

CS601: Software Development for Scientific Computing

Autumn 2022

Week6: Motifs – Matrix Computations with
Dense and Sparse Matrices

Last week..

- Three fundamental ways to multiply two matrices
 - Comprising of dot products, linear combination of the left matrix columns, outer product updates
 - Commonly occurring algorithmic patterns / kernels :
Dot product, AXPY and GAXPY, outer product, matrix-vector product, matrix-matrix product
- Linear algebra software (BLAS, LAPACK)
 - BLAS routines and categorization
- Algorithmic costs
 - Arithmetic cost
 - Data movement cost
- Computational intensity (examples: axpy, matvec, matmul)

Last week - Communication Cost

//Assume A, B, C are all nxn

```
for i=1 to n
  for j=1 to n
    for k=1 to n
      C(i,j)=C(i,j) + A(i,k)*B(k,j)
```

- loop k=1 to n: read C(i,j) into fast memory and update in fast memory
- End of loop k=1 to n: write C(i,j) back to slow memory

- n^2 words read: each row of A read once for each i.
- Assume that row i of A stays in fast memory during j=2, .. J=n
- Reading a row i of A

n^2 words read and n^2 words written (each entry of C read/written to memory once).
= $2 n^2$ words read/written

- Reading column j of B
- Suppose there is space in fast memory to hold only one column of B (in addition to one row of A and 1 element of C), then every column of B is read from slow memory to fast memory once in **inner two loops**.
- Each column of B read n times including **outer i loop** = n^3 words read

total cost = $3 n^2 + n^3$ (if the cache size is $n+n+1$)

Last week – Computational Intensity of Matmul (ijk)

- Words moved = $n^3 + 3n^2 = n^3 + O(n^2)$
- Number of arithmetic operations = $2n^3$ (from slide 35)
- computational intensity $q \approx 2n^3/n^3 = 2$. (computation to communication ratio)

Same as q for matrix-vector?

What if the fast memory has more space ? more than just two columns + one element space?

- Can we do better?

Last week - Blocked Matrix Multiply

- For $N=4$:

$$\begin{bmatrix} C1 & C2 & C3 & C4 \end{bmatrix} = \begin{bmatrix} C1 & C2 & C3 & C4 \end{bmatrix} + \begin{bmatrix} A \end{bmatrix} * \begin{bmatrix} B1 & B2 & B3 & B4 \end{bmatrix}$$

$$\begin{bmatrix} C_j \end{bmatrix} = \begin{bmatrix} C_j \end{bmatrix} + \begin{bmatrix} A \end{bmatrix} * \begin{bmatrix} B_j \end{bmatrix} = \begin{bmatrix} C_j \end{bmatrix} + \sum_{k=1}^n \begin{bmatrix} A(:,k) \end{bmatrix} * \begin{bmatrix} B_j(k,:) \end{bmatrix}$$

```

for j=1 to N
  //Read entire Bj into fast memory
  //Read entire Cj into fast memory
  for k=1 to n
    //Read column k of A into fast memory
    Cj=Cj + A(*,k) * Bj(k,*)
  //Write Cj back to slow memory
  
```

Last week – Computational Intensity

```
for j=1 to N
//Read entire Bj into fast memory →  $n^2$  words read: each column
//Read entire Cj into fast memory
  for k=1 to n
    //Read column k of A into fast memory →  $Nn^2$  words read: each
    //column of A read N times
     $C(*,j) = C(*,j) + A(*,k) * B_j(k,*)$  //outer-product
    //Write Cj back to slow memory →  $2n^2$  words read:
    //read/write each entry of C
    //to memory once.
```

- Number of arithmetic operations = $2n^3$
- $q = 2n^3 / (N + 3)n^2 = 2n/N$. **Good!**

Blocked Matrix Multiply - General

$$\begin{array}{ccc}
 C & A & B \\
 \begin{bmatrix} C_{11} & C_{12} & \dots & C_{1r} \\ C_{21} & C_{22} & \dots & C_{2r} \\ \vdots & \vdots & \ddots & \vdots \\ C_{q1} & C_{q2} & \dots & C_{qr} \end{bmatrix} & \begin{bmatrix} A_{11} & A_{12} & \dots & A_{1p} \\ A_{21} & A_{22} & \dots & A_{2p} \\ \vdots & \vdots & \ddots & \vdots \\ A_{q1} & A_{q2} & \dots & A_{qp} \end{bmatrix} & \begin{bmatrix} B_{11} & B_{12} & \dots & B_{1r} \\ B_{21} & B_{22} & \dots & B_{2r} \\ \vdots & \vdots & \ddots & \vdots \\ B_{p1} & B_{p2} & \dots & B_{pr} \end{bmatrix} \\
 \begin{array}{c} \downarrow \quad \rightarrow \\ q \quad r \end{array} & \begin{array}{c} \downarrow \quad \rightarrow \\ q \quad p \end{array} & \begin{array}{c} \downarrow \quad \rightarrow \\ p \quad r \end{array}
 \end{array}$$

- $A, B, C \in \mathbb{R}^{n \times n}$
- We wish to update C block-by-block: $C_{ij} = C_{ij} + \sum_{k=1}^p A_{ik} B_{kj}$
 - Assume that blocks of A , B , and C fit in cache. C_{ij} is roughly n/q by n/r , A_{ij} is roughly n/q by n/p , B_{ij} is roughly n/p by n/r .
 - But how to choose block parameters p, q, r such that assumption holds for a cache of size M ?
 - i.e. given the constraint that $\frac{n}{q} \times \frac{n}{r} + \frac{n}{q} \times \frac{n}{p} + \frac{n}{p} \times \frac{n}{r} \leq M$

Blocked Matrix Multiply - General

- Maximize $\frac{2n^3}{qrp}$ subject to $\frac{n}{q} \times \frac{n}{r} + \frac{n}{q} \times \frac{n}{p} + \frac{n}{p} \times \frac{n}{r} \leq M$
 - $q_{opt} = p_{opt} = r_{opt} \approx \sqrt{\frac{3n^2}{M}}$
- Each block should roughly be a square matrix and occupy one third of the cache size
- Can we design algorithms that are independent of cache size?

Recursive Matrix Multiply

- Cache-oblivious algorithm
 - No matter what the size of the cache is, the algorithm performs at a near-optimal level
- Divide-conquer approach

$$\begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \times \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix} = \begin{bmatrix} A_{11}B_{11} + A_{12}B_{21} & A_{11}B_{12} + A_{12}B_{22} \\ A_{21}B_{11} + A_{22}B_{21} & A_{21}B_{12} + A_{22}B_{22} \end{bmatrix}$$

- Apply the formula recursively to $A_{11}B_{11}$ etc.
 - Works neat when n is a power of 2.
- What layout format is preferred for this algorithm?
 - Row-major or Col-major? Neither.

Recursive Matrix Multiply

- Cache-oblivious Data structure

$$\begin{bmatrix} 1 & 2 & 5 & 6 & 17 & 18 & 21 & 22 \\ 3 & 4 & 7 & 8 & 19 & 20 & 23 & 24 \\ 9 & 10 & 13 & 14 & 25 & 26 & 29 & 30 \\ 11 & 12 & 15 & 16 & 27 & 28 & 31 & 32 \\ 33 & 34 & 37 & 38 & 49 & 50 & 53 & 54 \\ 35 & 36 & 39 & 40 & 51 & 52 & 55 & 56 \\ 41 & 42 & 45 & 46 & 57 & 58 & 61 & 62 \\ 43 & 44 & 47 & 48 & 59 & 60 & 63 & 64 \end{bmatrix}.$$

- Matrix entries are stored in the order shown
 - E.g. row-major would have 1-8 in the first row, followed by 9-16 in the second and so on.

Summary- matmul

- Unblocked Matrix Multiplication - Loop Orderings and Properties

Loop Order	Inner Loop	Inner Two Loops	Inner Loop Data Access
i j k	dot	Vector x Matrix	A by row, B by column
j k i	saxpy	gaxpy	A by column, C by column
k j i	saxpy	Outer product	A by column, C by column
.. (3 more rows here..)			

Ref: Matrix Computations, 4th Ed., Golub and Van Loan

- Blocked matrix multiplication
 - Column blocking, row blocking, tiling
- Recursive matrix multiplication
 - Divide-conquer, Strassen's
- Many more?

Efficiency Considerations for a High-Performing Implementation

- Cache details (size)
- Data movement overhead
- Storage layout
- Parallel and 'special' functional Units (e.g. Vector units and fused multiply-add)

Parallel Functional Units

- IBM's RS/6000 and Fused Multiply Add (FMA)
 - Fuses multiply and an add into one functional unit ($c=c+a*b$)
 - The functional unit consists of 3 independent subunits : *Pipelining*
 - Example: Suppose the FMA unit takes 3 cycles to complete,

```
sum=0.0
for (i=0;i<n;i++)
    sum=sum+a[i]*b[i]
```

how many cycles do you need to execute this code snippet?

```
sum=0.0
for (i=0;i<n;i+=4)
    sum=sum+a[i]*b[i]
    sum=sum+a[i+1]*b[i+1]
    sum=sum+a[i+2]*b[i+2]
    sum=sum+a[i+3]*b[i+3]
```

how many cycles do you need to execute this code snippet?

Matrix Structure and Efficiency

- | | | |
|--|-----------------------------------|---|
| <ul style="list-style-type: none">• Sparse Matrices<ul style="list-style-type: none">– E.g. banded matrices– Diagonal– Tridiagonal etc.• Symmetric Matrices | <i>Admit optimizations w.r.t.</i> | <ul style="list-style-type: none">• Storage• Computation |
|--|-----------------------------------|---|

Sparse Matrices - Motivation

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

$$\begin{array}{ccc}
 \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ 0 & a_{22} & a_{23} \\ 0 & 0 & a_{33} \end{bmatrix} & \begin{bmatrix} b_{11} & b_{12} & b_{13} \\ 0 & b_{22} & b_{23} \\ 0 & 0 & b_{33} \end{bmatrix} & = \\
 \textcolor{blue}{A} & \textcolor{blue}{B} & \\
 & & \begin{bmatrix} a_{11}b_{11} & a_{11}b_{12}+a_{12}b_{22} & a_{11}b_{13}+a_{12}b_{23}+a_{13}b_{33} \\ 0 & a_{22}b_{22} & a_{22}b_{23}+a_{23}b_{33} \\ 0 & 0 & a_{33}b_{33} \end{bmatrix} \\
 & & \textcolor{blue}{AB}
 \end{array}$$

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
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?)
- 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

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

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

- 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	
---	---	---	----	----	----	--

Nikhil Hegde

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 a 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>

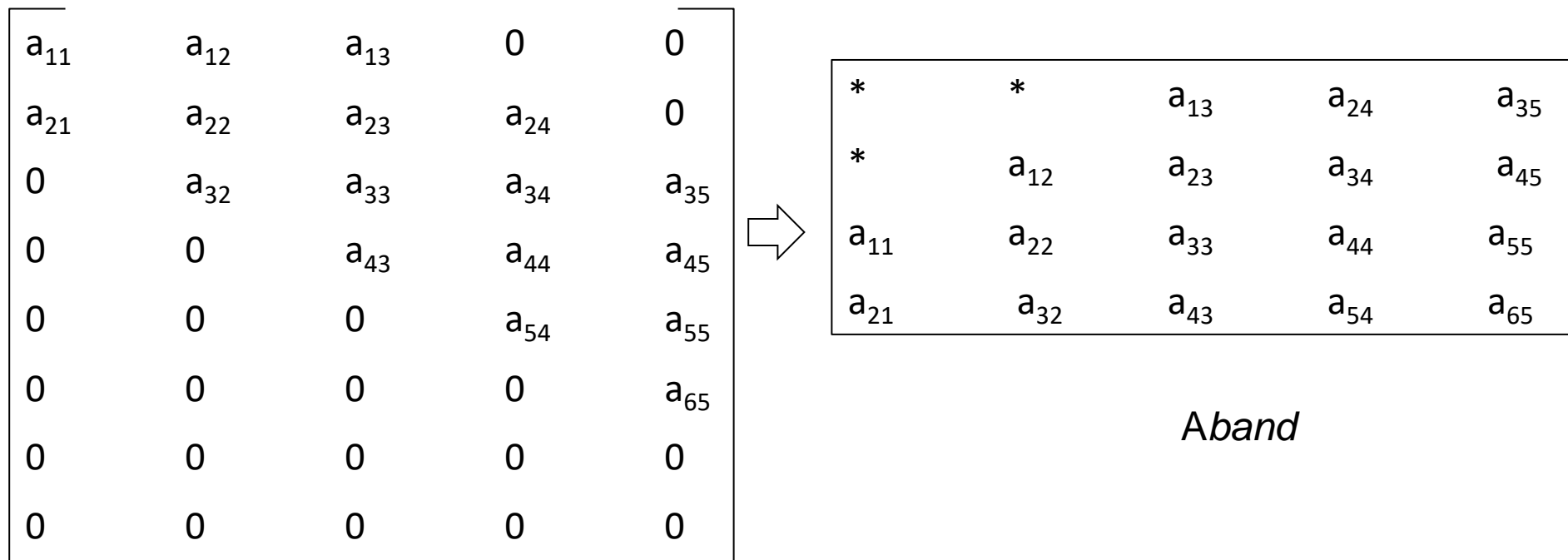
Banded Matrices

- 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 8×5 matrix
(x represents non-zero element)

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

Banded Matrices - Representation

- Optimizing storage (specific to banded matrices)



A

Aband

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

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

Banded Matrices: $y = y + A \text{band } x$

- $A = A_{\text{band}}$: 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)