

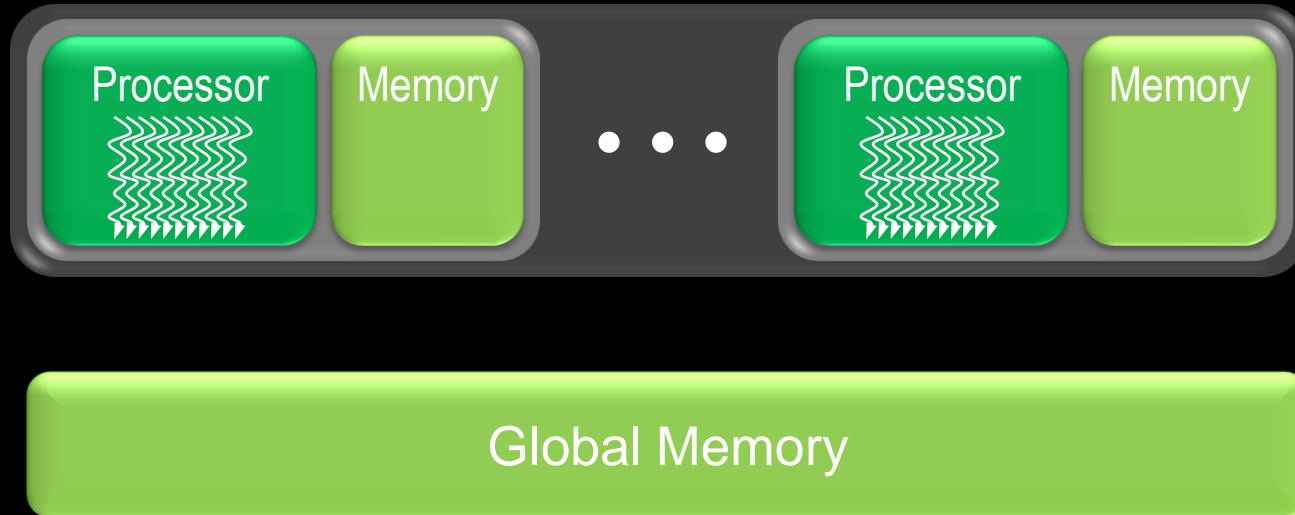
Algorithm Design for Manycore GPUs

Michael Garland

NVIDIA Research



GPU: Manycore Microprocessor



- Many processors each supporting **many hardware threads**
- **On-chip memory** near processors
- **Shared global memory** space (external DRAM)

Some perspective



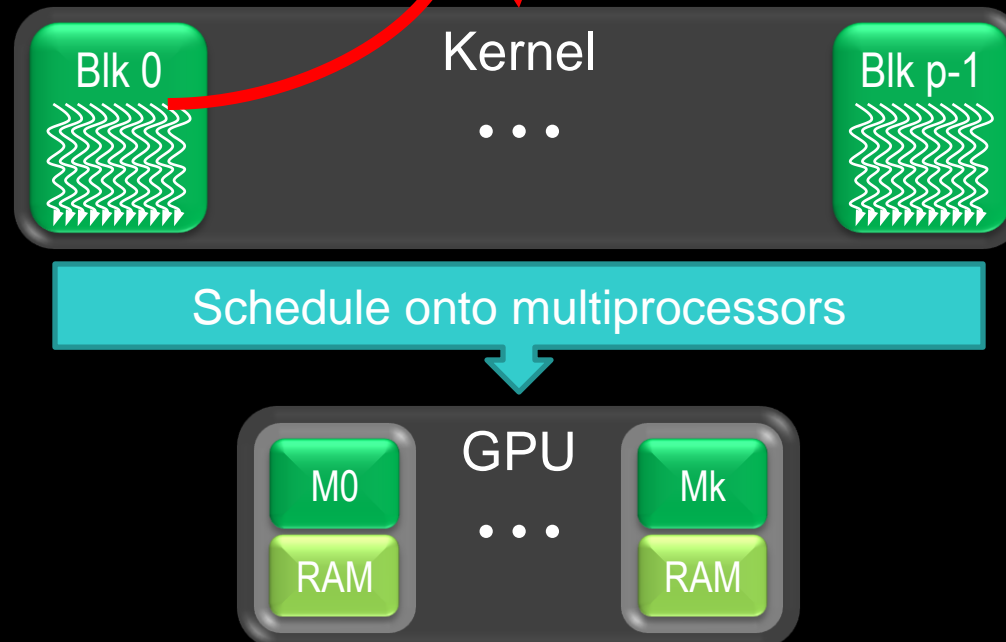
- GPUs are parallel co-processors, *not* accelerators
- 10 threads don't matter; 10,000 threads do
- Divide & conquer is often the way to win
- Some irregularity is ok if the common case is regular

CUDA in a Nutshell

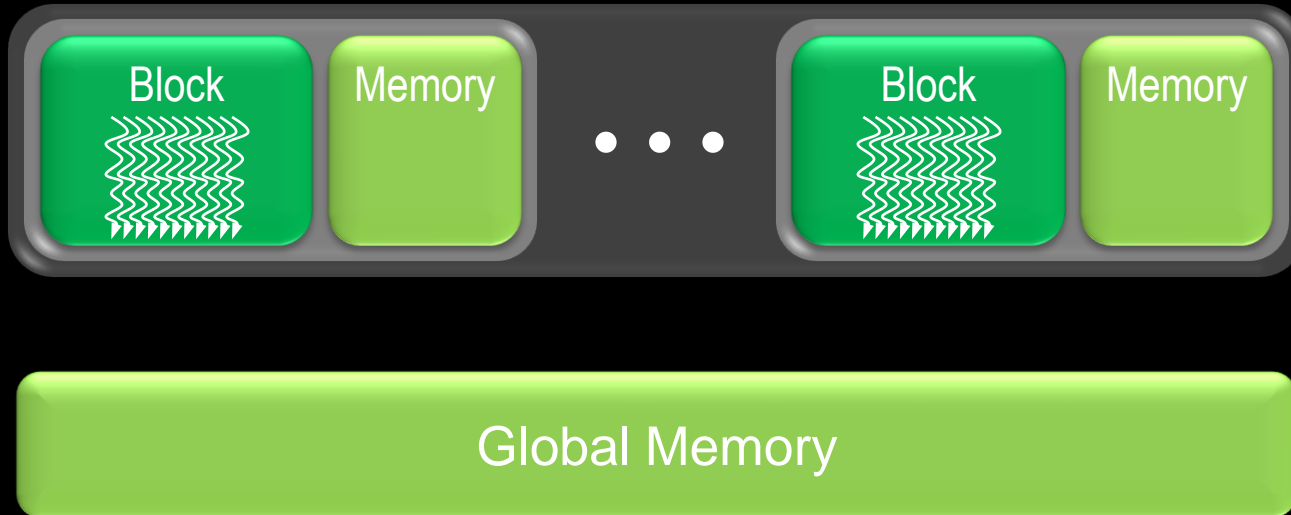


```
__host__  
void example()  
{  
    int B = 128,  
        P = ceil(n/B);  
    saxpy<<<P,B>>>(n, a, x, y);  
}
```

```
__global__  
void saxpy(int n, float a,  
           float *x, float *y)  
{  
    int i = blockIdx.x * blockDim.x  
           + threadIdx.x;  
    if( i < n ) y[i] = a * x[i] + y[i];  
}
```

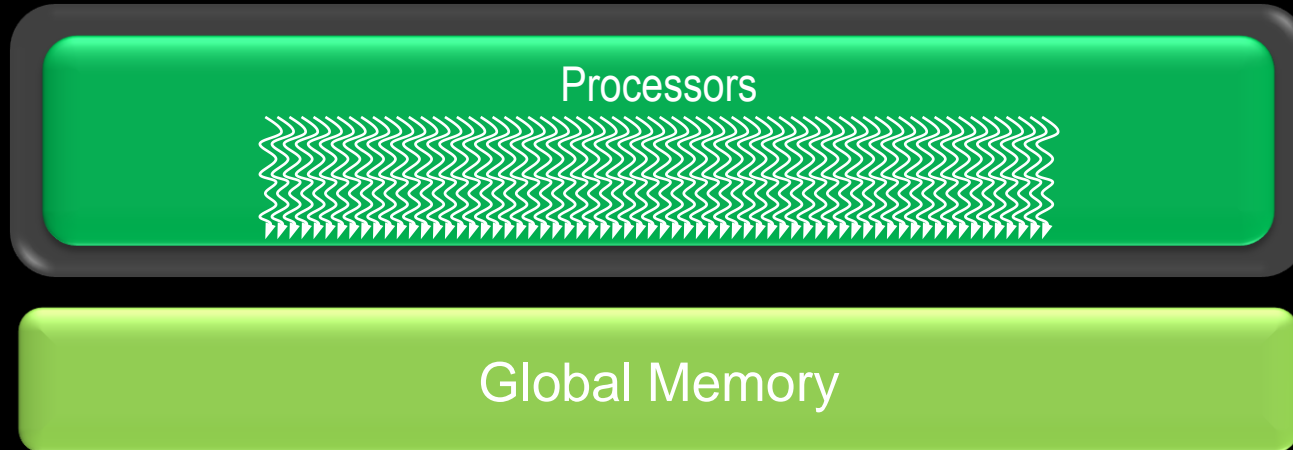


CUDA Model of Parallelism



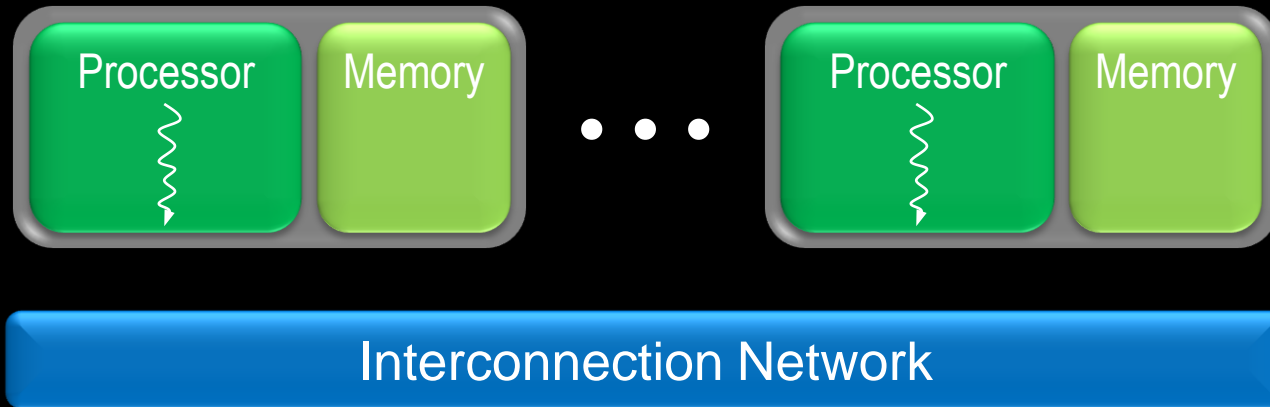
- **CUDA virtualizes the physical hardware**
 - thread is a virtualized scalar processor (registers, PC, state)
 - block is a virtualized multiprocessor (threads, shared mem.)
- **Scheduled onto physical hardware without pre-emption**
 - threads/blocks launch & run to completion
 - blocks should be independent

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. BSP (Bulk Synchronous Parallel) model, MPI**

Imperatives for Efficient Design



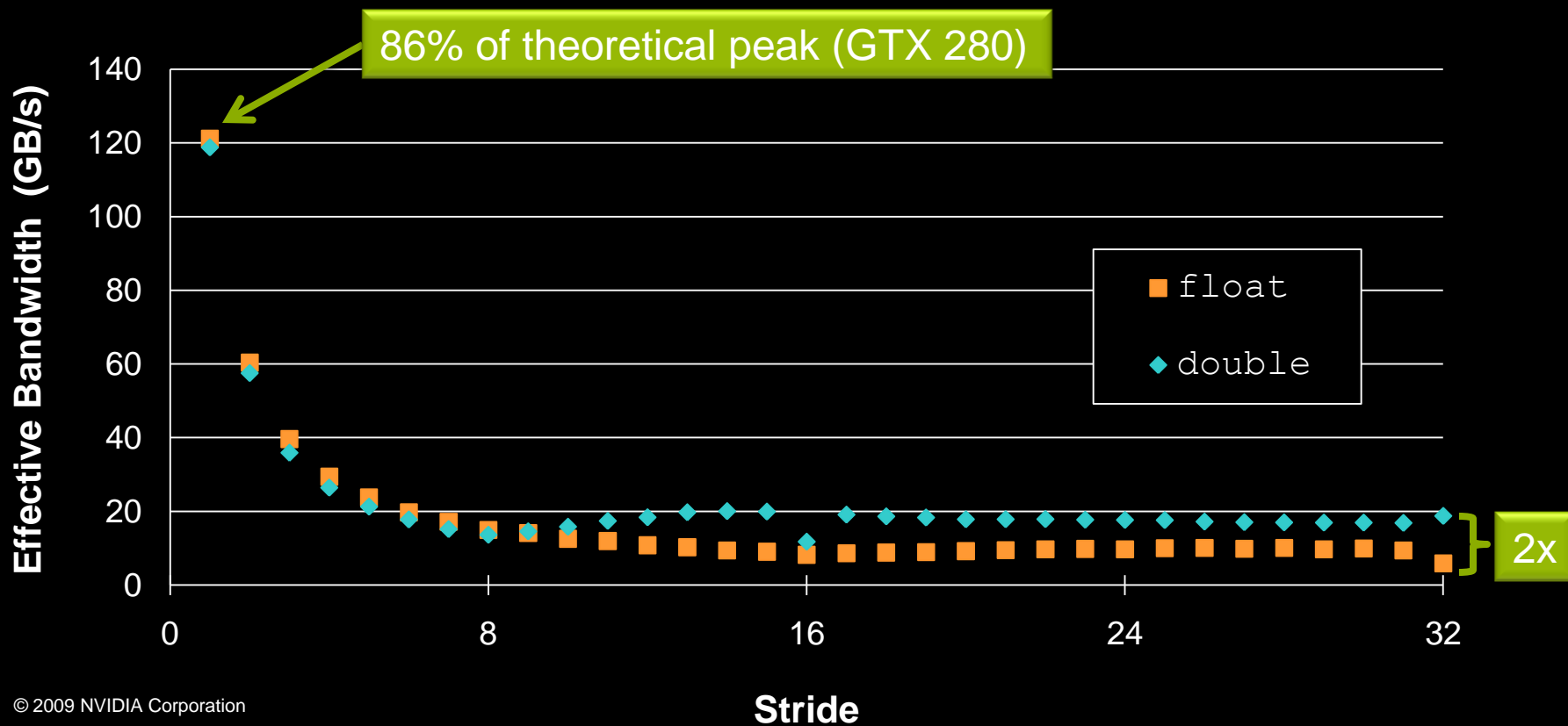
- **Expose abundant fine-grained parallelism**
 - need 1000's of threads for full utilization (30K max)
- **Maximize on-chip work**
 - on-chip memory orders of magnitude faster
- **Minimize execution divergence**
 - SIMT execution of threads in 32-thread warps
- **Minimize memory divergence**
 - coalesced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



REDUCTION

Parallel Reduction



- **Summing up a sequence with 1 thread:**

```
int sum = 0;  
for(int i=0; i<N; ++i)    sum += x[i];
```

- **Parallel reduction within 1 thread block:**

- each thread holds 1 element
- stepwise partial sums in a tree fashion
- B threads need $\log B$ steps

- **Parallel reduction of arbitrary arrays:**

- coordinate reductions within multiple blocks

Illustrating intra-block reduction



Input (shared memory)

10	1	8	-1	0	-2	3	5	-2	-3	2	7	0	11	0	2
----	---	---	----	---	----	---	---	----	----	---	---	---	----	---	---

0 1 2 3 4 5 6 7 active threads

$x[i] += x[i+8];$

8	-2	10	6	0	9	3	7	-2	-3	2	7	0	11	0	2
---	----	----	---	---	---	---	---	----	----	---	---	---	----	---	---

$x[i] += x[i+4];$

8	7	13	13	0	9	3	7	-2	-3	2	7	0	11	0	2
---	---	----	----	---	---	---	---	----	----	---	---	---	----	---	---

$x[i] += x[i+2];$

21	20	13	13	0	9	3	7	-2	-3	2	7	0	11	0	2
----	----	----	----	---	---	---	---	----	----	---	---	---	----	---	---

$x[i] += x[i+1];$

41	20	13	13	0	9	3	7	-2	-3	2	7	0	11	0	2
----	----	----	----	---	---	---	---	----	----	---	---	---	----	---	---

Final result

Reduction tree implementation



```
template<typename T>
__device__ T reduce(T *x)
{
    unsigned int i = threadIdx.x;
    unsigned int n = blockDim.x;

    for(unsigned int offset=n/2; offset>0; offset/=2)
    {
        if(tid < offset)
            x[i] += x[i + offset];
        __syncthreads();
    }

    return x[0]; // Note that only thread 0 has full sum
}
```

Generic reduction implementation



```
template<typename T, typename OP>
__device__ T reduce(T *x, OP op)
{
    unsigned int i = threadIdx.x;
    unsigned int n = blockDim.x;

    for(unsigned int offset=n/2; offset>0; offset/=2)
    {
        if(tid < offset)
            x[i] = op(x[i], x[i + offset]);
        __syncthreads();
    }

    return x[0]; // Note that only thread 0 has final result
}
```

must be commutative & associative

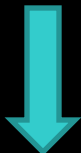
Coordinating blocks recursively



- **Divide N elements amongst $P=N/B$ blocks**



- **Reduce partial sums with $(N/B)/B$ blocks & repeat**



$\log_B N$ steps

Coordinating blocks directly



- **Assign N/P elements to each of P blocks**



- **Collect P sums & reduce with 1 block**



- **We're done in **2 steps**, but make sure P fills machine**

Directions for improvement



- **Pay attention to operator properties**
 - associative? commutative? has inverse? has identity?
 - the fewer that are true, the more careful the code must be
 - e.g., reduction examples assume commutative w/ identity
- **Many opportunities for code optimization**
 - remove inefficiencies
 - (auto-) tuning for different cases

Directions for optimization



- **Loop unrolling with fixed block sizes**
 - almost always a good idea
- **Minimize unutilized threads**
 - reduction example leaves half its threads idle
- **Vector loads can improve efficiency**
 - `float2, float4, ...`
- **Extra serial work per thread can improve efficiency**
 - cf. vector loads

Designing efficient sorting algorithms for manycore GPUs,
Nadathur Satish, Mark Harris, and Michael Garland.
IPDPS 2009.

MERGE SORT

Why sort?



- **To sort data**
- **To bring together all records with the same key**
 - cf. hashing
- **To build data structures**
 - CSR matrix, Bounding Volume Hierarchies, ...
- **Once you have a fast sort, many things are easier**

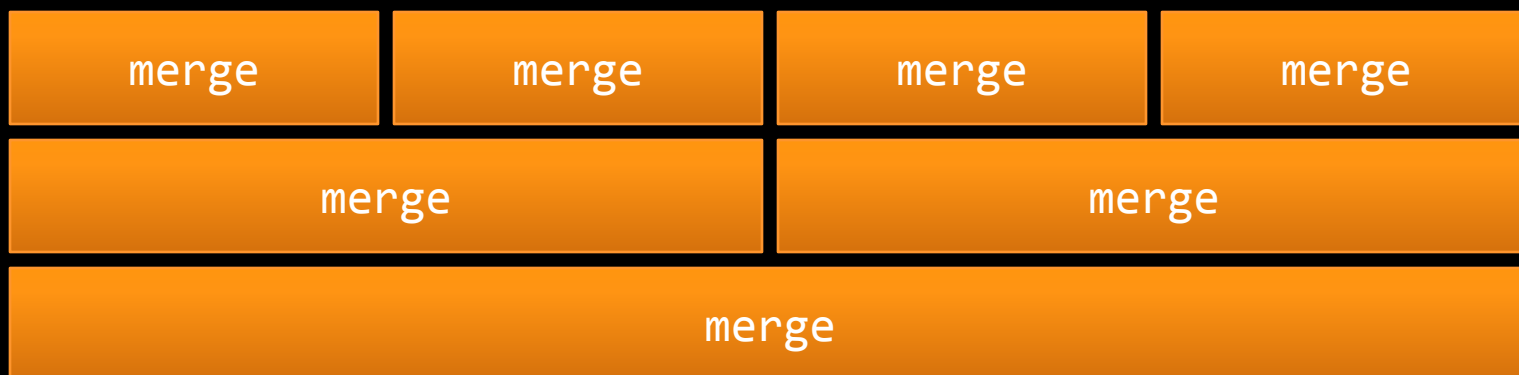
Merge Sort



- **Divide input array into 256-element tiles**
- **Sort each tile independently**



- **Produce sorted output with tree of merges**



Traditional sequential approach



- **Sorting each tile: take your pick**
 - quicksort, heap sort, insertion sort, ...
- **Merging two tiles: take next element one at a time**
 - `merge(A, B):`
 - `... if A or B is empty return the other ...`
 - `if A[0]<B[0]:`
 - `first, A = A[0], A[1:]`
 - `else:`
 - `first, B = B[0], B[1:]`
 - `return first + merge(A, B)`

Sorting a tile in parallel



- **Tiles are sized so that:**
 - a single thread block can sort them efficiently
 - they fit comfortably in on-chip memory
- **Sorting networks are most efficient in this regime**
 - we use **odd-even merge sort**
 - about 5-10% faster than comparable bitonic sort
- **Caveat: sorting networks may reorder equal keys**

Odd-Even Merge Sort

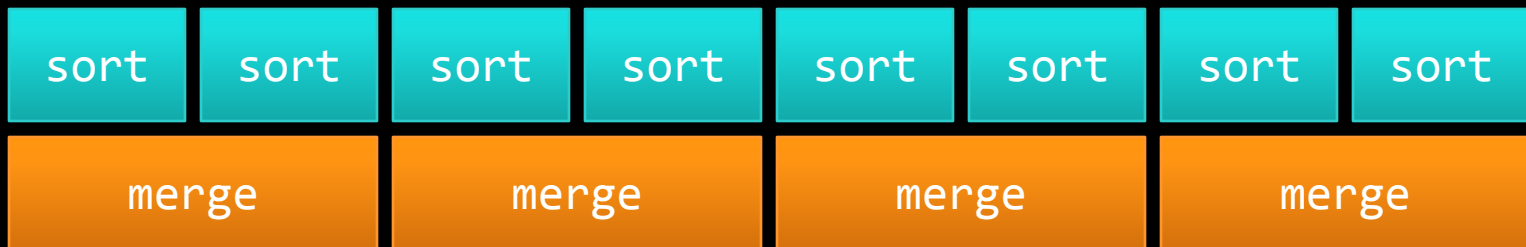
```
template<typename T, typename Cmp>
__device__ void oddeven_sort(T *keys, int i, int n, Cmp lt)
{
    for(unsigned int p=n/2; p>0; p/=2) {
        unsigned int q=n/2, r=0, d=p;
        while( q>=p ) {
            if( i<(n-d) && (i&p)==r ) {
                unsigned int j = i+d;
                T xi = keys[i], xj = keys[j];

                if( lt(xj,xi) ) {
                    keys[i] = xj;
                    keys[j] = xi;
                }
            }

            d = q-p; q = q/2; r = p;
            __syncthreads();
        }
    }
}
```

Algorithm M, Section 5.2.2
The Art of Computer Programming, Vol 3
D. E. Knuth

Merging pairs of sorted tiles



- Launch 1 thread block to process each pair of tiles
- Load tiles into on-chip memory
- Perform **counting merge**
- Store merged result to global memory

Counting Merge



$\text{upper_bound}(A[i], B) = \text{count}(j \text{ where } A[i] \leq B[j])$



$\text{lower_bound}(B[j], A) = \text{count}(i \text{ where } B[j] < A[i])$

Use binary search since A & B are sorted

Counting Merge



$\text{upper_bound}(A[i], B) = \text{count}(j \text{ where } A[i] \leq B[j])$

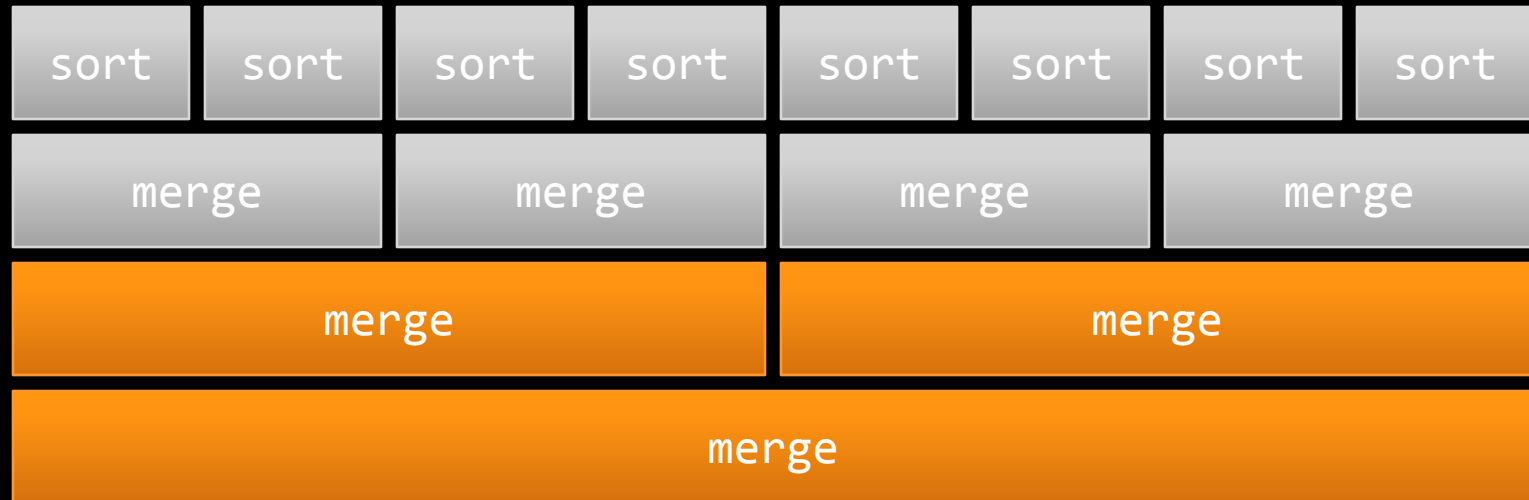


$\text{lower_bound}(B[j], A) = \text{count}(i \text{ where } B[j] < A[i])$

$\text{scatter}(A[i] \rightarrow C[i + \text{upper_bound}(A[i], B)])$

$\text{scatter}(B[j] \rightarrow C[\text{lower_bound}(B[j], A) + j])$

Merging Larger Subsequences



- Partition larger sequences into collections of tiles
- Apply counting merge to each pair of tiles

Two-way Partitioning Merge

- Pick a splitting element from either A or B



- Divide A and B into elements below/above splitter



found by binary search

- Recurse



Multi-way Partitioning Merge



- **Pick every 256th element of A & B as splitter**



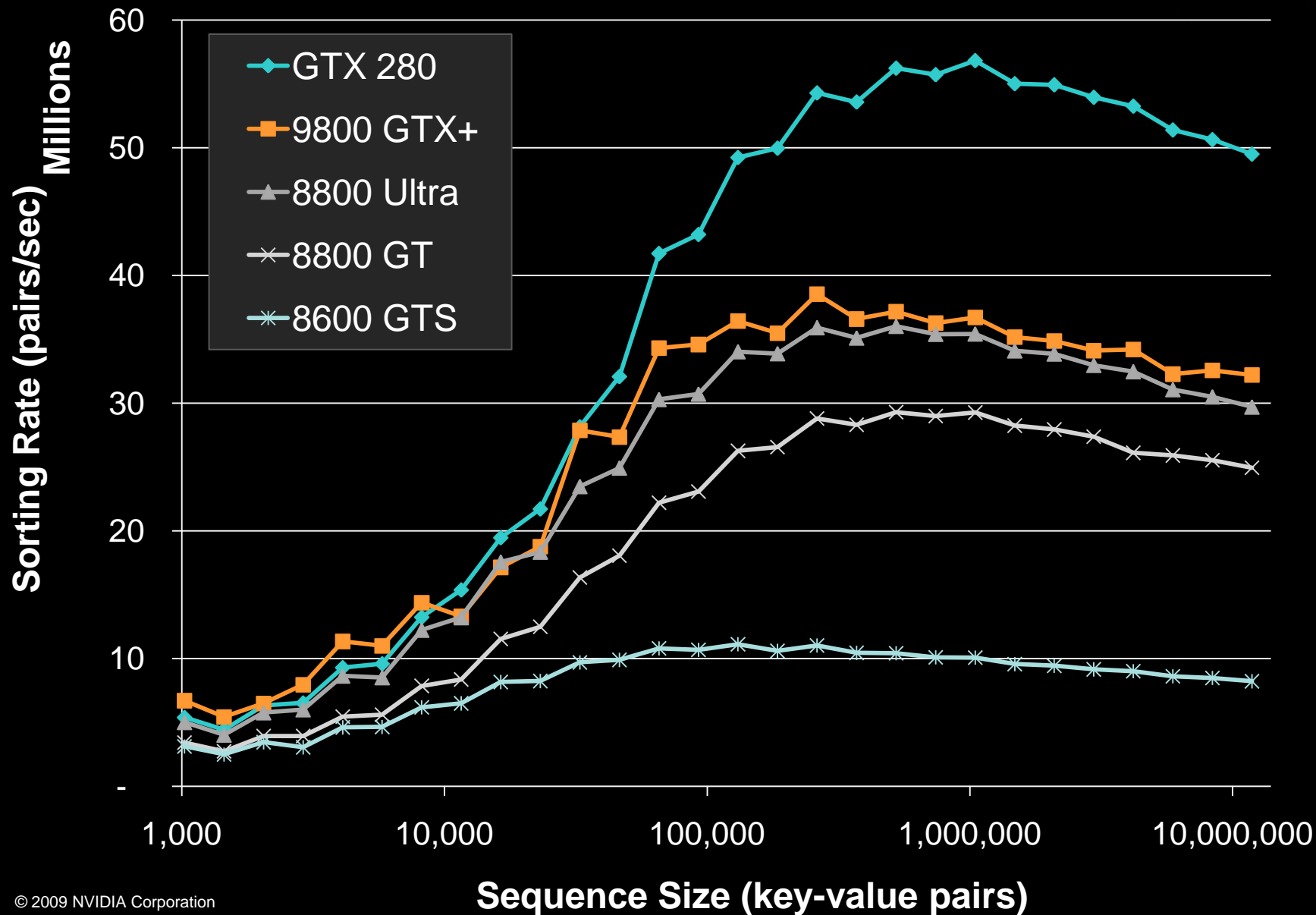
- **Apply merge recursively to merge splitter sets**
 - recursively apply merge procedure

- **Split A & B with merged splitters**



- **Merge resulting pairs of tiles (at most 256 elements)**

Merge Sorting Rate



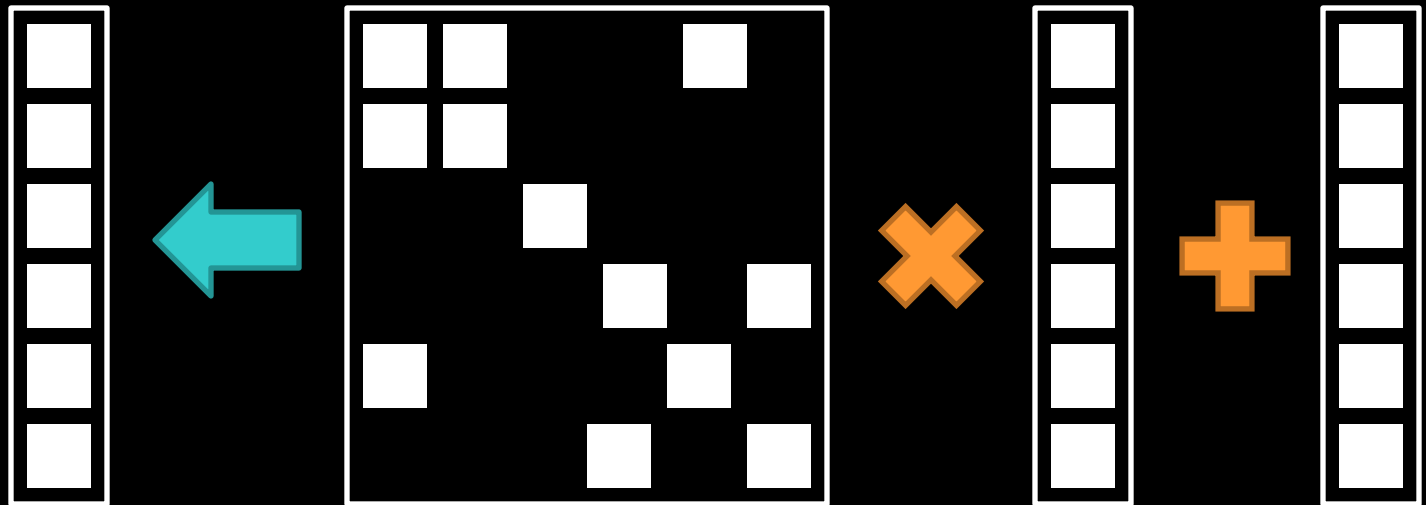
Implementing sparse matrix-vector multiplication on throughput-oriented processors,
Nathan Bell and Michael Garland.
Supercomputing '09

SPARSE MATRICES

Sparse matrix-vector multiplication



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

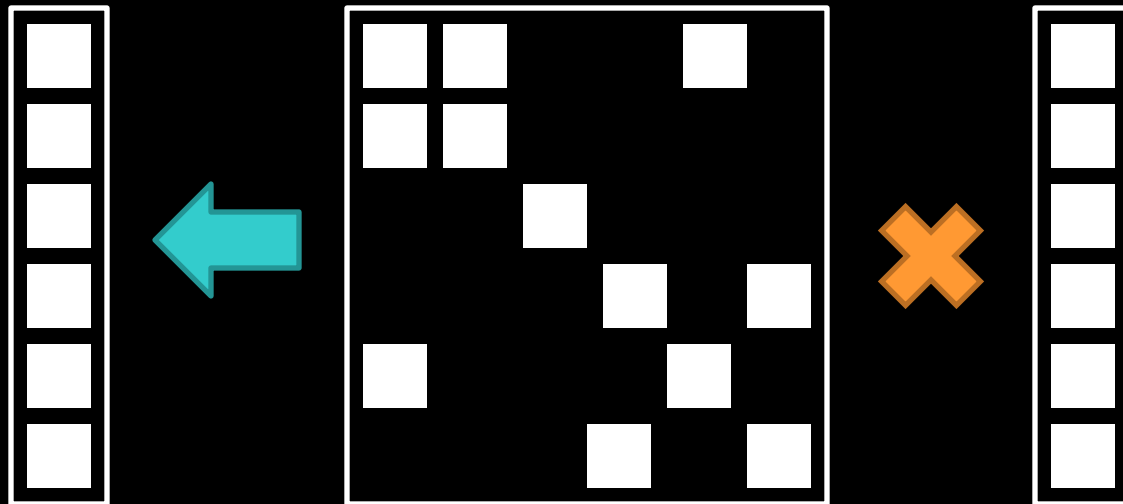


- **Unlike dense methods, SpMV is generally**
 - unstructured / irregular
 - entirely bound by memory bandwidth

Application



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

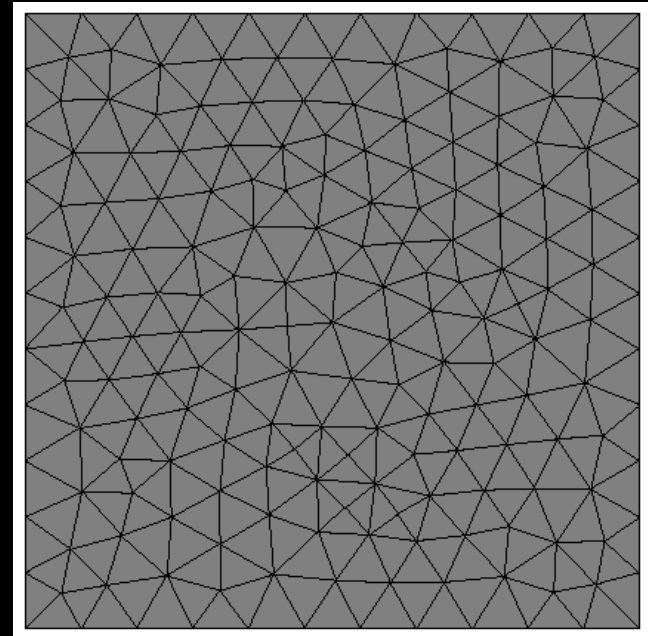
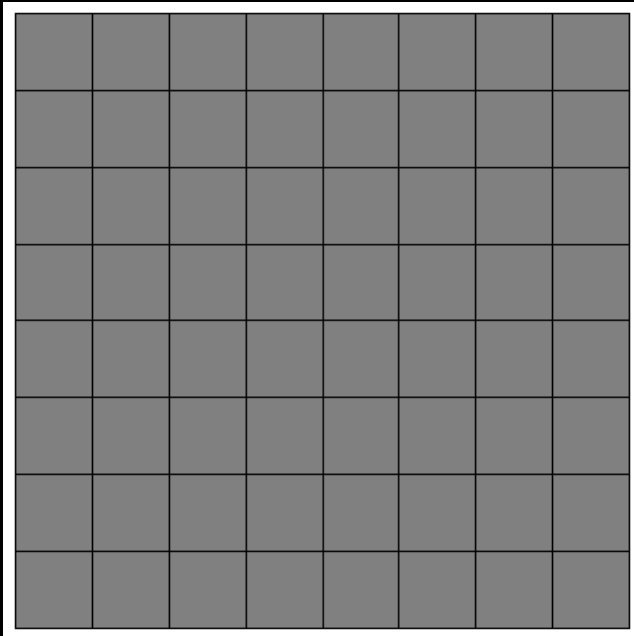


Application



- **Finite-Element Methods**

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



Compressed Sparse Row (CSR)



$$\begin{pmatrix} 3 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 2 & 4 & 1 \\ 1 & 0 & 0 & 1 \end{pmatrix}$$

			Row 0	Row 2	Row 3
Nonzero values	<code>data[7]</code>	= {	3, 1,	2, 4, 1,	1, 1 };
Column indices	<code>indices[7]</code>	= {	0, 2,	1, 2, 3,	0, 3 };
Row pointers	<code>ptr[5]</code>	= {	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	<code>data[7]</code>	= { 3, 1,	2, 4, 1,	1, 1 };
Column indices	<code>indices[7]</code>	= { 0, 2,	1, 2, 3,	0, 3 };
Row pointers	<code>ptr[5]</code>	= { 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
<i>Nonzero values</i>	<code>data[7]</code>	= { 3, 1,	2, 4, 1,	1, 1 };
<i>Column indices</i>	<code>indices[7]</code>	= { 0, 2,	1, 2, 3,	0, 3 };
<i>Row pointers</i>	<code>ptr[5]</code>	= { 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 csrmul_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 csrmul_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	<code>data[7]</code>	= {	3	1	2	4	1	1	1	}
Column indices	<code>indices[7]</code>	= {	0	2	1	2	3	0	3	}
Row pointers	<code>ptr[5]</code>	= {	0	2	2	5	7			}

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

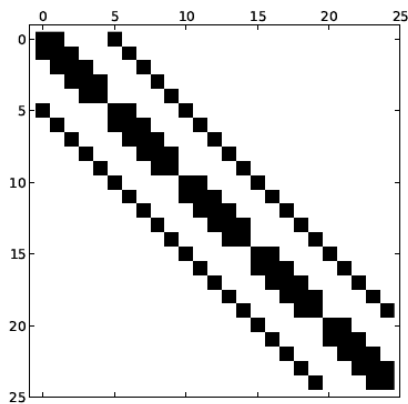
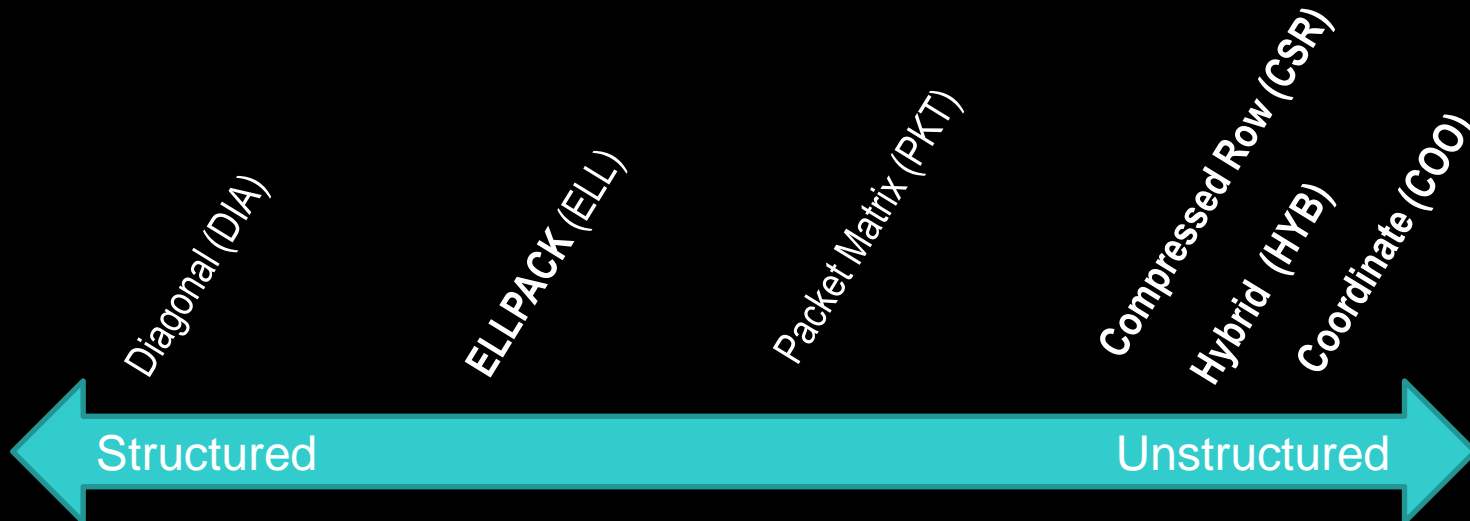
Nonzero values `data[7] = { 3, 1, 2, 4, 1, 1, 1 };`

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

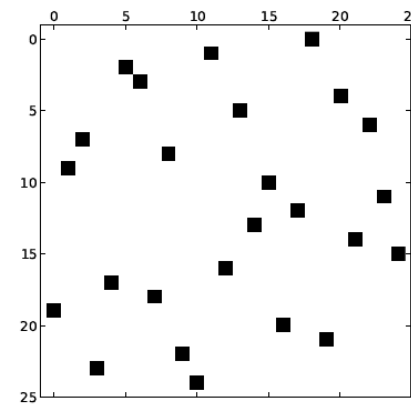
Row indices `rows[7] = { 0, 0, 1, 1, 1, 2, 2 };`

- 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

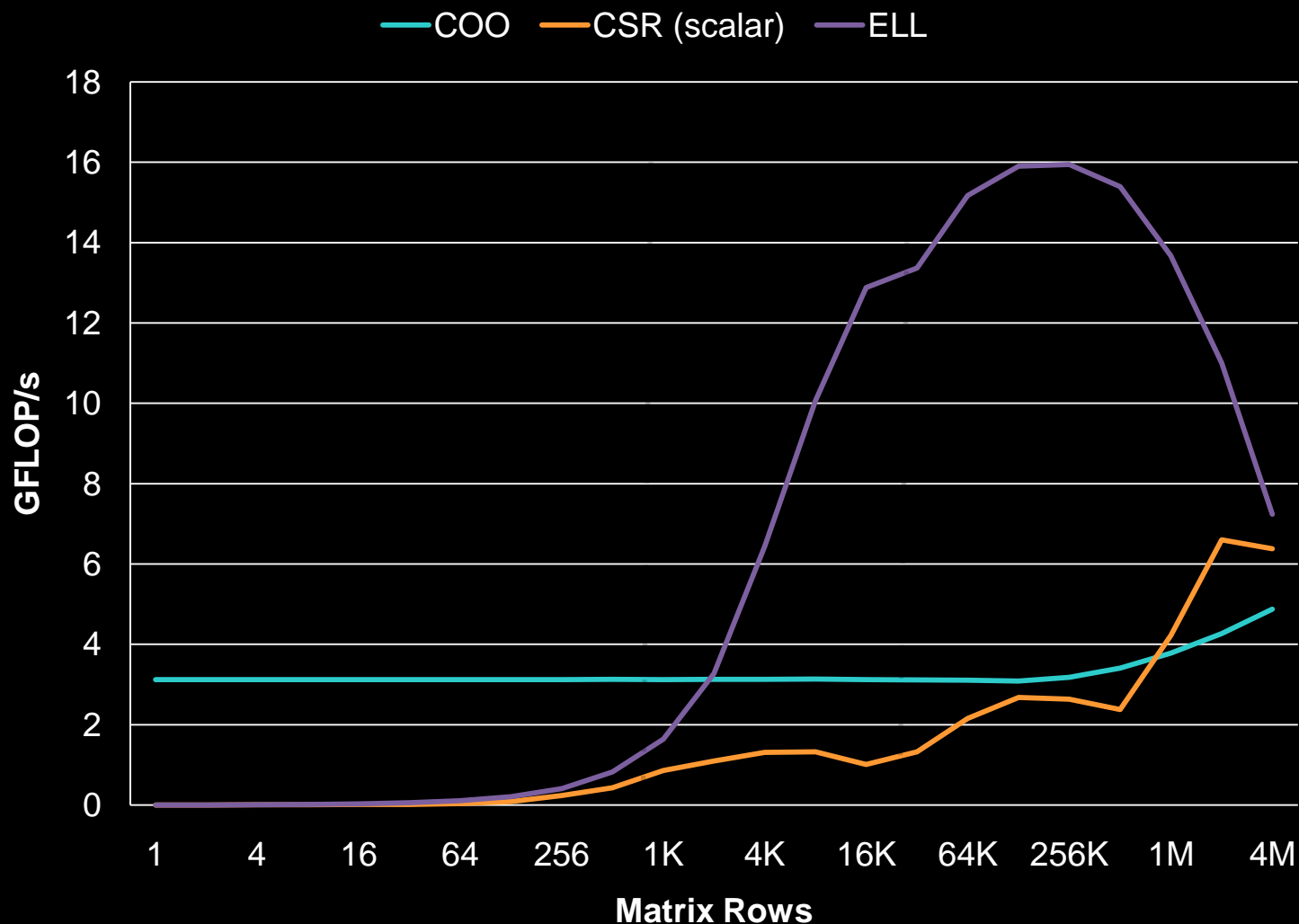
Tradeoffs: Matrix Representations



Format	Threads	Coalescing
CSR	per row	rare
ELL	per row	full
COO	per entry	full
HYB	ELL+COO	full



Granularity effects with 4M nonzeros



Multicore comparison

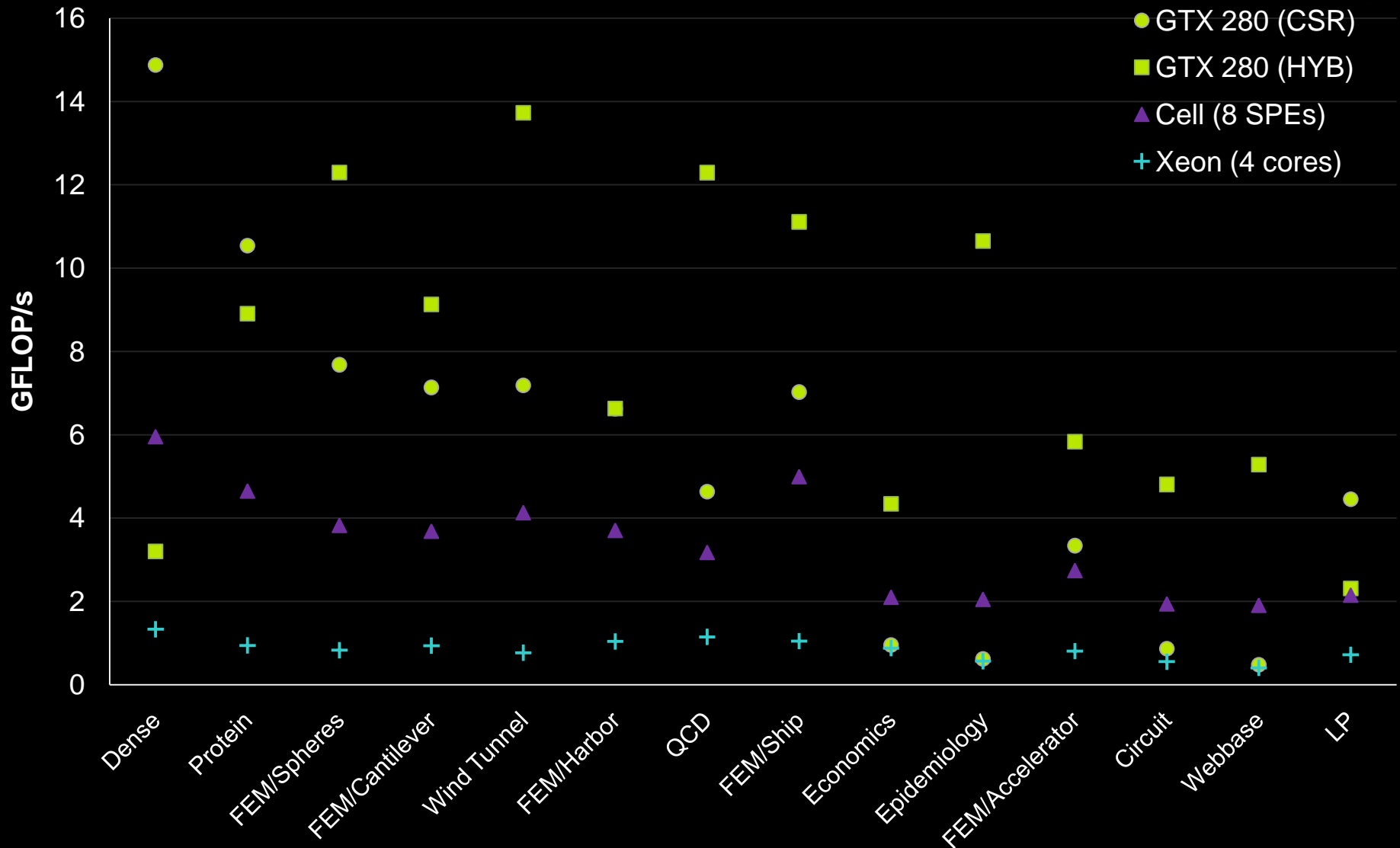


Name	Sockets	Cores	Clock (GHz)	Notes
Cell	1	8 (SPEs)	3.2	IBM QS20 Blade (half)
Xeon	1	4	2.3	Intel Clovertown
Dual Cell	2	16 (SPEs)	3.2	IBM QS20 Blade (full)
Dual Xeon	2	8	2.3	2x Intel Clovertown

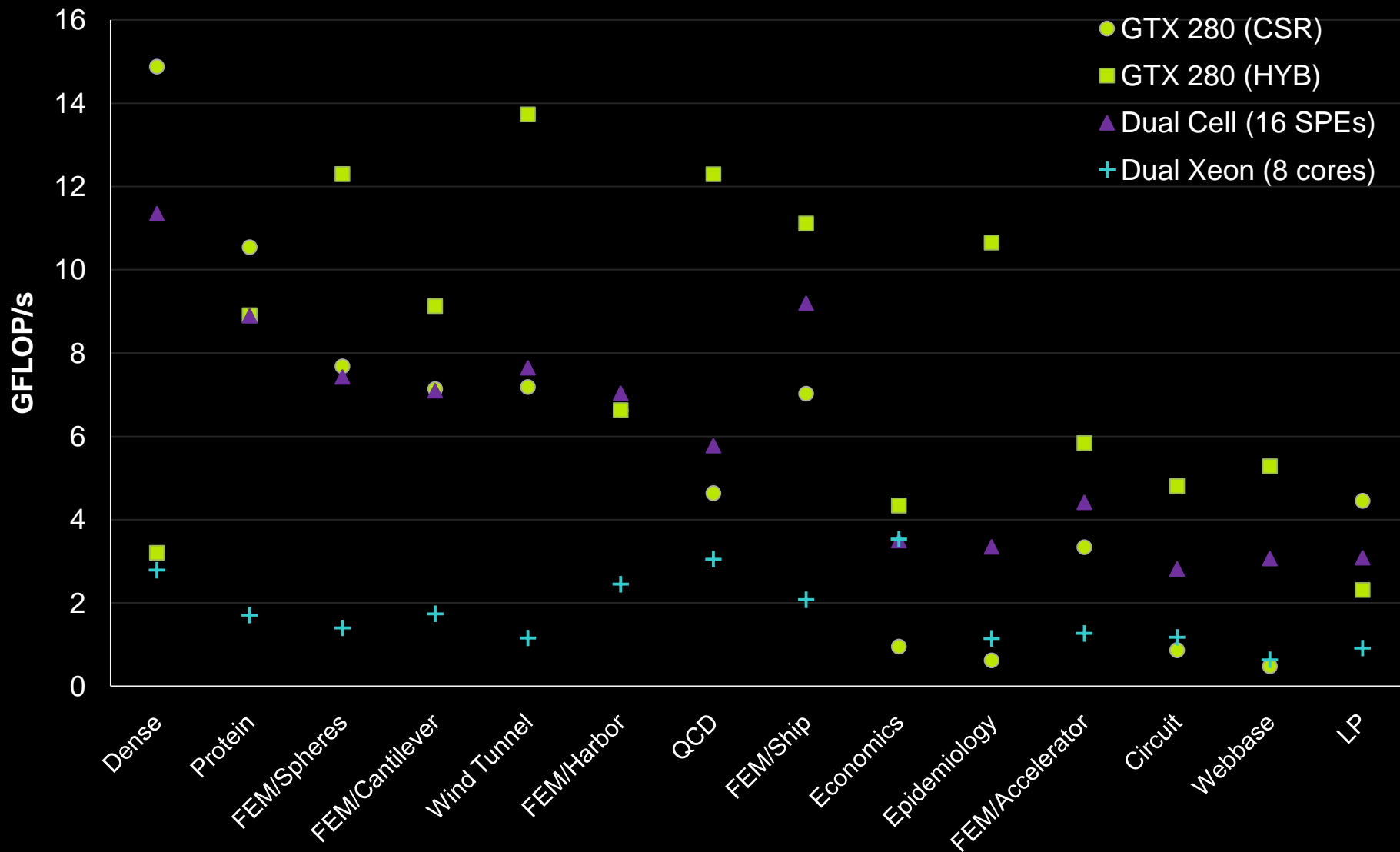
Source:

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

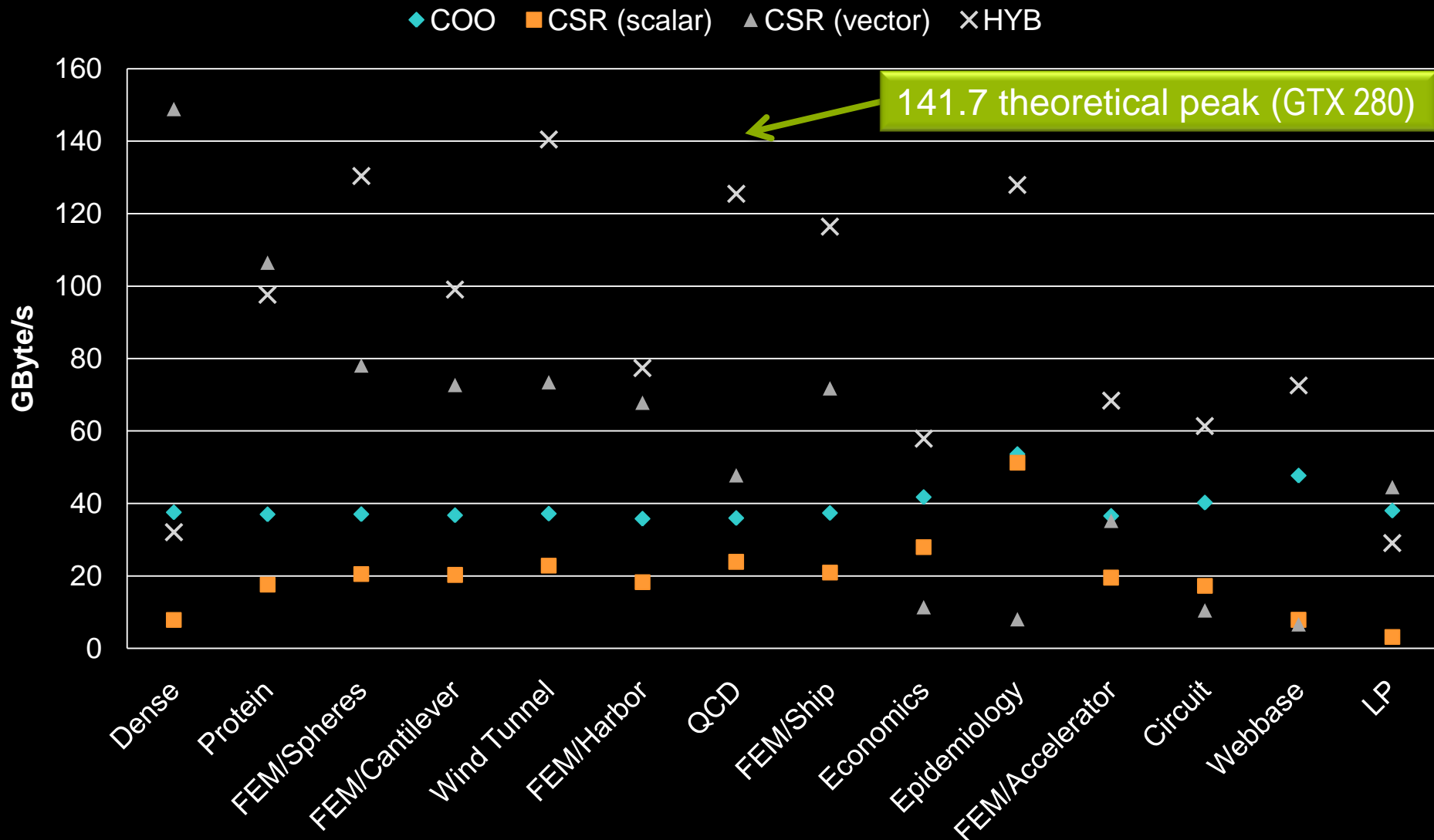
Sparse Matrix-Vector Multiplication (FP64)



Sparse Matrix-Vector Multiplication (FP64)



Effective bandwidth (FP64)



Questions?

mgarland@nvidia.com

<http://www.nvidia.com/research>