

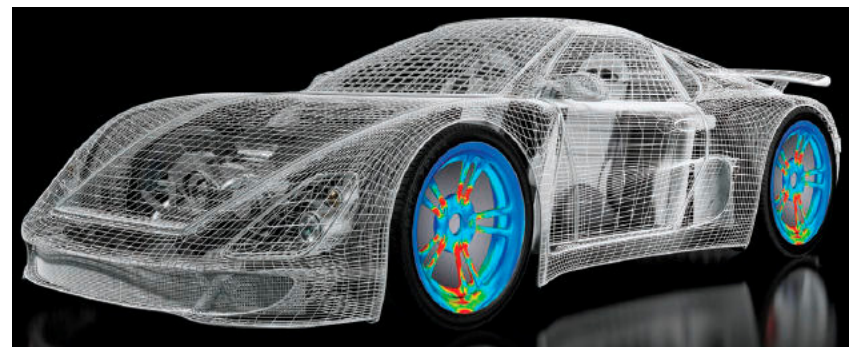
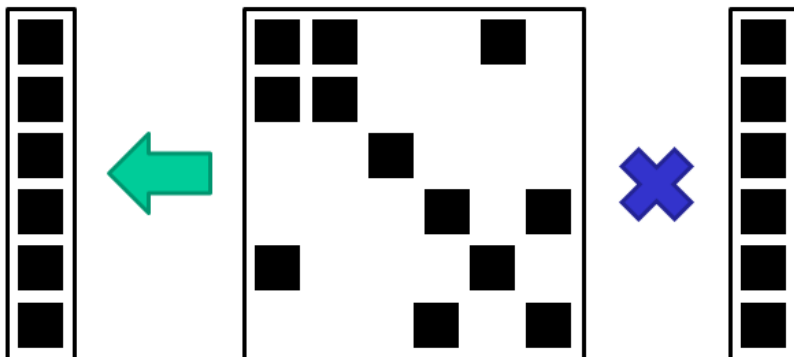


# CUDA 5 Sparse Matrix-Vector Multiplication

CS121 Parallel Computing  
Fall 2022

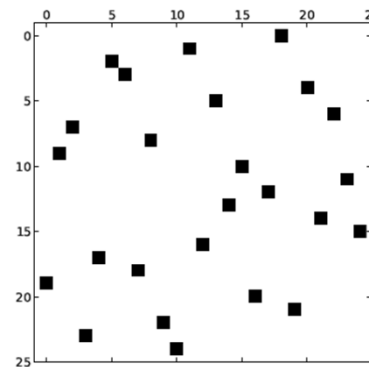
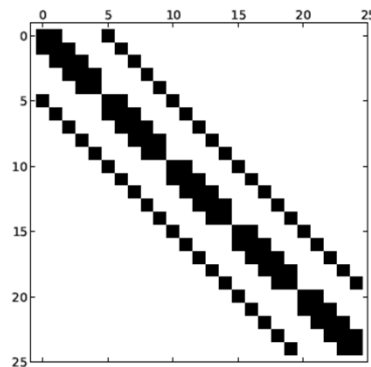
# SpMV

- Sparse matrix vector multiplication.
- Many scientific algorithms require multiplying a matrix by a vector.
  - Optimization (e.g. conjugate gradient), iterative methods (solving linear systems), eigenvalue methods (e.g. graph partitioning), simulations (e.g. finite elements), data analysis (e.g. Pagerank).
- The matrices are often sparse.
  - In an  $n \times n$  matrix, there are  $o(n^2)$  nonzero elements.
  - Ex For finite elements, matrix comes from low degree mesh.
  - Ex For Pagerank, the matrix is the web connectivity matrix.



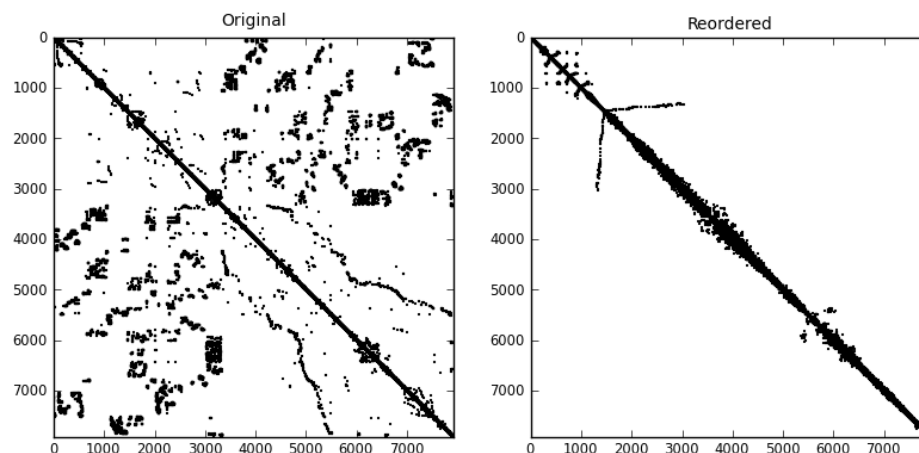
# SpMV challenges

- Compute  $b = Ax$ .  $A$  is a sparse matrix,  $x$  is a vector.
  - $b[i] = \sum_{j=1}^n A[i, j] x[j]$ , for  $i = 1, \dots, n$ .
- Computation is memory bound.
  - 2 reads for 2 computes.
  - Ex GTX 680 has 1.5 TFLOPS compute, 200 GB/s bandwidth.
- Matrices may be regular or irregular.
  - Irregular matrices cause work imbalance, uncoalesced memory accesses.
  - Ex Finite element grids are regular.
  - Ex Web matrices for Pagerank have power law degree distribution.



# SpMV techniques

- Matrix and vector both stored in global memory.
- Nothing we can do about memory boundedness.
  - Unlike matrix-matrix multiply, few values are read multiple times.
- To address irregularity of matrix accesses
  - Store only the nonzero matrix elements.
  - Different matrix storage formats improve memory coalescing.
  - Formats also improve load balancing.
  - Assign threads to work to minimize divergence.
- To regularize vector accesses, permute elements to make matrix more block diagonal and cache vector elements.
  - Expensive, but done once per matrix and can be reused.



# DIA format

DIA format:

$$A = \begin{bmatrix} 1 & 7 & 0 & 0 \\ 0 & 2 & 8 & 0 \\ 5 & 0 & 3 & 9 \\ 0 & 6 & 0 & 4 \end{bmatrix} \quad \text{data} = \begin{bmatrix} * & 1 & 7 \\ * & 2 & 8 \\ 5 & 3 & 9 \\ 6 & 4 & * \end{bmatrix} \quad \text{offsets} = [-2 \quad 0 \quad 1]$$

- Look at values along the diagonal of the matrix.
- Data stored in column major form.
  - Column  $i$  contains values on  $i$ 'th nonzero diagonal.
    - \* indicates no value at location.
  - `offsets[i]` stores offset of  $i$ 'th diagonal from main diagonal.
    - $-i$  means  $i$  diagonals to left,  $+i$  means  $i$  diagonals to right.
- Only effective for matrices where nonzeros lie on a few diagonals.
  - Stencils, grids, finite element meshes.

# ELL format

ELL format:

$$A = \begin{bmatrix} 1 & 7 & 0 & 0 \\ 0 & 2 & 8 & 0 \\ 5 & 0 & 3 & 9 \\ 0 & 6 & 0 & 4 \end{bmatrix} \quad \text{data} = \begin{bmatrix} 1 & 7 & * \\ 2 & 8 & * \\ 5 & 3 & 9 \\ 6 & 4 & * \end{bmatrix} \quad \text{indices} = \begin{bmatrix} 0 & 1 & * \\ 1 & 2 & * \\ 0 & 2 & 3 \\ 1 & 3 & * \end{bmatrix}$$

- data and indices have one row for each row of A.
- data[i,j] is value of j'th nonzero in i'th row of A.
  - If no j'th nonzero, store padding value \*.
- indices[i,j] is column of j'th nonzero in i'th row of A.
- Number of columns in data and indices equals maximum number of nonzeros in any row of A.
- Store data and indices in column major format.
- Efficient only for matrices with roughly same number of columns per row.

# COO format

COO format:

$$A = \begin{bmatrix} 1 & 7 & 0 & 0 \\ 0 & 2 & 8 & 0 \\ 5 & 0 & 3 & 9 \\ 0 & 6 & 0 & 4 \end{bmatrix}$$

$$\begin{aligned} \text{row} &= [0 & 0 & 1 & 1 & 2 & 2 & 2 & 3 & 3] \\ \text{indices} &= [0 & 1 & 1 & 2 & 0 & 2 & 3 & 1 & 3] \\ \text{data} &= [1 & 7 & 2 & 8 & 5 & 3 & 9 & 6 & 4] \end{aligned}$$

- Store coordinates of all nonzeros in  $A$  in row major form.
  - $i$ 'th element in  $\text{row}[i]$ , column  $\text{indices}[i]$ , has value  $\text{data}[i]$ .
- Most general purpose format. Matrix can be any shape.
- Somewhat inefficient, as it repeatedly stores row index of elements in same row.
  - Uses more global memory to store.
  - Causes more global memory traffic when reading matrix.

# CSR format

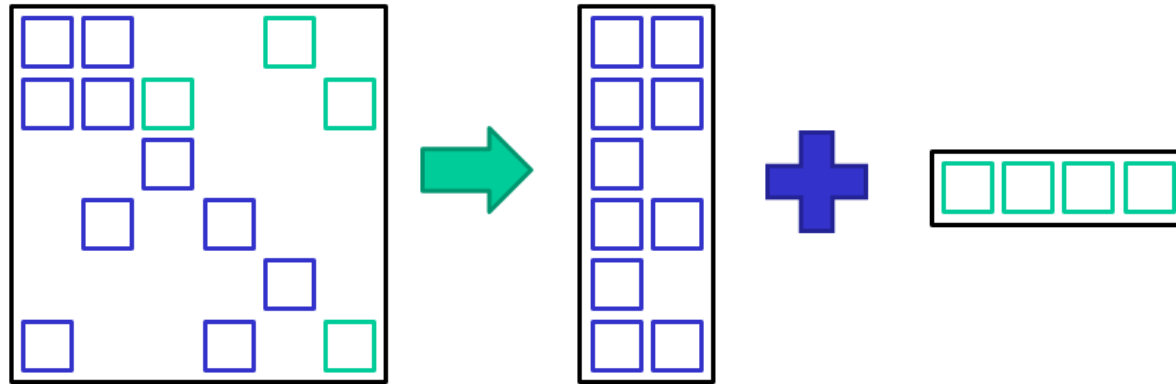
CSR format:

$$A = \begin{bmatrix} 1 & 7 & 0 & 0 \\ 0 & 2 & 8 & 0 \\ 5 & 0 & 3 & 9 \\ 0 & 6 & 0 & 4 \end{bmatrix}$$
$$\begin{aligned} \text{ptr} &= [0 \quad 2 \quad 4 \quad 7 \quad 9] \\ \text{indices} &= [0 \quad 1 \quad 1 \quad 2 \quad 0 \quad 2 \quad 3 \quad 1 \quad 3] \\ \text{data} &= [1 \quad 7 \quad 2 \quad 8 \quad 5 \quad 3 \quad 9 \quad 6 \quad 4] \end{aligned}$$

- Compressed sparse row.
- Like COO, but don't repeat row indices.
- ptr has n elements, one for each row.
- ptr[i] is the index in indices where i'th row starts.
  - Elements in i'th row have indices between ptr[i] and ptr[i+1]-1.
  - Column of j'th element in i'th row is indices[ptr[i]+j].
  - Value of j'th element in i'th row is data[ptr[i]+j].
- Flexible, efficient, widely used format.

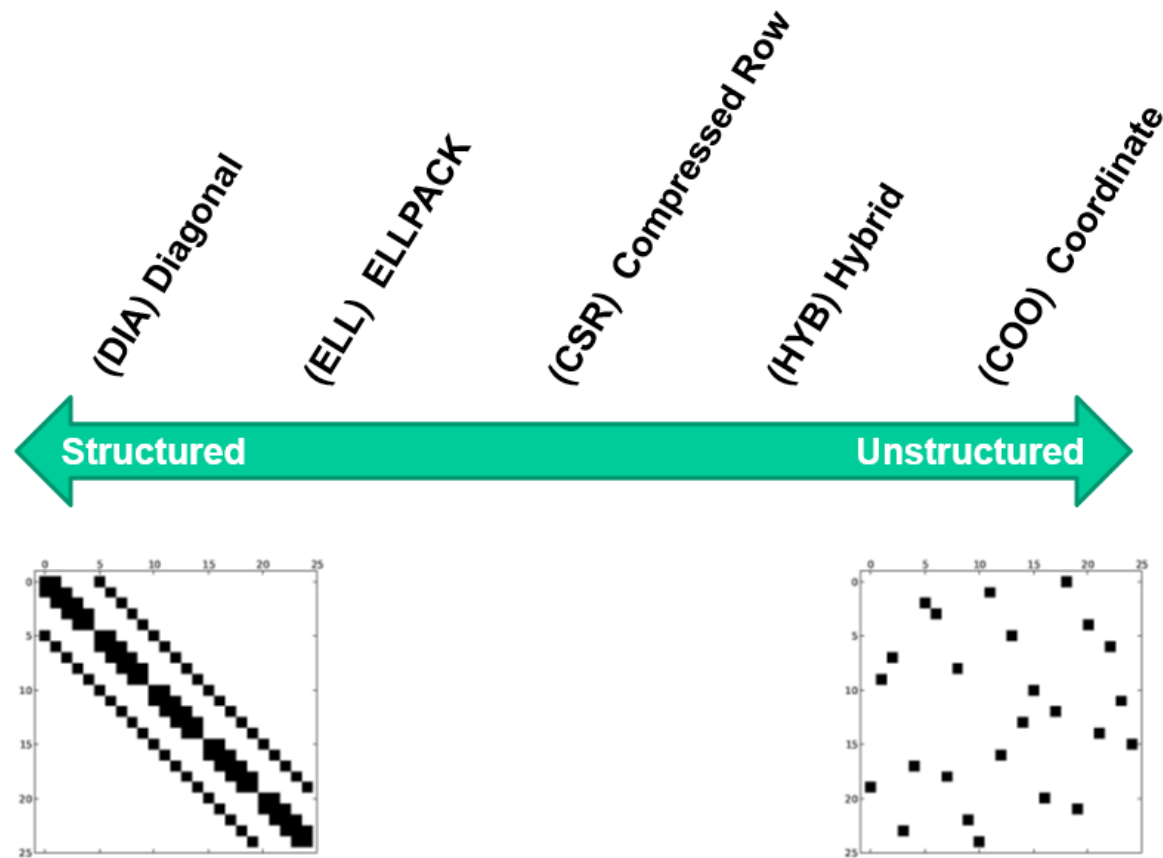


# Hybrid format



- A combination of ELL and COO.
- Assumes most rows have similar length  $L$ .
- Break  $A$  into two matrices, one containing first  $L$  nonzeros of each row of  $A$ , other containing remaining elements.
  - Store first matrix using ELL, other using COO.
- Another flexible, efficient format.

# Which kernel to use?



- Right kernel depends on structure of matrix.

# ELL kernel

$$A = \begin{bmatrix} 1 & 7 & 0 & 0 \\ 0 & 2 & 8 & 0 \\ 5 & 0 & 3 & 9 \\ 0 & 6 & 0 & 4 \end{bmatrix}$$

ELL format:

$$\text{data} = \begin{bmatrix} 1 & 7 & * \\ 2 & 8 & * \\ 5 & 3 & 9 \\ 6 & 4 & * \end{bmatrix}$$

$$\text{indices} = \begin{bmatrix} 0 & 1 & * \\ 1 & 2 & * \\ 0 & 2 & 3 \\ 1 & 3 & * \end{bmatrix}$$

data	[1 2 5 6 7 8 3 4 * * 9 *]
indices	[0 1 0 1 1 2 2 3 * * 3 *]
Iteration 0	[0 1 2 3]
Iteration 1	[0 1 2 3]
Iteration 2	[0 1 2 3]

- Assign i'th thread to read i'th row of data and indices.
- If all rows have similar lengths, get good load balancing.
  - Each thread takes about same number of steps to finish.
- Since data, indices stored in column major format, all memory accesses coalesced.
- Use indices to read coordinates of vector and perform dot product.

```
__global__ void
spmv_ell_kernel(const int num_rows,
                const int num_cols,
                const int num_cols_per_row,
                const int * indices,
                const float * data,
                const float * x,
                float * y)
{
    int row = blockDim.x * blockIdx.x + threadIdx.x;

    if(row < num_rows){
        float dot = 0;

        for(int n = 0; n < num_cols_per_row; n++){
            int col = indices[num_rows * n + row];
            float val = data[num_rows * n + row];

            if(val != 0)
                dot += val * x[col];
        }

        y[row] += dot;
    }
}
```

# CSR scalar kernel

ptr = [0 2 4 7 9]  
indices = [0 1 1 2 0 2 3 1 3]  
data = [1 7 2 8 5 3 9 6 4]

$$A = \begin{bmatrix} 1 & 7 & 0 & 0 \\ 0 & 2 & 8 & 0 \\ 5 & 0 & 3 & 9 \\ 0 & 6 & 0 & 4 \end{bmatrix}$$

indices [0 1 1 2 0 2 3 1 3]  
data [1 7 2 8 5 3 9 6 4]

Iteration 0 [0 1 2 3]  
Iteration 1 [ 0 1 2 3]  
Iteration 2 [ 2 ]

- Assign one thread per row.
- Not load balanced, since rows can be different lengths.
- Rarely memory coalesced, since elements of different rows likely stored far apart.
- Usually poor performance.

```
__global__ void  
spmv_csr_scalar_kernel(const int num_rows,  
                        const int * ptr,  
                        const int * indices,  
                        const float * data,  
                        const float * x,  
                        float * y)  
{  
    int row = blockDim.x * blockIdx.x + threadIdx.x;  
  
    if(row < num_rows){  
        float dot = 0;  
  
        int row_start = ptr[row];  
        int row_end = ptr[row+1];  
  
        for (int jj = row_start; jj < row_end; jj++){  
            dot += data[jj] * x[indices[jj]];  
        }  
  
        y[row] += dot;  
    }  
}
```

# CSR vector kernel

$$A = \begin{bmatrix} 1 & 7 & 0 & 0 \\ 0 & 2 & 8 & 0 \\ 5 & 0 & 3 & 9 \\ 0 & 6 & 0 & 4 \end{bmatrix}$$

CSR format:

```
ptr = [0 2 4 7 9]
indices = [0 1 1 2 0 2 3 1 3]
data = [1 7 2 8 5 3 9 6 4]
```

```
indices [0 1 1 2 0 2 3 1 3]
data    [1 7 2 8 5 3 9 6 4]
```

```
Warp 0 [0 0]
Warp 1 [    1 1]
Warp 2 [    2 2 2]
Warp 3 [    3 3]
```

- Assign one warp per row.
  - Thread  $i$  in warp reads elements  $i, i+32, i+64, \dots$
- Better memory coalescing.
- Some threads in warp idle if row length too small or not divisible by 32.
- Different warps not load balanced if rows have different lengths.
  - But inter-warp imbalance less serious than intra-warp imbalance, since SM scheduler can switch between warps.
  - This still hides memory latency as long as enough active warps.

# CSR vector kernel

```
__global__ void
spmv_csr_vector_kernel(const int num_rows,
                       const int * ptr,
                       const int * indices,
                       const float * data,
                       const float * x,
                       float * y)
{
    __shared__ float vals[];

    int thread_id = blockDim.x * blockIdx.x + threadIdx.x; // global thread index
    int warp_id = thread_id / 32; // global warp index
    int lane = thread_id & (32 - 1); // thread index within the warp

    // one warp per row
    int row = warp_id;

    if (row < num_rows){
        int row_start = ptr[row];
        int row_end = ptr[row+1];

        // compute running sum per thread
        vals[threadIdx.x] = 0;
        for(int jj = row_start + lane; jj < row_end; jj += 32)
            vals[threadIdx.x] += data[jj] * x[indices[jj]];

        // parallel reduction in shared memory
        if (lane < 16) vals[threadIdx.x] += vals[threadIdx.x + 16];
        if (lane < 8) vals[threadIdx.x] += vals[threadIdx.x + 8];
        if (lane < 4) vals[threadIdx.x] += vals[threadIdx.x + 4];
        if (lane < 2) vals[threadIdx.x] += vals[threadIdx.x + 2];
        if (lane < 1) vals[threadIdx.x] += vals[threadIdx.x + 1];

        // first thread writes the result
        if (lane == 0)
            y[row] += vals[threadIdx.x];
    }
}
```

- Thread  $i$  in warp multiplies matrix elements  $i$ ,  $i+32$ ,  $i+64$ , ... by corresponding elements in vector and sums these.
- So each warp produces 32 partial sums.
- Warp does parallel reduction on partial sums to get sum of row.

# COO kernel

COO format:

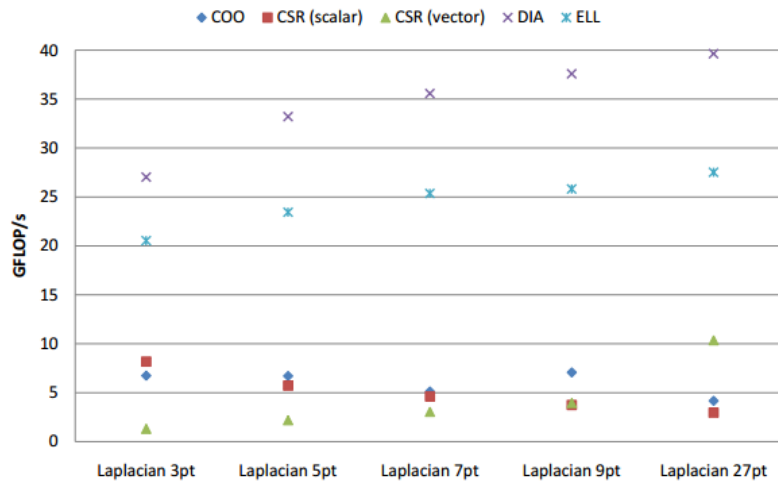
```
row = [0 0 1 1 2 2 2 3 3]
indices = [0 1 1 2 0 2 3 1 3]
data = [1 7 2 8 5 3 9 6 4]
```

```
__device__ void
segmented_reduction(const int lane, const int * rows, float * vals)
{
    // segmented reduction in shared memory
    if( lane >= 1 && rows[threadIdx.x] == rows[threadIdx.x - 1] )
        vals[threadIdx.x] += vals[threadIdx.x - 1];
    if( lane >= 2 && rows[threadIdx.x] == rows[threadIdx.x - 2] )
        vals[threadIdx.x] += vals[threadIdx.x - 2];
    if( lane >= 4 && rows[threadIdx.x] == rows[threadIdx.x - 4] )
        vals[threadIdx.x] += vals[threadIdx.x - 4];
    if( lane >= 8 && rows[threadIdx.x] == rows[threadIdx.x - 8] )
        vals[threadIdx.x] += vals[threadIdx.x - 8];
    if( lane >= 16 && rows[threadIdx.x] == rows[threadIdx.x - 16] )
        vals[threadIdx.x] += vals[threadIdx.x - 16];
}
```

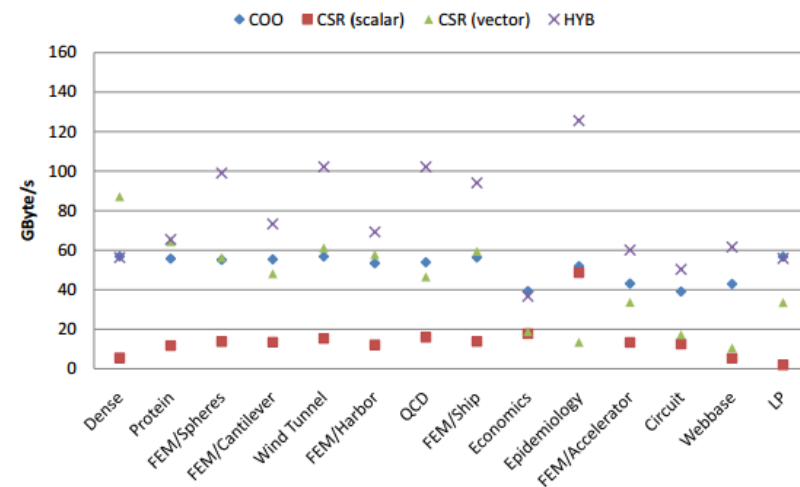
row	[0 0 1 1 2 2 2 3 3]
indices	[0 1 1 2 0 2 3 1 3]
data	[1 7 2 8 5 3 9 6 4]
Iteration 0	[0 1 2 3                   ]
Iteration 1	[                   0 1 2 3 ]
Iteration 2	[                           0]

- Assign one thread per nonzero.
- Perfect load balancing.
- Completely coalesced memory accesses.
- One warp may span several (short) rows.
  - Use parallel segmented reduction.
- Code above assumes each row spans at most one warp.
  - For general case see Bell and Garland's SC2009 paper.

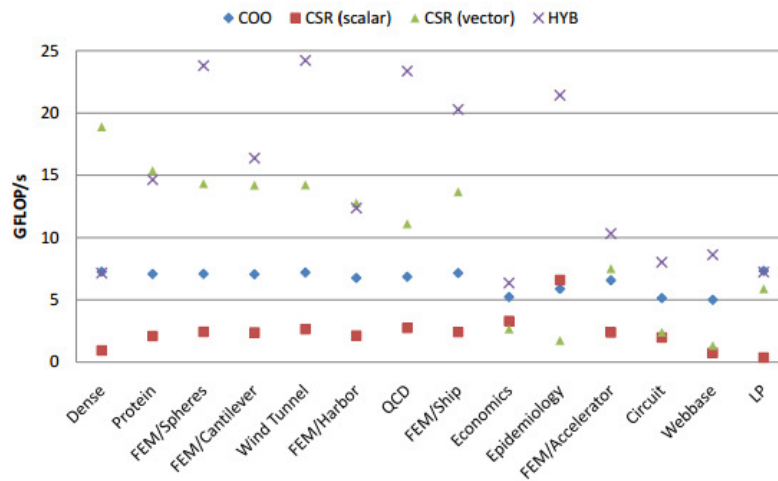
# Performance



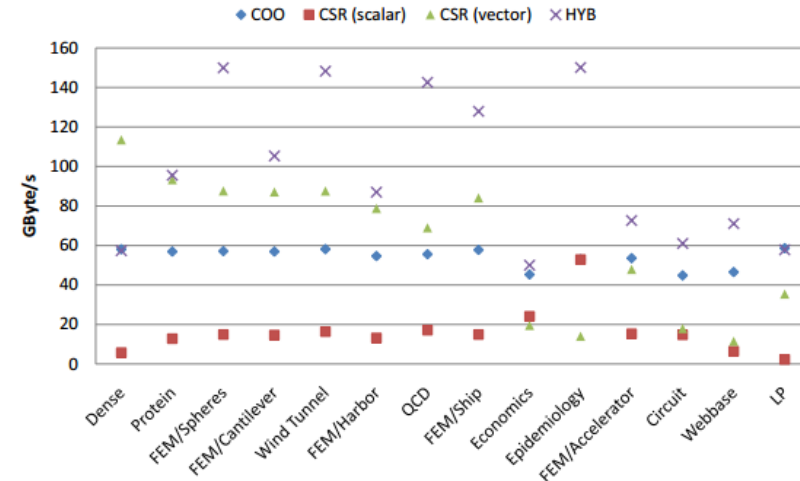
structured matrices throughput



unstructured matrix bandwidth, no cache



unstructured matrices throughput



unstructured matrix bandwidth, with cache