Week 4: Monday, Sep 14

Timing matrix multiplies

At the end of the last lecture, we (briefly) discussed the format that MATLAB uses for *sparse* matrices. When most of the entries in the matrix are zero, this storage format is fairly compact, and the built-in MATLAB routines for sparse matrix multiply are fairly efficient. But what about if a sparse matrix format is used to represent a dense matrix (one in which most of the entries are nonzero)? Compared to sparse matrix multiplication, dense matrix multiplication requires less memory traffic (since there is no need to look up indices), and what memory traffic it requires is more regular. But how much difference is there? The only way I know to tell is by timing.

There are a couple points to consider in our experiments:

1. The granularity of the MATLAB `tic/toc` timer is limited, so any experiment that runs for too little time will yield inaccurate times. We get around this by insisting that our experiments execute a certain amount of arithmetic (about two million flops in our case) or run ten times, whichever takes longer.

2. After data has been moved into cache on the first access, subsequent accesses will be faster — unless something else replaces the cached data. This means that the second time a routine runs on the same input data, it might run more quickly. In our timing experiment, we will "warm up" the cache with one untimed multiplication before we do the actual timing.

A test harness that runs the timing experiments is listed as an appendix to these notes. You may wish to look at it when thinking about the first problem on the homework. The results, plotted in Figure 1, show that for a wide range of n values, the dense matrix multiply runs about six times faster than the sparse multiply on the same (dense) matrix.

Gaussian elimination

For the next few lectures, we will be exploring the solution of linear systems. Our main tool will be the factorization $A = PLU$, where P is a permuta-

tion, L is a lower triangular matrix, and U is an upper triangular matrix. As we will see, the Gaussian elimination algorithm learned in a first linear algebra class implicitly computes this decomposition; but by thinking about the decomposition explicitly, we can come up with other organizations for the computation.

Recall the following basic scheme for solving the linear system $Ax = b$, usually the first one learned in an introductory linear algebra class:

1. Form the augmented matrix $\begin{bmatrix} A & b \end{bmatrix}$
2. Subtract multiples of the first row from the subsequent rows in order to zero out all but the first entry in the first column.
3. For each remaining column j up to column $n-1$, subtract multiples of row j to zero out all entries below the diagonal element $\hat{A}_{jj}$.
4. Once the augmented system has been put into the form $\begin{bmatrix} U & \hat{b} \end{bmatrix}$, solve the triangular system $Ux = \hat{b}$ by back-substitution.

Putting the first three steps together, we have the following MATLAB code:

```
% Form the augmented matrix
Ab = [A; b];

for j = 1:n-1
  % Subtract multiples of row j to zero out Ab(i,j) for i > j
  for i = j+1:n
    Ab(i,:) = Ab(i,:) - Ab(i,j)/Ab(j,j) * Ab(j,:);
  end
end
```

Notice that the inner loop of this matrix is subtracting multiples of the *same* jth row from every subsequent row. We can write a matrix consisting of multiples of one row very easily: it's an outer product. With this idea in hand, we have the following equivalent algorithm:

```
% Form the augmented matrix
Ab = [A; b];

for j = 1:n-1
```

```
  % Subtract multiples of row j to zero out Ab(i,j) for i > j
  % i.e. subtract a rank one matrix from the trailing submatrix.
  Ab(j+1:n,:) = Ab(j+1:n,:) - Ab(j+1:n,j)/Ab(j,j)*Ab(j,:);
end
```

So far, we have thought about the transformations done at each step in an algorithmic way: take a submatrix and do some operation to it, where the operation is defined by a little program. But we could also think of these operations in terms of linear transformations. For example, we can write the first step of elimination for a 3-by-3 problem as a matrix multiply:

$$M_1 \begin{bmatrix} A & b \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ -a_{21}/a_{11} & 1 & 0 \\ -a_{31}/a_{11} & 0 & 1 \end{bmatrix} \begin{bmatrix} a_{11} & a_{12} & a_{13} & b_1 \\ a_{21} & a_{22} & a_{23} & b_2 \\ a_{31} & a_{32} & a_{33} & b_3 \end{bmatrix} = \begin{bmatrix} a_{11} & a_{12} & a_{13} & b_1 \\ 0 & \tilde{a}_{22} & \tilde{a}_{23} & \tilde{b}_2 \\ 0 & \tilde{a}_{32} & \tilde{a}_{33} & \tilde{b}_3 \end{bmatrix}.$$

The second step is

$$M_2 M_1 \begin{bmatrix} A & b \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & -\tilde{a}_{32}/\tilde{a}_{22} & 1 \end{bmatrix} \begin{bmatrix} a_{11} & a_{12} & a_{13} & b_1 \\ 0 & \tilde{a}_{22} & \tilde{a}_{23} & \tilde{b}_2 \\ 0 & \tilde{a}_{32} & \tilde{a}_{33} & \tilde{b}_3 \end{bmatrix} = \begin{bmatrix} a_{11} & a_{12} & a_{13} & b_1 \\ 0 & \tilde{a}_{22} & \tilde{a}_{23} & \tilde{b}_2 \\ 0 & 0 & \hat{a}_{33} & \hat{b}_3 \end{bmatrix}.$$

The matrices M_1 and M_2 are *unit lower triangular* matrices: that is, they have nonzero entries only on or below the main diagonal, and the entries on the main diagonal are all one. The products and inverses of unit lower triangular matrices are also lower triangular, so we can write $L^{-1} = M_1 M_2$, where L^{-1} is unit lower triangular. That is, we have computed

$$L^{-1} \begin{bmatrix} A & b \end{bmatrix} = \begin{bmatrix} U & L^{-1}b \end{bmatrix},$$

i.e. $A = LU$ and $A^{-1}b = U^{-1}(L^{-1}b)$.

Appendix: Matvec timing code and results

```
n = 10:10:1000;

for kn = 1:length(n);

  % It's useful to adapt the number of trials to the problem size,
  % since we want to have enough experiments to overcome timer
```

```
  % resolution issues.
  %
  ntrials = min(10,ceil(5000000/n(kn)^2));

  A  = rand(n(kn));
  As = sparse(A);
  v  = rand(n(kn),1);

  u = A*v;  % Cache warm-up
  tic;
  for trial = 1:ntrials
    u = A*v;
  end
  timeA(kn) = toc/ntrials;

  u = As*v; % Cache warm-up
  tic;
  for trial = 1:ntrials
    u = As*v;
  end
  timeAs(kn) = toc/ntrials;

end

figure(1);
plot(n,timeA, n,timeAs);
ylabel('s');
xlabel('n');
legend('Dense matvec', 'Sparse matvec');

% We're going to also report the time in GFlop/s (billions of
% flops per second).  This gives us a consistent way to compare
% the performance of the two routines across problem sizes.
%
figure(2);
plot(n,2e-9*n.^2./timeA, n,2e-9*n.^2./timeAs);
ylabel('MFlop/s');
xlabel('n');
```

```
legend('Dense matvec', 'Sparse matvec');
```

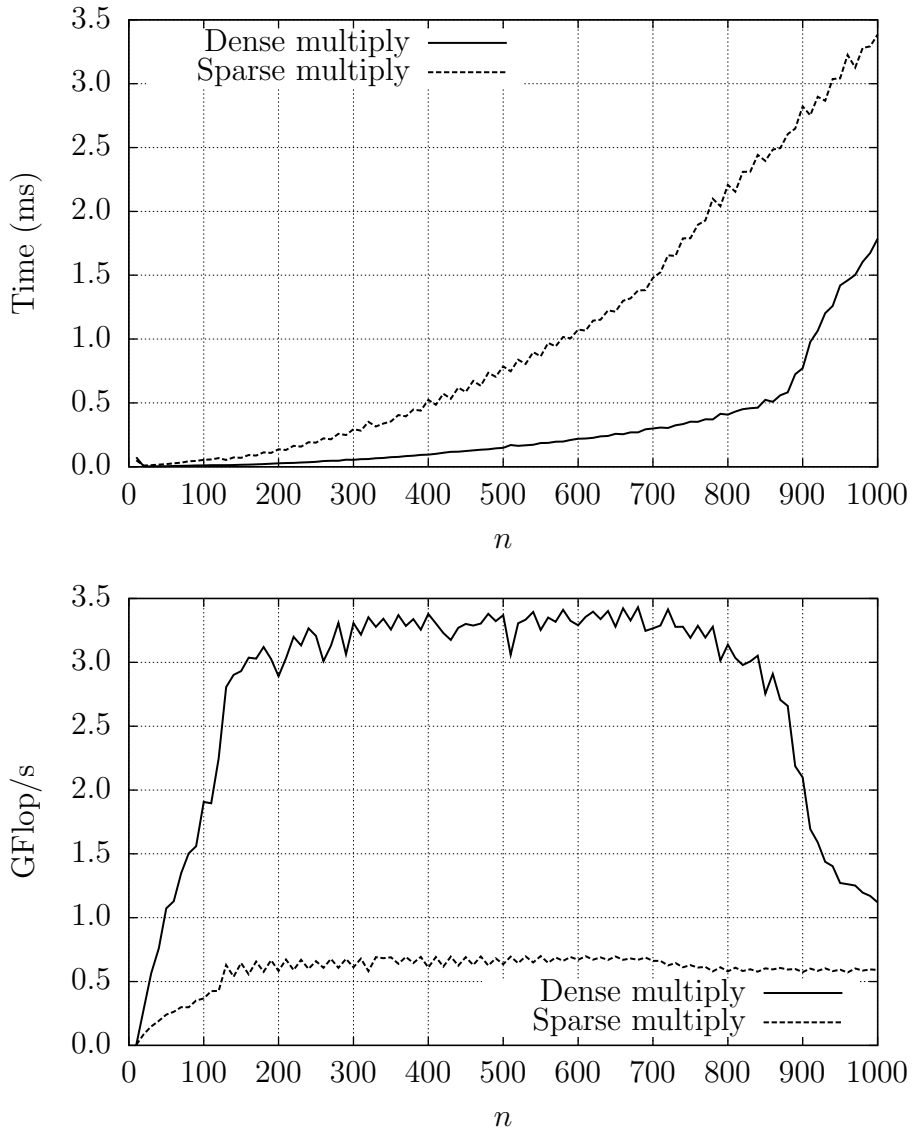

Figure 1: Times (top) and flop rates (bottom) for MATLAB sparse and dense matrix multiplication routines on a dual-core MacBook Pro.