

# ADVANCED ARCHITECTURES

## *INTENSIVE COMPUTATION*

---

### **Sparse matrices**

**Annalisa Massini**

2023-2024

*Lecture 13*

# COMPACT STORAGE FORMAT

---

Most of the material is from:

L. Formaggia, F. Saleri, A. Veneziani **Solving Numerical PDEs: Problems, Applications, Exercises** - Appendix *The treatment of sparse matrices*

BSR format from:

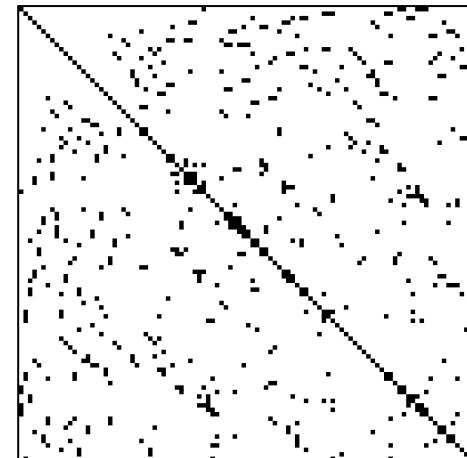
<https://software.intel.com/en-us/mkl-developer-reference-c-sparse-blas-bsr-matrix-storage-format>

# Storage Methods for Sparse Matrix

- A matrix is **sparse** if it contains a large number of zeros
- **sparsity** of the matrix =  
number of zero-valued elements / total number of elements
- **density** =  $1 - \text{sparsity}$
- A matrix is **sparse** if its sparsity is  $> 0.5$
- **But** *sparsity is interesting* if in a matrix of size  $n \times n$   
→ the number of non-zero entries is  $O(n)$
- This means that the average number of non-zero entries in each row is bounded independently from  $n$
- A non-sparse matrix is said **full** or **dense** if the number of non-zero elements is  $O(n^2)$

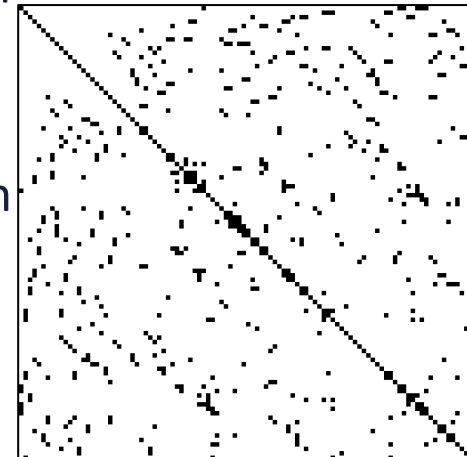
# Storage Methods for Sparse Matrix

- If the location of the zero elements is known a-priori, we can avoid reserving storage for them
- The distribution of non-zero elements of a sparse matrix may be described by:
  - the **sparsity pattern**, defined as the set  $\{(i, j): A_{ij} \neq 0\}$
  - the matrix graph, where nodes  $i$  and  $j$  are connected by an edge if and only if  $A_{ij} \neq 0$
- In order to take advantage of the large number of zero elements, **special schemes** are required to store sparse matrices



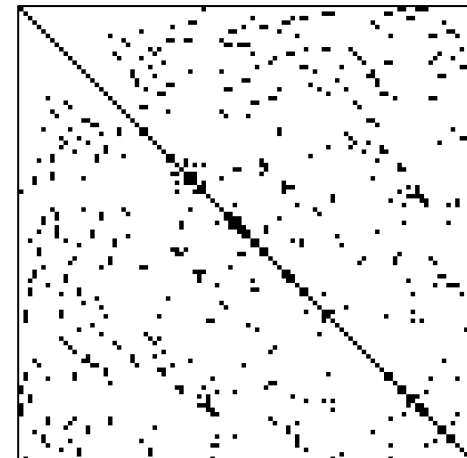
# Storage Methods for Sparse Matrix

- The use of **adequate storage techniques** for sparse matrices is fundamental, especially with large-scale problems
- Large sparse matrices often appear in scientific or engineering applications when solving partial differential equations
- Example
  - Suppose we want to solve the Navier-Stokes equations on **a two-dimensional grid formed by 10.000 vertexes** with finite elements
  - The number of degrees of freedom is around  $10^5$  for the pressure and  $4 \times 10^5$  for each component of the velocity
  - The **associated matrix** will then be **90000×90000**
  - If we store all  $8.1 \times 10^9$  coefficients, using double precision (8 bytes), around **60 Gigabytes** are necessary!
  - ***This is too much and memory handle is very inefficient***



# Storage Methods for Sparse Matrix

- In case of a **three-dimensional problem** the situation becomes even **worse**, since the number of degrees of freedom grows very rapidly as the grid gets finer
  - Nowadays it is common to deal with millions of degrees of freedom
- Therefore to store **sparse matrices** efficiently we need **data formats** that are **more compact** than the classical array



# Storage Methods for Sparse Matrix

- The adoption of **sparse formats** may affect the ***speed of certain operations***
- For example, with a sparse format we **cannot access** or search for a **particular element** (or group of elements) directly, using the two indexes  $i$  and  $j$  to determine where entry  $A_{ij}$  is located in the memory
- On the other hand, even if the operation of accessing an entry of a matrix in sparse format turns out to be less efficient, by adopting a **sparse format** we will nevertheless **access only nonzero elements, thus executing only useful operations**

# Storage Methods for Sparse Matrix

- Hence, in general, the **sparse format is preferable** in terms of **storage** as well as in terms of **computing time**, as long as the matrix is **sufficiently sparse**
- The main goal of sparse formats is:
  - to **represent only the nonzero elements**
  - to be able **to efficiently perform the common matrix operations**



# Storage Methods for Sparse Matrix

- We can distinguish different kinds of **operations** on a matrix
- The most important operations are:
  1. **accessing** a **generic element** (random access)
  2. **accessing** the elements of a **whole row**: important when multiplying a matrix by a vector
  3. **accessing** the elements of a **whole column**, or equivalently, of a row in the transpose matrix (relevant for operations such as symmetrizing the matrix after imposing Dirichlet conditions)
  4. **adding a new element** to the matrix pattern: this is a critical issue if the pattern is not known beforehand or it can change throughout the computations
  5. And, **multiplying a matrix and a vector** that is a very **common** intermediate **operation** used in many numerical methods

# Storage Methods for Sparse Matrix

- It is important to characterize formats for sparse matrices by the **computational cost of these operations** and by how the latter depends on the **matrix size**
- **Different formats** for sparse matrices exist due to the fact that there is *no format that is simultaneously optimal for all the above operations*, and at the same time *efficient in terms of storage capacity*

# Storage Methods for Sparse Matrix

- In the following:
  - $n$  is the matrix' size
  - $nz$  is the number of non-zero entries
  - We adopt the convention of **indexing** entries of matrices and vectors (arrays) **starting from 1** (as in Matlab)
  - $A_{ij}$  will denote the entry of the matrix A on **row  $i$  and column  $j$**
- To estimate how much memory the matrix occupies we assume that:
  - an **integer** occupies **4 bytes**
  - a **real number** (floating point repres.) **8 bytes** (double precision)
  - For example, storing a square matrix having  $n = 12$  would require  $12 \times 12 \times 8 = 1152$  bytes

# The Coordinate format: COO format

- The simplest storage scheme for sparse matrices is the format by **coordinate**
- The data structure consists of three arrays:
  - **A** - a real array containing all the real (or complex) **values of the nonzero elements** in **any order**
  - **I** - an integer array containing their **row indices**
  - **J** - a second integer array containing their **column indices**
  - I, J and A all have ***nz* elements**, as many as the number of non-zero elements of the matrix

# The Coordinate format: COO format

## Example

The matrix A

```

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.

```

is represented (*for example*) by

```

A  12. 9.  7.  5.  1.  2. 11. 3.  6.  4.  8. 10.
I   5   3   3   2   1   1   4   2   3   2   3   4
J   5   5   3   4   1   4   4   1   1   2   4   3

```

- Notice that elements are listed in an *arbitrary order*

# The Coordinate format: COO format

**Example** -  $n=12$  and  $nz=58$

|     |     |     |     |     |     |     |     |     |     |     |     |
|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|
| 101 | 102 | 0   | 103 | 0   | 0   | 0   | 0   | 0   | 0   | 0   | 0   |
| 104 | 105 | 106 | 107 | 108 | 0   | 0   | 0   | 0   | 0   | 0   | 0   |
| 0   | 109 | 110 | 0   | 111 | 112 | 0   | 0   | 0   | 0   | 0   | 0   |
| 113 | 114 | 0   | 115 | 116 | 0   | 117 | 0   | 0   | 0   | 0   | 0   |
| 0   | 118 | 119 | 120 | 121 | 122 | 123 | 124 | 0   | 0   | 0   | 0   |
| 0   | 0   | 125 | 0   | 126 | 127 | 0   | 128 | 129 | 0   | 0   | 0   |
| 0   | 0   | 0   | 130 | 131 | 0   | 132 | 133 | 0   | 134 | 0   | 0   |
| 0   | 0   | 0   | 0   | 135 | 136 | 137 | 138 | 139 | 140 | 141 | 0   |
| 0   | 0   | 0   | 0   | 0   | 142 | 0   | 143 | 144 | 0   | 145 | 146 |
| 0   | 0   | 0   | 0   | 0   | 0   | 147 | 148 | 0   | 149 | 150 | 0   |
| 0   | 0   | 0   | 0   | 0   | 0   | 0   | 151 | 152 | 153 | 154 | 155 |
| 0   | 0   | 0   | 0   | 0   | 0   | 0   | 0   | 156 | 0   | 157 | 158 |

The space occupied is:

$$\begin{aligned}
 & - 8 \times n \times n = \\
 & = 8 \times 12 \times 12 = \mathbf{1152} \text{ bytes}
 \end{aligned}$$

**in full format**

# The Coordinate format: COO format

**Example** -  $n=12$  and  $nz=58$

|     |     |     |     |     |     |     |     |     |     |     |     |
|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|
| 101 | 102 | 0   | 103 | 0   | 0   | 0   | 0   | 0   | 0   | 0   | 0   |
| 104 | 105 | 106 | 107 | 108 | 0   | 0   | 0   | 0   | 0   | 0   | 0   |
| 0   | 109 | 110 | 0   | 111 | 112 | 0   | 0   | 0   | 0   | 0   | 0   |
| 113 | 114 | 0   | 115 | 116 | 0   | 117 | 0   | 0   | 0   | 0   | 0   |
| 0   | 118 | 119 | 120 | 121 | 122 | 123 | 124 | 0   | 0   | 0   | 0   |
| 0   | 0   | 125 | 0   | 126 | 127 | 0   | 128 | 129 | 0   | 0   | 0   |
| 0   | 0   | 0   | 130 | 131 | 0   | 132 | 133 | 0   | 134 | 0   | 0   |
| 0   | 0   | 0   | 0   | 135 | 136 | 137 | 138 | 139 | 140 | 141 | 0   |
| 0   | 0   | 0   | 0   | 0   | 142 | 0   | 143 | 144 | 0   | 145 | 146 |
| 0   | 0   | 0   | 0   | 0   | 0   | 147 | 148 | 0   | 149 | 150 | 0   |
| 0   | 0   | 0   | 0   | 0   | 0   | 0   | 151 | 152 | 153 | 154 | 155 |
| 0   | 0   | 0   | 0   | 0   | 0   | 0   | 0   | 156 | 0   | 157 | 158 |

The space occupied is:

-  $8 \times n \times n = 1152$  bytes  
in full format

-  $(4+4+8) \times nz = 16 \times 58 =$   
 $= 928$  bytes  
in COO format

|     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |
|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|
| 101 | 102 | 103 | 104 | 105 | 106 | 107 | 108 | 109 | 110 | 111 | 112 | 113 | 114 | 115 | 116 | 117 | 118 | 119 | 120 | ... |
| ... | 121 | 122 | 123 | 124 | 125 | 126 | 127 | 128 | 129 | 130 | 131 | 132 | 133 | 134 | 135 | 136 | 137 | 138 | 139 | ... |
| ... | 140 | 141 | 142 | 143 | 144 | 145 | 146 | 147 | 148 | 149 | 150 | 151 | 152 | 153 | 154 | 155 | 156 | 157 | 158 |     |
| 1   | 1   | 1   | 2   | 2   | 2   | 2   | 2   | 3   | 3   | 3   | 3   | 4   | 4   | 4   | 4   | 4   | 5   | 5   | 5   | ... |
| ... | 5   | 5   | 5   | 5   | 6   | 6   | 6   | 6   | 6   | 7   | 7   | 7   | 7   | 7   | 7   | 8   | 8   | 8   | 8   | ... |
| ... | 8   | 8   | 9   | 9   | 9   | 9   | 9   | 9   | 10  | 10  | 10  | 10  | 10  | 11  | 11  | 11  | 11  | 11  | 12  | 12  |
| 1   | 2   | 4   | 1   | 2   | 3   | 4   | 5   | 2   | 3   | 5   | 6   | 1   | 2   | 4   | 5   | 7   | 2   | 3   | 4   | ... |
| ... | 5   | 6   | 7   | 8   | 3   | 5   | 6   | 8   | 9   | 4   | 5   | 7   | 8   | 10  | 5   | 6   | 7   | 8   | 9   | ... |
| ... | 10  | 11  | 6   | 8   | 9   | 11  | 12  | 7   | 8   | 10  | 11  | 8   | 9   | 10  | 11  | 12  | 9   | 11  | 12  |     |

# The Coordinate format: COO format

- COO format *does not guarantee rapid access to an element, nor to rows or columns*
- **Finding the generic element** of the matrix from the row and column indexes normally requires a number of operations proportional to  $nz$
- In fact, it is necessary to go through all elements of I and J until one hits those indexes, using **expensive comparison operations**
- Using specific techniques to store the indexes in special search data structure, it is possible to reduce the cost to  $O(\log_2(nz))$ , but at a higher storing price



# The Coordinate format: COO format

- The operation of **multiplying** a matrix  $A$  and a vector  $\mathbf{x}$  can be done directly, by running through the elements of the three arrays

- A possible code for the product  $\mathbf{y} = \mathbf{A}\mathbf{x}$  using MATLAB

```
y=zeros(nz,1);
```

```
for k=1:nz
```

```
    i=I(k);
```

```
    j=J(k);
```

```
    y(i)= y(i) + A(k)*x(j); % notice the use of i and j
```

```
end
```

|     |     |     |     |     |     |     |     |     |     |     |     |
|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|
| 101 | 102 | 0   | 103 | 0   | 0   | 0   | 0   | 0   | 0   | 0   | 0   |
| 104 | 105 | 106 | 107 | 108 | 0   | 0   | 0   | 0   | 0   | 0   | 0   |
| 0   | 109 | 110 | 0   | 111 | 112 | 0   | 0   | 0   | 0   | 0   | 0   |
| 113 | 114 | 0   | 115 | 116 | 0   | 117 | 0   | 0   | 0   | 0   | 0   |
| 0   | 118 | 119 | 120 | 121 | 122 | 123 | 124 | 0   | 0   | 0   | 0   |
| 0   | 0   | 125 | 0   | 126 | 127 | 0   | 128 | 129 | 0   | 0   | 0   |
| 0   | 0   | 0   | 130 | 131 | 0   | 132 | 133 | 0   | 134 | 0   | 0   |
| 0   | 0   | 0   | 0   | 135 | 136 | 137 | 138 | 139 | 140 | 141 | 0   |
| 0   | 0   | 0   | 0   | 0   | 142 | 0   | 143 | 144 | 0   | 145 | 146 |
| 0   | 0   | 0   | 0   | 0   | 0   | 147 | 148 | 0   | 149 | 150 | 0   |
| 0   | 0   | 0   | 0   | 0   | 0   | 0   | 151 | 152 | 153 | 154 | 155 |
| 0   | 0   | 0   | 0   | 0   | 0   | 0   | 0   | 156 | 0   | 157 | 158 |

|     |     |     |     |     |     |     |     |     |     |     |     |
|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|
| 101 | 102 | 103 | 104 | 105 | 106 | 107 | 108 | 109 | 110 | 111 | ... |
| 1   | 1   | 1   | 2   | 2   | 2   | 2   | 2   | 3   | 3   | 3   | ... |
| 1   | 2   | 4   | 1   | 2   | 3   | 4   | 5   | 2   | 3   | 5   | ... |

# The Coordinate format: COO format

## Observations

- The additional cost of this operation (compared to the analogue for a full matrix) depends essentially on **indirect addressing**:
  - accessing  $\mathbf{y}(\mathbf{i})$  requires first of all to access  $\mathbf{I}(\mathbf{k})$
- The **access and update** of arrays  $\mathbf{x}$  and  $\mathbf{y}$  does **not proceed by consecutive elements**  $\rightarrow$  the possibility of optimizing the use of the processor's cache is greatly reduced

# The Coordinate format: COO format

## Observations

- **Operations** are performed **only on non-zero** elements and in general we have  $nz \ll n^2$
- An advantage of this format is that:
  - It is **easy to add a new** element to the matrix
  - In fact, it is enough to add a new entry to the arrays I, J and A
- For this reason, COO is often used when the **pattern is not known** a priori

# The *skyline* format

- The format called **skyline** is among the first used to store matrices arising from the method of finite elements
- The idea is to store the area formed by the **elements between the first and last non-zero coefficient**, on **each row**

|     |     |     |     |     |     |     |     |     |     |     |     |
|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|
| 101 | 102 | 0   | 103 | 0   | 0   | 0   | 0   | 0   | 0   | 0   | 0   |
| 104 | 105 | 106 | 107 | 108 | 0   | 0   | 0   | 0   | 0   | 0   | 0   |
| 0   | 109 | 110 | 0   | 111 | 112 | 0   | 0   | 0   | 0   | 0   | 0   |
| 113 | 114 | 0   | 115 | 116 | 0   | 117 | 0   | 0   | 0   | 0   | 0   |
| 0   | 118 | 119 | 120 | 121 | 122 | 123 | 124 | 0   | 0   | 0   | 0   |
| 0   | 0   | 125 | 0   | 126 | 127 | 0   | 128 | 129 | 0   | 0   | 0   |
| 0   | 0   | 0   | 130 | 131 | 0   | 132 | 133 | 0   | 134 | 0   | 0   |
| 0   | 0   | 0   | 0   | 135 | 136 | 137 | 138 | 139 | 140 | 141 | 0   |
| 0   | 0   | 0   | 0   | 0   | 142 | 0   | 143 | 144 | 0   | 145 | 146 |
| 0   | 0   | 0   | 0   | 0   | 0   | 147 | 148 | 0   | 149 | 150 | 0   |
| 0   | 0   | 0   | 0   | 0   | 0   | 0   | 151 | 152 | 153 | 154 | 155 |
| 0   | 0   | 0   | 0   | 0   | 0   | 0   | 0   | 156 | 0   | 157 | 158 |

- This forces to **store some null entries**

# The *skyline* format

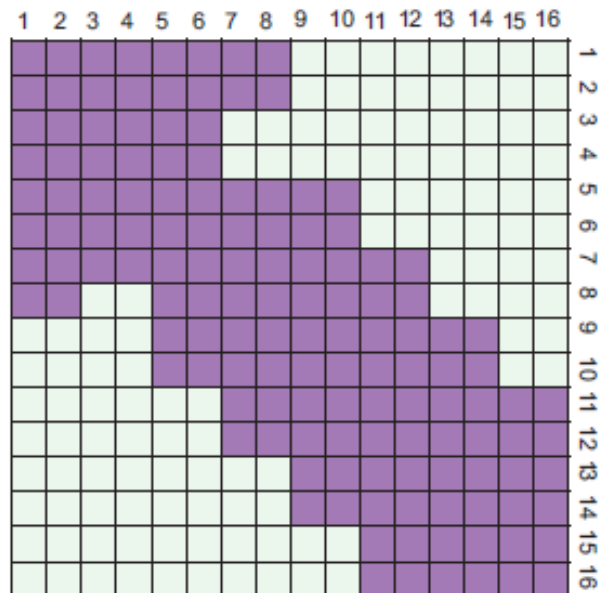
- This extra cost will be small if the matrix has non-zero entries clustered around the diagonal

|     |     |     |     |     |     |     |     |     |     |     |     |
|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|
| 101 | 102 | 0   | 103 | 0   | 0   | 0   | 0   | 0   | 0   | 0   | 0   |
| 104 | 105 | 106 | 107 | 108 | 0   | 0   | 0   | 0   | 0   | 0   | 0   |
| 0   | 109 | 110 | 0   | 111 | 112 | 0   | 0   | 0   | 0   | 0   | 0   |
| 113 | 114 | 0   | 115 | 116 | 0   | 117 | 0   | 0   | 0   | 0   | 0   |
| 0   | 118 | 119 | 120 | 121 | 122 | 123 | 124 | 0   | 0   | 0   | 0   |
| 0   | 0   | 125 | 0   | 126 | 127 | 0   | 128 | 129 | 0   | 0   | 0   |
| 0   | 0   | 0   | 130 | 131 | 0   | 132 | 133 | 0   | 134 | 0   | 0   |
| 0   | 0   | 0   | 0   | 135 | 136 | 137 | 138 | 139 | 140 | 141 | 0   |
| 0   | 0   | 0   | 0   | 0   | 142 | 0   | 143 | 144 | 0   | 145 | 146 |
| 0   | 0   | 0   | 0   | 0   | 0   | 147 | 148 | 0   | 149 | 150 | 0   |
| 0   | 0   | 0   | 0   | 0   | 0   | 0   | 151 | 152 | 153 | 154 | 155 |
| 0   | 0   | 0   | 0   | 0   | 0   | 0   | 0   | 156 | 0   | 157 | 158 |

- Indeed, algorithms have been developed to cluster non zero elements by permuting the rows and columns of the matrix (see, for example, the Cuthill-McKee algorithm)

# Skyline for symmetric matrices

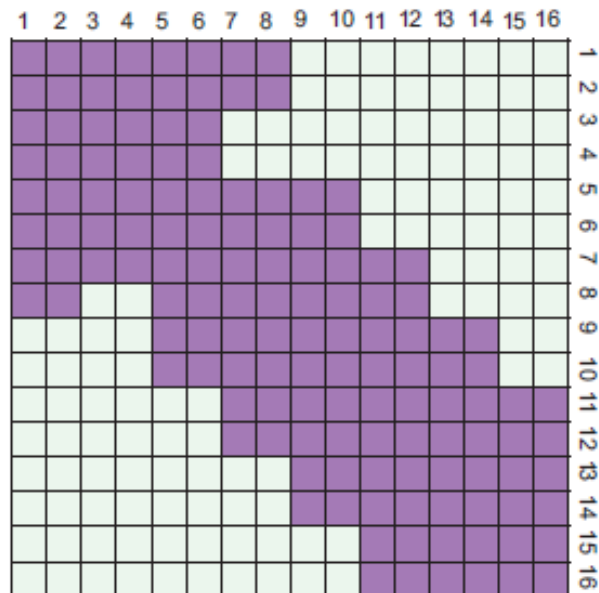
- If a matrix is **symmetric** we can store only:
  - Its **lower triangular part** (diagonal included)
  - Or we can store the **diagonal on an auxiliary array** and treat the **off-diagonal entries** separately, having the advantage of allowing the *direct access to the diagonal elements*



# Skyline for symmetric matrices

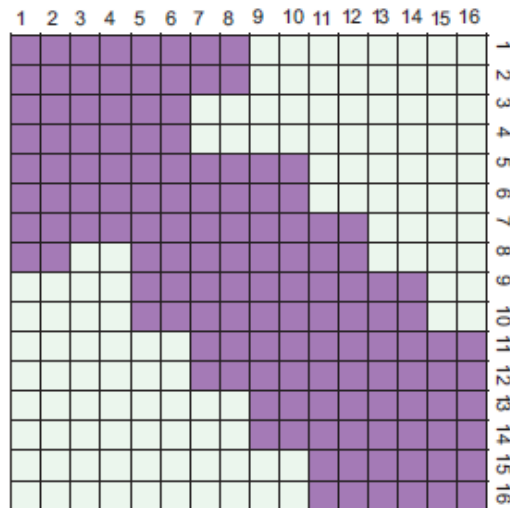
The **skyline format** with diagonal array is given by:

- **D** - real array storing **diagonal entries**
- **AL** - real array storing all **skyline elements row-wise** (except the diagonal)  
This can clearly include null coefficients
- **I** - integer array storing **pointers to rows** of matrix A
  - The  $k$ th component of array I points to the first element of row  $(k + 1)$  in AL



# Skyline for symmetric matrices

- All elements of AL from position  $I(k)$  to  $I(k+1) - 1$  are the off-diagonal elements belonging to row  $k + 1$ , in column order
- Notice that:
  - *The first row is not stored*, since it only has the diagonal element
  - $I(k)$  points to the first non-zero element on the  $(k+1)$ -th row
  - The difference  $I(k+1) - I(k)$  gives the number of the off-diagonal elements on row  $k + 1$  belonging to the skyline





# Skyline for symmetric matrices

## Example

- We want to store the **symmetric matrix** obtained from the lower triangular part of matrix A (seen before) **using the skyline format**
- This matrix can be obtained with the Matlab instruction `tril(A) + tril(A, -1)'`

|     |     |     |     |     |     |     |     |     |     |     |     |
|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|
| 101 | 104 | 0   | 113 | 0   | 0   | 0   | 0   | 0   | 0   | 0   | 0   |
| 104 | 105 | 109 | 114 | 118 | 0   | 0   | 0   | 0   | 0   | 0   | 0   |
| 0   | 109 | 110 | 0   | 119 | 125 | 0   | 0   | 0   | 0   | 0   | 0   |
| 113 | 114 | 0   | 115 | 120 | 0   | 130 | 0   | 0   | 0   | 0   | 0   |
| 0   | 118 | 119 | 120 | 121 | 126 | 131 | 135 | 0   | 0   | 0   | 0   |
| 0   | 0   | 125 | 0   | 126 | 127 | 0   | 136 | 142 | 0   | 0   | 0   |
| 0   | 0   | 0   | 130 | 131 | 0   | 132 | 137 | 0   | 147 | 0   | 0   |
| 0   | 0   | 0   | 0   | 135 | 136 | 137 | 138 | 143 | 148 | 151 | 0   |
| 0   | 0   | 0   | 0   | 0   | 142 | 0   | 143 | 144 | 0   | 152 | 156 |
| 0   | 0   | 0   | 0   | 0   | 0   | 147 | 148 | 0   | 149 | 153 | 0   |
| 0   | 0   | 0   | 0   | 0   | 0   | 0   | 151 | 152 | 153 | 154 | 157 |
| 0   | 0   | 0   | 0   | 0   | 0   | 0   | 0   | 156 | 0   | 157 | 158 |

# Skyline for symmetric matrices

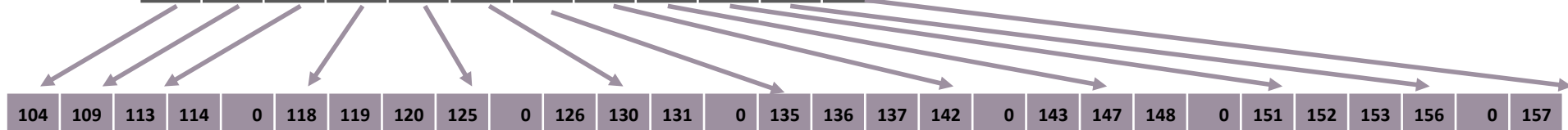
## Example

- Diagonal **D**

101 105 110 115 120 121 127 132 138 144 149 154 158

- Pointer **I** and lower skyline elements **AL**

1 2 3 6 9 12 15 18 21 24 27 30



|     |     |     |     |     |     |     |     |     |     |     |     |
|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|
| 101 | 104 | 0   | 113 | 0   | 0   | 0   | 0   | 0   | 0   | 0   | 0   |
| 104 | 105 | 109 | 114 | 118 | 0   | 0   | 0   | 0   | 0   | 0   | 0   |
| 0   | 109 | 110 | 0   | 119 | 125 | 0   | 0   | 0   | 0   | 0   | 0   |
| 113 | 114 | 0   | 115 | 120 | 0   | 130 | 0   | 0   | 0   | 0   | 0   |
| 0   | 118 | 119 | 120 | 121 | 126 | 131 | 135 | 0   | 0   | 0   | 0   |
| 0   | 0   | 125 | 0   | 126 | 127 | 0   | 136 | 142 | 0   | 0   | 0   |
| 0   | 0   | 0   | 130 | 131 | 0   | 132 | 137 | 0   | 147 | 0   | 0   |
| 0   | 0   | 0   | 0   | 135 | 136 | 137 | 138 | 143 | 148 | 151 | 0   |
| 0   | 0   | 0   | 0   | 0   | 142 | 0   | 143 | 144 | 0   | 152 | 156 |
| 0   | 0   | 0   | 0   | 0   | 0   | 147 | 148 | 0   | 149 | 153 | 0   |
| 0   | 0   | 0   | 0   | 0   | 0   | 0   | 151 | 152 | 153 | 154 | 157 |
| 0   | 0   | 0   | 0   | 0   | 0   | 0   | 0   | 156 | 0   | 157 | 158 |

# Skyline for symmetric matrices

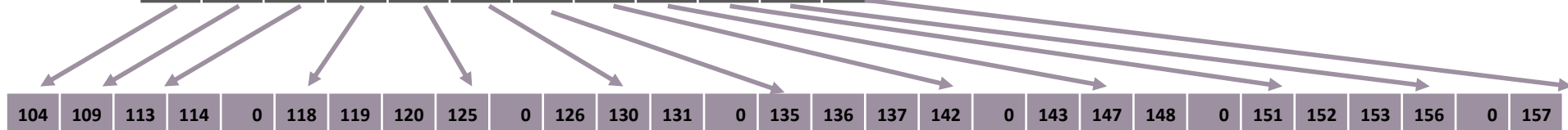
## Example

- Diagonal **D**

101 105 110 115 120 121 127 132 138 144 149 154 158

- Pointers **I** and lower skyline elements **AL**

1 2 3 6 9 12 15 18 21 24 27 30



- Note that in the  $n$ th place of the array **I** we have left a pointer at the beginning of an hypothetical position. In this way:
  - We can compute the number of skyline elements in the last row, that is  $I(n) - I(n-1)$
  - $I(n) - 1$  is the total number of elements in the skyline

# Skyline for symmetric matrices

The product  $\mathbf{y} = \mathbf{A}\mathbf{x}$  following MATLAB syntax is given by:

```

y=D.*x;
for k=2:n
    nex = I(k)-I(k-1);
    ik = I(k-1):I(k)-1;
    jcol= k-nex:k-1;
    y(k) = y(k)+dot(AL(ik),x(jcol));
    y(jcol)= y(jcol)+AL(ik)*x(k);
end

```

- We *operate symmetrically on rows and columns* to exploit the fact that only the lower triangular part was stored in AL

# *Skyline* for symmetric matrices

- The **memory** needed to store the matrix in this format depends on how effectively the **skyline** reproduces the actual pattern
- In our case:
  - The **full format** requires:  $12 \times 12 \times 8 = \mathbf{1152}$
  - Array **AL** contains **29 real numbers** (including six 0s)
  - Array **D** of **fixed length  $n=12$**  containing reals
  - Array **I** of **fixed length  $n=12$**  containing integers
  - **Total**:  $(29 + 12) \times 8 + n \times 4 = \mathbf{376}$
- In general, we need  $(n_{AL} + n) \times 8 + n \times 4$
- Generally, **Skyline is more convenient than the COO** format if the coefficients are well clustered around the diagonal

# *Skyline* for general matrices

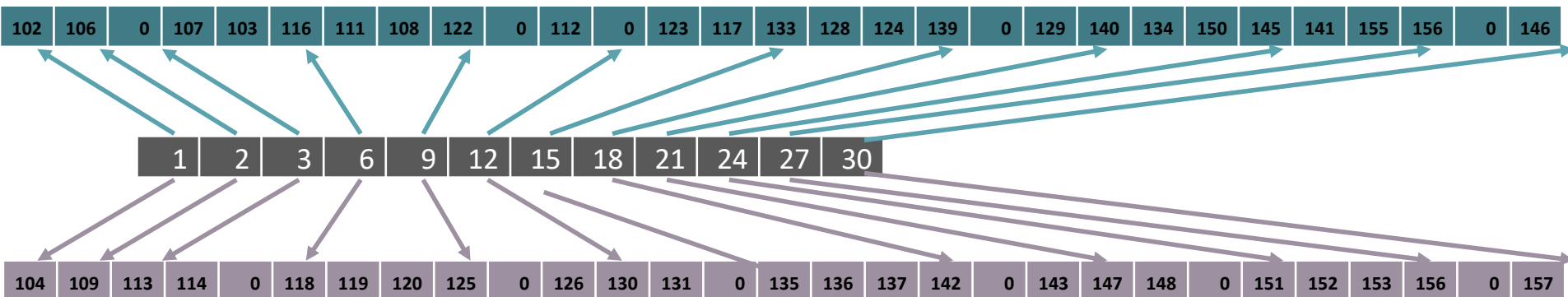
- As for **non-symmetric matrices**, a reasonable way to proceed is to split A into:
  - The diagonal D
  - The strictly lower triangular part E
  - The strictly upper triangular part F
- Using the Matlab syntax, these matrices would be defined as:  
**D=diag(diag(A)) ;**  
**E=tril(A,-1) ;**  
**F=triu(A,1) ;**

# Skyline for general matrices

- In general, we need two arrays of indexes: one for pointer to array E and one for pointers to array FT
- If the pattern of A is symmetric, the skyline of E coincides with that of FT, and the same array of pointers I is for both triangular parts
- Diagonal **D**

101 105 110 115 121 127 132 138 144 149 154 158

- Pointer **I**, lower *skyline* elements **E** and upper *skyline* elements **FT**



# Skyline for general matrices

- The product matrix-vector  $\mathbf{y} = \mathbf{A}\mathbf{x}$  now reads

```

y=D.*x;
for k=2:n
    nex = I(k)-I(k-1);
    ik = I(k-1):I(k)-1;
    jcol = k-nex:k-1;
    y(k) = y(k)+dot(E(ik),x(jcol));
    y(jcol)= y(jcol)+FT(ik)*x(k);
end

```

- icol** and **ik** contain all indexes corresponding to the columns of row k, so the scalar product **dot(E(ik),x(jcol))** and the multiplication vector-constant **FT(ik)\*x(k)** are optimized



# *Skyline* for general matrices

- Notice that in this format the **access to diagonal entries is direct**
- Being able to access diagonal entries directly has certain advantages:
  - For instance there are methods that, to impose essential boundary condition, only need the access to diagonal elements
- The **cost of extracting a row** is independent of the matrix' size
- The fact that the data relative to a row are stored consecutively in the memory allows the system to optimize the processor's cache memory when multiplying a matrix by a vector
- The **extraction of column** is an **expensive** operation that requires many comparisons, and whose cost grows linearly in  $n$

# The Compressed Sparse Row (CSR) format

- The problem with the *skyline* format is that the memory used depends on the numeration of elements and is in general impossible to avoid the unnecessary storage of zero elements
- The CSR (Compressed Sparse Row) format can be seen as a compressed version of COO, and also as an improved *skyline*, that stores non-zero elements only

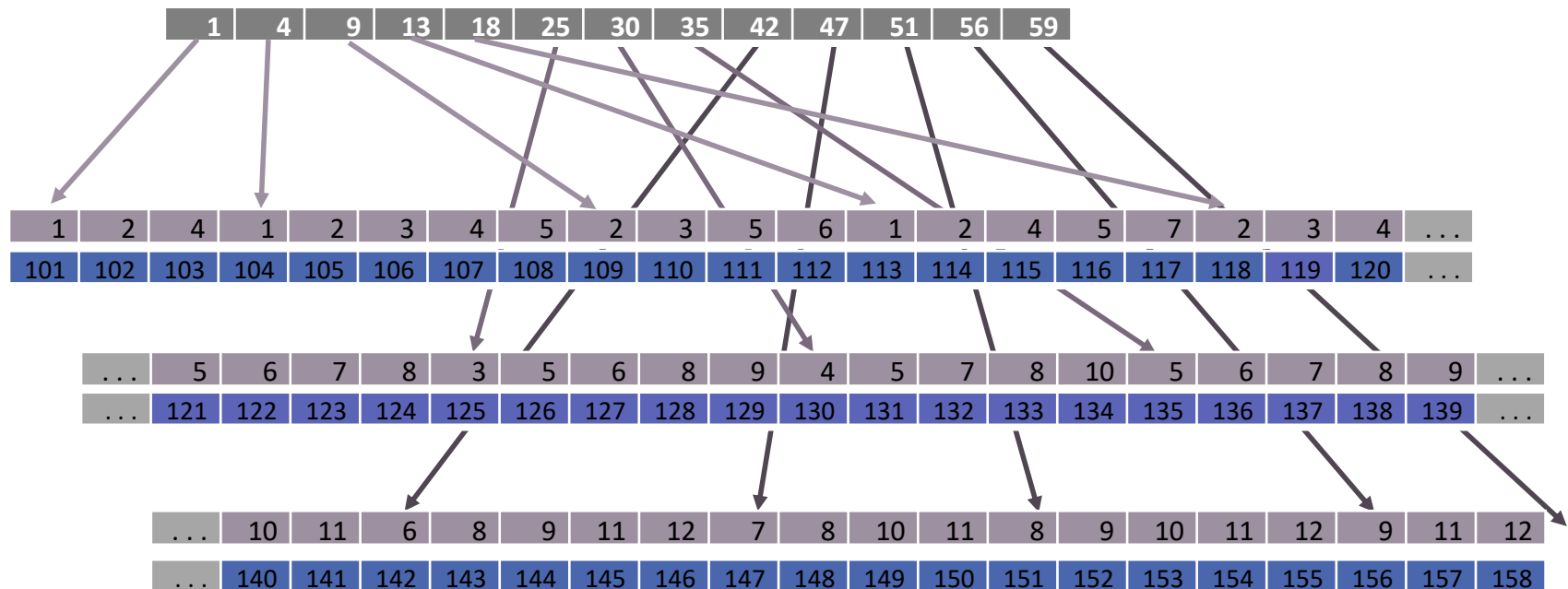
# The Compressed Sparse Row (CSR) format

The CSR format uses three arrays:

- **A** - real array of length  $nz$  storing the **non-zero entries** of the matrix, ordered row-wise. It coincides with array A of the COO format
- **J** - integer array of length  $nz$ , whose entry  $J(k)$  indicates the **column** of the element  $A(k)$ . It coincides with array J of the COO format
- **I** - integer array of length  $n$  containing **pointers** to the rows. Then **I(k)** gives the position where the  $k$ -th row in A and J begins

# The Compressed Sparse Row (CSR) format

- Array  $I$  is usually of length  $n + 1$ , so that the number of non-zero entries on row  $k$  is always  $I(k+1) - I(k)$
- To make this hold, the last element  $I(n+1)$  will contain  $nz + 1$  and in this way we also have that  $nz = I(n+1) - I(1)$



# The Compressed Sparse Row (CSR) format

- The CSR format uses  $8 \times nz + 4 \times (nz + n + 1)$  bytes
- CSR format suits square and rectangular matrices alike
- Operations:
  - quick **extraction of row  $i$**   $\rightarrow$  elements between  $I(i)$  and  $I(i+1) - 1$
  - **column extraction** requires localizing on *each row* the values of  $J$  corresponding to the wanted column
    - If we adopt no particular ordering, the cost operation is proportional to  $nz$
    - If, instead, column indexes of each row in  $J$  are ordered in increasing order as in our example, with a binary-search algorithm the extraction cost for a column becomes proportional to  $n \log_2(m)$ , where  $m$  is the mean number of elements on each row
  - the **access to a generic element** has normally a cost proportional to  $m$ , yet if we order columns it reduces to  $\log_2 m$

# The Compressed Sparse Row (CSR) format

The matrix-vector product  $\mathbf{y} = \mathbf{A}\mathbf{x}$  is given by

```

y=zeros(n,1);
% y=A(I(1:n)).*x if the diagonal is stored first
for k=1:n
    ik=I(k):I(k+1)-1;
    % ik=I(k)+1:I(k+1)-1; if the diagonal is stored first
    jcol =J(ik); y(k)=y(k)+dot(A(ik),x(jcol));
end
  
```

# The CSC (Compressed Sparse Column) format

- The **CSC (Compressed Sparse Column) format** stores matrices by ordering them column-wise
- It is **easy to extract a column** as opposed to rows
- The **roles of vectors I and J is exchanged** compared with the CSR format
- When performing matrix-vector multiplication with a sparse matrix in CSC format it is preferable to compute the result as a linear combination of the columns of the matrix
- Indeed, if  $\mathbf{c}_i$  indicates the  $i$ -th column of matrix  $A$ , we have that
$$A\mathbf{x} = \sum_i x_i \mathbf{c}_i$$

# The CSC (Compressed Sparse Column) format

- Therefore, the matrix-vector product  $\mathbf{y} = \mathbf{A}\mathbf{x}$  on a CSC matrix may be computed as:

```

y=zeros(n,1);
for k=1:n
    xcoeff=x(k);
    jk=I(k):I(k+1)-1;
    ik=J(jk);
    y(ik)=y(ik) + xcoeff * A(jk)';
end

```



# The MSR (Modified Sparse Row) format

- The **MSR (Modified Sparse Row)** format is a **special version of CSR for square matrices** exploiting the fact that:
  - The **diagonal elements** of many matrices are usually **nonzero** (matrices generated by finite elements)
  - The **diagonal elements** are **accessed more often** than the rest of the elements
- **Diagonal** entries can be stored in one single array, since their *indexes are implicitly known* from their position in the array
- As for the symmetric skyline, **only off-diagonal elements are stored** in a special fashion, i.e. through a format akin to CSR

# The MSR (Modified Sparse Row) format

The MSR format uses two arrays:

- **V** - real array of values:
  - In the **first  $n$  entries** of  $V$  we store the **diagonal**
  - The **place  $n+1$**  in  $V$  is left with ***no significant value*** (may sometimes carry some information concerning the matrix)
  - From **place  $n+2$  onwards** off-diagonal elements are stored, row-wise
  - $V$  has length  $nz + 1$
- **B** - Bind
  - $B$  has the same length as  $V \rightarrow nz + 1$
  - The **first  $n + 1$  point** to where rows begin
  - **From  $n+2$  to  $nz+1$**  there are the column indexes of the elements stored in the corresponding places in  $V$

# The MSR (Modified Sparse Row) format

## Example

|     |     |     |     |     |     |     |     |     |     |     |     |
|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|
| 101 | 102 | 0   | 103 | 0   | 0   | 0   | 0   | 0   | 0   | 0   | 0   |
| 104 | 105 | 106 | 107 | 108 | 0   | 0   | 0   | 0   | 0   | 0   | 0   |
| 0   | 109 | 110 | 0   | 111 | 112 | 0   | 0   | 0   | 0   | 0   | 0   |
| 113 | 114 | 0   | 115 | 116 | 0   | 117 | 0   | 0   | 0   | 0   | 0   |
| 0   | 118 | 119 | 120 | 121 | 122 | 123 | 124 | 0   | 0   | 0   | 0   |
| 0   | 0   | 125 | 0   | 126 | 127 | 0   | 128 | 129 | 0   | 0   | 0   |
| 0   | 0   | 0   | 130 | 131 | 0   | 132 | 133 | 0   | 134 | 0   | 0   |
| 0   | 0   | 0   | 0   | 135 | 136 | 137 | 138 | 139 | 140 | 141 | 0   |
| 0   | 0   | 0   | 0   | 0   | 142 | 0   | 143 | 144 | 0   | 145 | 146 |
| 0   | 0   | 0   | 0   | 0   | 0   | 147 | 148 | 0   | 149 | 150 | 0   |
| 0   | 0   | 0   | 0   | 0   | 0   | 0   | 151 | 152 | 153 | 154 | 155 |
| 0   | 0   | 0   | 0   | 0   | 0   | 0   | 0   | 156 | 0   | 157 | 158 |

- Array B

|     |    |    |    |    |    |    |    |    |    |    |    |    |   |    |   |   |    |    |   |     |     |
|-----|----|----|----|----|----|----|----|----|----|----|----|----|---|----|---|---|----|----|---|-----|-----|
| 14  | 16 | 20 | 23 | 27 | 33 | 37 | 41 | 47 | 51 | 54 | 58 | 60 | 2 | 4  | 1 | 3 | 4  | 5  | 2 | 5   | ... |
| ... | 6  | 1  | 2  | 5  | 7  | 9  | 2  | 3  | 4  | 6  | 7  | 8  | 3 | 5  | 8 | 9 | 4  | 5  | 8 | ... |     |
| ... | 10 | 5  | 6  | 7  | 9  | 10 | 11 | 6  | 8  | 11 | 12 | 7  | 8 | 11 | 8 | 9 | 10 | 12 | 9 | 11  |     |

- Array V

|     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |
|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|
| 101 | 105 | 110 | 115 | 121 | 127 | 132 | 138 | 144 | 149 | 154 | 158 | *   | 102 | 103 | 104 | 106 | 107 | 108 | 109 | ... |
| ... | 111 | 112 | 113 | 114 | 116 | 117 | 118 | 119 | 120 | 122 | 123 | 124 | 125 | 126 | 128 | 129 | 130 | 131 | 133 | ... |
| ... | 134 | 135 | 136 | 137 | 139 | 140 | 141 | 142 | 143 | 145 | 146 | 147 | 148 | 150 | 151 | 152 | 153 | 155 | 156 | 157 |

# The MSR (Modified Sparse Row) format

- The MSR format turns out to be **very efficient** in memory terms
- It is one of the most **compact** formats for sparse matrices
- It is used in several linear algebra libraries for large problems
- The **drawback** is that it **only** applies to **square matrices**
- The matrix-vector product is coded as:

```

y=V(1:n) .* x;
for k=1:n
    ik=B(k) : B(k+1) - 1;
    jcol = B(ik);
    y(k)=y(k)+dot(A(ik), x(jcol));
end

```

# BSR (Block Sparse Row) format

- The **BSR format** is a CSR with **dense submatrices** of fixed shape instead of scalar items
- The block size must evenly divide the shape of the matrix

|     |     |     |     |     |     |     |     |     |     |     |     |
|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|
| 101 | 102 | 0   | 103 | 0   | 0   | 0   | 0   | 0   | 0   | 0   | 0   |
| 104 | 105 | 106 | 107 | 108 | 0   | 0   | 0   | 0   | 0   | 0   | 0   |
| 0   | 109 | 110 | 0   | 111 | 112 | 0   | 0   | 0   | 0   | 0   | 0   |
| 113 | 114 | 0   | 115 | 116 | 0   | 117 | 0   | 0   | 0   | 0   | 0   |
| 0   | 118 | 119 | 120 | 121 | 122 | 123 | 124 | 0   | 0   | 0   | 0   |
| 0   | 0   | 125 | 0   | 126 | 127 | 0   | 128 | 129 | 0   | 0   | 0   |
| 0   | 0   | 0   | 130 | 131 | 0   | 132 | 133 | 0   | 134 | 0   | 0   |
| 0   | 0   | 0   | 0   | 135 | 136 | 137 | 138 | 139 | 140 | 141 | 0   |
| 0   | 0   | 0   | 0   | 0   | 142 | 0   | 143 | 144 | 0   | 145 | 146 |
| 0   | 0   | 0   | 0   | 0   | 0   | 147 | 148 | 0   | 149 | 150 | 0   |
| 0   | 0   | 0   | 0   | 0   | 0   | 0   | 151 | 152 | 153 | 154 | 155 |
| 0   | 0   | 0   | 0   | 0   | 0   | 0   | 0   | 156 | 0   | 157 | 158 |

|   |   |   |   |
|---|---|---|---|
| A | B |   |   |
| C | D | E |   |
|   | F | G | H |
|   |   | I | L |

- In this example the block size is 3 x 3

# BSR (Block Sparse Row) format

- The BSR format store the **non-zero blocks** of the sparse matrix
- A **non-zero block** is the block that contains *at least one non-zero element*

|     |     |     |     |     |     |     |     |     |     |     |     |
|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|
| 101 | 102 | 0   | 103 | 0   | 0   | 0   | 0   | 0   | 0   | 0   | 0   |
| 104 | 105 | 106 | 107 | 108 | 0   | 0   | 0   | 0   | 0   | 0   | 0   |
| 0   | 109 | 110 | 0   | 111 | 112 | 0   | 0   | 0   | 0   | 0   | 0   |
| 113 | 114 | 0   | 115 | 116 | 0   | 117 | 0   | 0   | 0   | 0   | 0   |
| 0   | 118 | 119 | 120 | 121 | 122 | 123 | 124 | 0   | 0   | 0   | 0   |
| 0   | 0   | 125 | 0   | 126 | 127 | 0   | 128 | 129 | 0   | 0   | 0   |
| 0   | 0   | 0   | 130 | 131 | 0   | 132 | 133 | 0   | 134 | 0   | 0   |
| 0   | 0   | 0   | 0   | 135 | 136 | 137 | 138 | 139 | 140 | 141 | 0   |
| 0   | 0   | 0   | 0   | 0   | 142 | 0   | 143 | 144 | 0   | 145 | 146 |
| 0   | 0   | 0   | 0   | 0   | 0   | 147 | 148 | 0   | 149 | 150 | 0   |
| 0   | 0   | 0   | 0   | 0   | 0   | 0   | 151 | 152 | 153 | 154 | 155 |
| 0   | 0   | 0   | 0   | 0   | 0   | 0   | 0   | 156 | 0   | 157 | 158 |

|   |   |   |   |
|---|---|---|---|
| A | B |   |   |
| C | D | E |   |
|   | F | G | H |
|   |   | I | L |

# BSR (Block Sparse Row) format

The **BSR format** consists of four arrays:

- **Values** - real array containing the elements of the non-zero blocks of a sparse matrix
  - The elements are stored block-by-block in row-major order
  - All elements of non-zero blocks are stored, even if some of them are equal to zero
  - Within each non-zero block elements are stored in column-major order in the case of one-based indexing, and in row-major order in the case of the zero-based indexing

|   |   |   |   |
|---|---|---|---|
| A | B |   |   |
| C | D | E |   |
|   | F | G | H |
|   |   | I | L |

# BSR (Block Sparse Row) format

The **BSR format** consists of four arrays:

- **Columns** - integer array where **element  $i$**  is the number of the column in the block matrix that contains the  **$i$ -th non-zero block**
- **PointerB** - integer array where **element  $j$**  gives the **index** of the element in the **columns array** that is **first non-zero block in row  $j$**  of the block matrix
- **PointerE** - integer array where **element  $j$**  gives the **index** of the element in the **columns array** that contains the **last non-zero block in a row  $j$**  of the block matrix plus 1



# BSR (Block Sparse Row) format

## Example

- Values

|     |     |     |     |     |     |     |     |     |     |     |     |   |     |     |     |   |     |     |     |
|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|---|-----|-----|-----|---|-----|-----|-----|
| 101 | 104 | 0   | 102 | 105 | 109 | 0   | 106 | 110 | 103 | 107 | 0   | 0 | 108 | 111 | 0   | 0 | 112 | ... |     |
| ... | 113 | 0   | 0   | 114 | 118 | 0   | 0   | 119 | 125 | 115 | 120 | 0 | 116 | 121 | 126 | 0 | 122 | 127 | ... |
| ... | 117 | 123 | 0   | 0   | 124 | 128 | 0   | 0   | 129 | 130 | 0   | 0 | 131 | 135 | 0   | 0 | 136 | 142 | ... |
| ... | 132 | 137 | 0   | 133 | 138 | 143 | 0   | 139 | 144 | 134 | 140 | 0 | 0   | 141 | 145 | 0 | 0   | 146 | ... |
| ... | 147 | 0   | 0   | 148 | 151 | 0   | 0   | 152 | 156 | 149 | 153 | 0 | 150 | 154 | 157 | 0 | 155 | 158 | ... |

- Columns

|   |   |   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|---|---|
| 1 | 2 | 1 | 2 | 3 | 2 | 3 | 4 | 3 | 4 |
|---|---|---|---|---|---|---|---|---|---|

- PointerB

|   |   |   |   |
|---|---|---|---|
| 1 | 3 | 6 | 9 |
|---|---|---|---|

- PointerE

|   |   |   |    |
|---|---|---|----|
| 3 | 6 | 9 | 11 |
|---|---|---|----|

|   |   |   |   |
|---|---|---|---|
| A | B |   |   |
| C | D | E |   |
|   | F | G | H |
|   |   | I | L |

# BSR (Block Sparse Row) format

- The **length** of the **values** array is equal to the number of all elements in the non-zero blocks
- The **length** of the **columns** array is equal to the number of non-zero blocks
- The **length** of the **pointerB** and **pointerE** arrays is equal to the number of block rows in the block matrix

|   |   |   |   |
|---|---|---|---|
| A | B |   |   |
| C | D | E |   |
|   | F | G | H |
|   |   | I | L |

# Diagonal format

- **Diagonally structured matrices** are matrices whose nonzero elements are located along a **small number of diagonals**

The **diag format** consist of:

- **DIAG** - a rectangular real array storing the diagonals
  - DIAG has size  $n \times Nd$ , where  $Nd$  is the number of diagonals
- **IOFF** - an integer array containing the offsets of each diagonal with respect to the main diagonal
  - IOFF has size  $Nd$

# Diagonal format

- The **order** in which the diagonals are stored in of DIAG is generally **unimportant**
- Since several more **operations** are performed with the **main diagonal**, storing it in the **first column** may be slightly advantageous
- Note that all the **diagonals** except the main diagonal have **fewer than n elements**, so there are positions in DIAG that will not be used

# Diagonal format

## Example

- Matrix

|     |     |     |     |     |     |     |     |     |     |     |     |
|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|
| 101 | 102 | 0   | 103 | 0   | 0   | 0   | 0   | 0   | 0   | 0   | 0   |
| 104 | 105 | 106 | 107 | 108 | 0   | 0   | 0   | 0   | 0   | 0   | 0   |
| 0   | 109 | 110 | 0   | 111 | 112 | 0   | 0   | 0   | 0   | 0   | 0   |
| 113 | 114 | 0   | 115 | 116 | 0   | 117 | 0   | 0   | 0   | 0   | 0   |
| 0   | 118 | 119 | 120 | 121 | 122 | 123 | 124 | 0   | 0   | 0   | 0   |
| 0   | 0   | 125 | 0   | 126 | 127 | 0   | 128 | 129 | 0   | 0   | 0   |
| 0   | 0   | 0   | 130 | 131 | 0   | 132 | 133 | 0   | 134 | 0   | 0   |
| 0   | 0   | 0   | 0   | 135 | 136 | 137 | 138 | 139 | 140 | 141 | 0   |
| 0   | 0   | 0   | 0   | 0   | 142 | 0   | 143 | 144 | 0   | 145 | 146 |
| 0   | 0   | 0   | 0   | 0   | 0   | 147 | 148 | 0   | 149 | 150 | 0   |
| 0   | 0   | 0   | 0   | 0   | 0   | 0   | 151 | 152 | 153 | 154 | 155 |
| 0   | 0   | 0   | 0   | 0   | 0   | 0   | 0   | 156 | 0   | 157 | 158 |

Element  $\text{DIAG}(i, j)$  is located in position  $a_{i, i+\text{ioff}(j)}$  of the original matrix

- DIAG

|     |     |     |     |     |     |     |
|-----|-----|-----|-----|-----|-----|-----|
| 113 | 0   | 104 | 101 | 102 | 0   | 103 |
| 118 | 114 | 109 | 105 | 106 | 107 | 108 |
| 125 | 119 | 0   | 110 | 0   | 111 | 112 |
| 130 | 0   | 120 | 115 | 116 | 0   | 117 |
| 135 | 131 | 126 | 121 | 122 | 123 | 124 |
| 142 | 136 | 0   | 127 | 0   | 128 | 129 |
| 147 | 0   | 137 | 132 | 133 | 0   | 134 |
| 151 | 148 | 143 | 138 | 139 | 140 | 141 |
| 156 | 152 | 0   | 144 | 0   | 145 | 146 |
| 0   | 0   | 153 | 149 | 150 | 0   | 0   |
| 0   | 0   | 157 | 154 | 155 | 0   | 0   |
| 0   | 0   | 0   | 158 | 0   | 0   | 0   |

## IOFF

|    |    |    |   |   |   |   |
|----|----|----|---|---|---|---|
| -3 | -2 | -1 | 0 | 1 | 2 | 3 |
|----|----|----|---|---|---|---|

# Ellpack-Itpack format

- The **Ellpack-Itpack format** is a general scheme, popular on vector machines
- The Ellpack-Itpack format consists of two rectangular arrays:
  - **COEF** - real array (similar to DIAG) that contains the nonzero elements of A (completing the row by zeros as necessary)
  - **JCOEF** - integer array that contains the column positions of each entry in COEF
  - COEF and JCOEF have size  $n \times Nd$ , where  $n$  is the number of rows and  $Nd$  is the maximum number of nonzero elements per row, with  $Nd$  small

# Ellpack-Itpack format

## Example

- Matrix

|     |     |     |     |     |     |     |     |     |     |     |     |
|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|
| 101 | 102 | 0   | 103 | 0   | 0   | 0   | 0   | 0   | 0   | 0   | 0   |
| 104 | 105 | 106 | 107 | 108 | 0   | 0   | 0   | 0   | 0   | 0   | 0   |
| 0   | 109 | 110 | 0   | 111 | 112 | 0   | 0   | 0   | 0   | 0   | 0   |
| 113 | 114 | 0   | 115 | 116 | 0   | 117 | 0   | 0   | 0   | 0   | 0   |
| 0   | 118 | 119 | 120 | 121 | 122 | 123 | 124 | 0   | 0   | 0   | 0   |
| 0   | 0   | 125 | 0   | 126 | 127 | 0   | 128 | 129 | 0   | 0   | 0   |
| 0   | 0   | 0   | 130 | 131 | 0   | 132 | 133 | 0   | 134 | 0   | 0   |
| 0   | 0   | 0   | 0   | 135 | 136 | 137 | 138 | 139 | 140 | 141 | 0   |
| 0   | 0   | 0   | 0   | 0   | 142 | 0   | 143 | 144 | 0   | 145 | 146 |
| 0   | 0   | 0   | 0   | 0   | 0   | 147 | 148 | 0   | 149 | 150 | 0   |
| 0   | 0   | 0   | 0   | 0   | 0   | 0   | 151 | 152 | 153 | 154 | 155 |
| 0   | 0   | 0   | 0   | 0   | 0   | 0   | 0   | 156 | 0   | 157 | 158 |

COEF

|     |     |     |     |     |     |     |
|-----|-----|-----|-----|-----|-----|-----|
| 101 | 102 | 103 | 0   | 0   | 0   | 0   |
| 104 | 105 | 106 | 107 | 108 | 0   | 0   |
| 109 | 110 | 111 | 112 | 0   | 0   | 0   |
| 113 | 114 | 115 | 116 | 117 | 0   |     |
| 118 | 119 | 120 | 121 | 122 | 123 | 124 |
| 125 | 126 | 127 | 128 | 129 | 0   | 0   |
| 130 | 131 | 132 | 133 | 134 | 0   | 0   |
| 135 | 136 | 137 | 138 | 139 | 140 | 141 |
| 142 | 143 | 144 | 145 | 146 | 0   | 0   |
| 147 | 148 | 149 | 150 | 0   | 0   | 0   |
| 151 | 152 | 153 | 154 | 155 | 0   | 0   |
| 156 | 157 | 158 | 0   | 0   | 0   | 0   |

JCOEF

|   |    |    |    |    |    |    |
|---|----|----|----|----|----|----|
| 1 | 2  | 4  | 0  | 0  | 0  | 0  |
| 1 | 2  | 3  | 4  | 5  | 0  | 0  |
| 2 | 3  | 5  | 6  | 0  | 0  | 0  |
| 1 | 2  | 4  | 5  | 7  | 0  |    |
| 2 | 3  | 4  | 5  | 6  | 7  | 8  |
| 3 | 5  | 6  | 8  | 9  | 0  | 0  |
| 4 | 5  | 7  | 8  | 10 | 0  | 0  |
| 5 | 6  | 7  | 8  | 9  | 10 | 11 |
| 6 | 8  | 9  | 11 | 12 | 0  | 0  |
| 7 | 8  | 10 | 11 | 0  | 0  | 0  |
| 8 | 9  | 10 | 11 | 12 | 0  | 0  |
| 9 | 11 | 12 | 0  | 0  | 0  | 0  |