CS601: Software Development for Scientific Computing

Autumn 2023

Week6: Matrix Computations with Sparse Matrices, Tools for debugging and more

LAPACK – Linear Algebra Package

- LAPACK – uses BLAS-3 (1989 – now)
  - Ex: Obvious way to express Gaussian Elimination (GE) is adding multiples of one row to other rows – BLAS-1
    - How do we reorganize GE to use BLAS-3 ?
  - Contents of LAPACK (summary)
    - Algorithms that are (nearly) 100% BLAS-3
      - Linear Systems, Least Squares
    - Algorithms that are only ≈50% BLAS-3
      - Eigenproblems, Singular Value Decomposition (SVD)
    - Generalized problems (eg Ax = l Bx)
    - Error bounds for everything
    - Lots of variants depending on A's structure (banded, $A=A^T$, etc.)
  - How much code? (Release 3.9.0, Nov 2019) (www.netlib.org/lapack)
    - Source: 1982 routines, 827K LOC, Testing: 1210 routines, 545K LOC

slide source: www.cs.berkeley.edu/~demmel (CS267)

Matrix Data and Efficiency

- Sparse Matrices
  - E.g. banded matrices
  - Diagonal
  - Tridiagonal etc.
- Symmetric Matrices

$\xrightarrow{\textit{Admit optimizations w.r.t.}}$

- Storage
- Computation

Sparse Matrices - Motivation

- Matrix Multiplication with Upper Triangular Matrices (C=C+AB)

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

$$\underbrace{\begin{bmatrix} a_{11}b_{11} & a_{11}b_{12}+a_{12}\,b_{22} & a_{11}b_{13}+a_{12}\,b_{23} + a_{13}\,b_{13} \\ 0 & a_{22}b_{22} & a_{22}b_{23}+a_{23}\,b_{33} \\ 0 & 0 & a_{33}b_{33} \end{bmatrix}}_{A*B}$$

The result, A*B, is also upper triangular.

The non-zero elements appear to be like the result of *inner-product*

Sparse Matrices - Motivation

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

```
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 = $\Sigma_{i=1}^{N}\Sigma_{j=i}^{N}2(j-i+1)$ flops (why 2?)
- Using $\Sigma_{i=1}^{N}i \approx \frac{n^2}{2}$ and $\Sigma_{i=1}^{N}i^2 \approx \frac{n^3}{3}$
- $\Sigma_{i=1}^{N}\Sigma_{j=i}^{N}2(j-i+1) \approx \frac{n^3}{3}$, 1/3rd the number of flops required for dense matrix-matrix multiplication

Sparse Matrices

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

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

- Representation
  - Many formats available
  - Compressed Sparse Row (CSR)

<u>Implementation:Three arrays:</u>

```
double *val;
int *ind;
int *rowstart;
```

Sparse Matrices - Example

- Using Arrays

A

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

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

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	

Gaxpy with 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? Good spatial locality here. Less likely for a sparse matrix in general.

Gaxpy with 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
  - 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 a should be e.g. located closely

These optimizations will not work for y=y+Ax pseudocode in general. When you know the data pattern and metadata info as mentioned above, you can reorder computations (scheduling optimization), reorganize data for better locality.