

BLAS and LAPACK + Data Formats for Sparse Matrices

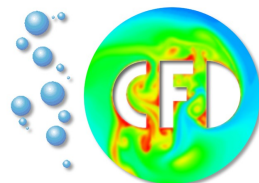
Part of the lecture „Wissenschaftliches Rechnen“

Hilmar Wobker

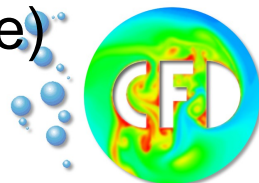
Institute of Applied Mathematics and Numerics, TU Dortmund

email: `hilmar.wobker@math.tu-dortmund.de`

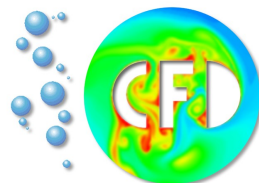
SS 2010



- BLAS = Basic Linear Algebra Subprograms
- building blocks for basic scalar/vector/matrix operations
(*only for dense and banded matrices!*)
- specification and Fortran77 reference implementation:
<http://netlib.org/blas/>
- efficient, portable, freely available
- machine-specific optimised BLAS implementations:
 - by hardware-vendors:
AMD (ACML), Apple (Velocity Engine), Compaq (CXML),
Cray (libsci), HP (MLIB), IBM (ESSL), Intel (MKL), NEC
(PDLIB/SX), SGI (SCSL), SUN (Sun Performance Library)
 - others:
ATLAS (Automatically Tuned Linear Algebra Software)
GotoBLAS

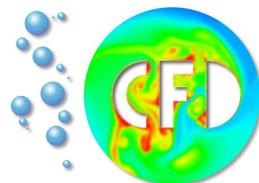


- first letter → data type:
 - s: single precision (4 byte)
 - d: double precision (8 byte)
 - c: complex single precision (2*4 byte)
 - z: complex double precision (2*8 byte)
- then description of matrix type (if necessary):
 - GE: general, SY: symmetric, TR: triangular, SB: symmetric band, ...
- then: shortcut of the operation
 - COPY: copy, DOT: dot product, ...

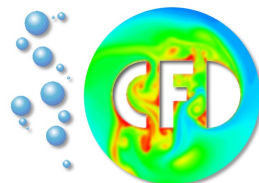


- level 1: scalar-, vector- and vector-vector operations
- level 2: matrix-vektor-operations
- level 3: matrix-matrix operations
- examples (also see BLAS quick reference):

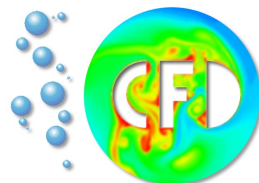
<code>_SCAL</code>	$x \leftarrow \alpha x$
<code>_COPY</code>	$x \leftarrow y$
<code>_DOT</code>	$\mu \leftarrow x^t y$
<code>_AXPY</code>	$y \leftarrow \alpha x + y$
<code>_GER</code>	$A \leftarrow \alpha x y^t + A$
<code>_GEMV</code>	$y \leftarrow \alpha A x + \beta y$
<code>_GEMM</code>	$C \leftarrow \alpha A B + \beta C$



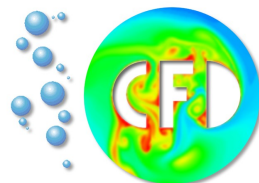
- LAPACK = Linear Algebra Package
→ <http://netlib.org/lapack/>
- library for solving more complex linear algebra tasks:
 - solving linear systems of equations
 - least-squares
 - eigenvalue problems
 - singular value decomposition
 - matrix factorisations (LU, Cholesky, ...)
- Examples:
 - `dsteqr`: diagonalisation of real, symmetric, tridiagonal matrices
 - `dgetrf`: LU decomposition, `dgetrs`: solve based on LU
 - `dgesv` = `dgetrf` + `dgetrs`
 - BLAS notation!



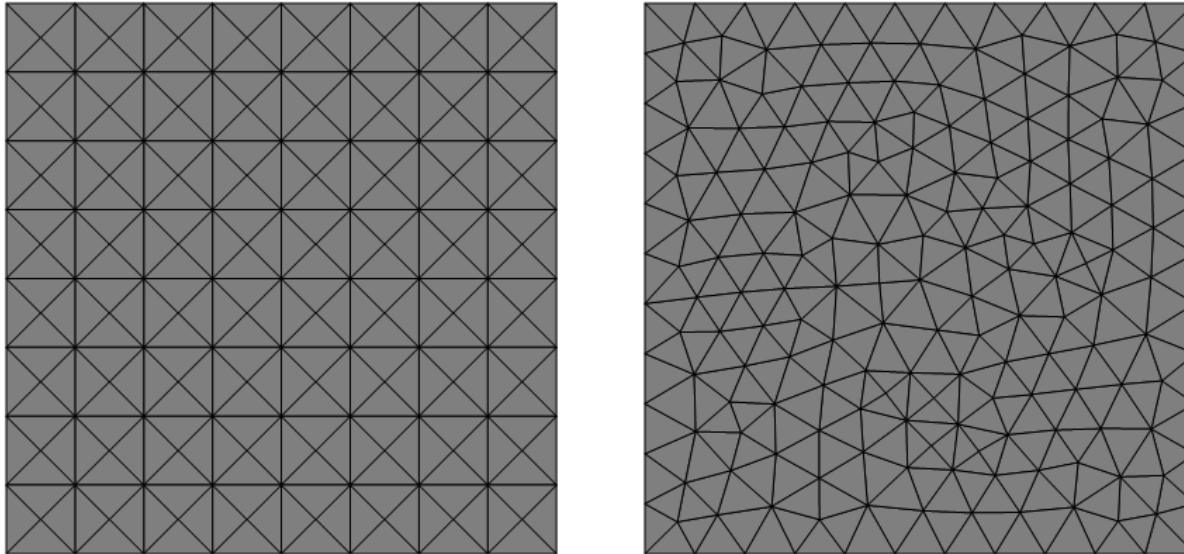
- supports dense and banded matrices
(here 'banded' means: one central band)
- does not support general sparse matrices!
- real + complex, single + double
- written in Fortran90
- heavy use of BLAS
- => if BLAS is efficient, then e.g. LU decomposition is efficient



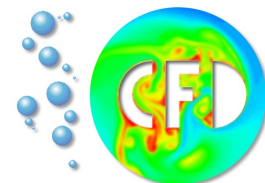
- fundamental property of finite elements:
basis functions with local support
- nonzero entry (NZ) a_{ij}
 - basis functions i and j have common support
 - nodes i and j are neighboured
- number of nonzero entries (NNZ) in i -th row depends on
 - degree of node i in the mesh
(i.e., #adjacent nodes, #incident elements)
 - FE space (Q1, Q2, ...)
 - spatial dimension



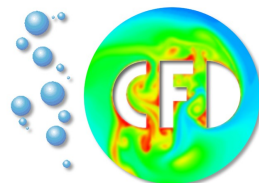
- typical mesh: maximal node degree only slightly greater than average node degree



- N mesh vertices
 - full matrix: $O(N^2)$ entries
 - $NNZ = O(N)$



- only represent nonzero entries
- efficiency in terms of
 - memory utilisation
 - arithmetic operations
- → highly dependent on matrix structure and the specific problem
- → many different formats



DNS Dense format

BND Linpack Banded format

★ **CSR** Compressed Sparse Row format

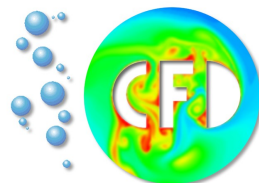
★ **CSC** Compressed Sparse Column format

★ **COO** Coordinate format

★ **ELL** Ellpack-Itpack generalized diagonal format

★ **DIA** Diagonal format

BSR Block Sparse Row format



★ **MSR** Modified Compressed Sparse Row format

SSK Symmetric Skyline format

NSK Nonsymmetric Skyline format

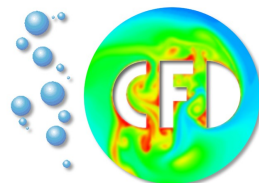
LNK Linked list storage format

★ **JAD** The Jagged Diagonal format

SSS The Symmetric Sparse Skyline format

USS The Unsymmetric Sparse Skyline format

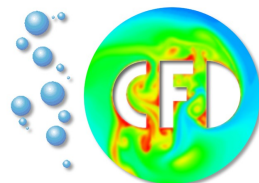
VBR Variable Block Row format



$$A = \begin{pmatrix} 1. & 0. & 0. & 2. & 0. \\ 3. & 4. & 0. & 5. & 0. \\ 6. & 0. & 7. & 8. & 9. \\ 0. & 0. & 10. & 11. & 0. \\ 0. & 0. & 0. & 0. & 12. \end{pmatrix}$$

AA	12.	9.	7.	5.	1.	2.	11.	3.	6.	4.	8.	10.
JR	5	3	3	2	1	1	4	2	3	2	3	4
JC	5	5	3	4	1	4	4	1	1	2	4	3

- 3 arrays of length NNZ:
 - AA = NZ matrix entries
 - JR / JC = row / column indices
- arbitrary order



$$A = \begin{pmatrix} 1. & 0. & 0. & 2. & 0. \\ 3. & 4. & 0. & 5. & 0. \\ 6. & 0. & 7. & 8. & 9. \\ 0. & 0. & 10. & 11. & 0. \\ 0. & 0. & 0. & 0. & 12. \end{pmatrix}$$

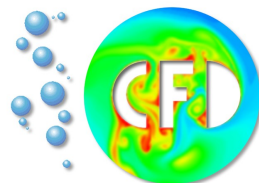
- order entries rowwise:

$$AA = [1. \ 2. \ 3. \ 4. \ 5. \ 6. \ 7. \ 8. \ 9. \ 10. \ 11. \ 12.]$$

$$JR = [1 \ 1 \ 2 \ 2 \ 2 \ 3 \ 3 \ 3 \ 3 \ 4 \ 4 \ 5]$$

$$JC = [1 \ 4 \ 1 \ 2 \ 4 \ 1 \ 4 \ 5 \ 6 \ 3 \ 4 \ 5]$$

- → redundant information in JR



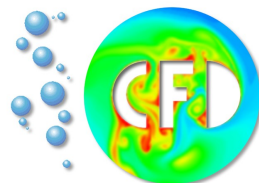
$$A = \begin{pmatrix} 1. & 0. & 0. & 2. & 0. \\ 3. & 4. & 0. & 5. & 0. \\ 6. & 0. & 7. & 8. & 9. \\ 0. & 0. & 10. & 11. & 0. \\ 0. & 0. & 0. & 0. & 12. \end{pmatrix}$$

- order NZ entries rowwise:

$$\begin{array}{l} AA = [1. \ 2. \ 3. \ 4. \ 5. \ 6. \ 7. \ 8. \ 9. \ 10. \ 11. \ 12.] \\ JR = [1 \ 1 \ 2 \ 2 \ 2 \ 3 \ 3 \ 3 \ 3 \ 4 \ 4 \ 5] \\ JC = [1 \ 4 \ 1 \ 2 \ 4 \ 1 \ 4 \ 5 \ 6 \ 3 \ 4 \ 5] \end{array}$$

↑ ↑ ↑ ↑ ↑ ↑

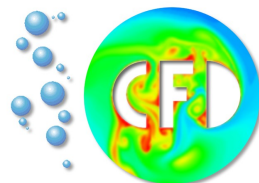
- → redundant information in JR
- idea: only store array indices (↑) where matrix rows change: [1 3 6 10 12]

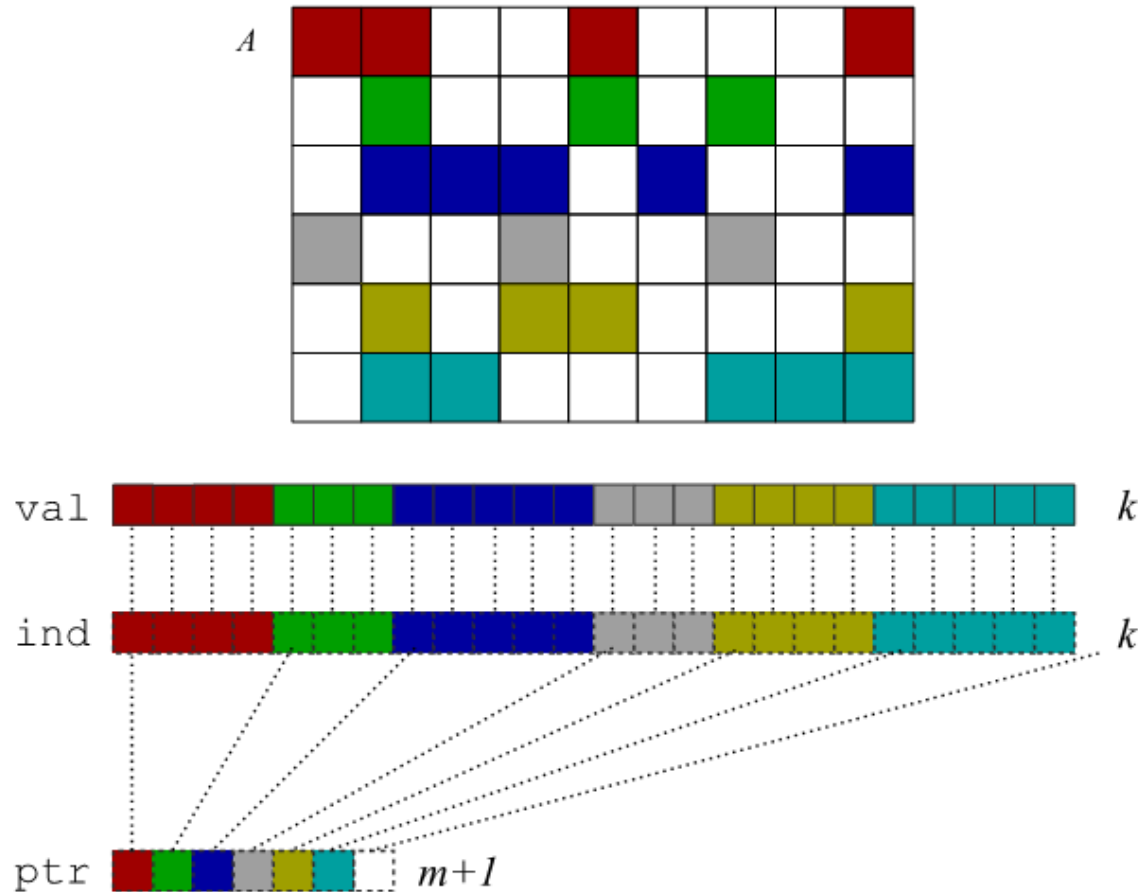


$$A = \begin{pmatrix} 1. & 0. & 0. & 2. & 0. \\ 3. & 4. & 0. & 5. & 0. \\ 6. & 0. & 7. & 8. & 9. \\ 0. & 0. & 10. & 11. & 0. \\ 0. & 0. & 0. & 0. & 12. \end{pmatrix}$$

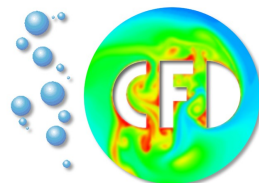
AA	1.	2.	3.	4.	5.	6.	7.	8.	9.	10.	11.	12.
JA	1	4	1	2	4	1	3	4	5	3	4	5
IA	1	3	6	10	12	13						

- 2 arrays of length NNZ, 1 array of length N+1 (N=5, NNZ=12)
- JA = column indices
- IA = pointers (positions in AA and JA) where new rows begin
- $IA(N+1) = IA(1) + NNZ$: pointer to 'virtual' (N+1)-th row
- Note: when AA subarray $\rightarrow IA(1)$ not equal to 1 !





Most popular format
for general sparse matrices!



$$A = \begin{pmatrix} 1. & 0. & 0. & 2. & 0. \\ 3. & 4. & 0. & 5. & 0. \\ 6. & 0. & 7. & 8. & 9. \\ 0. & 0. & 10. & 11. & 0. \\ 0. & 0. & 0. & 0. & 12. \end{pmatrix}$$

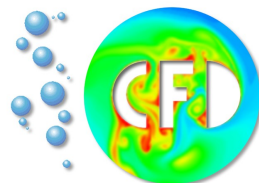
- order NZ entries columnwise:

$$AA = [1. \ 3. \ 6. \ 4. \ 7. \ 10. \ 2. \ 5. \ 8. \ 11. \ 9. \ 12.]$$

$$IA = [1 \ 2 \ 3 \ 2 \ 3 \ 4 \ 1 \ 2 \ 3 \ 4 \ 3 \ 5]$$

$$JA = [1 \ 4 \ 5 \ 7 \ 11 \ 13]$$

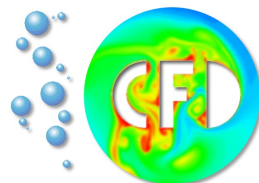
- IA = row indices
- JA = pointers (positions in AA and IA) where new columns begin; plus entry 'JA(1) + NNZ'



$$A = \begin{pmatrix} 1. & 0. & 0. & 2. & 0. \\ 3. & 4. & 0. & 5. & 0. \\ 6. & 0. & 7. & 8. & 9. \\ 0. & 0. & 10. & 11. & 0. \\ 0. & 0. & 0. & 0. & 12. \end{pmatrix}$$

AA	1.	4.	7.	11.	12.	*	2.	3.	5.	6.	8.	9.	10.
JA	7	8	10	13	14	14	4	1	4	1	4	5	3

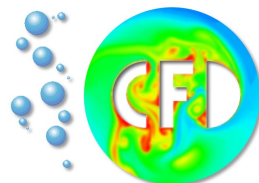
- diagonal elements usually nonzero and used more often (e.g., Jacobi) → store them separately and contiguously
- 2 arrays of length $NNZ+1$
- $AA[1:N]$: diagonal elements
 $AA[N+1]$: not used (or special information)
 $AA[N+2:NNZ+1]$: remaining entries (rowwise)



$$A = \begin{pmatrix} 1. & 0. & 0. & 2. & 0. \\ 3. & 4. & 0. & 5. & 0. \\ 6. & 0. & 7. & 8. & 9. \\ 0. & 0. & 10. & 11. & 0. \\ 0. & 0. & 0. & 0. & 12. \end{pmatrix}$$

AA	1.	4.	7.	11.	12.	*	2.	3.	5.	6.	8.	9.	10.
JA	7	8	10	13	14	14	4	1	4	1	4	5	3

- $JA[1:N+1]$: pointers to beginning of new rows
 $JA[N+2:NNZ+1]$: column indices
- $JA[N+1]$: pointer to virtual $(N+1)$ -th row
($14 = JA(1) - (N+1) + NNZ + 1$)
- $JA[N]=14$?
→ without diagonal element 12., the N -th row is zero → point to $(N+1)$ -th row



$$A = \begin{pmatrix} 1. & 0. & 2. & 0. & 0. \\ 3. & 4. & 0. & 5. & 0. \\ 0. & 6. & 7. & 0. & 8. \\ 0. & 0. & 9. & 10. & 0. \\ 0. & 0. & 0. & 11. & 12. \end{pmatrix}$$

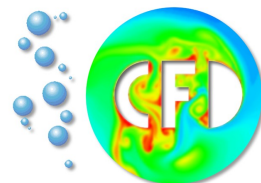
$$\text{DIAG} = \begin{array}{|c|c|c|} \hline * & 1. & 2. \\ \hline 3. & 4. & 5. \\ \hline 6. & 7. & 8. \\ \hline 9. & 10. & * \\ \hline 11 & 12. & * \\ \hline \end{array}$$

$$\text{IOFF} = \begin{array}{|c|c|c|} \hline -1 & 0 & 2 \\ \hline \end{array}$$

- all NZ entries are located inside a few (ND) diagonals
- $\text{DIAG}(1:N, 1:ND)$: stores all diagonals
- $\text{IOFF}(1:ND)$: offset from the main diagonal

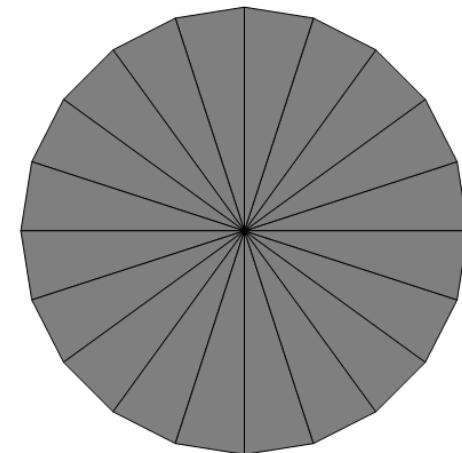
$$\text{DIAG}(i, j) \leftarrow a_{i, i+\text{ioff}(j)}$$

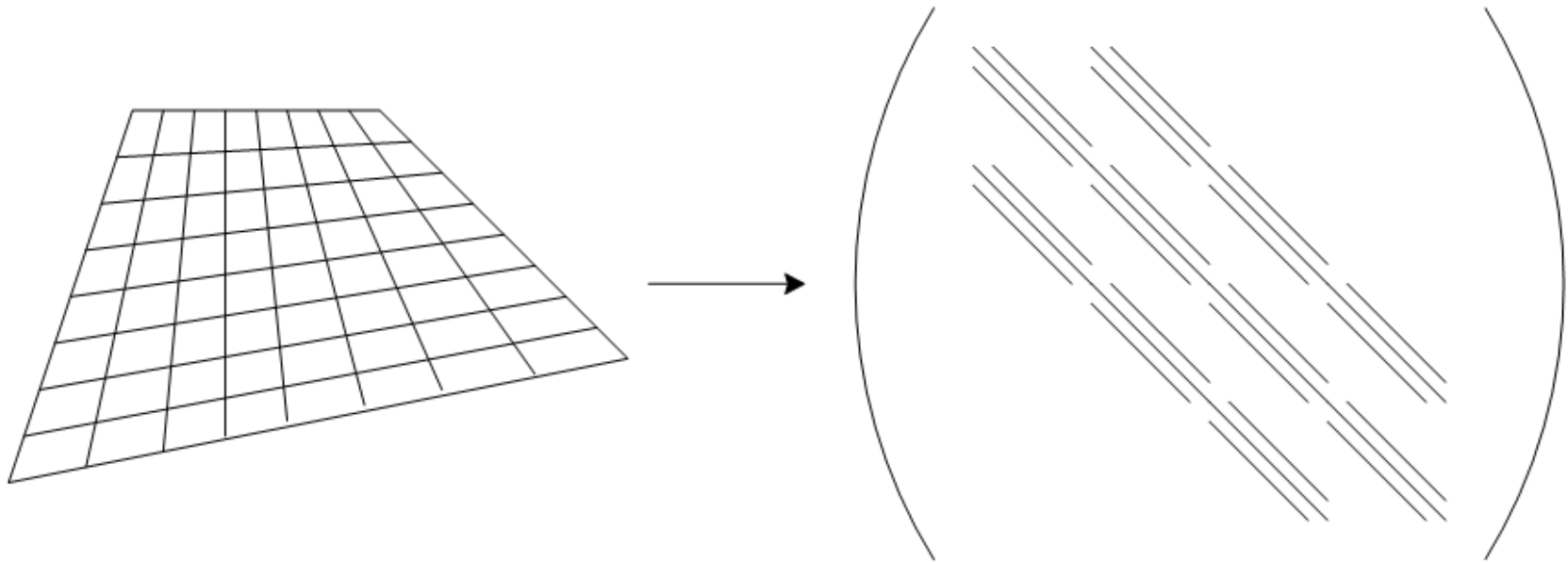
- some zeros and non-existing elements (*) are stored
- order of diagonals in DIAG unimportant, often main diagonal first



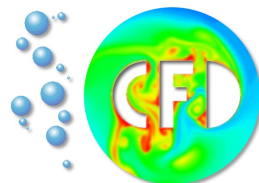
$$A = \begin{pmatrix} 1. & 0. & 2. & 0. & 0. \\ 3. & 4. & 0. & 5. & 0. \\ 0. & 6. & 7. & 0. & 8. \\ 0. & 0. & 9. & 10. & 0. \\ 0. & 0. & 0. & 11. & 12. \end{pmatrix} \quad \text{COEF} = \begin{array}{|c|c|c|} \hline 1. & 2. & 0. \\ \hline 3. & 4. & 5. \\ \hline 6. & 7. & 8. \\ \hline 9. & 10. & 0. \\ \hline 11 & 12. & 0. \\ \hline \end{array} \quad \text{JCOEF} = \begin{array}{|c|c|c|} \hline 1 & 3 & 1 \\ \hline 1 & 2 & 4 \\ \hline 2 & 3 & 5 \\ \hline 3 & 4 & 4 \\ \hline 4 & 5 & 5 \\ \hline \end{array}$$

- assumption: only few (ND) NZ elements per row
- $\text{COEF}(1:N, 1:ND)$: NZ entries per row, filled with zeros
- $\text{JCOEF}(1:N, 1:ND)$: column indices of COEF entries (choose, e.g., row index for zeros)
- similar to DIAG , but more general
- problematic: strongly varying $\#NZ$ per row (e.g., 'wheel') \rightarrow many zeros
- improvement: JAD Jagged Diagonal format (sort rows by $\#NZ$ (descending), store permutation array)





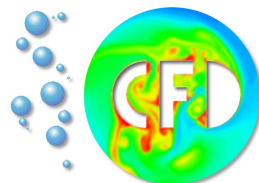
- special situation: regular tensor product grids with $N=M \times M$ vertices, rowwise numbering
- discretisation with bilinear finite elements
→ 9 matrix diagonals



```

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

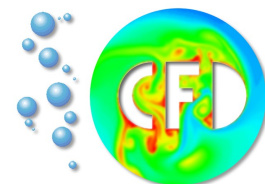
- specialised DIA format
- numbers = indices of the diagonals (main diag. = 1)
→ ordering in the array



FEAST's SparseBanded Format

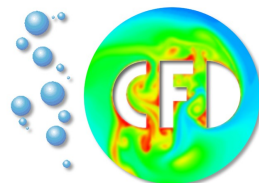
	1	2	3	4	5	6	7	8	9	0	1	2	3	4	5	6	7	8	9	0	1	2	3	4	5												
1	x	x	x		x		*	*		0	*	*																									
2		x	x	x			*	*	*		*	*	*																								
3			x	x	x		*	*	*		*	*	*																								
4				x	x	x		*	*	*		*	*	*																							
5					x	x		0		*	*	0		*	*	0																					
6						x		*	*		0	*	*		0	*	*																				
7								*	*	*		*	*	*		*	*	*																			
8									*	*	*		*	*	*		*	*	*																		
9										*	*	*		*	*	*		*	*	*																	
10											*	*	0		*	*	0		*	*	0																
11												0	*	*		0	*	*		0	*	*															
12													*	*	*		*	*	*		*	*	*														
13														*	*	*		*	*	*		*	*	*													
14															*	*	*		*	*	*		*	*	*												
15																*	*	0		*	*	0		*	*	0											
16																	0	*	*		0	*	*		0	*	*										
17																		*	*	*		*	*	*		*	*	*									
18																			*	*	*		*	*	*		*	*	*								
19																				*	*	*		*	*	*		*	*	*							
20																					*	*	0		*	*	0		*	*		x					
21																						0	*	*		0	*	*		0		x	x				
22																							*	*	*		*	*	*			x	x	x			
23																								*	*	*		*	*	*			x	x	x		
24																									*	*	*		*	*	*			x	x	x	
25																										*	*	0		*	*		x		x	x	x

- * = NZ entries
- 0 = zeros in diagonals
- x = padding with non-matrix entries (zeros)
- numbers 1-25 = vertex indices
- M sub-blocks of size M



	1	2	3	4	5	6	7	8	9	0	1	2	3	4	5	6	7	8	9	0	1	2	3	4	5
1:	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*
2:	x	x	x	x	x	x	*	*	*	*	0	*	*	*	*	0	*	*	*	*	0	*	*	*	*
3:	x	x	x	x	x	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*
4:	x	x	x	x	0	*	*	*	*	0	*	*	*	*	0	*	*	*	*	0	*	*	*	*	0
5:	x	*	*	*	*	0	*	*	*	*	0	*	*	*	*	0	*	*	*	*	0	*	*	*	*
6:	*	*	*	*	0	*	*	*	*	0	*	*	*	*	0	*	*	*	*	0	*	*	*	*	x
7:	0	*	*	*	*	0	*	*	*	*	0	*	*	*	*	0	*	*	*	*	0	x	x	x	x
8:	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	x	x	x	x	x
9:	*	*	*	*	0	*	*	*	*	0	*	*	*	*	0	*	*	*	*	x	x	x	x	x	x

- actually stored in one 1D array of length $9*N$



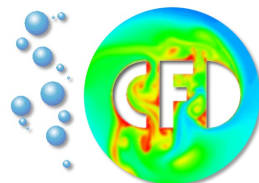
$$A = \begin{pmatrix} 1. & 0. & 0. & 2. & 0. \\ 3. & 4. & 0. & 5. & 0. \\ 6. & 0. & 7. & 8. & 9. \\ 0. & 0. & 10. & 11. & 0. \\ 0. & 0. & 0. & 0. & 12. \end{pmatrix}$$

AA	1. 2. 3. 4. 5. 6. 7. 8. 9. 10. 11. 12.
JA	1 4 1 2 4 1 3 4 5 3 4 5
IA	1 3 6 10 12 13

```

DO 10 I=1, N
    K1 = IA(I)
    K2 = IA(I+1)-1
    Y(I) = DOTPRODUCT( A(K1:K2) , X(JA(K1:K2)) )
10 CONTINUE
    
```

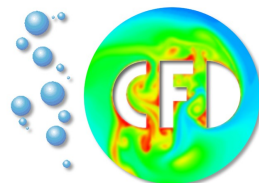
- explains the necessity of IA (N+1)
- 2*NNZ FLOPs (compared to 2N² for dense matrix vector mult.)
- critical disadvantages:
 - indirect addressing in X (JA (K1 :K2)) , elements not contiguous in memory
 - X entries have to be loaded several times
- advantage: Y and A entries only loaded/stored once



$$A = \begin{pmatrix} 1. & 0. & 2. & 0. & 0. \\ 3. & 4. & 0. & 5. & 0. \\ 0. & 6. & 7. & 0. & 8. \\ 0. & 0. & 9. & 10. & 0. \\ 0. & 0. & 0. & 11. & 12. \end{pmatrix} \quad \text{COEF} = \begin{array}{|c|c|c|} \hline 1. & 2. & 0. \\ \hline 3. & 4. & 5. \\ \hline 6. & 7. & 8. \\ \hline 9. & 10. & 0. \\ \hline 11 & 12. & 0. \\ \hline \end{array} \quad \text{JCOEF} = \begin{array}{|c|c|c|} \hline 1 & 3 & 1 \\ \hline 1 & 2 & 4 \\ \hline 2 & 3 & 5 \\ \hline 3 & 4 & 4 \\ \hline 4 & 5 & 5 \\ \hline \end{array}$$

```
      DO 10 J=1, JMAX
        DO 20 I=1, N
          Y(I) = Y(I) + COEFF(I,J)*X(JCOEFF(I,J))
20      CONTINUE
10     CONTINUE
```

- JMAX: max. NZ entries per row
- indirect addressing in X (JCOEFF (I , J))
- Y entries have to be loaded/stored JMAX times



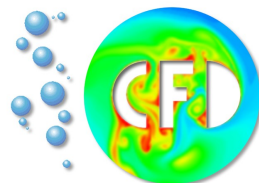
$$A = \begin{pmatrix} 1. & 0. & 2. & 0. & 0. \\ 3. & 4. & 0. & 5. & 0. \\ 0. & 6. & 7. & 0. & 8. \\ 0. & 0. & 9. & 10. & 0. \\ 0. & 0. & 0. & 11. & 12. \end{pmatrix}$$

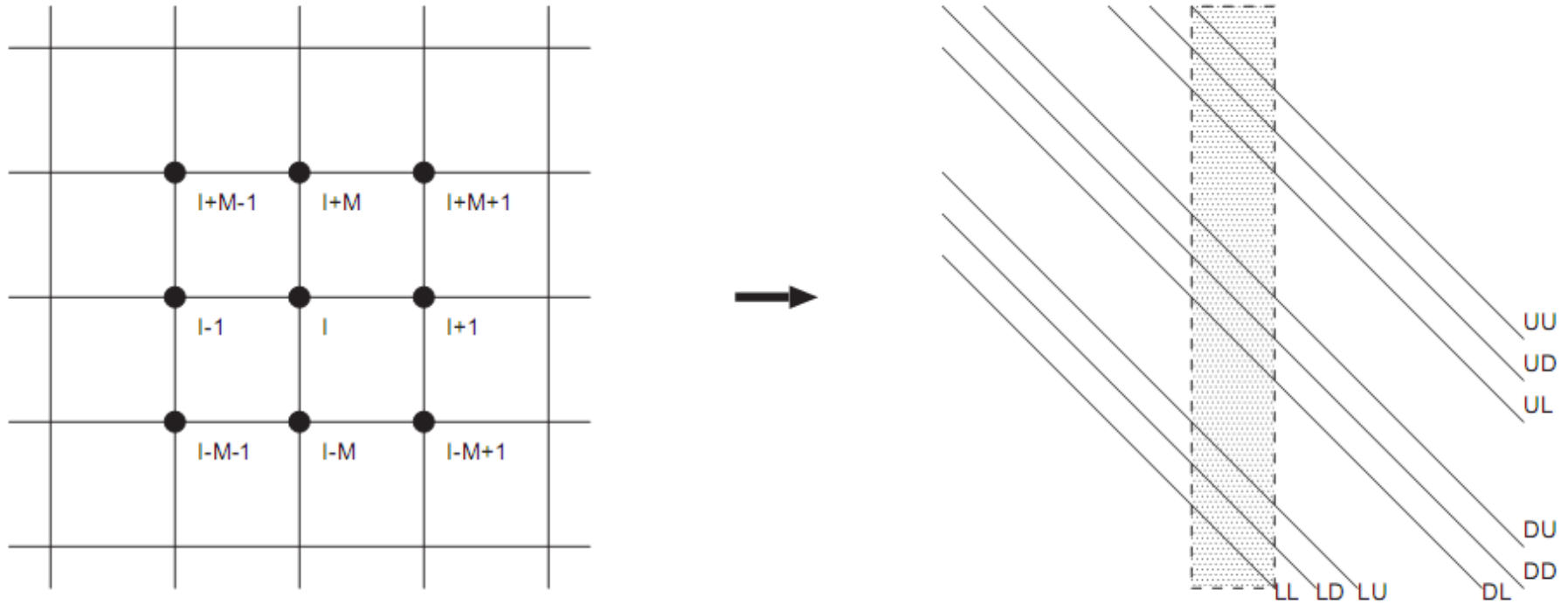
$$\text{DIAG} = \begin{array}{|c|c|c|} \hline * & 1. & 2. \\ \hline 3. & 4. & 5. \\ \hline 6. & 7. & 8. \\ \hline 9. & 10. & * \\ \hline 11 & 12. & * \\ \hline \end{array}$$

$$\text{IOFF} = \begin{array}{|c|c|c|} \hline -1 & 0 & 2 \\ \hline \end{array}$$

```
DO 10 J=1, NDIAG
    JOFF = IOFF(J)
    DO 20 I=1, N
        Y(I) = Y(I) + DIAG(I,J)*X(JOFF+I)
20    CONTINUE
10    CONTINUE
```

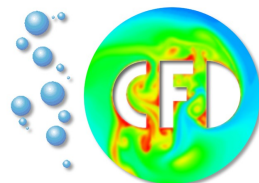
- decisive: no indirect addressing, X (JOFF+1 : JOFF+n) can be loaded contiguously
- Y-entries have to be loaded/stored NDIAG times
- X-entries have to be loaded several times





- special situation: regular tensor product grids with $N=M \times M$ vertices, rowwise numbering
- discretisation with bilinear finite elements \rightarrow 9 matrix diagonals from left to right:

LL LD LU DL DD DU UL UD UU
 (L = lower, U = upper, D = central diagonal)



```

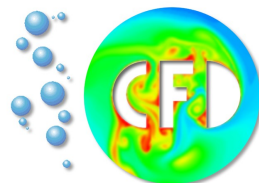
DO 10 I=1,N
10   Y(I)=Y(I)+DD(I)*X(I)
DO 20 I=1,N-1
20   Y(I)=Y(I)+DL(I)*X(I-1)
DO 30 I=1,N-1
30   Y(I)=Y(I)+DU(I)*X(I+1)

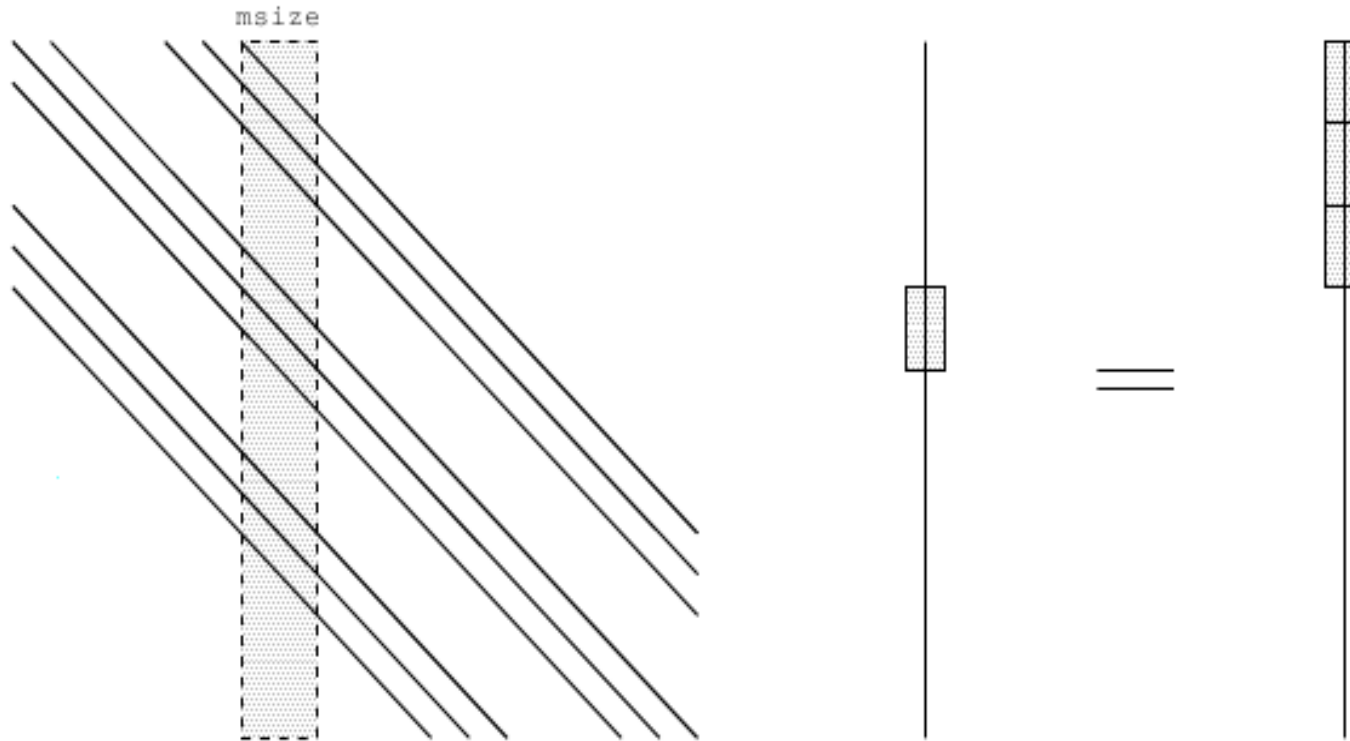
DO 40 I=1,N-M
40   Y(I)=Y(I)+UD(I)*X(I-M)
DO 50 I=1,N-1-M
50   Y(I)=Y(I)+UL(I)*X(I-1-M)
DO 60 I=1,N-1-M
60   Y(I)=Y(I)+UU(I)*X(I+1-M)

DO 70 I=1,N-M
70   Y(I)=Y(I)+UD(I)*X(I+M)
DO 80 I=1,N-1-M
80   Y(I)=Y(I)+UL(I)*X(I-1+M)
DO 90 I=1,N-1-M
90   Y(I)=Y(I)+UU(I)*X(I+1+M)

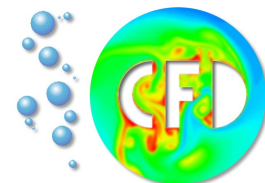
DO 10 J=1, NDIAG
    JOFF = IOFF(J)
DO 20 I=1, N
    Y(I) = Y(I) + DIAG(I,J)*X(JOFF+I)
20   CONTINUE
10   CONTINUE
    
```

- same algorithm as MV multiply in DIA format
- specialised for TP matrix
- same disadvantages (several loads (stores) for X and Y)
- improvement: blocking (next slide)





- idea: treat diagonals partially (loop blocking/loop fusion)
- consider a 'window' of width $K \cdot M$
(in figure: `msize`) $\rightarrow N / (K \cdot M)$ windows
- E.g., $K=1 \rightarrow N / (1 \cdot M) = M$ windows



```

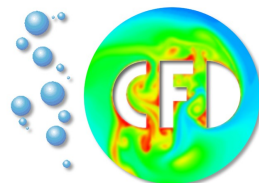
DO 10 IM=0,N/(M*K)-1
  DO 100, I=IM*K*M+1,IM*K*M+K*M
100    Y(I)=Y(I)+DD(I)*X(I)+DL(I)*X(I-1)+DU(I)*X(I+1)
  DO 200, I=IM*K*M+1,IM*K*M+K*M
200    Y(I-M)=Y(I-M)+LD(I)*X(I)+LL(I)*X(I-1)+LU(I)*X(I+1)
  DO 300, I=IM*K*M+1,IM*K*M+K*M
300    Y(I+M)=Y(I)+UD(I)*X(I)+UL(I)*X(I-1)+UU(I)*X(I+1)
10    CONTINUE

```

- outer loop: 'move' window from left to right
- three inner loops: separate treatment of the 3 diag. groups

$$L[L, D, U] \quad D[L, D, U] \quad U[L, D, U]$$

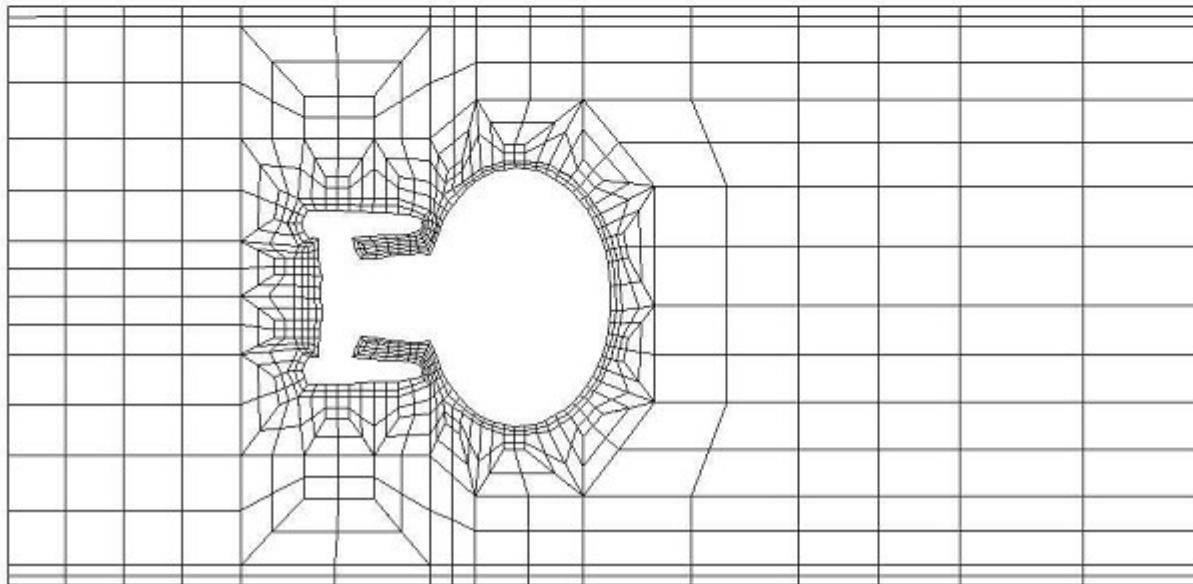
- decisive advantage:
corresponding part of vector X can be kept in the cache
- K free parameter, can be adapted to the hardware



Local MFLOP/s rates:

#DOF	MV (Sparse)	MV (SparseBanded)	
		var	const
65^2	422	1111	1605
257^2	106	380	1214
1025^2	54	362	1140

Sun V40z 'Opteron' (1800 MHz, peak perf. \approx 2900 MFLOP/s)



C. Becker, 2006

