Algorithm Design for Manycore GPUs

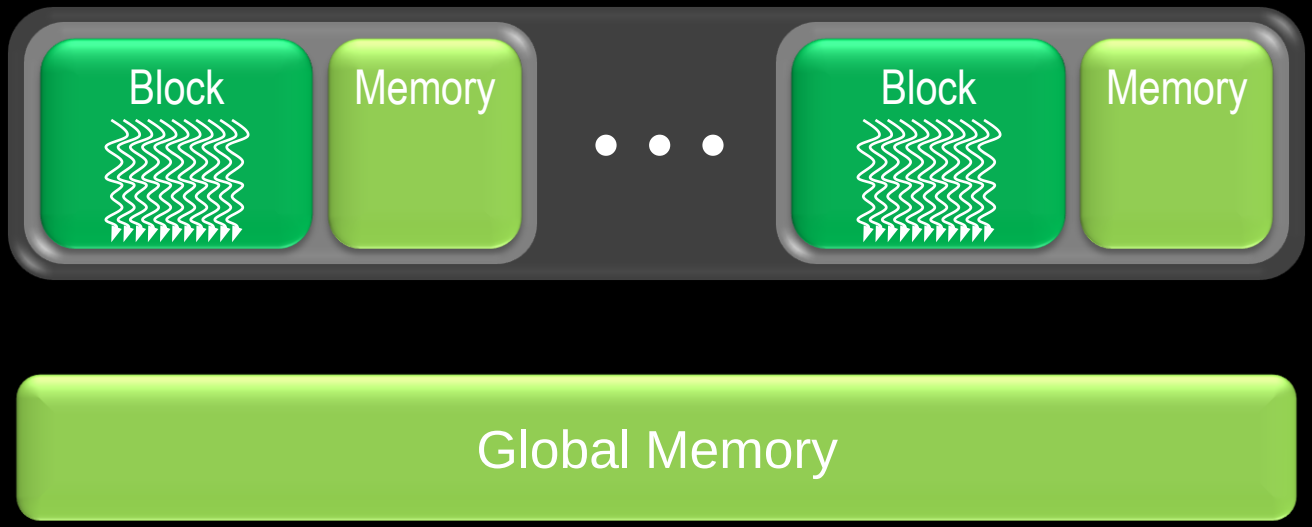
Michael Garland

NVIDIA Research

Some perspective

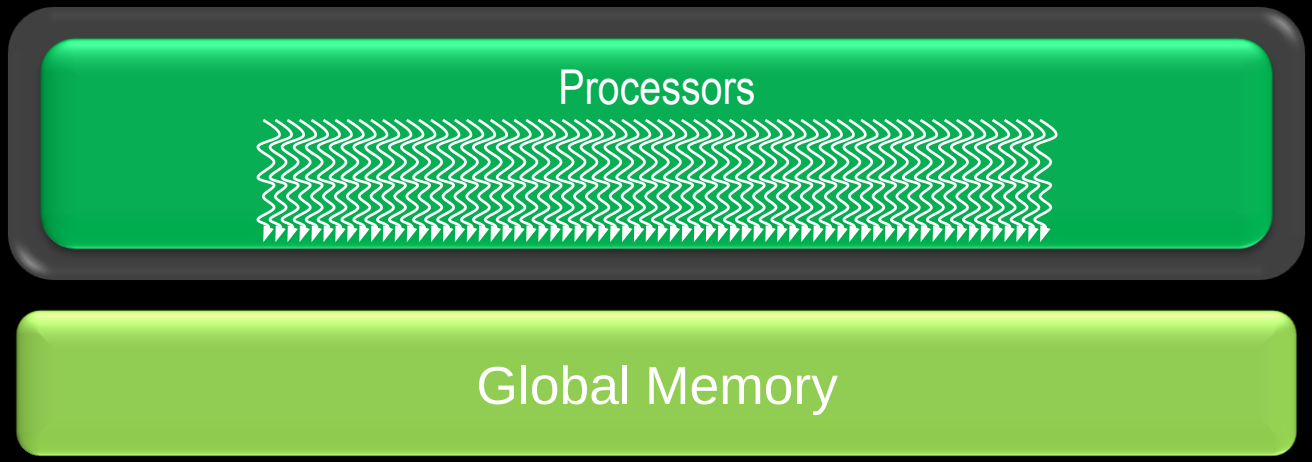
- GPUs are parallel co-processors, *not* accelerators
- 10 threads don't matter; 10,000 threads do
- Divide & conquer is often the way to win
- Some irregularity is ok if the common case is regular

CUDA Model of Parallelism



- CUDA virtualizes the physical hardware
  - thread is a virtualized scalar processor (registers, PC, state)
  - block is a virtualized multiprocessor (threads, shared mem.)
- Scheduled onto physical hardware without pre-emption
  - threads/blocks launch & run to completion
  - blocks should be independent

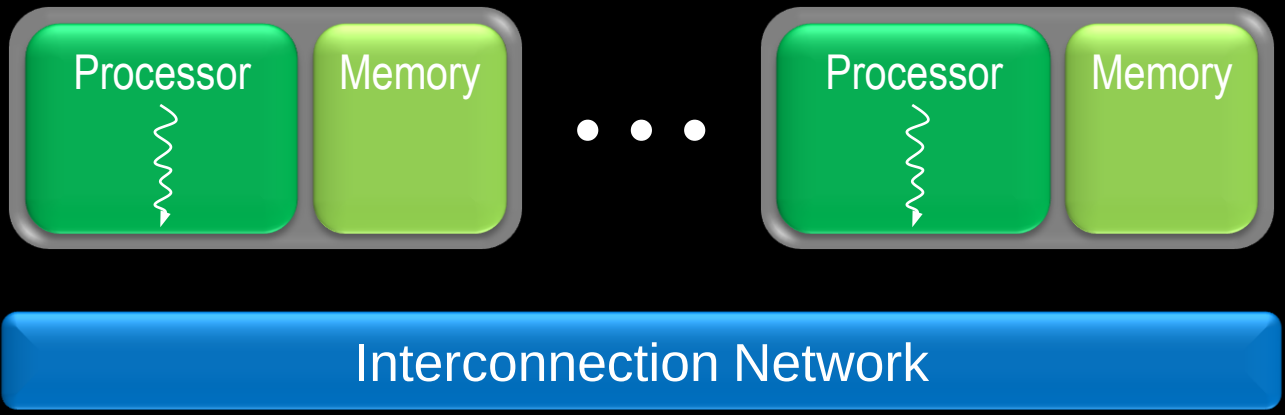
NOT: Flat Multiprocessor



- Global synchronization isn't cheap
- Global memory access times are expensive

- cf. PRAM (Parallel Random Access Machine) model

NOT: Distributed Processors



- Distributed computing is a different setting

- cf. MPI

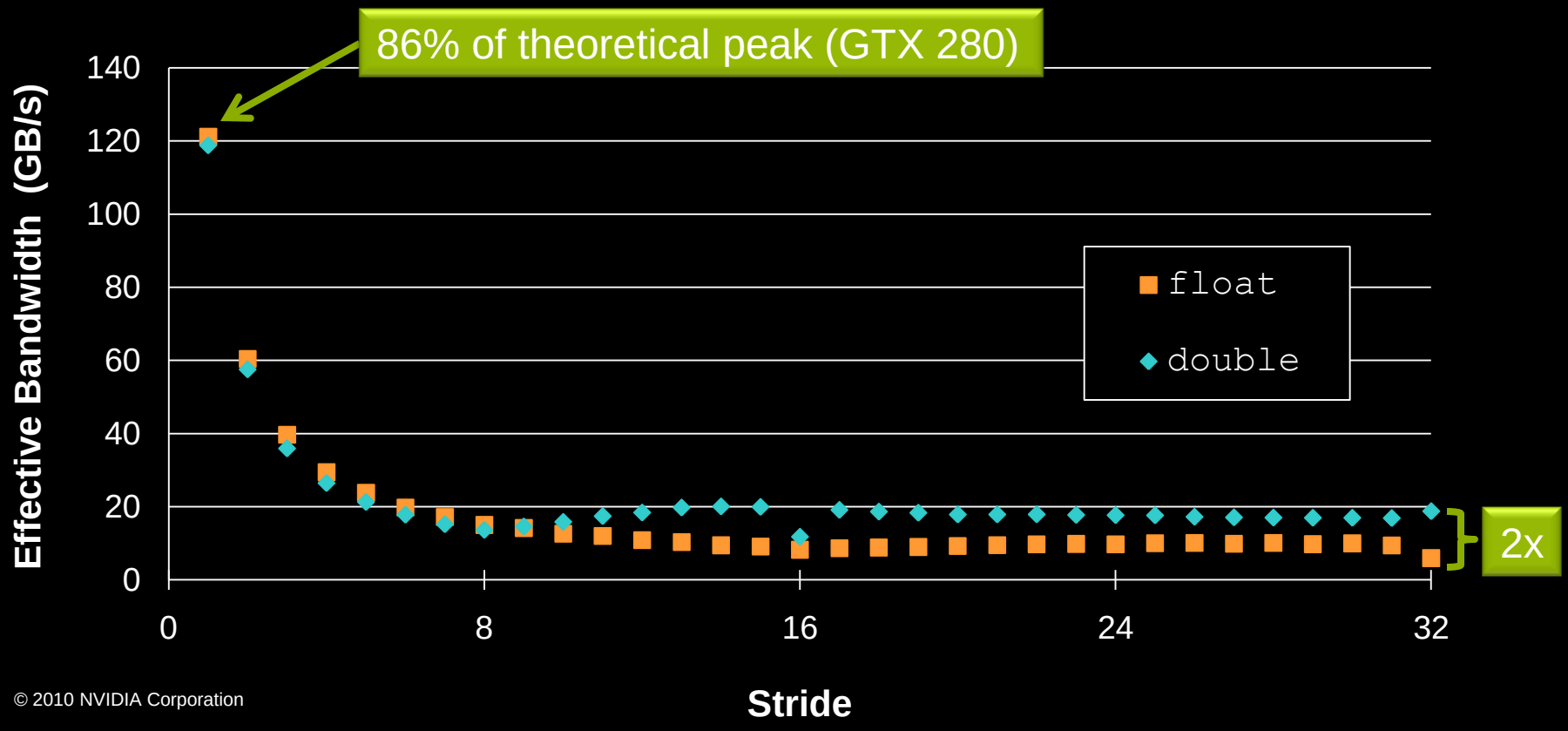
Imperatives for Efficient Design

- Expose abundant fine-grained parallelism
  - need 1000's of threads for full utilization (30K max)
- Maximize on-chip work
  - on-chip memory orders of magnitude faster
- Minimize execution divergence
  - SIMT execution of threads in 32-thread warps
- Minimize memory divergence
  - coalesced load/store across warp (~ vector load/store)

Coalescing Adjacent Loads

```
void saxpy(int n, float a, float *x, float *y, int stride)
{
    int i = blockIdx.x * blockDim.x + threadIdx.x;

    if( i<n )  y[i*stride] = a * x[i*stride] + y[i*stride];
}
```



Two regimes of parallel tasks

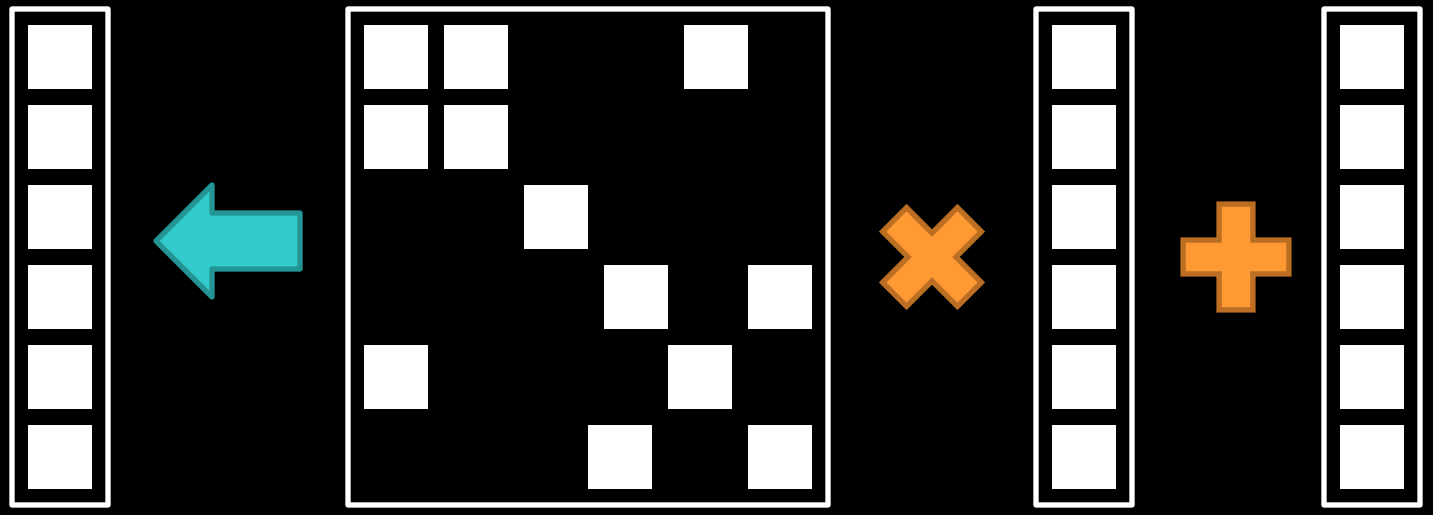
- Many independent fine-grained tasks
  - assign 1 task to each thread
  - coordination mostly at kernel boundaries
- Collection of coordinated parallel tasks
  - assign 1 task to each thread block
  - common case in divide & conquer algorithms

Implementing sparse matrix-vector multiplication on throughput-oriented processors,
Nathan Bell and Michael Garland.
Supercomputing ‘09

SPARSE MATRICES

Sparse matrix-vector multiplication

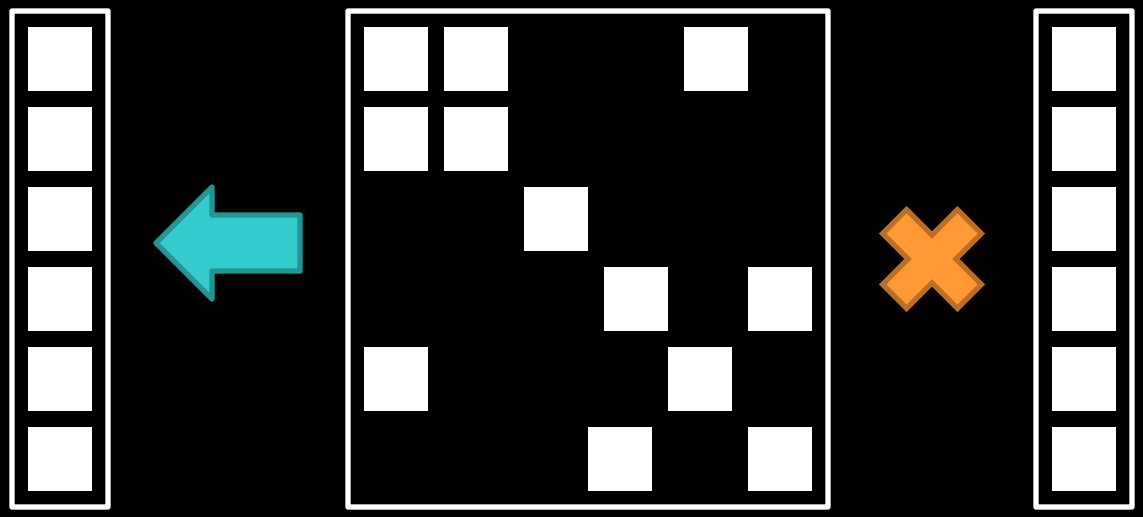
- **Compute** $y \leftarrow Ax + y$
  - **where A is sparse and x, y are dense**



- **Unlike dense methods, SpMV is generally**
  - **unstructured / irregular**
  - **entirely bound by memory bandwidth**

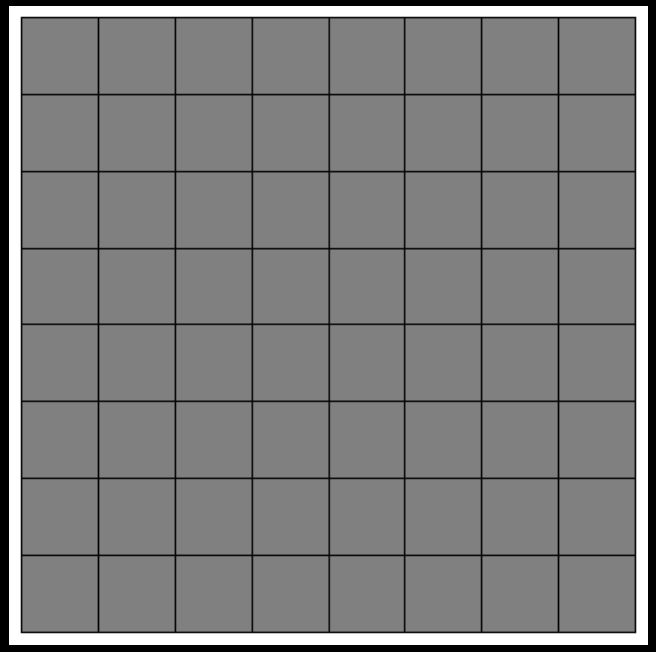
Application

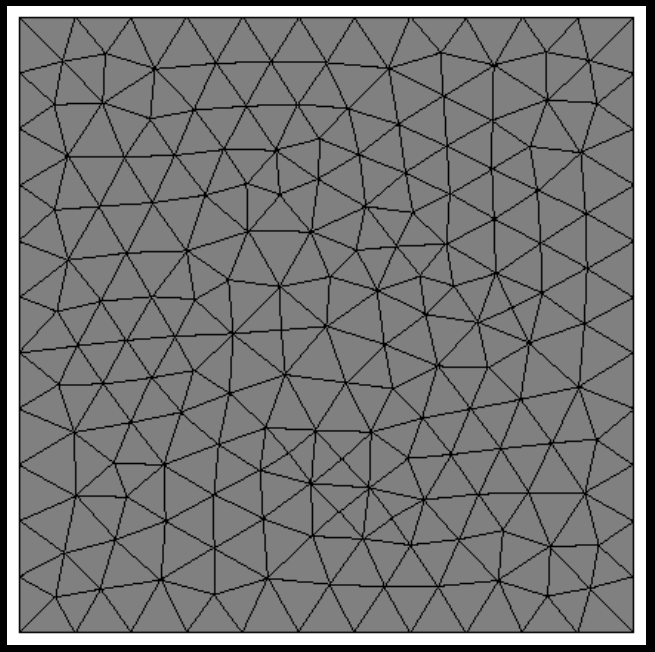
- Iterative methods for linear systems
  - Conjugate Gradient, GMRES, etc.
  - 100s or 1000s of SpMV operations (y = A x)



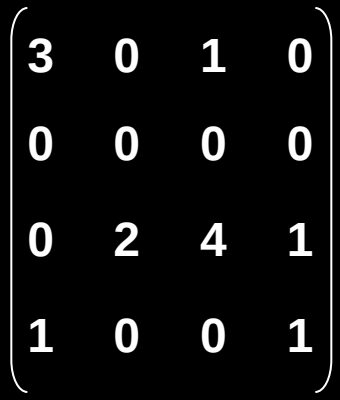
Application

- 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 (Serial)

```
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 (Serial)


```
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;
   }
}
```

Compare with OpenMP


```
void csr_openmp(… … … … … …)
{
    #pragma omp parallel for
    for(uint row=0; row<num_rows; ++row)
    {
        uint row_begin = Ap[row];
        uint row_end   = Ap[row+1];

        ... compute y[row] ...
    }
}
```

OpenMP Kernel

CUDA Kernel

```
__global__ void csr_kernel(… … … … … …)
{
    uint row = blockIdx.x*blockDim.x + threadIdx.x;

    if( row<num_rows )
    {
        uint row_begin = Ap[row];
        uint row_end   = Ap[row+1];

        ... compute y[row] ...
    }
}
```

Problems with simple CSR kernel

- **Execution divergence**
  - varying row lengths

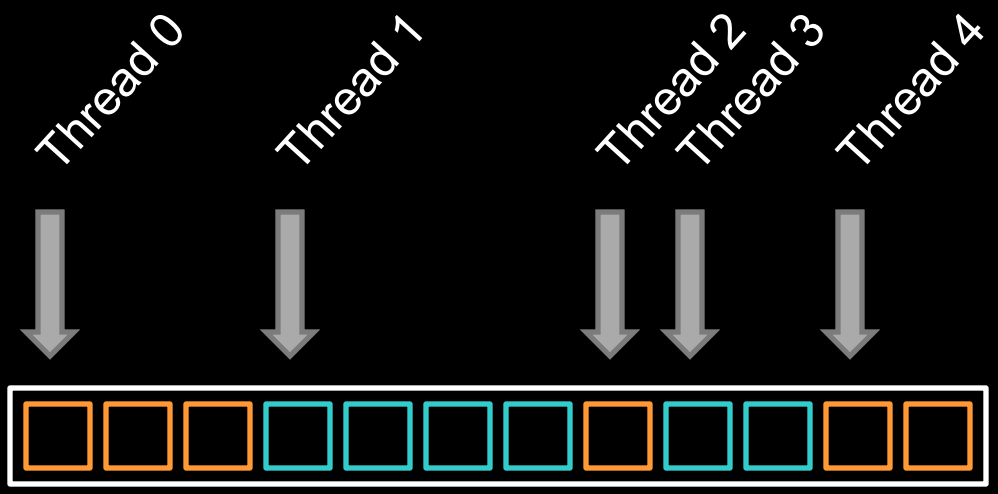
Thread 0	3	0	1	0
Thread 1	0	0	0	0
Thread 2	0	2	4	1
Thread 3	1	0	0	1

- **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 };`							

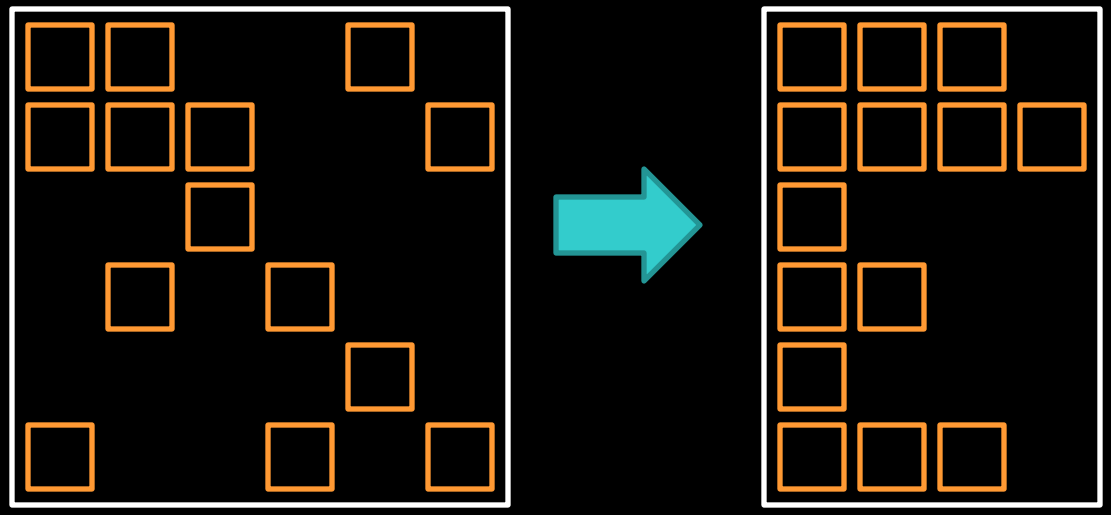
Problems with simple CSR kernel

- Memory divergence
  - minimal coalescing



Regularizing SpMV with ELL format

- Storage for K nonzeros per row
  - pad rows with fewer than K nonzeros
  - inefficient when row length varies



Regularizing SpMV with ELL format

- Quantize each row to a fix length K

	Values			Columns		
Thread 0	3	1	*	0	2	*
Thread 1	*	*	*	*	*	*
Thread 2	2	4	1	1	2	3
Thread 3	1	1	*	0	3	*

- Layout in column-major order
  - yields full coalescing

Exposing maximal parallelism

- Use COO (Coordinate) format
  - list row, column, and value for every non-zero entry

Nonzero values `data[7] = { 3, 1, 2, 4, 1, 1, 1 };`

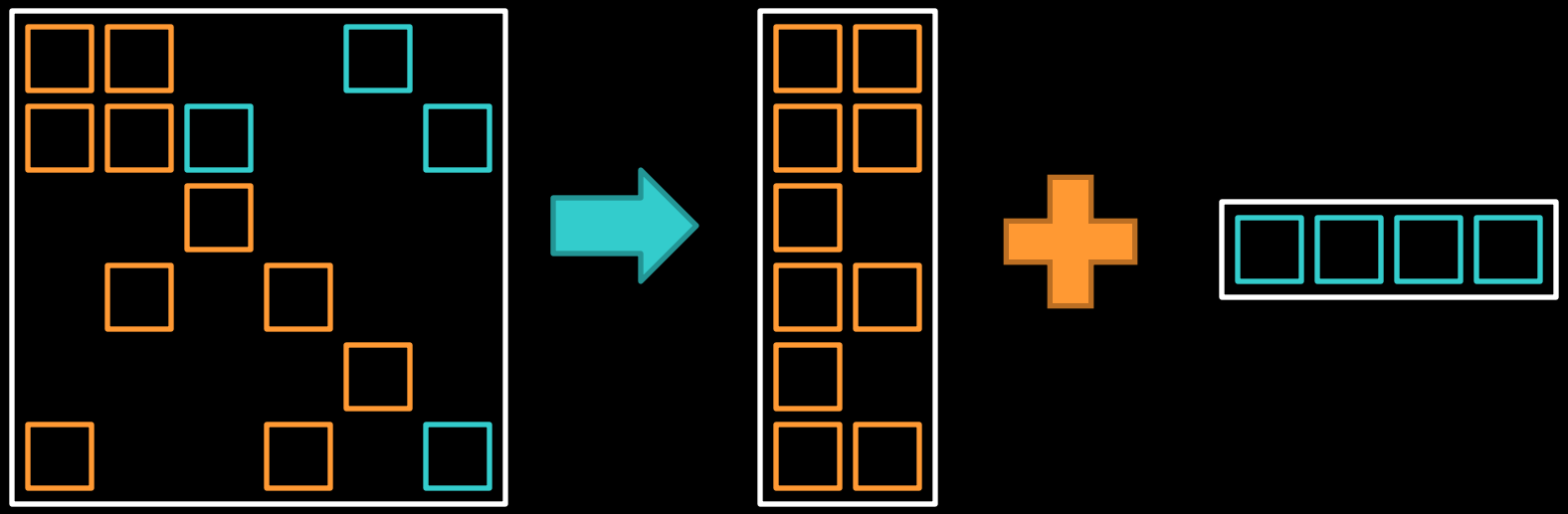
Column indices `cols[7] = { 0, 2, 1, 2, 3, 0, 3 };`

Row indices `rows[7] = { 0, 0, 1, 1, 1, 2, 2 };`

- Assign one thread to each non-zero entry
  - each thread computes an `A[i,j]*x[j]` product
  - sum products with segmented reduction algorithm
  - largely insensitive to row length distribution

Hybrid Format

- ELL handles *typical* entries
- COO handles *exceptional* entries
  - Implemented with segmented reduction



Sparse formats for different matrices

(DIA) Diagonal

(ELL) ELLPACK

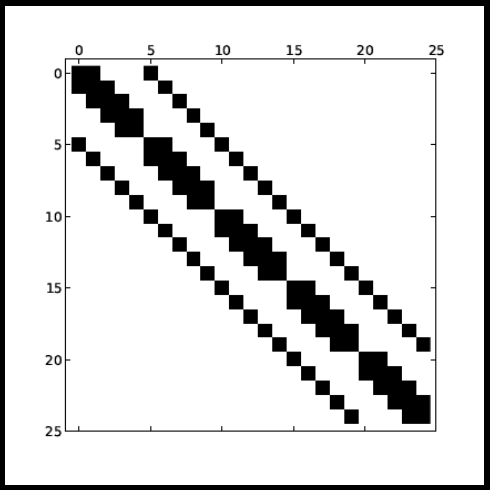
(CSR) Compressed Row

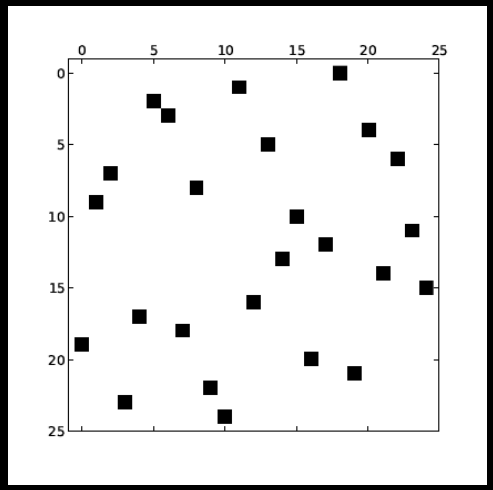
(HYB) Hybrid

(COO) Coordinate

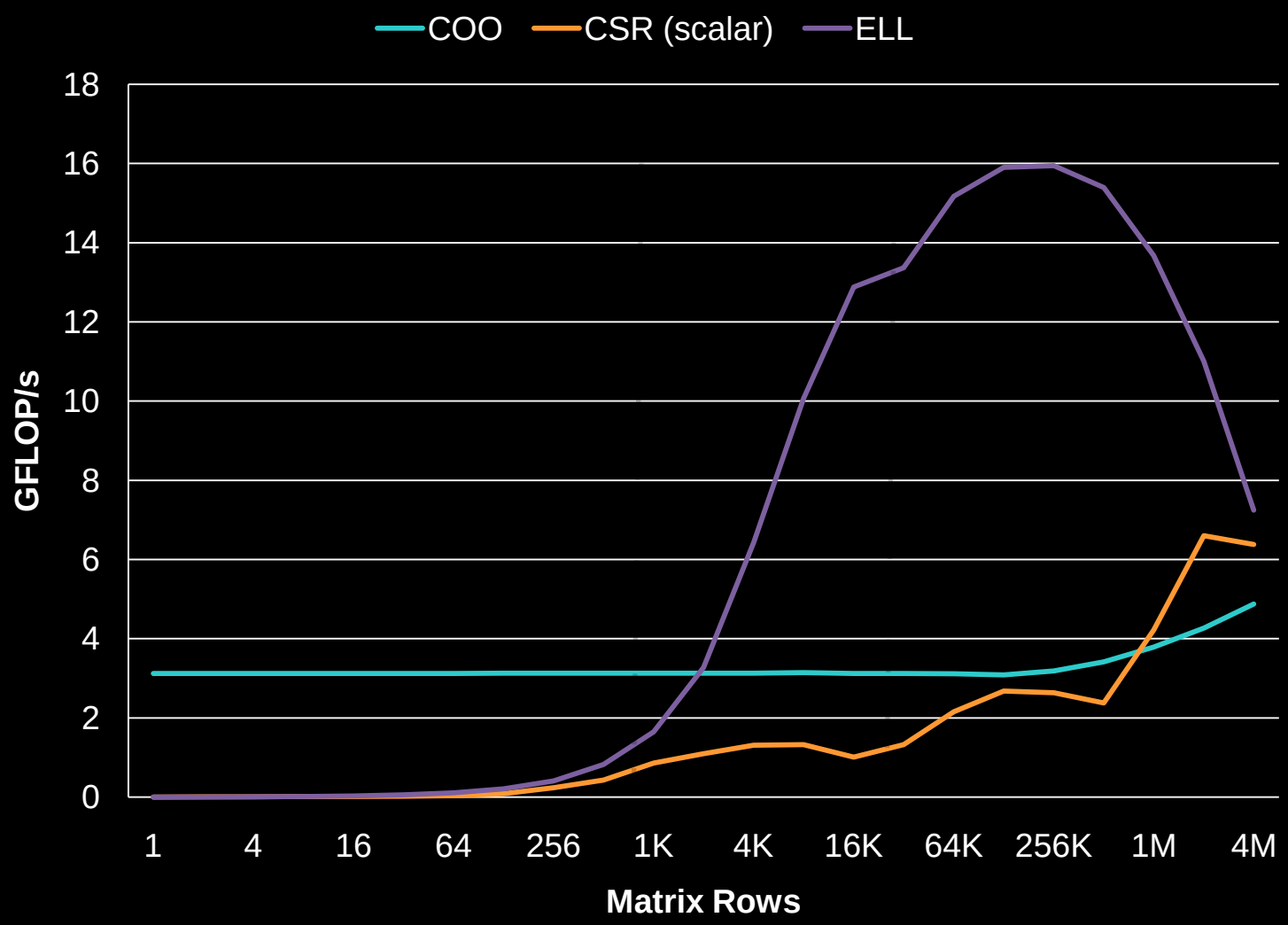
Structured

Unstructured





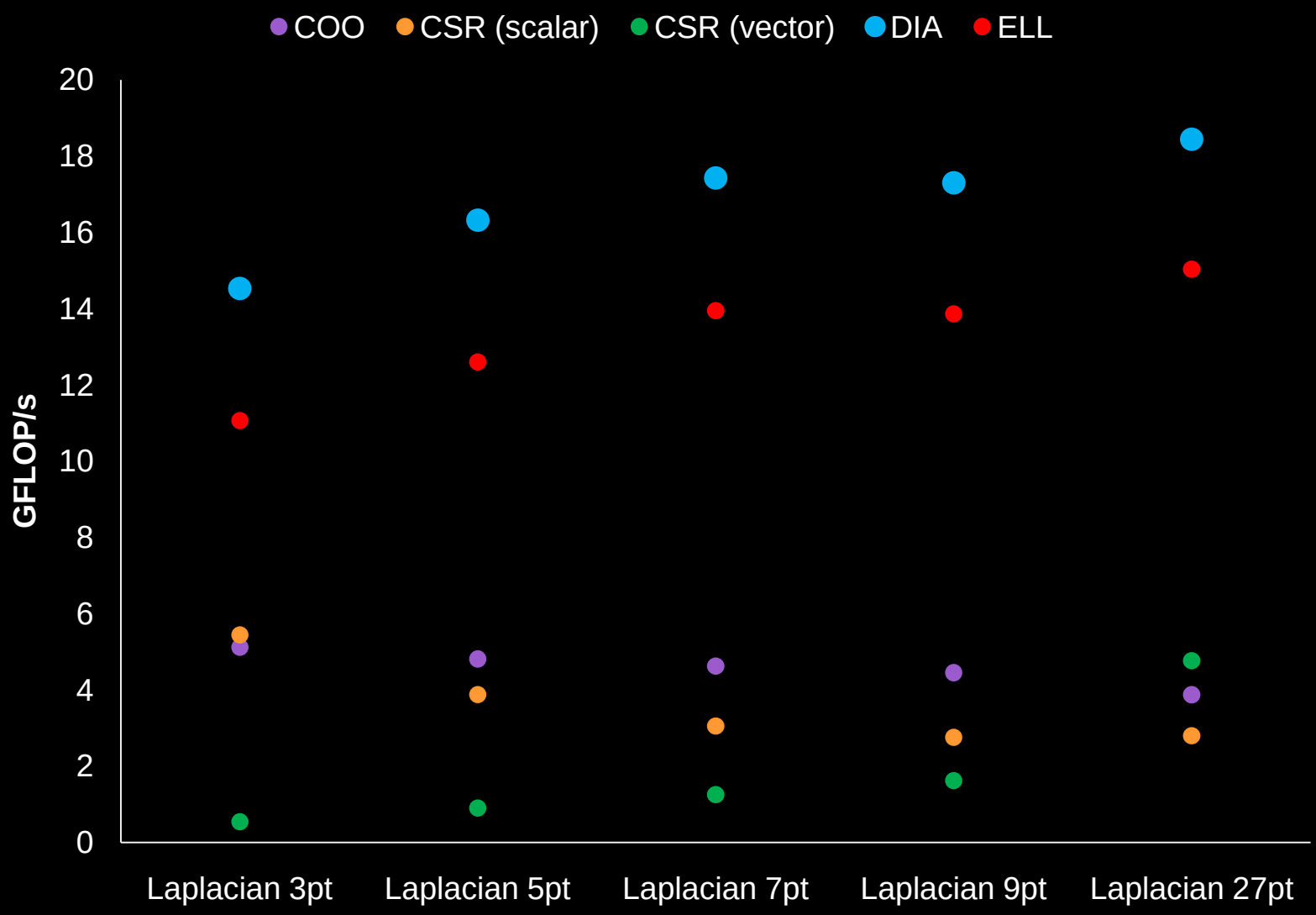
Granularity effects with 4M nonzeros



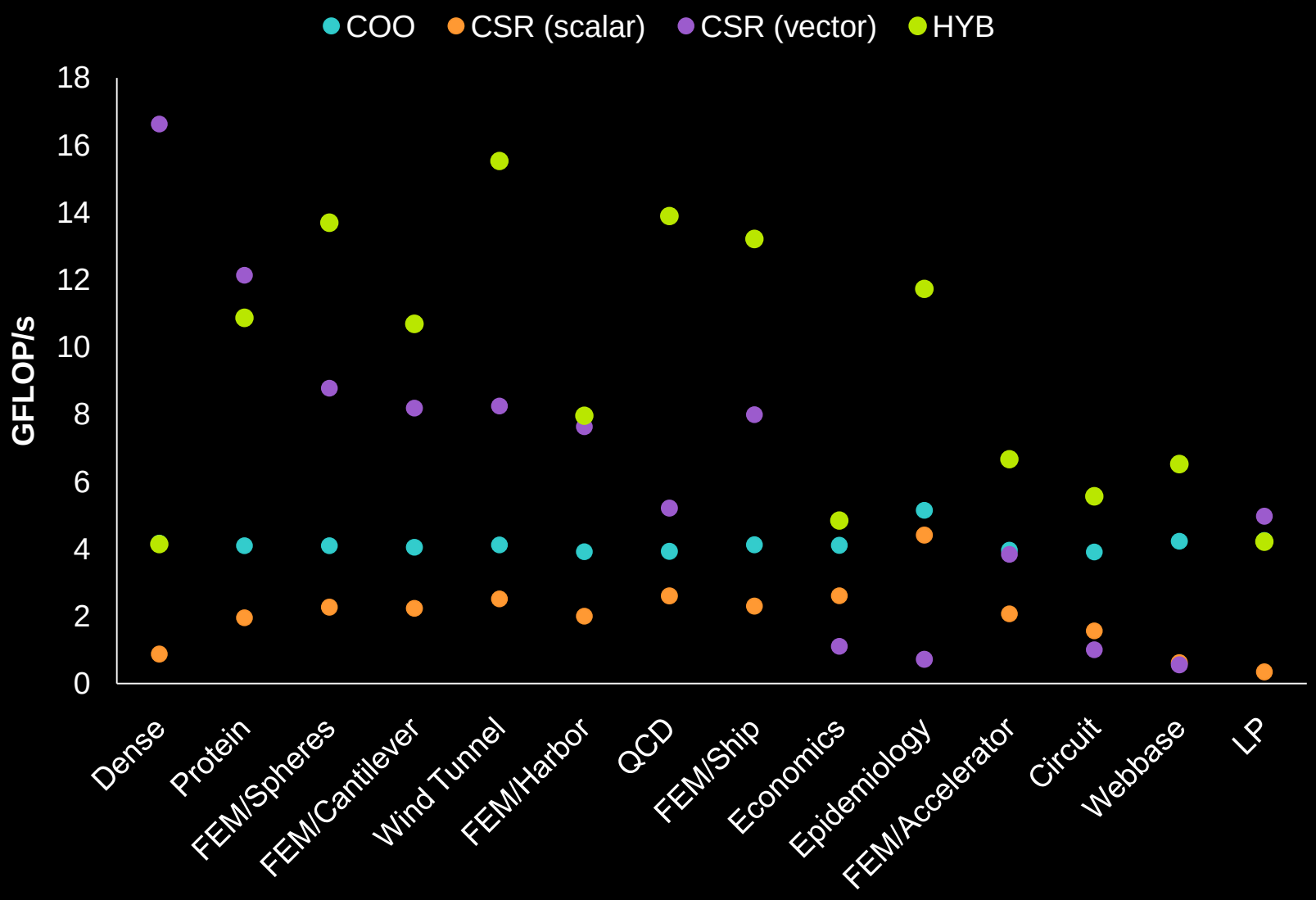
Performance Results

- GeForce GTX 285
  - Peak Memory Bandwidth: 159 GByte/s
  - All results in double precision
  - Source vector accessed through texture cache
- Structured Matrices
  - Common stencils on regular grids
- Unstructured Matrices
  - Wide variety of applications and sparsity patterns

Structured Matrices



Unstructured Matrices



Performance Comparison

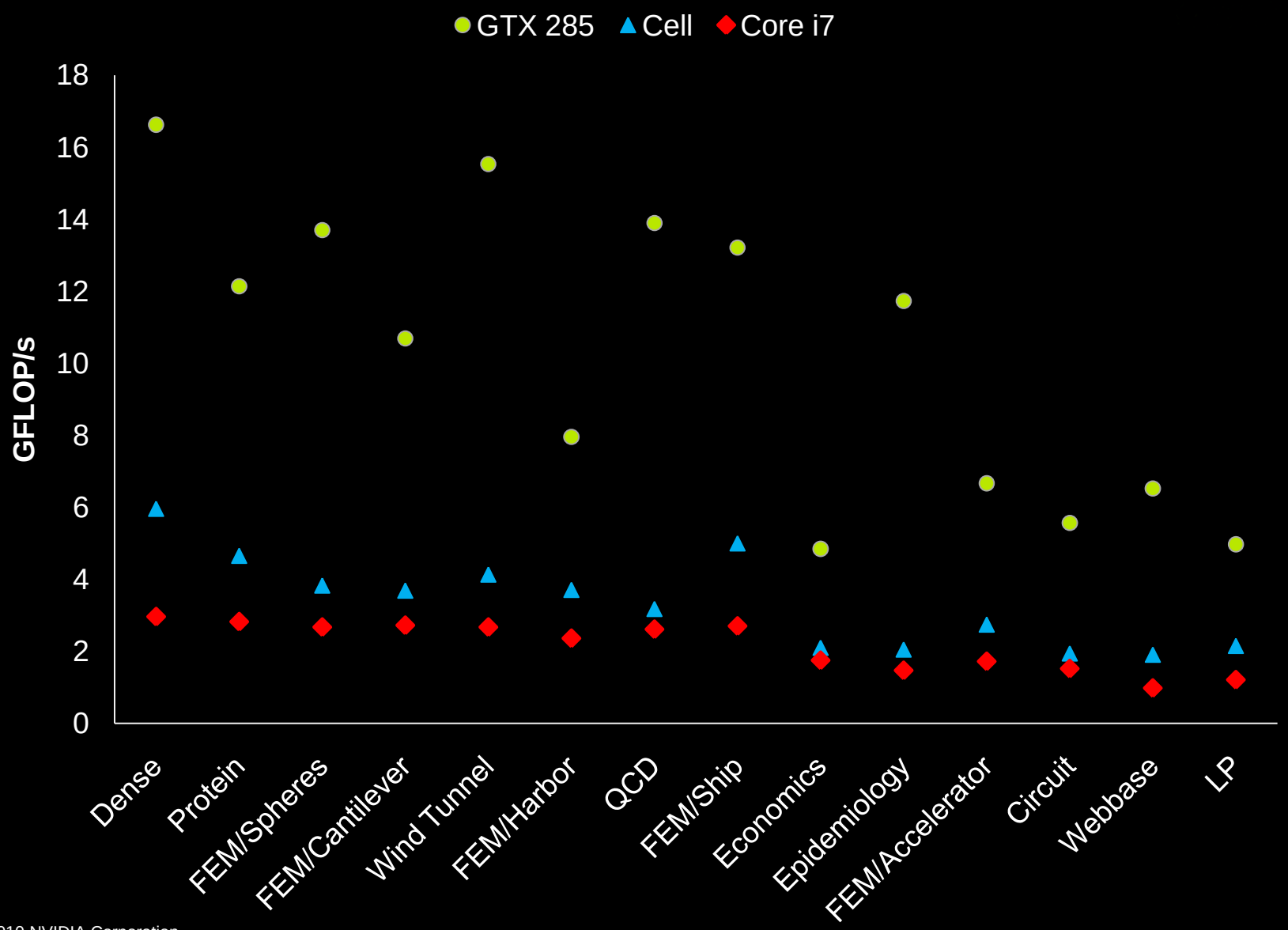
System	Cores	Clock (GHz)	Notes
GTX 285	240	1.5	NVIDIA GeForce GTX 285
Cell	8 (SPEs)	3.2	IBM QS20 Blade (half)
Core i7	4	3.0	Intel Core i7 (Nehalem)

Sources:

Implementing Sparse Matrix-Vector Multiplication on Throughput-Oriented Processors
N. Bell and M. Garland, Proc. Supercomputing '09, November 2009

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

Performance Comparison



Questions?

mgarland@nvidia.com

http://www.nvidia.com/cuda