CSE 599 I
Accelerated Computing - Programming GPUS

Parallel Pattern: Sparse Matrices

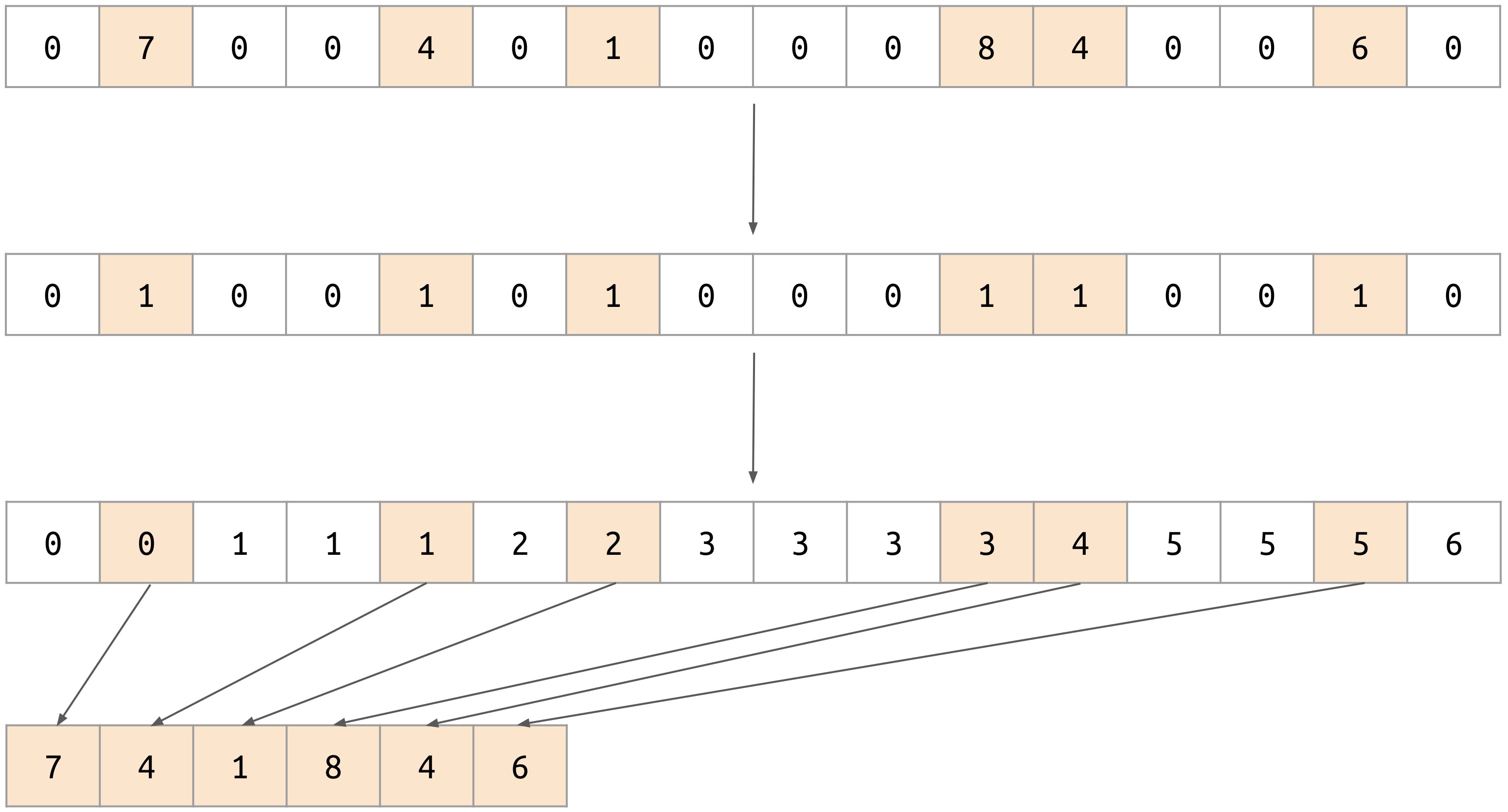
Objective

- 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

Sparse Vector Representation

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

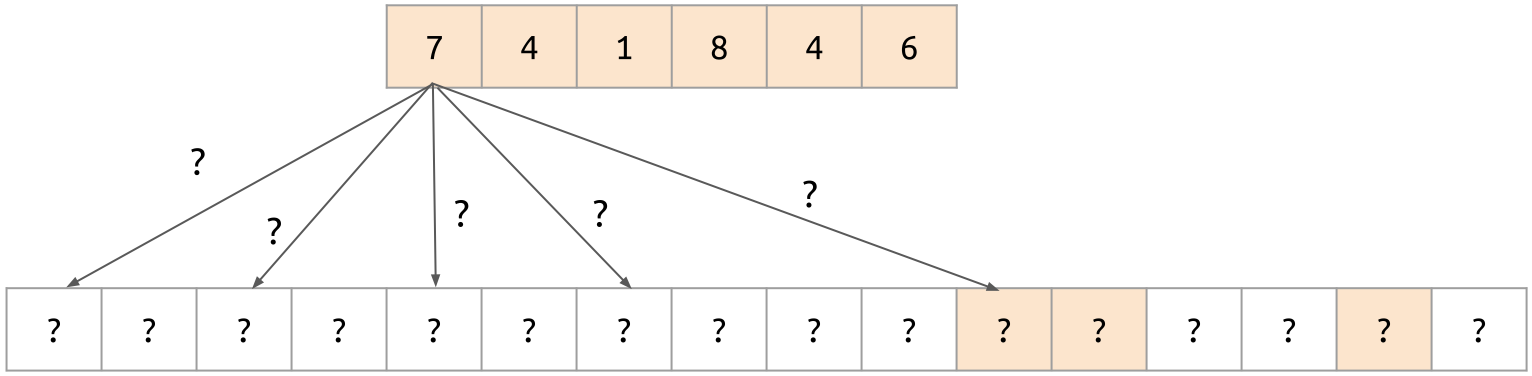
Sparse Vector Representation



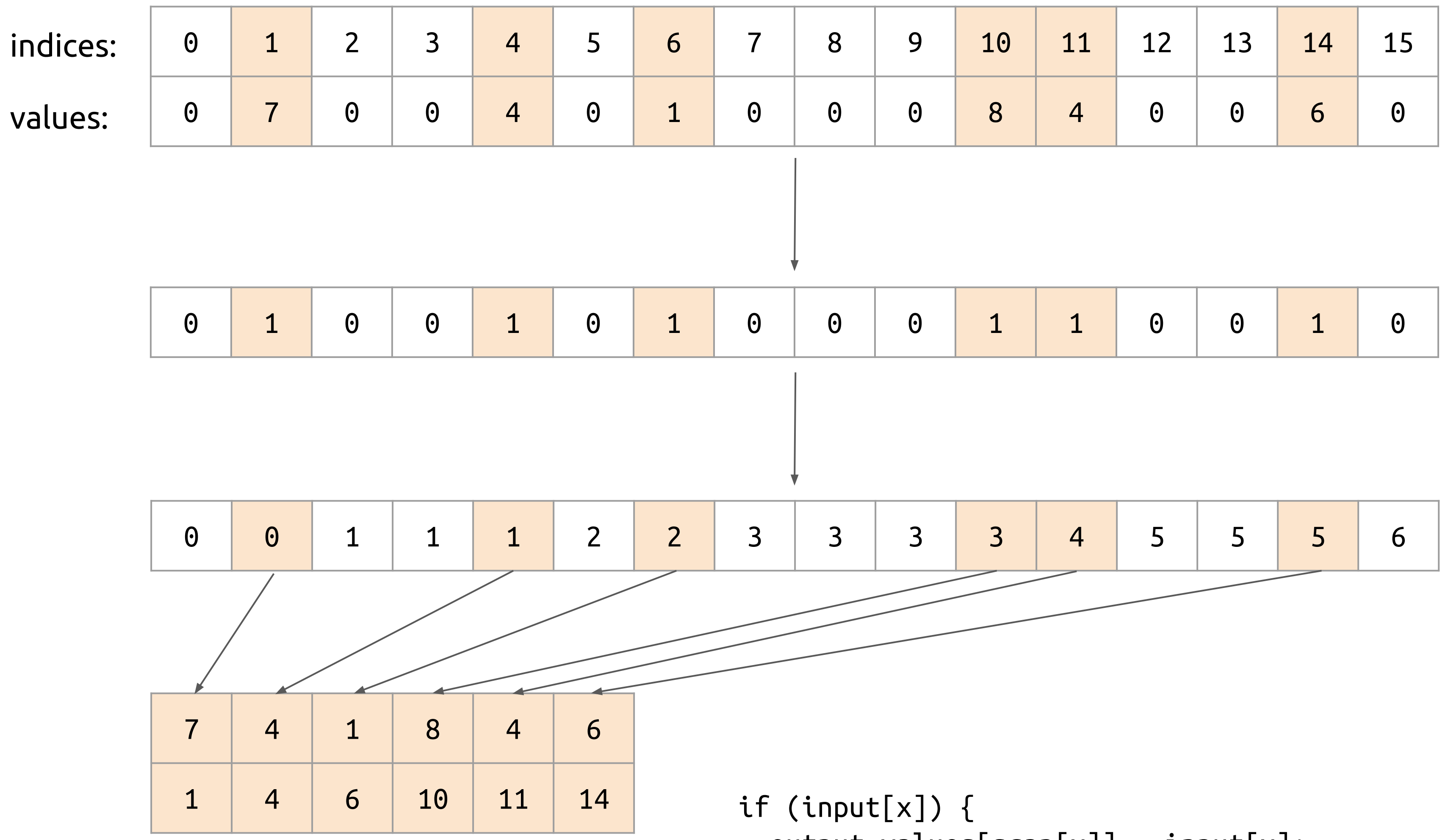
```
if (input[x]) {
  output_values[scan[x]] = input[x];
}
```

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

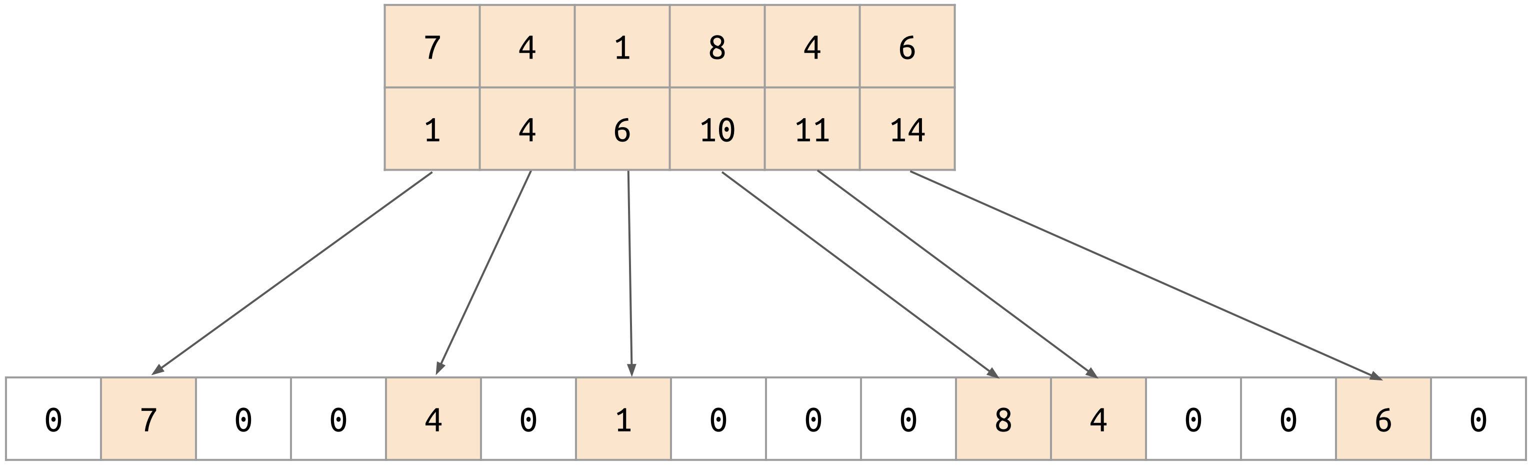
0	0	1	1	1	2	2	3	3	3	3	4	5	5	5	6

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

$3x_1 + x_3 = y_1$

$-x_2 = y_2$

$2x_2 + 4x_3 + x_4 = y_3$

$x_1 + x_4 = y_4$

$\longrightarrow$

A:

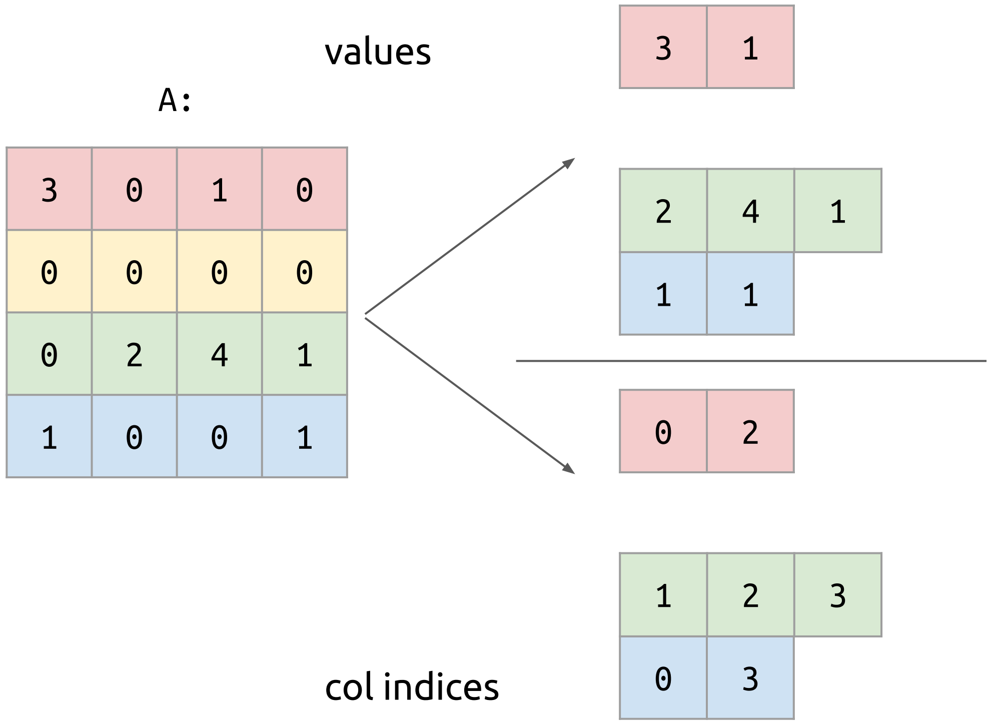
3	0	1	0
0	-1	0	0
0	2	4	1
1	0	0	1

$Ax = y$

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

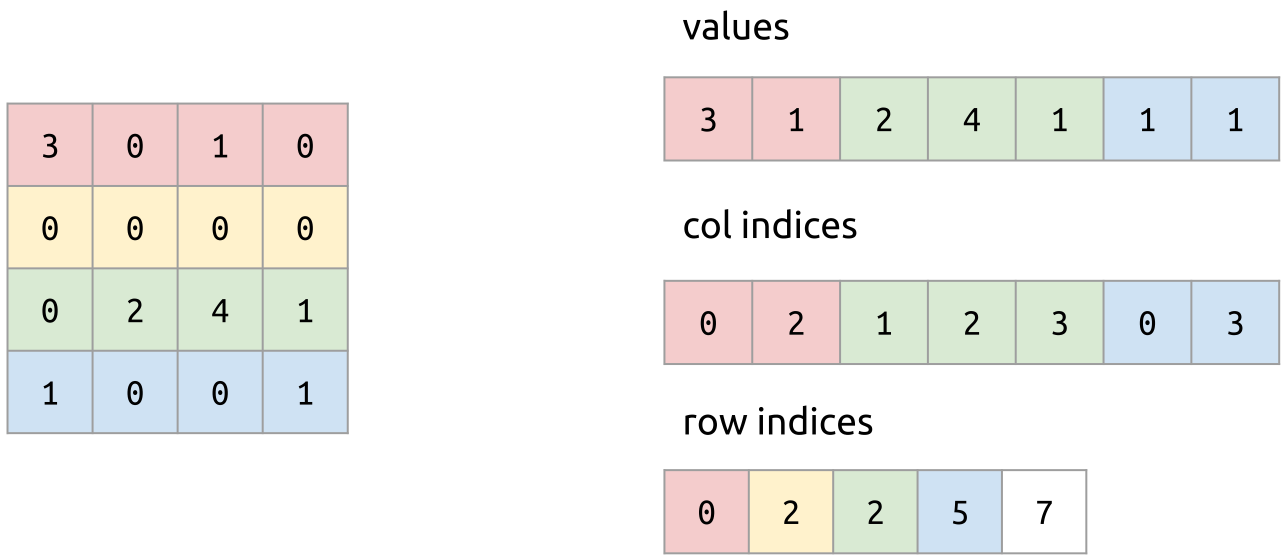
0	2	1	2	3	0	3

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`

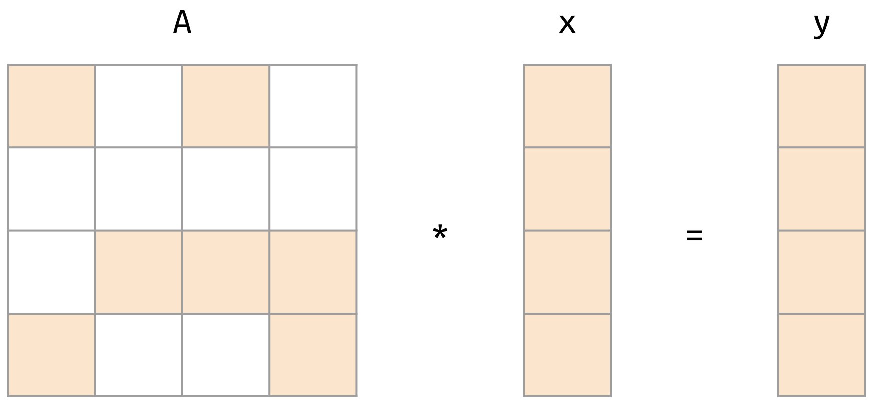
CSR only saves space if $S < (1 - N^{-1}) / 2$

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]`

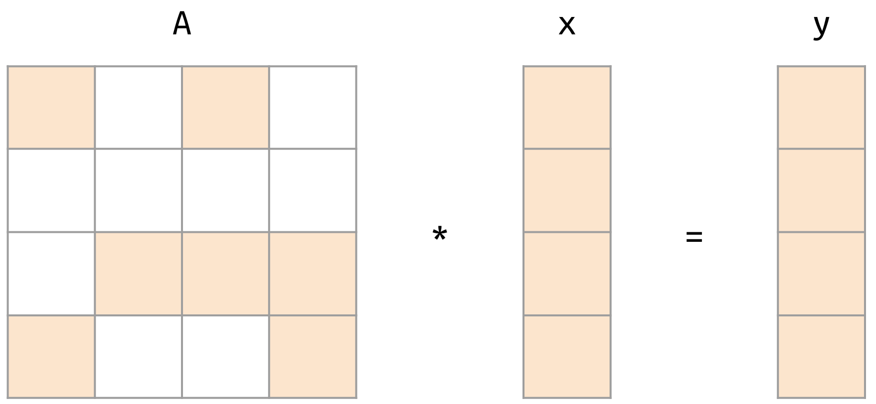
You are allowed to pass structs (and classes) by value to a CUDA kernel!

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

Parallel SpMV / CSR Kernel

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

     const int row = blockIdx.x * blockDim.x + threadIdx.x;

     if (row < A.M) {

          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;

     }

}
```

Parallel SpMV / CSR Kernel Shortcomings

1. Non-coalesced memory access

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

   Neighboring threads process neighboring rows, resulting in strided access

2. Control flow divergence

   Each row involves a variable amount of computation, which depends on the input

3. Additional global memory reads

```
const int row_start = A.row_indices[row];
const int row_end = A.row_indices[row+1];
```

   This will decrease our computation-to-global-memory-access (CGMA) ratio, which can have significant impact for smaller and / or highly - sparse matrices

Instead of “fixing” this kernel, let’s consider other representations

ELL Sparse Matrix Format

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

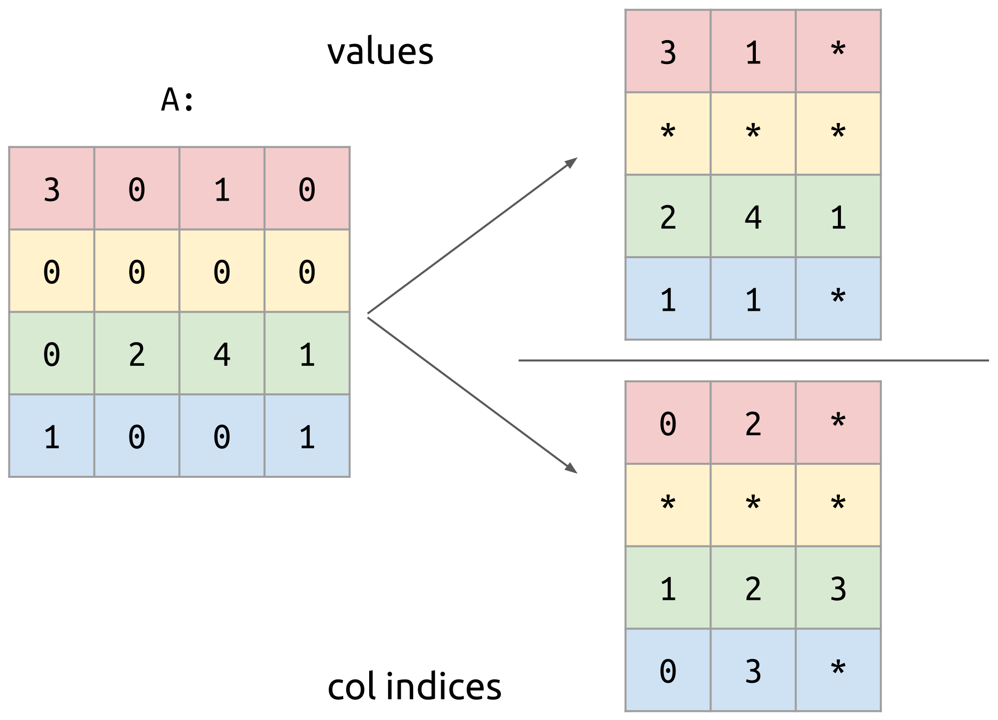
ELL builds on CSR with two modifications:

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



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	*

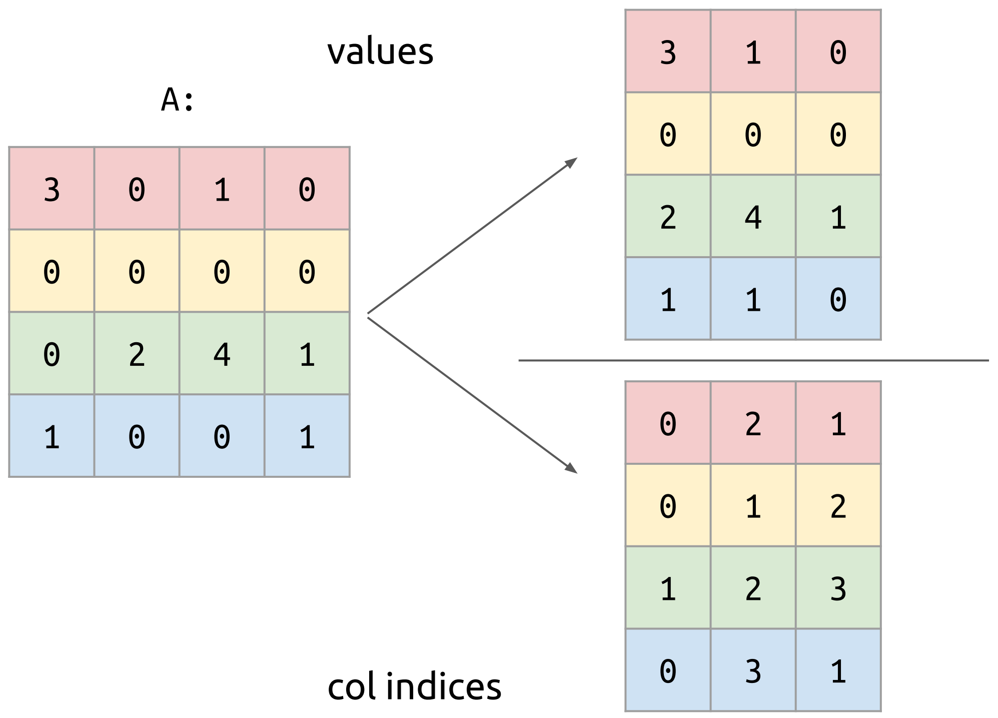
We must pick a strategy for setting the padded values to satisfy the reconstructability requirement

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

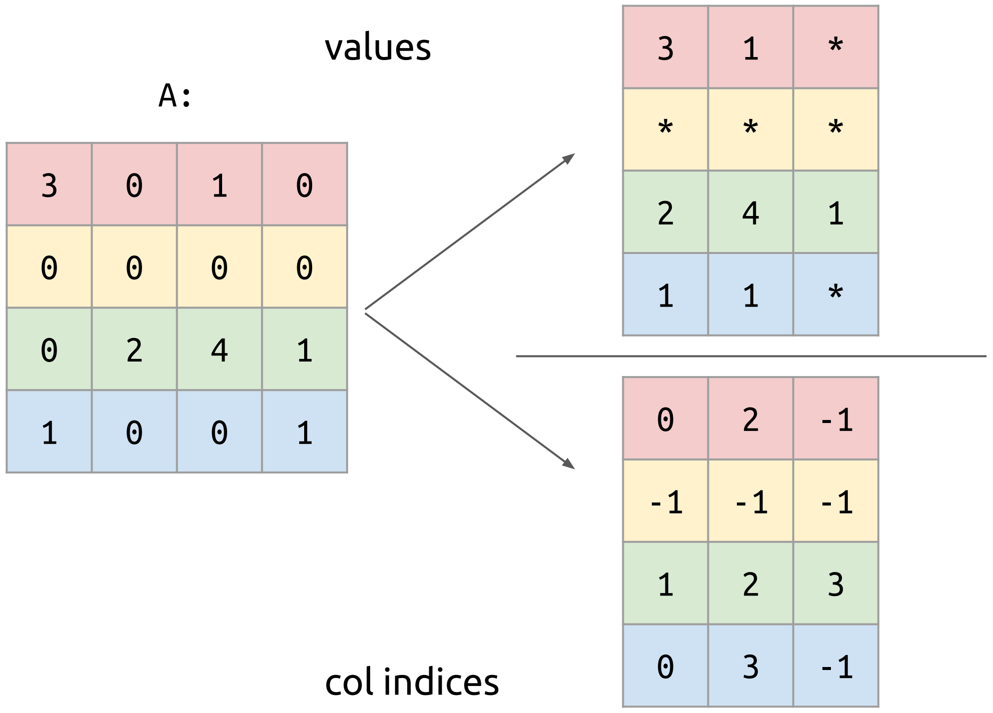
col indices

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	*

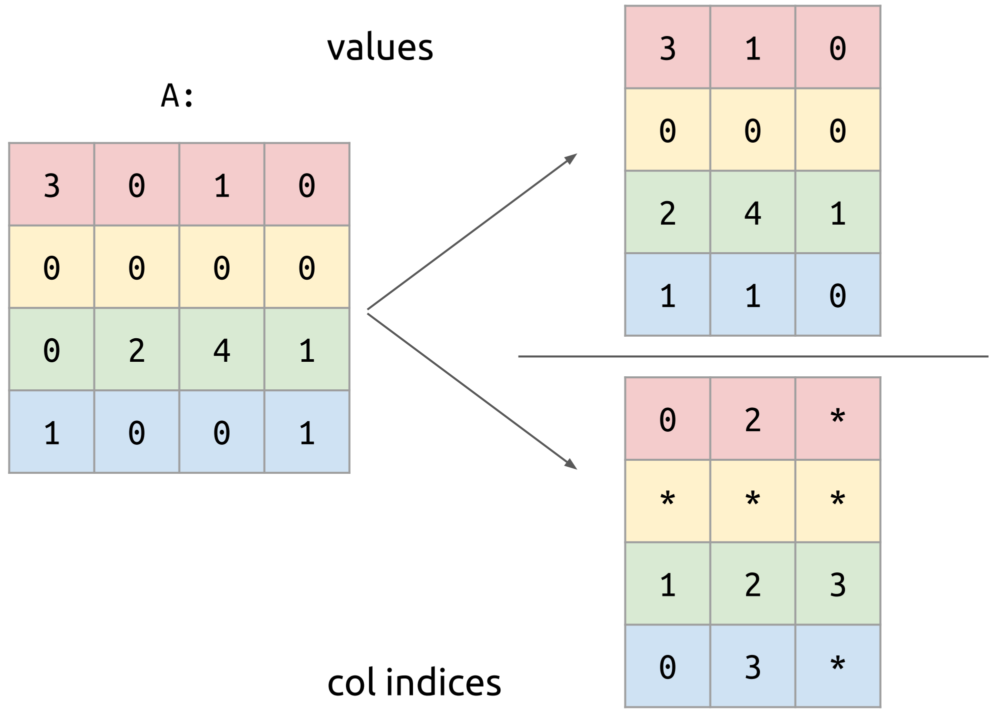
col indices

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

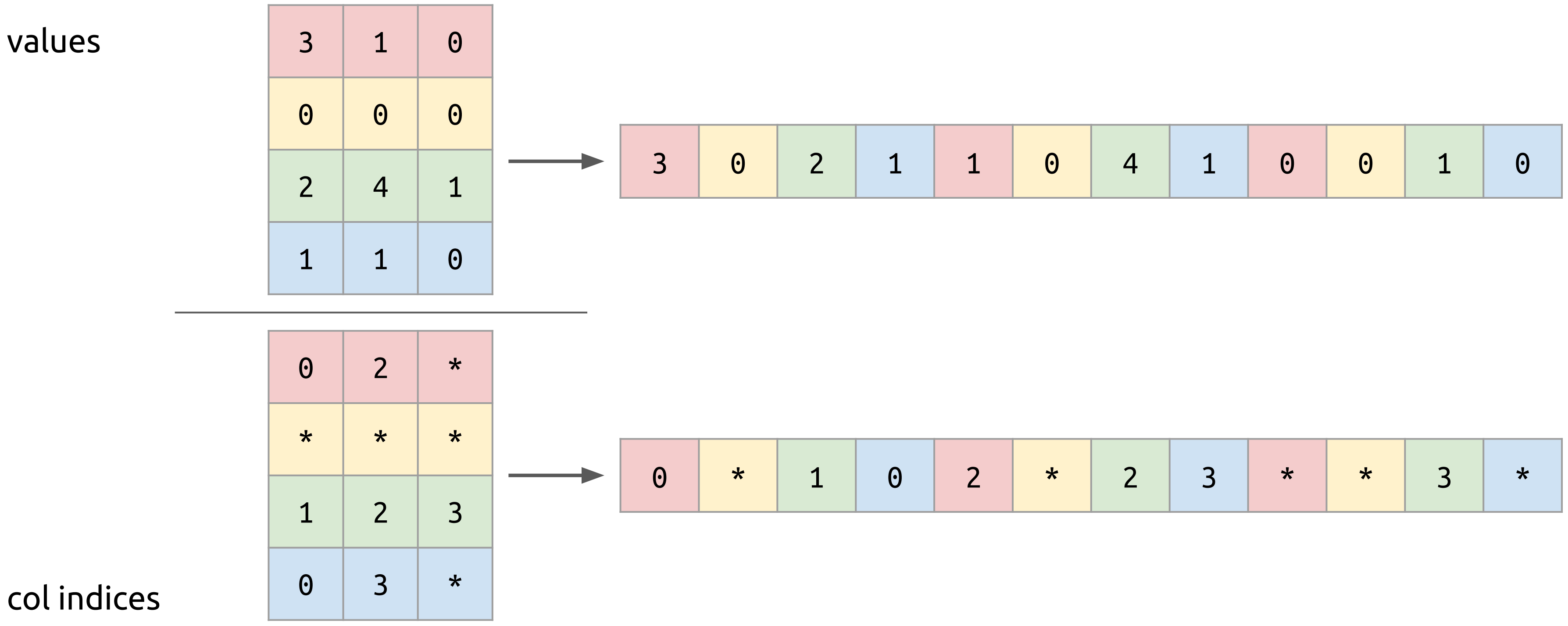
col indices

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



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 (Option A)

```
__global__ void SpMV_ELL_kernel(const SparseMatrixELL A, const float * x, float * y) {

     const int row = blockIdx.x * blockDim.x + threadIdx.x;

     if (row < A.M) {

          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;

     }

}
```

Global memory access depends on `row`, which has consecutive values for consecutive `threadIdx.x`

All threads iterate the same number of times

Global memory access for row indices is no longer required

Parallel SpMV / ELL Kernel (Option B)

```
__global__ void SpMV_ELL_kernel(const SparseMatrixELL A, const float * x, float * y) {

    const int row = blockIdx.x * blockDim.x + threadIdx.x;

    if (row < A.M) {

        float dotProduct = 0;

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

            const int elementIndex = row + element* A.M;
            if (!A.values[elementIndex]) {
                dotProduct += A.values[elementIndex] * x[A.col_indices[elementIndex]];
            }

        }

        y[row] = dotProduct;

    }

}
```

We've re-introduced control flow divergence

On the other hand, we avoid multiplication by 0

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:

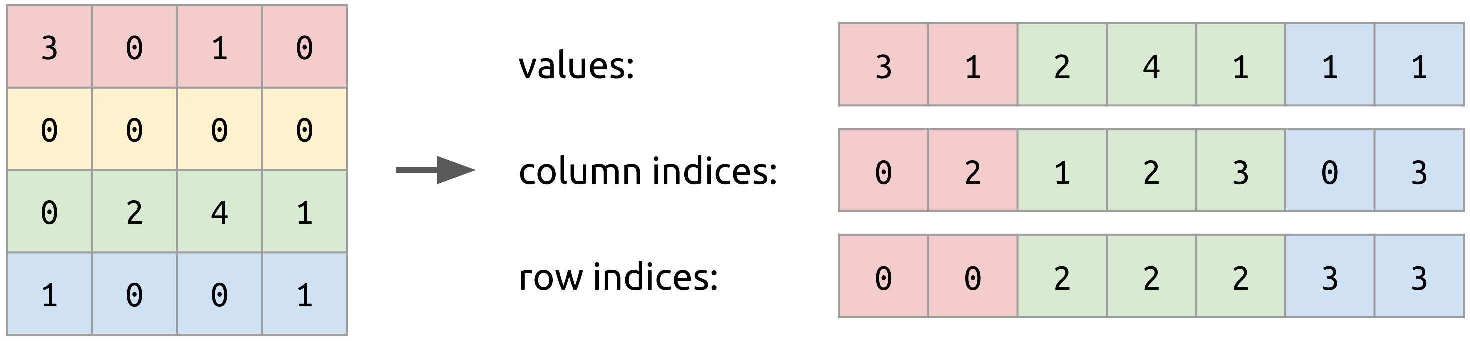
- There are 1000 * 1000 * 0.01 = 10,000 multiply / adds to do
- If the densest row has 200 nonzero values, then the kernel will perform 1000 * 200 = 200,000 multiply adds
- By using an ELL representation, we have increased the amount of computation AND memory access by 20x
- This is really bad worst-case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

$\longrightarrow$

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

Parallel SpMV / COO

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

     for (int element = threadIdx.x + blockIdx.x * blockDim.x;
          element < A.count;
          element += blockDim.x * gridDim.x) {

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

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

     }
}
```

Output interference!

Parallel SpMV / COO

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

     for (int element = threadIdx.x + blockIdx.x * blockDim.x;
          element < A.count;
          element += blockDim.x * gridDim.x) {

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

          atomicAdd(&y[row], A.values[element] * x[column]);

     }

}
```

Switching to an atomic addition will make the output of this kernel correct…

It will also serialize a potentially large number of writes

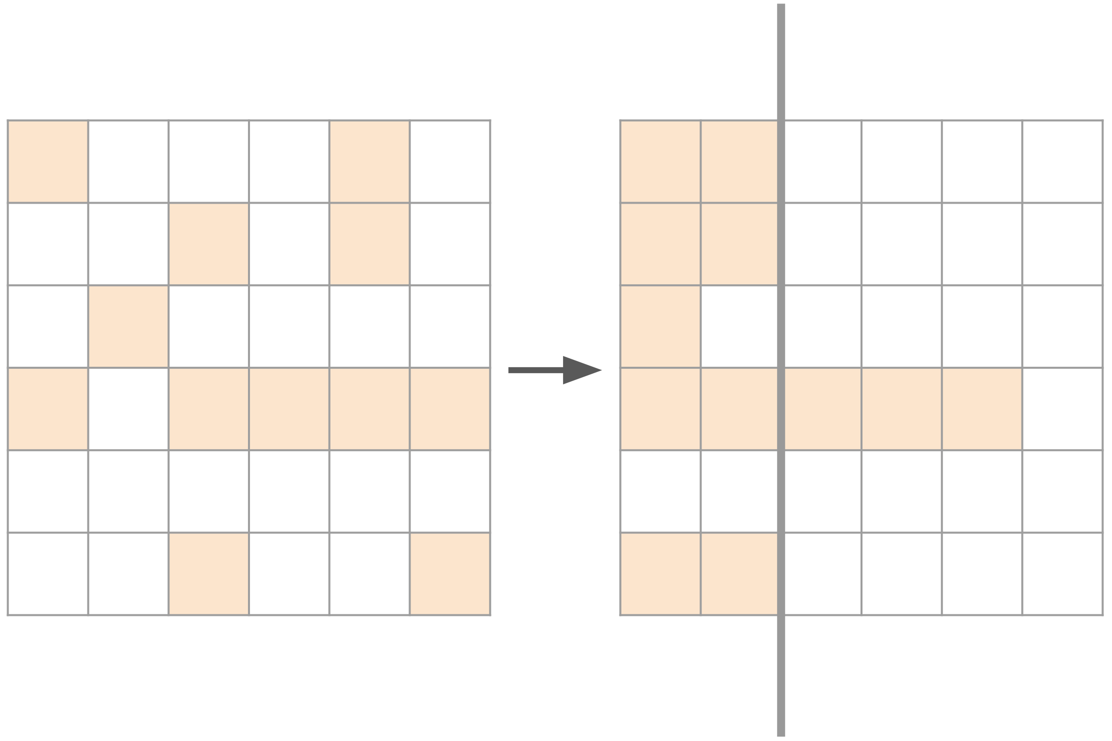
We could solve this using techniques from the histogram pattern (i.e. privitization)

We'll note that this representation is better suited to sequential hardware and take a different approach

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

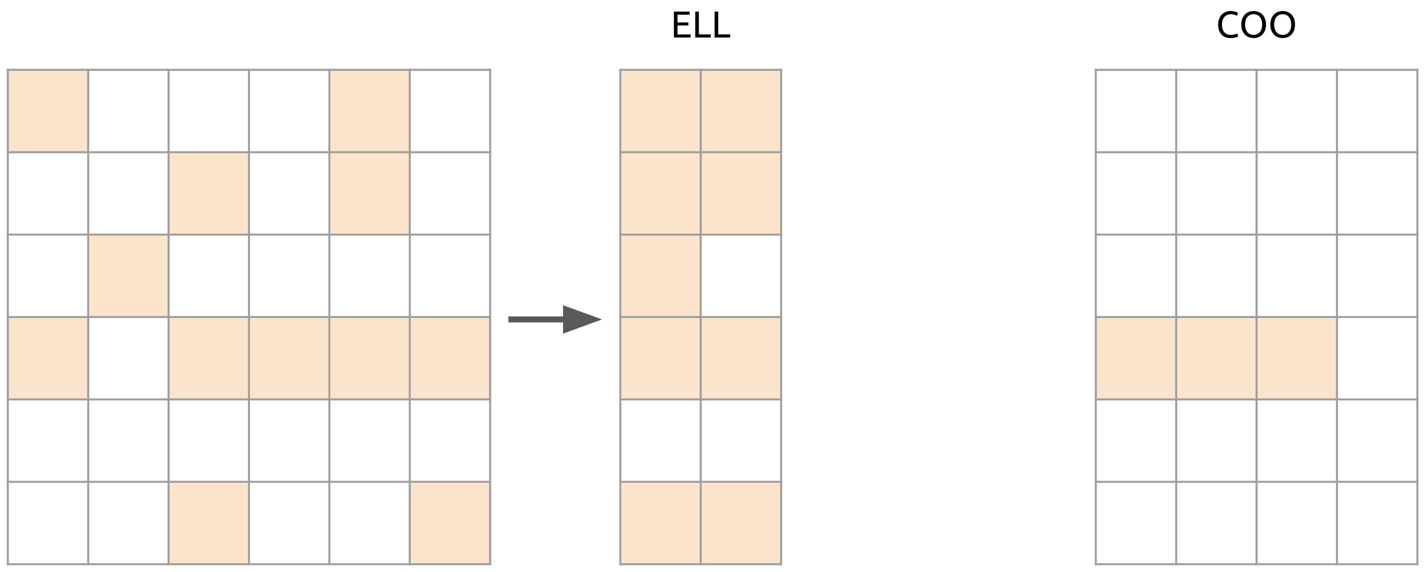
Each element will be stored in the ELL or the COO matrix, not both



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

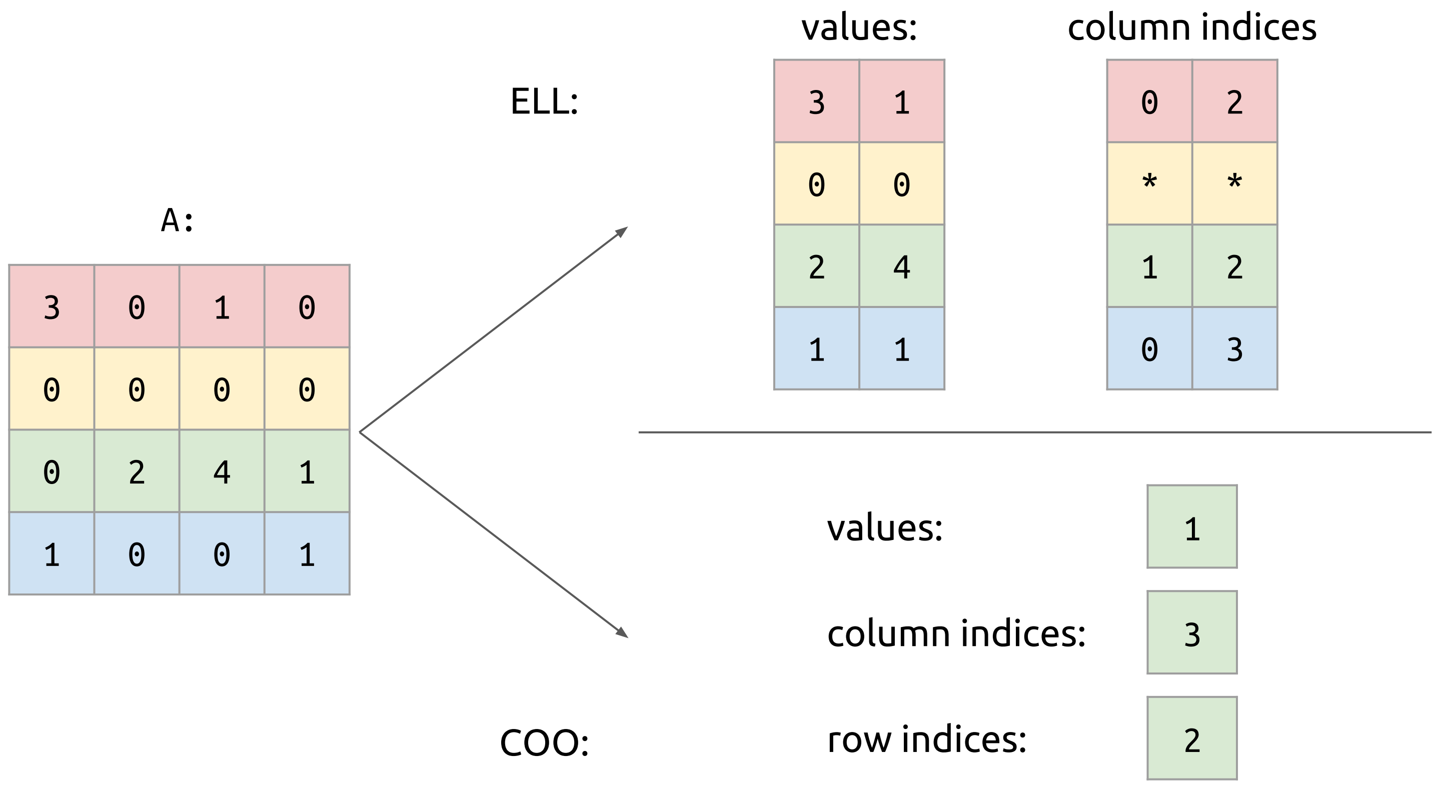
Each element will be stored in the ELL or the COO matrix, not both



```
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`

Hybrid ELL / COO SpMV

- Perform the SpMV / ELL in parallel on the GPU
  - As we've seen, the ELL format is handled efficiently by the GPU as long as the rows are roughly the same density
- Perform the SpMV / COO sequentially on the CPU
  - The irregular memory access and output interference of the SpMV / COO is better suited to the CPU with it's large cache memory

Be careful that the time spent building a more complicated representation is justified by the usage

SpMV / HYB Host Code

```
void hybridSpMV(const float * A, const int M, const int N const float * x, float * y) {

     float * d_y;
     float * d_x;
     float * y_ELL;
     SparseMatrixELL d_A_ELL;
     SparseMatrixCOO A_COO;

     // build sparse matrix representations, allocate / initialize host and device memory
     …

     // launch ELL kernel
     SpMV_ELL_kernel<<<(A.M + 127)/128,128>>>(d_A_ELL, d_x, d_y);

     // copy device result back to host
     cudaMemcpy(y_ELL, d_y, A.N * sizeof(float), cudaMemcpyDeviceToHost );

     // perform host computation
     SpMV_COO(A_COO, x, y_ELL);

}
```

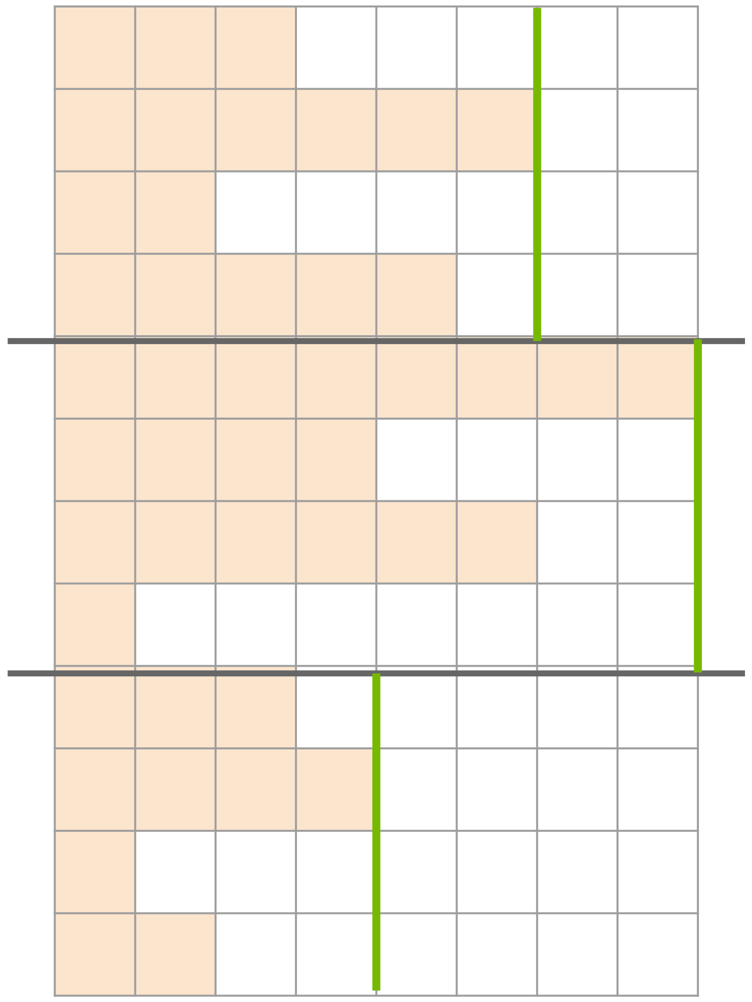
This function waits for the
kernel launch to complete

SpMV / HYB Host Code

```
void hybridSpMV(const float * A, const int M, const int N const float * x, float * y) {

     float * d_y;
     float * d_x;
     float * y_ELL;
     SparseMatrixELL d_A_ELL;
     SparseMatrixCOO A_COO;

     // build sparse matrix representations, allocate / initialize host and device memory
     …

     // launch ELL kernel
     SpMV_ELL_kernel<<<(A.M + 127)/128,128>>>(d_A_ELL, d_x, d_y);

     // perform host computation
     SpMV_COO(A_COO, x, y);

     // copy device result back to host
     cudaMemcpy(y_ELL, d_y, A.N * sizeof(float) );

     for (int i = 0; i < A.N; ++i) {
          y[i] += y_ELL[i];
     }

}
```

$\leftarrow$ This computation might happen “for free”!

CSR revisited



In the parallel SpMV / CSR kernel running on a CUDA device, the output is computed in blocks

The runtime of each block is determined by the densest row in the block

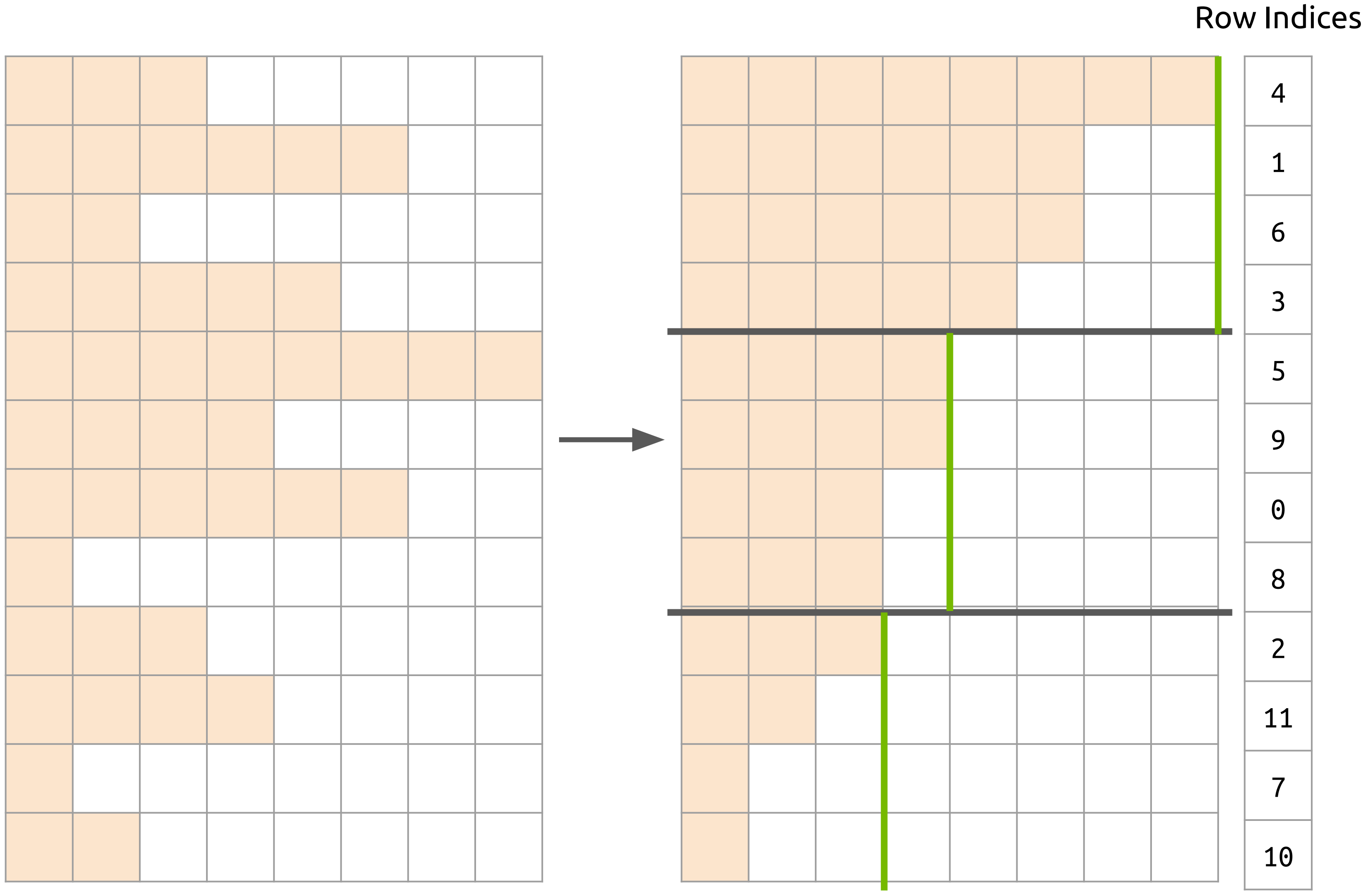
Jagged Diagonal Storage (JDS) Format

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



Jagged Diagonal Storage (JDS) Format

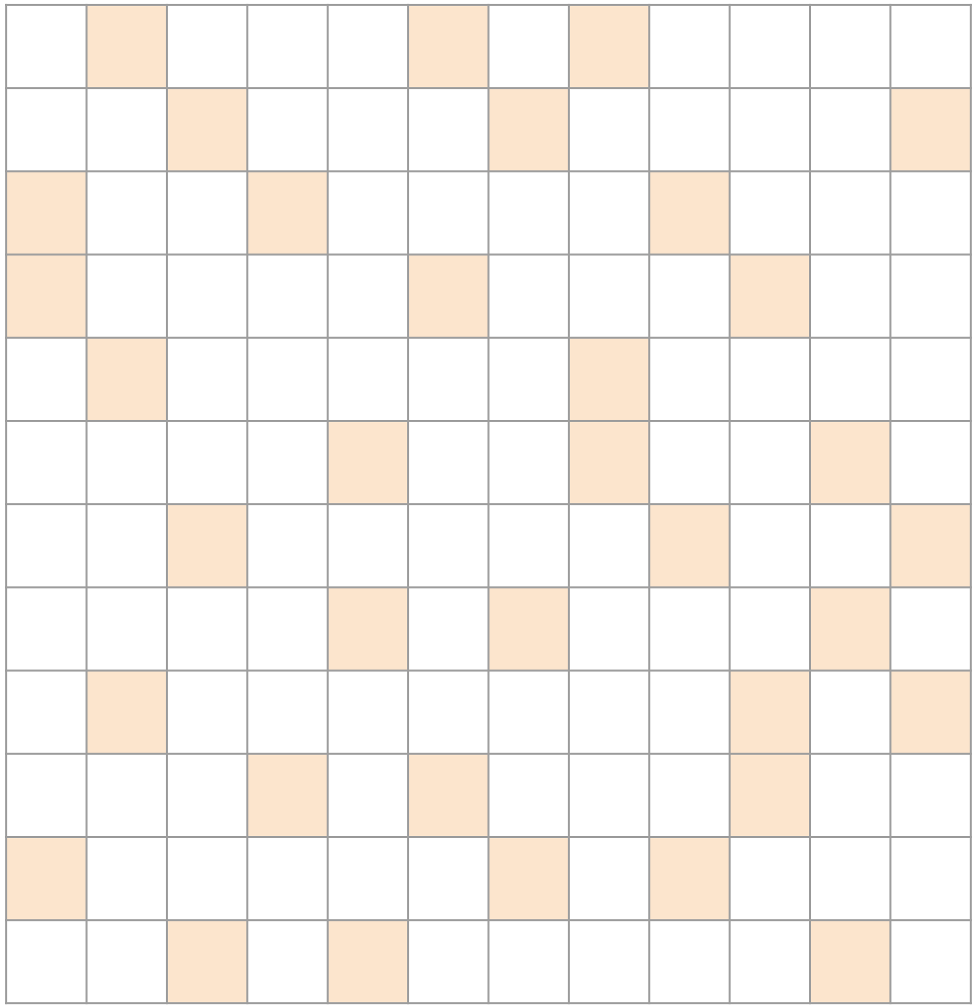
The JDS groups can be naturally mapped to CUDA blocks

While the blocks may require variable amounts of computation, threads within the same block will do similar amounts of computation

This will help maximize bandwidth

Best Format for SpMV?

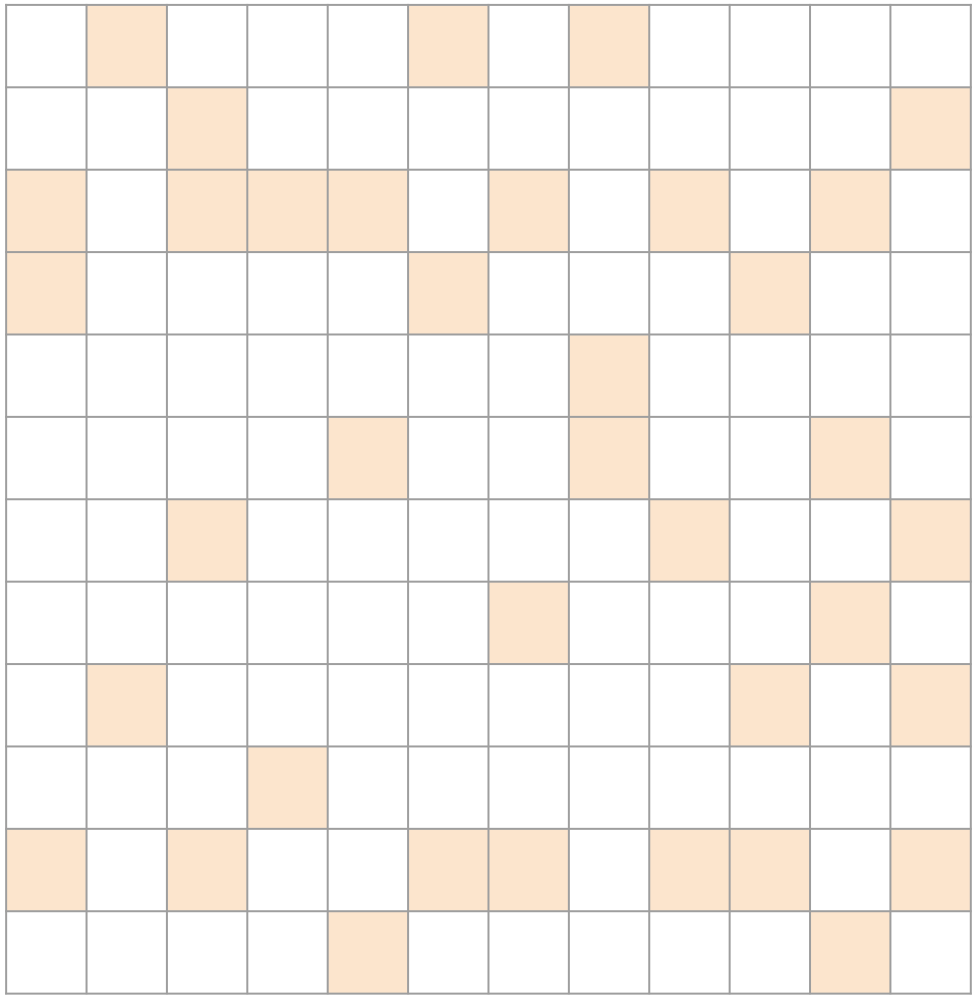
Roughly random?



Probably best with ELL

Best Format for SpMV?

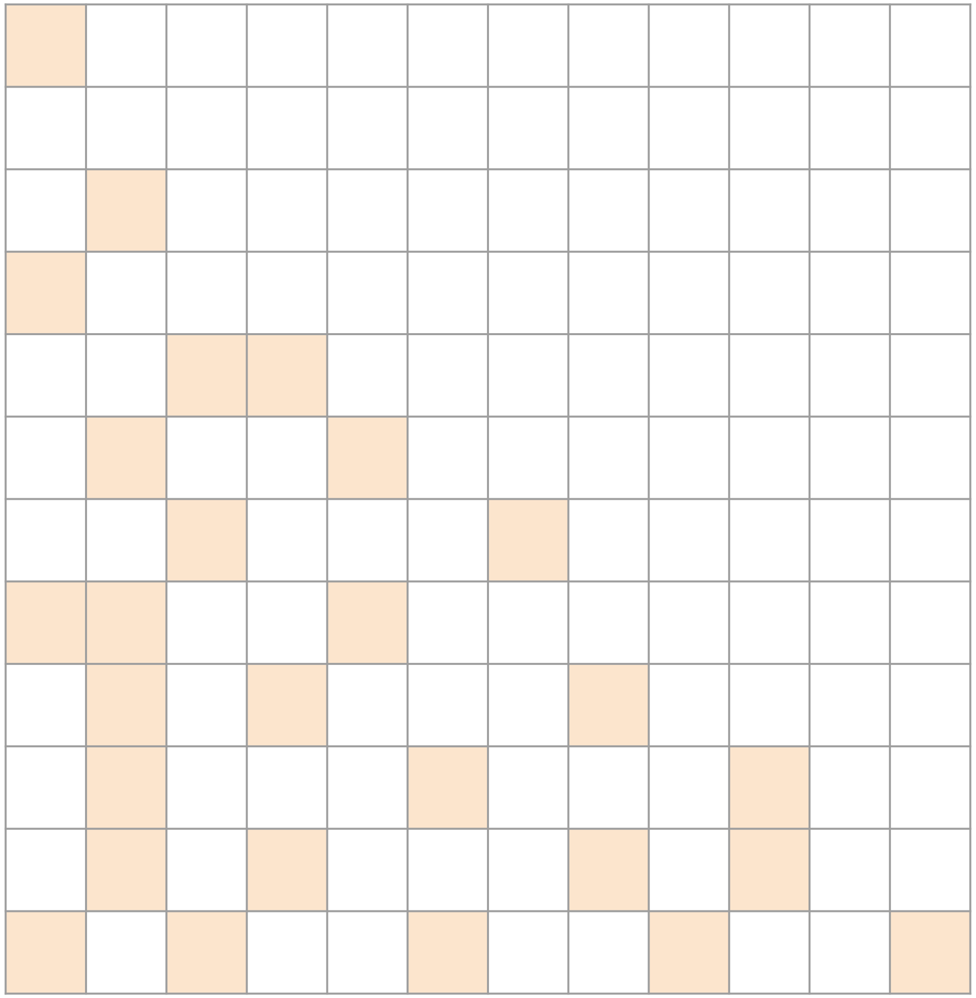
Roughly random, but with more variance in the sparsity between rows?



Probably best with a hybrid COO / ELL representation

Best Format for SpMV?

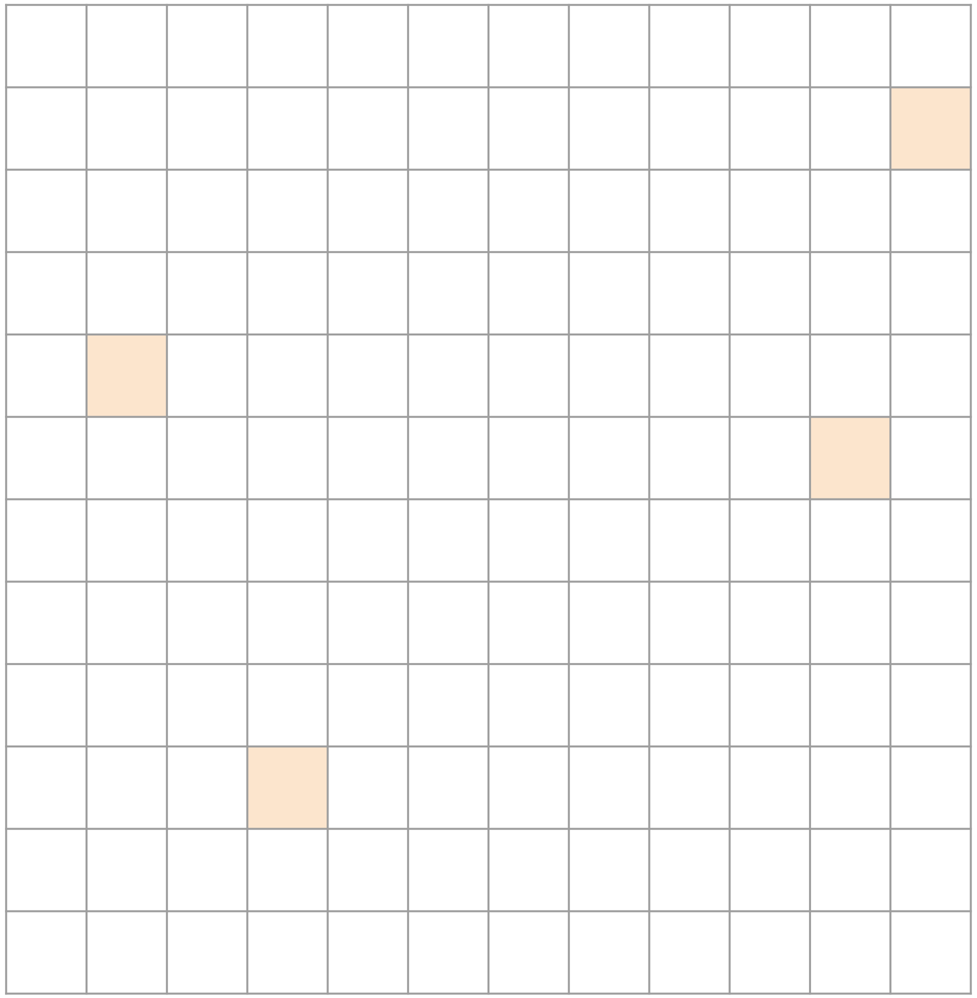
Roughly triangular?



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

Appendix: Sparse Matrix Libraries

If I’m never going to implement my own sparse matrix multiplication, who cares?

- Dealing with data-dependent performance and avoiding irregularity are common issues in massively-parallel programming
- If it’s hard for you to write sparse matrix algorithms that work efficiently in all cases, it’s hard for library implementers as well!
- Knowing the trade-offs can help you make better use of sparse matrix libraries

cuSPARSE

https://developer.nvidia.com/cusparse

cuSPARSE is an nVidia library implemented a set of basic linear algebra subroutines (BLAS)

cuSPARSE

Key Features

- Supports dense, COO, CSR, CSC, ELL/HYB and Blocked CSR sparse matrix formats
- Level 1 routines for sparse vector x dense vector operations
- Level 2 routines for sparse matrix x dense vector operations
- Level 3 routines for sparse matrix x multiple dense vectors (tall matrix)
- Routines for sparse matrix by sparse matrix addition and multiplication
- Conversion routines that allow conversion between different matrix formats
- Sparse Triangular Solve
- Tri-diagonal solver
- Incomplete factorization preconditioners ilu0 and ic0

cuSPARSE

cuSPARSE offers a fairly low-level API

For the most part, operations are performed on raw data buffers:

```
cusparseStatus_t
cusparseScsrmv(cusparseHandle_t handle, cusparseOperation_t transA, int m, int n, int nnz,
               const float *alpha, const cusparseMatDescr_t descrA, const float *csrValA,
               const int *csrRowPtrA, const int *csrColIndA, const float *x,
               const float *beta, float *y);
```

Full documentation at: http://docs.nvidia.com/cuda/cusparse

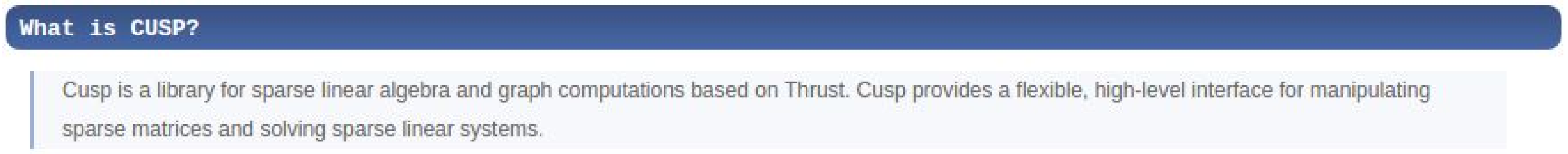
cuSPARSE

```
// create the cuSPARSE handle
cusparseCreate(&handle);

// Allocate device memory for vectors and the dense form of matrix A
...

// Construct a descriptor of the matrix A
cusparseCreateMatDescr(&descr);
cusparseSetMatType(descr, CUSPARSE_MATRIX_TYPE_GENERAL);
cusparseSetMatIndexBase(descr, CUSPARSE_INDEX_BASE_ZERO);

// Transfer the input vectors and dense matrix A to the device
...

// Compute the number of non-zero elements in A
cusparseSnnz(handle, CUSPARSE_DIRECTION_ROW, M, N, descr, dA, M, dNnzPerRow, &totalNnz);

// Allocate device memory to store the sparse CSR representation of A
...

// Convert A from a dense format to a CSR format, using the GPU
cusparseSdense2csr(handle, M, N, descr, dA, M, dNnzPerRow, dCsrValA, dCsrRowPtrA, dCsrColIndA);

// Perform matrix-vector multiplication with the CSR-format matrix A
cusparseScsrmv(handle, CUSPARSE_OPERATION_NON_TRANSPOSE, M, N, totalNnz, &alpha, descr, dCsrValA,
               dCsrRowPtrA, dCsrColIndA, dX, &beta, dY);

// Copy the result back ot the host
cudaMemcpy(Y, dY, sizeof(float) * M, cudaMemcpyDeviceToHost);
```

Cheng et al.

CUSP

https://cusplibrary.github.io/


What is CUSP?

Cusp is a library for sparse linear algebra and graph computations based on Thrust. Cusp provides a flexible, high-level interface for manipulating sparse matrices and solving sparse linear systems.

CUSP is an open source project

It is focused on solving linear systems, providing a number of conjugate-gradient solvers

CUSP

Supported matrix types:

- COO
- CSR
- DIA
- ELL
- HYB

CUSP

```
#include <cusp/csr_matrix.h>
#include <cusp/hyb_matrix.h>

int main() {

     // Allocate storage for a 5 by 8 sparse matrix in CSR format with 12
     // nonzero entries on the host
     cusp::csr_matrix<int,float,cusp::host_memory> csr_host(5,8,12);

     // Transfer the matrix to the device
     cusp::csr_matrix<int,float,cusp::device_memory> csr_device(csr_host);

     // Convert the matrix to HYB format on the device
     cusp::hyb_matrix<int,float,cusp::device_memory> csr_device(csr_device);

}
```

[https://cusplibrary.github.io/md_quickstart.html]

CUSP

```
#include <cusp/csr_matrix.h>
#include <cusp/hyb_matrix.h>

int main() {

     // Allocate storage for a 5 by 8 sparse matrix in CSR format with 12
     // nonzero entries on the host
     cusp::csr_matrix<int,float,cusp::host_memory> csr_host(5,8,12);

     // Transfer the matrix to the device
     cusp::csr_matrix<int,float,cusp::device_memory> csr_device(csr_host);

     // Convert the matrix to HYB format on the device
     cusp::hyb_matrix<int,float,cusp::device_memory> csr_device(csr_device);

}
```

[https://cusplibrary.github.io/md_quickstart.html]

CUSP

```
// include the csr_matrix header file
#include <cusp/csr_matrix.h>
#include <cusp/print.h>
int main() {
      // allocate storage for (4,3) matrix with 4 nonzeros
      cusp::csr_matrix<int,float,cusp::host_memory> A(4,3,6);
      // initialize matrix entries on host
      A.row_offsets[0] = 0;  // first offset is always zero
      A.row_offsets[1] = 2;
      A.row_offsets[2] = 2;
      A.row_offsets[3] = 3;
      A.row_offsets[4] = 6; // last offset is always num_entries
      A.column_indices[0] = 0; A.values[0] = 10;
      A.column_indices[1] = 2; A.values[1] = 20;
      A.column_indices[2] = 2; A.values[2] = 30;
      A.column_indices[3] = 0; A.values[3] = 40;
      A.column_indices[4] = 1; A.values[4] = 50;
      A.column_indices[5] = 2; A.values[5] = 60;
      // A now represents the following matrix
      //    [10  0 20]
      //    [ 0  0  0]
      //    [ 0  0 30]
      //    [40 50 60]
      // copy to the device
      cusp::csr_matrix<int,float,cusp::device_memory> B(A);
      cusp::print(B);
}
```

[https://cusplibrary.github.io/classcusp_1_1csr__matrix.html]

CUSP

Provides a number of matrix operations:

- Matrix addition
- Transposition
- SpMV
- Matrix-matrix multiplication
- Generic element-wise operations
- etc.

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).