Lab Sheet: Cache-friendly C Programs

This lab sheet covers C programming, and in particular how to improve the performance of a C program by improving its cache behaviour. The tasks should be performed on one of the Linux lab (EM 2.50) machines in the department, mainly for convenience. You can try the same exercise directly on a RPi2, but expect to get quite difference performance numbers on this device, which uses an ARM rather than an Intel chip. You will need to install valgrind separately if you work on an RPi2, by typing:
`sudo apt-get install valgrind`

Samples of chache-friendly C programs

In the lecture on the Memory Hierarchy we discussed two versions of computing the sum over all elements in a matrix `arr` of size `n`×`n`. The first one uses **row-wise iteration** like this:

```
for (i = 0; i<n; i++)    /* iterate over rows */
  for (j = 0; j<n; j++)  /* iterate over columns */
    sum += arr[i][j];
```

and the second uses **column-wise iteration** like this:

```
for (j = 0; j<n; j++)    /* iterate over columns */
  for (i = 0; i<n; i++)  /* iterate over rows */
    sum += arr[i][j];
```

Task: Compare the cache performance of these programs

Write a C program, implementing each of the above functions to compute the sum over all matrix elements. You will need to write a function to generate a random matrix of a given size as test input (see Note 4 in the CW spec for Greg's C programming CW). Then complete the two functions for doing the sum, one row-wise, one column-wise, based on the core code above. Finally, add timing commands to measure the time for executing the sum operation (use the function `clock()`, imported from `time.h` to get a time-stamp).

Measure and compare both versions of the program. Explain why you see a big difference in performance for both versions. Use `valgrind` (see below) to get more detailed information about the cache performance of your programs.

Extensions: Modify the code of the two loops above to use different types for the elements in the matrix (import from `stdint.h`): use `int`, `long`, `uint16_t`, `uint8_t`. Based on your knowledge of the size of these types in memory and of the cache-line size on an Raspberry Pi 2, how do you expect the runtimes to change when you switch to one of the above types? Measure the two versions of the programs and test you hypothesis on the predicted runtime against the measured times.

Optionally: If you have a Raspberry Pi 3, which is a 64-bit architecture, run the same set of measurements on the RPi3 as well. What are your predicted runtimes on the RPi3, based on your measured runtimes on the RPi2, the performance comparisons of these devices on the slides on "Systems Programming on the RPi2" and your knowledge about the two architectures (in particular the cache management)?

Matrix multiplication (Week 6)

Download and execute the two versions of matrix multiplication discussed in class of Week 5:

- naive matrix multiplication: matrix2.c
- improved matrix multiplication: matrix9.c
- using this input data of 2 matrices: Input matrices: 2000x2000, density 10%, max value 65536
- to test execution with a smaller input first, you can use this input file: Input matrices: 5x5, 80%, max value 99

Task: Time the execution of each of these programs. Can you explain why the second program is much faster than the first one? Check this explanation of blocked matrix multiplication, as in `matrix9.c`, and why it is faster than the naive one.

Use the valgrind tool, as introduced in class, to measure the cache behaviour of these two programs. What does the output statistics tell you about the two programs?

Cache-behaviour of your CW1 application (Week 8)

After finishing CW1, take the C program that you have developed, and analyse the cache performance of your implementation. Do you get a good cache hit rate in your program? If not, can you restructure the program to get a better (or worse) cache hit rate?

Consider reordering the loops in your implementation, and use different optimisation options (`-O`, `-O2` or `-O0`) and observe how the cache behaviour and the program performance changes.

How to compile and run the programs

To execute and run the two versions of matrix multiplication, download the source code for both and the file with input matrices, and then do:

```
> gcc -O -o matrix2 matrix2.c
> ./matrix2 2MAT_2000_10_65536
2000 * 2000; SEQUENTIAL; 447.350000 secs
```

How to use Valgrind

On the Linux lab machines, `valgrind` is already installed and you can run it there like this:

```
> valgrind --tool=cachegrind ./matrix2 2MAT_5_80_99
==5316== Cachegrind, a cache and branch-prediction profiler
...
```

Look out for a line like this, which summarises the *(data) cache miss rate for the level 1 cache*:

```
...
==5316== D1  miss rate:       0.3% (    0.4%     +     0.1%  )
...
```

The lower this number, the better the performance of the program will be.
If you don't have `valgrind` installed, do the following with the RPi2 connected to your local network:

```
> sudo apt-get install valgrind
```