Sparse Matrix Computation

Sparse Matrices

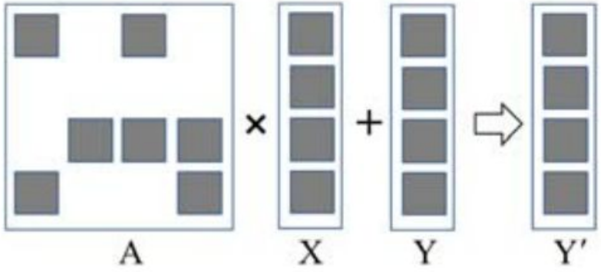
- The majority of elements are zeros
  - Storing and processing these zero elements is a waste in terms of memory and bandwidth
- Different methods used to compact sparse matrices
  - COO
  - ELL
  - CSR
- They introduce irregularity in data representation
- Unfortunately, such irregularity can lead to
  - Underutilization of memory bandwidth
  - Control flow divergence
  - Load imbalance

Why does it matter?

- Sparse matrices arise in many scientific, engineering, and financial modeling problems
  - Matrices can be used to represent the coefficients in a linear system of equations
  - Each row of the matrix represents one equation of the linear system
- Usually, scientific problem modeling involves many variables
  - Only a small number of them are relevant with non-zero coefficients
- Matrices are often used in solving linear systems of N equations of N variables in the form of $AX + Y = 0$
- Graph computation
  - For example, large networks sparsely connected

SpMV

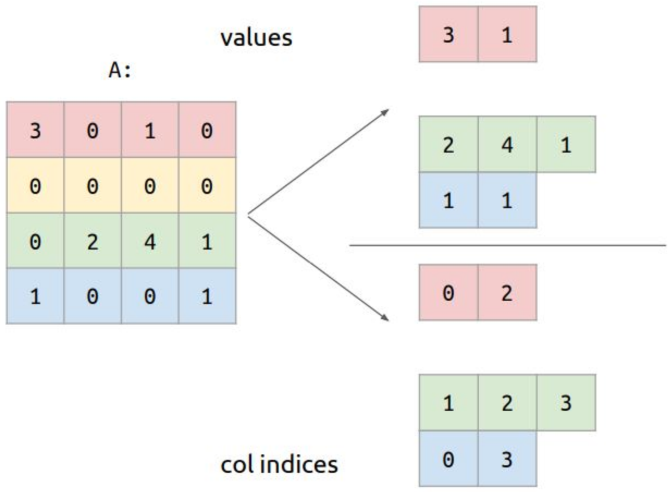
- Sparse Matrix-Vector multiplication and accumulation (SpMV)
  - Owing to its importance, standardized library function interfaces have been created to perform this operation
- SpMV is an example of the tradeoffs between storage formats
  - Space Efficiency
  - Flexibility
  - Accessibility
  - Memory Bandwidth
  - Load Balance



- Different sparse matrix storage formats remove all the zero elements from the matrix representation
  - No need to fetch them
  - No useless multiplications

Compressed Sparse Row (CSR) Format

- Store each row as a sparse (row) vector
  - Each row is of variable length depending on the sparsity pattern



Compressed Sparse Row (CSR) Format

- Additional storage is required to locate the start of each row

A:

3	0	1	0
0	0	0	0
0	2	4	1
1	0	0	1

values

3	1	2	4	1	1	1

col indices

0	2	1	2	3	0	3

row indices

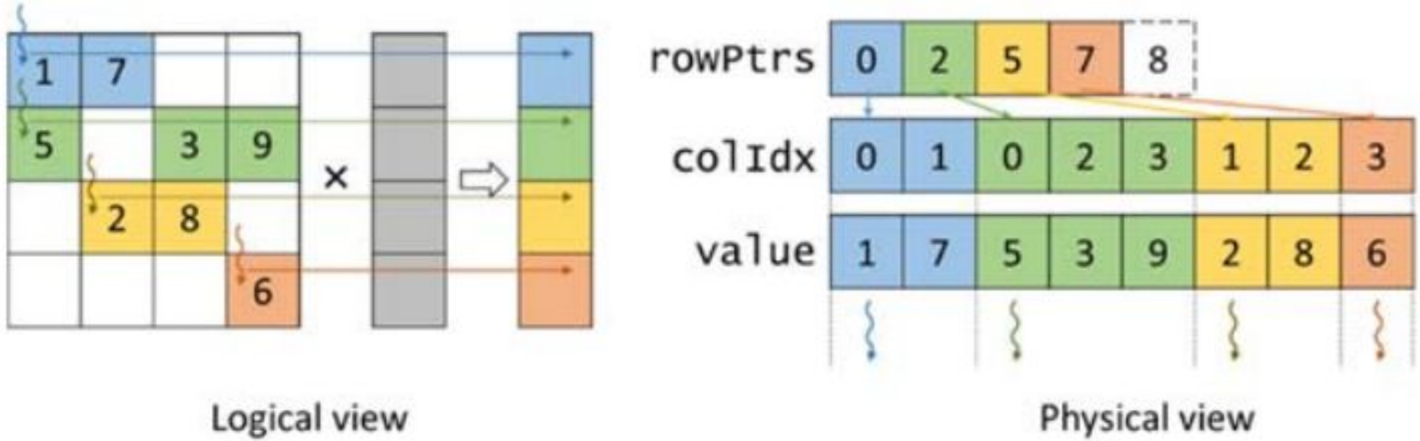
0	2	2	5	7

Compressed Sparse Row (CSR) Format

- **M** number of rows in the matrix
- **N** number of columns in the matrix
- **S** sparsity level [0 -1], 1 being fully-dense
- **Space Requirements**
    - Dense Representation $M*N$
    - Sparse representation with CSR $2MNS + M + 1$
- CSR only saves space if $S < (1 - N-1) / 2$
    - Otherwise, indexes saving introduce overhead

Parallel SpMV CSR

- Assign a thread to each row of the matrix
  - The thread loops through the nonzero elements of its row to perform the dot product
  - A single thread traverses each row
  - So, each thread will write to a distinct output value



Parallel SpMV CSR - Considerations

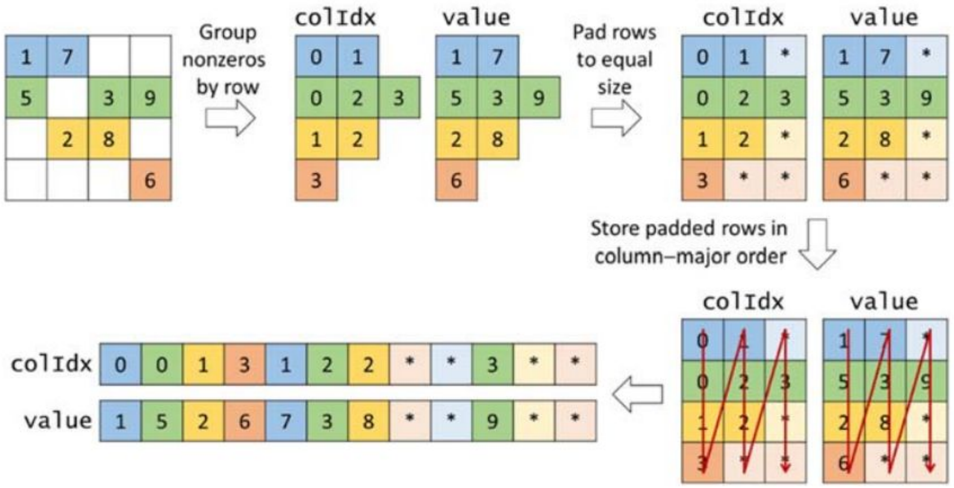
- CSR is space efficient
  - Higher than COO as the row pointers vector is smaller than the row indexes one
- CSR is not flexible
  - Adding non-zero elements to the matrix is expensive
    - As it involves shifting elements
- CSR is easy to access by row
  - Easy to find non-zero elements in a row
    - The same does not hold for columns
  - This is true with big matrices with a lot of rows

Parallel SpMV CSR - Considerations

- CSR is not using the memory bandwidth efficiently
  - Consecutive threads will access memory locations far apart from each other
  - No coalesced memory access
- CSR potential flow divergence in warps
  - The number of iterations that are taken by a thread in the dot product loop depends on the number of nonzero elements in the row that is assigned to the thread
- <u>NO Atomic operation is required</u>

ELL Format

- The name came from the sparse matrix package in ELLPACK
    - A package for solving elliptic boundary value problems
    - Relies on the maximum number of non-zeros elements per row
    - Usage of paddings elements

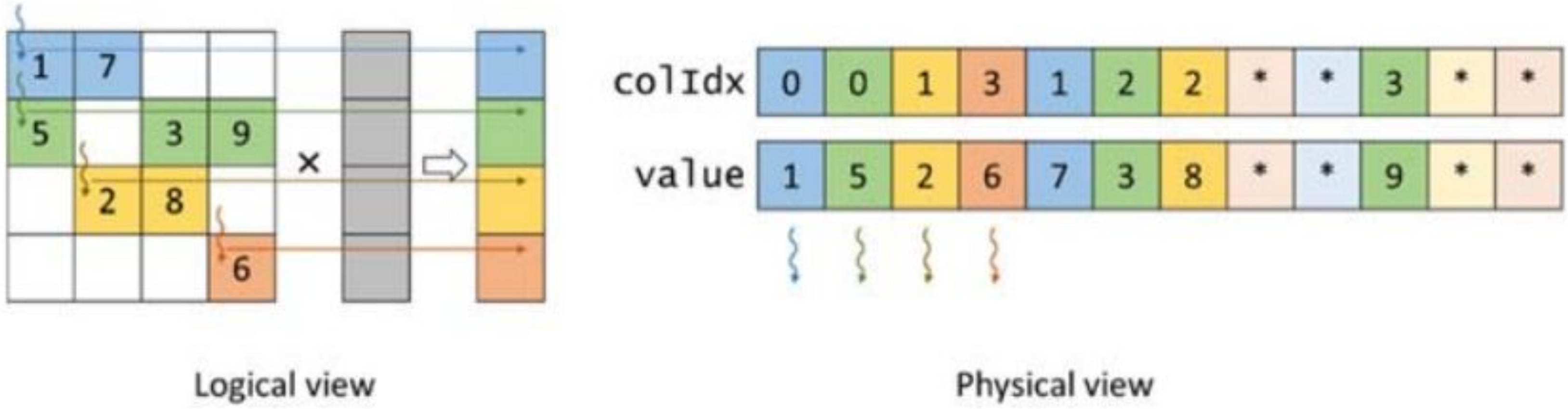


ELL Format

- **M** number of rows in the matrix
- **N** number of columns in the matrix
- **K** number of nonzero entries in the densest row
- **S** sparsity level [0 -1], 1 being fully-dense
- **Space Requirements**
  - Dense Representation $M*N$
  - Sparse representation with ELL $2MK$
- <u>ELL only saves space if</u> $K < N / 2$

Parallel SpMV ELL

- Each thread is assigned to a different row of the matrix
  - A dot product loop goes through the non-zero elements of each row
  - Each thread iterates only through the non-zeros in its assigned row
    - Thanks to the maximum number of non-zero elements per row



Parallel SpMV ELL - Considerations

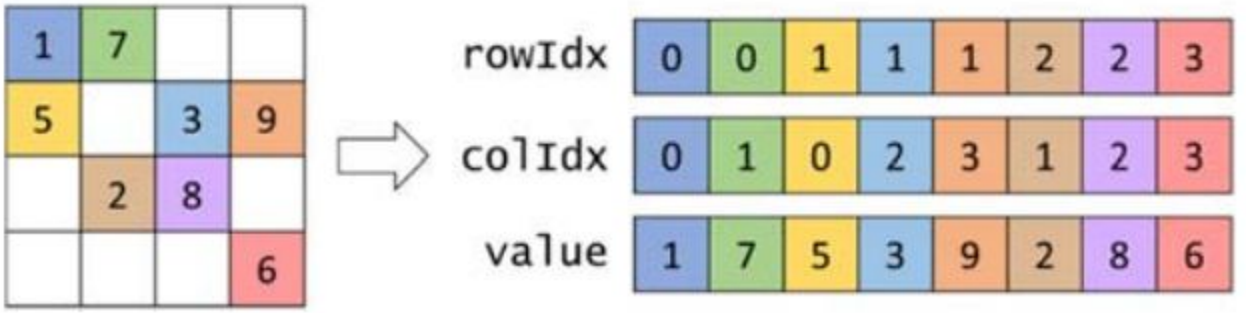
- ELL is less space efficient than CSR
  - Due to the presence of padding elements
  - The compression rate depends of the maximum number on non-zeros elements per row
- ELL is more flexible than CSR
  - A non-zero element can be added by replacing a padding element
- ELL is easy to access
  - Given a value i, we can get its column value and row easily
  - We can get parallelize by rows and by non-zero elements

Parallel SpMV ELL - Considerations

- ELL uses the memory bandwidth efficiently
  - Consecutive threads perform coalesced access to memory
- ELL computation is imbalanced
  - As each thread iterates over the non-zero elements of its row
- <u>NO Atomic operation is required</u>

The Coordinate (COO) Format

- Store both the column index and row index for every non-zero
  - No ordering is required for indexes
  - Overhead because all indexes are stored
- COO and CSR format differs since CSR replaces the row indexes array with raw pointers that store the starting offset of each row's nonzeros in the other arrays

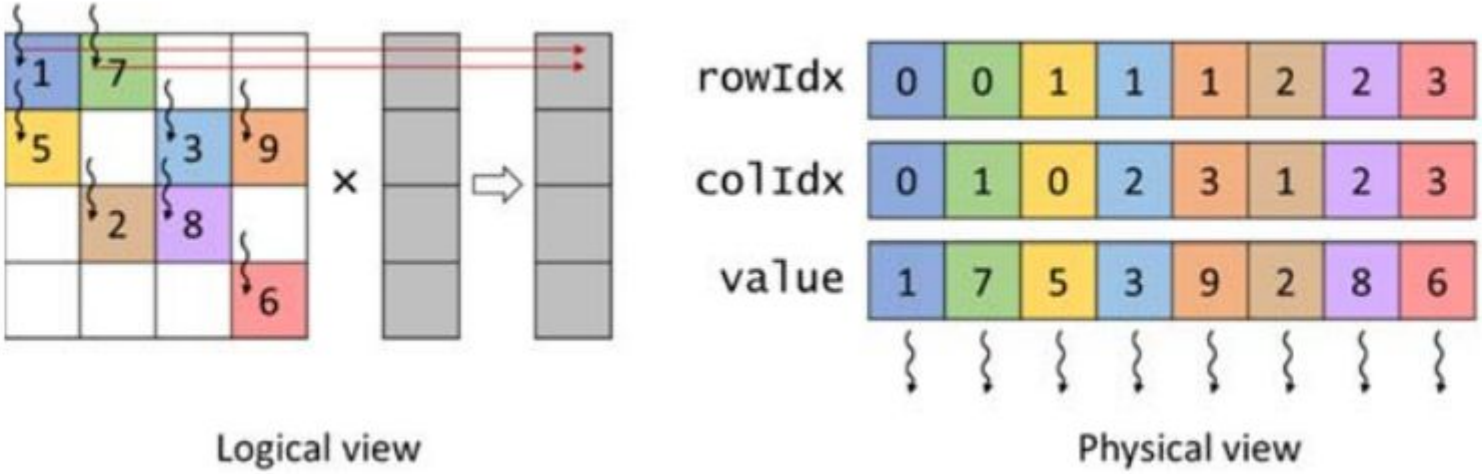


The Coordinate (COO) Format

- **M** number of rows in the matrix
- **N** number of columns in the matrix
- **S** sparsity level [0 -1], 1 being fully-dense
- **Space Requirements**
  - Dense Representation MN
  - Sparse representation with COO $3MNS$
- COO only saves space if $S < 1 / 3$

Parallel SpMV COO

- Each thread is responsible for the computation of a non-zero element in the matrix



Parallel SpMV COO - Considerations

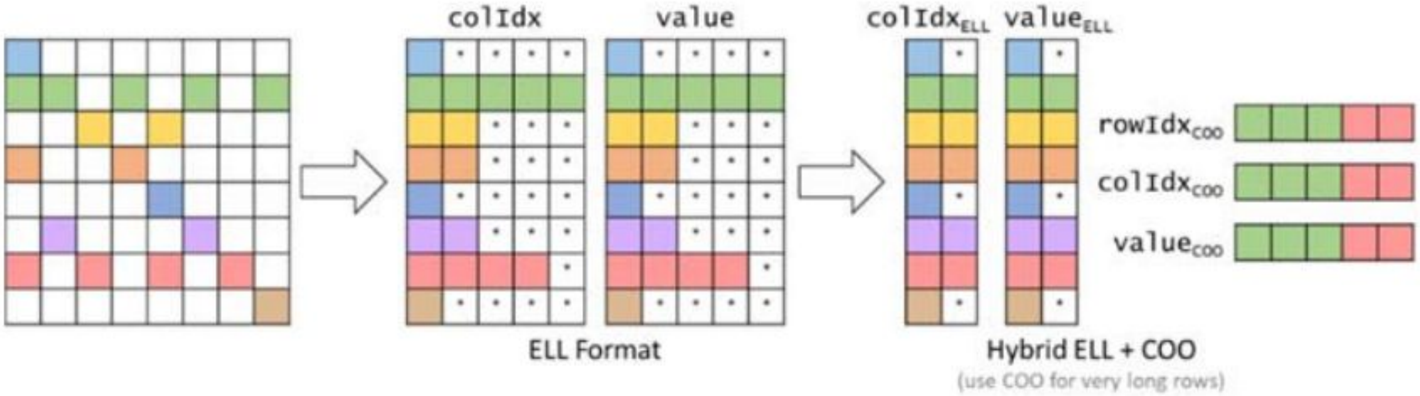
- COO is less space efficient than CSR
  - It requires more space due to the repeated indexes
- COO is flexible
  - Non-zero elements can be pushed back into the arrays
- COO accessibility varies
  - We can process elements in any order
  - We cannot access all non-zeros elements by row and by column easily

Parallel SpMV COO - Considerations

- COO uses the memory bandwidth efficiently
    - Threads access the indexes array in a coalesced way
- COO computation is balanced
    - All threads are responsible for the same amount of work
- <u>Atomic operations ARE required</u>

Hybrid ELL/COO Format

- ELL format suffers from rows with a large amount of non-zero elements
  - Mostly in terms of space efficiency and control divergence
- Place non-zeros from the densest rows in a COO sparse matrix, leading to a more efficient ELL representation for the remainder
  - Each element will be stored in the ELL or the COO matrix, not both



Parallel SpMV Hybrid ELL/COO Format

- Perform the SpMV / ELL in parallel on the GPU
  - Rows have a similar density, thus the execution is more efficient
- Perform the SpMV / COO sequentially on the CPU
  - CPUs better handle the COO access irregularity
- The increased overhead has to be justified
  - If the SpMV is computed only once probably, the execution time will increase
  - If the SpMV is done inside an iterative solver, the execution time is likely to decrease

Parallel SpMV Hybrid ELL/COO - Considerations

- Hybrid ELL/COO is more space efficient than ELL
  - It reduces padding
- Hybrid ELL/COO is more flexible than ELL
  - The non-zero elements can be added
    - For the ELL part, if the element in the column is available
    - Otherwise, it is pushed back in the COO part
- Hybrid ELL/COO accessibility is reduced wrt to ELL
  - If the elements are not available in the ELL part, a search on the COO part is required

Parallel SpMV Hybrid ELL/COO - Considerations

- Hybrid ELL/COO uses the memory bandwidth efficiently
  - Both the ELL and COO parts are accessed in a coalesced way
- Hybrid ELL/COO computation is more balanced than ELL
  - Less padding, less divergence

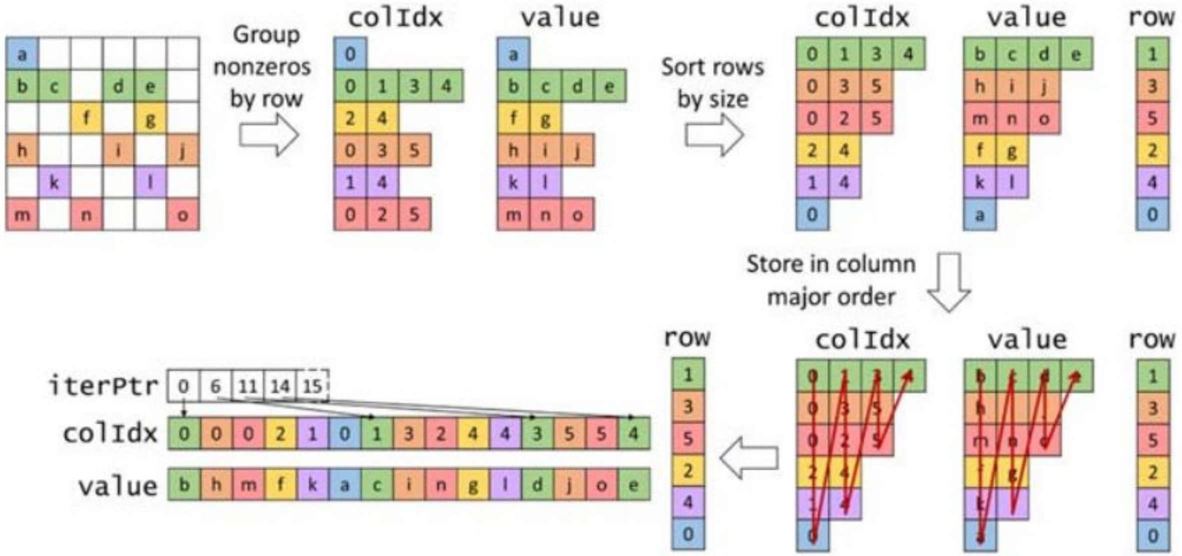
Storage Requirements Summary

- **M** number of rows in the matrix
- **N** number of columns in the matrix
- **K** number of nonzero entries in the densest row
- **S** sparsity level [0 -1], 1 being fully-dense

Format	Storage Requirements(words)
Dense	MN
Compressed Sparse Row (CSR)	$2MNS + M + 1$
ELL	$2MK$
Coordinate (COO)	$3MNS$
Hybrid ELL / COO (HYB)	$2MK < <3MNS$

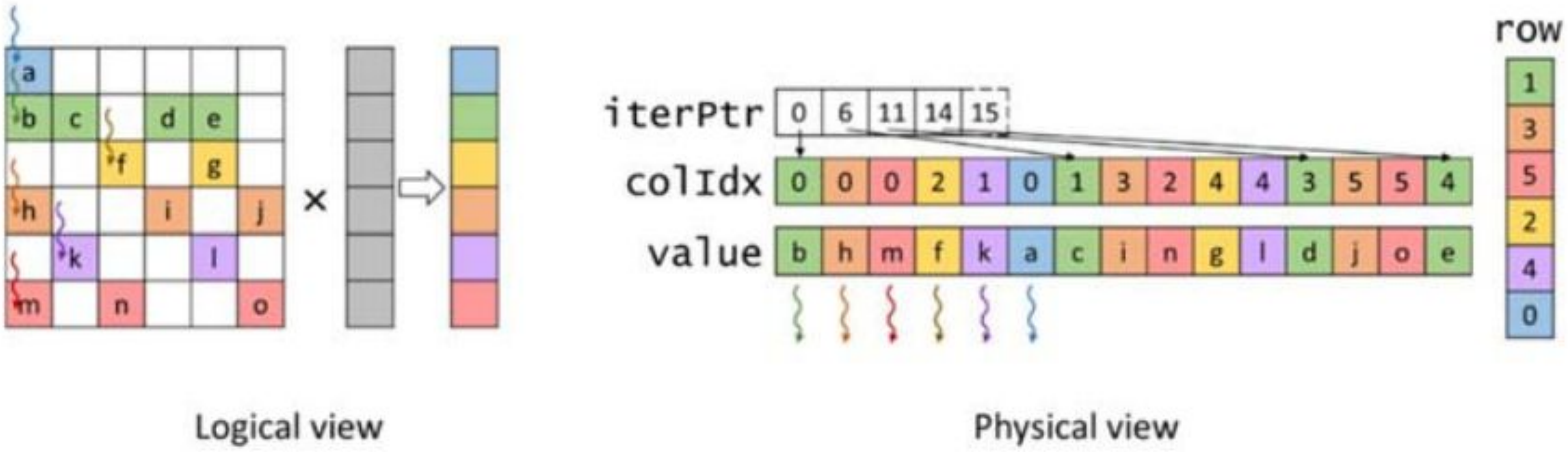
Jagged Diagonal Storage (JDS) Format

- Group similarly dense rows into evenly sized partitions and represent each section independently using either CSR or ELL
  - Sort rows by density
- Need to store the original rows of the sorted rows
  - To preserve re-constructability



Parallel SpMV Jagged Diagonal Storage (JDS) Format

- Each thread is assigned to a row of the matrix and iterates through the nonzeros of that row
  - performing the dot product along the way
- You can also partition the matrix into sections of similar rows
  - These partitions can be compressed using ELL



Parallel SpMV JDS - Considerations

- JDS is more space efficient than ELL
  - It avoids and reduces padding
- JDS is not flexible
  - Add new elements is complex
    - It may require new sorting
- JDS accessibility is similar to CSR
  - It allows us to access, given a row index, the nonzero elements of that row
  - It does not make it easy to access, given a nonzero, the row index and column index of that nonzero

Parallel SpMV Hybrid ELL/COO - Considerations

- JDS uses the memory bandwidth efficiently
  - JDS allows memory access to be coalesced
- JDS computation is balanced
  - Rows are sorted so that threads in the same warp execute similar work

Other Formats

- **Diagonal (DIA)**
  - Stores only a sparse set of dense diagonal vectors
  - For each diagonal, the offset from the main diagonal is stored
- **Packet (PKT)**
  - Reorders rows and columns to concentrate nonzeros into roughly diagonal submatrices
  - This improves cache performance as nearby rows access nearby x elements
- **Dictionary of Keys (DOK)**
  - Matrix is stored as a map from (row,column) index pairs to values
  - This can be useful for building or querying a sparse matrix, but iteration is slow
- **Compressed Sparse Column (CSC)**
  - Like CSR, but stores a dense set of sparse column vectors
  - Useful for when column sparsity is much more regular than row sparsity
- **Blocked CSR**
  - The matrix is divided into blocks stored using CSR with the indices of the upper left corner
  - Useful for block-sparse matrices
- **Additional Hybrid Methods**
  - For example, DIA is very inefficient when there are a small number of mostly-dense diagonals, but a few additional sparse entries
  - In this case, a hybrid DIA / COO or DIA / CSR representation can be used

Conclusions

- In general, the FLOPS ratings that are achieved by either CPUs or GPUs are much lower for sparse matrix computation than for dense matrix computation
  - In SpMV computatio, there is no data reuse in the sparse matrix
- The OP/B is essentially 0.25, limiting the achievable FLOPS rate to a small fraction of the peak performance
- The various formats are important for both CPUs and GPUs since both are limited by memory bandwidth

Other Libraries

- **cuSPARSE**
  - cuSPARSE is an Nvidia library implemented in set of basic linear algebra subroutines (BLAS)
  - Reference available here
- **CUSP**
  - CUSP is an open-source project
  - It is focused on solving linear systems, providing a number of conjugate-gradient solvers
  - Repository available here

Credits

- CSE 599 I Accelerated Computing - Programming GPUS - Parallel Pattern: Sparse Matrices, Author *Tanner Schmidt*

Thank you for your attention!

Gianmarco Accordi
gianmarco.accordi@polimi.it