Assignment 3 (15%)

CompSys304 2020

A/Prof Oliver Sinnen

Submission (1%)

- **Due:** Friday 23 October, 11am
  - **Late penalties**: up to 24h late: -20% of achieved marks, 24h to 48h late: -30% of achieved marks, more than 48h late: 0 marks
- **Where:** Canvas
- **What:** 1 zip file, containing:
  - `assignment3_answer.pdf` with answers;
  - folder `task_1`: `Makefile`, `cachetest1.c`, `cachetest2.c` (one for each case);
  - folder `task_2`: `Makefile`, `matrix1.c`, `matrix2.c`, `matrix3.c` (one for each implementation).
  - There must not be other files in the zip file than the ones specified – you better check, also for hidden files!

This assignment can be done on any computer system you have access to, where the compiler `gcc` and the build tool `make` are installed. This is standard in most Linux distributions, under Windows you can use cgywin (https://www.cygwin.com/). When using a virtual machine, the measurements will be less reliable and less repeatable. Use the `assignment.c` and `Makefile` templates from Canvas to write your programs and pay careful attention to the comments in those files. *Adhering correctly to these specifications gives you 1% (out of 15%)!*

1 Cache measurement (6%)

Your task is to write a short C program that measures the access time differences of the caches in your system. To perform the measurement write a short program that consists of a loop with N iterations over an integer array `a` of size N, where you sum the value of each element of `a`. The execution of this loop is repeated M times, where M needs to be a large number. Measure the time it takes to execute this and divide by the total number of iterations, i.e. $M \cdot N$, to get the time per iteration. Repeat this measurement for many different sizes of N (also changing the size of the array `a`), ranging from smaller than half the L1 cache to much bigger than L3 cache. Do these measurements in two different ways:

1. Go linearly through the array `a`
2. Go randomly through the array `a`

To achieve the two different ways and have a fair comparison, use a helper array `b` which you initialise with values from 0 to $N-1$ in order. Use `b` as the index for array `a`, i.e. `a[b[i]]`. For case 1 you use `b` directly after initialising, in case 2 you first shuffle the values. Do this by randomly choosing two elements of `b` (using `rand()`) and swap their values. Repeat this swapping at least N times.

Details

Initialise array `a` with some values and make sure that you get the same sum in both cases. The array size and number of repeats should be input parameters of your program, see template. The runtime of what you measure must be at least 400ms, otherwise your measurements will not be precise. Seconds is better.

Document the following in your pdf

- Name of your processor, e.g. Intel Core i7 7500U (use e.g. from /proc/cpuinfo)
- Cache sizes of your system. Determine the cache sizes of your system for example using `getconf -a`, or `cat /proc/cpuinfo` or any other method.
- Table of measured time per iteration for each case. An example for such a table is this [1]:

N	size of `a`	time per iteration/ns	time per iteration/ns
		case 1	case 2
2048	8 KB	5	25
...	...	...	...
16M	64 MB	70	300

- Chart (time-per-iteration over size of `a`) of the previous table visualising the behaviour of the two cases.
- Briefly explain the differences (or similarities) of the results. Explain changes due to the different sizes of `a` and differences between the two cases.

Notes

To avoid unexpected results in the measurements, it is important that the computed results of the measured code are used. Compilers perform an optimization called 'dead code elimination'. If a calculation is done in your code but never used, the compiler might completely drop the code. For your assignment that means if you calculate the sum, but never use it, the compiler generates no code for it. To avoid this, have a `printf` of the sum after the timing.

Also, when timing your code, make sure that nothing else is running on your system. It is recommend not to run your experiments in a virtual machine for that reason.

2 Matrix product (8%)

Your task is to write a short C program that measures the time to perform the product of two $N \times N$-size matrices A and B, $C = AB$. The entries of result matrix C are calculated as

$$c_{ij} = \sum_{k=1}^{m} a_{ik} b_{kj}$$

1. Write a straight forward implementation of this matrix product. Measure and document the time it takes in your pdf. Identify what the problematic access pattern is in this implementation.
2. Write a new implementation where you solve this problem by using a temporary matrix (hint: transpose one matrix). Measure and document the time it takes in your pdf. Include the creation of the temporary matrix in your time measurement.
3. For large matrices it might not be possible to have a temporary matrix copy due to memory size. Anyway, we can do even better. Based on the original approach, use blocking, that is process $k \times k$ blocks at a time. (Hint: k should relate to the cache line size). Write an implementation based on blocking. Measure and document the time it takes in your pdf. Explain your approach briefly. (Hint: it might not be easy to get a good improvement, an in-principle-correct attempt will be considered good enough.)

Use $N = 1000$ and arrays of `double` for the data type.

[1] Values are not necessarily realistic.