Homework 1b - CIS501 Fall 2011

Instructor:	Prof. Milo Martin
Due:	Noon, Thursday, Sept 29th (at the start of class)
Instructions:	This is an *individual* work assignment. Sharing of answers or code is strictly prohibited.
Note:	This is part of a two-part assignment (HW1a and HW1b) due on the same day. Please print out your answers for each part and turn them in as **two distinct stapled documents**.

Effectiveness of Compiler Optimizations

The compiler's goal is to generate the fastest code for the given input program. The programmer specifies the "optimization" level the compiler performs. To ask the compiler to perform no compiler optimizations, the "-O0" flag can be used (that is "dash oh zero"). This is useful when debugging. It is also the fastest compilation and the least likely to encounter bugs in the compiler. The flag "-O2" (dash oh two) flag enables optimizations, the "-O3" (dash oh three) enables even more optimizations. Finally, the flag "-funroll-loops" enables yet another compiler optimization.

This question asks you to explore the impact of compiler optimizations using the following simple single-precision "a*x+y" (saxpy) vector-vector operation:

```
#include <omp.h>

#define ARRAY_SIZE 65536

float x[ARRAY_SIZE];
float y[ARRAY_SIZE];
float z[ARRAY_SIZE];
float a;

void saxpy()
{
#pragma omp parallel for
  for(int i = 0; i < ARRAY_SIZE; i++) {
    z[i] = a*x[i] + y[i];
  }
}
```

This is available as code.c and is run by linking it with driver.c. Log into minus.seas.upenn.edu via SSH, and compile and run the code with different levels of optimization:

```
gcc -std=c99 -O0 -S code.c -o code-O0.s
gcc -std=c99 -O3 -fno-tree-vectorize -S code.c -o code-O3.s
gcc -std=c99 -O3 -funroll-loops -fno-tree-vectorize -S code.c -o code-O3-unrolled.s
gcc -std=c99 -O3 -funroll-loops -ftree-vectorize -S code.c -o code-O3-unrolled-vector.s
gcc -std=c99 -O3 -funroll-loops -ftree-vectorize -c -fopenmp code.c -o code-O3-unrolled-vector-openmp.o

gcc -static code-O0.s driver.c -o code-O0
gcc -static code-O3.s driver.c -o code-O3
gcc -static code-O3-unrolled.s driver.c -o code-O3-unrolled
gcc -static code-O3-unrolled-vector.s driver.c -o code-O3-unrolled-vector
gcc -static code-O3-unrolled-vector-openmp.o driver.c -o code-O3-unrolled-vector-openmp -fopenmp

./code-O0
./code-O3
./code-O3-unrolled
./code-O3-unrolled-vector
./code-O3-unrolled-vector-openmp
```

> **Note**
>
> Some of these versions of the code will take up to a minute to complete executing, so be patient.

The above code builds and runs several versions of the program. To run the code, it links each source file with driver.c, which calls the function 200,000 times and reports the runtime in seconds.

In addition to building and running the code, the above commands will also generate the intermediary .s assembly files in the directory. These files are the x86 assembly of this function. You can ignore everything except the code between the label *saxpy* and the return statement (`ret`).

Answer the following questions:

1. **Total speedup.** Record the runtime of each of these versions. Plot the overall speedups (y-axis) over the -O0 baseline. The x-axis has one bar per configuration (five total). The first bar is -O0 and it has a speedup of 1x (by definition). The other bars are all of increasing height, and represent the total speedup over the -O0 baseline (larger speedup means smaller runtime).

   > **Note**
   >
   > If you're not seeing a significant speedup for the last one (-fopenmp), you should run the code a few times and take the fastest of the execution times reported.

2. **Relative Speedup.** How much faster is -O3 than -O0? How much faster is unrolled over -O3? How much faster is unrolled & vectorized over just unrolled? How much faster is "-fopenmp" than unrolled & vectorized?

3. **Static instruction counts.** As you can see from the data, there is a large performance difference between -O0 and the most optimized version. To better understand why, the following questions ask you to look at the .s files to determine instructions counts and such. First, what is the static instruction count (the number of instructions in the source code listing) of just the loop (ignore the header and footer code to count just the number of instructions in the loop)?

   > **Note**
   >
   > The flag -ftree-vectorize enables "automatic vectorization". This computation is one that GCC can vectorize, so GCC uses "packed" versions of the various instructions that operate on four elements of the array at a time (that is, load four values, multiple four value, store four values, etc.).

   > **Note**
   >
   > You don't have to understand much x86 code to answer the following questions, but you may find seeing actual x86 code interesting. For example, instead of using "load"

> and "store" instructions, in x86 these instructions are both "mov" (move) instructions. An instruction like `add %ecx, (%esp)` performs both a memory access and an add (an example of a CISC-y x86 instruction). The x86 ISA also has some more advanced memory addressing modes, such Also, x86 uses condition codes, so jump (branch) instructions don't have explicit register inputs. If you want to lookup an x86 instruction, you may find the Intel x86 reference manual useful: http://www.intel.com/products/processor/manuals/

4. **Dynamic instruction count.** Based on the number of instructions in the loop (above) and the number of iterations of the loop in each .s file, calculate the approximate number of **dynamic** instructions executed by the loop in the function (approximately, to the nearest few hundred instructions). Do this calculating only the first four configurations (not the -fopenmp one).

> **Hint**
>
> Be careful to consider the number of iterations, which does change for some of the optimizations levels.

> **Note**
>
> The assembly code generated with the -fopenmp flag is too hard to follow, so you don't need to look at the assembly code for that configuration.

5. **Static vs dynamic instruction count.** When comparing the most optimized .s files with the less optimized files, how does the number of static instructions relate to the dynamic instruction count? Give a brief explanation.
6. **Performance estimate based on dynamic instruction count.** Just using the dynamic instruction counts, for a processor with a CPI of 1 how much "faster than" is the -O3 than -O0? How much is unrolled faster than -O3? How much is vectorized over -O3 + unrolling?
7. **Loop unrolling pro/con.** Using what you learned from this data, what is one advantage and one disadvantage of loop unrolling?
8. **Estimated vs. actual performance**. How does the above performance estimates based on instruction counts compare to the actual runtimes you recorded earlier?
9. **CPI calculation.** The clock frequency of the processors in "minus.seas.upenn.edu" is 3.0 Ghz (3 billion cycles per second). Based on your estimate of the number of instructions in each call to the function and that driver.c calls the function multiple times, calculate the CPI of the processor running each of these programs.
10. **CPI and compiler optimizations**. What do the above CPI numbers tell you about these compiler optimizations? You may wish to calculate the IPC of each, as sometimes it is easier to reason about IPC (instructions per cycle) rather than CPI (the inverse metric).

Addendum

- None, yet.

Generated on: 2011-09-19 20:26 UTC.