Week 7 — October 25, 2018

Homework

- Read chapter 12 pp. 156–170 in "Writing Scientific Software"

Exercises

Part I

In this first exercise today, you should write a program that "times" the function `datasize1()` from the lecture. The goal is to produce a performance graph, as the one shown in the lecture, that illustrates the performance dependency on the different cache levels in your computer.

1. Write a `main()` function, that calls `datasize1()` for different numbers of elements, and time the execution. To get reliable results, you might need to time several function calls, and then take the average, e.g. by adding a loop around the call. Use `clock()` to measure the CPU time. You may use the following template for your program:

```c
/* ex1.c */
#include <stdlib.h>
#include <stdio.h>
#include <time.h>

#define MAX_SIZE 16777216  // 128*1024*1024/8 elements
extern int datasize1(int);
double arr[MAX_SIZE];      // global array of length MAX_SIZE

int main(void) {
    clock_t t1,t2;

    /* Insert your code here */

    return EXIT_SUCCESS;
}
```

   Create a *separate* file with the `datasize1()` function:

```c
/* datasize1.c */
extern double arr[];
int datasize1(int elem) {
    for (int i=0; i<elem; i++) arr[i] *= 3;
    return(elem);
}
```

And create a Makefile:

```
CFLAGS=-Wall -std=c99 $(OPT)
ex1: datasize1.o
.PHONY: clean
clean:
    -$(RM) ex1 *.o
```

You can *hardcode* the optimization flags by adding these instead of `$(OPT)` as part of the specification of `CFLAGS`, or you can supply these by setting the variable `OPT` when invoking `make`. For example, to compile with level-3 optimization, call `make` as follows:

```
$ make OPT=-O3 ex1
```

Note that if you may need to recompile everything to ensure that all pieces of your program have been compiled with the same optimization flags. In other words, start by executing `make clean` if you want to recompile with a different set op optimization flags.

2. Instead of looking at the timings, you can look at the performance, measured in Flop/s (floating point operations/second). This can be easily calculated from the timings. How?

   Hint: The loop in `datasize1()` has one floating-point operation (multiplication) per loop element.

3. Plot your performance numbers as a function of the memory size of the array, ranging from a few kilobytes (kB) to some Megabytes (MB). Can you observe the 'stepping' behavior, and do the different steps correspond to the cache size levels in your computer? Use the commands shown in the lectures to get the specifications of your computer.

   Why is the performance in Flop/s a better measure than the *raw* timings in this case?

   Hints:

   - You may need to compile with full optimization to see the steps.
   - Use a log-scale with base 2 for the x-axis in the plot.

Part II

In the following exercises, you will implement your own **dgemv** routine, i.e., a function that computes the matrix-vector product

$$y \leftarrow \alpha Ax + \beta y$$

where A is a matrix of order $m \times n$, x is a vector of length n, y is a vector of length m, and α and β are scalars.

We will compare different implementations by measuring the CPU time with the `clock()` funtion for different values of m and n. You can re-use the framework from Part I to measure the timings. In two weeks, we will parallelize the code using OpenMP.

1. Go to CodeJudge and complete the "Week 07" exercises. You will need the two functions that you write as part of these exercises (`my_dgemv_v1()` and `my_dgemv_v1()`) in the remaining exercises.

2. Write a program that measures the CPU time required by `my_dgemv_v1()` for $m = n$ and $n \in \{100, 200, \ldots, 1000\}$.

3. Measure the CPU time required by `my_dgemv_v2()` for $m = n$ and $n \in \{100, 200, \ldots, 1000\}$. Compare with the CPU time required by `my_dgemv_v1()`. Which of the two methods is faster and why?

4. Compile your code with different compiler optimization flags and repeat the two timing experiments (i.e., measure the CPU time required by `my_dgemv_v1()` and `my_dgemv_v2()`). Try with each of the following flags:

   - `-O0` — no optimization
   - `-O1` — enables basic optimizations; no speed-space trade-offs
   - `-O2` — enables further optimization; no speed-space trade-offs
   - `-O3` — most expensive optimizations; may increase size of executable

   How much does compiler optimization affect the results? If you are using GCC, you may also try to enable loop-unrolling with the `-funroll-loops` compiler flag.

   Remark: The optimization flags do not work the same for all C compilers. For example, `-funroll-loop` is recognized by GCC and works independently of the `-O` compiler options, but Clang enables loop-unrolling with the `-O1` option. For more information about GCC and optimizations, see Options That Control Optimization.

5. (Optional) Instead of looking at the timings, you can also measure the performance in Flop/s of your `my_dgemv_..()` functions. How can this be achieved?

   Hint: Count the number of floating-point operations in the loop body. The total number of operations depends on m and n.