CS601: Software Development for Scientific Computing

Autumn 2021

Week15:

Matrix Algebra

Course Progress..

- Last week: Matrix Algebra
  - Three fundamental ways to multiply two matrices
    - Commonly occurring algorithmic patterns
  - BLAS routines and categorization, Computational intensity
  - Efficiency considerations
    - Cache, Storage Layout, Data movement, Parallel Functional Units, Blocked Matrix Multiplication, Recursive Matrix Multiplication
- This week: Matrix algebra contd.

Matrix Structure and Efficiency

- Sparse Matrices
  - Banded matrices
    - Tridiagonal
    - Diagonal
    - Triangular
    - etc.
- Symmetric Matrices

- Storage
- Computation

How can we exploit the matrix structure to optimize for storage and computation?

Sparse Matrices - Motivation

- Matrix Multiplication with Upper Triangular Matrices (C=C+AB)
  - The result, A*B, is also upper triangular

$$A = \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ 0 & a_{22} & a_{23} \\ 0 & 0 & a_{33} \end{bmatrix} \quad B = \begin{bmatrix} b_{11} & b_{12} & b_{13} \\ 0 & b_{22} & b_{23} \\ 0 & 0 & b_{33} \end{bmatrix}$$

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

Sparse Matrices - Motivation

- C=C+AB when A, B, C are upper triangular

```
for i=1 to N
    for j=i to N
        for k=i to j
            C[i][j] = C[i][j] + A[i][k]*B[k][j]
```

- Cost = $\sum_{i=1}^{N}\sum_{j=i}^{N} 2(j-i+1)$ flops (why 2? refer last week's slides)
- Using $\sum_{i=1}^{N} i \approx \frac{n^2}{2}$ and $\sum_{i=1}^{N} i^2 \approx \frac{n^3}{3}$
- $\sum_{i=1}^{N}\sum_{j=i}^{N} 2(j-i+1) \approx \frac{n^3}{3}$, 1/3rd the number of flops required for dense matrix-matrix multiplication

Sparse Matrices - Motivation

- Matrix Multiplication with Upper Triangular Matrices (C=C+AB)
  - Crude estimation of flop count = 1/3rd normal MatMul flop count.

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

A B

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

AB

Sparse Matrices

- Have lots of zeros (a *large* fraction)

X	X	O	O	X	O	O	O	X
O	X	O	O	X	O	X	O	O
O	X	X	X	O	X	O	O	X
X	O	O	X	O	O	X	O	O
O	X	O	X	X	O	O	O	X
O	X	X	O	O	O	X	X	X

- Representation
  - Many formats available
  - Compressed Sparse Row (CSR)
    - Two Vector of Vectors:
    - Three arrays:

```
vector<vector<double>> val;
vector<vector<int>> ind;
```

```
double *val; //size= NNZ
int *ind; //size=NNZ
int *rowstart; //size=M=Number of rows
```

Sparse Matrices - Example

- Using Arrays

```
double *val; //size= NNZ
int *ind; //size=NNZ
int *rowstart; //size=M=Number of rows
```

A

$$
A = \begin{bmatrix}
a_{11} & a_{12} & 0 & 0 & a_{15} & 0 & 0 & 0 & a_{19} \\
0 & a_{22} & 0 & 0 & a_{25} & 0 & a_{27} & 0 & 0 \\
0 & a_{32} & a_{33} & a_{34} & 0 & a_{36} & 0 & 0 & a_{39} \\
a_{41} & 0 & 0 & a_{44} & 0 & 0 & a_{47} & 0 & 0 \\
0 & a_{52} & 0 & a_{54} & a_{55} & 0 & 0 & 0 & a_{59} \\
0 & a_{62} & a_{63} & 0 & 0 & 0 & a_{67} & a_{68} & a_{69}
\end{bmatrix}
$$

val:

a_{11}	a_{12}	a_{15}	a_{19}	a_{22}	a_{25}	a_{27}	a_{32}	a_{33}	a_{34}	a_{36}	a_{39}	a_{41}	a_{44}	a_{47}	a_{52}	a_{54}	a_{55}	a_{59}	a_{62}	a_{63}	a_{67}	a_{68}	a_{69}

ind:

1	2	5	9	2	5	7	2	3	4	6	9	1	4	7	2	4	5	9	2	3	7	8	9

rowstart:

0	4	7	12	15	19	

Sparse Matrices - Example

$$
\mathbf{A} = \begin{pmatrix}
1.5 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 2.3 & 0 & 1.4 & 0 & 0 & 0 & 0 \\
0 & 0 & 3.7 & 0 & 0 & 0 & 0 & 0 \\
0 & -1.6 & 0 & 2.3 & 9.9 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 5.8 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 7.4 & 0 & 0 \\
0 & 0 & 1.9 & 0 & 0 & 0 & 4.9 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 3.6
\end{pmatrix}
$$

Using vectors:

```
vector<vector<double>> val;
vector<vector<int>> ind;
```

`ind`

1		
2	4	
3		
2	4	5
5		
6		
7	3	
8		

`val`

1.5		
2.3	1.4	
3.7		
-1.6	2.3	9.9
5.8		
7.4		
4.9	1.9	
3.6		

We represent a sparse matrix as two vectors of vectors:
`vector<vector<double> >`
to hold the matrix elements,
`vector<vector<int> >`
to hold the column indices.

Compressed-sparse-row (CSR) representation.

Slide example taken from CS294-73, Philip Colella

Sparse Matrices: y=y+Ax

- Using arrays

```
for i=0 to numRows
  for j=rowstart[i] to rowstart[i+1]-1
     y[i] = y[i] + val[j]*x[ind[j]]
```

- Does the above code reuse y, x, and val ? (we want our code to reuse as much data elements as possible while they are in fast memory):
  - y ? Yes. Read and written in close succession.
  - x ? Possible. Depends on how data is scattered in val.
  - val ? Less likely for a sparse matrix.

Sparse Matrices: y=y+Ax

- Optimization strategies:

```
for i=0 to numRows
  for j=rowstart[i] to rowstart[i+1]-1
     y[i] = y[i] + val[j]*x[ind[j]]
```

  - Unroll the j loop // we need to know the number of non-zeros per row
  - Move y[i] outside the loop //Possible only if y is not aliased.
  - Eliminate ind[i] and thereby the indirect access to elements of x. Indirect access is not good because we cannot predict the pattern of data access in x. //We need to know the column numbers
  - Reuse elements of x //The elements of val should be e.g. located closely

Sparse Matrices

- Further reading:

Refer to Lecture 15 (Spring 2018) at

https://inst.eecs.berkeley.edu/~cs267/archives.html

Banded Matrices

- Special case of sparse matrices, characterized by two numbers:
  - Lower bandwidth p, and upper bandwidth q
  - $a_{ij} = 0$ if $i > j+p$
  - $a_{ij} = 0$ if $j > i+q$
  - E.g. p=1, q=2
    for a 8x5 matrix
  (x represents non-zero element)

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

Banded Matrices - Representation

- Optimizing storage (specific to banded matrices)

a_{11}	a_{12}	a_{13}	0	0
a_{21}	a_{22}	a_{23}	a_{24}	0
0	a_{32}	a_{33}	a_{34}	a_{35}
0	0	a_{43}	a_{44}	a_{45}
0	0	0	a_{54}	a_{55}
0	0	0	0	a_{65}
0	0	0	0	0
0	0	0	0	0

A

*	*	a_{13}	a_{24}	a_{35}
*	a_{12}	a_{23}	a_{34}	a_{45}
a_{11}	a_{22}	a_{33}	a_{44}	a_{55}
a_{21}	a_{32}	a_{43}	a_{54}	a_{65}

A*band*

$A_{ij}=Aband(i-j+q+1, j)$

E.g. $A_{44} = Aband_{34}$

Banded Matrices: y= y + Aband x

- A=Aband: optimizing computation and storage

```
for j=1 to n
  alpha1=max(1, j-q)
  alpha2=min(n, j+p)
  beta1=max(1, q+2-j)
  for i=alpha1 to alpha2
      y[i]=y[i] + Aband(beta1+i-alpha1,j)*x[j]
```

- Cost? 2(p+q+1) time! Much lesser than $2N^2$ time required for regular y=y+Ax (assuming p and q are much smaller than n)

Banded Matrices

- Exercise: how much savings in memory do we get in A*band* compared to the vector of vectors representation in slide 6? Assume that the matrix is 8x5.

Faster y=Ax: Discrete Fourier Transforms (DFT)

- Very widely used
  - Image compression (jpeg)
  - Signal processing
  - Solving Poisson's Equation
- Represent A with F, a *Fourier Matrix* that has the following (remarkable) properties:
  - F^{-1} is easy to compute and consists of real numbers
  - Multiplications by F and F^{-1} is fast.
- F has complex numbers in its entries.
  - Every entry is a power of a single number w such that $w^n=1$
  - Any entry of a Fourier matrix can be written using $f_{ij} = w^{ij}$ (row and col indices start from 0)

Example: Fourier Matrix

- 4x4: $F_4 = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & w & w^2 & w^3 \\ 1 & w^2 & w^4 & w^6 \\ 1 & w^3 & w^6 & w^9 \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & w & w^2 & w^3 \\ 1 & w^2 & 1 & w^2 \\ 1 & w^3 & w^2 & w^1 \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & i & i^2 & i^3 \\ 1 & i^2 & i^4 & i^6 \\ 1 & i^3 & i^6 & i^9 \end{bmatrix}, i = \sqrt{-1}$
  - Here, w=i (also denoted as w_4=i). $w^4 = 1$ => i is a root.
- 8x8: $F_8 = \begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & w & w^2 & w^3 & w^4 & w^5 & w^6 & w^7 \\ 1 & w^2 & w^4 & w^6 & w^8 & w^{10} & w^{12} & w^{14} \\ 1 & w^3 & w^6 & w^9 & w^{12} & w^{15} & w^{18} & w^{21} \\ 1 & w^4 & w^8 & w^{12} & w^{16} & w^{20} & w^{24} & w^{28} \\ 1 & w^5 & w^{10} & w^{15} & w^{20} & w^{25} & w^{30} & w^{35} \\ 1 & w^6 & w^{12} & w^{18} & w^{24} & w^{30} & w^{36} & w^{42} \\ 1 & w^7 & w^{14} & w^{21} & w^{28} & w^{35} & w^{42} & w^{49} \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & w & w^2 & w^3 & w^4 & w^5 & w^6 & w^7 \\ 1 & w^2 & w^4 & w^6 & 1 & w^2 & w^4 & w^6 \\ 1 & w^3 & w^6 & w & w^4 & w^7 & w^2 & w^5 \\ 1 & w^4 & 1 & w^4 & 1 & w^4 & 1 & w^4 \\ 1 & w^5 & w^2 & w^7 & w^4 & w^1 & w^6 & w^3 \\ 1 & w^6 & w^4 & w^2 & 1 & w^6 & w^4 & w^2 \\ 1 & w^7 & w^6 & w^5 & w^4 & w^3 & w^2 & w^1 \end{bmatrix}$

Here, $w = \frac{1+\sqrt{\text{i}}}{2}$
(sqrt of i)

Example: Fourier Matrix

- $$\begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & w & w^2 & w^3 & w^4 & w^5 & w^6 & w^7 \\ 1 & w^2 & w^4 & w^6 & 1 & w^2 & w^4 & w^6 \\ 1 & w^3 & w^6 & w & w^4 & w^7 & w^2 & w^5 \\ 1 & w^4 & 1 & w^4 & 1 & w^4 & 1 & w^4 \\ 1 & w^5 & w^2 & w^7 & w^4 & w^1 & w^6 & w^3 \\ 1 & w^6 & w^4 & w^2 & 1 & w^6 & w^4 & w^2 \\ 1 & w^7 & w^6 & w^5 & w^4 & w^3 & w^2 & w^1 \end{bmatrix} = \left[\begin{array}{cccc|cccc} 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & \omega^2 & \omega^4 & \omega^6 & \omega & \omega^3 & \omega^5 & \omega^7 \\ 1 & \omega^4 & 1 & \omega^4 & \omega^2 & \omega^6 & \omega^2 & \omega^6 \\ 1 & \omega^6 & \omega^4 & \omega^2 & \omega^3 & \omega & \omega^7 & \omega^5 \\ 1 & 1 & 1 & 1 & -1 & -1 & -1 & -1 \\ 1 & \omega^2 & \omega^4 & \omega^6 & -\omega & -\omega^3 & -\omega^5 & -\omega^7 \\ 1 & \omega^4 & 1 & \omega^4 & -\omega^2 & -\omega^6 & -\omega^2 & -\omega^6 \\ 1 & \omega^6 & \omega^4 & \omega^2 & -\omega^3 & -\omega & -\omega^7 & -\omega^5 \end{array}\right]$$

(Writing columns 1,3,5,7 first and then columns 2,4,6,8
Also, using the fact that $w^4 = w^{2*} w^2 = i*i = -1$)

Example: Fourier Matrix

- $$\begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & w & w^2 & w^3 & w^4 & w^5 & w^6 & w^7 \\ 1 & w^2 & w^4 & w^6 & 1 & w^2 & w^4 & w^6 \\ 1 & w^3 & w^6 & w & w^4 & w^7 & w^2 & w^5 \\ 1 & w^4 & 1 & w^4 & 1 & w^4 & 1 & w^4 \\ 1 & w^5 & w^2 & w^7 & w^4 & w^1 & w^6 & w^3 \\ 1 & w^6 & w^4 & w^2 & 1 & w^6 & w^4 & w^2 \\ 1 & w^7 & w^6 & w^5 & w^4 & w^3 & w^2 & w^1 \end{bmatrix} = \left[\begin{array}{cccc|cccc} 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & \omega^2 & \omega^4 & \omega^6 & \omega & \omega^3 & \omega^5 & \omega^7 \\ 1 & \omega^4 & 1 & \omega^4 & \omega^2 & \omega^6 & \omega^2 & \omega^6 \\ 1 & \omega^6 & \omega^4 & \omega^2 & \omega^3 & \omega & \omega^7 & \omega^5 \\ \hline 1 & 1 & 1 & 1 & -1 & -1 & -1 & -1 \\ 1 & \omega^2 & \omega^4 & \omega^6 & -\omega & -\omega^3 & -\omega^5 & -\omega^7 \\ 1 & \omega^4 & 1 & \omega^4 & -\omega^2 & -\omega^6 & -\omega^2 & -\omega^6 \\ 1 & \omega^6 & \omega^4 & \omega^2 & -\omega^3 & -\omega & -\omega^7 & -\omega^5 \end{array}\right]$$

(Partitioning into 4 matrix blocks of size 4x4.)

Example: Fourier Matrix

- $$\begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & w & w^2 & w^3 & w^4 & w^5 & w^6 & w^7 \\ 1 & w^2 & w^4 & w^6 & 1 & w^2 & w^4 & w^6 \\ 1 & w^3 & w^6 & w & w^4 & w^7 & w^2 & w^5 \\ 1 & w^4 & 1 & w^4 & 1 & w^4 & 1 & w^4 \\ 1 & w^5 & w^2 & w^7 & w^4 & w^1 & w^6 & w^3 \\ 1 & w^6 & w^4 & w^2 & 1 & w^6 & w^4 & w^2 \\ 1 & w^7 & w^6 & w^5 & w^4 & w^3 & w^2 & w^1 \end{bmatrix} = \begin{array}{c|c} F_4 & \Omega_4 F_4 \\ \hline F_4 & -\Omega_4 F_4 \end{array}$$

(because $w^2 = w_4$)

$$\Omega_4 = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & w & 0 & 0 \\ 0 & 0 & w^2 & 0 \\ 0 & 0 & 0 & w^3 \end{bmatrix}$$ (note: $w = w_8 = \frac{1+\sqrt{\mathrm{i}}}{2}$)

- So, $F_8 = \begin{bmatrix} F_4 & \Omega_4 F_4 \\ F_4 & -\Omega_4 F_4 \end{bmatrix}$

FFT

- We can obtain 8 point DFT from 4 point DFT.
- How do we obtain the result of F_8x, i.e. y, from F_4 and x?
- y[1] to y[4] = y^{top} + d * y^{bottom}
  - d=[1, w, w^2, w^3] (note: w= w_8= $\frac{1+\sqrt{i}}{2}$)
  - $y^{top} = F_4 x_{odd}$ (x_{odd}= elements at odd numbered indices of vector x)
  - $y^{bottom} = F_4 x_{even}$ (x_{even}= elements at even numbered indices of vector x)

Divide-and-Conquer FFT (D&C FFT)

FFT(v, ϖ, m) … assume m is a power of 2

if m = 1 return v[0]

else

v_{even} = FFT(v[0:2:m-2], ϖ^2, m/2)

v_{odd} = FFT(v[1:2:m-1], ϖ^2, m/2)

ϖ-vec = [ϖ^0, ϖ^1, … $\varpi^{(m/2-1)}$] ← precomputed

return [v_{even} + (ϖ-vec .* v_{odd}),

v_{even} - (ϖ-vec .* v_{odd})]

- Matlab notation: ".*" means component-wise multiply.

Cost: T(m) = 2T(m/2)+O(m) = O(m log m) operations.

FFT

- Refer to Lecture 20 (Spring 2018) at

https://inst.eecs.berkeley.edu/~cs267/archives.html

- Section 1.4, Matrix Computations, 4th Ed, Golub and Van Loan
- Section 3.5, Linear Algebra and Its Applications, 4th Ed, Gilbert Strang