332 Advanced Computer Architecture

Exercise 4 (Assessed): the Matrix Multiply Challenge

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];
}
```

Part A: Systematic investigation (20 marks)

Part A must be completed *individually*, deadline Monday 26th November. Submit your work at the DoC Student General Office in a suitable labelled folder.

In this part of the exercise, your job is to characterise the behaviour of two variants of the Jacobi program: a straightforward version based on the code shown above, and a more sophisticated, "tiled" version. You can find source code in

`/homes/phjk/ToyPrograms/MMChallenge/src`

and precompiled Linux/x86 binaries in

`/homes/phjk/ToyPrograms/MMChallenge/gcc`

You will also need binaries compiled for the SimpleScalar simulator; see

`/homes/phjk/ToyPrograms/MMChallenge/simplescalar/`

Before you start, you need to set some environment variables as follows:

```
prompt> setenv SSsim /homes/phjk/simplescalar/simplesim-2.0/
```

(if you use bash instead of tcsh you need to type "SSsim=/homes/phjk/simplescalar/simplesim-2.0/" instead).

Getting started

Before starting work on the assessed part of the exercise, you need to familiarise yourself with the test application you have been given, and the tools you have been provided with.

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

```
prompt> cp -r /homes/phjk/ToyPrograms/MMChallenge ./
```

(The "./" above is the destination of the copy - your current working directory). You should now have a copy of the MMChallenge directory. Now change directory into it, and list the contents:

```
prompt> cd MMChallenge
prompt> ls
gcc  msVisualC  simplescalar  src  vectorc
```

The "src" directory contains the source code for the MM application. The "gcc" directory contains the code you can run on the Linux systems. The "simplescalar" directory contains code you can run on the SimpleScalar simulator. The "msVisualC" and "vectorc" directories contain code which will only run under Windows and is provided for your curiosity only.

The MM application Now try running the MM application:

```
prompt> cd gcc
prompt> ./MM-gcc-O3
Initialisation: 10000
8570000 us      31.3    Size: 512 Expt 1 - unoptimised ijk (needed for checking)
2940000 us      91.3    Size: 512 Expt 2, unoptimised ikj
4190000 us      64.1    Size: 512 Blocksize 4 Expt 3 blocked kk,jj,i,k,j
2880000 us      93.2    Size: 512 Blocksize 8 Expt 3 blocked kk,jj,i,k,j
2360000 us      113.7   Size: 512 Blocksize 16 Expt 3 blocked kk,jj,i,k,j
2130000 us      126.0   Size: 512 Blocksize 32 Expt 3 blocked kk,jj,i,k,j
1950000 us      137.7   Size: 512 Blocksize 64 Expt 3 blocked kk,jj,i,k,j
2020000 us      132.9   Size: 512 Blocksize 128 Expt 3 blocked kk,jj,i,k,j
3690000 us      72.7    Size: 512 Blocksize 256 Expt 3 blocked kk,jj,i,k,j
1110000 us      241.8   Size: 512 block: 64 x 64 Expt 4 blocked kk,jj,i,k,j (compile-time)
2160000 us      124.3   Size: 512 block: 64 x 64 Expt 5, blocked kk,jj,i,k,j (compile-time)
1240000 us      216.5   Size: 512 block: 64 x 64 subblock 32 x 32 Expt 6, blocked kk,jj,i,k,j (compile-tim
```

This might take a little while, 10-60 seconds. Each line corresponds to one run of the matrix multiply loop. Column 1 gives the execution time in microseconds.

Column 2 gives the calculated execution speed in MegaFLOPs. The results of each run are checked against Experiment 1. You can select specific experiments using command-line arguments:

```
prompt> ./MM-gcc-O3 4 -nocheck
Initialisation: 10000
1110000 us      241.8   Size: 512 block: 64 x 64 Expt 4 blocked kk,jj,i,k,j (compile-time)
```

The default matrix size is 512×512. To change it you need to recompile. For example, type:

```
prompt> ./makeme -DSZ=256
prompt> ./MM-gcc-O3-DSZ=256
```

Note that the "makeme" script incorporates the compiler flags in the output file name, to avoid confusion. The default data type is "double" (64-bit floating point). You can change this in the same way:

```
prompt> ./makeme -DSZ=128 -DFLOATTYPE=float
prompt> ./MM-gcc-O3-DSZ=128-DFLOATTYPE=float
```

Now go and look at the source code.

Using the SimpleScalar simulator To study the effect of different architectural design choices, we can use a software simulation. The SimpleScalar suite provides a very flexible set of tools for such investigations. Of course, programs run rather slowly. Try this:

```
prompt> cd simplescalar
prompt> ./makeme -DSZ=128
prompt> $SSsim/sim-cache -cache:dl1 dl1:256:32:1:l MM-ss-O3-DSZ=128 1 -nocheck
```

This runs the "sim-cache" simulator. It tells it to use the default configuration except that the level-1 data cache ("dl1") should have 256 sets, with cache blocks of 32 bytes, 1-way associative (ie direct-mapped), with LRU ("l") replacement.

The SimpleScalar suite includes two main simulators: "sim-outorder", which simulates the complete microarchitecture, and "sim-cache", which simulates only the memory system (so runs somewhat faster). Both are used in the same way.

Sim-cache and sim-outorder produce a lot of output. The first half details the machine configuration being simulated. The second half gives various statistics from the execution.

Note that I told the simulator to run "MM-ss-O3-DSZ=128 1 -nocheck"; just Experiment 1, and without the extra work to check the results. This is so we only see statistics from the code of interest.

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. For example:

```
prompt> ./varycachesize MM-ss-O3-DSZ\=64 4
Size: 64 [2048 bytes] Config: dl1:64:32:1:l Hits: 1401311 Misses: 470887 Missrate: 0.2515
Size: 128 [4096 bytes] Config: dl1:128:32:1:l Hits: 1446141 Misses: 426098 Missrate: 0.2276
Size: 256 [8192 bytes] Config: dl1:256:32:1:l Hits: 1468392 Misses: 403693 Missrate: 0.2156
Size: 512 [16384 bytes] Config: dl1:512:32:1:l Hits: 1767997 Misses: 104199 Missrate: 0.0557
Size: 1024 [32768 bytes] Config: dl1:1024:32:1:l Hits: 1851057 Misses: 21143 Missrate: 0.0113
Size: 2048 [65536 bytes] Config: dl1:2048:32:1:l Hits: 1869147 Misses: 3052 Missrate: 0.0016
Size: 4096 [131072 bytes] Config: dl1:4096:32:1:l Hits: 1869664 Misses: 2533 Missrate: 0.0014
```

The output format has been designed to be easy to use with "gnuplot"; the miss rate is column 12:

```
prompt> ./varycachesize MM-ss-O3-DSZ\=64 4 >varycachesize-ss-O3-DSZ=64-4.dat
prompt> gnuplot
gnuplot> plot "varycachesize-ss-O3-DSZ=64-4.dat" using 12
```

See also "varycacheblocksize".

What to do

1. Plot a graph that shows how the L1 data cache miss rate (as measured with SimpleScalar) for experiments 1 and 4 varies with problem size (up to 1024), and explain the result. Note that the problem sizes for experiment 4 must be divisible by the blocking factor, 64.
2. Use Experiment 7 (see source code) to find the optimum blocking factor for your machine, for the default problem size of 512. Include in your answer details of the processor you are using ("cat /proc/cpuinfo").
3. Use Experiment 7 to find the optimum blocking factor for your machine for a range of problem sizes 508,509,510..515. Plot a graph showing how the optimum blocking factor and resulting performance varies with problem size. Comment on your results.
4. Use sim-outorder to study the effect of varying the register update unit (RUU) size (eg "sim-outorder -ruu:size 64"). Try a range of RUU sizes (eg 2–128 in powers of two). Plot a graph showing the resulting simulated performance for two variants, Experiment 1 and Experiment 4. Don't forget the "-nocheck" flag. Do this for two problem sizes: 64 and 128. Comment on your results.

In each case, you should write a few sentences commenting on your results; you should suggest at least one possible explanation for the behaviour you see.

What to hand in Hand in a *concise* report which details the experimental setup, presents the graphs and your explanation for the resulting behaviour, and explains how you arrived at your conclusions. Please do not write more than four pages.

Part B: All-out performance (10 marks)

Part B can be done in groups of up to three, deadline Monday 10 December. Submit your work at the DoC Student General Office in a suitable labelled folder.

Your job for this part of the assessed exercise is to implement the matrix multiply specified above as fast as you possibly can. You are invited to start from the implementation provided.

Competition rules

1. You can choose to enter in one of the following categories:
   - Any of the PCs available in the DoC labs. Stick to the same machine configuration for all your experiments. Detail in your report exactly what configuration you have.
   - Any other computer, subject to agreement. PalmPilots, DSPs, wrist-watches, fridges all considered.
2. Parallelism: you can choose a parallel implementation if you wish, subject to agreement.
3. Problem size: for PCs, report performance on a problem size of 512. If your execution time is less than 0.5 seconds, you should consider studying a larger problem size.

Assessment

Marks, and in suitable cases, prizes, will be awarded for

- Getting good performance
- Explaining the approaches you tried, and how you evaluated how well they work
- Drawing conclusions from the experience

You should produce a compact report in the style of an academic paper for presentation at an international conference such as Supercomputing (www.sc2000.org) (though perhaps a little shorter).

Tools and techniques

You may find it useful to find out about:

- PAPI - a library providing access under Linux to your processor's hardware performance counters. Available on CSG lab machines (try "man PAPI"; works better on Intel than AMD).
- gnuplot, a simple and useful graph plotting package for Linux
- Tiling/blocking; Chapter 5 of the lecture notes, available online at `http://www.doc.ic.ac.uk/~phjk/AdvancedCompArchitecture/Lectures/`
- ATLAS and PhiPAC `http://acts.nersc.gov/atlas/main.html`
- AC/DC (Architecture-cognizant Divide-and-Conquer): `http://www.cs.ucsd.edu/~kgatlin/papers.html`

- The TaskGraph library; see /homes/phjk/OldStudentProjects/AlastairHoughton/TaskGraph/samples/
- Iterative Compilation in a Non-Linear Optimisation Space. PACT'98 F. Bodin, T. Kisuki, P.M.W. Knijnenburg, M.F.P. O'Boyle and E. Rohou: http://citeseer.nj.nec.com/bodin98iterative.html
- CodePlay's VectorC compiler, available for free download from www.codeplay.com.
- Intel's optimising compiler for Linux, also available for free download (though there are problems installing it on lab machines).
- Julian Seward's "cacheprof" utility.

How to get started Here are some hints:

- Have a look at the assembly code generated by the compiler (use the "`-S`" flag, "`gcc -O3 -S file.c`"). Similarly with SimpleScalar,

  ```
  /homes/phjk/simplescalar/bin/sslittle-na-sstrix-gcc -O3 -S file.c
  ```

- Would unrolling help? Read "man gcc".
- Where are the likely pipeline stalls, missed instruction issue opportunities? How much scope for improvement is possible through improved instruction scheduling?
- Where/when do cache misses occur?
- Can you improve the spatial locality?
- Can you improve the temporal locality?
- Can you use the multimedia instruction set extensions (you might have to use `floats` instead of `doubles` - if so, let me know.
- Do you need to optimise for several levels in the memory hierarchy?

How to finish The main criterion for assessment is this: you should have a reasonably sensible hypothesis for how to improve performance, and you should evaluate your hypothesis in a systematic way, using experiments together, if possible, with analysis.

What to hand in Hand in a concise report which

- Explains what hardware and software you used,
- What hypothesis (or hypotheses) you investigated,
- How you evaluated what the potential advantage could be,
- How you explored the effectiveness of the approach experimentally
- What conclusions can you draw from your work
- If you worked in a group, indicate who was responsible for what.

Please do not write more than six pages.

Paul Kelly, Imperial College, 2001