Editor: Dianne P. O'Leary, oleary@cs.umd.edu

COMPUTER MEMORY AND ARITHMETIC: A LOOK UNDER THE HOOD

By Dianne P. O'Leary

YOU HAVE MANY PLACES IN WHICH TO STORE YOUR DATA. YOU MIGHT KEEP YOUR IDENTIFICATION AND CREDIT CARDS IN YOUR WALLET, WHERE YOU CAN GET TO THEM quickly. However, space is limited, so you can't keep *all* your important information there. Instead, you might carry current papers for work or school in a backpack or briefcase and store older papers in your desk or filing cabinet. Any papers you don't think you'll need but are afraid to throw out might be stored in an attic, basement, or storage locker.

We can think of your wallet, backpack, desk, and attic as a *hierarchy* of storage spaces. The small ones give you fast access to data that you often need; the larger ones give slower access but more space. For the same reasons, computers also have a hierarchy of storage units. Memory management systems try to store information that you will soon need in a unit that gives fast access. This means that large vectors and arrays are broken up and moved piece by piece as needed. You can write a correct computer program without ever knowing about memory management, but attention to memory management allows you to consistently write programs that don't have excessive memory delays.

In this homework assignment, we'll consider a model of computer memory organization. We'll hide some detail but give enough information to let us make decisions about how to organize our computations for efficiency. We'll use mathematical modeling to estimate a typical computer's memory parameters, and then we'll see how important these parameters are relative to the speed of floating-point arithmetic.

A Motivating Example

Suppose we have an $m \times n$ matrix $\boldsymbol{A}$ and an $n \times 1$ vector $\mathbf{x}$, and we want to form $\mathbf{y} = \boldsymbol{A}\mathbf{x}$. In Matlab, we just write `A * x`, but let's consider how this might be implemented.

We can define the vector $\mathbf{y}$ with *inner products* (also called *dot products*) between rows of $\boldsymbol{A}$ and $\mathbf{x}$: for $i = 1, \ldots, m$,

$y_i = \boldsymbol{A}(i, :)$ * $\mathbf{x}$.

We can also define $\boldsymbol{A}\mathbf{x}$ in a column-oriented way: $\boldsymbol{A}\mathbf{x} = x(1) * \boldsymbol{A}(:, 1) + x(2) * \boldsymbol{A}(:, 2) + \ldots + x(n) * \boldsymbol{A}(:, n)$. This scheme is based on an operation called `saxpy`, which is an abbreviation for $a\mathbf{x} + \mathbf{y}$. We work left to right through our expression, taking a scalar times a vector and adding it to a previously accumulated vector: initialize $\mathbf{y}$ to zero and then compute $\mathbf{y} = \mathbf{y} + x(j) * \boldsymbol{A}(:, j)$ for $j = 1, \ldots, n$.

Both algorithms have the same round-off properties and take almost the same number of numeric operations: mn multiplications and $m(n - 1)$ or mn additions. But, surprisingly, the time taken by the two algorithms is quite different.

PROBLEM 1.

Program the two algorithms in Matlab. Time them for a random matrix $\boldsymbol{A}$ and a random vector $\mathbf{x}$ for $m = n = 1{,}024$, and then verify that they yield the same product $\boldsymbol{A}\mathbf{x}$.

TOOLS

John Hennessy and David Patterson give a good, detailed description of memory hierarchies in Chapter 5 of their book.[1]

In Matlab, the underlying matrix decomposition software is drawn from the Lapack Fortran suite.[2] It's based on a set of basic linear algebra subroutines (BLAS), which are provided by hardware manufacturers to optimize operations such as inner product, `saxpy`, matrix-matrix multiplication, and so on. The Lapack routines implement stable algorithms and provide high performance on a variety of hardware.

References

1. J.L. Hennessy and D.A. Patterson, *Computer Architecture: A Quantitative Approach*, 2nd ed., Morgan Kaufmann, 1996.
2. E. Anderson et al., *LAPACK Users' Guide*, 3rd ed., SIAM Press, 1999.

Authorized licensed use limited to: to IEEExplore provided by Virginia Tech Libraries. Downloaded on October 24, 2008 at 09:48 from IEEE Xplore. Restrictions apply.

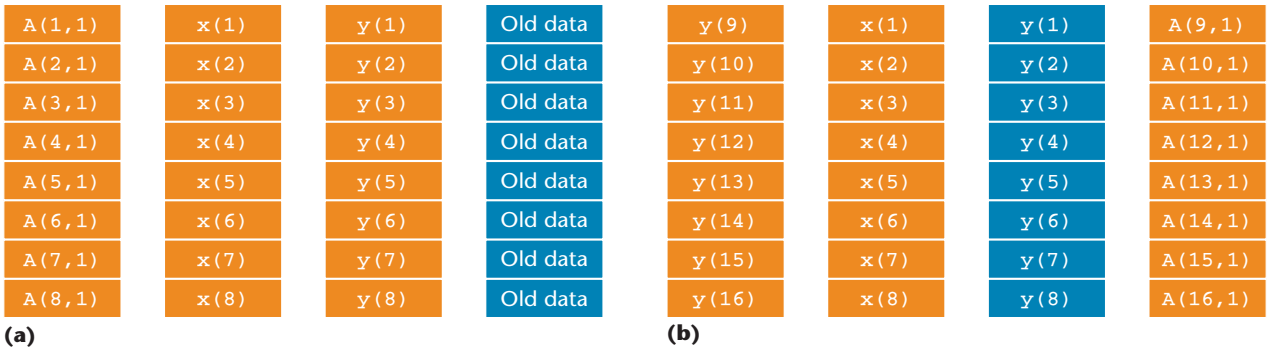

Figure 1. Memory hierarchy. State of a cache memory of four blocks, eight words each, during two stages of the matrix-vector product algorithm: (a) the `saxpy` implementation and (b) when blocks leave cache memory.

Your results should have indicated that the second algorithm is much faster than the first when m and n are large. The speed difference is due to memory management; Matlab stores matrices column by column, so if we want a fast implementation, we must use this fact in our algorithm's design.

Memory Management

The computer memory hierarchy includes registers, cache, main memory, and disk. Arithmetic and logical operations are performed on the contents of registers, and the other storage units act as temporary storage for data on its way to or from the registers. It's as if whenever you need to change some data in your attic, you move it first to your desk, then your backpack, and then your wallet, make the correction, and then move it back through your wallet, backpack, and desk, finally storing it back in the attic.

Figure 1 gives a small illustration of a memory hierarchy. We'll consider a one-level cache, although most machines have a hierarchy of cache units. Let's see how information moves between main memory and cache. Suppose that m = 128 and n = 32, and suppose for ease of counting that the first element in each matrix and vector lies in the first element of some page of main memory. The matrix elements are stored in the order `A(1,1),...,A(128,1),A(1,2),...,A(128,2),...,A(1,32),...,A(128,32)`. Cache memory is loaded by *block* (also called a *cache line*); in this example, this means eight elements at a time. So in the `saxpy` implementation, in which we successively add $x_j * A(:, j)$ to **y**, the computer first loads `A(1,1),..., A(8,1)` into one block of the cache, `x(1),..., x(8)` into a second block (because it needs the value of `x(1)`), and lets `y(1),..., y(8)` occupy a third block (see Figure 1a). After `x(1)*A(1:8,1)` is added into `y(1:8)`, we then need `A(9,1),..., A(16,1)` and `y(9),..., y(16)`. This would fill five blocks of cache, though, and we only have four, so we have to write over an old block—after assuring that any updated values are changed in the main memory. In our case, the old `y`-block or `A`-block disappears from cache (see Figure 1b).

Moving five blocks from main memory lets us do 16 of our 128 * 32 multiplications. We continue the count in Problem 2.

PROBLEM 2.

Count the number of blocks that move from main memory into cache for each of the two matrix-vector multiplication algorithms with m = 128 and n = 32. If a cache block must be written over, choose the least-recently-used block. How does your answer change if matrices are stored row-by-row (as in C, C++, or Java), rather than column-by-column (as in Matlab or Fortran)?

Clearly, memory management matters in matrix multiplication! It's certainly faster to use the algorithm that moves fewer blocks into cache, but will it really make a difference?

Determining Hardware Parameters

How much memory management truly matters depends on memory parameters such as

- b, the number of blocks that cache memory can hold;
- ℓ, the number of double-precision words in a block;
- α, the time needed to access a double-precision word in cache (nanoseconds); and
- μ, the extra time needed for access if the word isn't already in cache (also known as the cache-miss penalty [nanoseconds]).

To estimate the memory parameters, we can run a program that constructs a long vector **z** of length m and then steps through it, incrementing some elements. When we step through every element, we're almost always accessing a value that exists in cache. If we step through elements $\mathbf{z}(1)$, $\mathbf{z}(1 + s)$, $\mathbf{z}(1 + 2s)$, …, where the stride s is bigger than the block size ℓ, then we always get a cache-miss penalty. By varying s, we can estimate the block size, and by cycling through the computation several times, we can estimate the cache's size. Consider the following code fragment:

Authorized licensed use limited to: to IEEExplore provided by Virginia Tech Libraries. Downloaded on October 24, 2008 at 09:48 from IEEE Xplore. Restrictions apply.

```
steps = 0;
i = 1;
do
{ z(i);
  steps = steps + 1;
  i = i + s;
  if (i > m)
    i = 1;
  end
}
while (steps < naccess)
```

The loop makes `naccess` accesses to the array, where `naccess` is a suitably large number. If we time the loop, subtract off the loop overhead (estimated by timing a similar loop with the statement `z(i)` omitted), and divide the resulting time by `naccess`, we can estimate the average time for one access.

We see how this works in Problem 3.

Problem 3.

Suppose our cache memory has parameters $b = 4$, $\ell = 8$, $\alpha = 1$ ns, and $\mu = 16$ ns. Assume that we used the least-recently used strategy for replacing blocks in cache and that we set `naccess=256` in the previously listed code fragment. Consider the following table of estimated times per access in nanoseconds and show how each entry is derived:

s	$m = 16$	$m = 32$	$m = 64$	$m = 128$
1	1.125	1.250	3.000	3.000
2	1.125	1.250	5.000	5.000
4	1.125	1.250	9.000	9.000
8	1.125	1.250	17.000	17.000
16	1.063	1.125	1.250	17.000

If we work in a high-performance "compiled" language such as Fortran or a C-variant, we can use our timings of code fragments to estimate the cache-miss penalty. In "interpreted" Matlab (or even compiled Matlab), overhead masks the penalty.

Whenever we time a program, though, there are many sources of uncertainty:

- *Other processes are running*. Even if you're running on a laptop on which you're the only user, the operating system (Windows, Linux, and so on) still does many other tasks, such as refreshing the screen, updating the clock, and tracking the curser. Most systems have two timers, one that gives the elapsed time (for example, `tic, toc`) and one that tries to capture the time used by this process alone (for example, `cputime`).
- *There is uncertainty in the timer, so the data you collect are noisy*. Most timers give trash unless they're timing intervals that are at least a millisecond (in fact, they're much better at intervals near one second). Therefore, the loop you're timing should do as many operations as possible, but not so many that interruptions by other active processes contaminate the elapsed time.
- *The time for arithmetic operations often depends on the values of the operand*. Dividing by a power of 2 is usually much faster than dividing by other numbers, for example, and adding zero is usually faster than other additions.
- *The computer uses pipelining*. This occurs on many levels, but the fundamental idea is that the execution of each instruction we give the computer is partially overlapped with other instructions, so it's difficult to assign a cost to a single instruction.
- *Compilers optimize our code*. A compiler might recognize, for example, that `z(i)` isn't changed by our code fragment and thus will remove that statement from the loop.
- *Cache blocks might be prefetched*. Programs often access data in order, so computers might predict that the next sequential memory block should be loaded into cache while you're operating on the current one.

Because of factors like this, real data isn't as clean as that in Problem 3. If we run a program like the one considered in Problem 3 on my Sun workstation, we get the results shown in Table 1. Let's try to estimate the cache parameters from that data.

Problem 4.

Estimate the cache parameters from Table 1's data.

Extra Problem 1

Write a program in a high-performance language such as C or Fortran to estimate the cache size, the cache's block size, the time to access a value in cache, and the cache-miss penalty. Run it on your favorite computer. Find the manufacturer's claims for at least some of these parameters and determine whether your estimates agree or disagree, and why.

Authorized licensed use limited to: to IEEExplore provided by Virginia Tech Libraries. Downloaded on October 24, 2008 at 09:48 from IEEE Xplore. Restrictions apply.

Table 1. Average memory access times (nanoseconds) on my Sun workstation for various lengths *m* of (single-precision) arrays and various strides s.*

$\log_2 s$	$\log_2 m$=										
	10	11	12	13	14	15	16	17	18	19	20
0	2	3	4	5	5	7	7	8	7	8	9
1	4	4	3	4	4	8	10	9	8	10	11
2	7	7	6	7	8	15	18	17	18	20	22
3	6	7	7	7	8	19	21	21	21	21	25
4	13	12	12	10	11	21	22	21	22	23	23
5	14	15	15	14	15	25	26	26	27	27	26
6	10	12	10	10	9	20	19	17	17	18	19
7	11	11	10	9	10	19	20	18	17	19	19
8	1	11	11	10	9	18	18	17	18	17	18
9	1	0	10	11	10	20	18	18	19	17	18
10	3	0	1	10	11	21	18	18	17	17	19
11	3	2	0	11	30	57	58	58	60	59	62
12	1	3	2	0	10	51	60	60	61	58	62
13	3	3	3	3	0	10	48	61	60	60	61
14	2	3	3	2	4	–1	10	50	59	61	60
15	3	2	2	2	3	2	0	11	49	60	59
16	2	3	4	3	4	3	4	0	9	49	61
17	3	2	2	2	2	3	3	4	0	9	51
18	3	3	3	4	3	2	3	4	1	0	10
19	3	3	2	3	2	3	2	4	3	2	0

**The negative entry that occurred when overhead was subtracted off indicates the data's uncertainty.*

Speed of Computer Arithmetic

The order in which we access the elements of a matrix affects the time, but is it a significant effect? Let's time some arithmetic operations to see whether it matters.

Problem 5.

Determine the time your computer requires for floating-point operations (addition, subtraction, multiplication, division, and square-root) and integer operations (addition, subtraction, multiplication, and division). One way is to look up the peak speed claimed by the manufacturer; an alternative is to write a timing loop in a high-performance language as in Extra Problem 1. Average the time over enough operations to get an accurate estimate, and then estimate the variance in your measurements. Compare these times with the memory access times obtained either from Extra Problem 1 or from the manufacturer's data.

What we've discovered about the time for memory access and floating-point operations gives us the information we need to achieve speeds close to the manufacturer's peak-performance claims when doing matrix operations. The next problem challenges you to write a matrix multiplication program to do this.

Extra Problem 2

Use the information you gathered about your machine's memory access properties to write the best program you can for doing matrix-matrix multiplication on your computer. Use a high-performance language. The inputs to the function are two matrices $\boldsymbol{A}$ of dimension $m \times n$ and $\boldsymbol{B}$ of dimension $n \times p$, along with n, m, and p. The output of the function is the $m \times p$ matrix $\boldsymbol{F} = \boldsymbol{AB}$. The program should order the computations to minimize the number of cache misses.

An alternative to writing our own fast algorithms for basic matrix operations is to use the ones provided in the basic linear algebra subroutine (BLAS) implementation discussed in the "Tools" sidebar. In Matlab, we access the one for a matrix-vector product by typing `A*x`. For other matrix tasks, we now know to keep our Matlab algorithms column-oriented whenever possible.

Acknowledgments

I'm grateful to the staff in the Technical Support Department at MathWorks for discussions about the sources of overhead in Matlab's interpreted and compiled instructions.

Authorized licensed use limited to: to IEEExplore provided by Virginia Tech Libraries. Downloaded on October 24, 2008 at 09:48 from IEEE Xplore. Restrictions apply.

Partial Solution to Last Issue's Homework Assignment

UPDATING AND DOWNDATING MATRIX FACTORIZATIONS: A CHANGE IN PLANS

By Dianne P. O'Leary

IN THE LAST ISSUE'S HOMEWORK ASSIGNMENT, WE CONSIDERED NUMERICAL METHODS THAT MAKE IT EASIER TO REANALYZE A DESIGN AFTER SMALL CHANGES ARE MADE.

For definiteness, we focused on truss design, but the same principles apply to any linear model.

PROBLEM 1.

a. Suppose we want to change columns 6 and 7 in our matrix A. Express the new matrix as $A - ZV^T$, where Z and V have dimension $n \times 2$.

b. Suppose we want to change both column 6 and row 4 of A. Find Z and V so that our new matrix is $A - ZV^T$.

Answer:

a. Set the columns of Z to be the differences between the old columns and the new ones, and set the columns of V to be the 6th and 7th unit vectors.

b. The first column of Z can be the difference between the old column 6 and the new one; the second can be the 4th unit vector. The first column of V is then the 6th unit vector, and the second is the difference between the old row 4 and the new one.

PROBLEM 2.

a. Assume that A and $A - ZV^T$ are both nonsingular. Show that

$$(A - ZV^T)^{-1} = A^{-1} + A^{-1}Z(I - V^TA^{-1}Z)^{-1}V^TA^{-1}$$

by verifying that the product of this matrix with $A - ZV^T$ is the identity matrix I. This is called the *Sherman-Morrison-Woodbury formula*.

b. Suppose we have an LU decomposition of A. Assume that Z and V are $n \times k$ and $k << n$. Show that we can use this decomposition and the Sherman-Morrison-Woodbury formula to solve the linear system $(A - ZV^T)\mathbf{f} = \ell$ without forming any matrix inverses. (If A is dense, then we perform $O(n^2)$ operations using Sherman-Morrison-Woodbury, rather than the $O(n^3)$ operations needed to solve the linear system from scratch.) Hint: Remember that we can compute $A^{-1}\mathbf{y}$ by solving a linear system using forward- and back-substitution with the factors L and U.

Answer:

a. This is verified by direct computation.

b. We use several facts to get an algorithm that is $O(kn^2)$ instead of $O(n^3)$ for dense matrices:

- $\mathbf{x} = (A - ZV^T)^{-1}\mathbf{b} = (A^{-1} + A^{-1}Z(I - V^TA^{-1}Z)^{-1}V^TA^{-1})\mathbf{b}$.
- Forming A^{-1} from LU takes $O(n^3)$ operations, but forming $A^{-1}\mathbf{b}$ as $U\backslash(L\backslash\mathbf{b})$ uses forward- and back-substitution and just takes $O(n^2)$.
- $(I - V^TA^{-1}Z)$ is only $k \times k$, so factoring it is cheap: $O(k^3)$. However, forming it is more expensive: $O(kn^2)$.
- Matrix multiplication is associative.

Using Matlab notation, once we've formed `[L,U]=lu(A)`, the resulting algorithm is

```
y = U \ (L \ b);
Zh = U \ (L \ Z);
t = (eye(k) - V'*Zh) \ (V'*y);
x = y + Zh*t;
```

PROBLEM 3.

Implement the Sherman-Morrison-Woodbury algorithm from Problem 2b. Debug it by factoring the matrix in the first truss example and then changing the model to the second truss example.

Answer:

See `sherman_mw.m` on the Web site (www.computer.org/cise/homework/).

PROBLEM 4.

For n taken to be various numbers between 10 and 1,000, generate a random $n \times n$ matrix A. Find the number of updates k_0 that makes the cost of the Sherman-Morrison-Woodbury method comparable to computing $A\backslash\ell$. Plot k_0 as a function of n.

Authorized licensed use limited to: to IEEExplore provided by Virginia Tech Libraries. Downloaded on October 24, 2008 at 09:48 from IEEE Xplore. Restrictions apply.

Answer:

The solution is given in `problem4.m` on the Web site (www.computer.org/cise/homework/); Figure A plots the results. In this experiment (run on a Sun UltraSPARC-III with clock speed 750 MHz and Matlab 6), Sherman-Morrison-Woodbury was faster for $n \geq 40$ when the update's rank was less than 0.25 n.

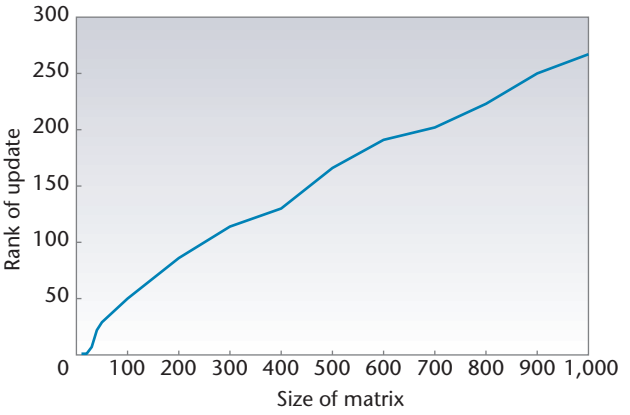

Figure A. Results of Problem 4. Sherman-Morrison-Woodbury was faster for $n \geq 40$ when the rank of the update was less than 0.25 n.

PROBLEM 5.

a. Given a vector $\mathbf{z} \neq \mathbf{0}$ of dimension 2×1, find the Givens rotation $\boldsymbol{G}$ so that $\boldsymbol{G}\mathbf{z} = x\mathbf{e}_1$, where $x = \|\mathbf{z}\|$ and $\mathbf{e}_1$ is the vector with a 1 in the first position and zeros elsewhere.

b. We will use the notation $\boldsymbol{G}_{ij}$ to denote an $m \times m$ identity matrix with its ith and jth rows modified to include the Givens rotation: for example, if $m = 6$, then

$$\boldsymbol{G}_{25} = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & c & 0 & 0 & s & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & s & 0 & 0 & -c & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix}.$$

Multiplication of a vector by this matrix leaves all but rows 2 and 5 of the vector unchanged. Show that we can finish our QR decomposition by (left) multiplying the $\boldsymbol{R}$ matrix in Equation 3 first by $\boldsymbol{G}_{16}$, then by $\boldsymbol{G}_{26}$, and finally by $\boldsymbol{G}_{36}$, where the angle defining each of these matrices is suitably chosen. To preserve the equality, we multiply the $\boldsymbol{Q}$-matrix by $\boldsymbol{G}_{16}{}^T\boldsymbol{G}_{26}{}^T\boldsymbol{G}_{36}{}^T$ on the right, and we have the updated factorization.

Answer:

a. $$\boldsymbol{G}\mathbf{z} = \begin{bmatrix} cz_1 + sz_2 \\ sz_1 - cz_2 \end{bmatrix} = x\mathbf{e}_1 .$$

Multiplying the first equation by c, the second by s, and adding yields

$$(c^2 + s^2)z_1 = cx,$$

so

$$c = z_1/x.$$

Similarly, we can determine that

$$s = z_2/x.$$

Because $c^2 + s^2 = 1$, we conclude that

$$z_1{}^2 + z_2{}^2 = x^2,$$

so

$$c = \frac{z_1}{\sqrt{z_1^2 + z_2^2}}$$

$$s = \frac{z_2}{\sqrt{z_1^2 + z_2^2}}.$$

b. The first rotation matrix is chosen to zero a_{61}. The second zeros the resulting entry in row 6, column 2, and the final one zeros row 6, column 3.

PROBLEM 6.

Write a Matlab function that updates a QR decomposition of a matrix $\boldsymbol{A}$ when a single column is changed. Apply it to the truss examples in Problem 3.

Answer:

See `problem6.m` and `qrcolchange.m` on the Web site (www.computer.org/cise/homework/).

Dianne P. O'Leary is a professor of computer science and a faculty member in the Institute for Advanced Computer Studies and the Applied Mathematics Program at the University of Maryland. She has a BS in mathematics from Purdue University and a PhD in computer science from Stanford. O'Leary is a member of SIAM, the ACM, and AWM. Contact her at oleary@cs.umd.edu; www.cs.umd.edu/users/oleary/.

Authorized licensed use limited to: to IEEExplore provided by Virginia Tech Libraries. Downloaded on October 24, 2008 at 09:48 from IEEE Xplore. Restrictions apply.