332 Advanced Computer Architecture

Exercise 5 (ASSESSED): instruction-level-parallelism in sparse matrix-vector multiply

This is the first of two equally-weighted assessed coursework exercises. Hand your solution into the Dept of Computing Student General Office (Huxley 345) by 5pm on Wednesday February 18th.

Background (as for exercise 4)

A sparse matrix is one with a large proportion of zero elements. Many important algorithms involve multiplying a sparse matrix by a vector, $y = Ax$. This key computational "kernel" is the focus of the main laboratory project for this course.

Applications include:

- Solving partial differential equations on unstructured (tetrahedronal) meshes - for example to simulate fluid flow
- Calculating the eigenvalues of the web's hyperlink connectivity graph, for example in order to compute Google's "Pagerank"
- Solving a Markov process - derive the probability of a system being in a particular state, given a matrix of state transition probabilities.

Dense (that is not sparse) matrix-vector multiplication looks like this:

```
double A[A_rows][A_cols];
double x[A_rows], y[A_rows];

for ( i=0; i<A_rows; i++ ) {
   sum = 0.0;
   for ( k=1; k<=A_cols; k++ )
     sum += A[i][k]*x[k];
   y[i] = sum;
}
```

To make this sparse we need to choose a storage scheme. A popular choice is "compressed row":

```
struct MatrixElement {
  long col;
  double val;
};
/* no of non-zeroes in each row */
int SA_entries[A_rows];

/* array of pointers to list of non-zeroes in each row */
MatrixElement** SA = (struct MatrixElement**)
                     malloc(sizeof(struct MatrixElement)*A_rows);
```

```
/* When SA_entries is known we can allocate just enough memory for data */
for(i=0; i<A_rows; i++) {
  SA[i] = (struct MatrixElement*)
          malloc(sizeof(struct MatrixElement)*A_entries[i]);
}
```

The idea here is that for each row, we keep an array of the non-zero matrix elements, pointed to by `SA[i]`. For each element `SA[i][j]` we need its value, and its column number. So, if `A[i][j]` is the nth non-zero in row `i`, its value will be stored at `SA[i][n].val`, and `SA[i][n].col` will be `j`.

Now the matrix-vector multiply can be written as:

```
for(i=0; i<A_rows; i++) {
  sum = 0;
  for(j=0; j<SA_entries[i]; j++) {
    sum += ((SA[i][j]).val)*x[(SA[i][j]).col];
  }
  y[i] = sum;
}
```

Compiling and running Csteady (slightly different from Ex4)

Copy the source code directory tree to your own directory:

```
cd
cp -r /homes/phjk/ToyPrograms/C/smv-V2 ./
```

Now run some setup scripts:

```
cd smv-V2
source SOURCEME.csh
./scripts/MakeMatrices.sh 1
```

This creates a directory /tmp/USERNAME/spmv/ and builds some sample matrices there. You should compile the application using the following commands:

```
pushd solver/csteady/
make
popd
```

Now you can run the program:

```
./solver/csteady/csteady inputs/courier1/
```

This reads input from the directory for problem size 1, and writes its output to a file called "STEADY" in the same directory. This is problem size 1 - very tiny. Problem size 3 is more substantial. [1]

[1] You can make much larger problem sizes using `/scripts/MakeMatrices.sh 4` (410MBytes), or `./scripts/MakeMatrices.sh 5` (more than 1400MBytes).

Running the csteady benchmark under the Simplescalar simulator

The makefile in solver/csteady also builds a binary for execution using the SimpleScalar simulator. You can execute it by typing:

```
sim-outorder ./solver/csteady/csteady-ss inputs/courier1/
```

What to do

Your job is to study the effect of various architectural features on the performance of the Csteady application running under the SimpleScalar simulator:

1. Use sim-outorder's "-ruu:size" flag to vary the RUU size between 2 and 256 (only powers of two are valid). Plot a graph showing your results. Explain what you see. (see "Tools and Tips" below for how to automate this).
2. Examine the other sim-outorder parameters which determine the CPU's instruction issue rate (leave the cache and memory system parameters unchanged). Where is the bottleneck (when running Csteady) in the default simulated architecture? Justify your answer.
3. What is the maximum number of instructions per cycle that can be achieved on this benchmark, assuming the cache and memory system is kept the same? Explore the benefits of increasing the RUU, Load-Store Queue, the number of arithmetic units and memory ports.

Write your results up in a short report (less than four pages including graphs and discussion).

Tools and tips

Use the fastest machine you can find. On a 3GHz Pentium 4 each simulation run (for problem size 1) takes about a minute. Use the "top" command to make sure you're not sharing it.

The most important output from the simulator is "sim_cycle" - the total number of cycles to complete the run. It's often also useful to look at "sim_IPC", the instructions per cycle - provided you always execute the same number of instructions. The time taken to perform the simulation "sim_elapsed_time" simply tells you how the simulator took.

Other outputs from the simulator can be helpful in guiding your search - eg "ruu_full", the proportion of cycles when the RUU is full.

Scripting the experiments

To do this you need to write a shell script. You might find the following Bash script useful:

```
#!/bin/sh -f
for ((x=2; x <= 128 ; x *= 2))
```

```
do
  echo -n "ruu-size" $x " "
  sim-outorder -ruu:size $x \
     ./solver/csteady/csteady-ss inputs/courier1/ \
     2>&1 | grep sim_IPC
done
```

The output is delivered to the file "results". You can find this script in

```
scripts/ruu-loop.sh
```

See also "lsq-loop.sh".

Plotting a graph

Try using the gnuplot program. Run the script above, and save the output in a file "table". Type "gnuplot". Then, at its prompt type:

```
set logscale x 2
plot [][] 'table' using 1:4 with linespoints
```

To save the plot as a postscript file, try:

```
set term postscript eps
set output "psfile.ps"
plot [][] 'table' using 1:4 with linespoints
```

Try "help postscript", "help plot" etc for further details.

Examining the assembler code

Compile the application using the "-S" flag.

```
cd solver/csteady
~phjk/simplescalar/bin/gcc -O3 -S csteady.c
```

The assembler code generated by the compiler is delivered in "csteady.s".

Experimenting with compiler options

The compiler accepts many parameters to control optimisation. Try "man gcc" to read about them. For example, "-funroll-loops" encourages the compiler to unroll loops:

```
~phjk/simplescalar/bin/gcc -O3 -funroll-loops csteady.c
```

Does this improve the performance of the simulated machine? What happens to the IPC?

Paul Kelly, Imperial College London, 2004