PHY905, section 4
Homework Assignment 4

Håkon V. Treider
MSU id: treiderh

March 10, 2016

Introduction

The tests have been run on Mac OS X (10.11.3) with a 2.5 GHz Intel Core i7.

In this homework assignment we are going to look at the matrix-matrix multiply and how we can make very basic and simple models for the performance. Our first model will use the assumption that data is read once and then remains in cache without any cost of access.

In the following formula, we have the different variables: c, r, w, n which corresponds to a floating point operation, a read from memory, a write to memory and the size of the matrix.
For a square $n \cdot n$-matrix we have $2n^3$ FLOPS and that requires $2n^2$ values to be readily available in memory - n squared values per matrix must be read into cache. When computation is done, half the number of values must be written back into memory (only C). This gives us the formula:

$$\begin{aligned} T_1 &= f(c, r, w, n) \\ &= 2c \cdot n^3 + 2r \cdot n^2 + w \cdot n^2 \\ &= 2n^2 \left(cn + r + \frac{w}{2} \right) \end{aligned} \tag{1}$$

Let us now make a bit more tricky assumption. We assume one of the matrices, i.e. B, does not fit in cache, but that everything else still does. How will this change our formula? We now need to read each value in B from memory each time it is used. This complicates the formula with rn^3-term, but we also get rid of the factor $\frac{1}{2}$ in the n squared term for memory-read above. We end up with:

$$\begin{aligned} T_2 &= 2c \cdot n^3 + r \cdot n^3 + r \cdot n^2 + w \cdot n^2 \\ &= n^3 (2c + r) + 2n (r + w) \end{aligned} \tag{2}$$

For the processor performance, we assume 5 GFlops and for the memory bandwidth, 8 GB/sec. This gives us the following numbers for the variables, - and we end up:

$$
\begin{aligned}
c &= 0.2 \cdot 10^{-9} \ \text{sec/FLOPS} \\
r = w &= 10^{-9} \ \text{sec/double precision word} \\
n &= 10^3 \\
T_1 &\approx 0.4 \ \text{sec} \qquad (3) \\
T_2 &\approx 1.4 \ \text{sec} \qquad (4)
\end{aligned}
$$

Results

In figure 1 we can see a few examples of how optimisation level of the compiler plays a role on performance. The first thing to notice is that for matrices larger than n = 1000, we *should* use the blocked code due to an enormous surge in speed. For smaller matrices, we can see that the compiler setting for optimisation is what matters the most. The blocked code needs a higher level of optimisation to actually perform better than the simple code, if we think of performance in orders of magnitude. I find this, in particular, interesting.

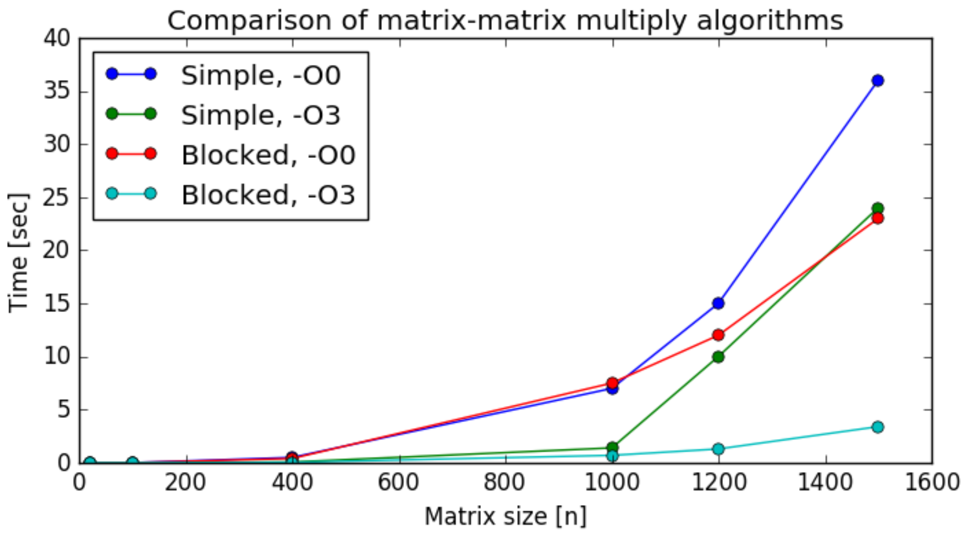
When it comes to what matrix size is too large for L3-cache, I found that to be around 850:

$$\frac{L_3 \text{ cache size}}{\text{Bytes per double}} = \frac{6 \text{ MB}}{8 \text{ B}} = 7.5 \cdot 10^5 \text{ elements} \qquad (5)$$

$$n = \sqrt{7.5 \cdot 10^5} \approx 866 \qquad (6)$$

We can see this behaviour from the figure; the time it takes to compute has a steep incline around the n=1000 mark for the simple code with high-level optimisation.

An important question to ask, is whether our performance models was ”on the right track” or not. With compiler optimisation turn on, we see that they are in very good agreement (0.4, 1.4, 1.4, 0.7 seconds). Without - we are off by an order of magnitude. The ”perfect” agreement of assumption 2 (1.4 sec) shows that it is a good indicator of what performance we would expect - it should act as a lower limit/bound that we can use for more advanced techniques like blocking.



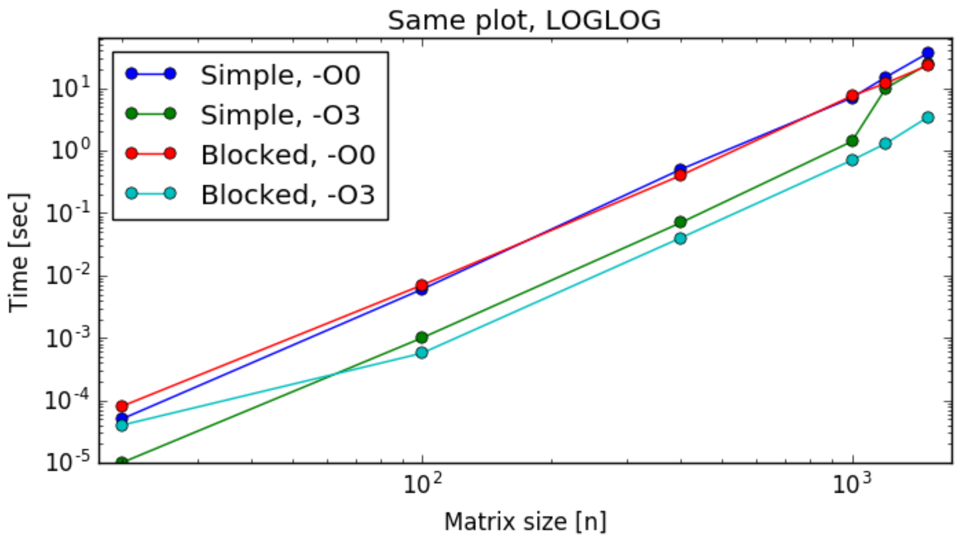


Figure 1: Showing speed gain/loss by blocking and different compiler flags (optimization)

Table 1: Runtime results of simple and blocked code with different compiler optimisation levels. All results in seconds

n	Simple -O0	Simple -O3	Blocked -O0	Blocked -O3
20	5E-5	1E-5	8E-5	4E-5
100	0.006	0.001	0.007	0.00057
400	0.5	0.07	0.4	0.04
1000	7.0	1.4	7.5	0.7
1200	15	10	12	1.3
1500	36	24	23	3.4

Code (blocked)

```
// Assumes "n" is a multiple of "bsize"
int kk,jj,i,j,k;
double sum = 0;
int en = bsize*(n/bsize);
for (kk = 0; kk < en; kk += bsize) {
    for (jj = 0; jj < en; jj += bsize) {
        for (i = 0; i < n; i++) {
            for (j = jj; j < jj + bsize; j++) {
                sum = C(i,j);
                for (k = kk; k < kk + bsize; k++) {
                    sum += A(i,k)*B(k,j);
                }
                C(i,j) = sum;
            }
        }
    }
}
```