332 Advanced Computer Architecture

Exercise 4: Matrix Multiply Exploration

Background

This exercise concerns a familiar computational kernel: matrix multiplication. Here's a simple implementation:

```
/*
 * mm
 *
 * Multiply A by B leaving the result in C.
 * A is assumed to be an l x m matrix, B an m x n matrix.
 * The result matrix C is of course l x n.
 * The result matrix is assumed to be initialised to zero.
 */
void mm1(double A[SZ][SZ], B[SZ][SZ], C[SZ][SZ])
{
  int i, j, k;
  double r;

  for (i = 0; i < SZ; i++)
    for (j = 0; j < SZ; j++)
      for (k = 0; k < SZ; 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 shown above, an interchanged "ikj" variant, and a more sophisticated, "tiled" version. You can find source code in

```
/homes/phjk/ToyPrograms/ACA07/MM
```

Getting started

The files provided Using one[1] of the DoC CSG Linux systems (level 2, Huxley), make your own copy of the ACA07/MM directory tree by executing the following Linux commands:

```
prompt> mkdir ACA07
prompt> cd ACA07
prompt> cp -r /homes/phjk/ToyPrograms/ACA07/MM ./
```

[1] Use ssh. See https://www.doc.ic.ac.uk/csg/computers for list of machines available. Choose a Pentium IV or Athlon machine first. Don't choose an Apple unless you want to modify the Makefile yourself.

(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> ls
Makefile  MM1.c  MM2.c  MM3.c  MM4.c  scripts  TaskGraph
```

The files MM1.c, MM2.c etc contain the various different versions of the matrix multiply that we're going to play with. The TaskGraph directory is to explore on your own if you have time.

Compiling the programs Compile the programs using make:

```
prompt> make all
```

This uses the GNU C compiler to build the binary for the linux x86 machines we're using. It also builds binaries for the SimpleScalar simulator.

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.

Compiling with a smaller problem size To use the simulators you need to work with a smaller problem size (for MM3 and MM4 this must be a multiple of the block size, which is 64 by default). The problem size is determined by the `#define` "SZ", which you can set using make (if you don't, it defaults to 576 (=512+64))[2]. Type:

```
prompt> make clean
prompt> make MYFLAGS=-DSZ=256
```

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

Using the cachegrind simulator Cachegrind is a cache simulator that works on x86 binaries. Try:

```
prompt> /homes/phjk/valgrind/bin/valgrind --tool=cachegrind --branch-sim=yes ./MM1.x86
```

[2]Using cachegrind (see below) with a problem size of 576 took more than 200 seconds when I tried it, ca. 1 MFLOPS on a 2GHz P4.

This produces a summary of the total memory accesses, and the cache miss rates for instructions and data: ca.154M instructions fetched, 34M data reads, 17M writes, L1D read miss rate 6%, L1D write miss rate 0.1%.

Note that the cache requirements of the application are quite different with a smaller problem size... With the default problem size, for MM1, the results are quite different:

```
==6187== I   refs:      1,544,592,375
==6187== I1  misses:              817
==6187== L2i misses:              814
==6187== I1  miss rate:          0.00%
==6187== L2i miss rate:          0.00%
==6187==
==6187== D   refs:        577,037,006  (383,256,512 rd   + 193,780,494 wr)
==6187== D1  misses:      194,591,990  (194,467,353 rd   +     124,637 wr)
==6187== L2d misses:          237,436  (    112,812 rd   +     124,624 wr)
==6187== D1  miss rate:          33.7% (       50.7%     +         0.0%  )
==6187== L2d miss rate:           0.0% (        0.0%     +         0.0%  )
==6187==
==6187== L2 refs:         194,592,807  (194,468,170 rd   +     124,637 wr)
==6187== L2 misses:           238,250  (    113,626 rd   +     124,624 wr)
==6187== L2 miss rate:            0.0% (        0.0%     +         0.0%  )
==6187==
==6187== Branches:        192,122,440  (192,122,252 cond +         188 ind)
==6187== Mispredicts:         337,203  (    337,165 cond +          38 ind)
==6187== Mispred rate:            0.1% (        0.1%     +        20.2%   )
```

kcachegrind: a GUI tool When you run cachegrind, it produces a log file in your current directory, called something like cachegrind.out.6187 (the number is the process id, and appears in the output above as well). kcachegrind is a user interface for exploring a program's profile data. Try:

```
prompt> kcachegrind cachegrind.out.6187
```

(you will have to adjust this: use the pid from your valgrind run instead of "6187").

Modelling different cache configurations You can give cachegrind parameters to control the cache being simulated - by default it chooses parameters simular to the machine you are using. For further details see

```
http://valgrind.org/docs/manual/cg-manual.html
```

For example:

```
/homes/phjk/valgrind/bin/valgrind --tool=cachegrind --branch-sim=yes --D1=1024,1,64 ./MM1.x86
```

Using simplescalar Now, rebuild for problem size 128:

```
prompt> make clean
prompt> make MYFLAGS=-DSZ=128
```

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[3].

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 256 8192
```

(it is worth finding a fast, unshared machine for this). The output format has been designed to be easy to use with "gnuplot"; the miss rate is column 4:

```
prompt> ./scripts/varycachesize ./MM1.ss 256 8192 > varycachesize_MM1_256_8192.dat
prompt> gnuplot
gnuplot> plot "varycachesize_MM1_256_8192.dat" using 4 with linespoints
```

Edit the script to study the effect of other cache parameters. 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 1,2,3 and 4 varies with cache size and explain the result. The range of 64-8192 is interesting.

Note:

You can plot several datasets at once as follows:

```
plot "varycachesize_MM1_256_8192.dat" using 4 with lp,\
     "varycachesize_MM2_256_8192.dat" using 4 with lp
```

What to do next

You might enjoy having a look at the TaskGraph subdirectory:

[3]For problem size 256 this took my 2GHz P4 almost 700 seconds

```
prompt> cd TaskGraph
prompt> make
prompt> ./mm
```

This version uses the TaskGraph Library, a prototype tool developed in my research group. You can write a program that builds the original loop nest, then programmatically applies transformations, before running the generated code. You could extend this to systematically search for the optimum version of the code.

You might also like to look at some of the other simplescalar simulators; see

```
http://www.simplescalar.com/v4test.html
http://www.simplescalar.com/
```

This includes microarchitecture simulators for ARM, Alpha and PowerPC, together with energy models, microarchitecture extensions and multiprocessor simulations.

Paul Kelly, Imperial College, 2008