# GPU ACCELERATED SELF-JOIN FOR THE DISTANCE SIMILARITY METRIC

MIKE GOWANLOCK

*NORTHERN ARIZONA UNIVERSITY*

*SCHOOL OF INFORMATICS, COMPUTING & CYBER SYSTEMS*

BEN KARSIN

*UNIVERSITY OF HAWAII AT MANOA*

*DEPARTMENT OF INFORMATION AND COMPUTER SCIENCES*

# THE SELF-JOIN

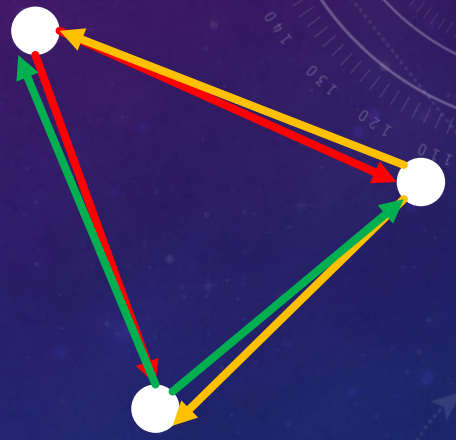
- The self-join is a fundamental operation in databases
- Find all objects within a threshold distance of each other
  - Range queries around each point
  - A table joined onto itself with a distance similarity predicate

# APPLICATIONS

- Building blocks of many clustering algorithms (e.g., DBSCAN)
- Can be used for kNN
- Time series data analysis
- Mining spatial association rules
- Many other applications/algorithms

# NESTED LOOP JOIN (BRUTE FORCE)

- Two for loops
- Each point performs a distance calculation between itself and the other points
- $O(n^2)$
- Performs well in high dimensionality



Example: 3 points, 9 total distance calculations

- 3 total distance calculations between a point and itself
- 6 total distance calculations between points

# LITERATURE ON THE (SELF-) JOIN

| **Low-dimensionality** | **High-dimensionality** |
| --- | --- |
| Denser data<br>• Many points within a search radius | Sparser Data<br>• Fewer points within a search radius |
| Major focus on indexing techniques<br>• Reduce the number of distance calculations needed to find points within a search radius | Indexing techniques do not perform well<br>• *Curse of dimensionality:* index cannot discriminate between points in the high-dimensional space |
| Challenge<br>• Large result sets and number of distance comparisons | Challenge<br>• Large fraction of time searching for candidate points that may be within the distance |

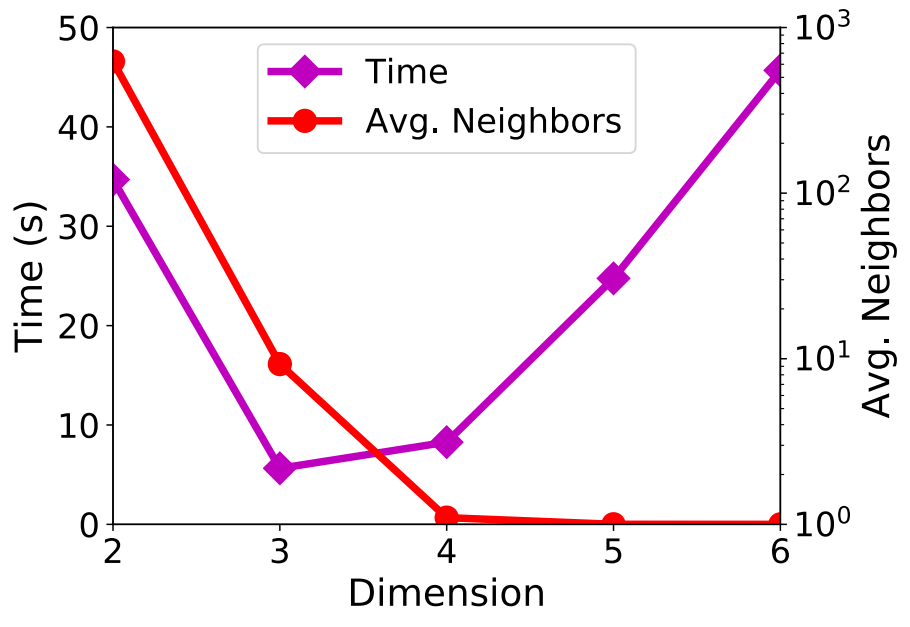
# LITERATURE ON THE (SELF-) JOIN

| Low-dimensionality | High-dimensionality |
| --- | --- |
| Denser data<br>• Many points within a search radius | S<br>• |
| Major focus on indexing techniques<br>• Reduce the number of distance calculations needed to find points within a search radius | Ir<br>• |
| Challenge<br>• Large result sets and number of distance comparisons | C<br>•<br>the distance |

The literature is (mostly) split between low- and high-dimensional contributions

- We focus on the low-dimensional case
- Dimensions 2-6

# PERFORMANCE EXAMPLE




- Using an R-tree index, with a fixed distance *epsilon*=1
- 2 million points
- The response times are greatest at 2-D and 6-D
- The number of neighbors decreases to 0 with dimension
- At 2-D the higher response time is due to many distance calculations
- A 6-D the higher response time is due to more exhaustive index searches

# UTILIZING THE GPU

- GPUs have thousands of cores
- High memory bandwidth
  - 700 GB/s on Pascal, 900 GB/s Volta
- CPU main memory bandwidth
  - ~100 GB/s
- Overall: The GPU's high memory bandwidth makes it an attractive alternative to the CPU

# UTILIZING THE GPU

- CPU-based self-joins are often characterized by an irregular instruction flow:
  - Spatial index searches use tree traversals
- Insights into spatial indexes for the GPU
  - J. Kim, W.-K. Jeong, and B. Nam, “Exploiting massive parallelism for indexing multi-dimensional datasets on the gpu,” IEEE Transactions on Parallel and Distributed Systems, vol. 26, no. 8, pp. 2258–2271, 2015.
  - J. Kim, B. Nam, “Co-processing heterogeneous parallel index for multi-dimensional datasets” JPDC, 113, pp. 195–203, 2018.

# UTILIZING THE GPU

- CPU-based self-joins are often characterized by an irregular instruction flow:
  - Spatial index searches use tree traversals
- Insights into spatial indexes for the GPU
  - J. Kim, W.-K. Jeong, and B. Nam, “Exploiting massive parallelism for indexing multi-dimensional datasets on the gpu,” IEEE Transactions on Parallel and Distributed Systems, vol. 26, no. 8, pp. 2258–2271, 2015.
  - J. Kim, B. Nam, “Co-processing heterogeneous parallel index for multi-dimensional datasets” JPDC, 113, pp. 195–203, 2018.

This paper implemented an R-tree on the GPU that avoided some of the drawbacks of the SIMD architecture

# UTILIZING THE GPU

- CPU-based self-joins are often characterized by an irregular instruction flow:
  - Spatial index searches use tree traversals
- Insights into spatial indexes for the GPU
  - J. Kim, W.-K. Jeong, and B. Nam, “Exploiting massive parallelism for indexing multi-dimensional datasets on the gpu,” IEEE Transactions on Parallel and Distributed Systems, vol. 26, no. 8, pp. 2258–2271, 2015.
  - J. Kim, B. Nam, “Co-processing heterogeneous parallel index for multi-dimensional datasets” JPDC, 113, pp. 195–203, 2018.

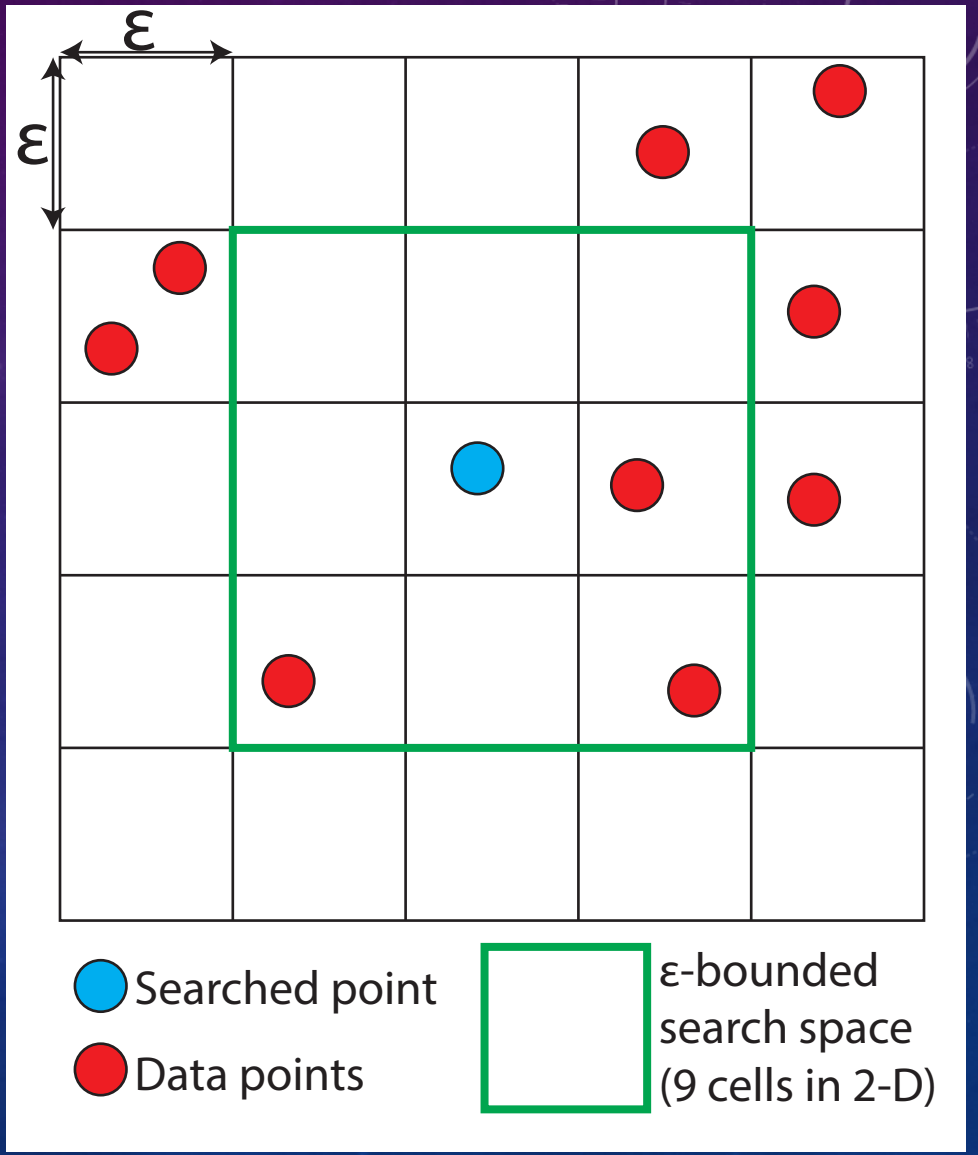
This paper showed its best to implement the traversal of the internal nodes (branching) on the CPU and the scan of the data elements in the leaf nodes on the GPU

# UTILIZING THE GPU: TAKEAWAY

- Due to the GPU's SIMT architecture, branching can significantly reduce performance when performing tree traversals
- It is better to have a bounded search, where all threads take the same or very similar execution pathways

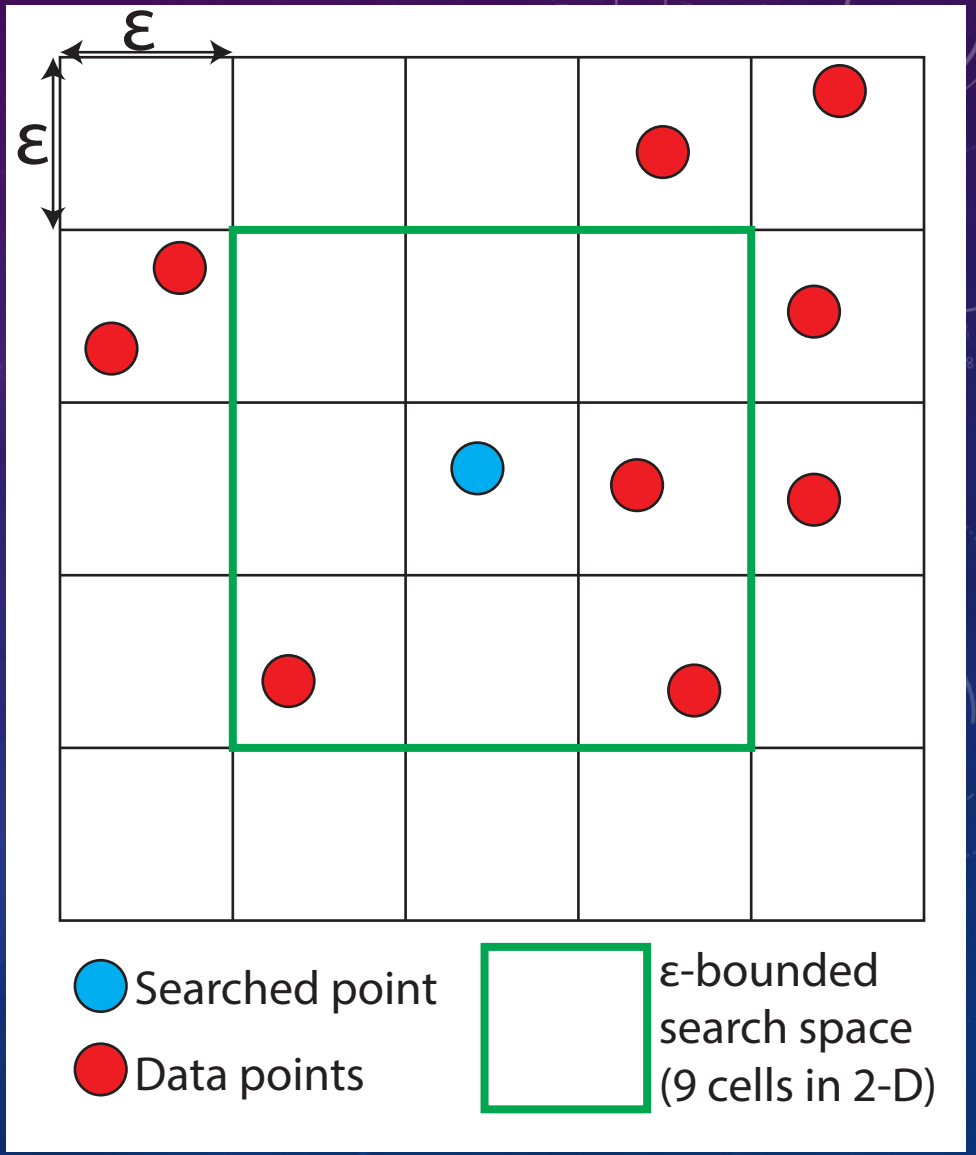
# GRID INDEX

- Grid Index
  - A grid is constructed with cells of length *epsilon*
- Point search
  - For each point in the dataset, the points within *epsilon* can be found by checking adjacent grid cells and performing distance calculations to the points in these cells
  - Adjacent cells: $3^n$ where *n* is the number of dimensions

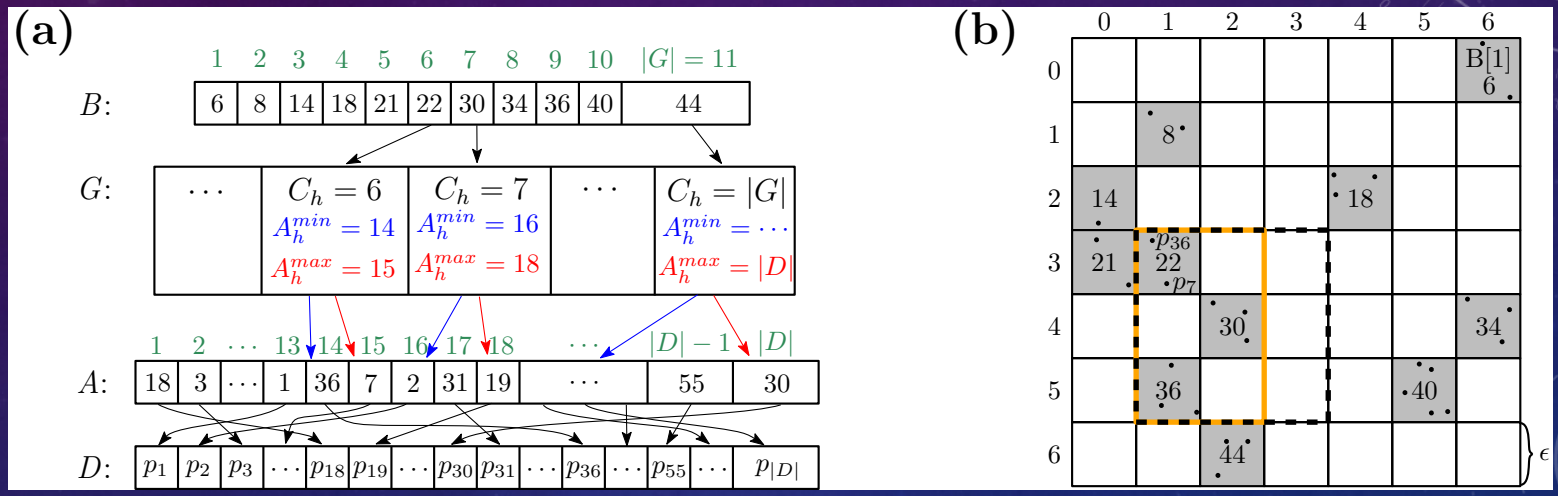


# GRID INDEX

- The search for the nearby points is bounded to the adjacent cells
  - No branching like spatial tree-based indexes
- Points within the same grid cell will return the same cells
  - Reduces thread divergence



# SPACE EFFICIENT GRID INDEX




- Store non-empty grid cells
- Use a series of lookup arrays
- Space complexity: $O(|D|)$, where $D$ is the number of data points in the dataset
- In practice in our experiments: a few MiB

# RESULT SET SIZES

- Self-join will generate large amounts of data that increase with:
  - *epsilon*
  - Size of the dataset
  - Point density distribution of the dataset
    - Large overdensities increase the total number of neighbors
- Need an efficient batching scheme to overlap computation and communication between the host and GPU

# EXAMPLE BATCHING SCHEME ILLUSTRATION


We use a minimum of 3 batches/kernel executions

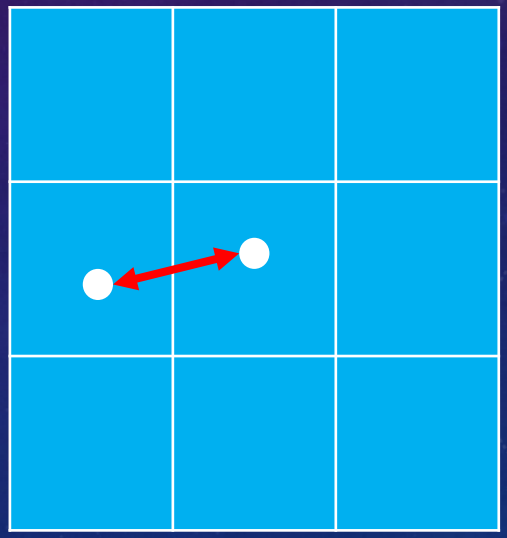
- Hide data transfers: overlap the kernel execution on the GPU with data transfer of the result sets

# BASELINE GLOBAL MEMORY GPU KERNEL

- Global memory kernel: one thread per point
- Number of GPU threads is the same as the dataset size
- A thread/point searches adjacent non-empty cells
- If a cell is non-empty, the thread computes the distance between it and all points in the cell

# AVOIDING DUPLICATE DISTANCE CALCULATIONS: UNIDIRECTIONAL COMPARISON

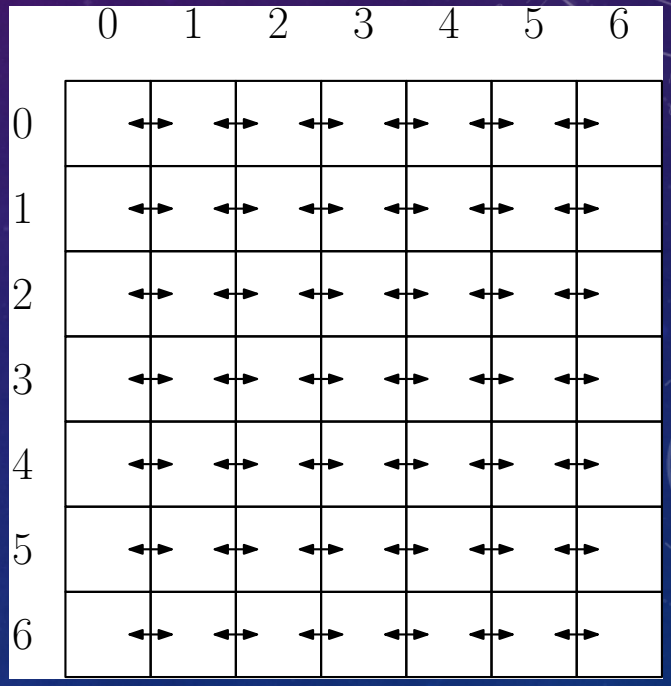
- We do not need to compute the distances between all pairs of points
- One nice property of the grid is that as the space is divided evenly we can reduce the number of distance comparisons between points



Example: these two points will find each other within *epsilon*. However, we can perform one distance calculation and record both results.

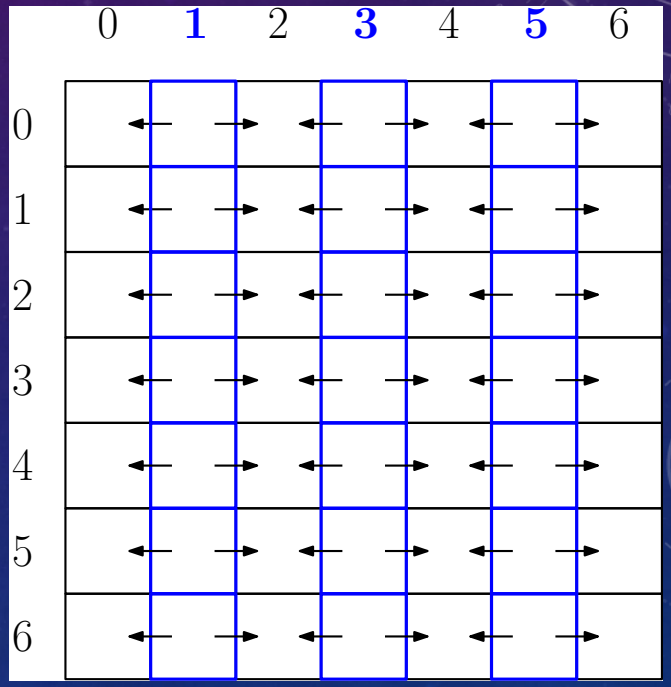
# UNIDIRECTIONAL COMPARISON: TWO DIMENSIONS

- We begin by comparing cells based on the first dimension (x-coordinate)
- Look at cells that share the *same* y-coordinate



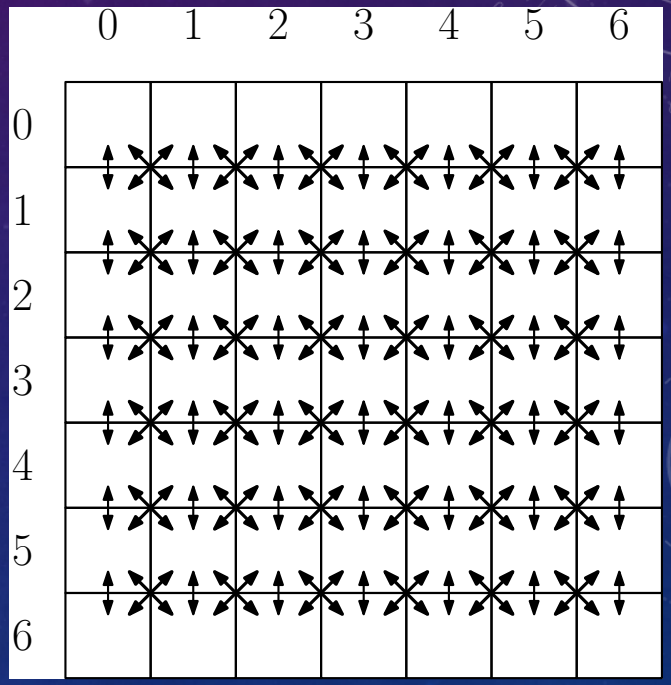
# UNIDIRECTIONAL COMPARISON: TWO DIMENSIONS

- We begin by comparing cells based on the first dimension (x-coordinate)
- Look at cells that share the *same* y-coordinate
- If the x-coordinate is odd, then we compare all points within the odd cell to the adjacent even coordinate cells
- If the x-coordinate is even, we do *nothing*



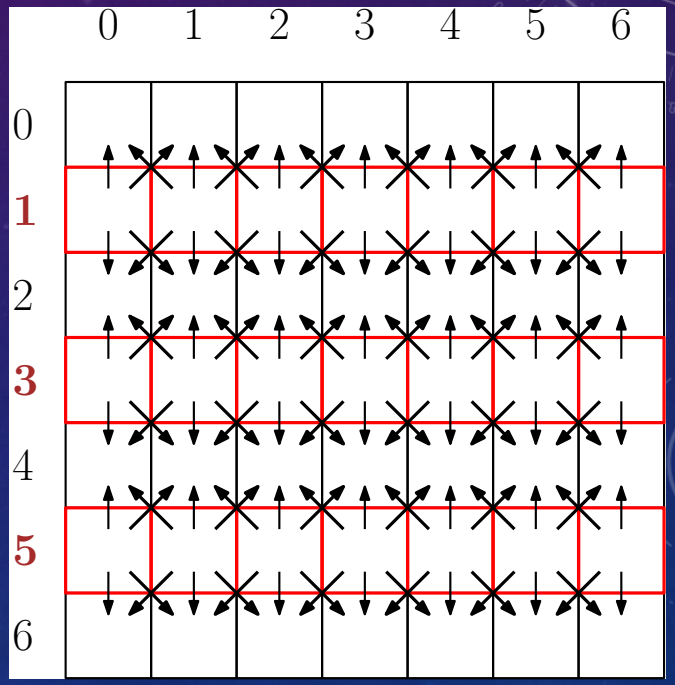
# UNIDIRECTIONAL COMPARISON: TWO DIMENSIONS

- We then compare neighbors with a *different* y-coordinate



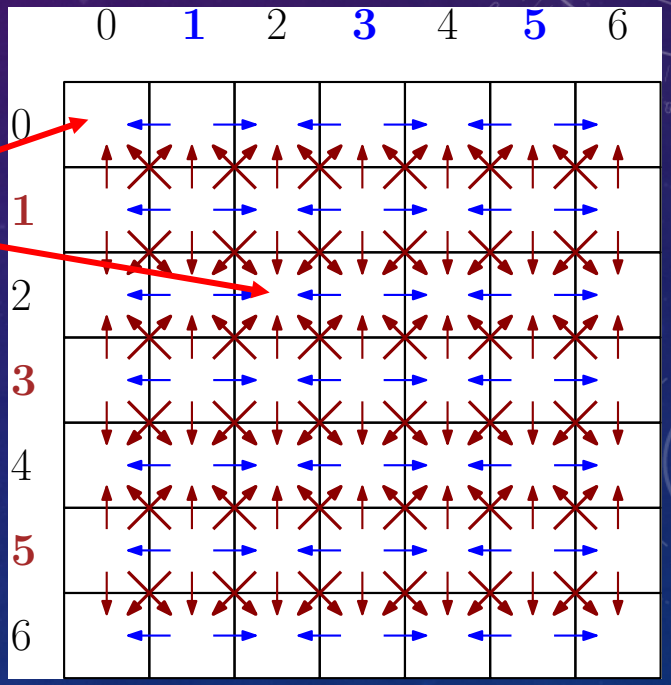
# UNIDIRECTIONAL COMPARISON: TWO DIMENSIONS

- We then compare neighbors with a *different* y-coordinate
- If the y-coordinate is odd, then we compare against the adjacent cells
- If the y-coordinate is even, then we do *nothing*



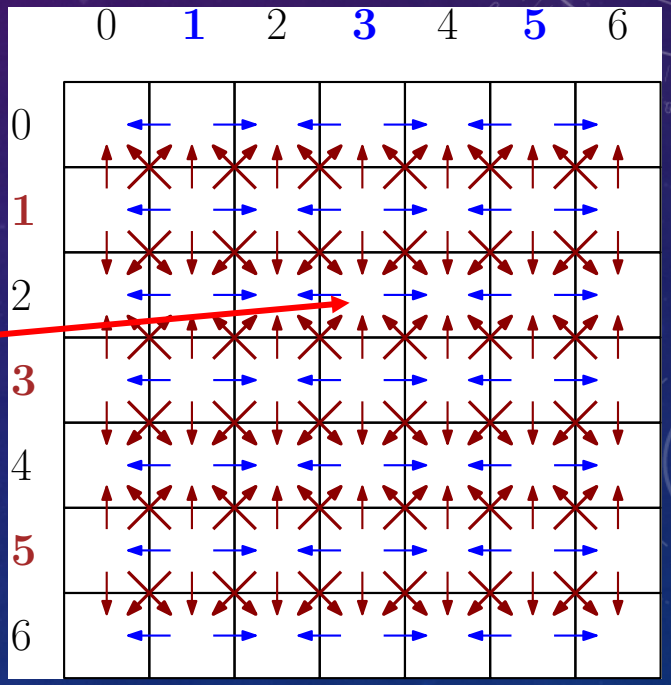
# UNIDIRECTIONAL COMPARISON: TWO DIMENSIONS

- In two dimensions, this is the final pattern of comparisons
- Note that in some cells, there are no searches originating from the cells



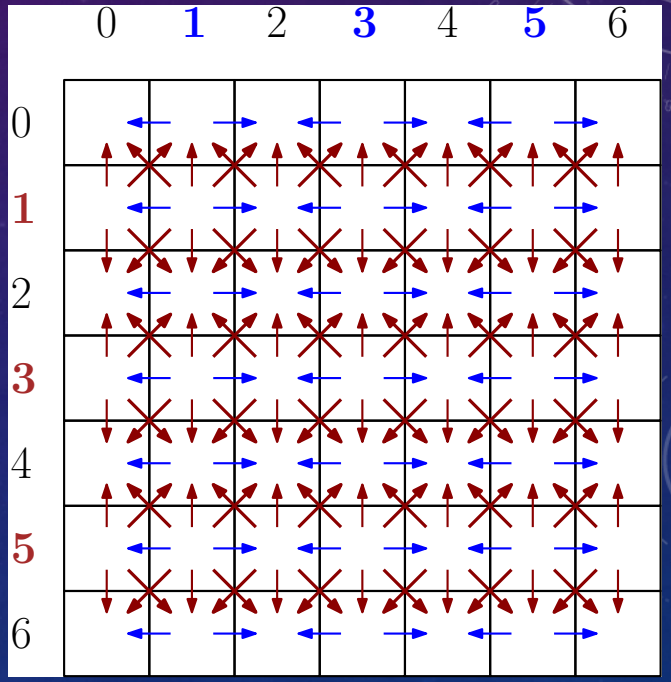
# UNIDIRECTIONAL COMPARISON: TWO DIMENSIONS

- In two dimensions, this is the final pattern of comparisons
- Note that in some cells, there are no searches originating from the cells
- In other cells there are fewer than 9 adjacent cells searched

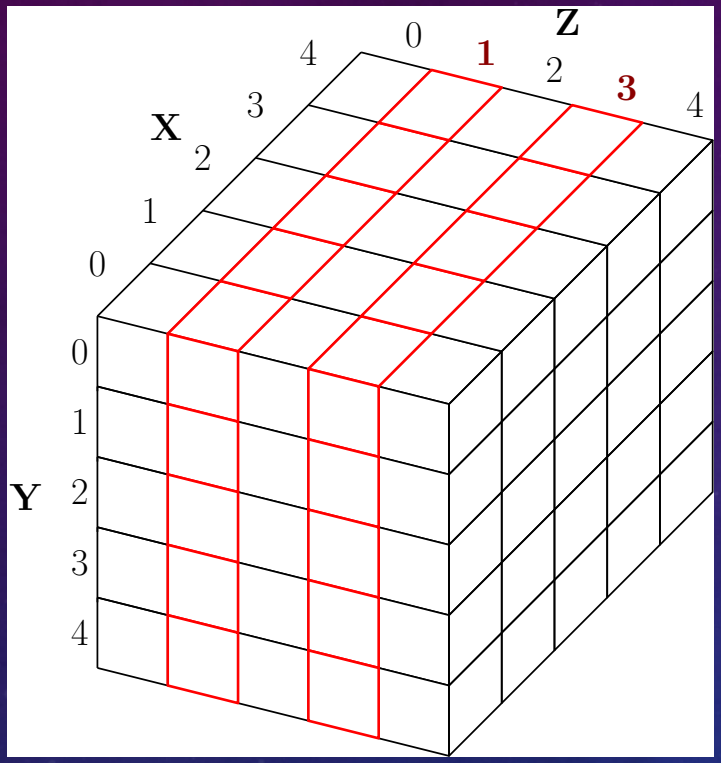


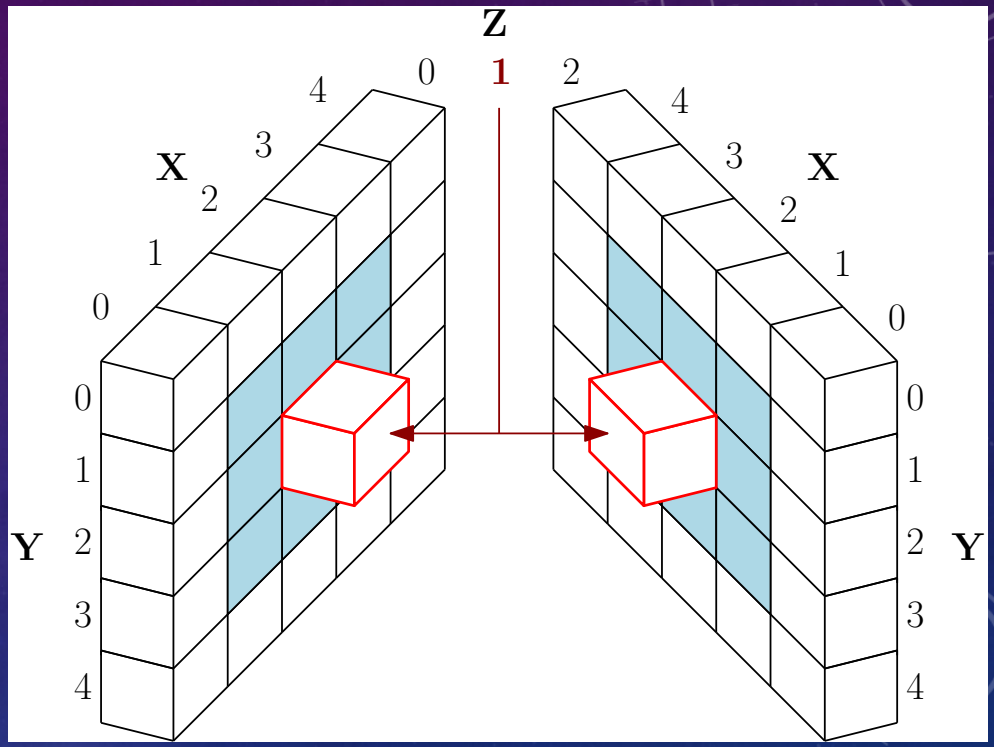
# UNIDIRECTIONAL COMPARISON: TWO DIMENSIONS

- In two dimensions, this is the final pattern of comparisons
- Note that in some cells, there are no searches originating from the cells
- In other cells there are fewer than 8 adjacent cells searched
- Result: Only half of the cells need to be searched on average
  - Reduction in searches
  - Reduction in point comparisons
- Half of the work (with additional overhead for executing the pattern)



# UNIDIRECTIONAL COMPARISON: 3 DIMENSIONS

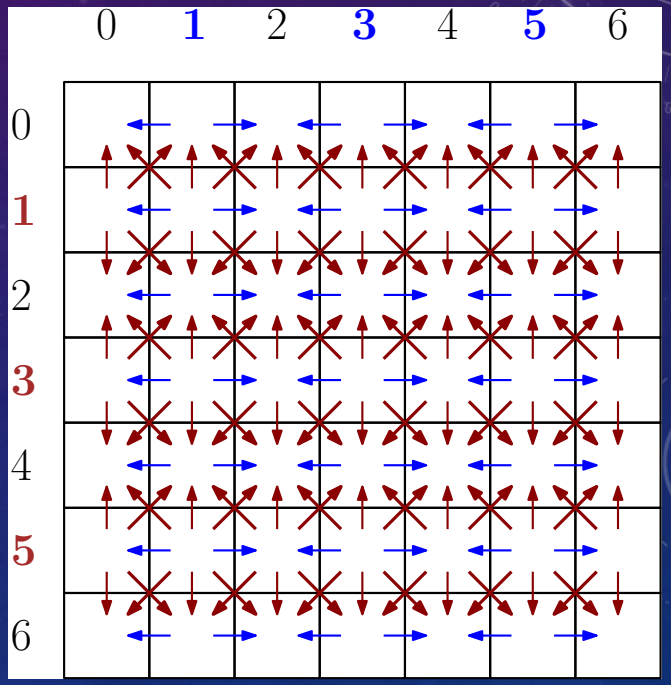



- Left: Same pattern as 2-D, cells in the odd z-coordinate are compared to adjacent cells
- Right: Can see the 9 cells shaded in blue that are compared with the cell with z-coordinate 1 (red outline)

# UNIDIRECTIONAL COMPARISON

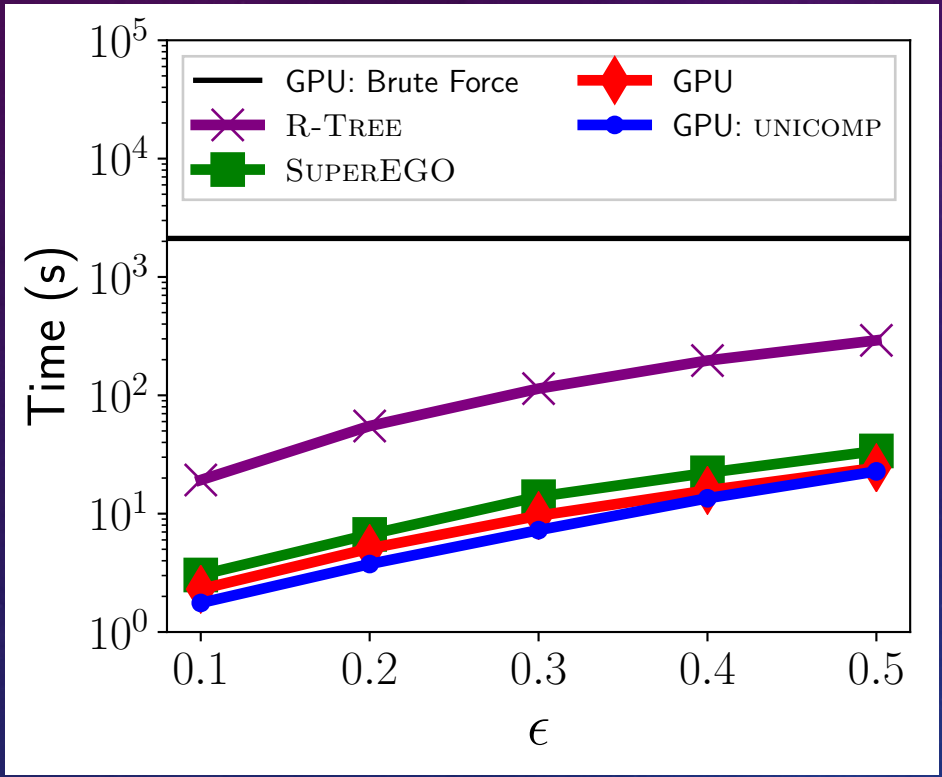
- The pattern can be applied to *n*-dimensions
- Reduces the number of cells searched and point comparisons by half



# EXPERIMENTAL EVALUATION

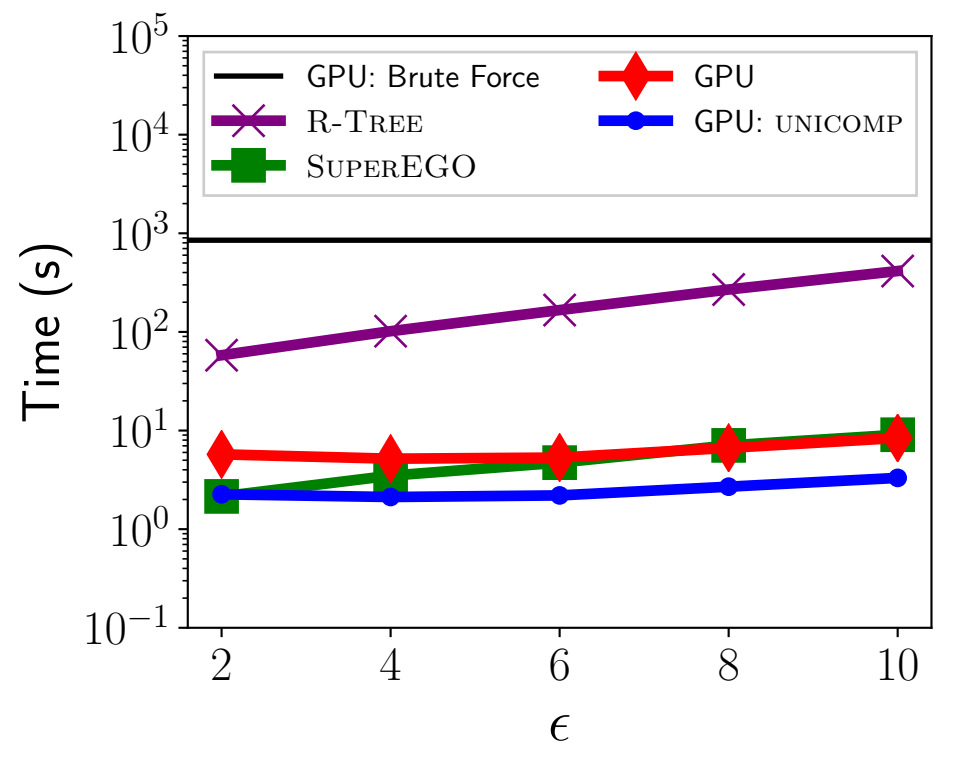
- Experiments performed on up to 32 cores of Intel E5-2683 v4 2.1 GHz CPUs
- Comparison implementations:
  - CPU-only sequential algorithm that uses an R-tree index
  - GPU Brute Force
  - *State-of-the-art:* Parallel SuperEgo algorithm (32 threads/cores)
- Host code in C++ -O3 compiler optimization flag
- GPU: NVIDIA Titan X, CUDA
- Our GPU implementation uses 64-bit floats
- SuperEGO executed with 32-bit floats

# RESPONSE TIME VS. *EPSILON*



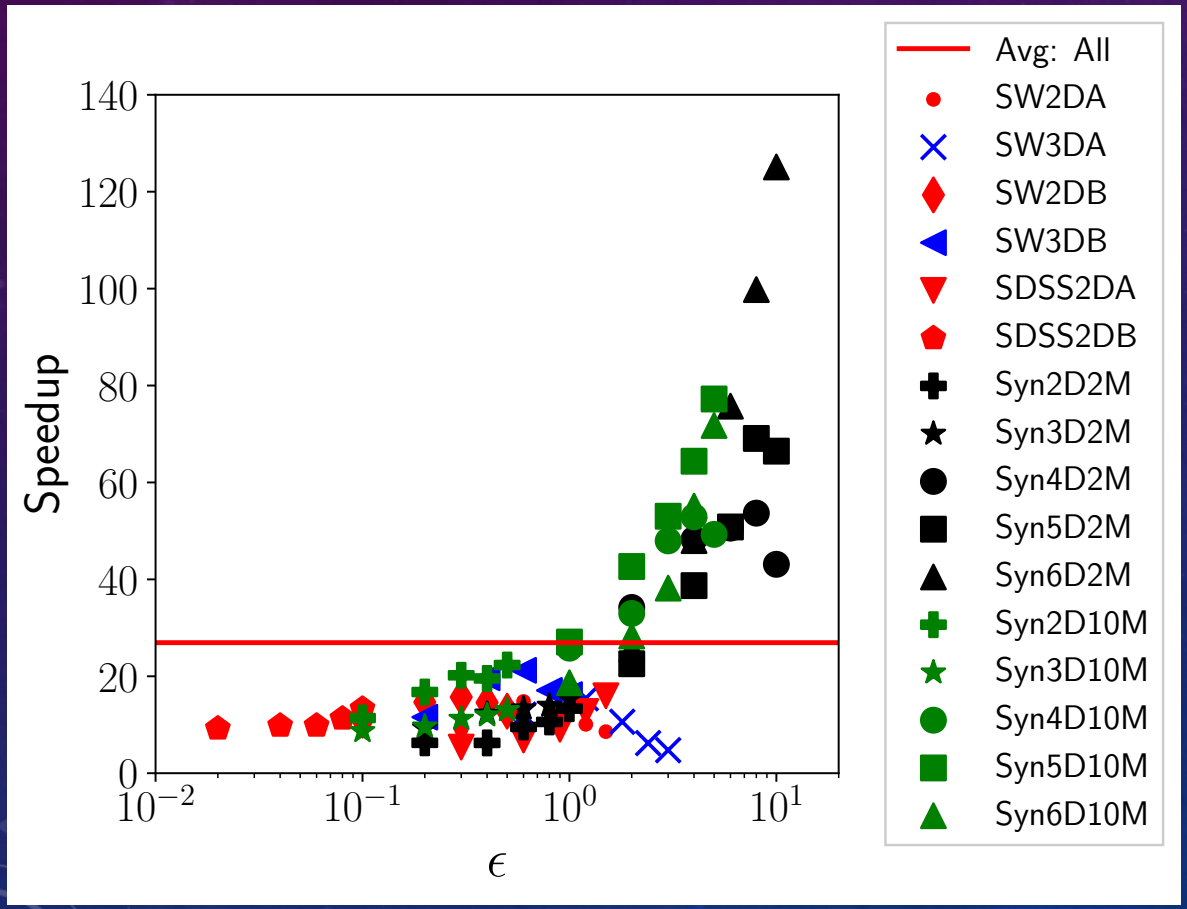

- 2-D Space Weather dataset
- Roughly 5 million points
- Unicomp yields a minor performance gain over executing GPU-SJ without the optimization

# RESPONSE TIME VS. *EPSILON*



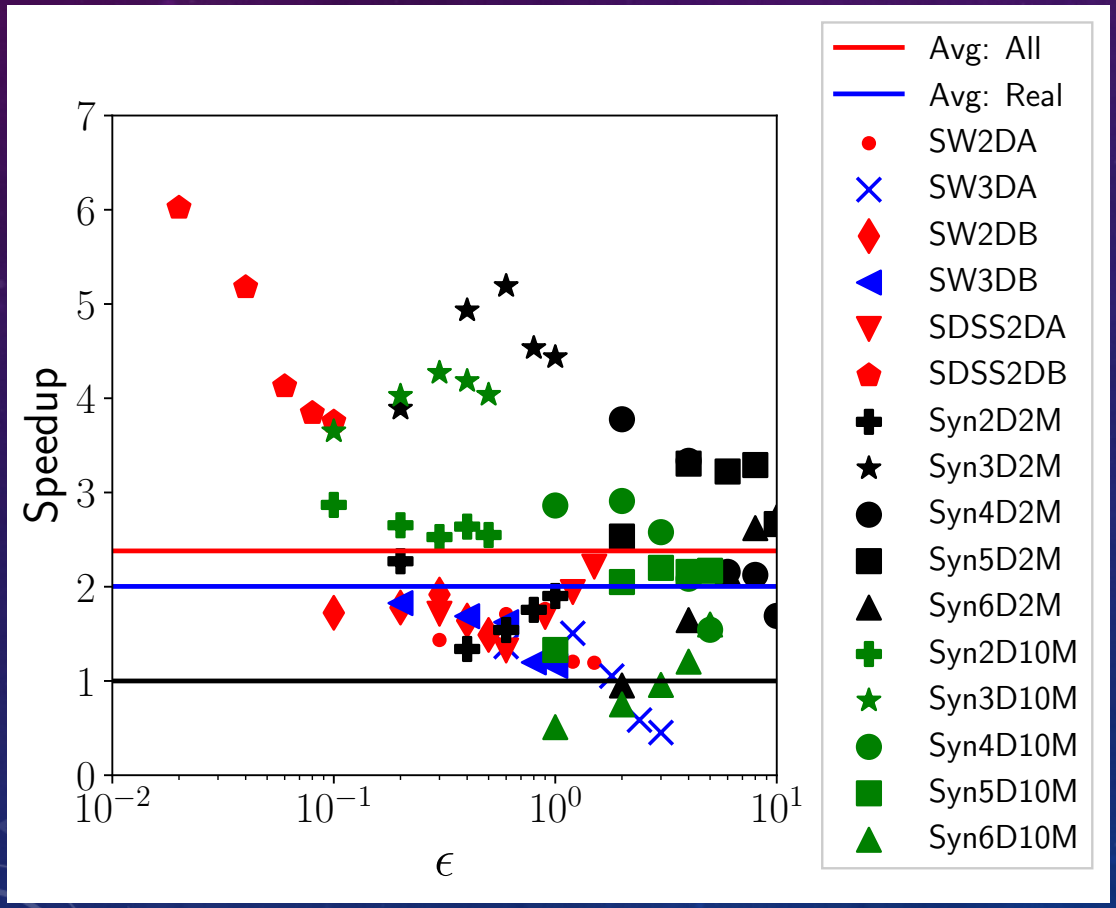

- 6-D Synthetic Dataset (Uniform Distribution)
- 2 million points
- Unicomp yields a larger performance gain over executing GPU-SJ without the optimization

# RESPONSE TIME VS. *EPSILON: R-TREE*




- Summary plot of speedup over the CPU sequential R-tree implementation on 16 datasets
- Speedup greatest for higher dimensions
- Average speedup: 26.9x

# RESPONSE TIME VS. *EPSILON: SUPEREGO*

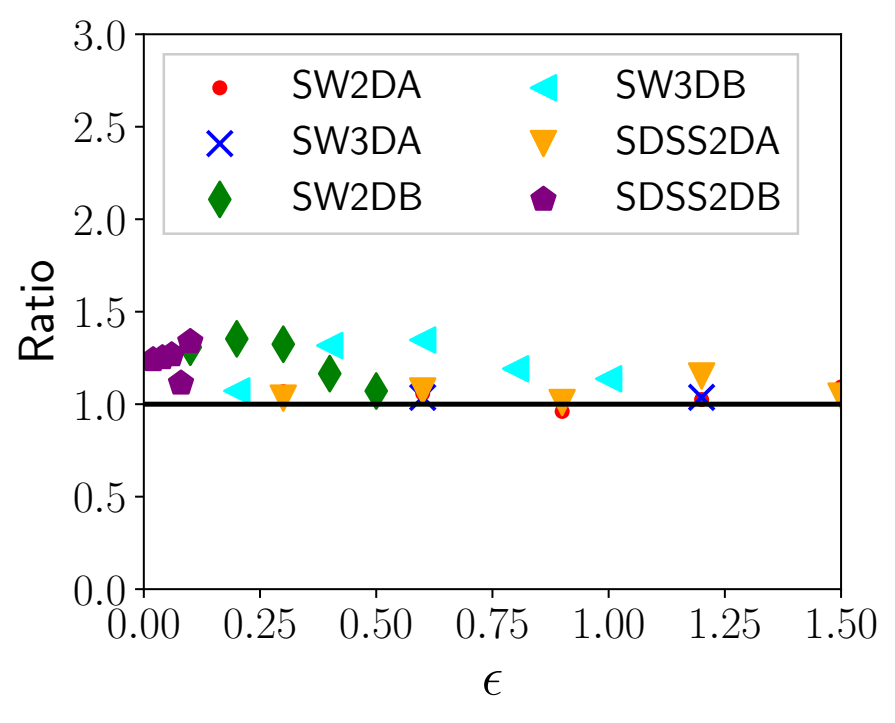


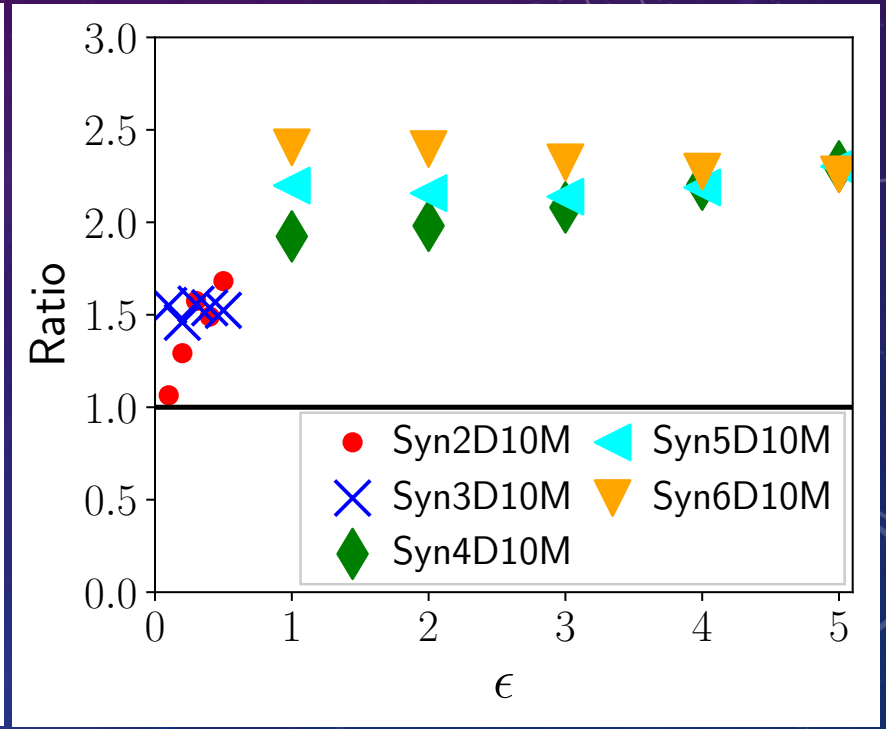

- Summary plot of speedup over the parallel SuperEGO algorithm
- Average speedup over *state-of-the-art*: 2.38x

# PERFORMANCE CHARACTERIZATION OF UNICOMP

- Our Unicomp optimization reduces the total number of cells searched
  - Half of the number of searched cells
- Under what scenarios is Unicomp effective?

# RATIO TIME WITHOUT/WITH UNICOMP




- Real world datasets (2-3-D)
- Synthetic Datasets (2-6-D)

- Performance of UNICOMP dependent on dimensionality
- Achieves >2x speedup in some scenarios
  - Surprising result

# CONCLUSIONS

- The self-join is a widely used operation in the database community
- We propose a GPU accelerated algorithm to perform the self-join on low dimensional data
- We utilize an index tailored for the GPU with a small memory footprint
- Our approach outperforms the CPU parallel state-of-the-art algorithm

# FUTURE WORK

- Examine the high-dimensional case
- Use the self-join as a building block for other algorithms, such as kNN