CS/ECE 347, Spring 1996
Lab Assignment 1:
Using Your Knowledge of MIPS to Optimize C Code

Assigned: Tue., Jan. 30
Due: Thu., Feb. 15

1. Introduction

Many programs performing scientific computations spent the vast majority of their CPU time performing algebraic computations on matrices and vectors. Quite often, most of the entries in the matrices being manipulated are zeros; in such situations, sparse matrix representations are used. In this assignment, you will use your knowledge of MIPS assembly language to optimizing C functions that multiply matrices by vectors, where the matrices are represented either densely or sparsely.

To get started, copy everything in

```
/afs/cs.cmu.edu/academic/class/15347-s96/public/labs/la1
```

to your personal directory.

Note that you *must* use a computer with a MIPS processor for this assignment!

At the beginning of your write-up, state clearly what type of computer you are using (brand and model number).

2. Initial Matrix and Vector Representation

In the file `csr.h`, you will find declarations for a structure called `csr_rec`. This structure can be used for representing square matrices in either of two formats: *dense* or *compressed sparse row*. The first field of the structure, `type`, specifies which of the two formats the matrix is using. The `nrow` field specifies how many rows (and columns) the matrix has; the `val` field points to an array containing actual matrix values. These are the only fields used by dense matrices — the `val` field in a dense matrix contains `nrow` * `nrow` values, with the element in row r row and column c stored at an offset of `nrow` $* r + c$.

In sparse matrices, the `nentries` field records how many non-zero entries are in the matrix; the array to which `val` points is of length `nentries`. The `rstart` field points to an array of `nrow` integers; the nth of these integers specifies where in `val` the first non-zero element of the nth row lies. Finally, each value stored in `val` has a corresponding entry in `cindex` that specifies in which column the value actually belongs.

In our implementation, a vector is represented simply as an array containing the vector's element values. We assume that the matrix by which we're multiplying the vector has as many columns as the vector has rows, so we're often not going to bother about passing around the vector's size separately.

3. Processor latency for multiplication instructions

Suppose we multiply a matrix A with a vector v. Each element z_i of the resulting vector v may be expressed as an inner product of v with the corresponding row of A:

$$z_i = a_{i1} * x_1 + a_{i2} * x_2 + \ldots + a_{in} * x_n$$

Consequently, in most implementations of matrix-by-vector multiplication, sequences similar to the following sequence show up in the instruction stream:

```
            R1 <- R2 * R3
                  .
                  .
      [a few other instructions here]
                  .
                  .
            R4 <- R4 + R1
```

Unfortunately, multiplication instructions take a long time to complete — more than the one or two cycles associated with other instructions like integer addition or conditional branches. This means that unless there are many instructions between the multiplication instruction and the addition instruction that uses its result, the processor will have to *stall* when it reaches the addition instruction until the required multiplication result has been produced.

Problem 1

With the help of the timing functions and programs you used for the first recitation assignment, determine the latency (in clock cycles) of (1) double-precision floating-point multiplication instructions, and (2) double-precision addition instructions. You may write your latency-tickling functions in either C or MIPS assembly.

Turn in:

- All source code
- An annotated disassembly of the innermost loop of your test routines (the one inside of which multiplications and/or additions are performed).
- A clear but brief exposition of what numbers you used, how you got them, and how you used them in order to arrive at your answer.

Remember: the timing functions we're using aren't very accurate. You should average over many timing trials, and report not only the means of these trials but also the numbers of trials and the minimum and maximum values.

Hint: you may want to try timing several simple assembly language functions with different numbers of instructions between the instruction of interest and the first instruction that uses its result.

4. Baseline Matrix By Vector Multiplication Implementation

In `csr.c` you will find a baseline implementation of two matrix-by-vector multiplication functions — one for dense matrices (`dense_mult_smpl`), and one for sparse matrices (`csr_mult_smpl`). Another

function (`csr_mult`) can be given a matrix of either form; it simply checks the type of the matrix and calls the appropriate specialized function.

You will also find a set of "dummy" functions identical in structure to the set of functions above. These dummy functions involve the same overhead as their corresponding "real" functions, but they don't actually do any useful computation. By subtracting the times these functions take from the times taken by the "real" functions, you can get a better idea for how much time the "real" functions are spending on useful computation rather than unavoidable overhead.

Finally, in `mdata.c`, you will find `gen_matrix`, a function that randomly generates a square matrix with a specified number of rows and a specified number of non-zero entries, as well as `rvec`, a function for randomly generating vectors of a specified length. There are also several other useful functions for transforming one type of matrix to another, printing out matrices generated with `gen_matrix`, and so forth; you should look these over and keep them in mind, as they may come in handy later on.

Problem 2

Part A

Use `gcc` to compile all of the `.c` files into .o files with standard optimization (`-O2`).

Using the usual timing functions, write a C program to measure the average time required for `csr_mult` to multiply matrices and vectors generated by `gen_matrix` and `rvec` with the following sizes:

- 200 by 200; 38,000 non-zero entries (95% density)
- 200 by 200; 4,000 non-zero entries (10% density)
- 20 by 20; 380 non-zero entries (95% density)
- 20 by 20; 40 non-zero entries (10% density)
- 3 by 3; 7 non-zero entries (78% density)

Report timing results for both dense and sparse matrices for each of the matrix sizes above. Also report timing measurements for "dummy" function calls on the same matrices; from these measurements, deduce what fraction of the time spent by the "real" functions is spent on "overhead."

Report not only the average times, but also the number of trials over which you averaged, as well as the minimum and maximum observed times.

Now, using the clock frequency estimated by `freq`, convert all of your average times into:

- average number of clock cycles required for entire multiplication
- average number of clock cycles required per nonzero matrix element

Each timing result should use at least 10 different randomly generated matrices, and you should multiply each of these matrices a sufficient number of times to ensure that the measured average number of cycles per nonzero element is accurate to within one cycle. You should only have one matrix in memory at any given point in time, and you may want to "prime" it by multiplying it once before performing your timing operations on subsequent multiplications. (Why?)

Turn in all source code for the test program.

Part B

Use `gdb` to disassemble `csr_mult_smpl`; examine the code. Estimate roughly how many clock cycles you would have expected this function to take on a 200 by 200 sparse matrix with 38,000 non-zero entries, assuming all accesses to memory have the usual one-cycle latency. Explain how you arrived at your estimate.

How does your estimate compare with your results from Part A? Any guesses as to what might account for the difference?

5. Representation trade-offs

Sparse matrices are much faster to use than dense matrices when there are, in fact, lots of non-zero entries. On the other hand, in cases where most of the entries are non-zero, working with dense representations is often faster. Choosing a representation appropriate to the particular problem on which one is working can be important.

Problem 3

Fixing the matrix dimensions at 200 by 200, find the density (percentage of non-zero elements) at which the dense matrix representation is faster for vector multiplication than the sparse representation, to within a density of 2%.

Use `gen_matrix` to generate your test matrices, and use the same timing methodologies you used in Problem 2A. Explain what numbers you used and how you used them in order to arrive at your conclusion.

6. Fine-tuning the loops

As you may have gathered from Recitation Assignment 2, it sometimes makes a significant difference whether you use array indexing or direct pointer manipulation in your C code — code that uses direct pointer manipulation often compiles down to significantly faster machine code. Our baseline functions to multiply dense and sparse matrices are just such cases.

Problem 4

Write new versions of `csr_mult_smpl` and `dense_mult_smpl` optimized for speed by using pointer manipulations in place of array indexing. Do so by iteratively modifying the original C code and examining the resulting MIPS code (compiled by `gcc` with the `-O2` option). For this exercise, you may not change the representation of either type of matrix, and your code should possess roughly the same structure as the original functions.

Turn in:

- A copy of your C code for both functions
- Annotated MIPS assembly versions of both functions
- Timing results for the same tests you ran in Problem 2A (but now using your optimized code instead of the baseline implementations).

Note that part of your grade on this assignment will depend upon how much faster your code is than the baseline implementation.

7. Further possibilities for optimization

There are still quite a few ways in which the multiplication code might be sped up, some of which involve changing the csr_rec data structure itself. Some possibilities (by no means exhaustive, and not even necessarily good ideas!):

- Use byte offsets in `cindex` instead of word offsets to get rid of those annoying little `sll` instructions
- Changing the representation so that both column *and row* numbers for the values in `val` are kept in arrays.
- Overhead reduction: weed out that extra function call and switch statement by embedding a pointer to the correct multiplication procedure right in the matrix data structure
- Loop unrolling: cut down on branch overhead and multiplication latency by doing multiple multiplications within one loop iteration

Problem 5

Part A

Throw any and all valid optimizations you like at the problem. Rules of the game:

- C code only — no hand-tweaked assembly routines.
- All compilations should be done with `gcc` using the `-O2` option.
- No making assumptions about the values of the matrix's elements. Even if all the non-zero values in `gen_matrix`'s matrices are either -1 or 1, your code had better work for arbitrary matrices.
- You may modify the representation used for either or both matrix types, *subject to the following restriction*: the "density structure" of the matrix must be represented in memory independently of the data itself. In particular, you may not change `cindex` so that it contains pointers to the absolute memory locations of the data (instead of offsets as it currently contains). Rationale: many important domains require the use of lots of matrices with identical density structures.
- You must use a top-level function or macro that can accept both sparse and dense matrices for multiplication.
- The answers your code produces must be correct.

Naturally, you will probably have to come up with your own versions of `get_ele`, `gen_matrix`, and/or a few other functions if you change the representation. Keep the original representation around, however, and provide functions for translating back and forth between old-style dense matrices and each of your new representations. Use `gen_dense_matrix` to generate random old-style matrices, then use your functions to translate the matrices into your representations. Make sure that your code is functioning correctly

by translating your multiplication results back into old-style dense matrices and then checking that these matrices are identical to the results produced by the original multiplication functions.

Turn in:

- A brief discussion of what optimization techniques you tried, and why you think they did or didn't work
- All modified or added source and header code (including your code to test for correctness), well commented
- An annotated disassembly of your optimized functions for multiplying (1) dense matrices and (2) sparse matrices.
- Timing results for all cases covered in Problem 2A. You don't need to run "dummy" functions, however. You may not factor out the "overhead" time from your reported results.

Note that part of your grade on this assignment will depend upon how much faster your final code runs than the original implementation.

Part B

Roughly how many 200 by 200 sparse matrices with 10% density would the computer have to multiply before the amount of CPU time saved by your optimizations in Part A exceeded the amount of time you spent on Part A?