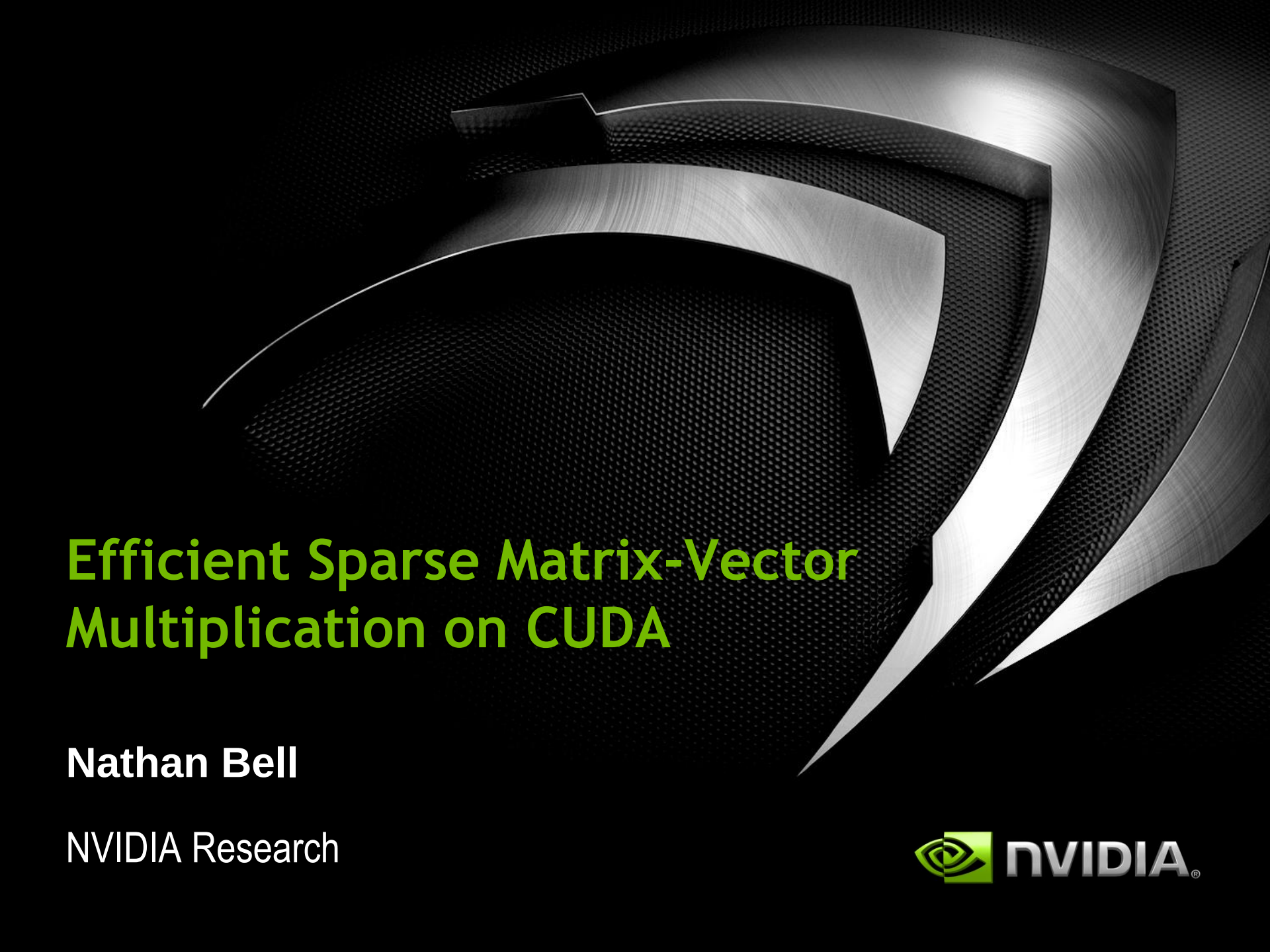


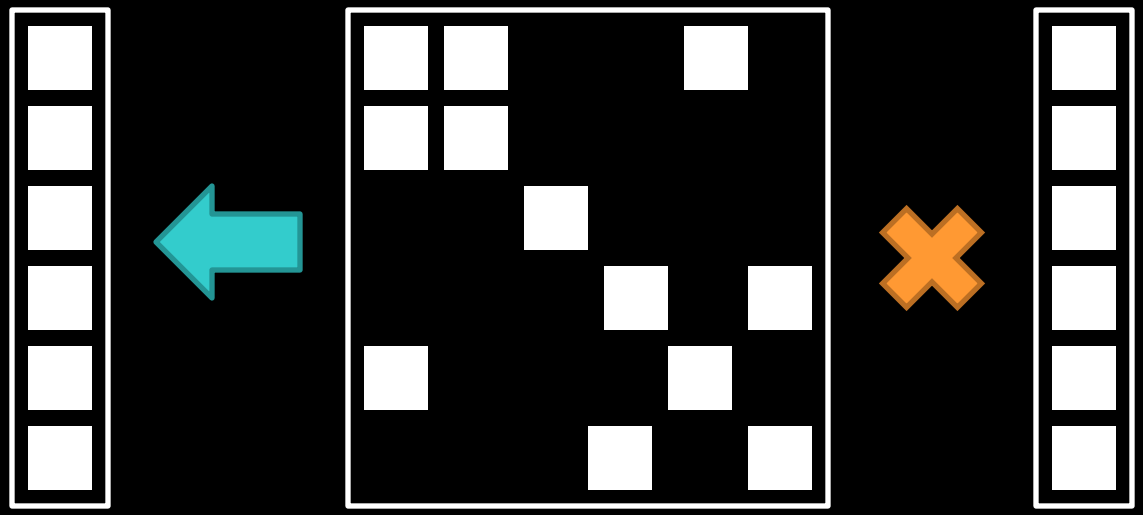
Efficient Sparse Matrix-Vector Multiplication on CUDA

Nathan Bell

NVIDIA Research

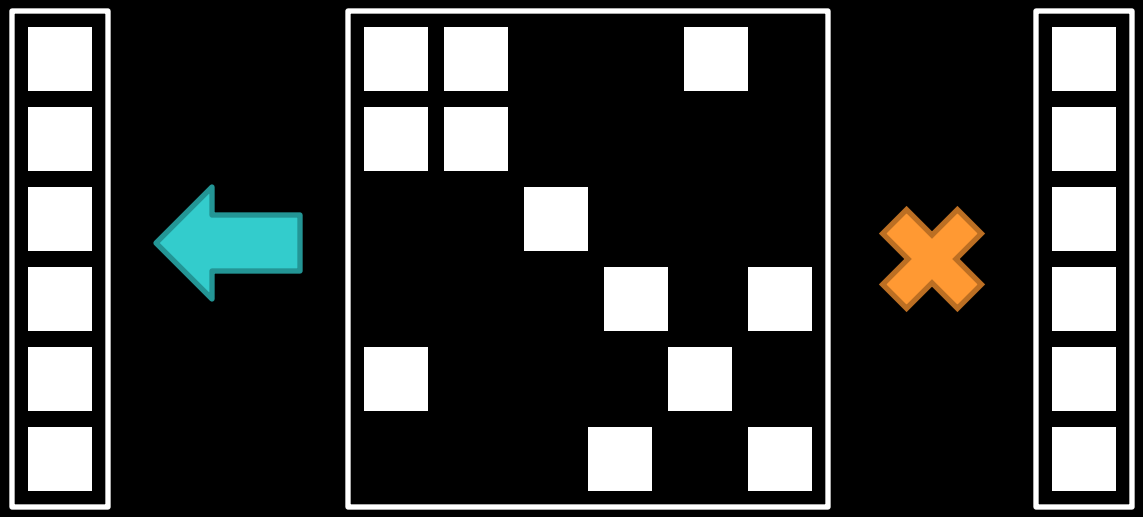
Characteristics of SpMV (y = A*x)

- Memory bound
  - Flop : Byte ratio is very low
- Generally irregular/unstructured
  - Unlike dense matrix operations (BLAS)



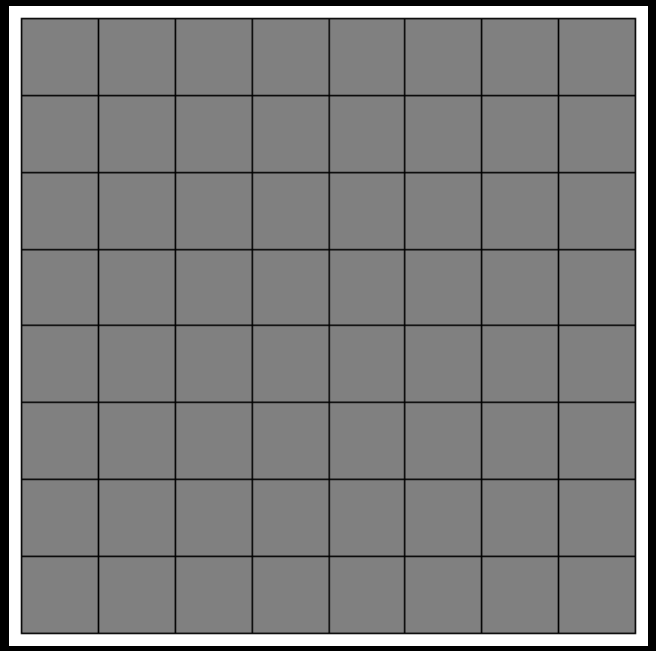
Typical Application

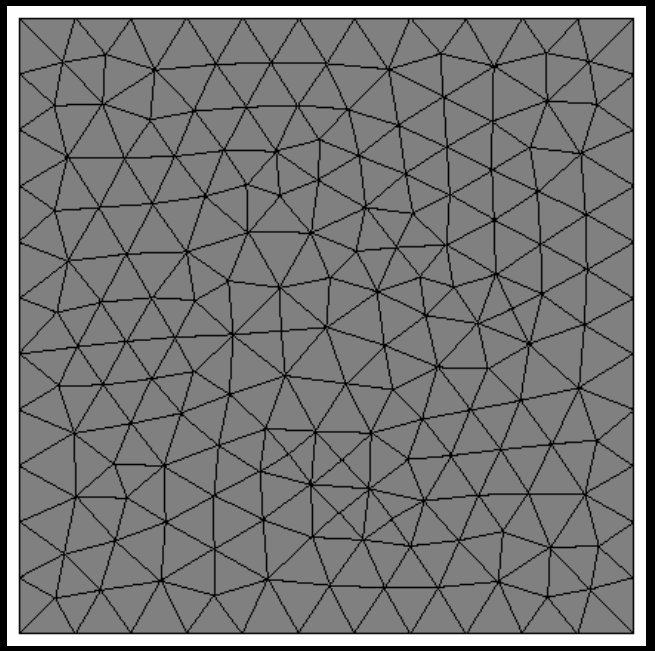
- Iterative methods for linear systems
  - Conjugate Gradient, GMRES, etc.
  - 100s or 1000s of SpMV operations



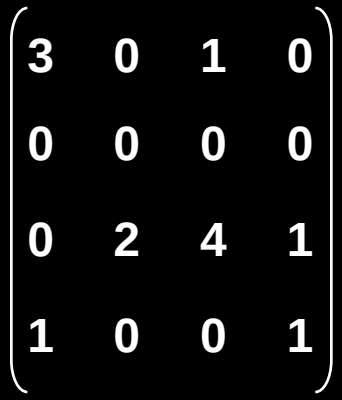
Typical Application

- Finite-Element Methods
  - Discretize PDEs on structured or unstructured meshes
  - Mesh determines matrix sparsity structure





Compressed Sparse Row (CSR)



		Row 0	Row 2	Row 3
Nonzero values	`data[7] = {`	`3, 1,`	`2, 4, 1,`	`1, 1` `};`
Column indices	`indices[7] = {`	`0, 2,`	`1, 2, 3,`	`0, 3` `};`
Row pointers	`ptr[5] = { 0, 2, 2, 5, 7 };`			

CSR SpMV Kernel (CPU)


```
for (int row = 0; row < num_rows; row++){
    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;
}
```

		Row 0	Row 2	Row 3
Nonzero values	`data[7]`	`= { 3, 1, 2, 4, 1, 1, 1 };`		
Column indices	`indices[7]`	`= { 0, 2, 1, 2, 3, 0, 3 };`		
Row pointers	`ptr[5]`	`= { 0, 2, 2, 5, 7 };`		

Parallelizing CSR SpMV

- Straightforward approach
  - One thread per matrix row

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

CSR SpMV Kernel (CUDA)


```
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;
}
```

		Row 0	Row 2	Row 3
Nonzero values	`data[7] = {`	`3, 1,`	`2, 4, 1,`	`1, 1 };`
Column indices	`indices[7] = {`	`0, 2,`	`1, 2, 3,`	`0, 3 };`
Row pointers	`ptr[5] = { 0, 2, 2, 5, 7 };`			

Comparing Kernels (CPU)


```
void
csr_spmv_kernel(const int num_rows,
                const int   * ptr,
                const int   * indices,
                const float * data,
                const float * x,
                      float * y)
{
    for (int row = 0; row < num_rows; row++){
        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;
   }
}
```

Comparing Kernels (CUDA)


```
__global__ void
csr_spmv_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;
   }
}
```

Problems with naïve CSR kernel

- **Execution divergence**
  - **Varying row lengths**

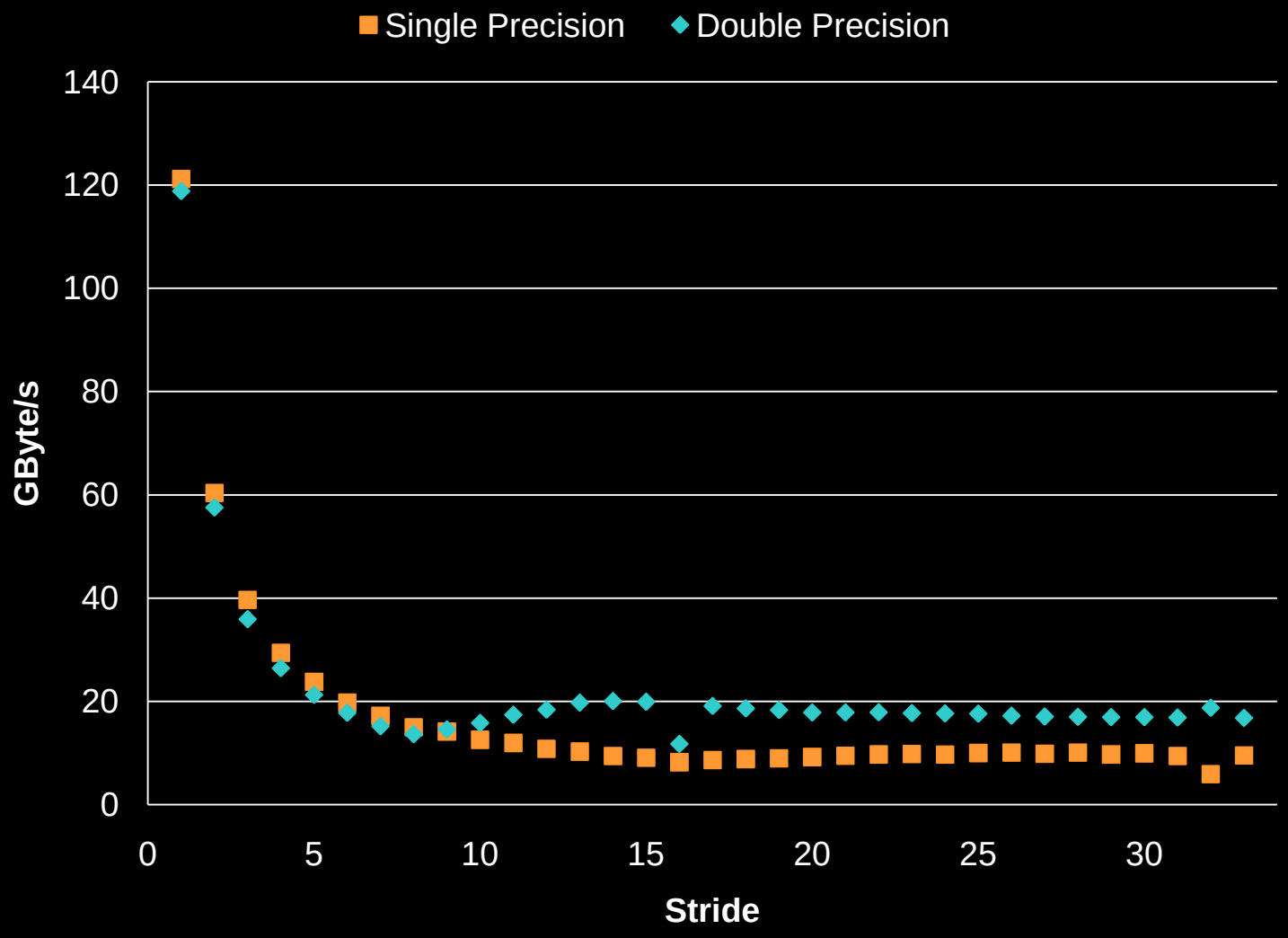
Thread 0	3	0	1	0
Thread 1	0	0	0	0
Thread 2	0	2	4	1
Thread 3	1	0	0	1

- **Memory divergence**
  - **Minimal coalescing**

		#0	#1	#0	#1	#0	#2	#1	*Iteration*
Nonzero values	`data[7]` = {	3,	1,	2,	4,	1,	1,	1	};
Column indices	`indices[7]` = {	0,	2,	1,	2,	3,	0,	3	};

Row pointers `ptr[5]` = { 0, 2, 2, 5, 7 };

Memory Bandwidth (axpy)



Alternative CSR kernel

- One warp per matrix row
  - Finer granularity
  - Sensitive to row size

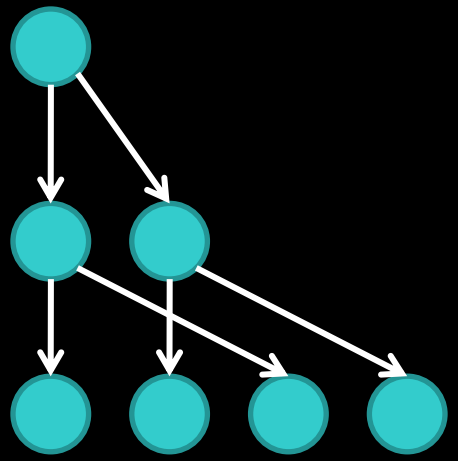
Warp 0	3 0 1 0
Warp 1	0 0 0 0
Warp 2	0 2 4 1
Warp 3	1 0 0 1

- Better coalescing
  - Contiguous access

```
                                  #0  #0  #2  #2  #2  #3  #3   Warp
Nonzero values   data[7]    = { 3,  1,  2,  4,  1,  1,  1 };
Column indices   indices[7] = { 0,  2,  1,  2,  3,  0,  3 };
Row pointers     ptr[5]     = { 0,  2,  2,  5,  7 };
```

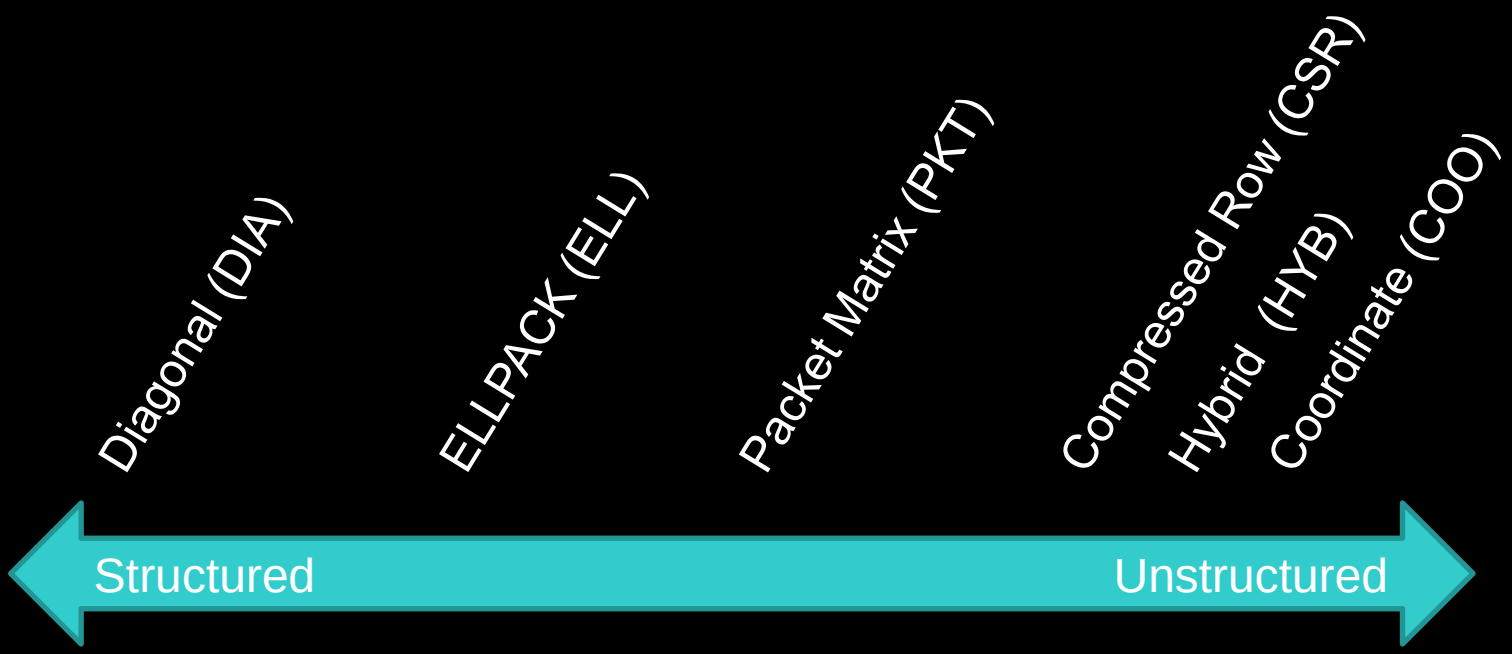
Alternative CSR kernel

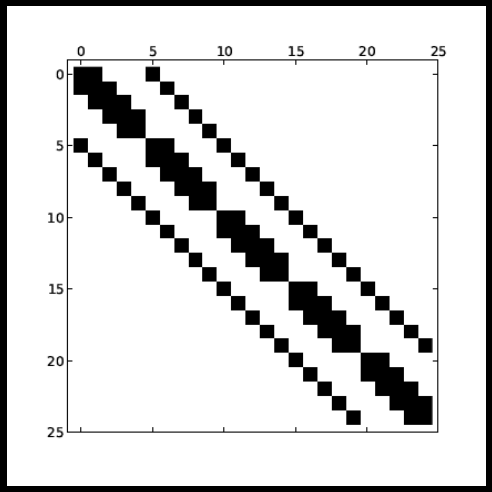
- Uses parallel reduction (warp)

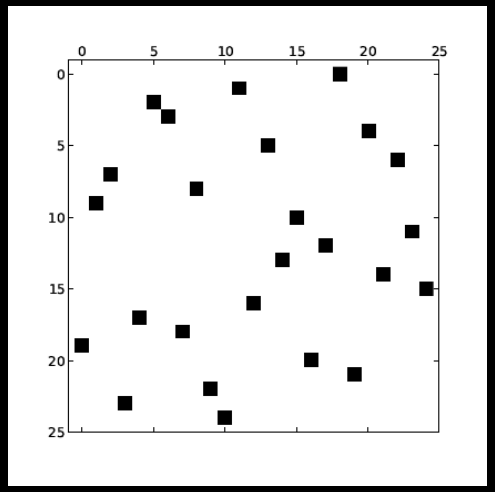


```
__shared__ sdata float[4];
if (threadIdx.x < 2)
        sdata[threadIdx.x] += sdata[threadIdx.x +  2];
if (threadIdx.x < 1)
        sdata[threadIdx.x] += sdata[threadIdx.x +  1];
```

Tradeoffs: Matrix Representations





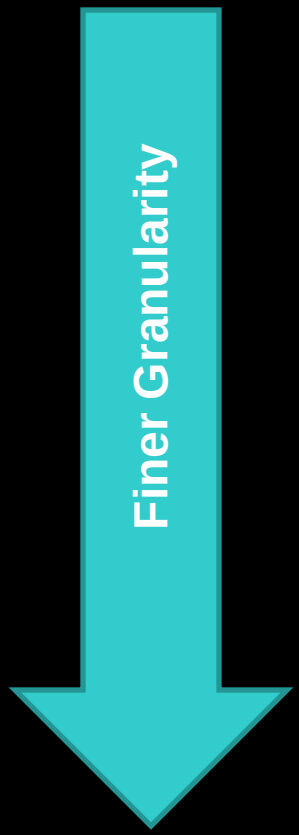


In the Tech Report…

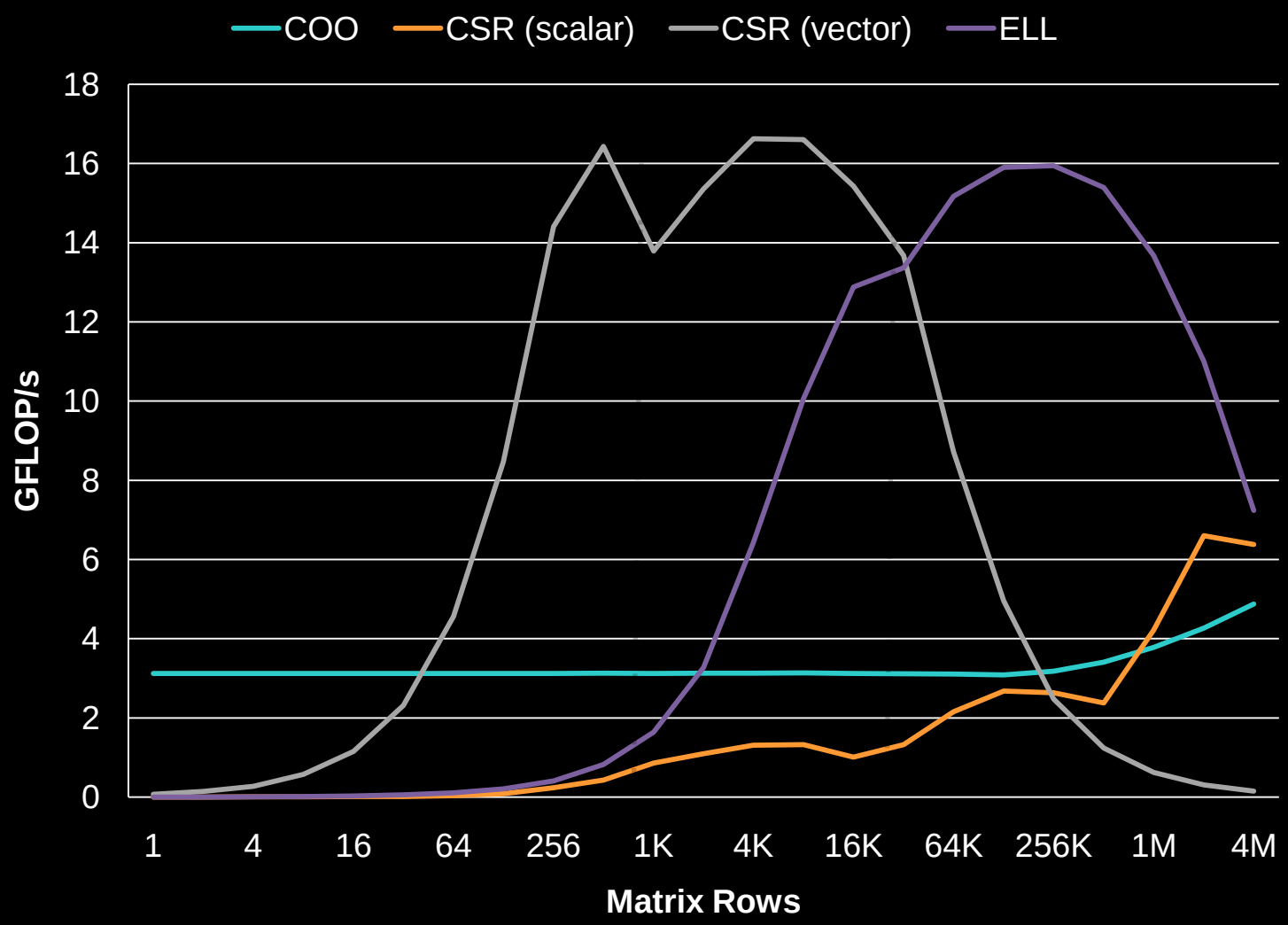
- Sparse matrix format tradeoffs
  - Generality vs. Speed
  - Computational intensity (FLOPs/Byte)
  - Appropriate formats for various matrix classes
- Hybrid format
  - Split into ELL + COO
  - Most matrix entries stored in ELL (fast)
  - Remaining entries stored in COO (robust)

Exposing Parallelism

- ELL & CSR (scalar)
  - One thread per row
- CSR (vector)
  - One warp per row
- COO
  - One thread per nonzero

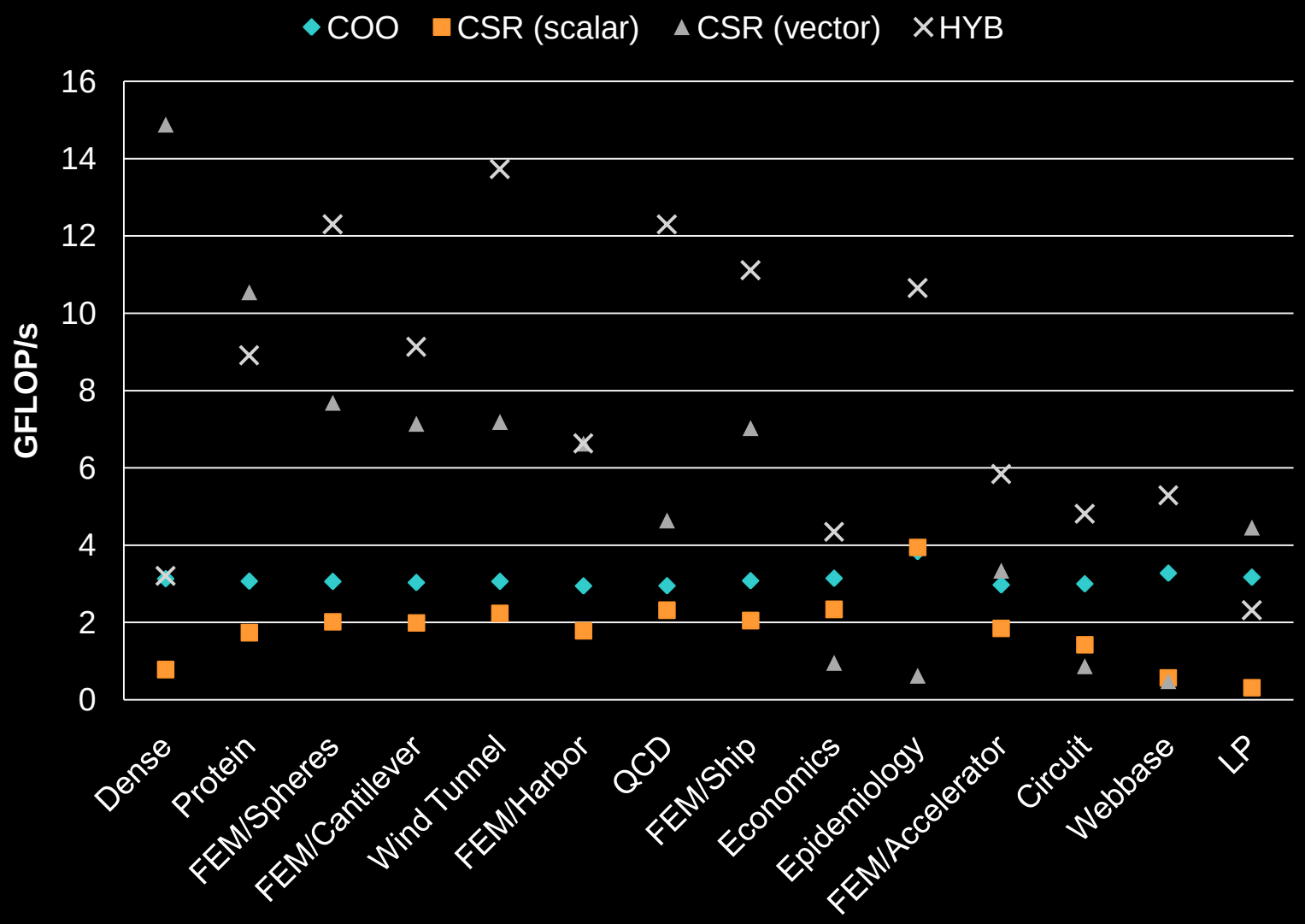


Exposing Parallelism

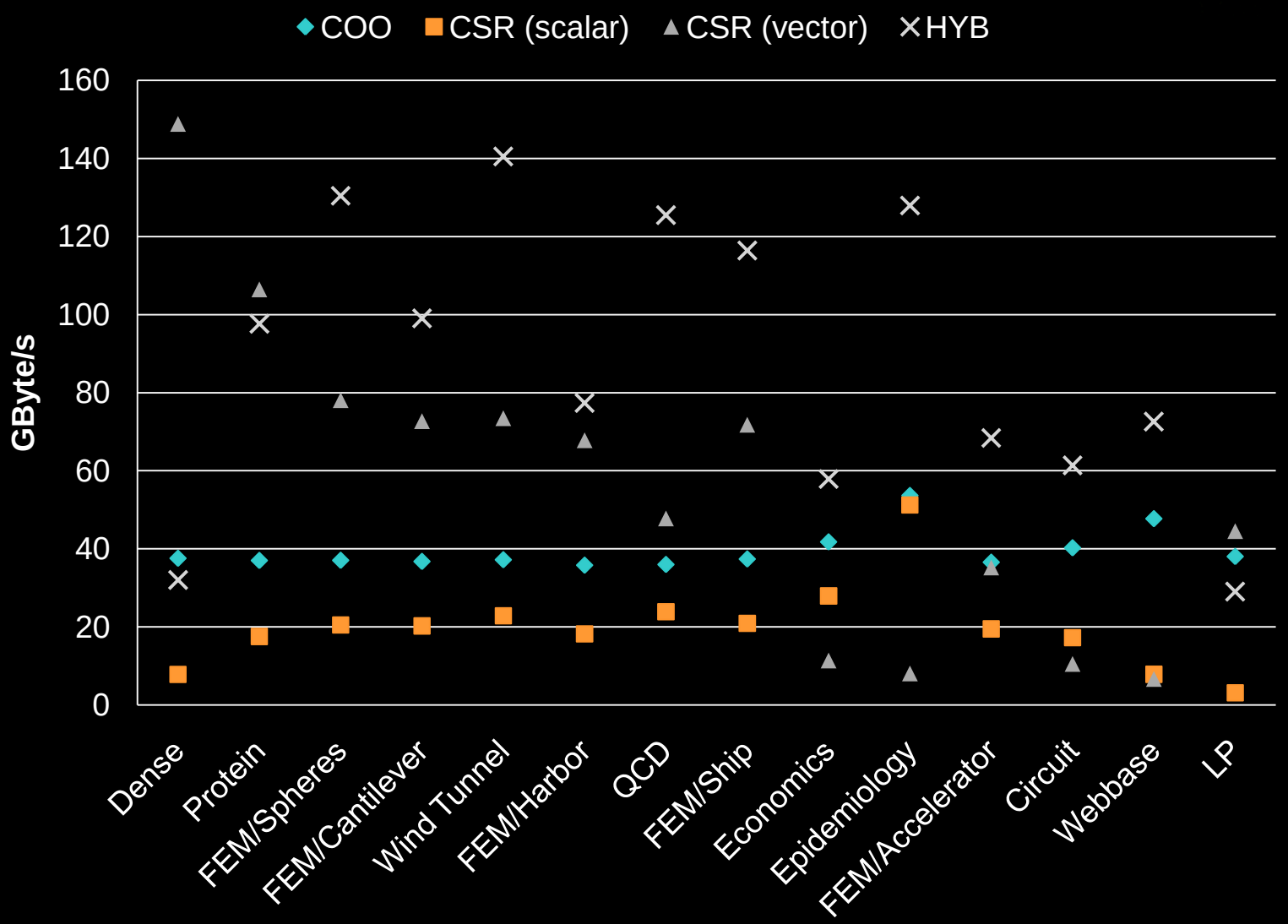


Dense matrices with 4M entries (rows * cols = 4M)

Unstructured Matrices



Unstructured Matrices



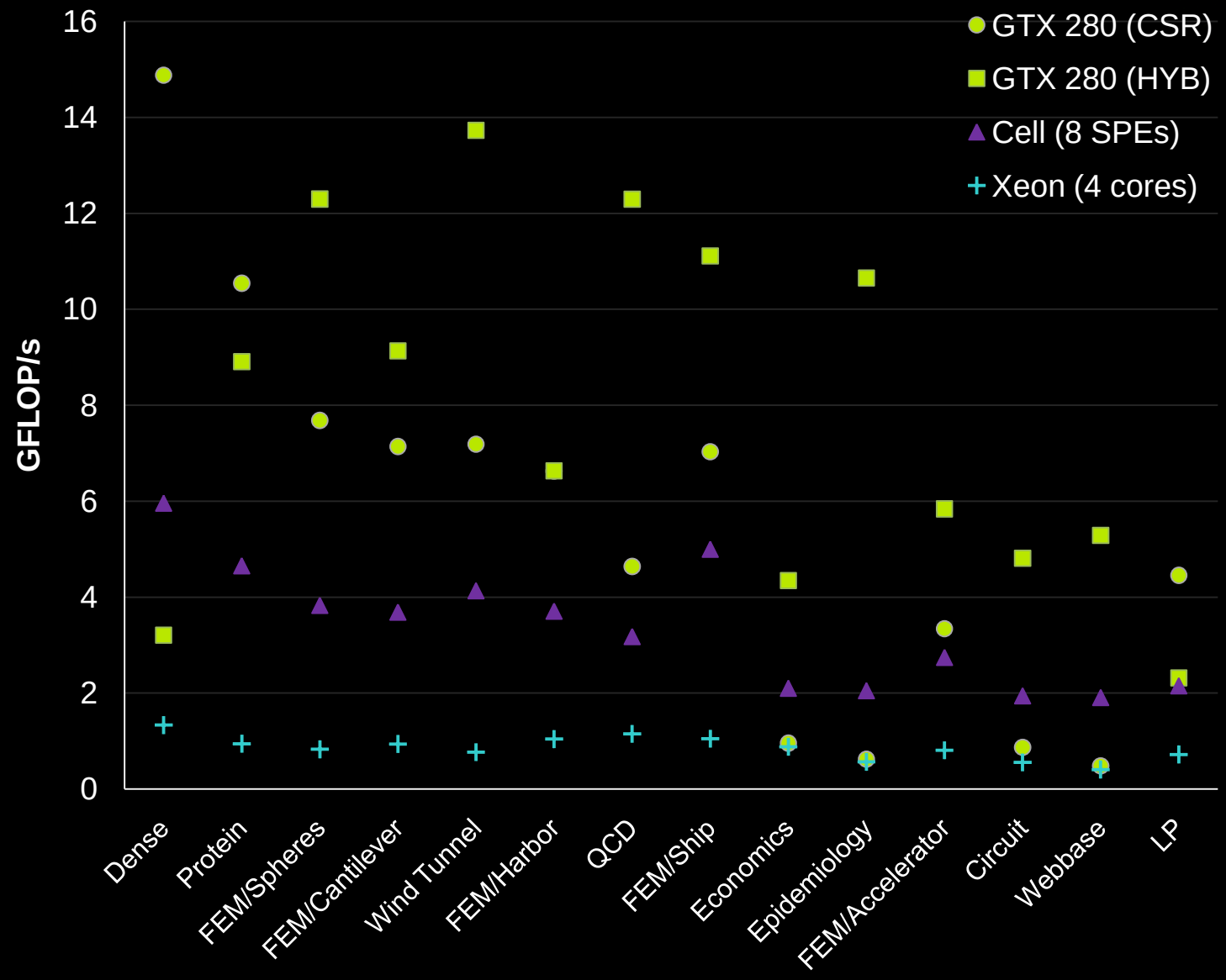
Performance Comparison

Name	Sockets	Cores	Clock (GHz)	Notes
Cell	1	8 (SPEs)	3.2	IBM QS20 Blade (half)
Xeon	1	4	2.3	Intel Clovertown
Dual Cell	2	16 (SPEs)	3.2	IBM QS20 Blade (full)
Dual Xeon	2	8	2.3	2x Intel Clovertown

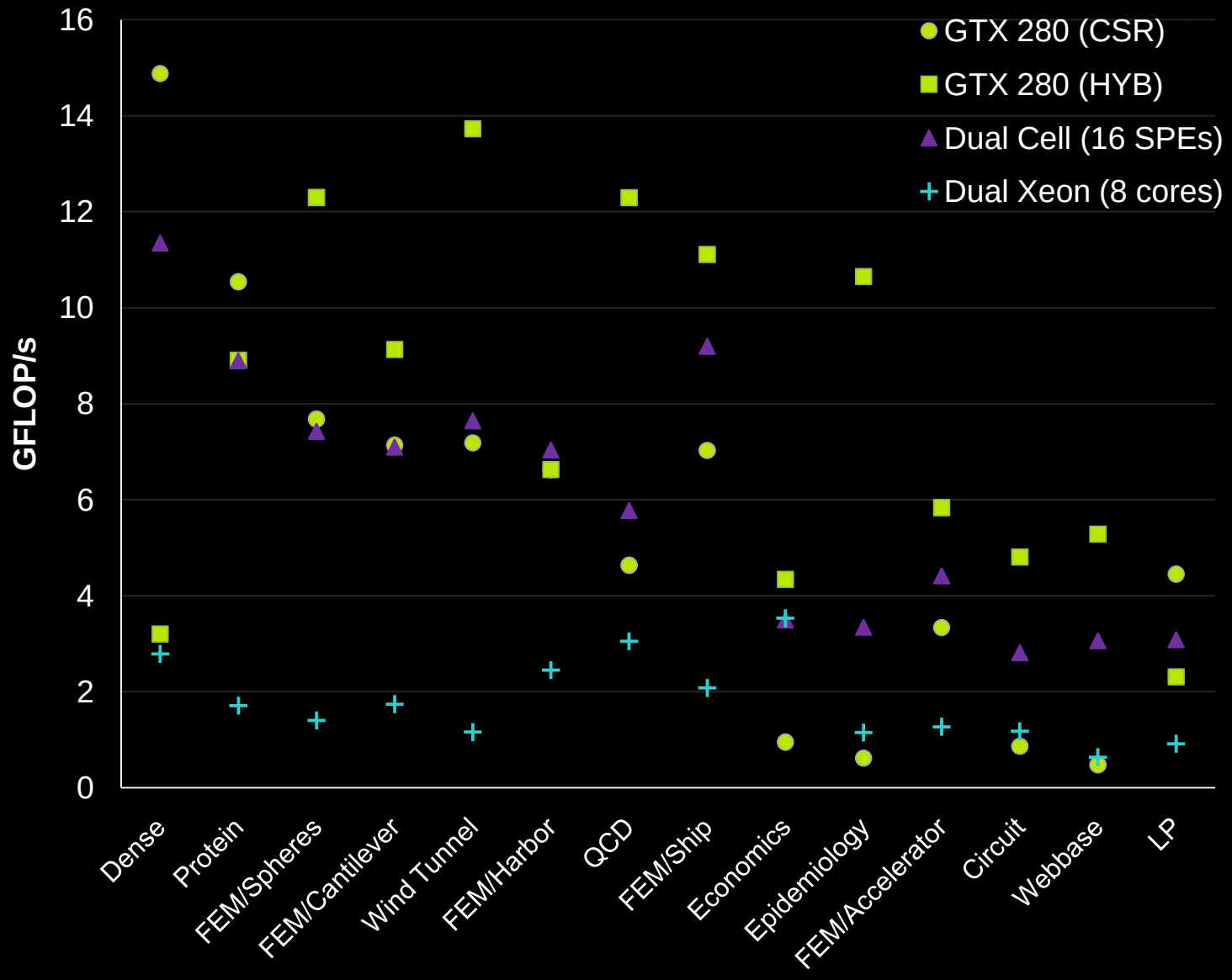
Source:

Optimization of Sparse Matrix-Vector Multiplication on Emerging Multicore Platforms.
Samuel Williams et al., Supercomputing 2007.

Performance Comparison



Performance Comparison



Further Reading

- **Tech Report**

  Efficient Sparse Matrix-Vector Multiplication on CUDA
  Nathan Bell and Michael Garland
  NVIDIA Tech Report NVR-2008-004, December 2008

- **Software and data sets**
  - **http://www.nvidia.com/research**

Questions?

nbell@nvidia.com