PASI Summer School

Advanced Algorithmic Techniques for GPUs

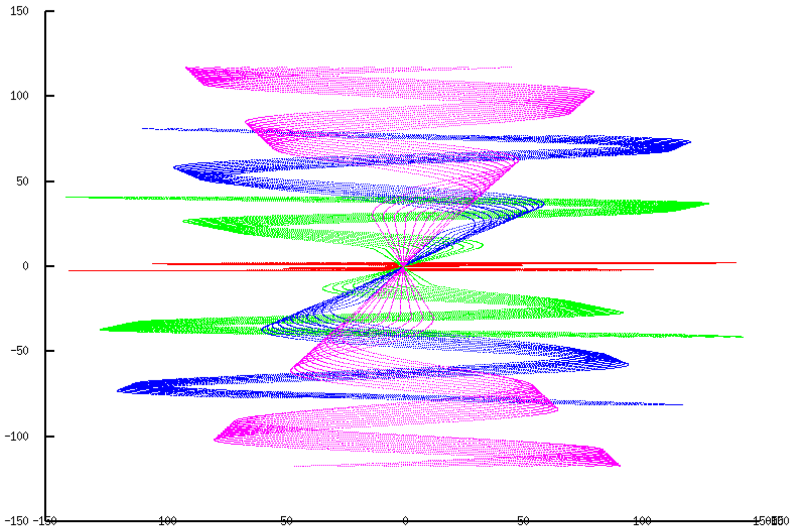
Lecture 6: Input Compaction and Further Studies

Objective

- To learn the key techniques for compacting input data for reduced consumption of memory bandwidth
  - Via better utilization of on-chip memory
  - As well as fewer bytes transferred to on-chip memory
- To understand the tradeoffs between input compaction and input binning/regularization

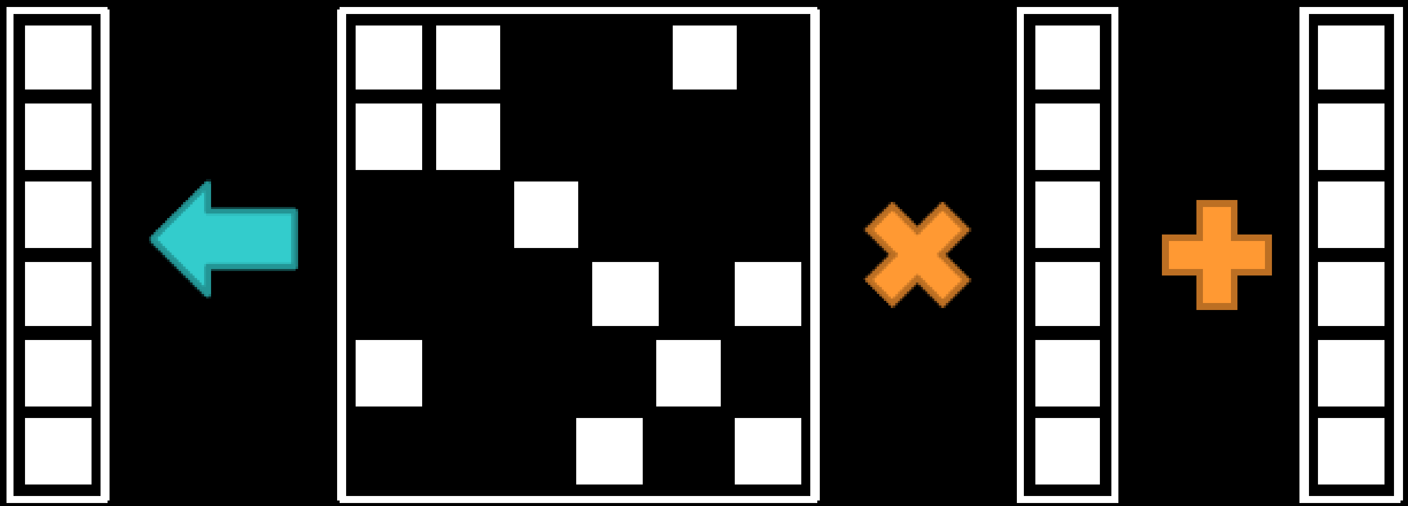
Sparse Data
Motivation for Compaction

- Many real-world inputs are sparse/non-uniform
- Signal samples, mesh models, transportation networks, communication networks, etc.



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

Parallelizing CSR SpMV

Compressed Sparse Row

- **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 };`			

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

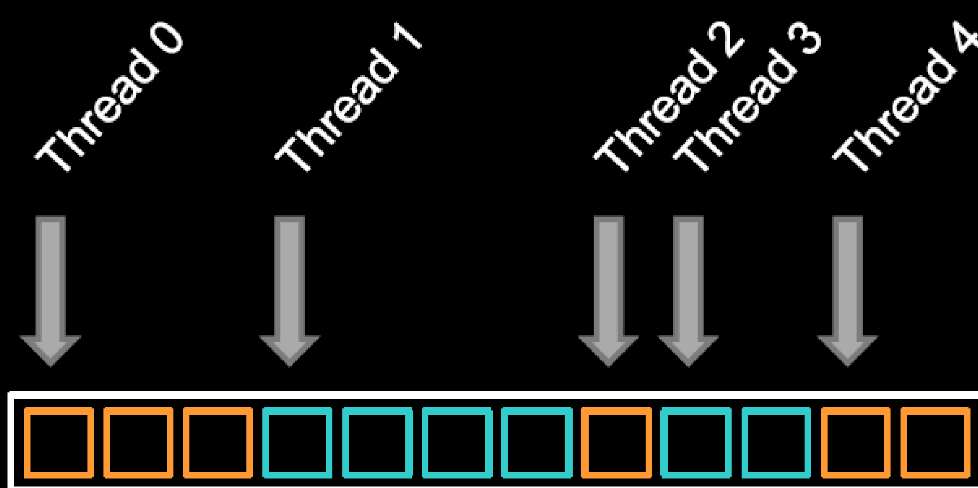
- **Memory divergence**
  - **minimal coalescing**

		#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

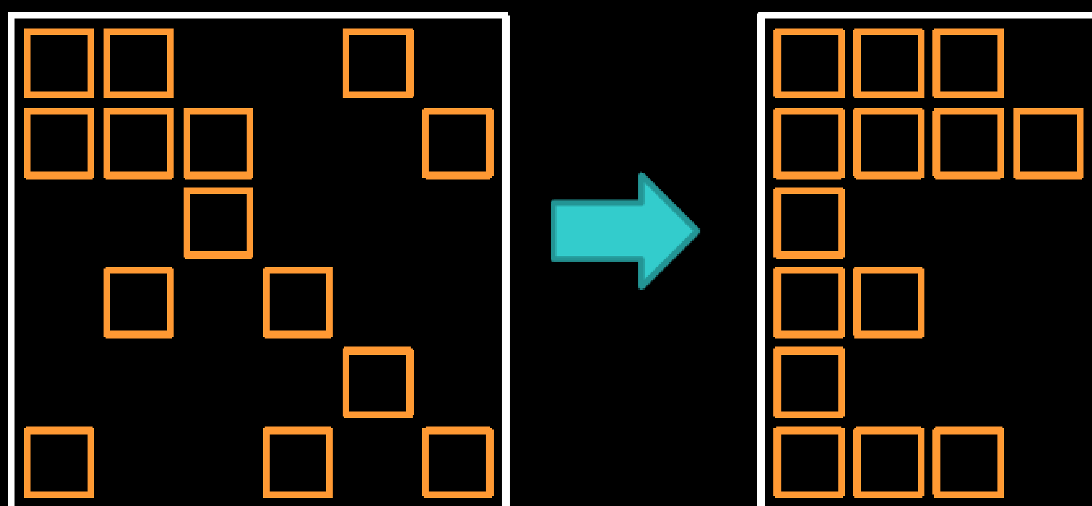
- Memory divergence
  - minimal coalescing



Regularizing SpMV with ELL format

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

ELLPACK



Regularizing SpMV with ELL format

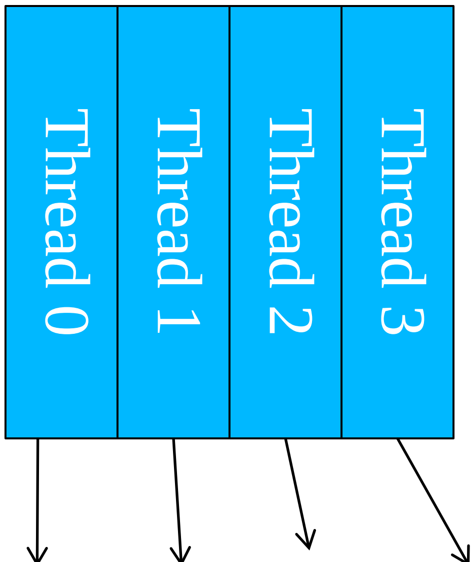
- **Quantize each row to a fix length K**

	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**

Memory Coalescing with ELL

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



data	3	*	2	1	1	*	4	1	*	*	1
index	0	*	1	1	2	*	2	3	*	*	3

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 };`

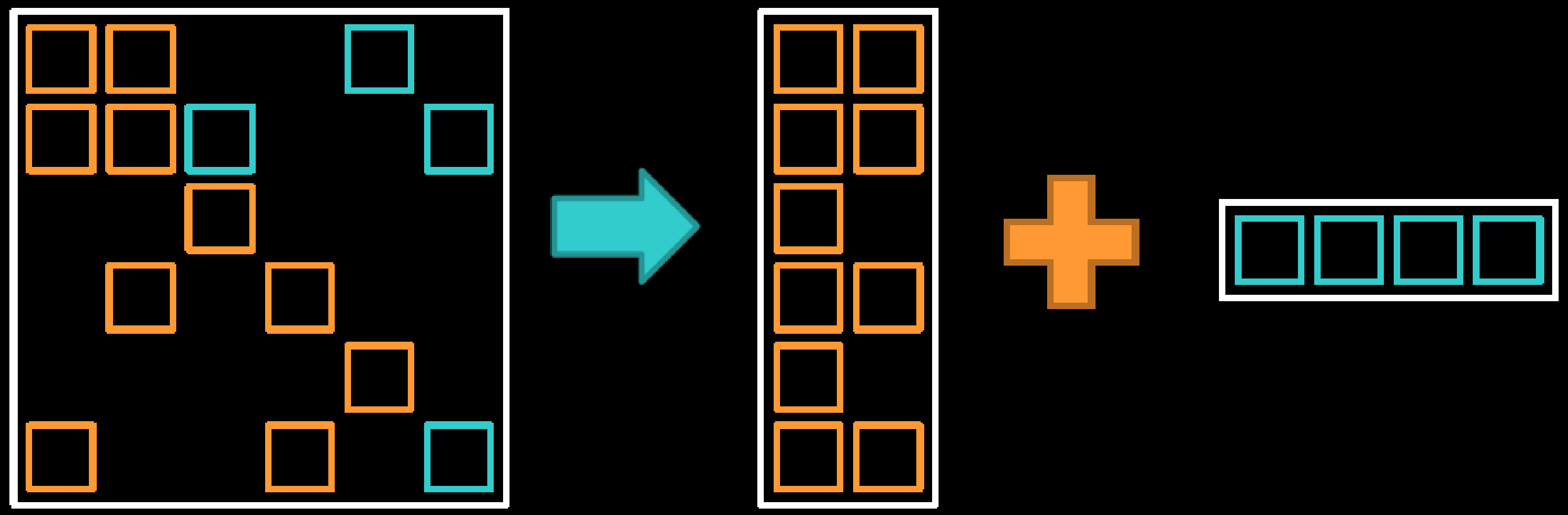
Column indices `cols[7] = { 0, 2, 1, 2, 3, 0, 3 };`

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

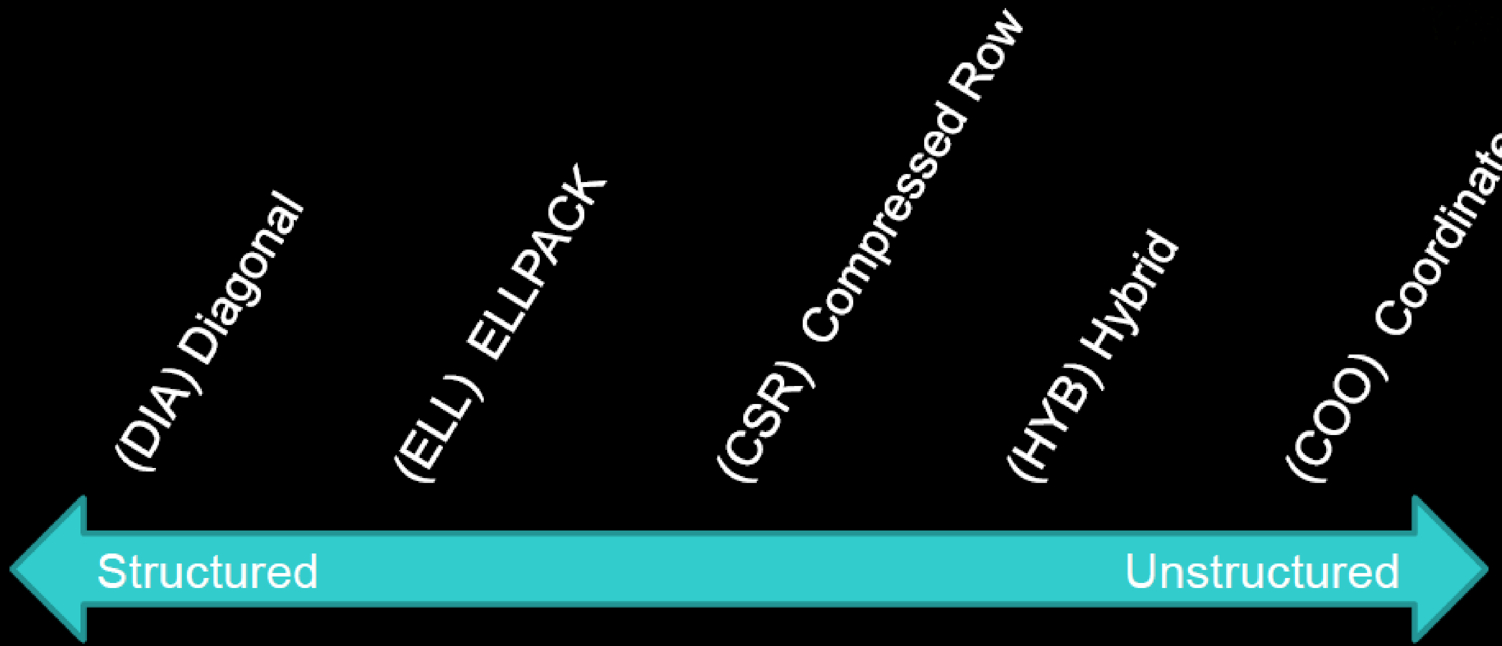


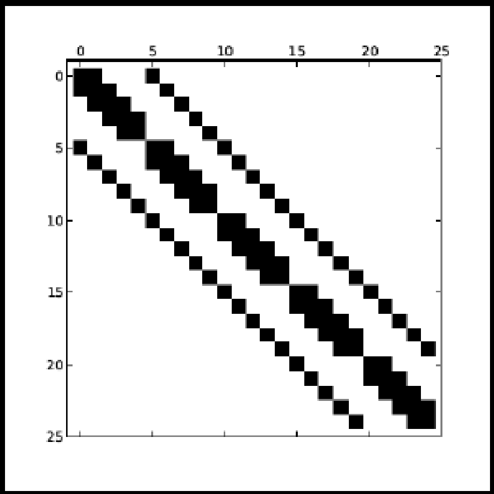
Any More Ideas?

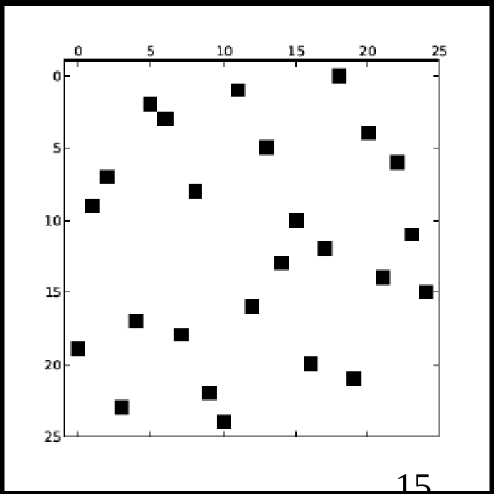
- JDS format
  – Sort rows according to their number of non-zero elements

- Can use Hybrid with JDS and and launch multiple kernels

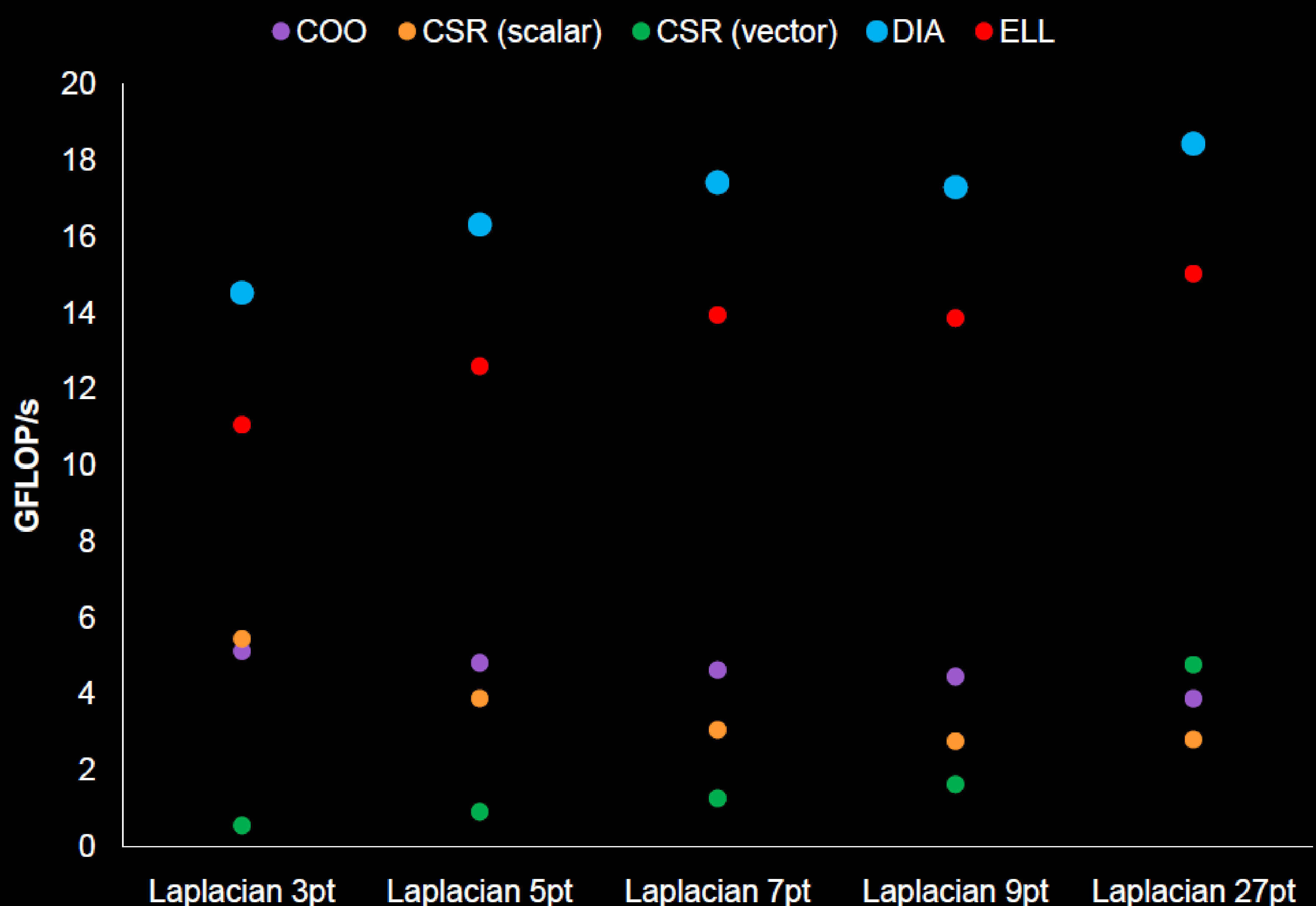
Sparse formats for different matrices



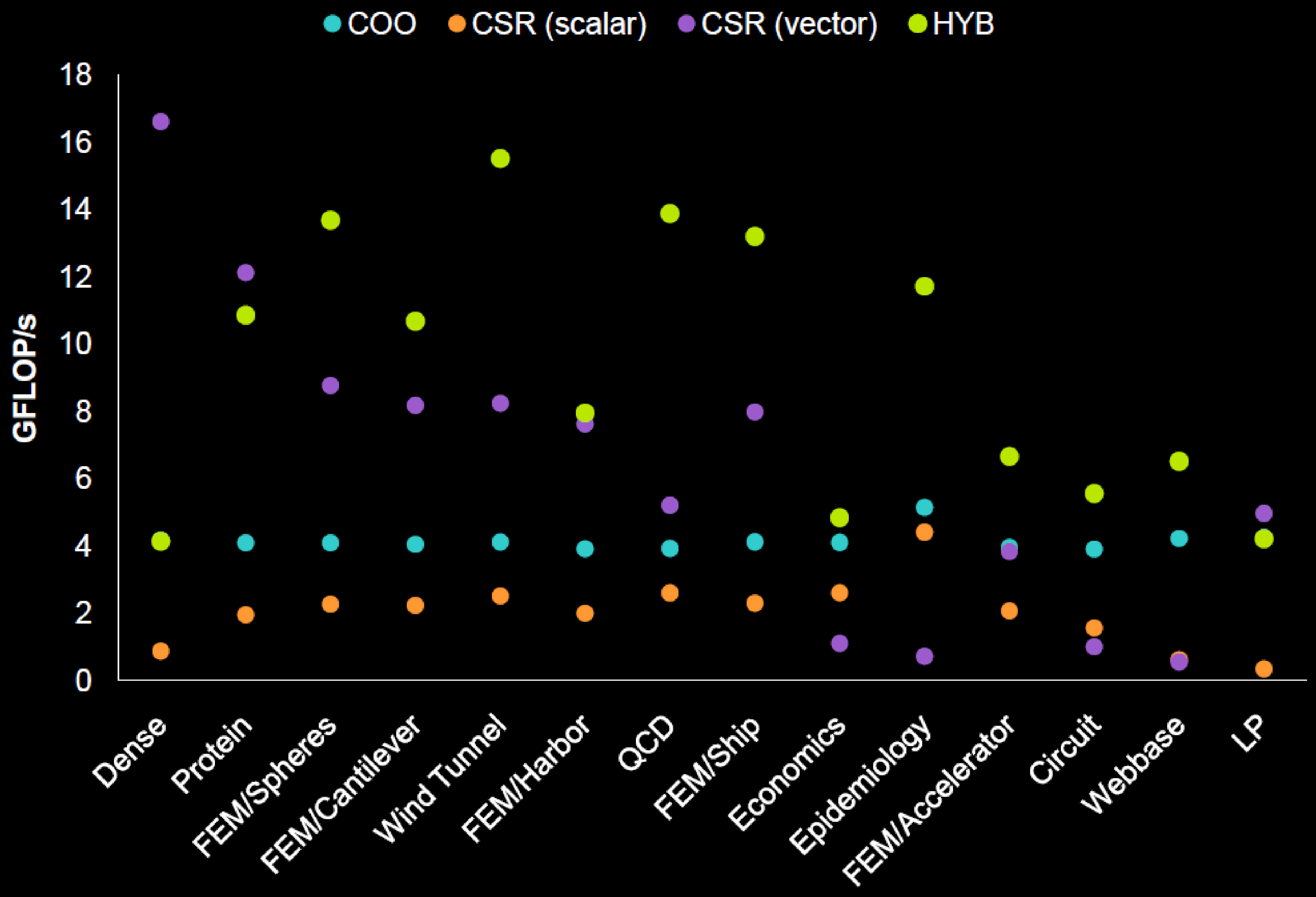




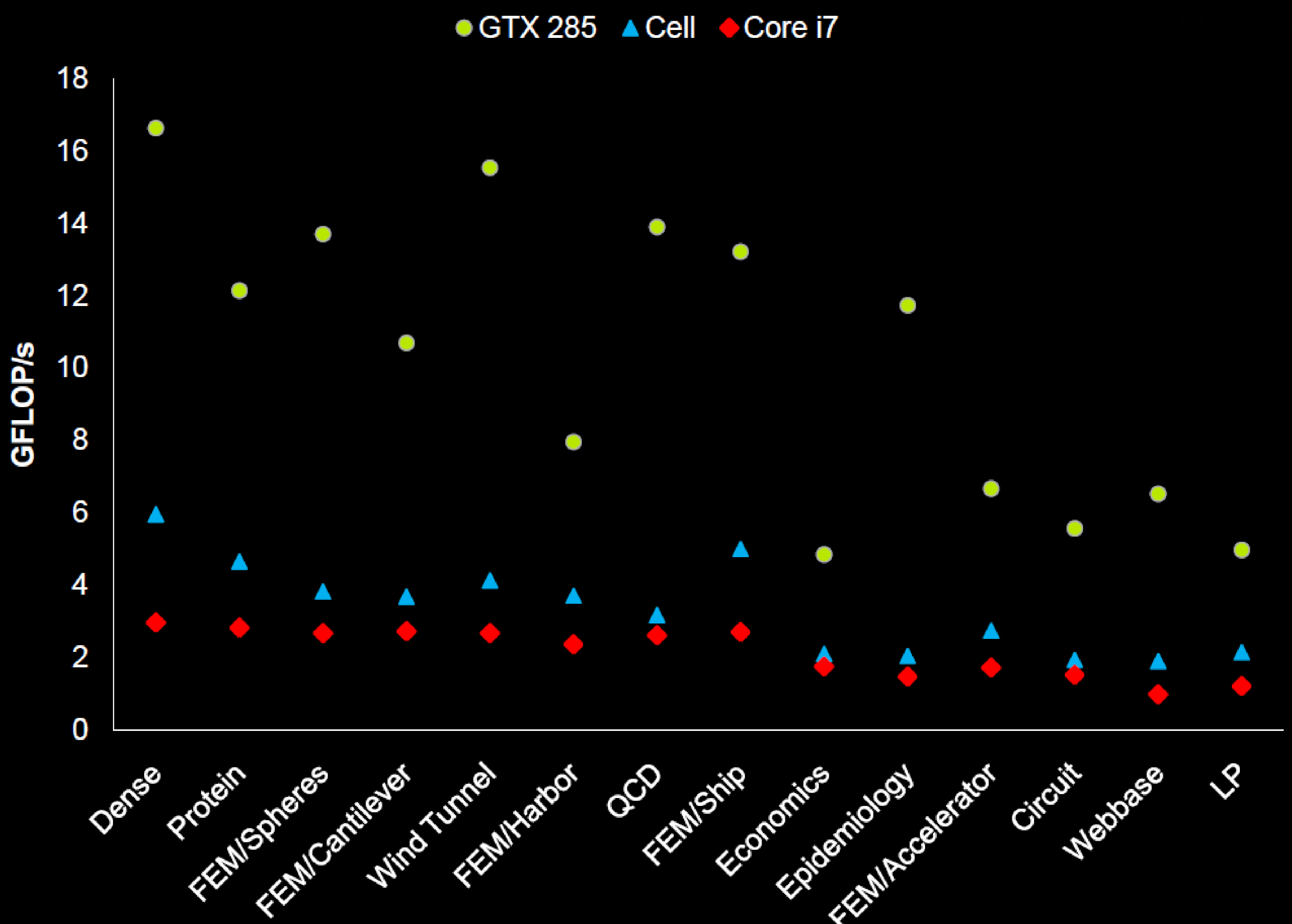
Structured Matrices



Unstructured Matrices

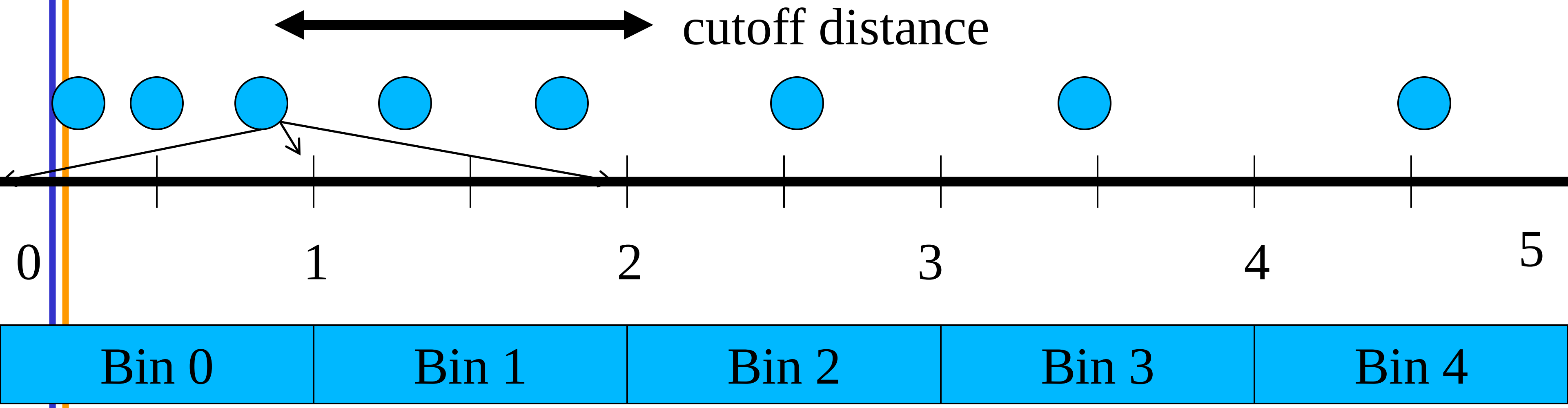


Performance Comparison

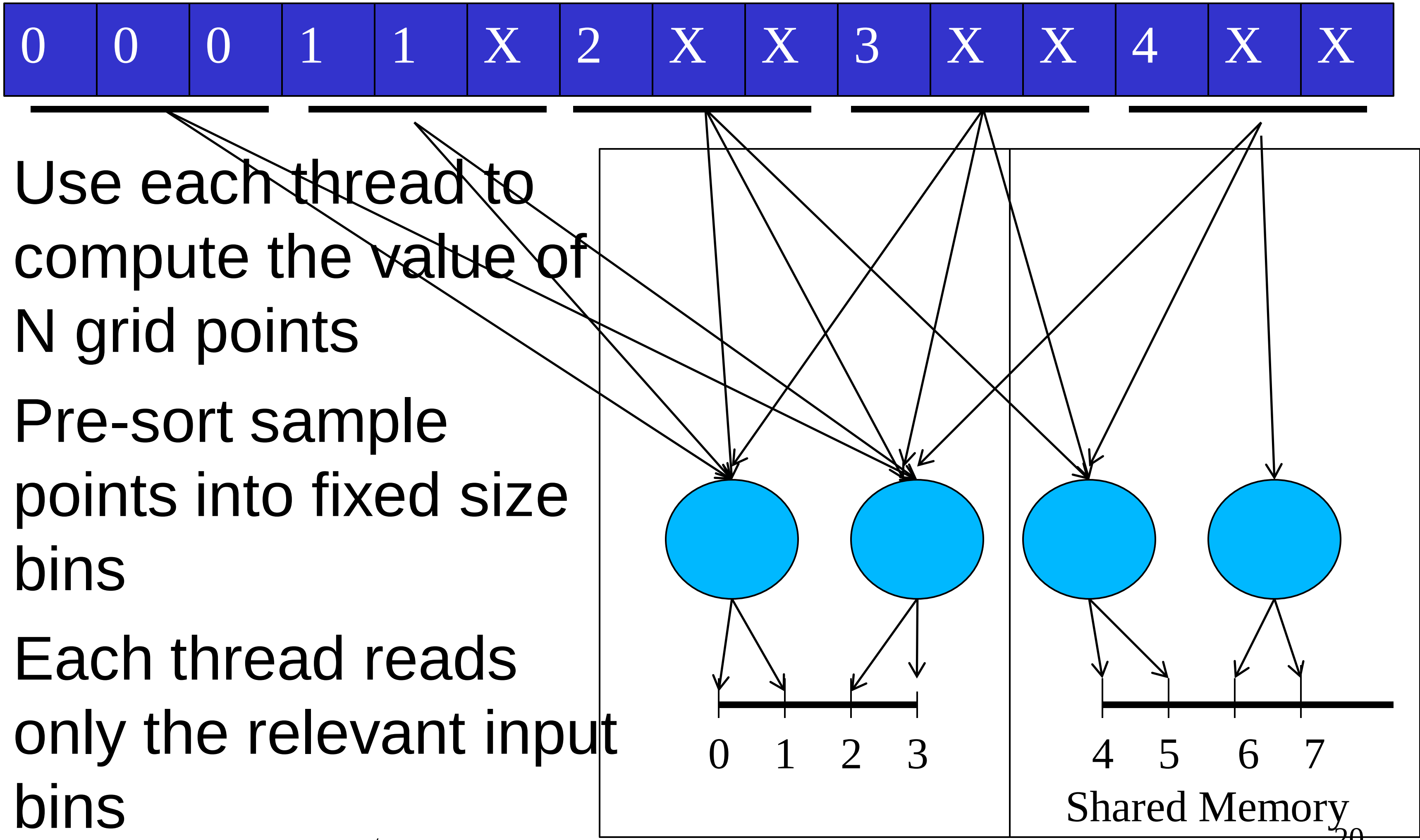


Binning of Sample Points

- For simplicity, we will use 1D gridding examples
- Each sample point has
  - S.x (will be represented with Bin#)
  - S.value (will be omitted unless necessary)

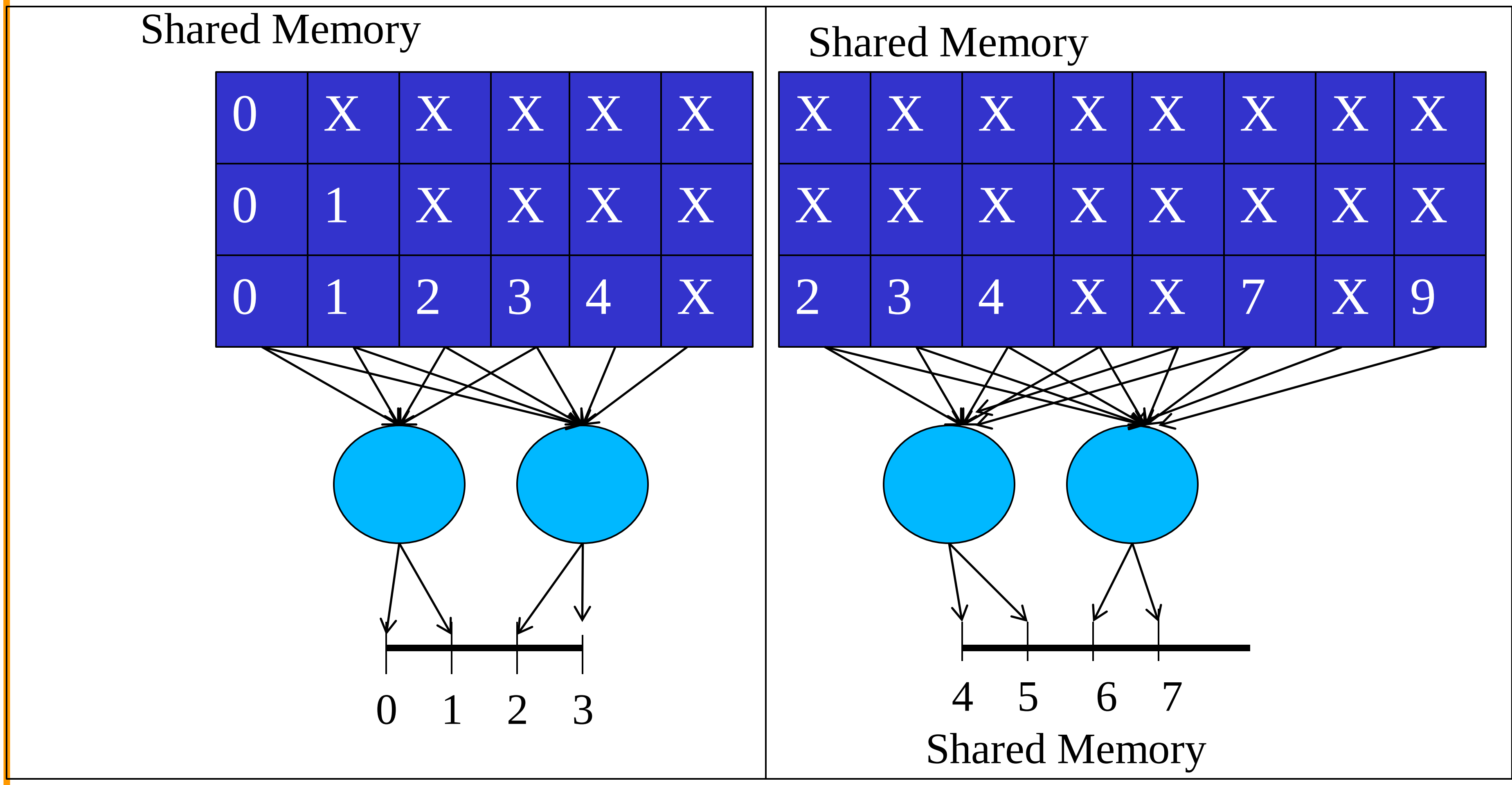


A Binned Gather Parallelization



- Use each thread to compute the value of N grid points
- Pre-sort sample points into fixed size bins
- Each thread reads only the relevant input bins

A Tiled Gather Implementation

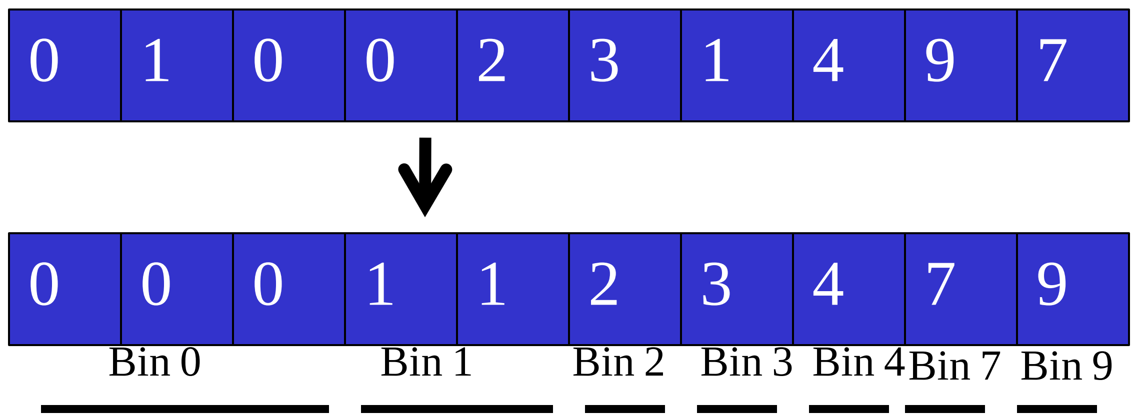


More on Tiled Gather

- Threads cooperate to load all the relevant bins from Global Memory to Shared Memory
- Each thread accesses relevant bins from Shared Memory
- Uniform binning for Non-uniform distribution
  - Large memory overhead for dummy cells
  - Reduced benefit of tiling
  - Many threads spend much time on dummy sample points

Compact Binning for Gather Parallelization

- Avoid pre-allocated fixed capacity bins (multi-dimensional array)
- Sort samples into bins of varying sizes in input array instead
  - Bins 5, 6, 8 are implicit, zero-sample



GPU Binning - Use Scatter to Generate Bin Capacities

0	1	0	0	2	3	1	4	9	7

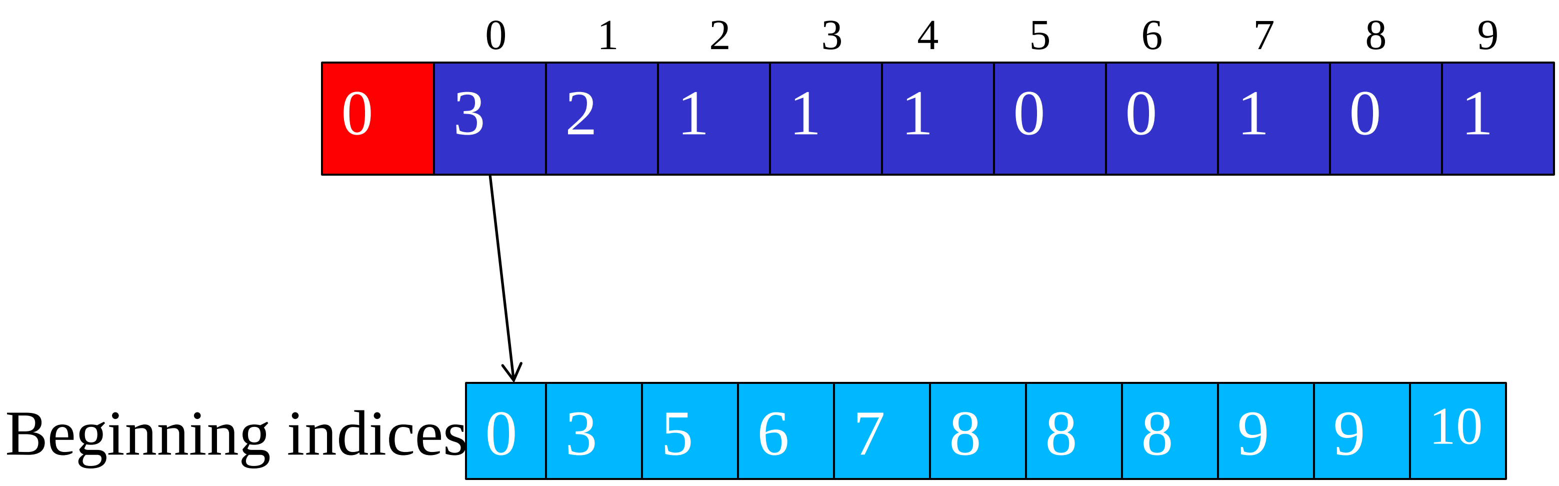
Capacity of Each bin

3	2	1	1	1	0	0	1	0	1
Bin 0	Bin 1	Bin 2	Bin 3	Bin 4			Bin 7		Bin 9

Need to use atomic operations for counting the capacity

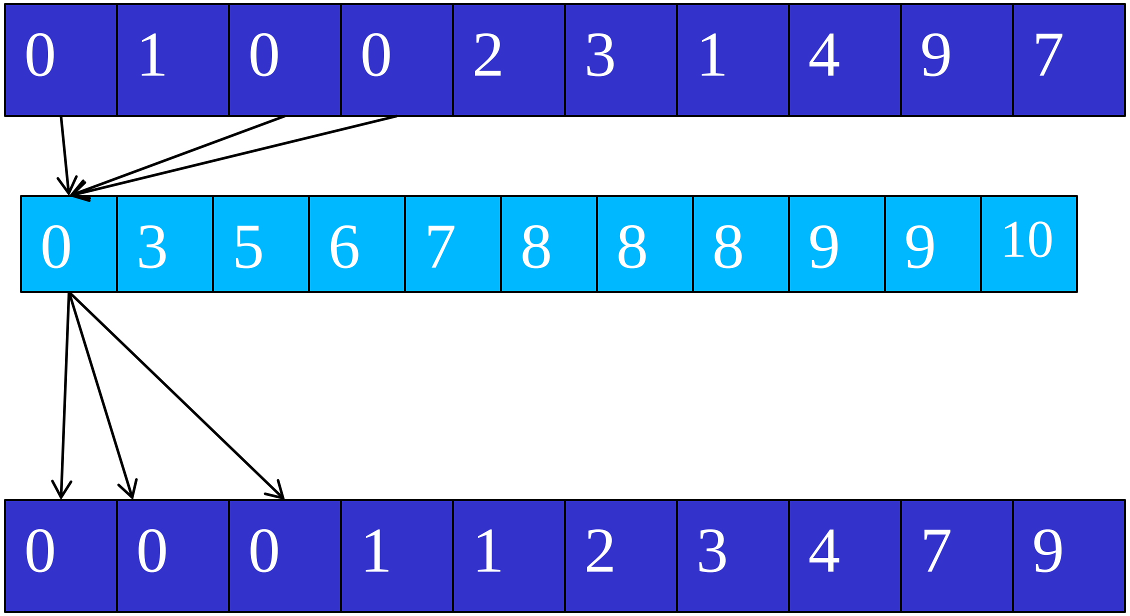
Determine Start and End of Bins

- Use parallel scan operations on the bin capacity array to generate an array of starting points of all bins (CUDPP)

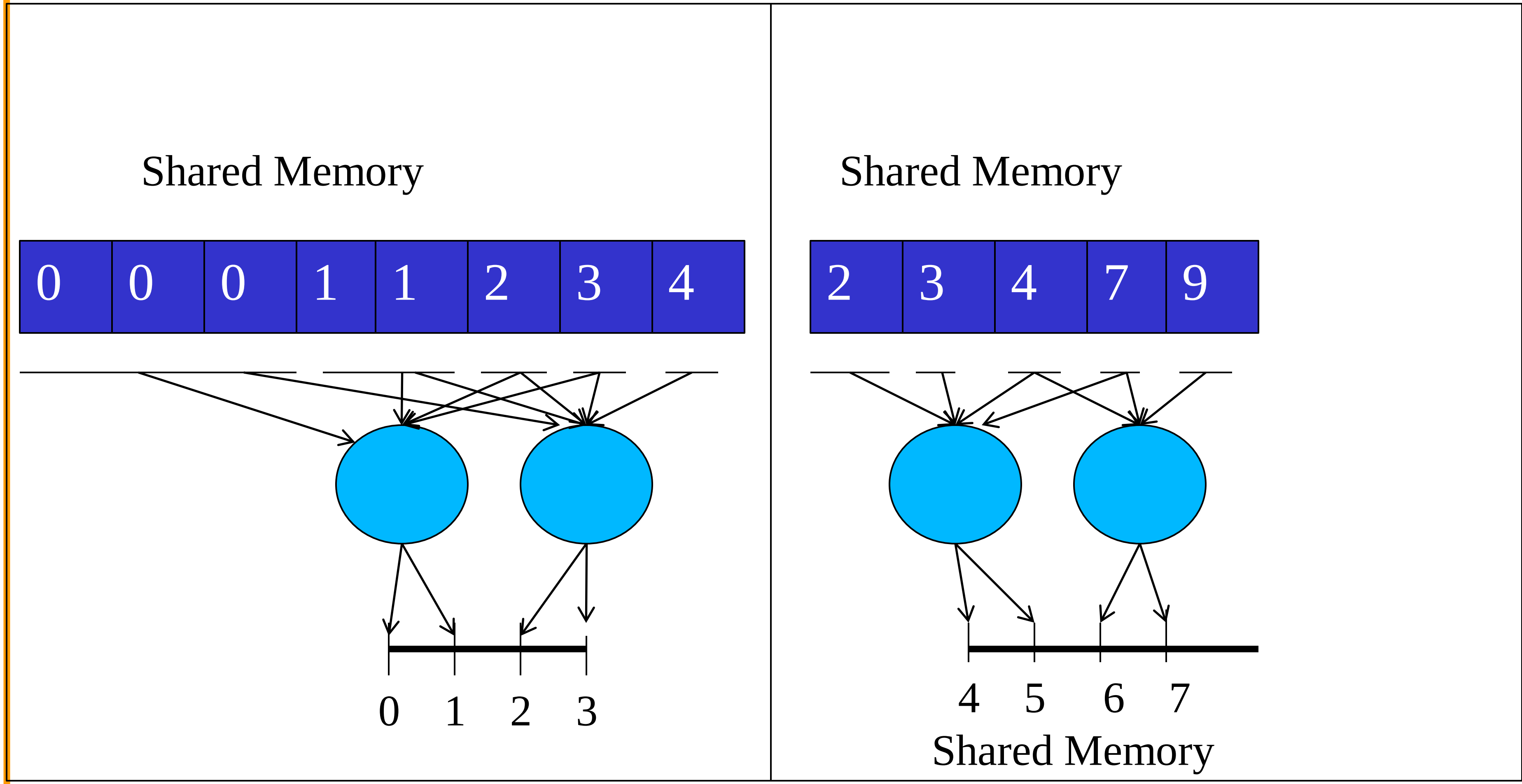


Actual Binning

- All inputs can now be placed into their bins in parallel, using atomic operations

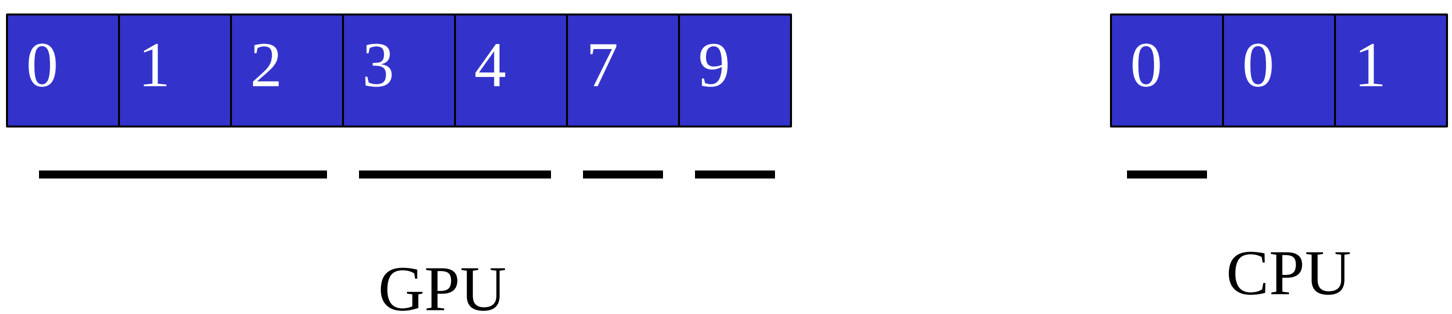


A Tiled Gather Implementation



Controlling Load Balance (done during capacity generation)

- Limit the size of each bin
  - When counter exceeds limit for a bin, the input samples are placed into a “CPU” overflow bin
  - CPU places excess sample points into a CPU list
  - CPU does gridding on the excess sample points in parallel with GPU
  - Eventually merge



Set a Limit on Bin Capacities

0	1	0	0	2	3	1	4	9	7

Capacity of each bin limited to 1

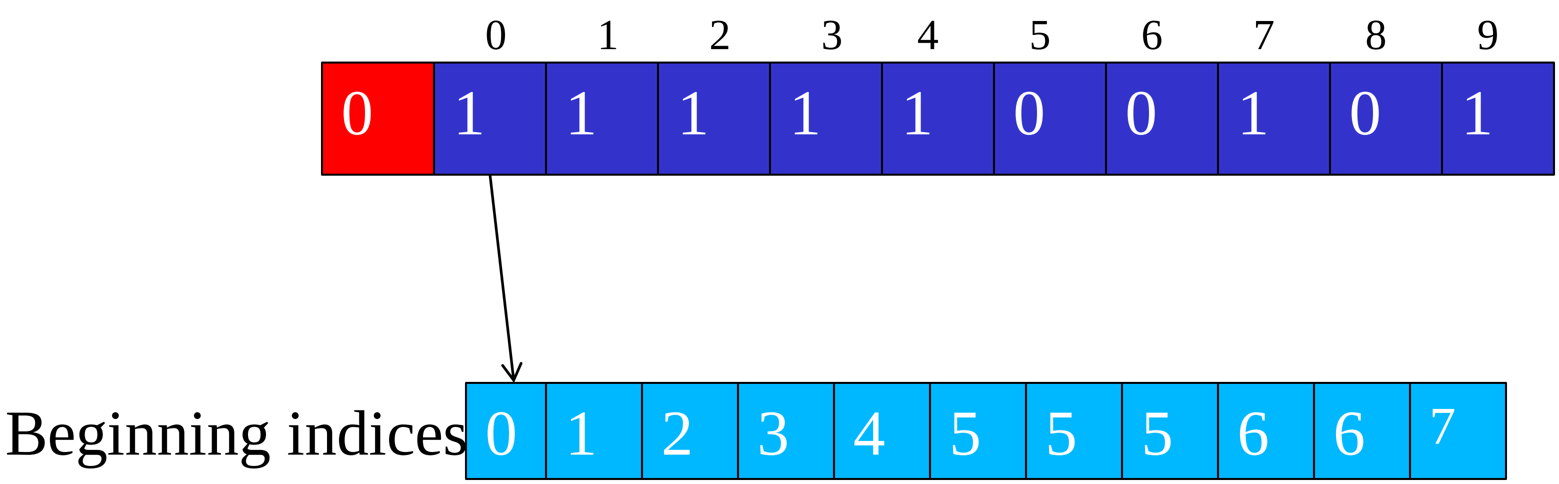
1	1	1	1	1	0	0	1	0	1
Bin 0	Bin 1	Bin 2	Bin 3	Bin 4			Bin 7		Bin 9

When a bin capacity reaches a preset limit, do not further increment the capacity counter

But place the excess input into an overflow bin

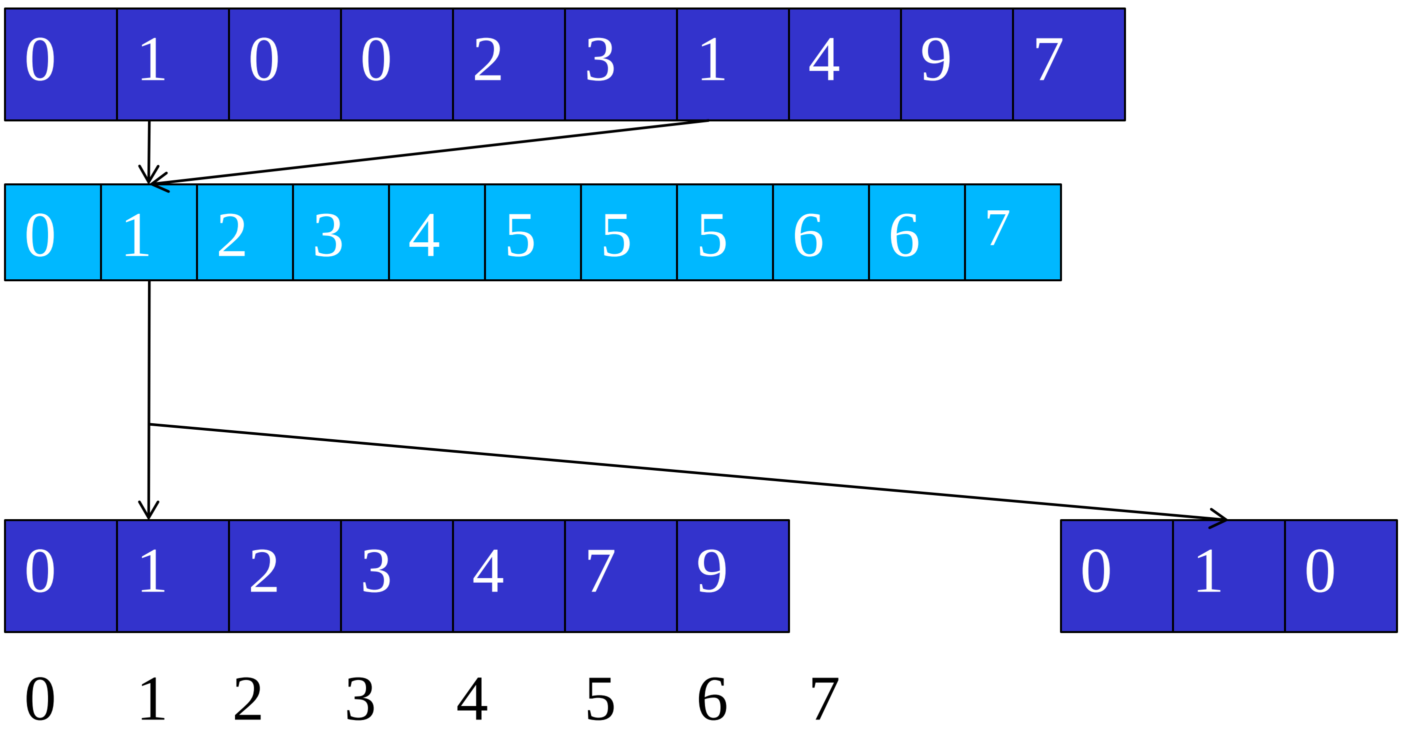
Determine Start and End of Bins

- Use parallel scan operations on the bin capacity array to generate an array of starting points of all bins (CUDPP)



Actual Binning

- All inputs can now be placed into their bins in parallel



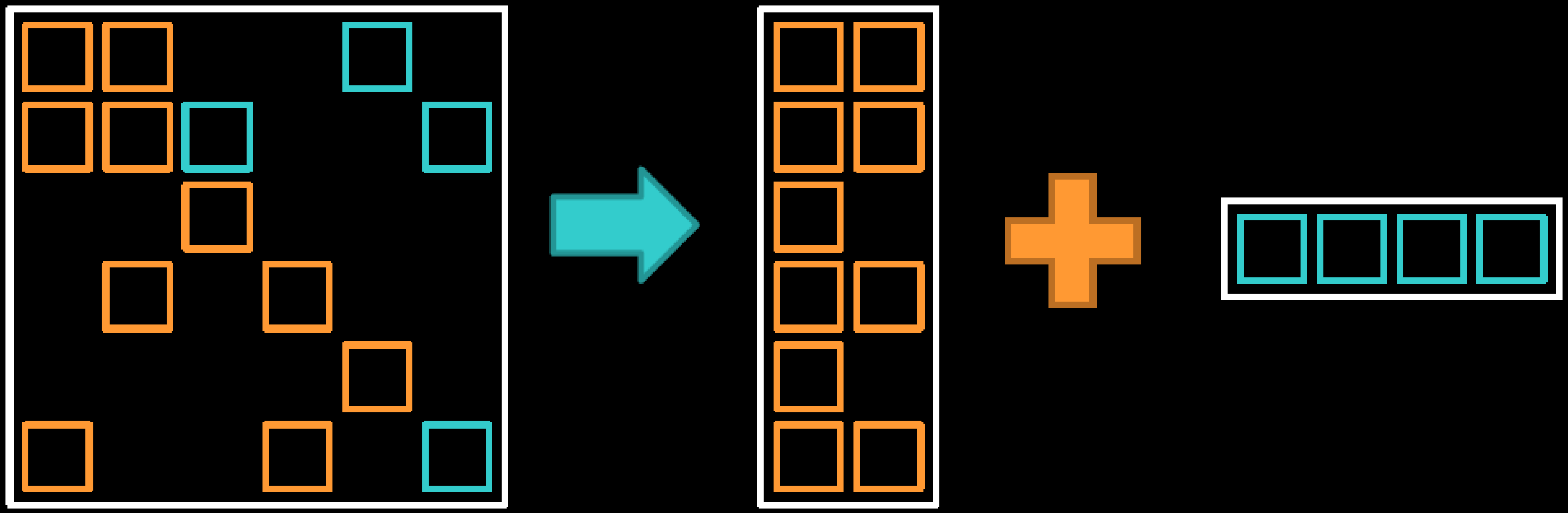
Note the similarity

- Compact bins – CSR
- Overflow bins - COO

- One could use ELL or JDS type of optimization on bins if desired

Hybrid Format

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



Eight Optimization Patterns for Algorithms (so far)

Technique	Contention	Bandwidth	Locality	Efficiency	Load Imbalance	CPU Leveraging
Tiling		X	X			
Privatization	X		X			
Regularization				X	X	X
Compaction		X				
Binning		X	X	X		X
Data Layout Transformation	X		X			
Thread Coarsening	X	X	X	X		
Scatter to Gather Conversion	X					

http://courses.engr.illinois.edu/ece598/hk/

Impact of Techniques on Apps

Benchmark	Unoptimized Implementation Bottleneck	Optimizations Applied	Optimized Implementation Bottleneck	Primary Limit of Efficiency
cutcp	Contention, Locality	Scatter-to-Gather, Binning, Regularization, Coarsening	Instruction Throughput	Reads/Checks of Irrelevant Bin Data
mri-q	Poor Locality	Data Layout Transformation, Tiling, Coarsening	Instruction Throughput	N/A (true bottleneck)
gridding	Contention, Load Imbalance	Scatter-to-Gather, Binning, Compaction, Regularization, Coarsening	Instruction Throughput	Reads/Checks of Irrelevant Bin Data
sad	Locality	Tiling, Coarsening	Memory Bandwidth/Latency	Register Capacity
stencil	Locality	Coarsening, Tiling	Bandwidth	Local Memory, Register Capacity
tpacf	Locality, Contention	Tiling, Privatization, Coarsening	Instruction Throughput	N/A (true bottleneck)
lbm	Bandwidth	Data Layout Transformation	Bandwidth	N/A (true bottleneck)
dmm	Bandwidth	Coarsening, Tiling	Instruction Throughput	N/A (true bottleneck)
spmv	Bandwidth	Data Layout Transformation	Bandwidth	N/A (true bottleneck)
bfs	Contention, Load Imbalance	Privatization, Compaction, Regularization	Bandwidth	Whole-Device Local Memory Capacity
histogram	Contention, Bandwidth	Privatization, Scatter-to-Gather	Bandwidth	Reads of Irrelevant Input (alleviated by cache)

Challenges of Parallel Programming

- Computations with no known scalable parallel algorithms
  - Shortest path, Delaunay triangulation, …
- Data distributions that cause catastrophical load imbalance in parallel algorithms
  - Free-form graphs, MRI spiral scan
- Computations that do not have data reuse
  - Matrix vector multiplication, …
- Algorithm optimizations that are require expertise
  - Locality and regularization transformations

Benefit from other people's experience

- GPU Computing Gems Vol 1
  - Coming January 2011
  - 50 gems in 10 applications areas
  - Scientific simulation, life sciences, statistical modeling, emerging data-intensive applications, electronic design automation, computer vision, ray tracing and rendering, video and imaging processing, signal and audio processing, medical imaging
- GPU Computing Gems Vol 2
  - Coming in May 2011
  - 50+ gems in more application areas, tools, environments

THANK YOU!