CS 740, Fall 1997
Assignment 3:
Optimizations and Computer Arithmetic

Assigned: Mon., Oct. 6
Due: Fri., Oct. 17

Policy

You may work in a group of up to 3 people in solving the problems for this assignment. You should turn in a single report for your entire group, identifying all of the group members.

Logistics

Any clarifications and revisions to the assignment will be posted on the class bboard and Web page. There is no electronic handin for this assignment. Turn in the two parts as separate documents, so that they can be graded independently.

1. Code Performance Optimization

For this part of the assignment, you will tune the performance of the sparse matrix code discussed in class on September 22 for a particular machine platform. The goal is to combine your knowledge of the algorithm, the machine, and the capabilities of your compiler to achieve optimum performance.

The particular routine you will optimize, computing the product of a sparse matrix times a dense vector, is called in the inner loop by many linear system solvers. Linear solvers, in turn, are called in the inner loop of many nonlinear system solvers, which in turn are in the inner loops of programs for simulating physical systems ranging from electronic circuits to earthquakes. Thus, any efficiency gain in the multiplication routine can have a major impact on the application performance.

1.1. Platform Choice

A first step is to choose the platform you will optimize your code for. It should be a machine with high floating-point performance, and for which there is good documentation available regarding its internal architecture. The following are recommended choices.

P6 Either Pentium II or PentiumPro systems.

PPC604 E.g., high-end Macintosh or IBM workstations

Alpha DEC Alpha. Preferrably 21164.

R10000 MIPS R10000

R5000 E.g., the SGI Indy's in the Wean Hall 5th floor cluster.

As a default, the R5000's would make a good choice, and they are publicly accessible. These machines are not heavily used in the mornings. If you would like to try a different platform, check with me (Randy) first.

The assignment information page of the class WWW page has more discussion on platforms and sources of literature for the recommended choices.

The default compiler for the Makefile is the system-supplied compiler CC. This is generally better tuned for the target machine than GCC.

1.2. Getting Started

1. Retrieve the file `csr.tar` from the directory:

   `/afs/cs.cmu.edu/academic/class/15740-f97/public/asst/asst3`

2. Set up and connect to a directory for your work.
3. Run `tar xf csr.tar`
4. Run `make`

This will compile the existing versions of matrix multiplication and time them over a benchmark test. The benchmark consists of 5 different 20×20 random matrices, each with a density of 25%, i.e., each containing 100 nonzero elements. You can vary these parameters with command-line arguments to the test procedure `matrix-test`. This might prove useful for measuring the overhead caused by the the outer loop of the multiplication routine.

1.3. Your Task

You'll notice in the file `csr.h` that there are four types of matrix representations defined:

`UNSCALED` : Standard, compressed-row format, as described in class

`SCALED` : Like `UNSCALED`, except that the elements in the array `cindex` are scaled by `sizeof(double)` to simplify array indexing.

`DENSE` : A standard, row-major representation of the matrix.

`NEW_REP` : The version that you will create

Right now, all operations for matrices of type `NEW_REP` operate identically to `UNSCALED`.

Your task will be to devise a new representation of the matrix and to write a new multiplication routine that achieves optimal performance for your chosen platform. You may choose any representation you like, although it must remain compatible with the existing calling convention for the routine `csr_mult`. That is, `csr_mult(m, x, z)` computes the product of matrix `m` with dense vector `x` and stores the results in `z`.

As a justification for these rules, typical applications call the multiplication routine by a single matrix for many different vector arguments. Thus, the overhead of creating some optimized data structure to represent the matrix gets amortized over many uses of that data structure. On the other hand, the vector argument typically changes with each call.

You will need to add your new multiplication routine to the file `csr.c`, and you will need to change the data generation routines in `mdata.c` for your new representation. You should not change any of the testing code. In particular, you should allow it to check that your new routine generates the correct result, and you should report your performance for the standard parameters it uses.

1.4. Ideas for Optimization

It would be best if you could work entirely in C. Try to write the code in such a way that your compiler can take care of low-level optimizations. You might find your compiler does better if you use array indexing rather than pointer arithmetic.

It might make sense to do some preprocessing based on the vector argument before you begin the multiplication, since each element will be used on average 10 times. Note that any such preprocessing cost must be included in overall running time.

This routine is a good candidate for optimizations such as loop unrolling and software pipelining. You might want to make optimizations to the matrix representation that will lower the overhead of these transformations.

1.5. Desired Outcome

You should be able to get your code running at nearly the optimal speed for your platform. This "optimum" should be based on the resource capabilities of your machine, e.g., the latencies and throughputs of the floating point units and the load and store units. You should be able to argue why the inner loop of your routine takes the number of cycles it is taking, and why this is the best achievable. You should also minimize sources of overhead such as the outer loop.

1.6. Hand In

Document the following:

- A description of your platform and the performance of the supplied multipliation routines.
- The representation you implemented
- A discussion of the optimizations you attempted and their performance impact.
- The final code for the multiplication routine
- An analysis of your routine's performance for your chosen target, and its degree of optimality.

2. Computing Floating-Point Reciprocals

In this problem, we consider the task of computing the reciprocal of an n-bit floating-point number a. To keep things simple, we will assume that a has been normalized so that $1/2 < a < 1$. Thus, we can view a as a fixed-point binary number with leading bits 0.1, followed by n additional bits. For example, if $n = 8$, then one possible value for a would be 0.110110101. Note that our limitation on the range of a also implies that $1 < 1/a < 2$, so that $1/a$ begins with 1., and is followed by n additional bits. Continuing with our previous example, $1/a = 1.00101010$ with remainder $r = 0.0000000110100111$. (You can check this by doing long division in binary.)

The result of a floating-point division operation does not include the remainder. Instead, the remainder is discarded, and the result is rounded. The IEEE Standard 754 for Binary Floating-Point Arithmetic specifies that the error in the result of a correctly rounded operation cannot exceed half the gap between the adjacent floating-point numbers on either side of the operation's ideal (i.e., unrounded) result. If the result is exactly half way between two adjacent floating-point numbers, the result is rounded to the one with last bit 0. Using fixed-point numbers to illustrate this rule, suppose that each number is stored to 4 bits of precision following the fixed point, and that the result of an operation is 0.10111. This result lies exactly half way between the fixed-point numbers 0.1011 and 0.1100, and so must be rounded to 0.1100.

2.1. Part A (easy):

The SRT algorithm described in class is essentially a fast way to implement long division. Suppose that after computing $1/a$ to n-bits of precision using the standard "grade-school" long-division algorithm (in binary), the remainder is exactly r.

1. Ignoring leading 0's, what is the maximum number of bits required to represent r?
2. Explain in at most three sentences how to correctly round the result $1/a$.

2.2. Part B:

An alternative to using long division, especially in the absence of dedicated hardware for floating-point division, is to use an iterative method such as Newton iteration. The reciprocal of a can be computed using the recurrence

$$x_{i+1} = 2x_i - ax_i^2,$$

where x_i is the estimate of $1/a$ at the end of the ith iteration. For our choice of a, an initial guess of $x_0 = 1.1$ (i.e., $3/2$) will guarantee convergence, provided that all operations are performed with infinite precision.

1. Using the MIPS instruction set, write a short segment of code that implements one iteration. Try to make your code as fast as possible. You may use addition and multiplication operations, but no other floating-point arithmetic operations. You should use double-precision floating-point registers, but you may ignore the issues of rounding and finite precision.
2. In class we saw that the number of bits of precision in our estimate of $1/a$ doubles after each iteration, provided that all of the operations are performed with infinite precision. Suppose instead that in the ith iteration, each operation is performed to only 2^{2^i} bits of precision, with rounding following the IEEE Standard 754. Show that the number of bits of precision is still improved by a factor of C in each iteration, for some fixed constant C, where $1 < C < 2$. (You may use table look-up to obtain a better initial estimate x_0, if that helps.)
3. Suppose that you have performed enough iterations to produce an estimate x of $1/a$ to n-bits of precision, along with an exact remainder r, which might be negative. Explain how to correctly round the result x.
4. Suppose that the remainder is not exact. Explain the difficulty with correctly rounding x.
5. The remainder r is given by the formula $1 = a \cdot x + r$. Explain the difficulty with computing the remainder exactly using addition and multiplication operations.
6. Some architectures, such as the PowerPC, provide a "Fused" Multiply-Add instruction for evaluating an expression such as $a \cdot b + c$. The instruction is called Fused because there is only one rounding error, at the end. In other words, the instruction does NOT have the same effect as first rounding $a \cdot b$, then adding the result to c and rounding again. Show that a Fused Multiply-Add instruction can be used to calculate the remainder r exactly.

7. **Extra Credit.** Explain how to compute the remainder exactly, or how to correctly round the final estimate x without using a Fused Multiply-Add instruction.