

## Exercise 2: Matrix Multiply Exploration

### Background

This exercise concerns a familiar computational kernel: matrix–matrix multiplication. Here’s a simple C implementation:

```
/*
 * Matrix-matrix multiplication for studying cache performance:
 * C = AB, where all the matrices are of dimensions N x N
 */
void mm(double A[M][M], B[M][M], C[M][M])
{
    int i, j, k;

    for (i = 0; i < N; i++) {
        for (j = 0; j < N; j++) {
            C[i][j] = 0;
            for (k = 0; k < N; k++) {
                C[i][j] += A[i][k] * B[k][j];
            }
        }
    }
}
```

### Systematic investigation

Your job is to characterise the behaviour of various variants of the matrix multiply program: the straightforward version, as shown in the code above, an interchanged “ikj” variant, and a more sophisticated, “tiled” version. You can find source code at

`~phjk/ToyPrograms/ACA22-23/MM`

### Getting started

#### Log into a Department of Computing Linux teaching lab desktop machine

Although it is possible to do this exercise using your own machine, it’s actually recommended to use a DoC lab machine - not least so that you can leave it running for hours while you use your own machine for something else! Note that this exercise is a warmup for the first assessed exercise.

You will need to connect via ssh remotely. First connect to one of the login servers `shell11`, `shell12`, `shell13`, `shell14` — for example as follows:

```
ssh shell12.doc.ic.ac.uk
```

Then, from there, connect to one of the DoC lab desktop machines. You are recommended to try `texel1`, `texel2`, ... `texel44`. For example:

```
ssh texel19
```

Not all the `texel` machines are available — keep trying til you find one (ideally an idle one) <sup>1</sup>

**The files provided** Using one of the DoC CSG Linux systems (level 2, Huxley), make your own copy of the ACA22-23/MM directory tree by executing the following Linux commands:

```
prompt> mkdir ACA22-23
prompt> cd ACA22-23
prompt> cp -r ~/phjk/ToyPrograms/ACA22-23/MM ./
```

(The `./` above is the destination of the copy – your current working directory). You should now have a copy of the MM directory. Now list the contents:

```
prompt> cd MM
prompt> make
prompt> ls
Makefile      Makefile-blas  MM1.c  MM2.c  MM2.s  MM4.c  MM5-blas.c  scripts
Makefile~     Makefile-blas~ MM1.c~  MM2.c~ MM3.c  MM4.c~ MM5-blas.c~
```

**Running them on Linux** Now run the programs as follows:

```
prompt> ./MM1.x86
prompt> ./MM2.x86
prompt> ./MM3.x86
prompt> ./MM4.x86
```

The goal of this exercise is for you to figure out why the four versions run at different speeds.

**Setting a different problem size** The default problem size is 2176 (=2048+128). You can set a different problem size, e.g. type:

```
prompt> make clean
prompt> make MYFLAGS=-DSZ=2048 x86
```

(the `make clean` deletes the old binaries).

**Using SimpleScalar** SimpleScalar (<http://www.simplescalar.com>) is a processor microarchitecture simulator which we will be using in this and other exercises.

---

<sup>1</sup>a list of all the DoC lab machines is here: <https://www.doc.ic.ac.uk/csg/facilities/lab/workstations>.

**Compiling the programs** SimpleScalar is very slow, so we will rebuild for problem size 192:

```
prompt> make clean
prompt> make MYFLAGS=-DSZ=192
```

And try using the SimpleScalar simulator:

```
prompt> /homes/phjk/simplesim/sim-outorder ./MM1.ss
```

The first thing this does is lists the parameters of the architecture being simulated – all of these details can be changed using command-line arguments. Then it runs the application, and outputs details of how the processor microarchitecture and memory hierarchy were used<sup>2</sup>.

Sim-outorder simulates the full microarchitecture of quite a complicated CPU. If we only want to model the memory hierarchy we can use sim-cache, which is much faster:

```
prompt> /homes/phjk/simplesim/sim-cache ./MM1.ss
```

**Using a script to run a sequence of simulations** A script “varycachesize” has been provided for running a sequence of experiments over a range of cache sizes of direct-mapped cache. For example:

```
prompt> ./scripts/varycachesize ./MM1.ss 64 8192
```

(it is worth finding a fast, unshared machine for this). The output format is comma-separated to be easy to plot; the miss rate is column 4:

```
prompt> ./scripts/varycachesize ./MM1.ss 256 8192 > varycachesize_MM1_256_8192.csv
```

You can edit the script to study the effect of other cache parameters. You can edit it to use sim-outorder, so you can study other microarchitecture features.

## What to do

Plot a graph that shows how the L1 data cache miss rate (as measured with SimpleScalar) for MM 1,2 and 3 varies with cache size. The range of 64–8192 is interesting.

Returning to the performance of the four variants on the x86 machine, explain why the four versions have different performance.

*Paul H.J. Kelly, Luigi Nardi, Anton Lokhmotov, Carlo Bertolli and Graham Markall, Imperial College, 2022*

---

<sup>2</sup>For problem size 192 this takes about 30 seconds