why or why not?
(d) Also compare this to the performance of the original code (with no attributes on the `add()` function). Based on this, do you think the compiler is in-lining the `add()` function by default?

4. (15 *points*) **Loop transformations.** For the code shown below:

```
...
int a[N][4];
int rand_number = rand();
for (i=0; i<4; i++) {
   threshold = 2.0 * rand_number;
   for (j=0; j<N; j++) {
      if (threshold < 4) {
         sum = sum + a[j][i];
      } else {
         sum = sum + a[j][i] + 1;
      }
   }
}
...
```

This C++ code contains opportunities for optimization with loop transformation(s). Identify any such loop transformations that seem suitable for optimization. Starting with the original code, for each transformation (in any order you would like to apply the transformations), show the code to reflect that transformation. In other words, progressively apply each transformation to the code (showing each version along the way). The final code should reflect the changes from all identified transformations.

5. (15 *points*) **Loop transformations.** For each loop transformation below, identify whether the transformation is safe or unsafe. The transformation is unsafe if the transformed code may produce a result that is different from the original code. For each transformation identified as unsafe, describe and show why the code does not meet the required safety condition(s) as described in the class lecture notes.

(a) Loop fusion

```
...
//Original code
for (i=1; i<8; i++) {
   S1: a[i] = b[i] + 2;
   S2: c[i] = a[i] + b[i];
}
for (i=1; i<8; i++) {
   S3: a[i+1] = d[i] + 4;
}
...
```

```
...
//Code after Loop Fusion
for (i=1; i<8; i++) {
   S1: a[i] = b[i] + 2;
   S2: c[i] = a[i] + b[i];
   S3: a[i+1] = d[i] + 4;
}
...
```

(b) Loop interchange

```
...
//Original code
for (i=1; i<=4; i++) {
  for (j=1; j<=4; j++)
    S1: a[i][j] = a[i+1][j-1];
}
...
```

```
...
//Code after Loop Interchange
for (j=1; j<=4; j++) {
  for (i=1; i<=4; i++)
    S1: a[i][j] = a[i+1][j-1];
}
...
```

(c) Loop fission

```
...
//Original code
for (i=1; i<=4; i++) {
   S1: a[i+1] = b[i] + c[i];
   S2: d[i] = a[i-1] + 4;
}
...
```

```
...
//Code after Loop Fission
for (i=1; i<=4; i++) {
   S1: a[i+1] = b[i] + c[i];
}
for (i=1; i<=4; i++)
   S2: d[i] = a[i-1] + 4;
}
...
```