VSCSE Summer School

Proven Algorithmic Techniques for Many-core Processors

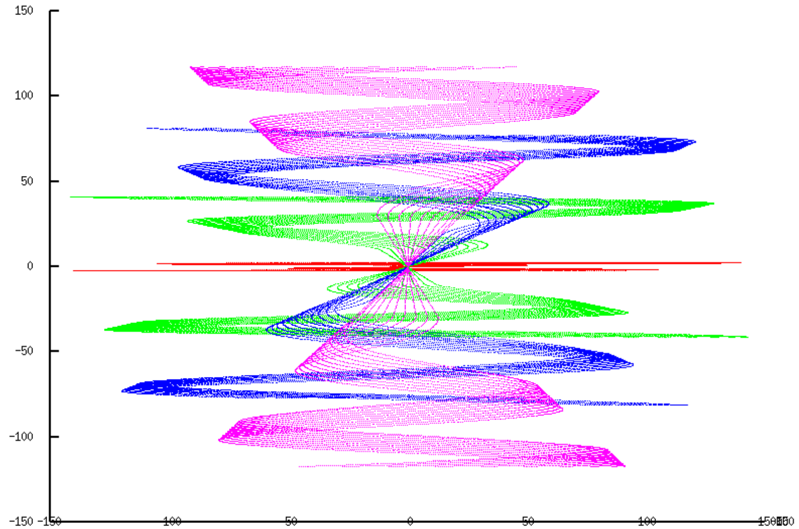
Lecture 7: Dealing with Non-Uniform Data

MRI Reconstruction

- Transform MR data samples from the k-space into the image space using IFFT

- MRI scanners increasingly use spiral trajectories in a **cylindrical** or **spherical** coordinate system

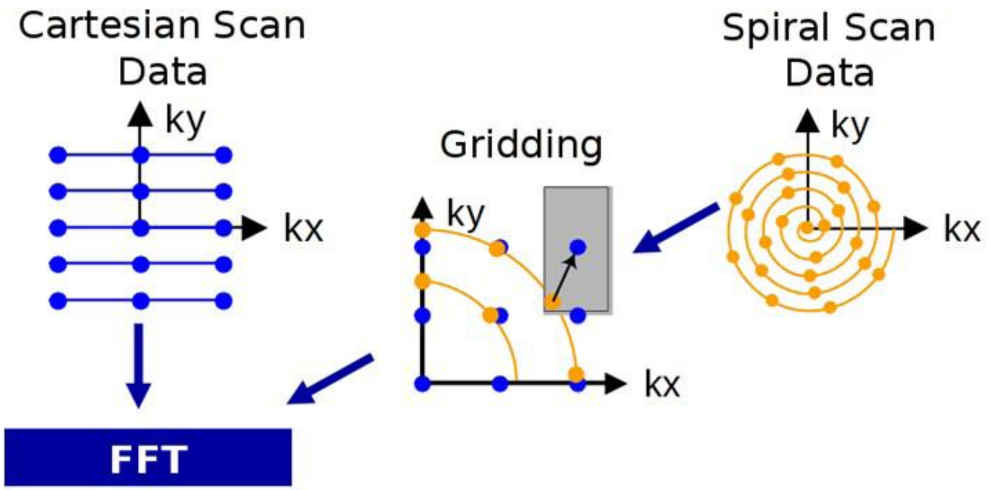
  $\Rightarrow$ The image cannot be reconstructed by directly applying IFFT to the k-space samples



Gridding to Enable IFFT

- Instead, map non-Cartesian samples in frequency domain onto a 3D Cartesian grid based on a Kaiser-Bessel function **(Gridding)**

- Next, perform IFFT on grid to transform it to the image domain



Large Number of Sample Points

- Each sample contains
  - Its K-space Coordinates, s.coordinates
  - Its strength value, s.value

- Given s.coordinates, it is easy to calculate the range of grid points affected
  - Cutoff distance

An Input Oriented Sequential Code

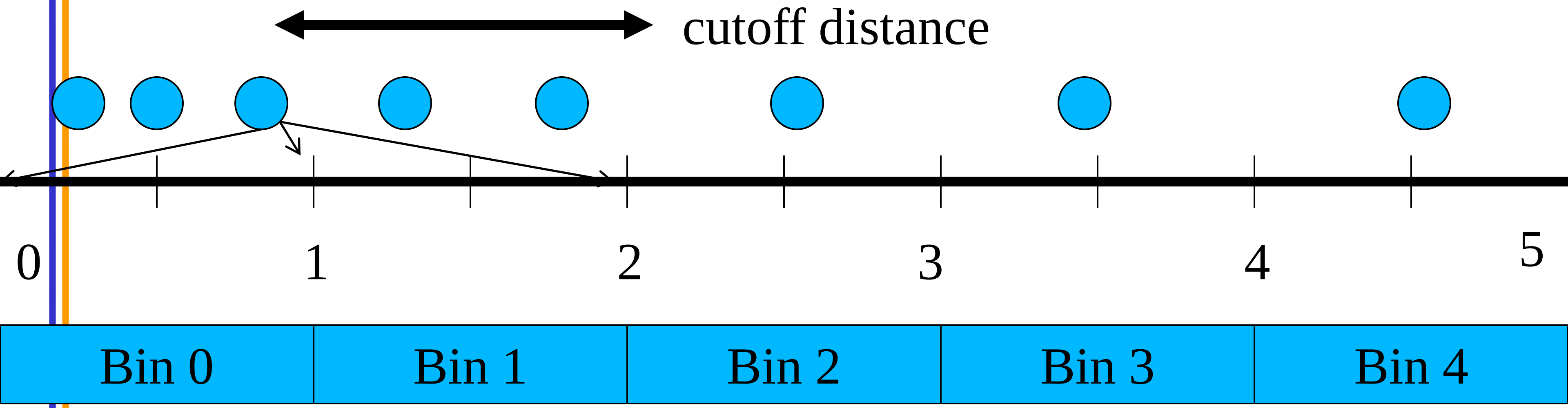
- Uses a window function

$$output = \left\{ \frac{\text{f(input) : within some cutoff distance}}{\text{Zero : beyond cutoff distance}} \right.$$

```
for (every sample point s){
    for(z in range){
        for(y in range){
            for(x in range){
                weight = kaiser_bessel(|<s.coords>-<x,y,z>|)
                grid[z][y][x] += s.value * weight;
            }
        }
    }
}
```

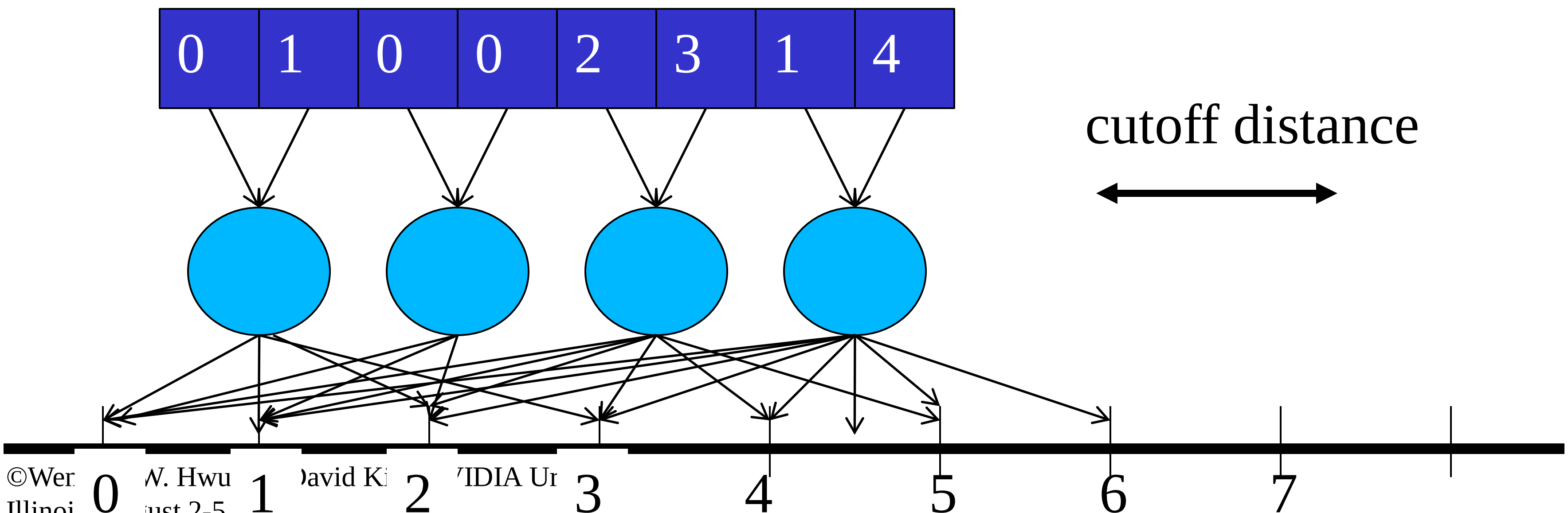
Binning of Sample Points

- 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)



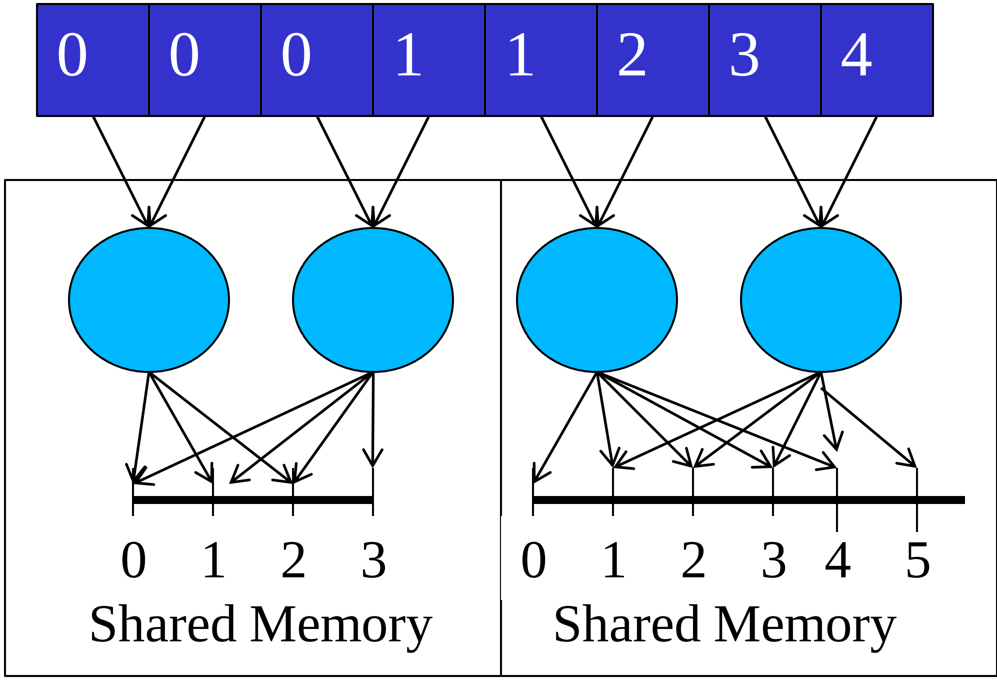
A Scatter Parallelization

- Use each thread to process N sample points
- Use Global Memory atomic operation to accumulate into grid points
    - Each sample point affects all grid points with cutoff distance
- Slow, but not pathologically slow
    - Fermi runs this faster than its predecessors



A Faster Scatter Implementation

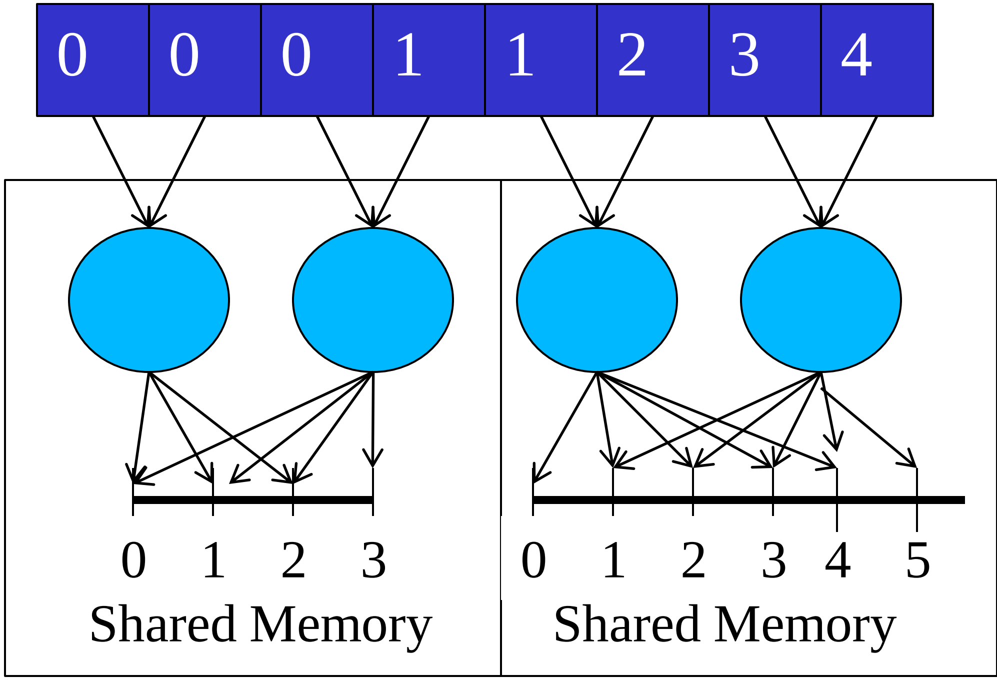
- Algorithm:
  - Sort input data
  - Each block processes one section of sample points
    - Each thread processes a smaller section of sample points
  - Create window of grid points in shared memory and compute into it when possible
    - Quick inspection of the end sample points of sample section determines window
  - Use atomic operation to coordinate across threads



Regularization helps scatter too.

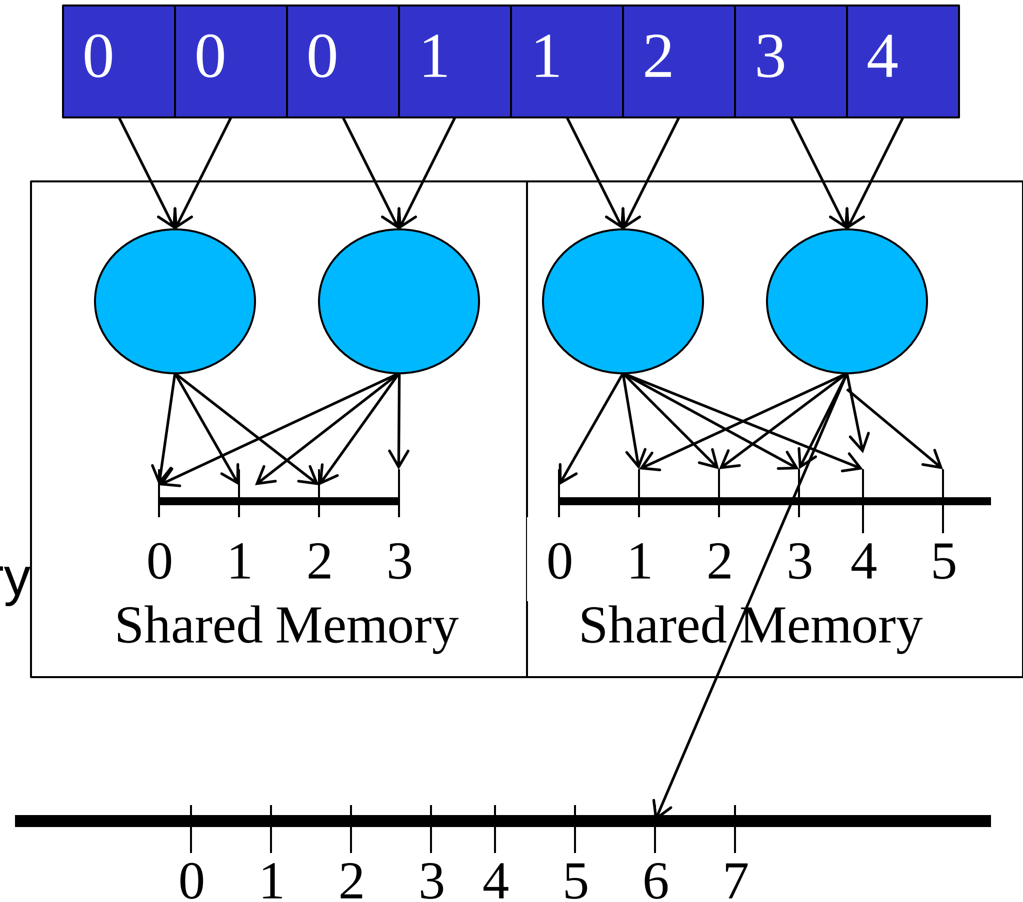
Limited Space in Shared Memory

- The sample sections vary in the span of grid points they cover
  - The algorithms works best in sample points are mostly concentrated in small grid regions
- Limit the size of grid point window for Share Memory limitation and copy-merge overhead
- Moderate level of contention



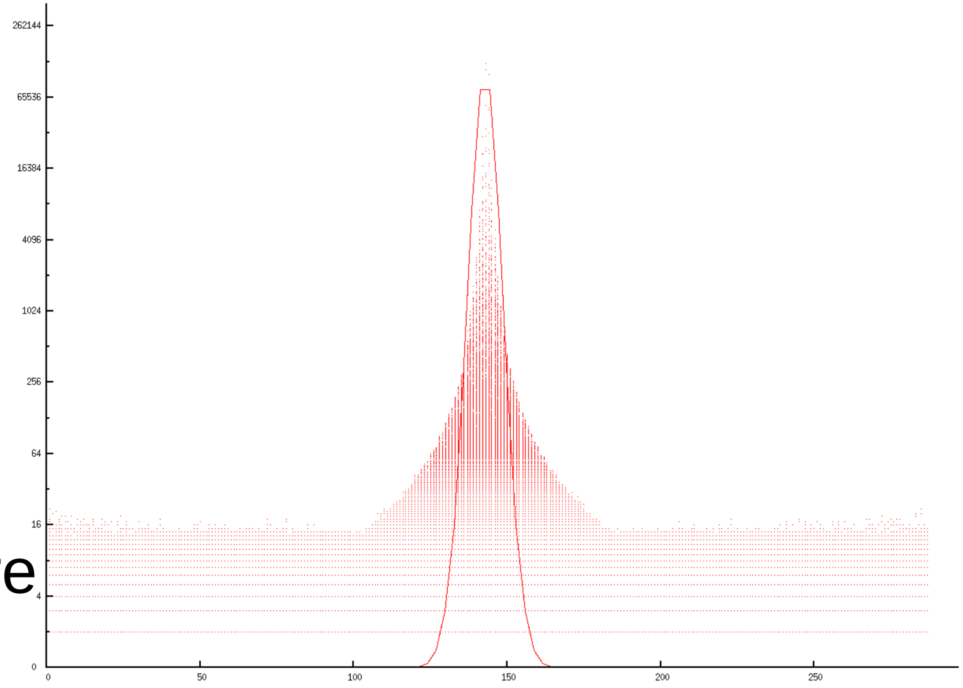
Accesses to Global Memory

- A condition test of bin value and widow span determines if an affected grid point is in Shared Memory window
  - Use atomic operation to accumulate into Global Memory
  - low contention
- At the end of block execution, thread collectively merge window back to Global Memory
  - Use atomic operation
  - Moderate to low contention

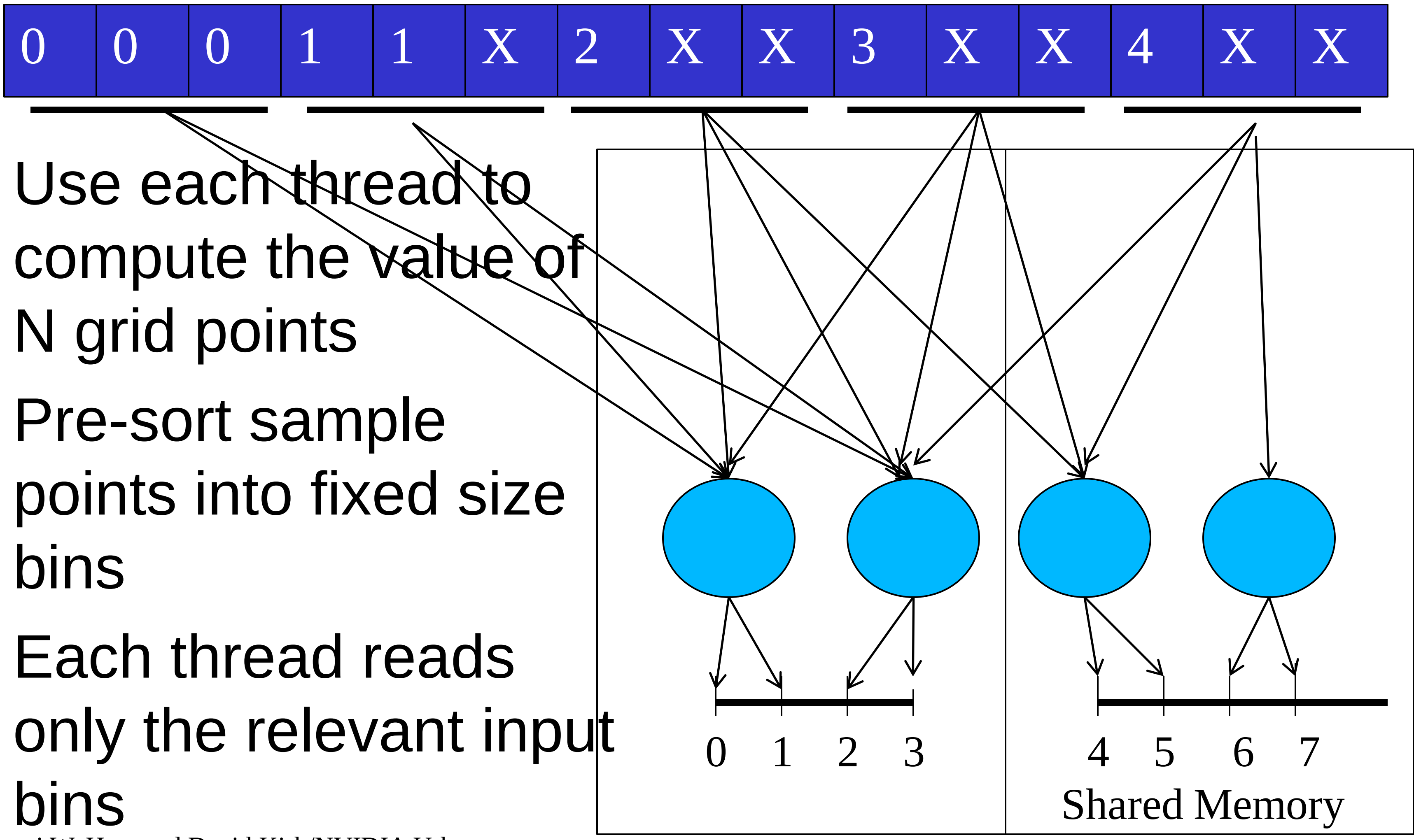


How about Scatter Parallelization

- No contention
- We know we can bin sample points
- However, there can be great load imbalance
  - Some grid points are affected by many more sample points than others

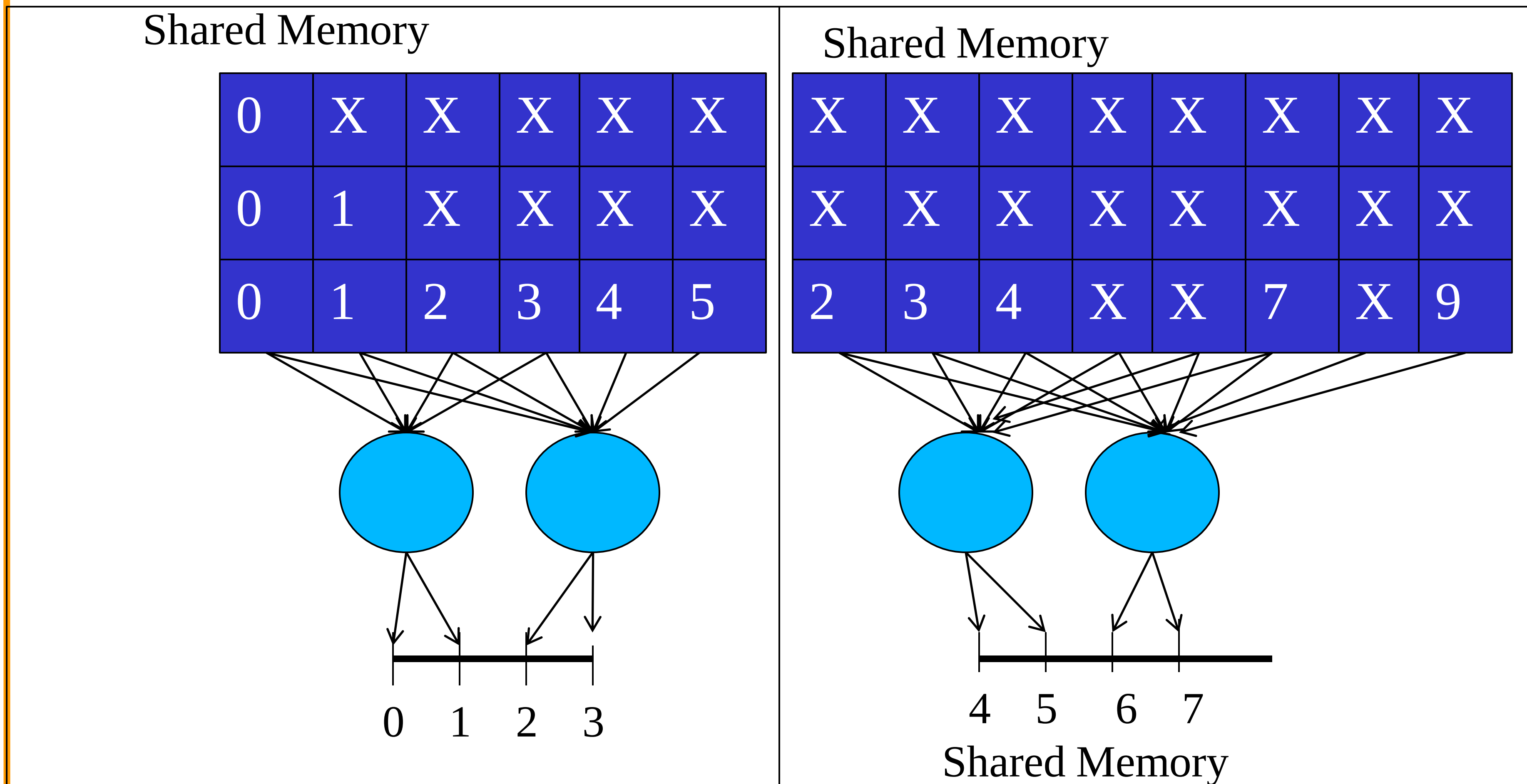


A Binned Gather Parallelization



- 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

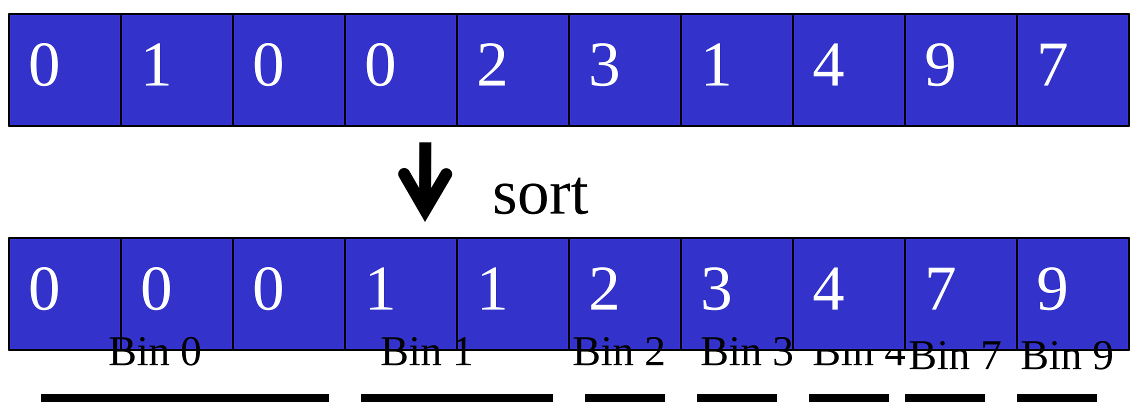


More on Tiled Gather

- Threads cooperate to load the relevant bins for all of them from Global Memory to Shared Memory
- Each thread only access 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

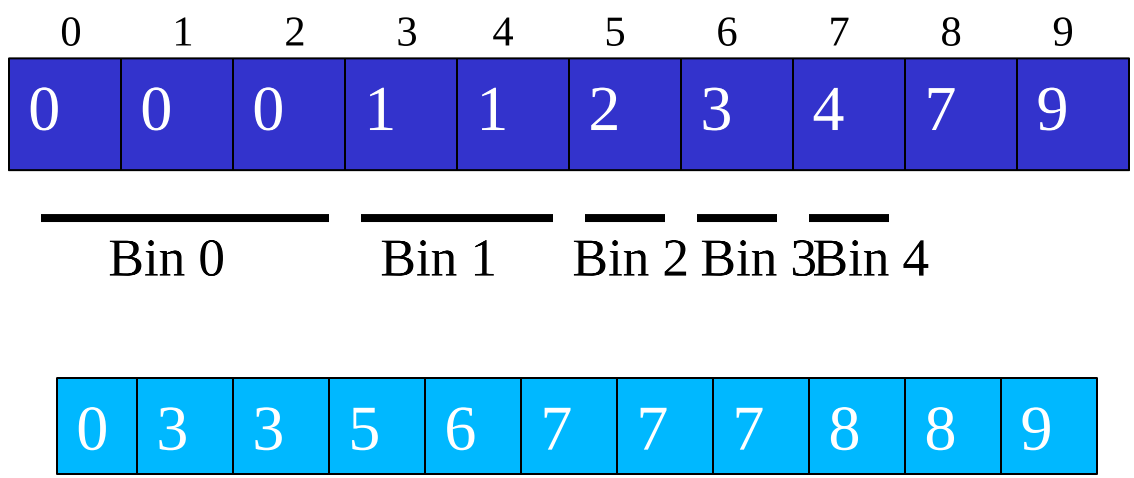
Implicit Binning for Gather Parallelization

- Don't use pre-allocated fixed size bins (multi-dimensional array)
- Sort samples into bins of varying sizes in input array instead
  - Bins 5, 6, 8 are implicit, zero-sample

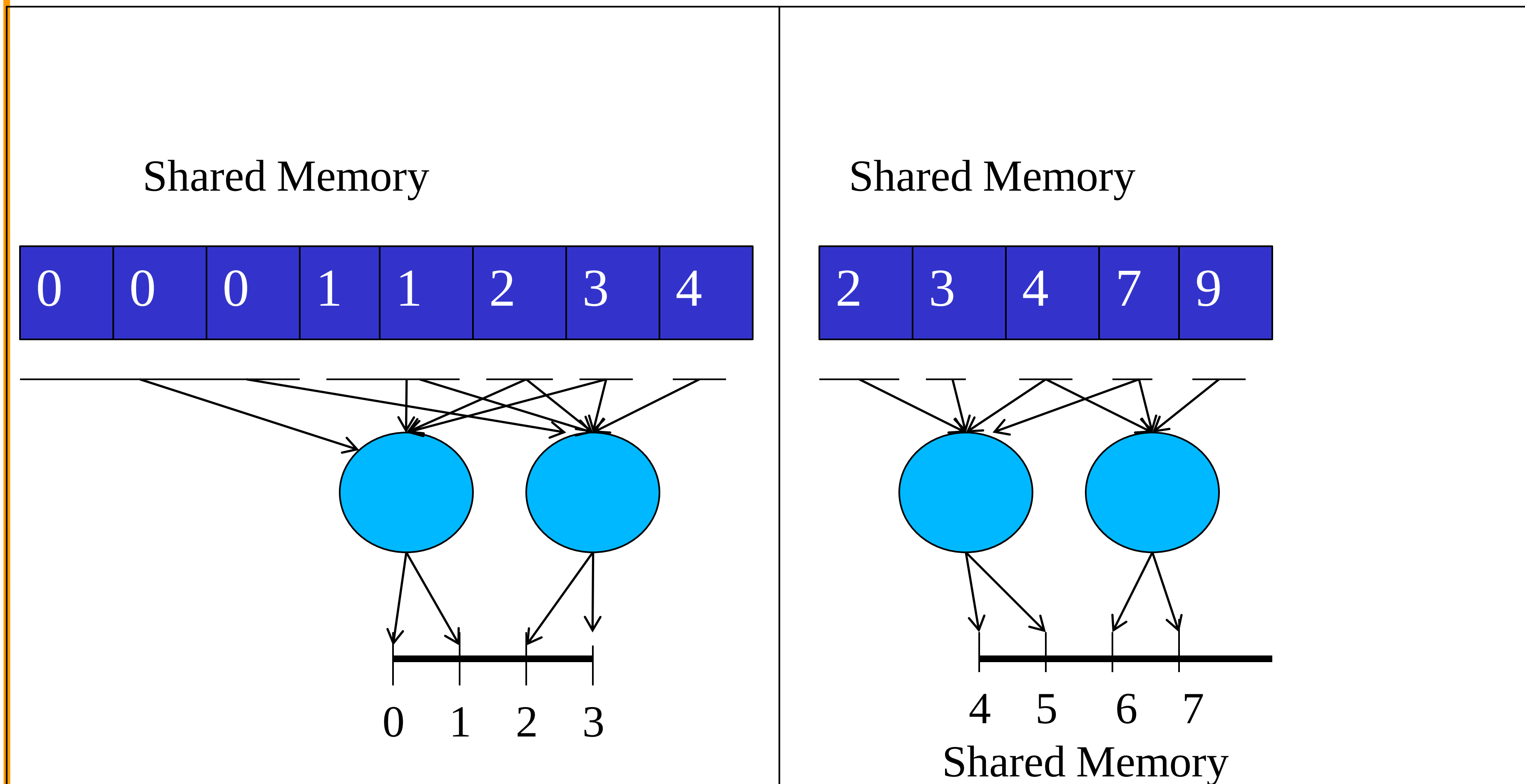


Determine Start and End of Bins

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

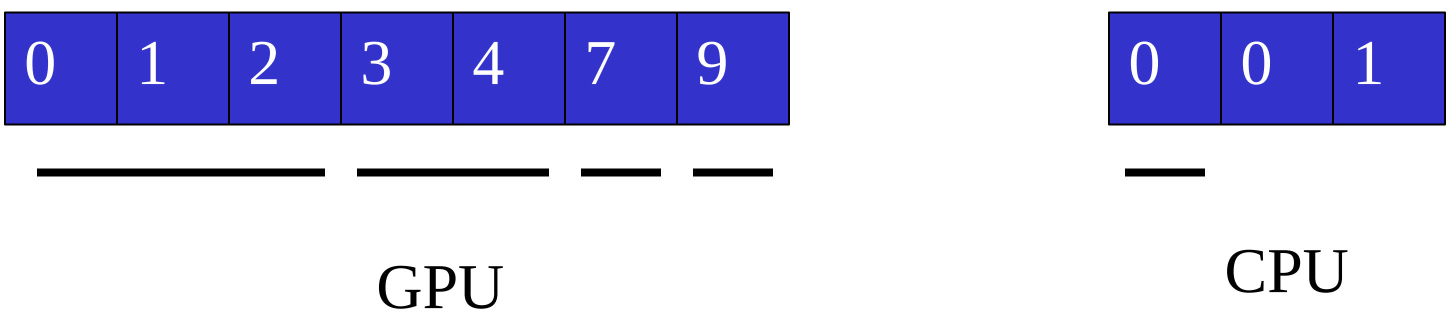


A Tiled Gather Implementation

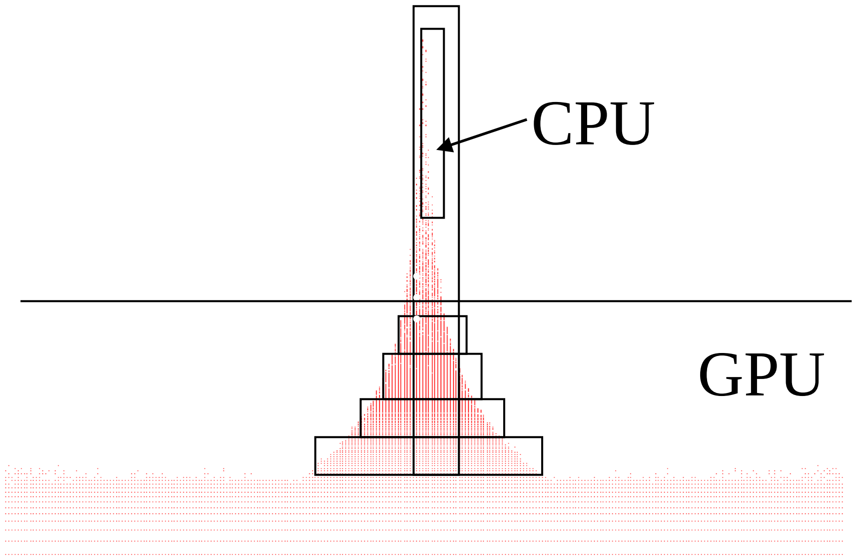


Controlling Load Balance

- Limit the size of each bin
  - Both uniform and variable/implicit bins
  - CPU places excess sample points into a CPU list
  - CPU does gridding on the excess sample points in parallel with GPU
  - Eventually merge



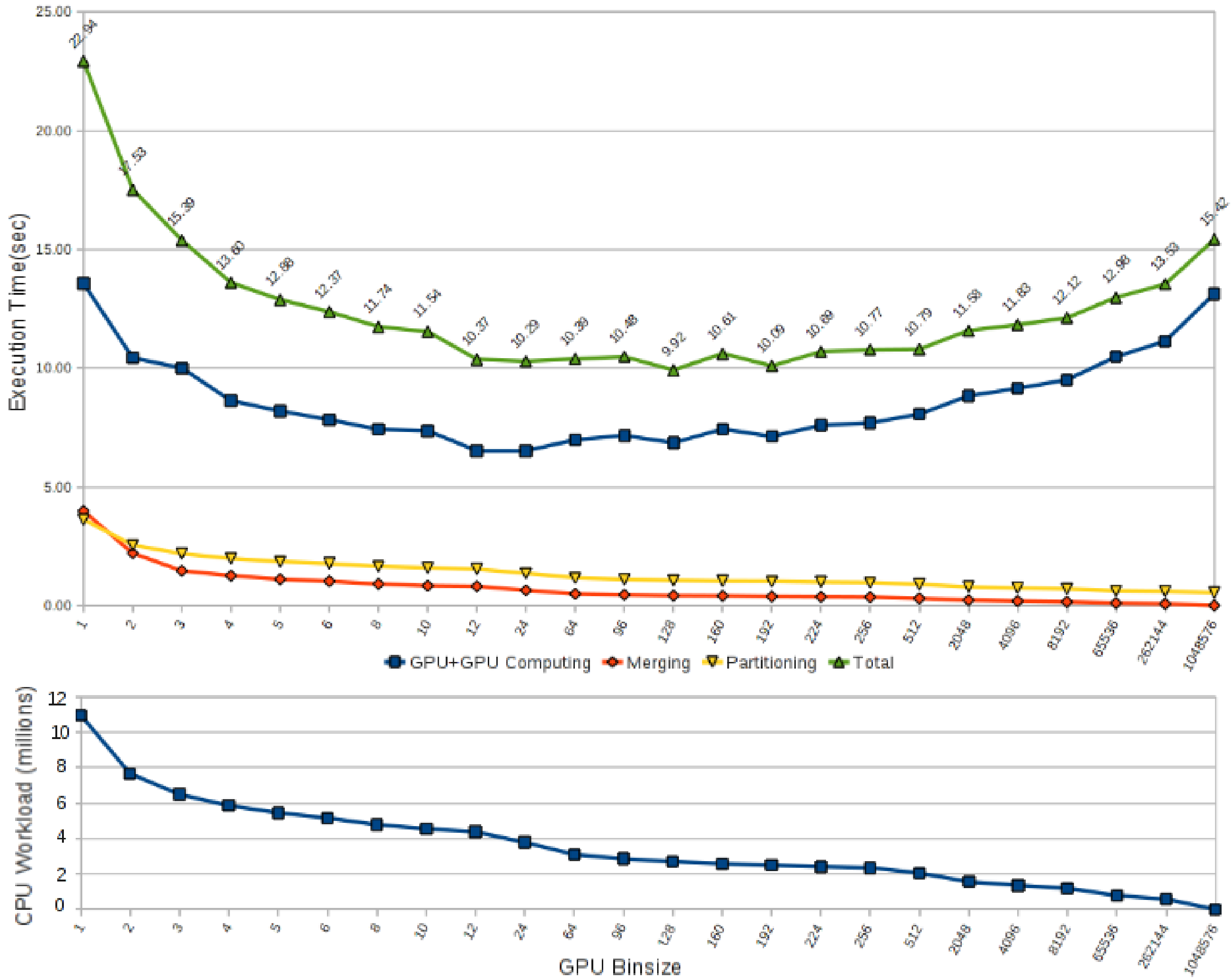
Determining the bin-size limit can be tricky.



- Higher limit creates more load imbalance on GPU
- Lower limit may cause too much CPU execution time

- What is the best bin size?

There is a range of good bin sizes for each processor.

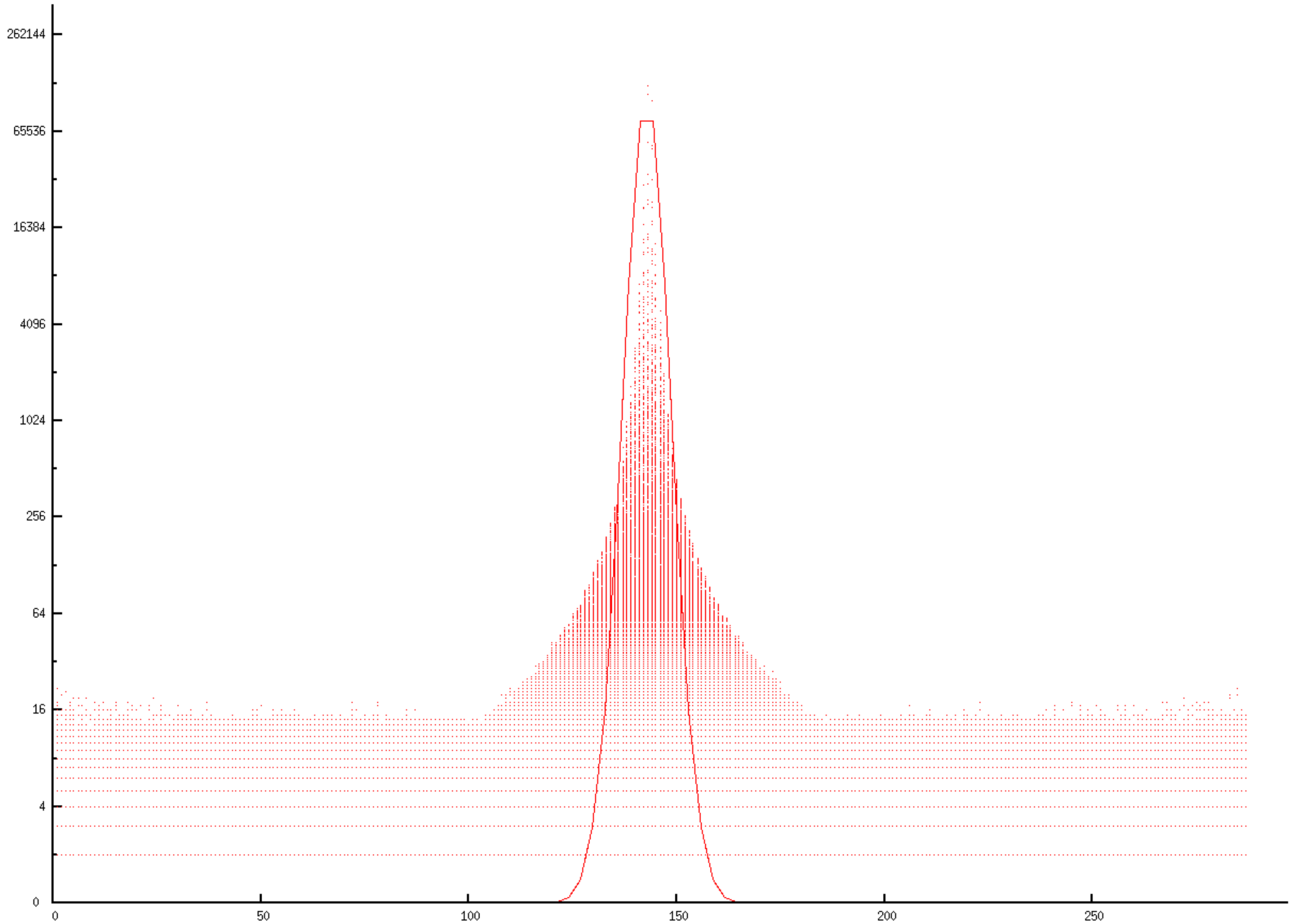


Performance Chart

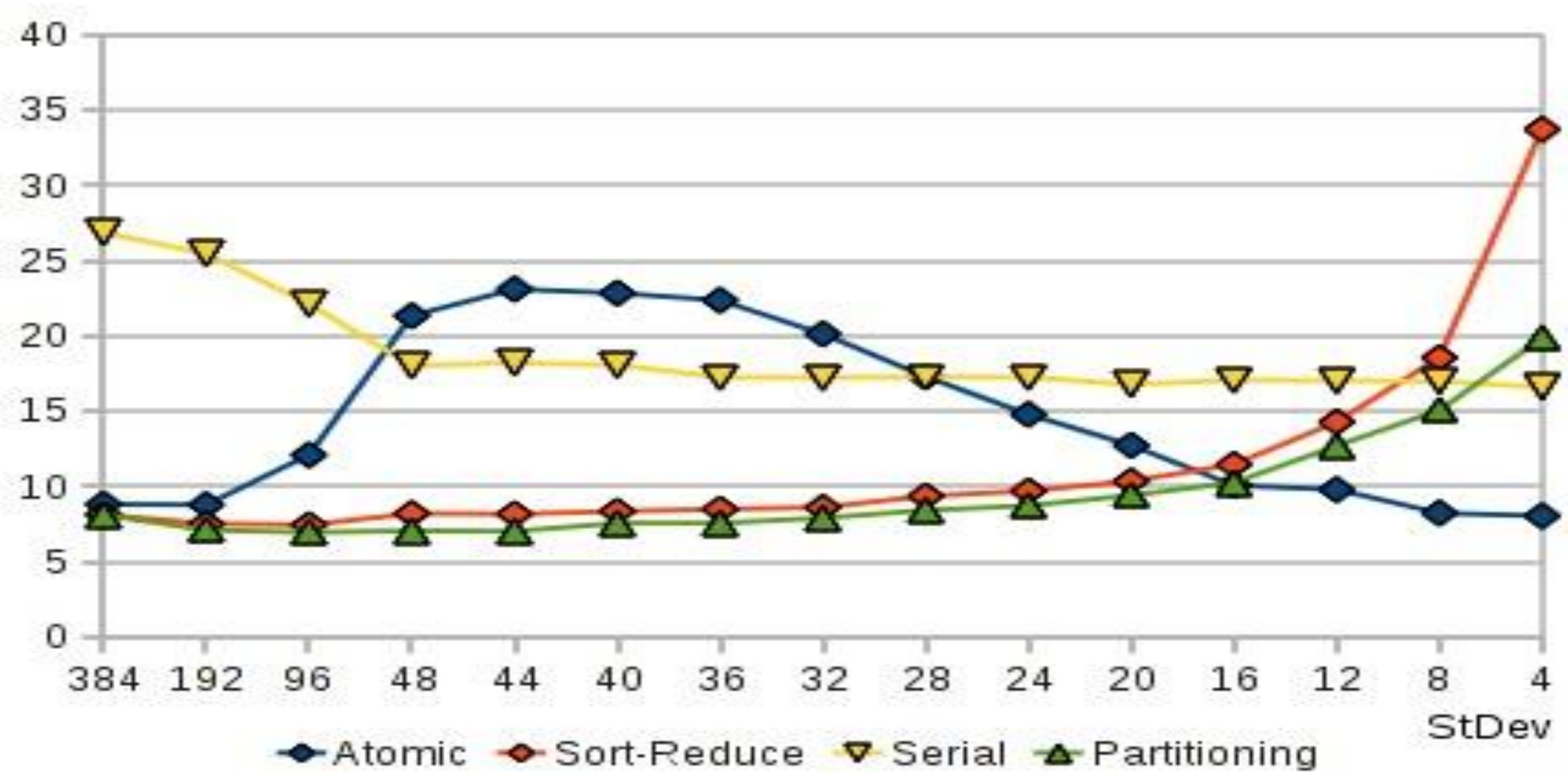
- Partitioning achieves 43% speedup over sort-reduce without partitioning

Algorithm	Time (sec.)	Speedup
Sequential	23.44	1X
Shared Atomic	15.4	1.52X
Sort-Reduce	14.25	1.64X
Sort-Reduce + Partitioning (binsize limit=128)	9.92	2.36X

Determining Appropriate Algorithm



The Answer is depends.



ANY FURTHER QUESTIONS?