332 Advanced Computer Architecture

Exercise 4: 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
 *
 * (The matrices can actually be submatrices of larger
 * matrices of dimensions M x M, where M > N, e.g. N=576,
 * M=1024, to study effects of associativity conflicts.)
 */
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 shown above, an interchanged "ikj" variant, and a more sophisticated, "tiled" version. You can find source code at

```
/vol/spo/ACA10/MM
```

Getting started

This exercise can mostly be done using ssh to connect to a CSG linux machine remotely (the only thing that won't work is gnuplot, unless you have an X server).

The files provided Using one of the DoC CSG Linux systems (level 2, Huxley), make your own copy of the ACA10/MM directory tree by executing the following Linux commands:

```
prompt> mkdir ACA10
```

```
prompt> cd ACA10
prompt> cp -r /vol/spo/ACA10/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> ls
atlas  Makefile  MM1.cloog  MM2.cloog  MM3.cloog MMa.c MM.c  MM.h
scripts  TaskGraph
```

The files `MM1.cloog`, `MM2.cloog` and `MM3.cloog` contain specifications for a tool called CLooG, which is used to generate the different C versions of the matrix multiply that we're going to play with: `MM1.c`, `MM2.c` and `MM3.c`.

Compiling the programs Compile the programs using make:

```
prompt> make all
```

¿From the `*.cloog` files CLooG generates the `*.c` files, which GCC then compiles to the binaries for the Linux x86 and for the SimpleScalar simulator.

Running them on Linux Now run the programs as follows:

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

The goal of this exercise is for you to figure out why the three versions run at different speeds. Try the experiment on a Pentium IV machine[1]; are the results different?

Setting a different problem size The default problem size is 576 (=512+64). You can set a different problem size, e.g. type:

```
prompt> make clean
prompt> make MYFLAGS=-DN=2048
```

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

You can also play with memory allocation, e.g. type:

```
prompt> make clean
prompt> make MYFLAGS="-DN=256 -DM=1024"
```

(This will allocate 1024×1024 matrices but actually perform 256×256 matrix multiplication. Why the results are different from the default case when $N = M$?)

[1] Use `ssh`. See `https://www.doc.ic.ac.uk/csg/computersforlistofmachines` available.

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

Set a smaller problem size,[2] e.g.

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

Try:

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

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").

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

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:

```
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=-DN=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 and 3 varies with cache size and explain the result. The range of 64–8192 is interesting.

[3]For problem size 128 this took my 1.6GHz Core2Duo almost 30 seconds

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 also like to look at some of the other SimpleScalar simulators; see

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

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

Finally, you might want to take a closer look at the CLooG and CLAn tools; see:

`http://www.cloog.org`
`http://http://www.lri.fr/~bastoul/development/clan/`

Paul H.J. Kelly, Anton Lokhmotov, and Graham Markall, Imperial College, 2010