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 = I 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

Matrix Data and Efficiency

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

Symmetric Matrices

– Thuiagonai etc.

- Admit optimizations w.r.t.
- Storage
- Computation

Sparse Matrices - Motivation

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

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

$$\begin{bmatrix} A & B & B & B \end{bmatrix}$$

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

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

- Cost = $\sum_{i=1}^{N} \sum_{j=i}^{N} 2(j-i+1)$ 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)

```
        X
        X
        0
        0
        X
        0
        0
        X

        0
        X
        0
        0
        X
        0
        0
        0
        0
        0
        0
        0
        0
        0
        0
        0
        0
        0
        0
        0
        0
        0
        0
        0
        0
        0
        0
        0
        0
        0
        0
        0
        0
        0
        0
        0
        0
        0
        0
        0
        0
        0
        0
        0
        0
        0
        0
        0
        0
        0
        0
        0
        0
        0
        0
        0
        0
        0
        0
        0
        0
        0
        0
        0
        0
        0
        0
        0
        0
        0
        0
        0
        0
        0
        0
        0
        0
        0
        0
        0
        0
        0
        0
        0
        0
        0
        0
        0
        0
        0
        0
        0
        0
        0
        0
        0
        0
        0
```

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

```
Implementation:Three arrays:
double *val;
int *ind;
int *rowstart;
```

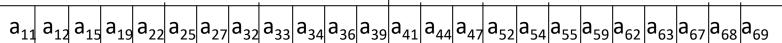
Sparse Matrices - Example

Using Arrays

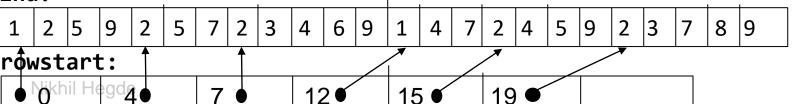
 A_{11} A_{12} A_{12} A_{13} A_{14} A_{15} A_{15} A

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

val:



ind:



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.

Nikhil Hegde

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.

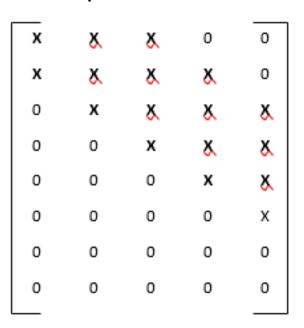
Banded Matrices

- Special case of sparse matrices, characterized by two numbers:
 - Lower bandwidth p, and upper bandwidth q

Exercise: When is $a_{ij} = 0$? (Write the constraints in terms of i, j, p, q)

$$- a_{ij} = 0 \text{ if } i > j+p$$

$$- a_{ij} = 0 \text{ if } j > i+q$$



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

E.g. A_{44 =} Aband₃₄

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

Gaxpy with 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? 2n(p+q+1) time! Much lesser than 2N² time required for regular y=y+Ax (assuming p and q are much smaller than n)

Nikhil Hegde

Tools

- Debugging
- Profiling
- Documenting

GDB

- GNU Debugger A tool for inspecting your
 C/C++ programs
 - How to begin inspecting a program using gdb?
 - How to control the execution?
 - How to display, interpret, and alter memory contents of a program using gdb?
 - Misc displaying stack frames, visualizing assembler code.