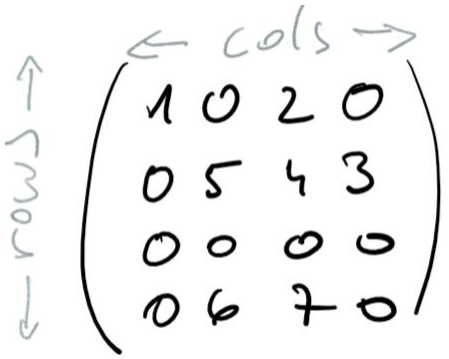
Patterns: matrix representations

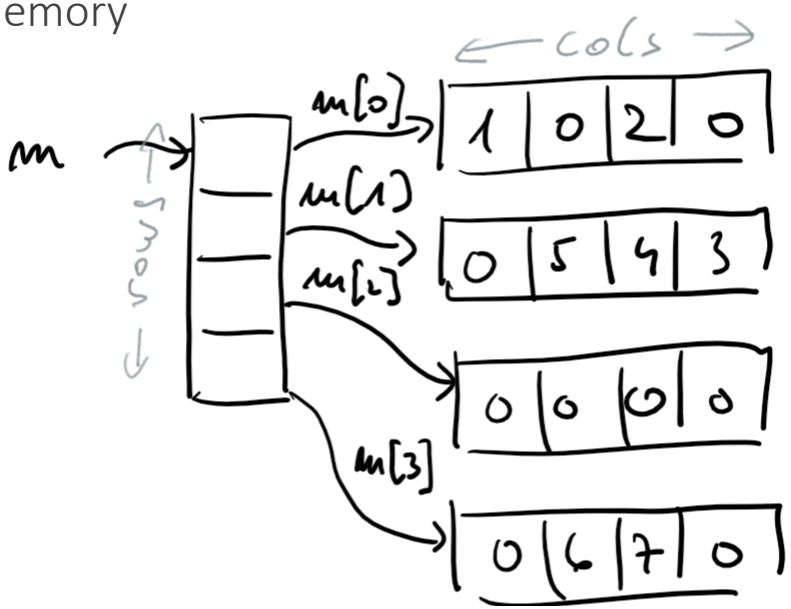
UROŠ LOTRIČ

Dense matrices

Memory allocation

- Number of elements
  - rows x cols
- Common approach
  - Allocate array of pointers to the beginning of rows
  - Allocate array of row elements separately for each row
  - There is no guarantee that rows will be adjacent to each other in memory
  - Access to elements: m[row][col]



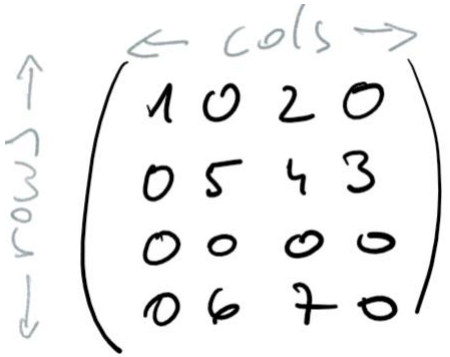


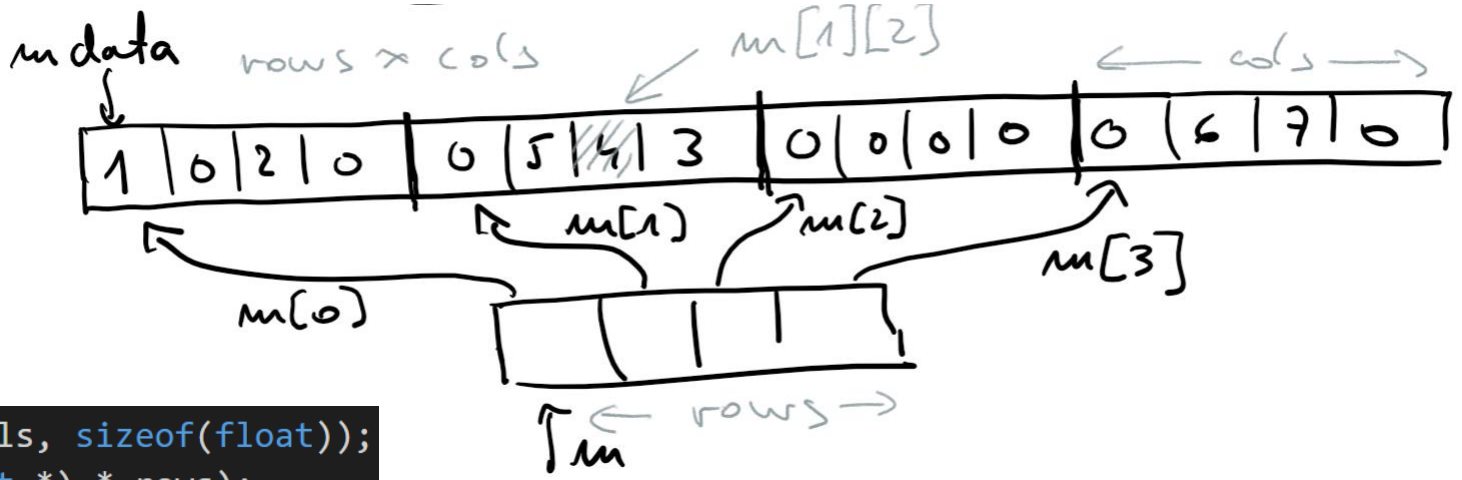

```
float** m = (float **)malloc(sizeof(float *) * rows);
for (int i = 0; i < rows; i++)
    m[i] = (float *)malloc(cols, sizeof(float));
```

Dense matrices

Memory allocation

- Continuous memory approach
  - Allocate array of matrix elements
  - If needed, allocate array of pointers to beginning of rows
  - Whole matrix data is stored in continuous memory space



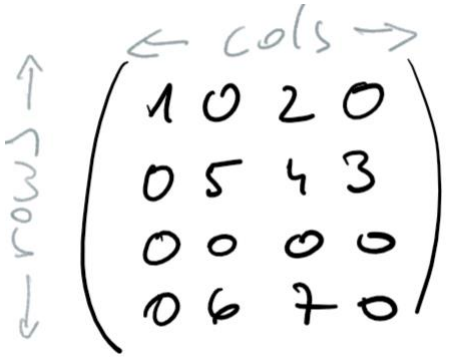


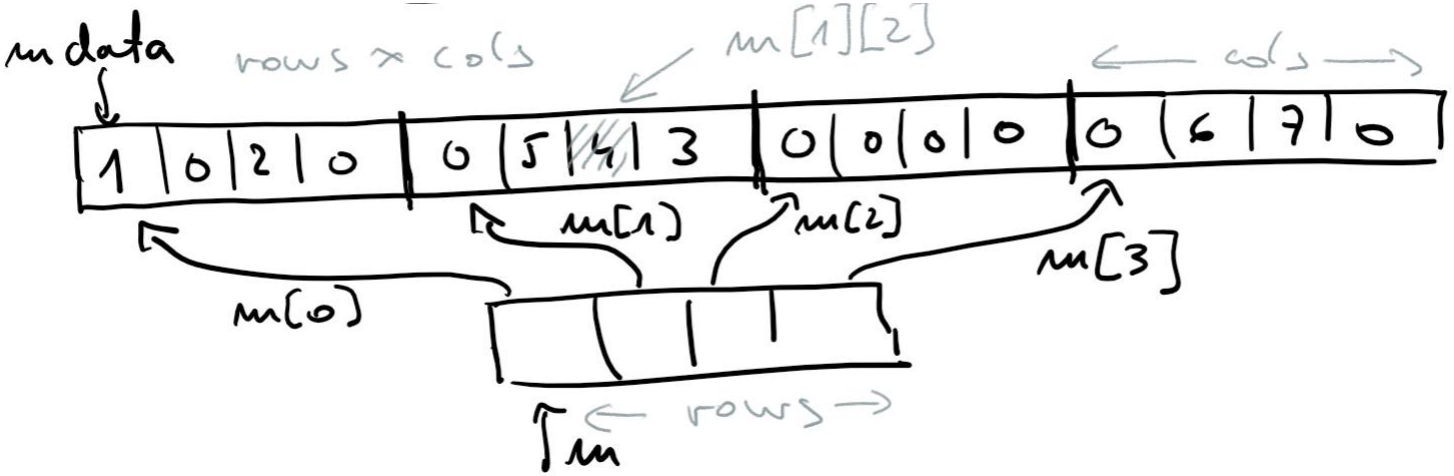

```
float* mdata = (float *)malloc(rows * cols, sizeof(float));
float** m = (float **)malloc(sizeof(float *) * rows);
for (int i = 0; i < rows; i++)
    m[i] = &mdata[i * cols];
```

Dense matrices

Data organization

- Row-major
  - m[row][col] = mdata[row*cols + col]

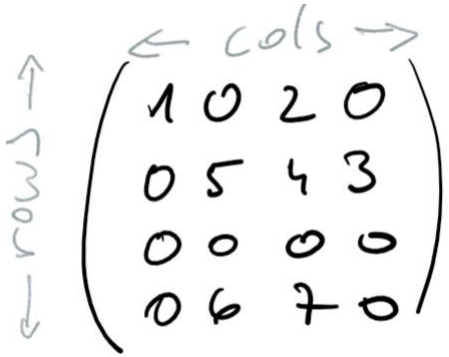


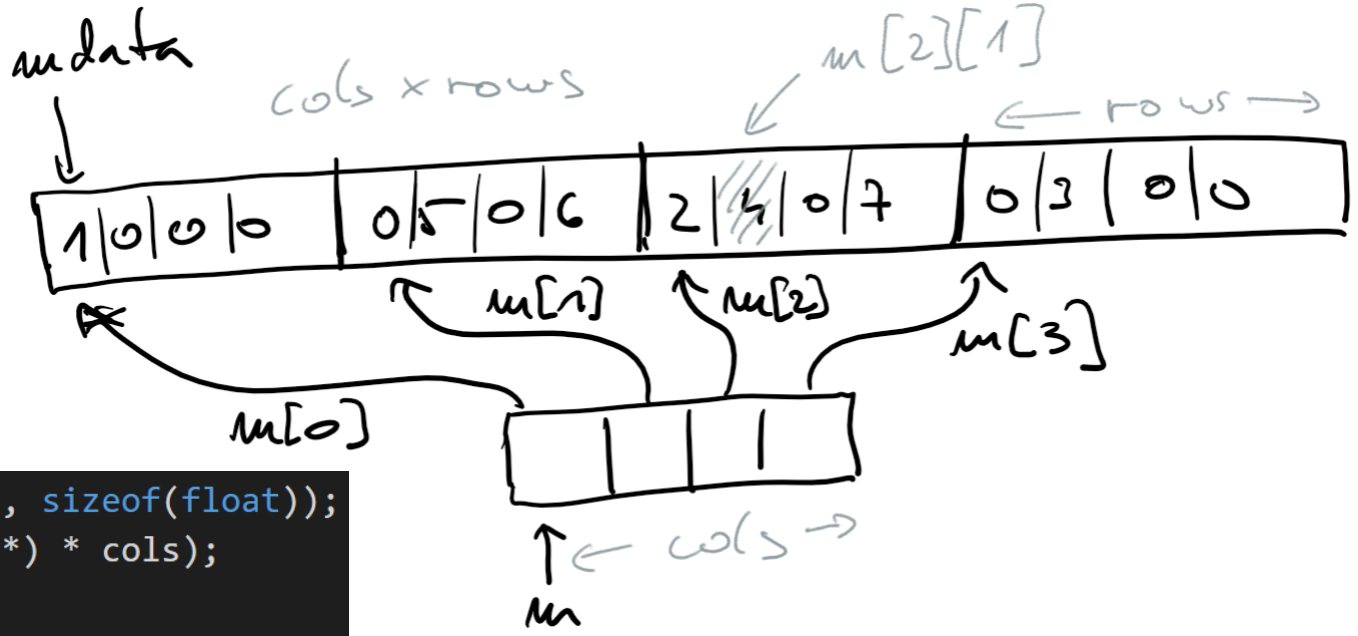


Dense matrices

Data organization

- Column-major
  - m[col][row] = mdata[col*rows + row]






```
float* mdata = (float *)malloc(rows * cols, sizeof(float));
float** m = (float **)malloc(sizeof(float *) * cols);
for (int i = 0; i < cols; i++)
    m[i] = &mdata[i * rows];
```

Matrix-vector multiplication

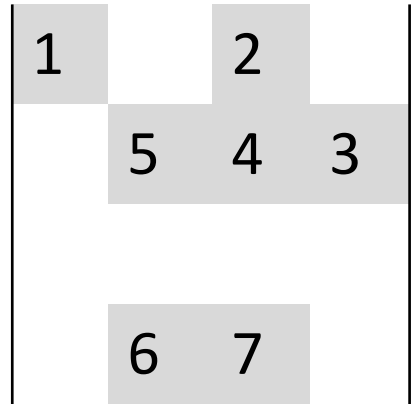
Common in science

- Neural network procession on fully connected layers
- Iterative solution of large system of equations
- Conjugate gradient solver
  - $\mathbf{Ax} + \mathbf{b} = 0$
  - The most complex part is matrix-vector multiplication
  - Matrix does not change, can be stored on device

Matrix-vector multiplication

$$\mathbf{v_{out}} = \mathbf{M}\mathbf{v_{in}}$$

$$\begin{vmatrix} 1 & 0 & 2 & 0 \\ 0 & 5 & 4 & 3 \\ 0 & 0 & 0 & 0 \\ 0 & 6 & 7 & 0 \end{vmatrix} \times \begin{vmatrix} 4 \\ 3 \\ 2 \\ 1 \end{vmatrix} = \begin{vmatrix} 8 \\ 26 \\ 0 \\ 32 \end{vmatrix}$$



Standard sequential approach

- dot-product of matrix rows and vector

```
for (int i = 0; i < num_rows; i++)
{
    v_out[i] = 0;
    for (int j = 0; j < num_cols; j++)
        v_out[i] += m[i][j] * v_in[j];
}
```

Matrix-vector multiplication

Parallelisation: each row can be processed in parallel

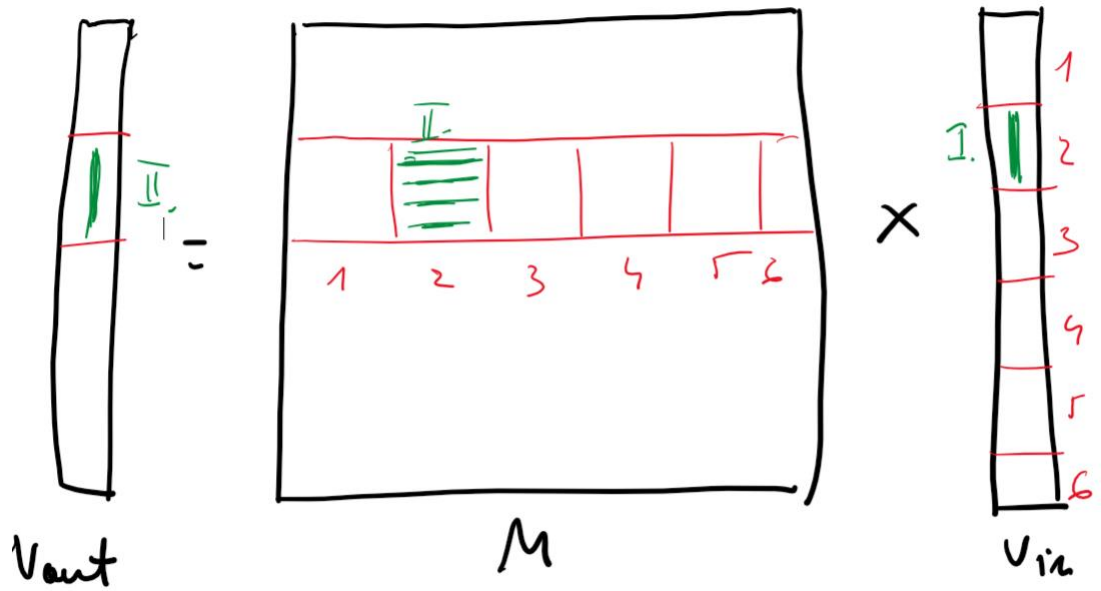
- Good approach when not many workers
  - favourable on multi-cores, each core sequentially processes more rows
- GPU, simple (MV-0.cl)
  - Same logic as above
  - Works well for matrices with many rows
  - Non-coalesced memory access

```
int i = get_global_id(0);

if(i < rows)
{
    float sum = 0.0;
    for (int j = 0; j < cols; j++)
        sum += mdata[i * cols + j] * vin[j];
    vout[i] = sum;
}
```

Matrix-vector multiplication

- GPU, tiled approach (MV-1.cl)
  - Each worker computes one element of output vector
  - Coalesced reading of input vector (phase I.)
    - All workers in a workgroup can simultaneously read input vector from global to local memory
  - Each worker calculates dot product on a tile (phase II.)
    - Matrix elements are read from global memory
    - Each worker reads its own matrix row which leads to non-coalesced access to global memory
  - When a tile is completed, process is repeated on the next



- GPU, tiled approach, column-major (MV-2.cl)
  - Adjacent workers read matrix data from adjacent memory locations

Matrix-vector multiplication

Tiled approach

- Can be efficiently extended to matrix-matrix multiplication

Comparison of OpenCL kernels

Approach	Time [s]
Sequential	1,3
MV-0.cl	0,97
MV-1.cl	0,81
MV-2.cl	0,25

Dense matrix representation

- Storage is limiting factor
- Nvidia CUDA K20
  - 12 GB of global memory
  - can store one square matrix of size 54.7k x 54.7k elements

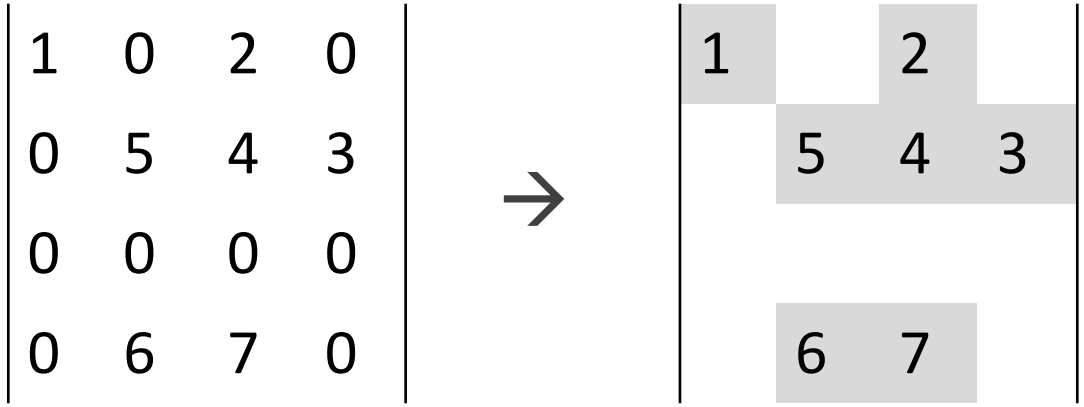
Sparse matrices

Are matrices with majority of elements zero

Usage: science, engineering, financial modelling

Many real-world applications work on sparse matrices

- Only non-zero elements are important



Sparse matrices

Compaction of representation

- Storing in a format which avoids storing zeros
- Introduces irregularity

Irregularity is a drawback in parallel implementations

- Underutilization of memory bandwidth
- Control flow divergence
- Load imbalance
- Illustrates data-dependant performance common to real-world applications

Storage formats

- Quest for good balance between compaction and regularization
  - Good compaction, high level of irregularity
  - Modest compaction, more regular representation

Coordinate representation (COO)

Each non-zero element is represented with

- data,
- row, and
- column information

Arbitrary order of data

- If we look at any single element we can place it to a matrix
- Format not appropriate for parallelization

Also known as “matrix market” format

Sequential implementation

```
for (int i = 0; i < num_nonzeros; i++)
    v_out[row[i]] += data[i] * v_in[col[i]];
```

data[7]	1	2	5	4	3	6	7
row[7]	0	0	1	1	1	3	3
col[7]	0	2	1	2	3	1	2

1		2	
	5	4	3
	6	7	

Compressed sparse row (CSR)

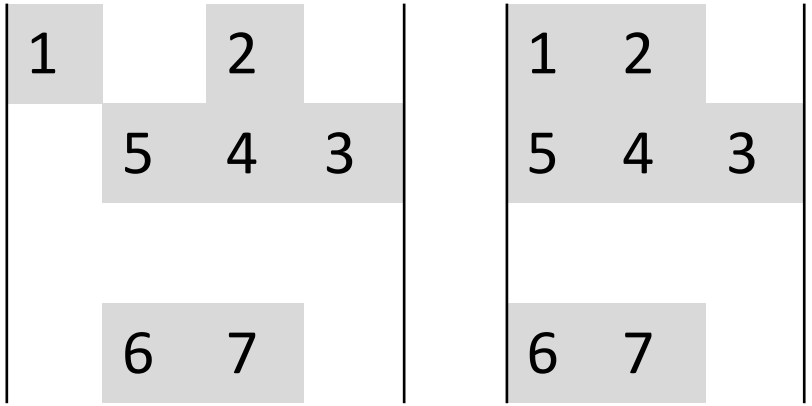
Elements are ordered row-wise

- Data and col struct remain equal to COO
- row_ptr replaces row
  - has row+1 element
  - row_ptr[r] points to the element in data where row r starts
  - last element points to the end of data
    - useful in implementations
    - number of elements in a row = row_ptr[r+1] – row_ptr[r]
  - row_ptr[r] of non-existing row points to a data element where it would start

data[7]	1	2	5	4	3	6	7
row_ptr[5]	0	2	5	5	7		
col[7]	0	2	1	2	3	1	2

Sequential implementation

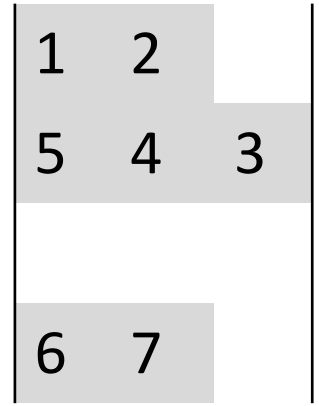
```
for (int i = 0; i < num_rows; i++)
    for (int j = row_ptr[i]; j < row_ptr[i + 1]; j++)
        v_out[i] += data[j] * v_in[col[j]];
```



Compressed sparse row (CSR)

Parallel implementation

- Millions of rows in real applications
- Each worker processes one row and has enough work
  - Inner loop is performed by each worker
- Non-coalesced memory access
  - Figure on the right
  - Worker r works on row r
  - In first iteration simultaneous access to data[0], data[2], -, data[5]
  - Accesses of adjacent workers not to adjacent locations
  - Inefficient use of bandwidth
- Control-flow divergence in warps
  - Number of worker iterations depends on number of elements in a row
  - Random distribution of non-zero elements → control-flow divergence in most warps



ELLPACK format (ELL)

Package for solving elliptic boundary value problems

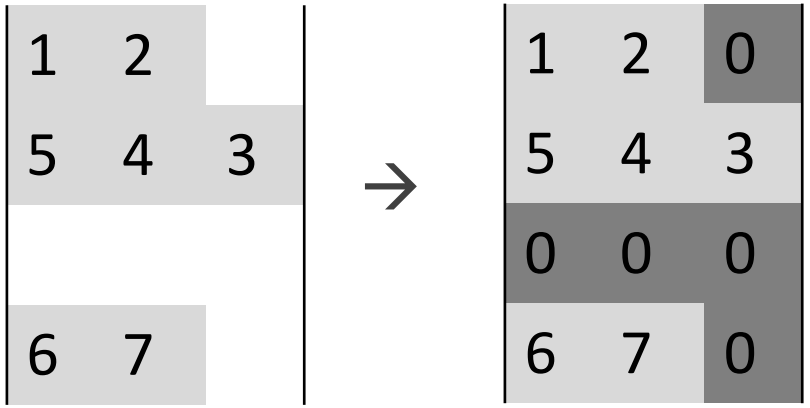
Data-padding and transposition to tackle

- non-coalesced memory access
- control-flow divergence

Padding of shorter rows with zeros

- All rows have the same number of elements
- Example
  - row 0 gets 1 zero, row 2 gets 3 zeros, row 3 gets 2 zeros
- Now all workers have the same amount of work
- Structures become larger
- * indicates inserted elements (0) in col array

data[12]	1 2 0 5 4 3 0 0 0 6 7 0
col[12]	0 2 * 1 2 3 * * * 1 2 *



ELLPACK format (ELL)

Matrix transposition

- Row-major to column-major representation
- No need for row_ptr as rows are of equal size
- We need to keep number of row elements num_elementsinrows

1	2	0
5	4	3
0	0	0
6	7	0

1	5	0	6
2	4	0	7
0	3	0	0

data[12]	1 2 0 5 4 3 0 0 0 6 7 0
col[12]	0 2 * 1 2 3 * * * 1 2 *

data[12]	1 5 0 6 2 4 0 7 0 3 0 0
col[12]	0 1 * 1 2 2 * 2 * 3 * *

ELLPACK format (ELL)

Matrix transposition

- First element of row r: data[r], i-th element of row r: data[r+i*num_rows]
- Bonus: all adjacent threads are now accessing adjacent memory locations
  - Example:
    - 4 workers access at the same time underlined data
    - Underlined data is adjacent for adjacent workers

1	5	0	6
2	4	0	7
0	3	0	0

data[12]	1 5 0 6 2 4 0 7 0 3 0 0
col[12]	0 1 * 1 2 2 * 2 * 3 * *

ELLPACK format (ELL)

Sequential code

```
for (int i = 0; i < num_rows; i++)
    for (int j = 0; j < num_elementsinrow; j++)
        v_out[i] += data[j * num_rows + i] * v_in[col[j * num_rows + i]];
```

Parallel solution

- Each worker works on one row

1	5	0	6
2	4	0	7
0	3	0	0

data[12]	1	5	0	6	2	4	0	7	0	3	0	0
col[12]	0	1	*	1	2	2	*	2	*	3	*	*

Experiments

Timing: 1000 x

- write v_{in} to device, multiply on device, read v_{out} from device

Discussion

- Parallel ELL wins in most cases
- Poor results on
  - max-ecn-fwd500.mtx
  - scircuit.mtx

Code

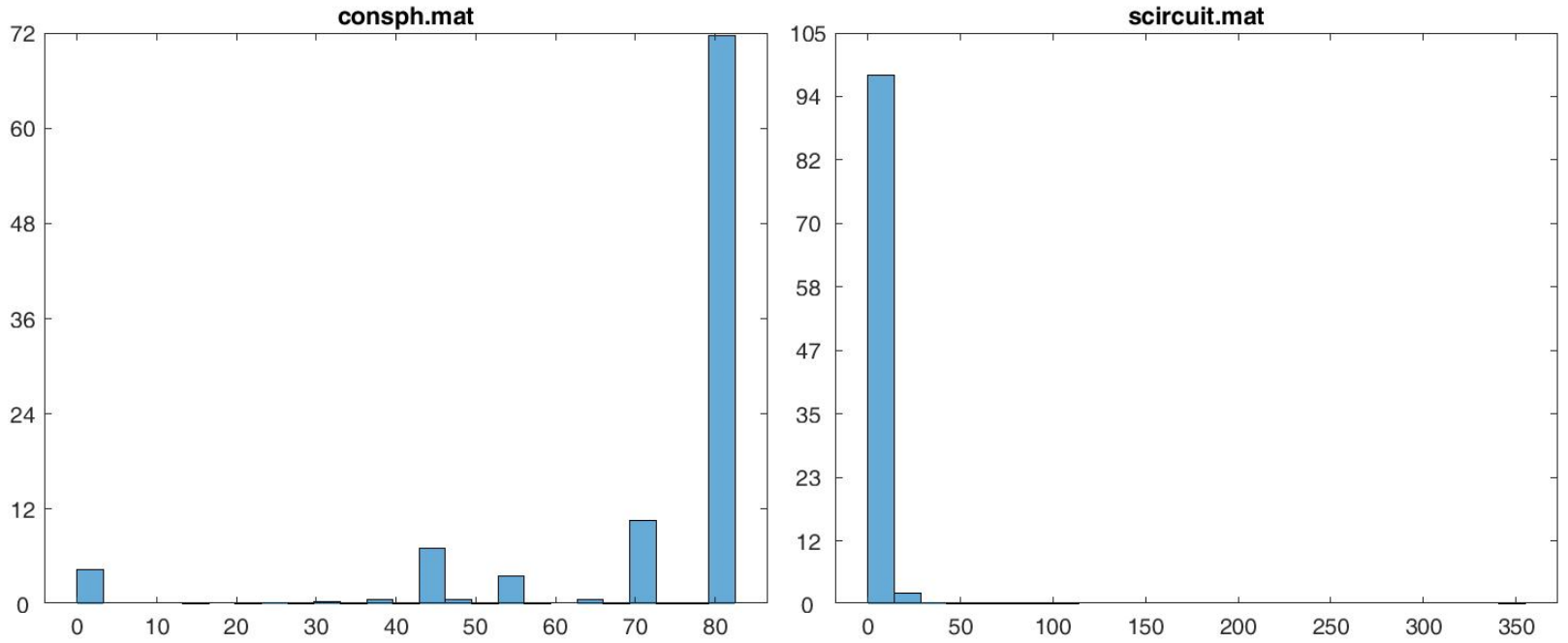
- SpMV_cl.c
- SpMV_cl.cl

data	size	non-zero	seq COO [ms]	par CSR [ms]	par ELL [ms]
cant.mtx	62k	4M	9.98	3.90	**0.49**
consph.mtx	83k	6M	15.8	5.85	**0.66**
mac-econ-fwd500.mtx	207k	1.3M	2.14	**0.86**	1.08
mc2depi.mtx	526k	2.1M	3.11	1.19	**1.08**
pdb1HYS.mtx	36k	4.3M	11.8	4.92	**0.69**
pwtk.mtx	218k	11.6M	28.3	11.7	**3.00**
scircuit.mtx	171k	59k	1.51	**0.61**	4.74
shipsec1.mtx	141k	4M	7.97	2.00	**1.01**

Experiments

Comparison of distribution of number of elements in a row

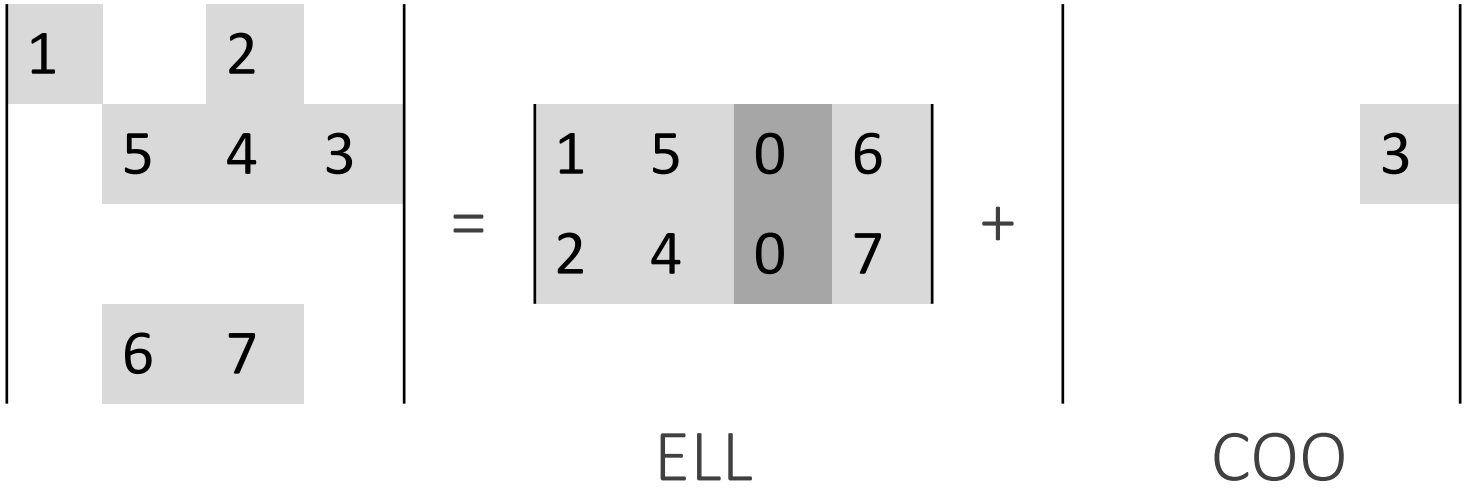
- consph.mtx
  - majority of rows has around 80 elements per row
- scircuit.mtx
  - majority of row with < 10 and few rows with 350+ elements
  - A lot of padding with zeros



Hybrid solution (HYB)

Split initial matrix to two parts

- One part has approximately the same number of rows
  - Converted to ELL
  - Runs on GPU
- The other part stores excessive elements in long rows
  - Converted to COO
  - Runs on host



Jagged diagonal storage (JDS)

Sort rows descending by number of elements

- Must keep record of initial row position

Split matrix horizontally to submatrices with similar number of elements in rows

Perform ELL sequentially on submatrices

No need for CPU computation

Better suited for sparse matrices with a wide distribution of row-lenghts