Objective

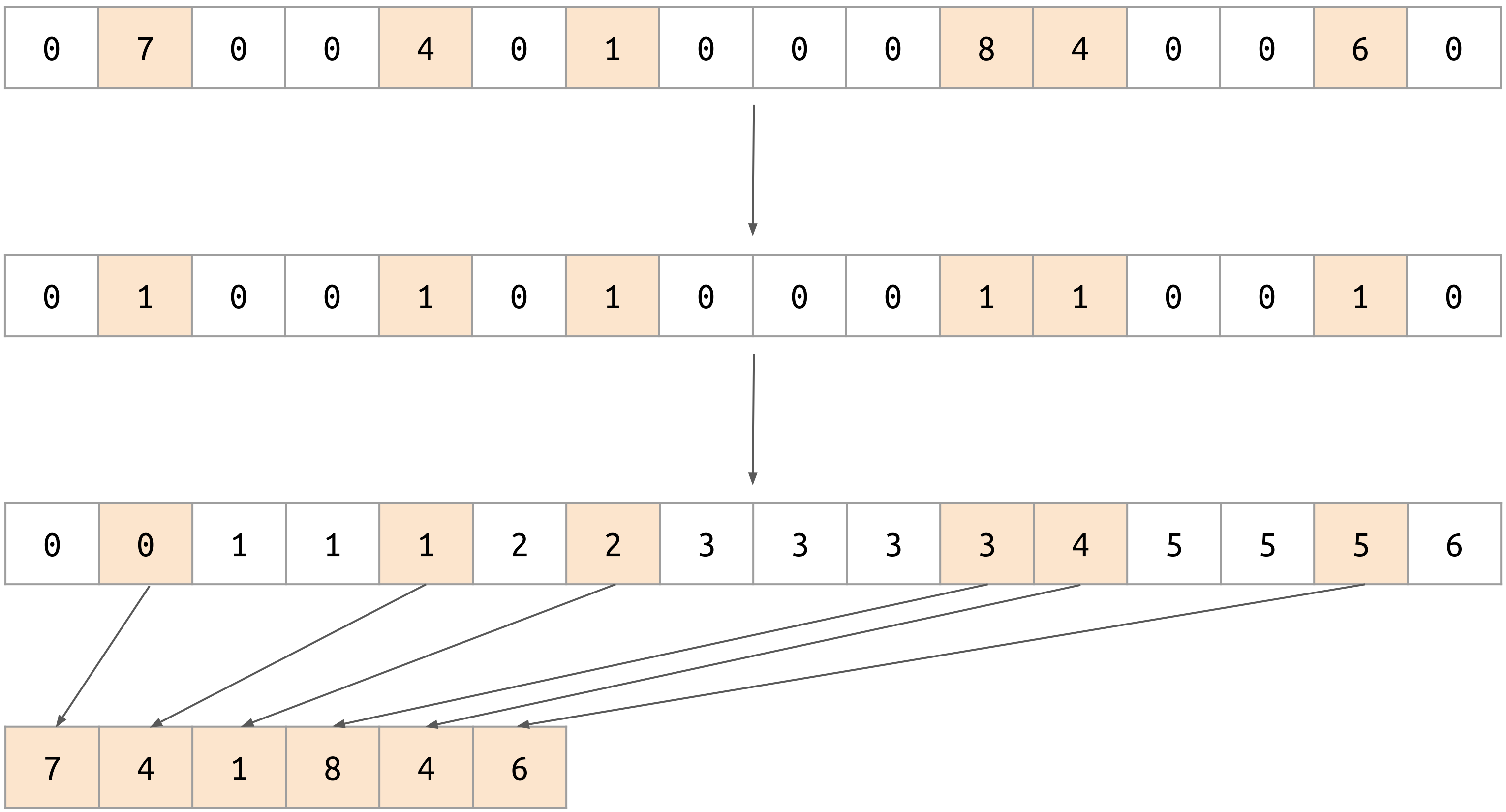
- Learn about various sparse matrix representations
- Consider how input data affects run-time performance of parallel sparse matrix algorithms
- Analyze trade-offs of different representations for various input types

Source:
Nvidia + University of Illinois

Sparse Vector Representation

0	7	0	0	4	0	1	0	0	0	8	4	0	0	6	0

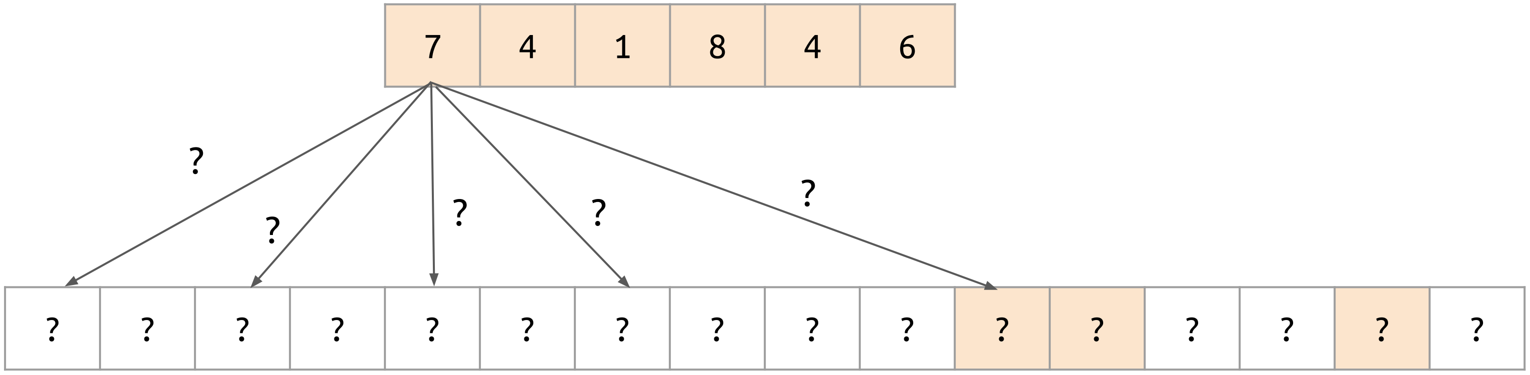
Sparse Vector Representation




```
if (input[x]) {
  output_values[scan[x]] = input[x];
}
```

Reconstructability

A successful sparse representation must allow for the reconstruction of the dense equivalent



Sparse Vector Representation

indices:	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
values:	0	7	0	0	4	0	1	0	0	0	8	4	0	0	6	0

0	1	0	0	1	0	1	0	0	0	1	1	0	0	1	0

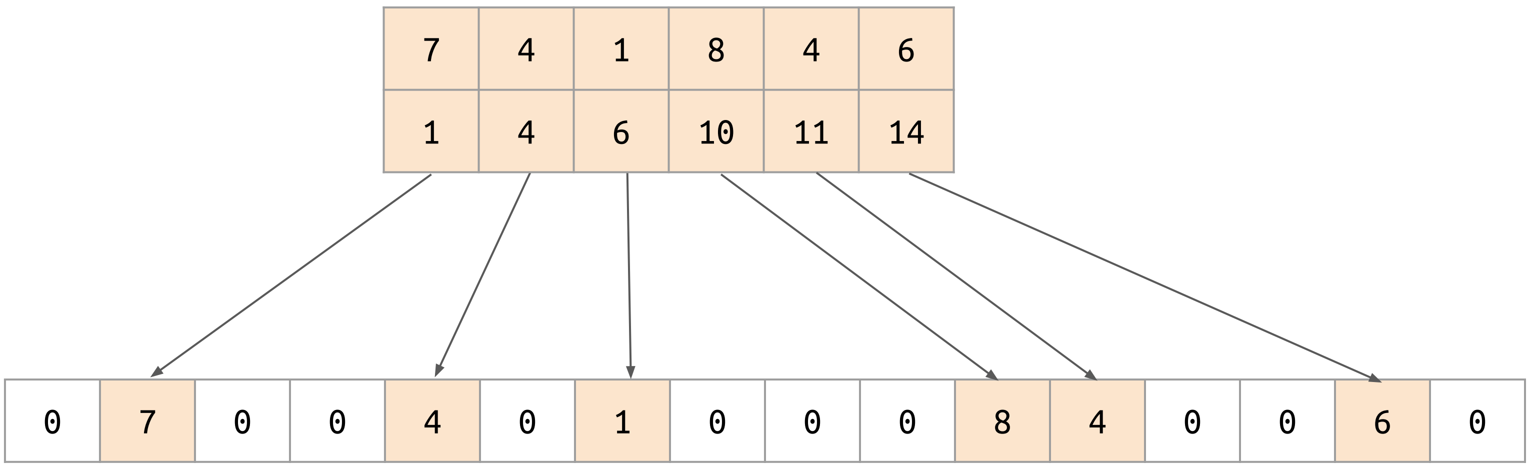
0	0	1	1	1	2	2	3	3	3	3	4	5	5	5	6

7	4	1	8	4	6
1	4	6	10	11	14

```
if (input[x]) {
  output_values[scan[x]] = input[x];
  output_indices[scan[x]] = x;
}
```

Reconstructability

The reconstructability requirement imposes additional storage requirements on sparse representations



Storage Requirements

N - number of elements in the vector
S - sparsity level [0 -1], 1 being fully-dense

Assume indices and values are the same size (e.g. 32-bit integers and floats)

0	7	0	0	4	0	1	0	0	0	8	4	0	0	6	0

Dense representation:
`N` words

7	4	1	8	4	6
1	4	6	10	11	14

Sparse representation:
`2NS` words

The sparse representation only saves space if `S < 1/2`

Sparse Matrices

A matrix with a majority of nonzero elements

Frequently used to solve systems of linear equations with sparse dependencies

$$3x_1 + x_3 = y_1$$

$$-x_2 = y_2$$

$$2x_2 + 4x_3 + x_4 = y_3$$

$$x_1 + x_4 = y_4$$

A:

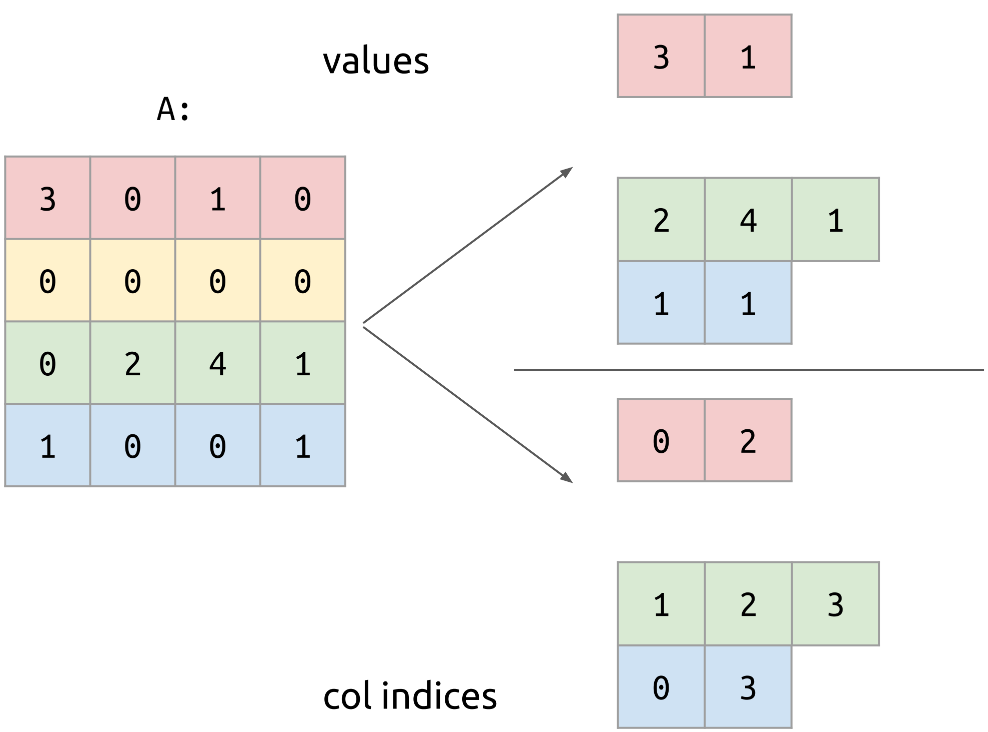
3	0	1	0
0	-1	0	0
0	2	4	1
1	0	0	1

$$Ax = y$$

Compressed Sparse Row (CSR) Format

High level idea: store each row as a sparse (row) vector

Each row is of variable length depending on the sparsity pattern



Compressed Sparse Row (CSR) Format

High level idea: store each row as a sparse (row) vector

Each row is of variable length depending on the sparsity pattern

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

CSR Format Storage Requirement

M - number of rows in the matrix
N - number of columns in the matrix
S - sparsity level [0 -1], 1 being fully-dense

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

Dense representation

MN

Sparse representation

$2MNS + M + 1$

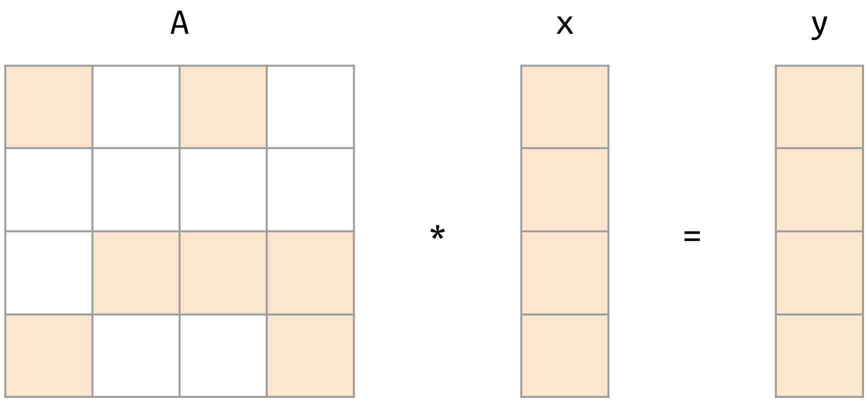
CSR only saves space if $S < (1 - N^{-1}) / 2$

Sparse Matrix-Vector Multiplication (SpMV)

We’ll consider the application of multiplying a sparse matrix and a dense vector

Commonly used in graph-based applications

This is the core computation of iterative methods for solving sparse systems of linear equations:



A CSR Struct

```
struct SparseMatrixCSR {
     float * values;
     int * col_indices;
     int * row_indices;
     int M;
     int N;
};
```

We assume `row_indices` is of length `M+1`

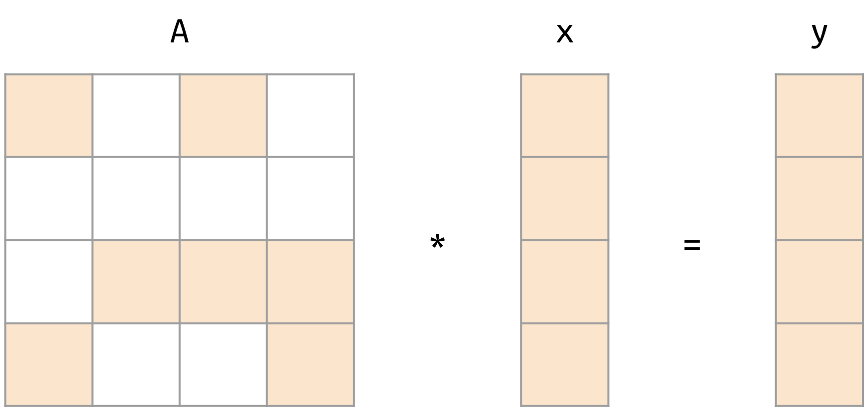
We assume `col_indices` and `values` are of length `row_indices[M]`

Sequential SpMV / CSR

```
void SpMV_CSR(const SparseMatrixCSR A, const float * x, float * y) {

     for (int row = 0; row < A.M; ++row) {

          float dotProduct = 0;
          const int row_start = A.row_indices[row];
          const int row_end = A.row_indices[row+1];
          for (int element = row_start; element < row_end; ++element) {

               dotProduct += A.values[element] * x[A.col_indices[element]];

          }

          y[row] = dotProduct;

     }

}
```

This for loop iterates `row_end - row_start` times.

`row_end - row_start` depends on the row

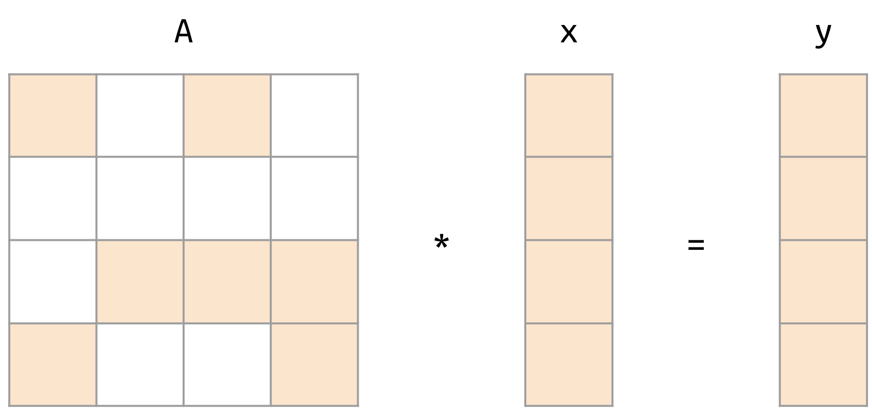


Sequential SpMV / CSR

```
void SpMV_CSR(const SparseMatrixCSR A, const float * x, float * y) {

     for (int row = 0; row < A.M; ++row) {

          float dotProduct = 0;
          const int row_start = A.row_indices[row];
          const int row_end = A.row_indices[row+1];
          for (int element = row_start; element < row_end; ++element) {

               dotProduct += A.values[element] * x[A.col_indices[element]];

          }

          y[row] = dotProduct;

     }

}
```

This for loop iterates `row_end - row_start` times.

`row_end - row_start` depends on the row



Parallel SpMV / CSR

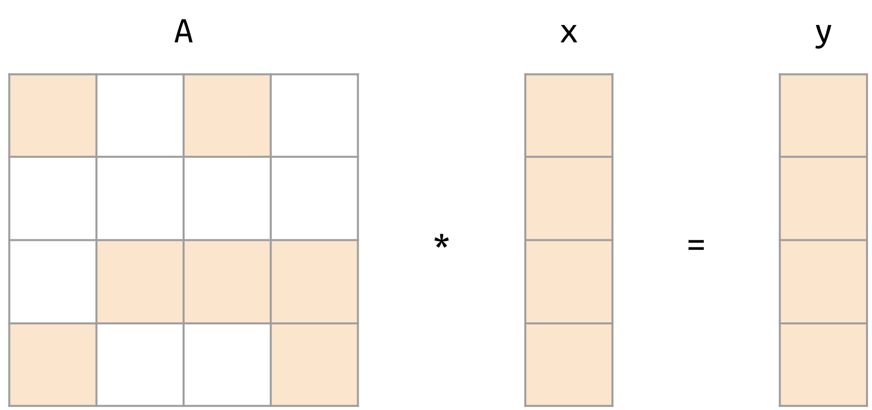
As in dense matrix - vector multiplication, SpMV is data parallel

We can compute the dot product of each row of `A` with `x` in parallel

```
void SpMV_CSR(const SparseMatrixCSR A, const float * x, float * y) {
    #pragma omp parallel for
    for (int row = 0; row < A.M; ++row) {

        float dotProduct = 0;
        const int row_start = A.row_indices[row];
        const int row_end = A.row_indices[row+1];
        for (int element = row_start; element < row_end; ++element) {

            dotProduct += A.values[element] * x[A.col_indices[element]];

        }

        y[row] = dotProduct;

    }

}
```

This for loop iterates `row_end - row_start` times.

`row_end - row_start` depends on the row



Think about the performance problems

Think about the performance problems

- Load imbalance

ELL Sparse Matrix Format

The name derives from the sparse matrix package in ELLPACK, a tool for solving elliptic boundary problems

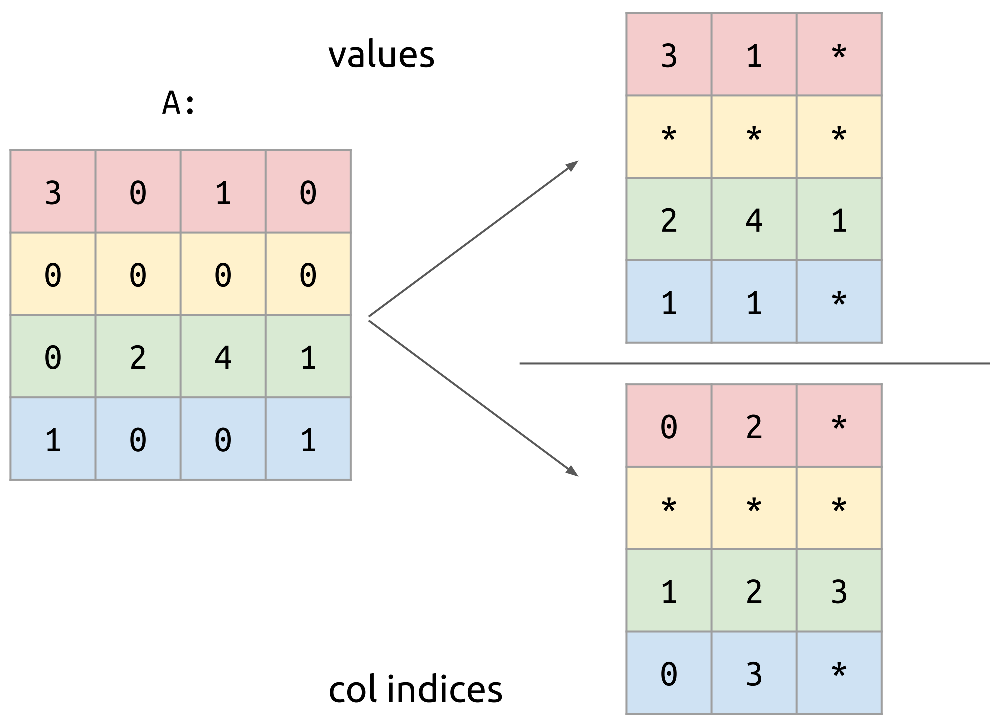
ELL builds on CSR with two modifications:

1. Padding
2. Transposition

Padding

To form a padding representation, identify the longest row

Allocate for each row enough space to hold the data for the longest row



We must pick a strategy for setting the padded values to satisfy the reconstructability requirement

Padding

Option A:

Place zeros in values
Give the column index of an actual 0

A:

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

values

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

col indices

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

Padding

Option B:

Place an invalidating indicator into either array
Requires algorithmic adjustment

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

Padding

Option B:

Place an invalidating indicator into either array
Requires algorithmic adjustment

A:

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

values

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

col indices

0	2	*
*	*	*
1	2	3
0	3	*

Transposition

Store the sparsified matrix in a column-major format (i.e. all elements in the same column are in contiguous memory locations)

This is the default for FORTRAN, but not for C

values

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

→

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

0	2	*
*	*	*
1	2	3
0	3	*

→

0	*	1	0	2	*	2	3	*	*	3	*

col indices

Storage Requirements

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 Requirement (words)
Dense	`MN`
Compressed Sparse Row (CSR)	`2MNS + M + 1`
ELL	`2MK`

ELL only saves space if `K < N / 2`

An ELL Struct

```
struct SparseMatrixELL {
     float * values;
     int * col_indices;
     int M;
     int N;
     int K;
};
```

We assume `col_indices` and `values` are of length `M * K;`

Parallel SpMV / ELL Kernel

```
#pragma omp parallel for
  for(int i=0; i<A.M;i++)
{
     float dotProduct = 0;
     for (int element = 0; element < A.K; ++element) {
       const int elementIndex = row + element* A.M;
       dotProduct += A.values[elementIndex] * x[A.col_indices[elementIndex]];
     }
     y[row] = dotProduct;
}
```

note that all threads have same amount of work

Parallel SpMV / ELL Kernel

```
#pragma omp parallel for
  for(int i=0; i<A.M;i++)
{
     float dotProduct = 0;

     for (int element = 0; element < A.K; ++element) {

       const int elementIndex = row + element* A.M;
       dotProduct += A.values[elementIndex] * x[A.col_indices[elementIndex]];

     }

     y[row] = dotProduct;

}
```

note that all threads have same amount of work

- Can we avoid multiplication by "0"?
- What are the advantages and disadvantages?

SpMV / ELL Kernel Shortcomings

This kernel will perform very well for matrices with similarly-dense rows

This approach is not equally well suited to all possible inputs

Consider a 1000 x 1000 matrix with sparsity level 0.01:

- There are 1000 * 1000 * 0.01 = 10,000 multiply / adds to do
- If the densest row has 200 nonzero values, then the kernel will perform 1000 * 200 = 200,000 multiply adds
- By using an ELL representation, we have increased the amount of computation AND memory access by 20x
- This is really bad worst-case performance!

The Coordinate (COO) Format

High-level idea: store both the column index AND row index for every nonzero

This introduces additional storage for the extra index

There is no longer any required ordering for the elements

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

→

values:	3	1	2	4	1	1	1
column indices:	0	2	1	2	3	0	3
row indices:	0	0	2	2	2	3	3

The Coordinate (COO) Format

High-level idea: store both the column index AND row index for every nonzero

This introduces additional storage for the extra index

There is no longer any required ordering for the elements

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

values:	1	2	1	3	4	1	1
column indices:	0	1	3	0	2	3	2
row indices:	3	2	2	0	2	3	0

Storage Requirements

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 Requirement (words)
Dense	MN
Compressed Sparse Row (CSR)	2MNS + M + 1
ELL	2MK
Coordinate (COO)	3MNS

COO only saves space if $S < 1 / 3$

A COO Struct

```
struct SparseMatrixCOO {
     float * values;
     int * col_indices;
     int * row_indices;
     int M;
     int N;
     int count;
};
```

We assume `row_indices`, `col_indices`, and `values` are of length `count`

Sequential SpMV / COO

```
void SpMV_COO(const SparseMatrixCOO A, const float * x, float * y) {

     for (int element = 0; element < A.count; ++element) {

          const int column = A.col_indices[element];
          const int row = A.row_indices[element];

          y[row] += A.values[element] * x[column];

     }

}
```

This is a very satisfyingly simple function

Compared to the sequential SpMV / CSR, the sequential SpMN / COO doesn’t waste time with fully-zero rows

```
#pragma omp parallel for
for (int element = 0; element < A.count; ++element) {

     const int column = A.col_indices[element];
     const int row = A.row_indices[element];

     y[row] += A.values[element] * x[column];

}
```

Is this code correct?

```
#pragma omp parallel for

for (int element = 0; element < A.count; ++element) {

    const int column = A.col_indices[element];
    const int row = A.row_indices[element];

    y[row] += A.values[element] * x[column];      Take a look at this statement

}
```

Is this code correct?

```
#pragma omp parallel for
for (int element = 0; element < A.count; ++element) {

    const int column = A.col_indices[element];
    const int row = A.row_indices[element];

    #pragma omp atomic
    y[row] += A.values[element] * x[column];

}
```

Take a look at this statement

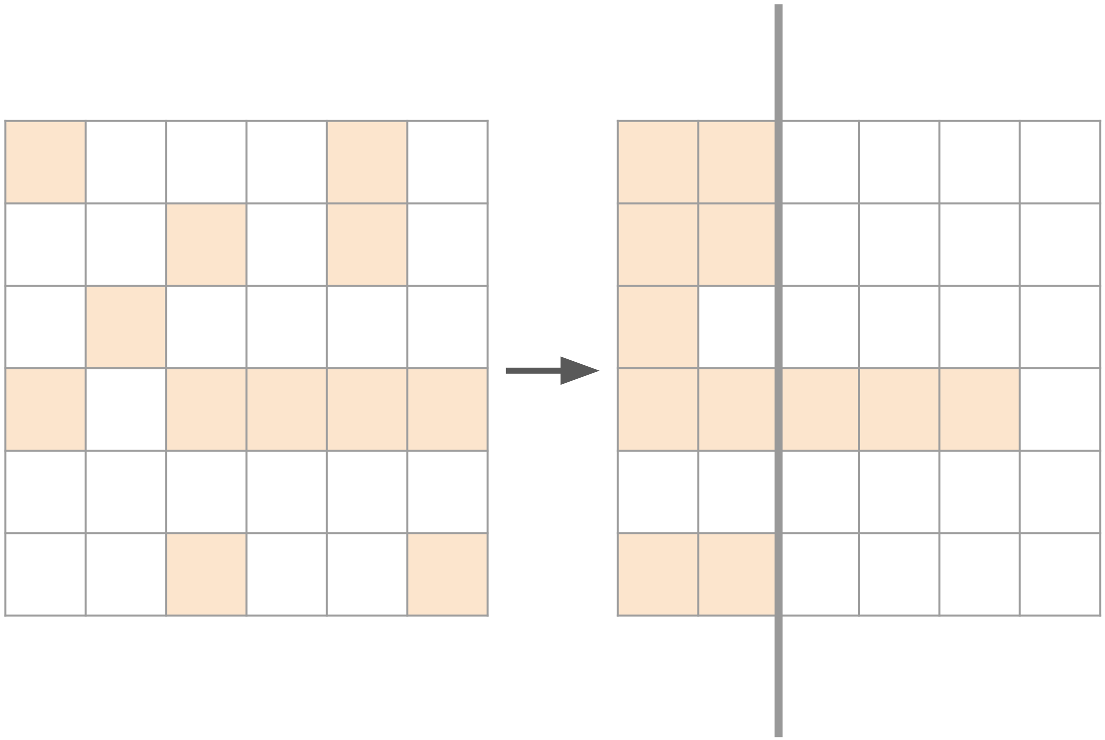
Is this code correct?

Write collision from multiple threads

Hybrid ELL / COO Representation

High-level idea: place nonzeros 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

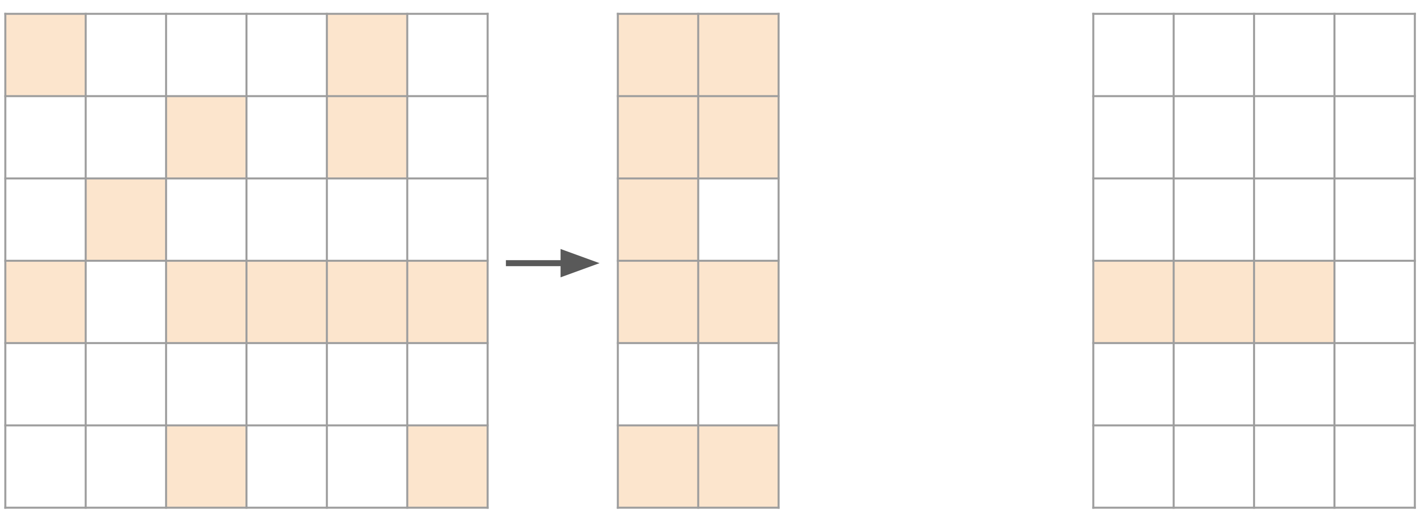


Hybrid ELL / COO Representation

High-level idea: place nonzeros 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

ELL COO



2 * 6 * 2 = 24 words

3 * 3 = 9 words

Hybrid ELL / COO Representation

A:

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

ELL:

values:

3	1
0	0
2	4
1	1

column indices

0	2
*	*
1	2
0	3

COO:

values:	1
column indices:	3
row indices:	2

Storage Requirements

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 Requirement (words)
Dense	`MN`
Compressed Sparse Row (CSR)	`2MNS + M + 1`
ELL	`2MK`
Coordinate (COO)	`3MNS`
Hybrid ELL / COO (HYB)	It's complicated!

Storage Requirements

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 Requirement (words)**
Dense	`MN`
Compressed Sparse Row (CSR)	`2MNS + M + 1`
ELL	`2MK`
Coordinate (COO)	`3MNS`
Hybrid ELL / COO (HYB)	`> 3MNS,` `< 2MK`

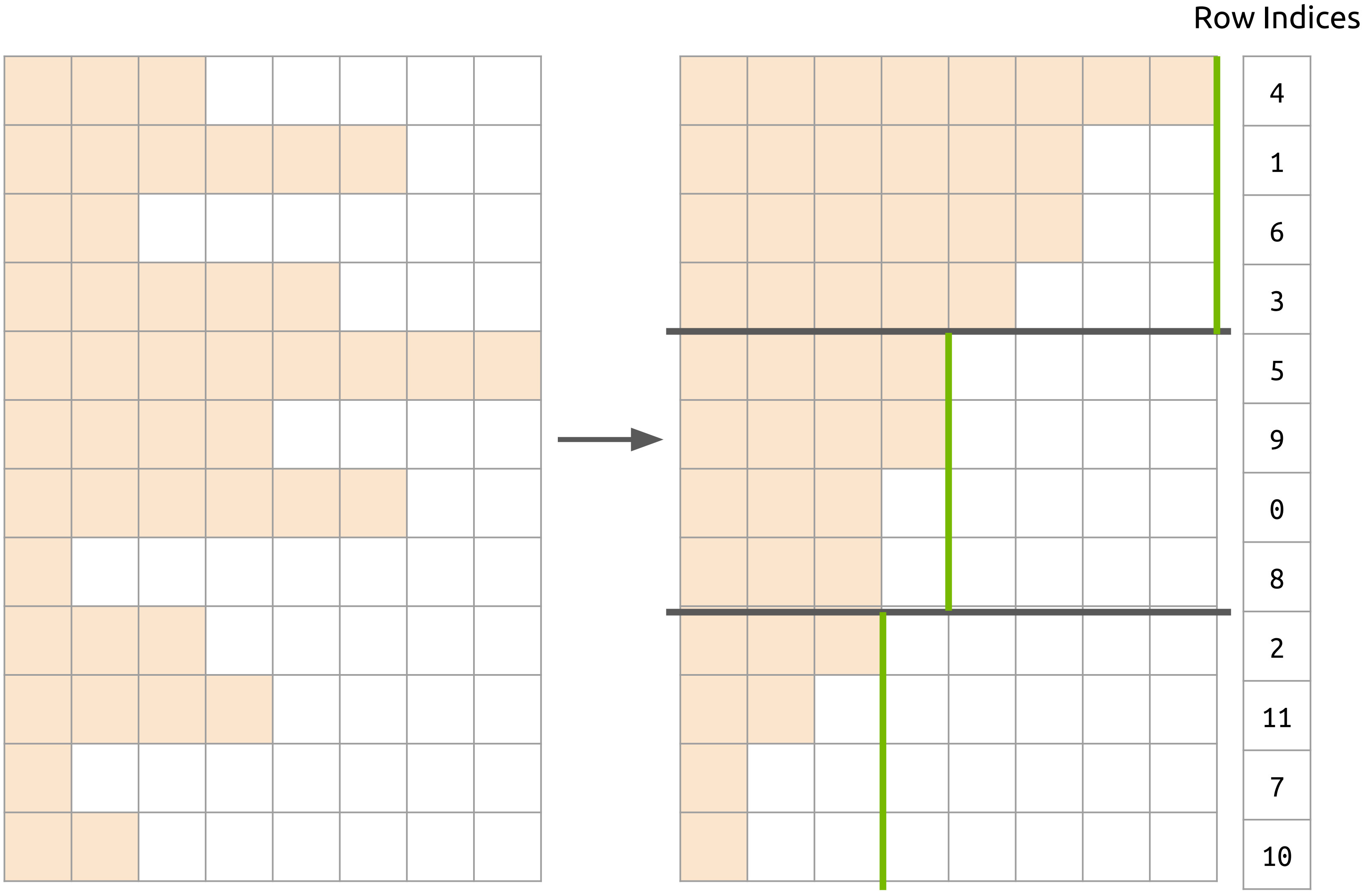
Jagged Diagonal Storage (JDS) Format

High-level idea: Group similarly dense rows into evenly-sized partitions, and represent each section independently using either CSR or ELL

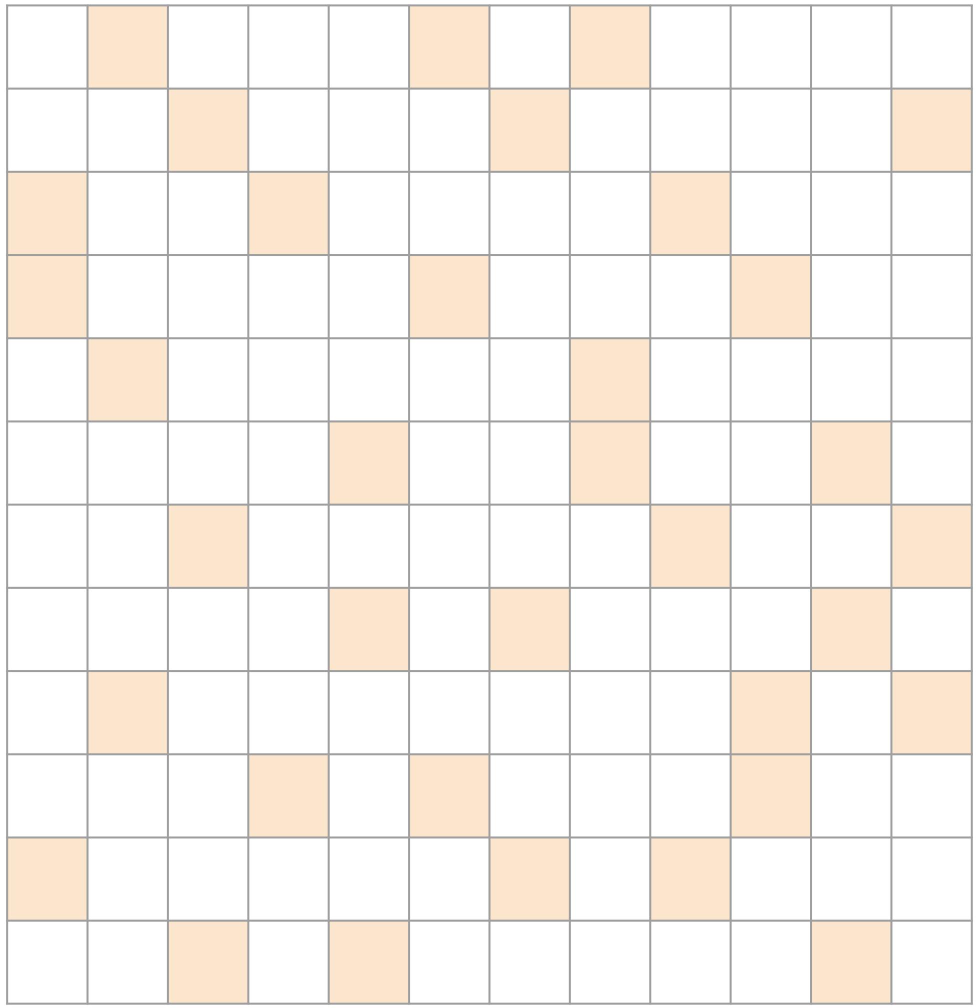
This can be done by sorting rows by density

We need to store the original indices of the sorted rows to satisfy the reconstructability requirement

Jagged Diagonal Storage (JDS) Format



Best Format for SpMV?

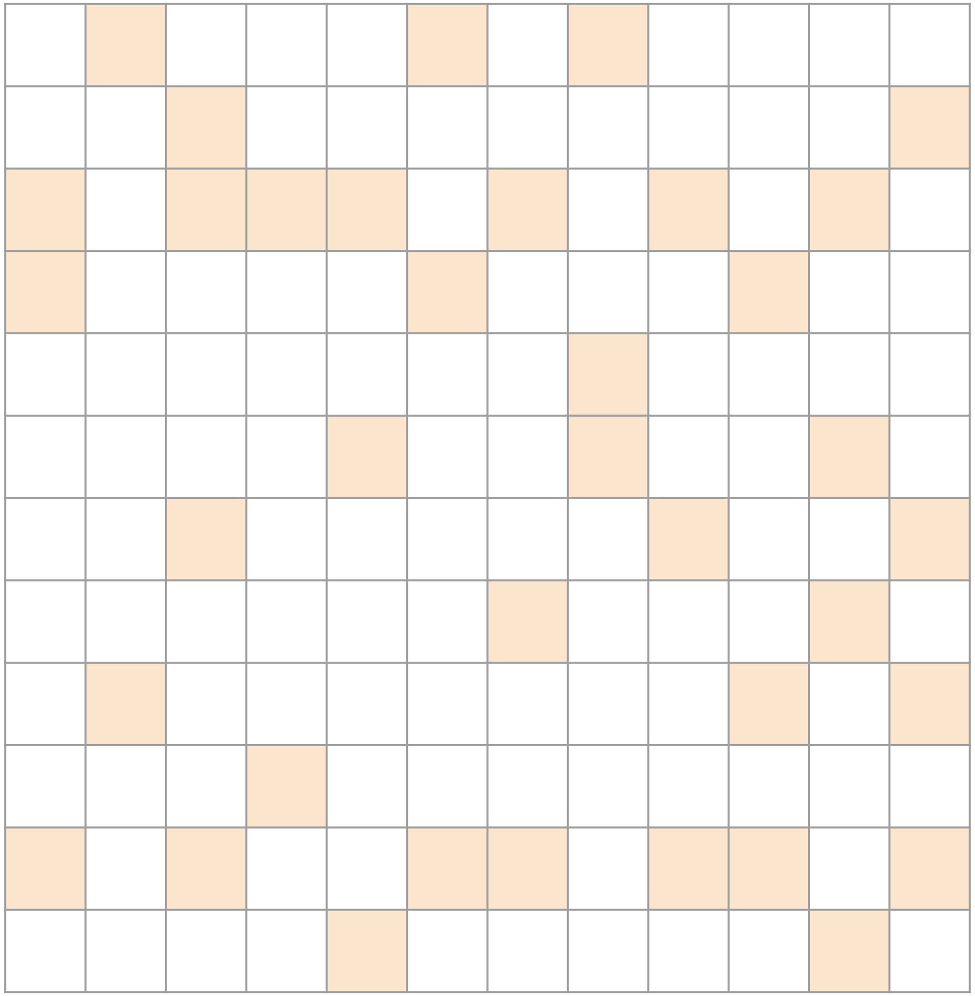


Roughly random?

Probably best with ELL

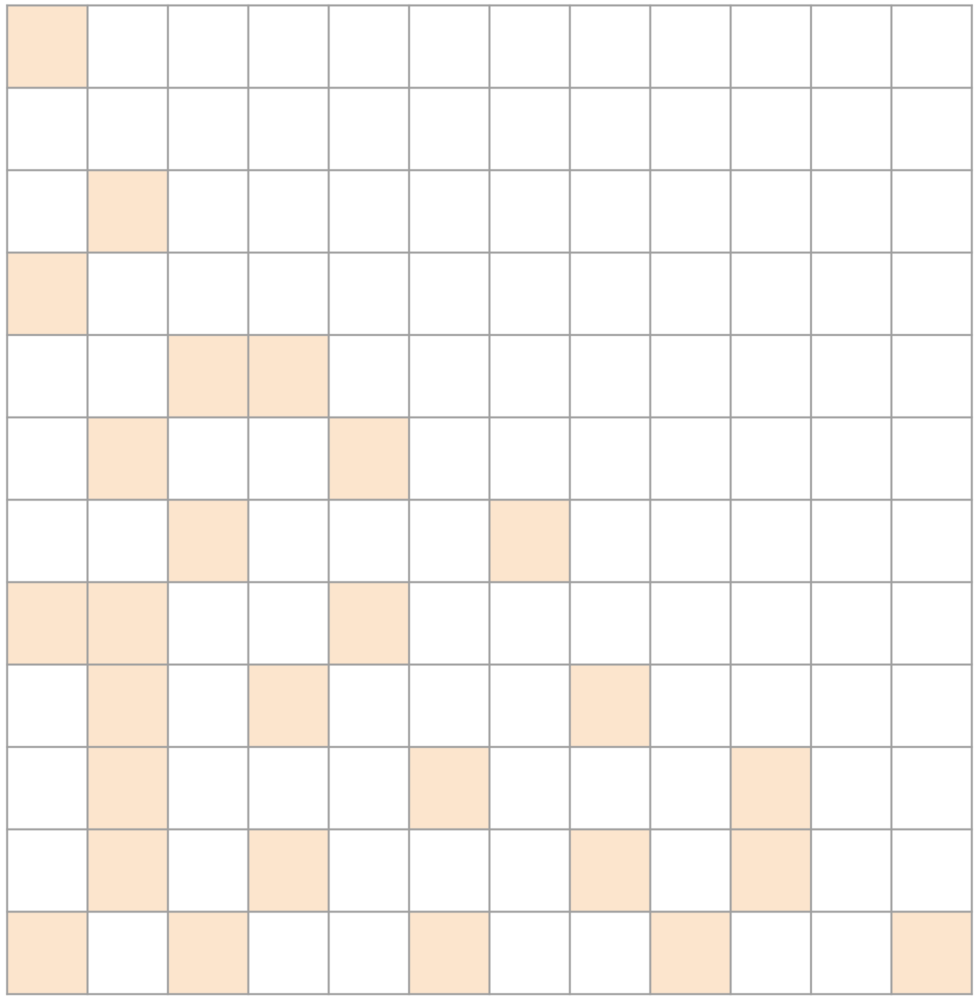
Best Format for SpMV?

Roughly random, but with more variance in the sparsity between rows?



Probably best with a hybrid COO / ELL representation

Best Format for SpMV?

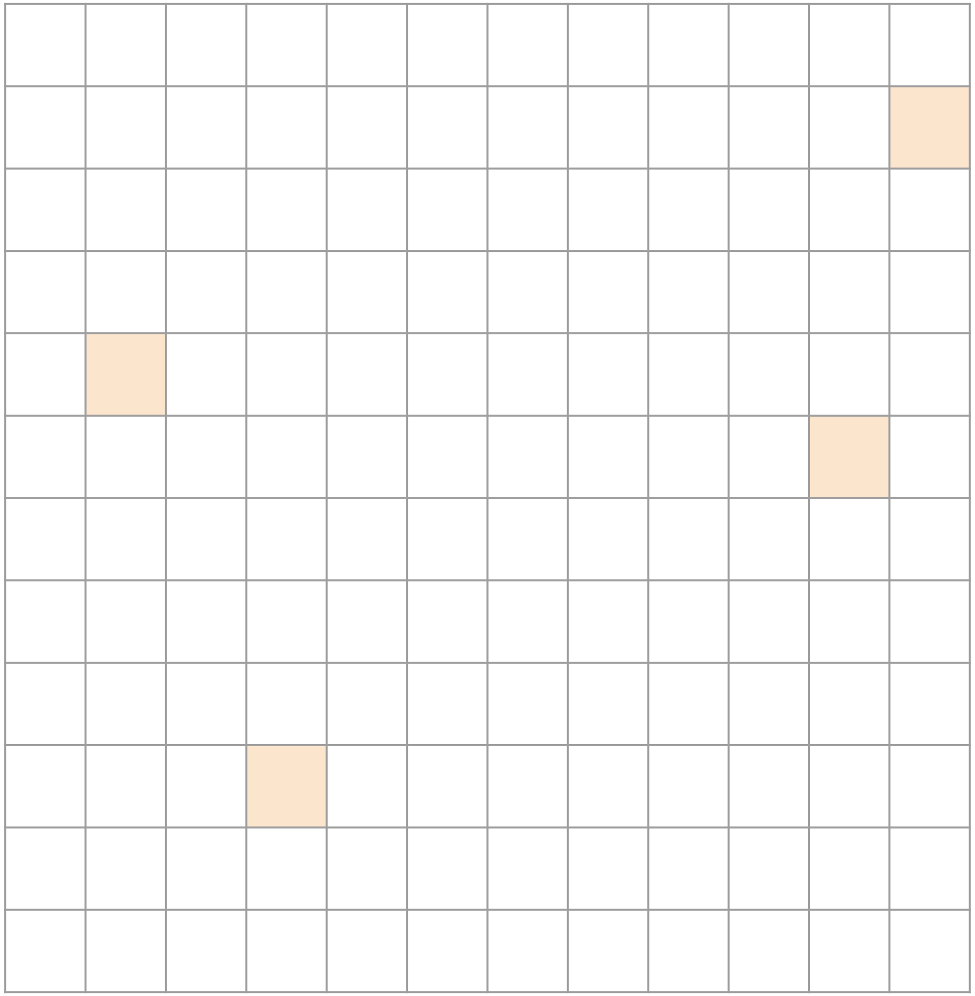


Roughly triangular?

Probably best with JDS

Best Format for SpMV?

Extremely sparse?



Probably best with COO

Other Sparse Matrix Representations

- **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

Conclusion / Takeaways

- Sparse matrices are hard!
- There are a lot of ways to represent sparse matrices
- Different representations have different storage requirements
- The storage requirements depend differently on the sparsity pattern
- There is sometimes a need to safeguard against worst-case input
- There is often a trade-off between regularity and efficiency
- Some representations are better suited to certain hardware than others
- It can be difficult to achieve a high compute-to-global-memory-access ratio (CGMA) when it comes to sparse matrices
    - The above is especially true in the case of SpMV, where each row participates in a separate computation

Sources

Bell, Nathan, and Michael Garland. Efficient sparse matrix-vector multiplication on CUDA. Vol. 2. No. 5. Nvidia Technical Report NVR-2008-004, Nvidia Corporation, 2008.

Cheng, John, Max Grossman, and Ty McKercher. Professional Cuda C Programming. John Wiley & Sons, 2014.

Hwu, Wen-mei, and David Kirk. "Programming massively parallel processors." Special Edition 92 (2009).