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

Admit optimizations w.r.t. →

- Storage
- Computation

Sparse Matrices - Motivation

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

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

A B

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

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

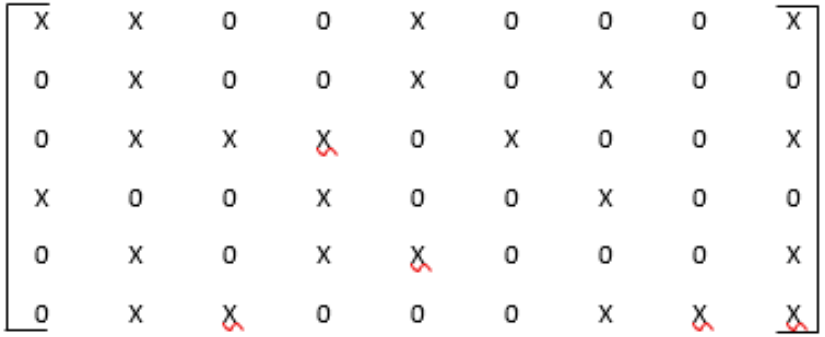
- C=C+AB when A, B, C are upper triangular, pseudocode:

```
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)



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

Implementation:Three arrays:

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

Sparse Matrices - Example

- Using Arrays

A

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}

```
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:

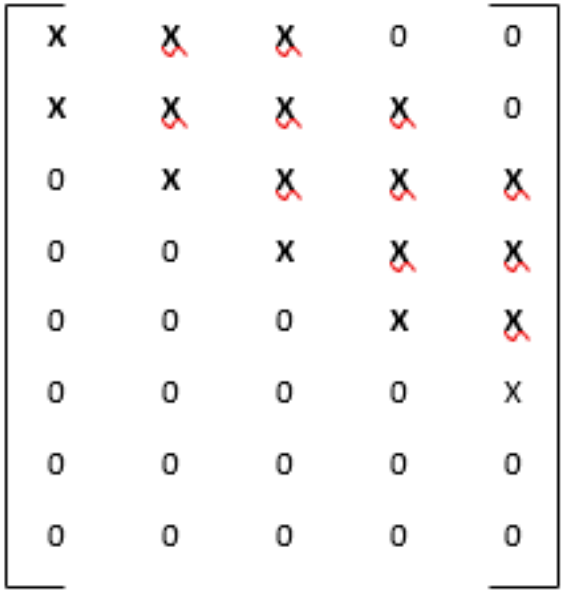
```
for i=0 to numRows
  for j=rowstart[i] to rowstart[i+1]-1
     y[i] = y[i] + val[j]*x[ind[j]]
```

  - 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.

Banded Matrices

- Special case of sparse matrices, characterized by two numbers:
  - Lower bandwidth p, and upper bandwidth q
  - E.g. p=1, q=2
    for a 8x5 matrix
    (x represents non-zero element)



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

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

Banded Matrices - Representation

- Optimizing storage (specific to banded matrices)

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

$$A_{band} = \begin{bmatrix} * & * & a_{13} & a_{24} & a_{35} \\ * & a_{12} & a_{23} & a_{34} & a_{45} \\ a_{11} & a_{22} & a_{33} & a_{44} & a_{55} \\ a_{21} & a_{32} & a_{43} & a_{54} & a_{65} \end{bmatrix}$$

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

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

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 $2N^2$ time required for regular y=y+Ax (assuming p and q are much smaller than n)

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.

GDB

– Compile your programs with –g option

```
hegden$gcc gdbdemo.c -o gdbdemo -g
hegden$
```

```
1 #include<stdio.h>
2 int foo(int a, int b)
3 {
4     int x = a + 1;
5     int y = b + 2;
6     int sum = x + y;
7
8     return x * y + sum;
9 }
10
11 int main()
12 {
13    int ret = foo(10, 20);
14    printf("value returned from foo: %d\n",ret);
15    return 0;
16 }
```

GDB – Start Debug

- Start debug mode (`gdb gdbdemo`)
  - Note the executable on first line (not .c files)
  - Note the last line before `(gdb)` prompt:
    - if –g option is not used while compiling, you will see "`(no debugging symbols found)`"

```
[ecegrid-thin4:~/ECE264] hegden$gdb gdbdemo
GNU gdb (GDB) Red Hat Enterprise Linux (7.2-92.el6)
Copyright (C) 2010 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "x86_64-redhat-linux-gnu".
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>...
Reading symbols from /home/min/a/hegden/ECE264/gdbdemo...done.
(gdb)
```

GDB – Set breakpoints

- Set breakpoints (b)
  - At line 14
  - Beginning of foo

```
1 #include<stdio.h>
2 int foo(int a, int b)
3 {
4     int x = a + 1;
5     int y = b + 2;
6     int sum = x + y;
7
8     return x * y + sum;
9 }
10
11 int main()
12 {
13    int ret = foo(10, 20);
14    printf("value returned from foo: %d\n",ret);
15    return 0;
16 }
```

```
(gdb) b gdbdemo.c:14
Breakpoint 1 at 0x400512: file gdbdemo.c, line 14.
(gdb) b foo
Breakpoint 2 at 0x4004ce: file gdbdemo.c, line 4.
(gdb)
```

GDB – Manage breakpoints

- Display all breakpoints set (`info b`)

```
(gdb) info b
Num     Type           Disp Enb Address            What
1       breakpoint     keep y   0x0000000000400512 in main at gdbdemo.c:14
2       breakpoint     keep y   0x00000000004004ce in foo at gdbdemo.c:4
(gdb)
```

- Delete a breakpoint (`d <breakpoint num>`)

```
(gdb) d 1
(gdb) info b
Num     Type           Disp Enb Address            What
2       breakpoint     keep y   0x00000000004004ce in foo at gdbdemo.c:4
(gdb)
```

- Disable a breakpoint (`disable <breakpoint num>`)

```
(gdb) disable 2
(gdb) info b
Num     Type           Disp Enb Address            What
2       breakpoint     keep n   0x00000000004004ce in foo at gdbdemo.c:4
(gdb)
```

- Enable breakpoint (`enable <breakpoint num>`)

```
(gdb) enable 2
(gdb) info b
Num     Type           Disp Enb Address            What
2       breakpoint     keep y   0x00000000004004ce in foo at gdbdemo.c:4
```

GDB – Start execution

- Start execution (`r <command-line arguments>`)
  - Execution stops at the first breakpoint encountered

```
(gdb) r
Starting program: /home/min/a/hegden/ECE264/gdbdemo

Breakpoint 3, main () at gdbdemo.c:13
13          int ret = foo(10, 20);
```

  - Continue execution (`c`)

```
(gdb) c
Continuing.

Program exited normally.
```

GDB – Printing

– Printing variable values (`p <variable_name>`)

```
Breakpoint 2, foo (a=10, b=20) at gdbdemo.c:4
4           int x = a + 1;
(gdb) n
5           int y = b + 2;
(gdb) p x
$3 = 11
```

– Printing addresses (`p &<variable_name>`)

```
(gdb) p &x
$5 = (int *) 0x7fffffffc4f4
```

GDB – Step in

– Steps inside a function call (s)

```
Breakpoint 3, main () at gdbdemo.c:13
13          int ret = foo(10, 20);
(gdb) s
foo (a=10, b=20) at gdbdemo.c:4
4       _     int x = a + 1;
```

GDB – Step out

– Jump to return address (`finish`)

```
(gdb) finish
Run till exit from #0  foo (a=10, b=20) at gdbdemo.c:4
0x000000000040050f in main () at gdbdemo.c:13
13          int ret = foo(10, 20);
Value returned is $2 = 275
```

GDB – Memory dump

– Printing memory content (`x/nfu <address>`)

- n = repetition (number of bytes to display)
- f = format (‘x’ – hexadecimal, ‘d’-decimal, etc.)
- u = unit (‘b’ – byte, ‘h’ – halfword/2 bytes, ‘w’ – word/4 bytes, ‘g’ – giga word/8 bytes)
- `E.g. x/16xb 0x7fffffffc500` (display the values of 16 bytes stored from starting address

```
(gdb) x/16xb 0x7fffffffc500
0x7fffffffc500: 0x20    0xc5    0xff    0xff    0xff    0x7f    0x00    0x00
0x7fffffffc508: 0x0f    0x05    0x40    0x00    0x00    0x00    0x00    0x00
```

GDB – Printing addresses

- `Registers ($rsp, $rbp)`
  - Note that we use the ‘x’ command and not the ‘p’ command.

```
(gdb) x $rsp
0x7fffffffc500: 0x20
(gdb) x $rbp
0x7fffffffc500: 0x20
```

GDB – Altering memory content

- Set command (`set variable <name> = value`)

```
(gdb) n
6           int sum = x + y;
(gdb) p x
$7 = 11
(gdb) p y
$8 = 22
(gdb) set variable y = 0
(gdb) n
8           return x * y + sum;
(gdb) p sum
$9 = 11
```

- Set command (`set *(<type *>addr) = value`)

GDB Demo

- Refer to the demo example