GEOS/JTS, GDAL/OGR, GRASS, PostGIS/PostgreSQL
ArcMap/ArcGIS, Oracle, SQLServer ……

GPU-Accelerated Spatial and Trajectory Data Management

Spatial Data Management
(Spatial Databases)

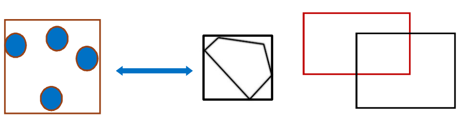
Spatial Indexing

Spatial Join

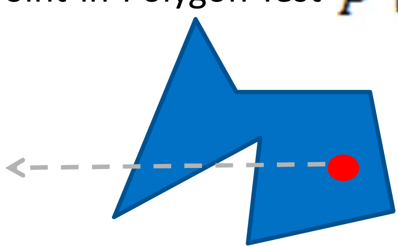
Filtering phase

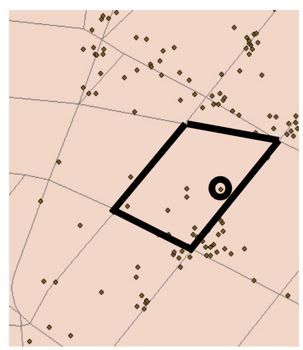
Refinement phases

MBBs



Point-in-Polygon Test $P \bowtie_{PIP} Q$

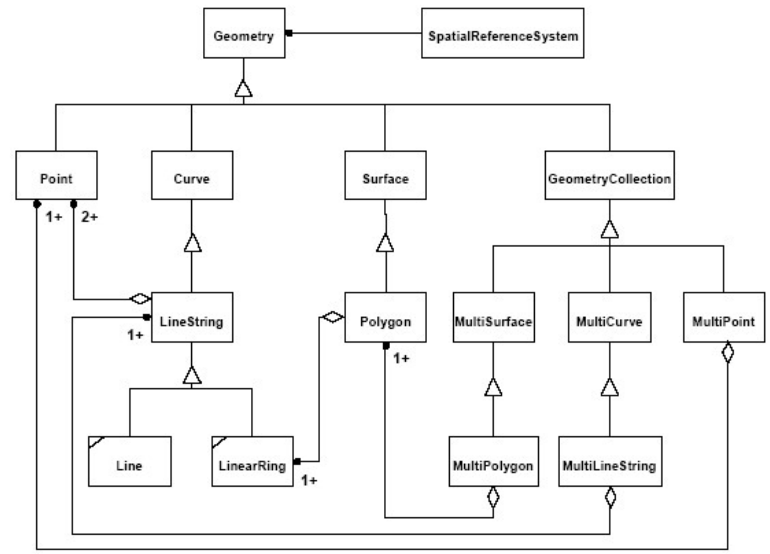




GDAL/OGR: OGRGeometry.**Contains**(OGRGeometry)

SQL: **ST_WITHIN**(Point.geometry, Polygon.Geometry)

OGC Simple Features Specification (SFS) for SQL



Methods for Geometry:

Basic Methods

Dimension, GeometryType, SRID, Envelope, AsText, AsBinary, IsEmpty, IsSimple, Boundary

Methods for testing Spatial Relations

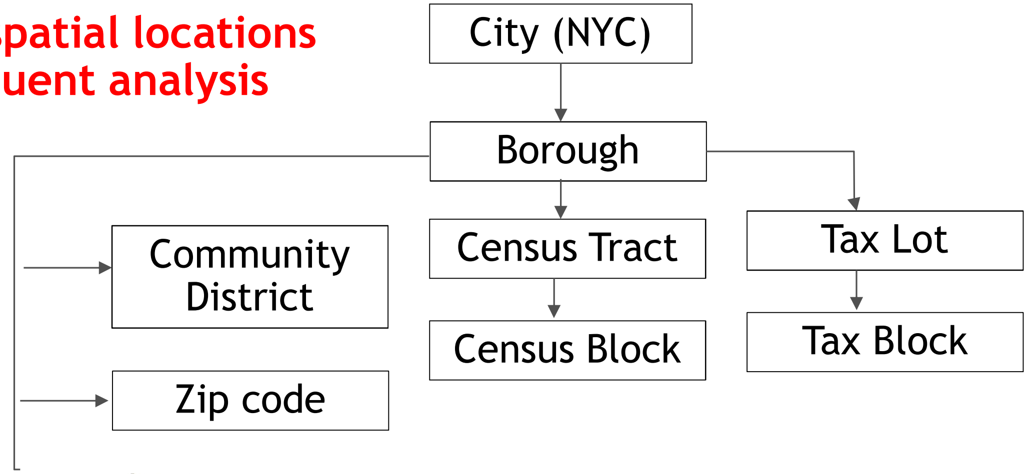
Equals, Disjoint, **Intersects**, Touches, Crosses, **Within**, **Contains**, Overlaps, Relate [more general]

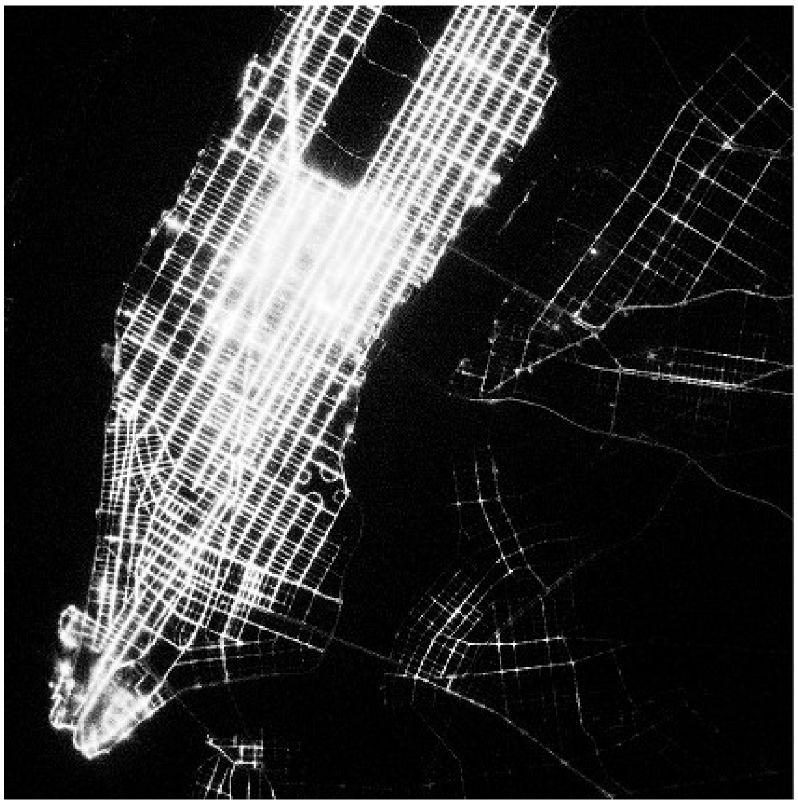
Methods that support Spatial Analysis

Distance, Buffer, ConvexHull, Intersection, Union, Difference, SymDifference

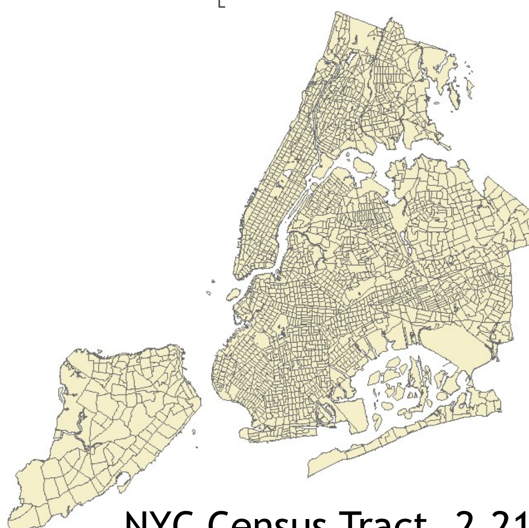
Point-In-Polygon Tests from real world applications

- **Polygons are the semantic framework of geospatial locations**
- **PIP gives points semantic meaning for subsequent analysis**





NYC Taxi Trip Pickup Locations
168,898,952 in 2009



NYC Census Tract, 2,216 polygons
(2,411 rings, 170,329 vertices)

For more NYC polygon datasets, see

https://www1.nyc.gov/site/planning/data-maps/open-data/districts-download-metadata.page
And
https://opendata.cityofnewyork.us/

Existing PIP Functionality in cuSpatial

{cuspatial}/python/cuspatial/cuspatial/tests/test_pip.py

```
def test_one_point_in():
    result = cuspatial.point_in_polygon_bitmap(
        cudf.Series([0]),
        cudf.Series([0]),
        cudf.Series([1]),
        cudf.Series([3]),
        cudf.Series([-1, 0, 1]),
        cudf.Series([-1, 1, -1]) )
```

{cuspatial}/python/cuspatial/cuspatial/core/gis.py
cpp_point_in_polygon_bitmap

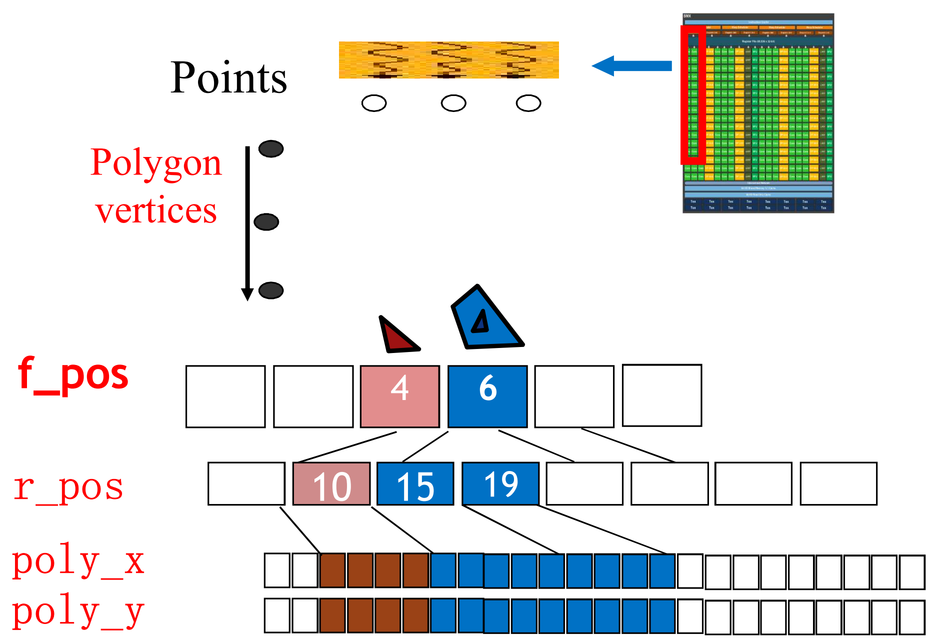
{cuspatial}/python/cuspatial/cuspatial/_lib/spatial.pyx
cpp_point_in_polygon_bitmap

{cuspatial}/cpp/src/spatial/point_in_polygon.cu

```
gdf_column point_in_polygon_bitmap(
const gdf_column & points_x, const gdf_column& points_y,
const gdf_column& poly_fpos, const gdf_column& poly_rpos,
const gdf_column& poly_x, const gdf_column& poly_y)
```

Assumptions:

- A point can be in any of the polygons ➔ use integer as bitvector for bitmap rep.
- The number of polygons is no more than sizeof(uint8_t/uint16_t/**uint32_t**)
- A thread processes a point and loops over multiple polygons
- No indexing on points
- no polygon BBox approximation
- Considered as a spatial function



Motivations for Spatial Indexing and PIP-based Spatial join

- Bitmap-based PIP result rep. is not suitable for large numbers of polygons
- Limiting a point to be in no more than one polygon is not always practical
- Brute force PIP test between every point and every polygon is not efficient

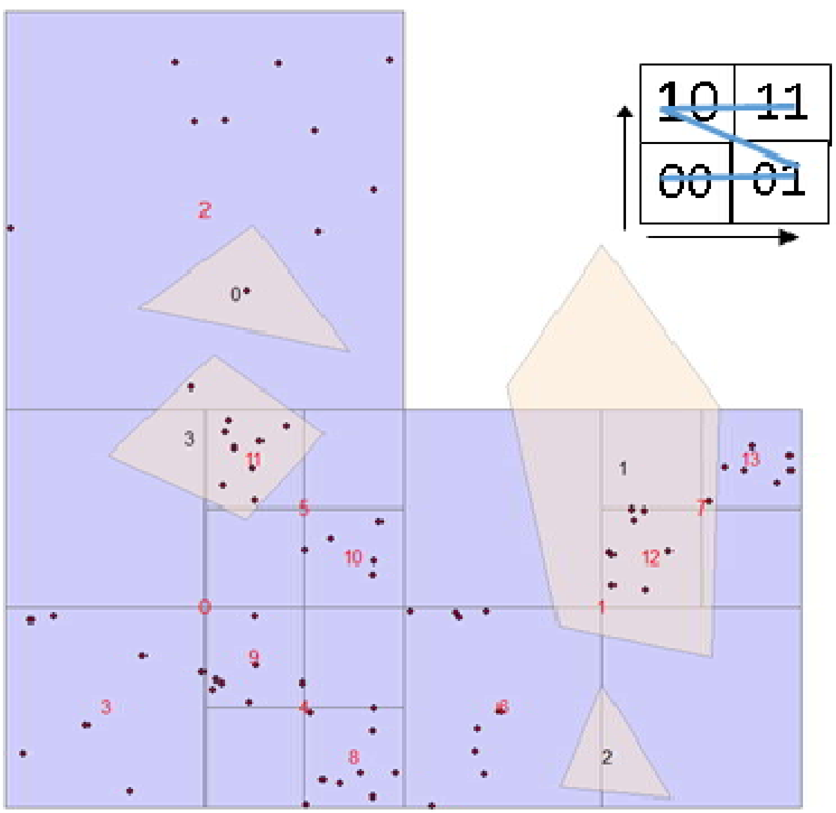
Proposed solution: Apply Indexed Spatial Join Framework

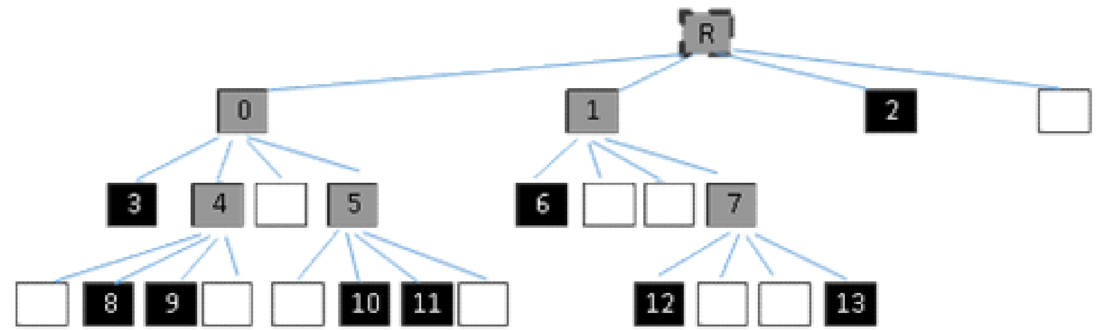
- High-performance indexing on large-number of points
- Using BBoxes to approximate complex polygons
- Paring point indices (quadrants for quadtree indexing) with polygon bboxes based on simple rectangle intersection test for effective pruning search space
- Performing PIP tests only for points in quadrants that are paired with polygons

Major Challenges

- (Hierarchical) spatial indexing involves irregular data accesses and high data movements
- Point-polygon paring has many-to-many relationship ➔ output sizes are data dependent

Quadtree Indexing for Large-Scale Point Data





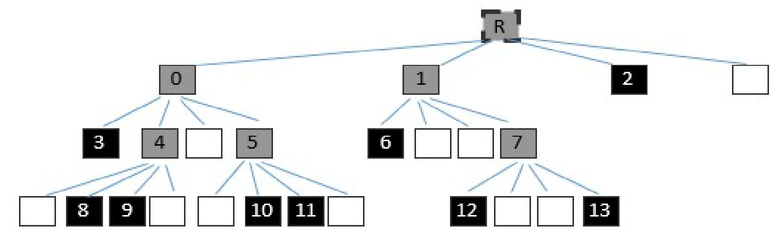
Recursively divides indexing space into four quadrants:

- White node: no points are distributed in the quadrant →stop
- Black node: the number of points in the quadrant is below a pre-defined threshold → stop
- Gray node: the number of points in the quadrant is above the threshold → recursive sub-division until reaching black nodes or the maximum level is reached.

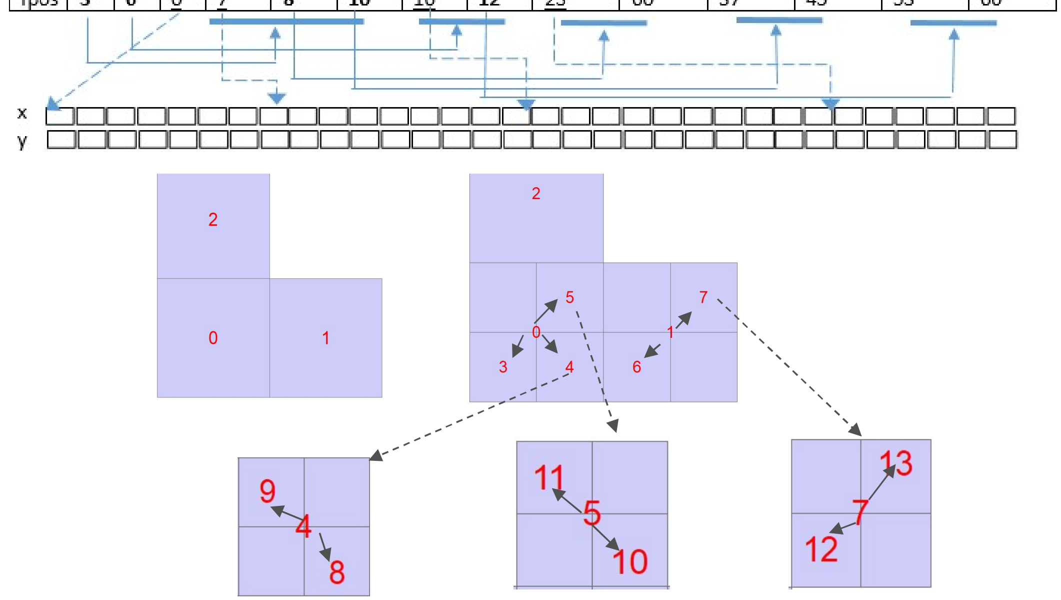
- Reduce searching from linear scanning to logarithmic (assuming reasonable balancing)
- Low overheads for implementations, better suitable for in-memory data processing (comparing with R-Trees)

Quadtree Indexing for Large-Scale Point Data

- Novel Structure of Array (SoA) design for GPUs
- Data parallel implementations using parallel primitives (sort/reduce/transform/gather/scatter…)
- Five arrays to encode resulting quadtree:
  - key (32b): Z-order or Morton code at a level
  - lev (8b): level (<16)
  - sign (1b): needs further division (non-leaf node) or not (leaf node)
  - len (32b): number of child nodes (for non-leaf node) or number of points (for leaf nodes)
  - fpos(32b): first child node position (for non-leaf node) or first point position (for leaf node)
- Level-by-level Breadth-First Search (BFS)
- Access to child nodes using array offsets
  - Node/point position= fpos[p]+i
  - no pointer chasing

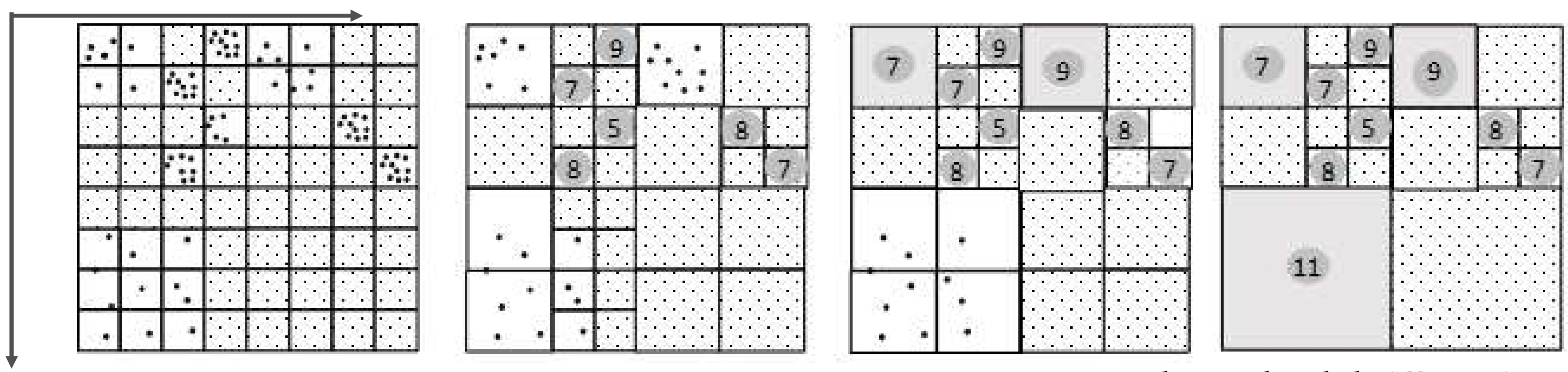


Seq	0	1	2	3	4	5	6	7	8	9	10	11	12	13
Key	00	01	10	0000	0010	0011	0100	0111	001001	001010	001101	001110	011100	011111
Lev	0	0	0	1	1	1	1	1	2	2	2	2	2	2
Sign	1	1	0	0	1	1	0	1	0	0	0	0	0	0
len	**3**	**2**	7	9	**2**	**2**	7	**2**	9	5	8	8	7	11
fpos	**3**	**6**	0	7	**8**	**10**	16	**12**	23	60	37	45	53	60



Quadtree Indexing for Large-Scale Point Data

- http://www.adms-conf.org/2019-camera-ready/zhang_adms19.pdf
- Compute all non-empty quadrants at the finest level in phase 1
- construct quadtree by removing non-qualified nodes in phase 2
- **Require sorting points only once at the finest level**



- **A quadtree node is a leaf node if:**
  - **$n_k > n_t$ AND at finest level**
  - **$n_k <= n_t < n_p \Rightarrow n_k <= n_t$ AND $n_p > n_t$**

•n_t: threshold (# points)
•n_k: #of points at node k
•n_p: #of points at the parent node of node k

Quadtree Indexing for Large-Scale Point Data

Phase 1

Transform point dataset ***P*** to key set ***l_key*** using Z-ordering at finest level
Sort_by_key using ***l_key*** as the key and ***P*** as the value
Reduce_by_key to count numbers of points in partitioned quadrants and set ***nlen***
t_key←**l_key**
for k=0, ***max_level***-1
    (*t_key*,*lev*)←transform(*t_key*) …(*t_key*[k]/=4)
    (***pkey***, ***clen***)+=*reduce_by_key*(*t_key*)

Phase 2 (next slide):

(sign,length,fpos)=genValidQuadrants (pkey,clen,nlen, n_t)

Inputs: ***pkey***: array of Morton codes of quadrants; ***clen***: numbers of non-empty sub-quadrants; ***nlen***: array of the numbers of points in these quadrants; $\mathbf{n_t}$**:** #of points threshold

Output: **indicator,** ***f_pos***

Algorithm **genValidQuadrants**

1 *tpos* $\leftarrow$ exclusive_scan(***clen***)

2 *tmap* $\leftarrow$ scatter ([0..|*clen*|],*tpos*)

3 *tmap* $\leftarrow$ inclusive_scan(*tmap*, *maximum*)

Computing parent node offsets for access in Step 5

4 *tlen* $\leftarrow$ ***clen*** //tlen is a copy of clen to avoid modification due to remove_if in the next step

5 {***pkey***,**clen**,**nlen**,tmap} $\leftarrow$ remove_if*(*{**pkey**,**tlen**, **nlen**,tmap}, (**nlen,** $\mathbf{n_t}$))

6 **indicator** $\leftarrow$ transform(**clen**,($\boldsymbol{n_t}$))

Remove unqualified nodes if #of points in their parent nodes are less than n_t

Determines leaf node

7 *nlen* $\leftarrow$ replace_if*(**nlen**, **indicator,0**)*

8.(**revision**) **reorder leaf nodes** based on their z-order codes at the finest level, compute **prefix-sum for ppos** and then **restore** the original order

Adjust nlen/clen to prepare for computing pos

9 clen $\leftarrow$ replace_if*(**clen**, ~ **indicator, 0**)*

10 cpos $\leftarrow$ exclusive_scan*(clen)*

11.1 ***length*** $\leftarrow$ transform*({nlen,clen},* ***indicator)***

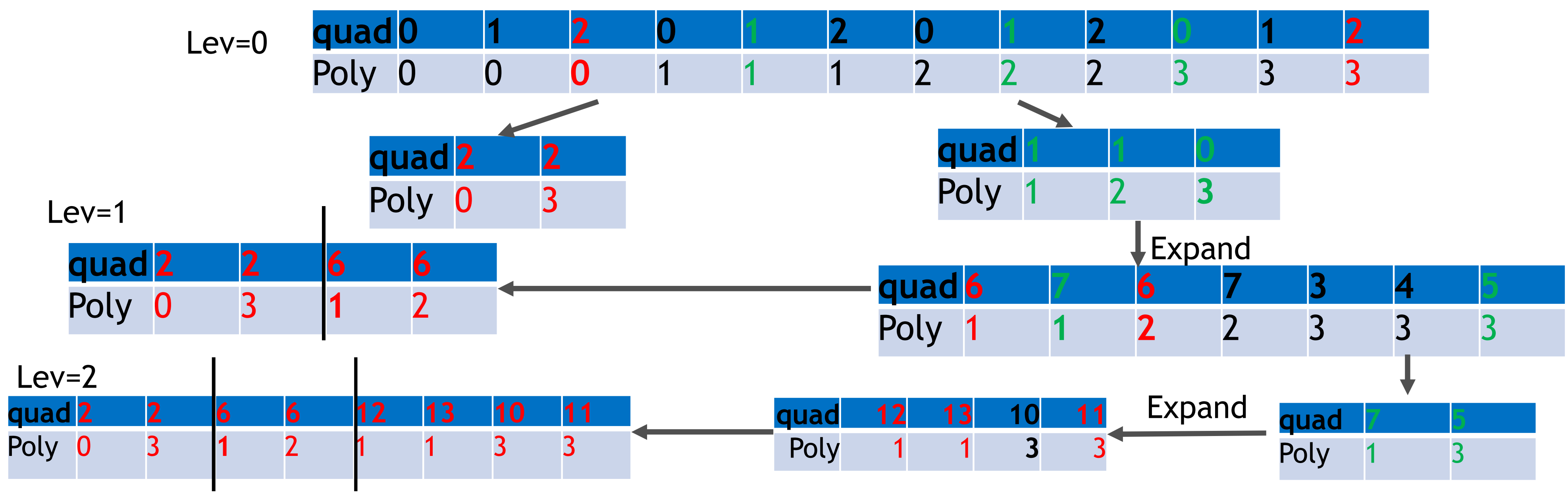
11.2 ***f_pos*** $\leftarrow$ transform*({ppos,cpos},* ***indicator)***

Filling f_pos

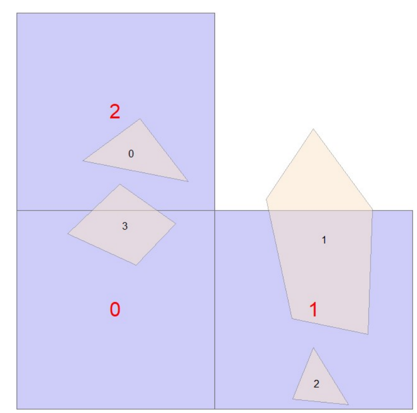
Quadrant and Polygon Intersection Test for Spatial Filtering

- **Basic idea**: If the quadrant that a point falls within does not intersect with the bounding box of a polygon, the point can not be within the polygon.
- The spatial filtering step pairs point quadrants and polygon bboxes for later spatial refinement step that actually test whether a point is inside a polygon.
- The paring process starts at the top level of a quadtree and pair quadrant and polygon **level-by-level**.
- All quadrants at a level are compared against all polygon bboxes. Three cases:
  - Quadrant does not intersect with polygon bbox ➔ **discard**
  - Quadrant intersect with polygon bbox and the corresponding quadtree node is a leaf node: ➔copy the pair to output array, **no further expansion**
  - Quadrant intersect with polygon bbox and the corresponding quadtree node is a non-leaf node➔ copy the pair to the expansion array, **expand it based on fpos and length**, and get ready for next level iteration

Quadrant and Polygon Intersection Test for Spatial Filtering

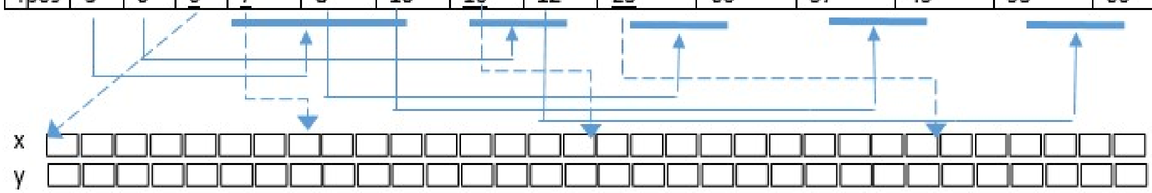


Seq	0	1	2	3	4	5	6	7	8	9	10	11	12	13
Key	00	01	10	0000	0010	0011	0100	0111	001001	001010	001101	001110	011100	011111
Lev	0	0	0	1	1	1	1	1	2	2	2	2	2	2
Sign	1	1	0	0	1	1	0	1	0	0	0	0	0	0
len	**3**	**2**	7	9	**2**	**2**	7	**2**	9	5	8	8	7	11
fpos	**3**	**6**	0	7	**8**	**10**	16	**12**	23	60	37	45	53	60

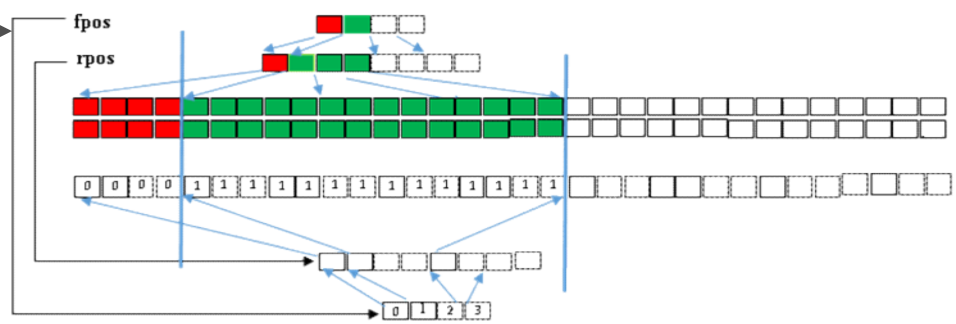


PIP-based Spatial Refinement

Seq	0	1	2	3	4	5	6	7	8	9	10	11	12	13
Key	00	01	10	0000	0010	0011	0100	0111	001001	001010	001101	001110	011100	011111
Lev	0	0	0	1	1	1	1	1	2	2	2	2	2	2
Sign	1	1	0	0	1	1	0	1	0	0	0	0	0	0
len	3	2	7	9	2	2	7	2	9	5	8	8	7	11
fpos	3	6	0	7	8	10	16	12	23	60	37	45	53	60

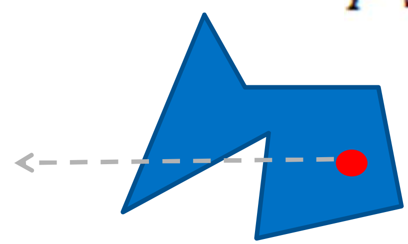


quad	2	2	6	6	12	13	10	11
Poly	0	3	1	2	1	1	3	3



Ray casting algorithm

$$P \bowtie_{PIP} Q$$



```
bool pnpoly(int npol, float *xp, float *yp, float x, float y)
{
    int i, j, c = 0;
    for (i = 0, j = npol-1; i < npol; j = i++) {
      if ((((yp[i] <= y) && (y < yp[j])) ||
           ((yp[j] <= y) && (y < yp[i]))) &&
          (x < (xp[j] - xp[i]) * (y - yp[i]) / (yp[j] - yp[i]) +
xp[i]))
        c = !c;
    }
    return c;
}
```

https://wrf.ecse.rpi.edu//Research/Short_Notes/pnpoly.html

The even-odd rule works for any closed set of polygons (polygons with multiple rings; polygons with holes…)

PIP-based Spatial Refinement

- Each quadrant-polygon pair is assigned to a thread block
- Each thread processes a point within a thread block
- All threads loop through all vertices of the polygon ➔coalesced memory access
- # of points in leaf-quadrants can be larger than the threshold (even if the threshold can be limited by the maximum number of threads in a block)➔ **decompose large quadrants**

Assuming minimum size for non-leaf nodes (threshold) is 512 and #of threads per block is 256

qsz	348	348	233	233	1000	375	432	212
quad	2	2	6	6	12	13	10	11
Poly	0	3	1	2	1	1	3	3

size	256	92	256	92	233	233	256	256	256	216	256	119	256	176	212
offset	0	256	0	256	0	0	0	256	512	784	0	256	0	256	0
quad	2	2	2	2	6	6	12	12	12	12	13	13	10	10	11
Poly	0	0	3	3	1	2	1	1	1	1	3	3	3	3	3

PIP-based Spatial Refinement

- Many-to-many relationship among quadrants (and their points) and polygons
- Output (point-polygon pairs) size is data dependent and can not pre-determined
- **Two-phase strategy**: count # of point-polygon pairs in phase 1 and write the actual output in phase 2➔run PIP twice.

Phase1

```
bool in_polygon= pip_test (…)
unsigned mask = __ballot_sync(0xFFFFFFFF, threadIdx.x < num_point);
uint32_t vote=__ballot_sync (mask, in_polygon);
 if(threadIdx.x%num_threads_per_warp==0)
        data[threadIdx.x/num_threads_per_warp]=__popc(vote);
   __syncthreads();
if(threadIdx.x<max_warps_per_block)
{ uint32_t num=data[threadIdx.x];
  for (uint32_t offset = max_warps_per_block/2; offset > 0; offset /= 2)
    num += __shfl_xor_sync(0xFFFFFFFF,num, offset);
  if(threadIdx.x==0)
     num_hits[blockIdx.x]=num; }
```

Combing ballot and popc to count # of points in polygon in a complete warp

Warp-level reduction (for no more than 1024 threads per block)

reduce (sum) on num_hits will give #of output point-polygon pair
scan(prefix-sum) on num_hits will give the offset to output for each block

PIP-based Spatial Refinement

- Pre-processing: sub-quadrant and polygon pairs may have zero point-polygon pairs and need to be removed➔remove_if
- **Phase 2**

```
….
if(threadIdx.x<num_threads_per_warp) {
    uint32_t num=data[threadIdx.x];
    for (uint8_t i=1; i<=num_threads_per_warp; i*=2) {
      int n = __shfl_up_sync(0xFFFFFFF,num, i, num_threads_per_warp);
      if (threadIdx.x >= i) num += n; }
      sums[threadIdx.x+1]=num;
    __syncthreads(); }
 __syncthreads();
if((threadIdx.x<num_point)&&(in_polygon)) {
    uint16_t num=sums[threadIdx.x/num_threads_per_warp];
    uint16_t warp_offset=__popc(vote>>(threadIdx.x%num_threads_per_warp))-1;
    uint32_t pos=d_num_hits[blockIdx.x]+num+warp_offset;
    d_res_poly_idx[pos]=pq_poly_idx[blockIdx.x];
    d_res_pnt_idx[pos]=qt_fpos[pq_quad_idx[blockIdx.x]]+sub_offset[blockIdx.x]+ threadIdx.x; }
```

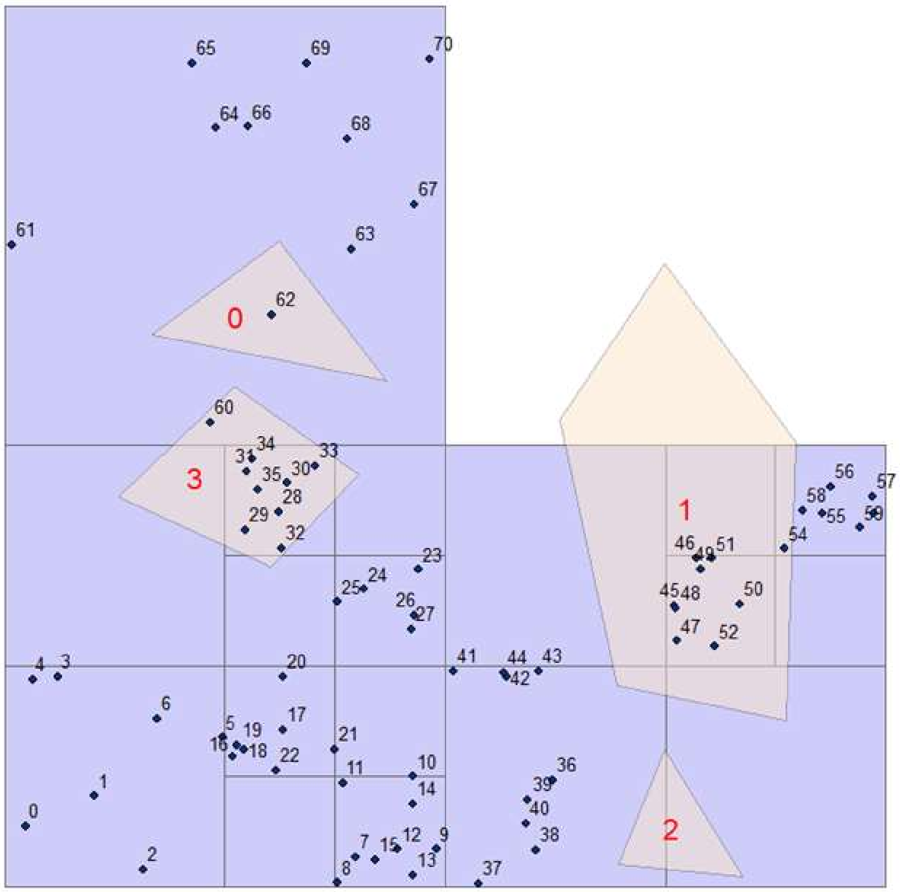
data stores # of positive PIP test per warp (as in Phase 1)

Warp-level scan (for no more than 1024 threads per block)

Output offset for each point-polygon pair (pos) is the sum of quadrant offset, warp offset and thread ID.

Point idx is the sum of first point position in a quadrant, offset within sub-quadrant and thread ID

PIP-based Spatial Refinement



size	11	11	9	9	8	7	5	8
offset	0	0	0	0	0	0	0	0
quad	2	2	6	6	12	13	10	11
Poly	0	3	1	2	1	1	3	3

Phase1

#	1	1	0	0	8	1	0	8

Remove Empty Pairs

1	1	8	1	8

Prefix-sum

0	1	2	10	11

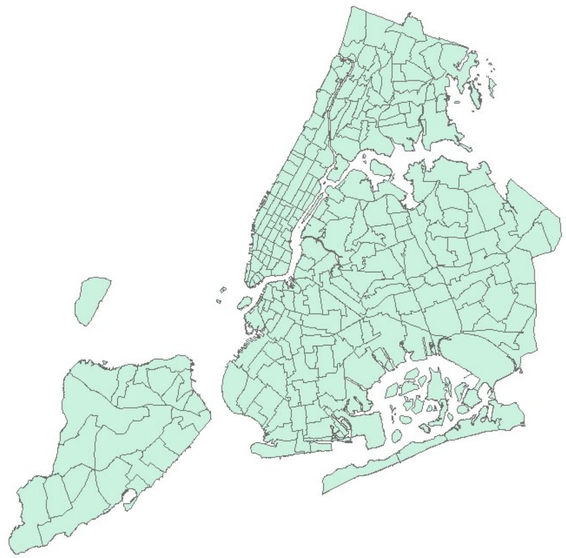
Point-idx	62	60	52	51	50	49	48	47	46	45	54	35	34	33	32	31	30	29	28
Polygon-idx	0	3	1	1	1	1	1	1	1	1	1	3	3	3	3	3	3	3	3

Performance on NYC Taxi Trip Data

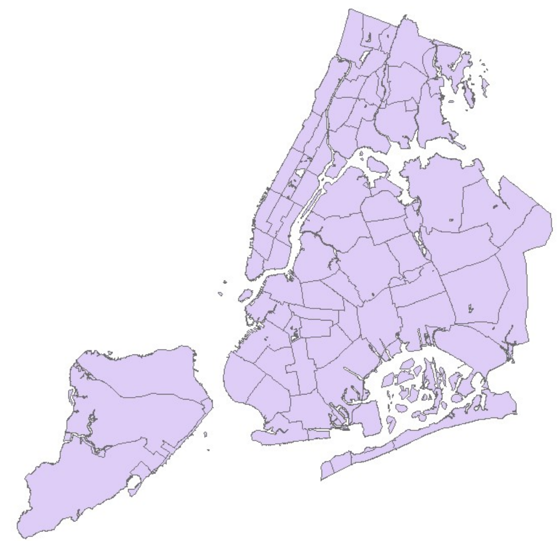
2009 NYC Taxi Tri Pickup Locations

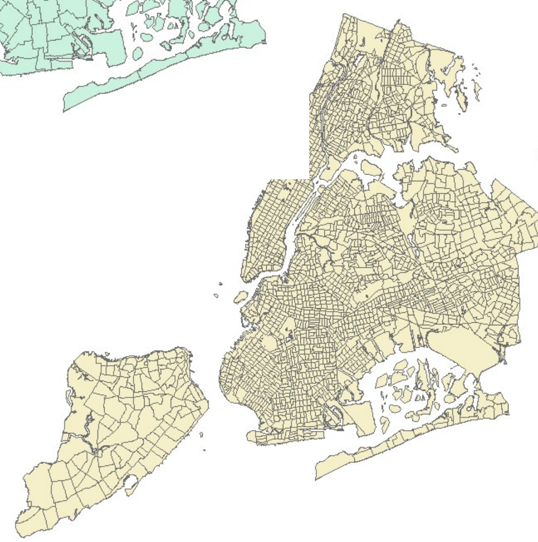
# Mo.	#of points
1	13,887,620
2	27,079,723
3	41,284,081
4	55,383,596
5	69,970,743
6	84,035,490
7	97,553,533
8	111,127,610
9	124,993,700
10	140,444,141
11	154,523,740
12	168,898,952

NYC Taxi Zones, 263 polygons, (354 rings, 98192 vertices)



NYC Community Districts , 71 polygons (243 rings, 100119 vertices)





NYC Census Tract, 2216 polygons (2411 rings, 170329 vertices)

Performance on NYC Taxi Trip Data –Titan V 12GB/i7-7800X

	Description	Taxi Zone	CD	CT
0	# of quad-poly pairs	1,623,908	1,496,150	1,860,074
1	Point Quadtree Indexing runtime (ms)	384.6	384.2	385.2
2	Spatial Filtering runtime (ms)	35.2	31.0	36.3
3	Spatial Refinement runtime (ms)	140.5	552.0	277.3
4	GPU End-to-End runtime (ms)	560.9	967.8	699.3
5	GDAL API on CPU runtime (ms) (10k random points - without indexing)	66,890.9	66,959.4	145,975
6	Speedup1= ([5]/10000)/([4]/168,898,952)	2.01E+06	1.17E+06	3.53E+06

7	GDAL API on CPU runtime (ms) (10000 random quad-poly pairs)	71,143.6	420,970	202,320
8	Speedup2=([7]/10000)/([4]/[0])	2.06E+04	6.51E+04	5.38E+04

Performance on NYC Taxi Trip Data –Titan RTX 24 GB/Intel i7-6700K

	Description	Taxi Zone	CD	CT
0	# of quad-poly pairs	1,623,908	1,496,150	1,860,074
1	Point Quadtree Indexing runtime (ms)	297.7	329.5	294.6
2	Spatial Filtering runtime (ms)	52.9	23.0	59.3
3	Spatial Refinement runtime (ms)	985.9	4299.8	2040.8
4	GPU End-to-End runtime (ms)	1347.0	4658.7	2406.3
5	GDAL API on CPU runtime (ms) (10k random points -without indexing)	63,220.3	62,763.5	142,826
6	Speedup1= ([5]/10000)/([4]/168,898,952)	1.90E+06	1.10E+06	3.45E+06
7	GDAL API on CPU runtime (ms) (10000 random quad-poly pairs)	46,758.9	277,164	140,984
8	Speedup2=([7]/10000)/([4]/[0])	1.35E+04	4.28E+04	3.75E+04