Exploiting Structure

P. Sam Johnson

National Institute of Technology Karnataka (NITK)
Surathkal, Mangalore, India

Introduction

The efficiency of a given matrix algorithm depends on many things. Most obvious and what we treat in this section is the amount of required arithmetic and storage. We continue to use matrix-vector and matrix-matrix multiplication as a vehicle for introducing the key ideas. As examples of exploitable structure we have chosen the properties of bandedness and symmetry. Band matrices have many zero entries and so it is no surprise that band matrix manipulation allows for many arithmetic and storage shortcuts. Arithmetic complexity and data structures are discussed in this context.

Symmetric matrices provide another set of examples that can be used to illustrate structure exploitation. Symmetric linear systems and eigenvalue problems have a very prominent role to play in matrix computations and so it is important to be familiar with their manipulation.

Band Matrices and the $x - 0$ Notation

We say that $A \in \mathbb{R}^{m \times n}$ has lower bandwidth p if $a_{ij} = 0$ whenever $i > j + p$ and upper bandwidth q if $j > i + q$ implies $a_{ij} = 0$. Here is an example of an 8-by-5 matrix that has lower bandwidth 1 and upper bandwidth 2:

$$\begin{bmatrix} x & x & x & 0 & 0 \\ x & x & x & x & 0 \\ 0 & x & x & x & x \\ 0 & 0 & x & x & x \\ 0 & 0 & 0 & x & x \\ 0 & 0 & 0 & 0 & x \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix}.$$

The x's designates arbitrary nonzero entries. This notation is handy to indicate the zero-nonzero structure of a matrix and we use it extensively.

Diagonal Matrix Manipulation

Band structures that occur frequently are tabulated in the following.

Matrices with upper and lower bandwidth zero are diagonal. If $D \in \mathbb{R}^{m \times n}$ is diagonal, then

$$D = diag(d_1, \ldots, d_q), \quad q = \min\{m, n\} \iff d_i = d_{ii}$$

If D is diagonal and A is a matrix, then DA is a row scaling of A and AD is a column scaling of A.

Type of Matrix	Lower Bandwidth	Upper Bandwidth
diagonal	0	0
upper triangular	0	$n-1$
lower triangular	$m-1$	0
tridiagonal	1	1
upper bidiagonal	0	1
lower bidiagonal	1	0
upper Hessenberg	1	$n-1$
lower Hessenberg	$m-1$	1

Table: Band Terminology for m-by-n Matrices

Triangular Matrix Multiplication

To introduce band matrix "thinking" we look at the matrix multiplication problem $C = AB$ when A and B are both n-by-n and upper triangular.

The 3-by-3 case is illuminating:

$$C = \begin{bmatrix} a_{11}b_{11} & a_{11}b_{12} + a_{12}b_{22} & a_{11}b_{13} + a_{12}b_{23} + a_{13}b_{33} \\ 0 & a_{22}b_{22} & a_{22}b_{23} + a_{23}b_{33} \\ 0 & 0 & a_{33}b_{33} \end{bmatrix}.$$

It suggests that the product is upper triangular and that its upper triangular entries are the result of abbreviated inner products.

Triangular Matrix Multiplication (Contd...)

Indeed, since $a_{ik}b_{kj} = 0$ whenever $k < i$ or $j < k$ we see that $c_{ij} = \sum_{k=i}^{j} a_{ik}b_{kj}$ and so we obtain:

Algorithm 1.2.1 (Triangular Matrix Multiplication) If $A, B \in \mathbb{R}^{n\times n}$ are upper triangular, then this algorithm computes $C = AB$.

```
$C = 0$
for i=l:n
    for j=i:n
        for k=i:j
            $C(i,j) = A(i,k)B(k,j) + C(i,j)$
        end
    end
end
```

Flops

To quantify the savings in this algorithm we need some tools for measuring the amount of work.

Obviously, upper triangular matrix multiplication involves less arithmetic than when the matrices are full. One way to quantify this is with the notion of a flop. A flop[1] is a floating point operation. A dot product or saxpy operation of length n involves $2n$ flops because there are n multiplications and n adds in either of these vector operations.

The gaxpy $y = Ax + y$ where $A \in \mathbb{R}^{m \times n}$ involves $2mn$ flops as does an m-by-n outer product update of the form $A = A + xy^T$.

[1]In the first edition of this book we defined a flop to be the amount of work associated with an operation of the form $a_{ij} = a_{ij} + a_{ik}a_{kj}$, i.e., a floating point add, a floating point multiply, and some subscripting. Thus, an "old flop" involves two "new flops." In defining a flop to be a single floating point operation we are opting for a more precise measure of arithmetic complexity.

Flops (Contd...)

The matrix multiply update $C = AB + C$ where $A \in \mathbb{R}^{m\times p}$, $B \in \mathbb{R}^{p\times n}$, and $C \in \mathbb{R}^{m\times n}$ involves $2mnp$ flops.

Flop counts are usually obtained by summing the amount of arithmetic associated with the most deeply nested statements in an algorithm.

For matrix-matrix multiplication, this is the statement,

$$C(i,j) = A(i,k)B(k,j) + C(i,j)$$

which involves two flops and is executed mnp times as a simple loop accounting indicates. Hence the conclusion that general matrix multiplication requires $2mnp$ flops.

Flops (Contd...)

Now let us investigate the amount of work involved in Algorithm 1.2.1. Note that c_{ij}, $(i \le j)$ requires $2(j - i + 1)$ flops. Using the heuristics

$$\sum_{p=1}^{q} p = \frac{q(q+1)}{2} \approx \frac{q^2}{2}$$

and

$$\sum_{p=1}^{q} p^2 = \frac{q^3}{3} + \frac{q^2}{2} + \frac{q}{6} \approx \frac{q^3}{3}$$

we find that triangular matrix multiplication requires one-sixth the number of flops as full matrix multiplication:

$$\sum_{i=1}^{n} \sum_{j=i}^{n} 2(j - i + 1) = \sum_{i=1}^{n} \sum_{j=1}^{n-i+1} 2j \approx \sum_{i=1}^{n} \frac{2(n-i+1)^2}{2} = \sum_{i=1}^{n} i^2 \approx \frac{n^3}{3}.$$

Flops (Contd...)

We throw away the low order terms since their inclusion does not contribute to what the flop count “says.”

For example, an exact flop count of Algorithm 1.2.1 reveals that precisely $n^3/3 + n^2 + 2n/3$ flops are involved. For large n (the typical situation of interest) we see that the exact flop count offers no insight beyond the $n^3/3$ approximation.

Flop counting is a necessarily crude approach to the measuring of program efficiency since it ignores subscripting, memory traffic, and the countless other overheads associated with program execution.

We must not infer too much from a comparison of flops counts. We cannot conclude, for example, that triangular matrix multiplication is six times faster than square matrix multiplication. Flop counting is just a “quick and dirty” accounting method that captures only one of the several dimensions of the efficiency issue.

The Colon Notation-Again

The dot product that the k-loop performs in Algorithm 1.2.1 can be succinctly stated if we extend the colon notation introduced in § 1.1.8.

Suppose $A \in \mathbb{R}^{m \times n}$ and the integers p, q, and r satisfy $1 \leq p \leq q \leq n$ and $1 \leq r \leq m$. We then define

$$A(r, p:q) = [a_{rp}, \ldots, a_{rq}] \in \mathbb{R}^{1 \times (q-p+1)}.$$

Likewise, if $1 \leq p \leq q \leq m$ and $1 \leq c \leq n$, then

$$A(p:q, c) = \begin{bmatrix} a_{pc} \\ \vdots \\ a_{qc} \end{bmatrix} \in \mathbb{R}^{q-p+1}.$$

The Colon Notation-Again (Contd...)

With this notation we can rewrite Algorithm 1.2.1 as

```
$C(1:n, 1:n) = 0$
for i=1:n
    for j=i:n
        $C(i,j) = A(i, i:j)B(i:j, j) + C(i,j)$
    end
end
```

We mention one additional feature of the colon notation. Negative increments are allowed. Thus, if x and y are n-vectors, then $s = x^T y(n:-1:1)$ is the summation

$$s = \sum_{i=1}^{n} x_i y_{n-i+1}.$$

Band Storage

Suppose $A \in \mathbb{R}^{n\times n}$ has lower bandwidth p and upper bandwidth q and assume that p and q are much smaller than n.

Such a matrix can be stored in a $(p+q+1)$-by-n array $A.band$ with the convention that

$$a_{ij} = A.band(i-j+q+1, j) \tag{1}$$

for all (i, j) that fall inside the band.

Band Storage (Contd...)

Thus, if

$$A = \begin{bmatrix} a_{11} & a_{11} & a_{11} & 0 & 0 & 0 \\ a_{21} & a_{22} & a_{23} & a_{24} & 0 & 0 \\ 0 & a_{32} & a_{33} & a_{34} & a_{35} & 0 \\ 0 & 0 & a_{43} & a_{44} & a_{45} & a_{46} \\ 0 & 0 & 0 & a_{54} & a_{55} & a_{56} \\ 0 & 0 & 0 & 0 & a_{65} & a_{66} \end{bmatrix},$$

then

$$A.band = \begin{bmatrix} 0 & 0 & a_{13} & a_{24} & a_{35} & a_{46} \\ 0 & a_{12} & a_{23} & a_{34} & a_{45} & a_{56} \\ a_{11} & a_{22} & a_{33} & a_{44} & a_{55} & a_{66} \\ a_{21} & a_{32} & a_{43} & a_{54} & a_{65} & 0 \end{bmatrix}.$$

Here, the "0" entries are unused. With this data structure, our column-oriented gaxpy algorithm transforms to the following:

Band Storage (Contd...)

Algorithm 1.2.2 (Band Gaxpy) Suppose $A \in \mathbb{R}^{n\times n}$ has lower bandwidth p and upper bandwidth q and is stored in the $A.band$ format (1.2.1). If $x, y \in \mathbb{R}^n$, then this algorithm overwrites y with $Ax + y$.

```
for j=1:n
    $y_{top} = \max(1, j-q)$
    $y_{bot} = \min(n, j+p)$
    $a_{top} = \max(1, q+2-j)$
    $a_{bot} = a_{top} + y_{bot} - y_{top}$
    $y(y_{top} : y_{bot}) = x(j)A.band(a_{top} : a_{bot}, j) + y(y_{top} : y_{bot})$
end
```

Band Storage (Contd...)

Notice that by storing A by column in $A.band$, we obtain a saxpy, column access procedure.

Indeed, Algorithm 1.2.2 is obtained from Algorithm 1.1.4 by recognizing that each saxpy involves a vector with a small number of nonzeros.

Integer arithmetic is used to identify the location of these nonzeros.

As a result of this careful zero/nonzero analysis, the algorithm involves just $2n(p + q + 1)$ flops with the assumption that p and q are much smaller than n.

Symmetry

We say that $A \in \mathbb{R}^{n\times n}$ is symmetric if $A^T = A$. Thus,

$$A = \begin{bmatrix} 1 & 2 & 3 \\ 2 & 4 & 5 \\ 3 & 5 & 6 \end{bmatrix}$$

is symmetric. Storage requirements can be halved if we just store the lower triangle of elements, e.g., $A.vec = \begin{bmatrix} 1 & 2 & 3 & 4 & 5 & 6 \end{bmatrix}$. In general, with this data structure we agree to store the a_{ij} as follows:

$$a_{ij} = A.vec((j-1)n - j(j-1)/2 + i) \quad (i \geq j) \tag{2}$$

Let us look at the column-oriented gaxpy operation with the matrix A represented in $A.vec$.

Symmetry (Contd...)

Algorithm 1.2.3 (Symmetric Storage Gaxpy) Suppose $A \in \mathbb{R}^{n\times n}$ is symmetric and stored in the *A.vec* style (2). If $x, y \in \mathbb{R}^n$, then this algorithm overwrites y with $Ax + y$.

```
for j=1:n
    for i=1:j-1
        $y(i) = A.vec((i-1)n - i(i-1)/2 + j)x(j) + y(i)$
    end
    for i=j:n
        $y(i) = A.vec((j-1)n - j(j-1)/2 + i)x(j) + y(i)$
    end
end
```

This algorithm requires the same $2n^2$ flops that an ordinary gaxpy requires. Notice that the halving of the storage requirement is purchased with some awkward subscripting.

Store by Diagonal

Symmetric matrices can also be stored by diagonal. If

$$A = \begin{bmatrix} 1 & 2 & 3 \\ 2 & 4 & 5 \\ 3 & 5 & 6 \end{bmatrix},$$

then in a store-by-diagonal scheme we represent A with the vector

$$A.diag = \begin{bmatrix} 1 & 4 & 6 & 2 & 5 & 3 \end{bmatrix}.$$

In general, if $i \geq j$, then

$$a_{i+k,i} = A.diag(i + nk - k(k-1)/2) \quad (k \geq 0) \tag{3}$$

Some notation simplifies the discussion of how to use this data structure in a matrix-vector multiplication.

Store by Diagonal (Contd...)

If $A \in \mathbb{R}^{m \times n}$, then let $D(A,k) \in \mathbb{R}^{m \times n}$ designate the kth diagonal of A as follows:

$$[D(A,k)]_{ij} = \begin{cases} a_{ij} & j = i + k, \quad 1 \le i \le m, \quad 1 \le j \le n \\ 0 & \text{otherwise.} \end{cases}$$

Thus,

$$A = \begin{bmatrix} 1 & 2 & 3 \\ 2 & 4 & 5 \\ 3 & 5 & 6 \end{bmatrix} = \underbrace{\begin{bmatrix} 0 & 0 & 3 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}}_{D(A,2)} + \underbrace{\begin{bmatrix} 0 & 2 & 0 \\ 0 & 0 & 5 \\ 0 & 0 & 0 \end{bmatrix}}_{D(A,1)}$$

$$+ \underbrace{\begin{bmatrix} 1 & 0 & 0 \\ 0 & 4 & 0 \\ 0 & 0 & 6 \end{bmatrix}}_{D(A,0)} + \underbrace{\begin{bmatrix} 0 & 0 & 0 \\ 2 & 0 & 0 \\ 0 & 5 & 0 \end{bmatrix}}_{D(A,-1)} + \underbrace{\begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 3 & 0 & 0 \end{bmatrix}}_{D(A,-2)}.$$

Store by Diagonal (Contd...)

Returning to our store-by-diagonal data structure, we see that the nonzero parts of $D(A,0), D(A,1), \ldots, D(A,n-1)$ are sequentially stored in the $A.diag$ scheme (3).

The gaxpy $y = Ax + y$ can then be organized as follows:

$$y = D(A,0)x + \sum_{k=1}^{n-1}(D(A,k) + D(A,k)^T)x + y.$$

Working out the details we obtain the following algorithm.

Store by Diagonal (Contd...)

Algorithm 1.2.4 (Store-By-Diagonal Gaxpy) Suppose $A \in \mathbb{R}^{n \times n}$ is symmetric and stored in the *A.diag* style (3).

If $x, y \in \mathbb{R}^n$, then this algorithm overwrites y with $Ax + y$.

```
for i=1:n
    $y(i) = A.diag(i)x(i) + y(i)$
end
for k=1:n-1
    $t = nk - k(k-1)/2$
    $\{y = D(A,k)x + y\}$
    for i=1:n-k
        $y(i) = A.diag(i+t)x(i+k) + y(i)$
    end
    $\{y = D(A,k)^T x + y\}$
    for i=1:n-k
        $y(i+k) = A.diag(i+t)x(i) + y(i+k)$
    end
end
```

Note that the inner loops oversee vector multiplications:

$$y(1:n-k) = A.diag(t+1:t+n-k).*x(k+1:n) + y(1:n-k)$$
$$y(k+1:n) = A.diag(t+1:t+n-k).*x(1:n-k) + y(k+1:n)$$

A Note on Overwriting and Workspaces

An undercurrent in the above discussion has been the economical use of storage. Overwriting input data is another way to control the amount of memory that a matrix computation requires. Consider the n-by-n matrix multiplication problem $C = AB$ with the proviso that the “input matrix” B is to be overwritten by the “output matrix” C. We cannot simply transform

```
$C(1:n, 1:n) = 0$
for j=1:n
    for k=1:n
        $C(:, j) = C(:, j) + A(:, k)B(k, j)$
    end
end
```

to

```
for j=1:n
    for k=1:n
        $B(:, j) = B(:, j) + A(:, k)B(k, j)$
    end
end
```

because $B(:, j)$ is needed throughout the entire k-loop.

A Note on Overwriting and Workspaces

A linear workspace is needed to hold the jth column of the product until it is “safe” to overwrite $B(:,j)$:

```
for j=1:n
    $\omega(1:n) = 0$
    for k=1:n
        $\omega(:) = \omega(:) + A(:,k)B(k,j)$
    end
    $B(:,j) = \omega(:)$
end
```

A linear workspace overhead is usually not important in a matrix computation that has a

2-dimensional array of the same order.

Reference Books

1. Gene H. Golub and Charles F. Van Loan, Matrix Computations, 3rd Edition, Hindustan book agency, 2007.
2. A.R. Gourlay and G.A. Watson, Computational methods for matrix eigen problems, John Wiley & Sons, New York, 1973.
3. W.W. Hager, Applied numerical algebra, Prentice-Hall, Englewood Cliffs, N.J, 1988.
4. D.S. Watkins, Fundamentals of matrix computations, John Wiley and sons, N.Y, 1991.
5. C.F. Van Loan, Introduction to scientific computing: A Matrix vector approach using Matlab, Prentice-Hall, Upper Saddle River, N.J, 1997.