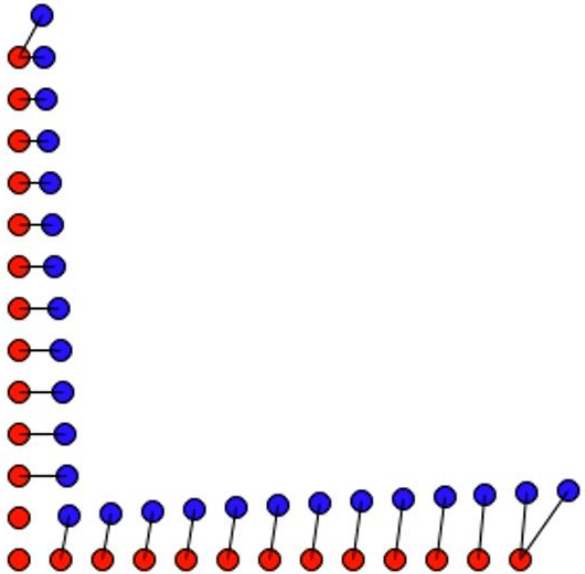
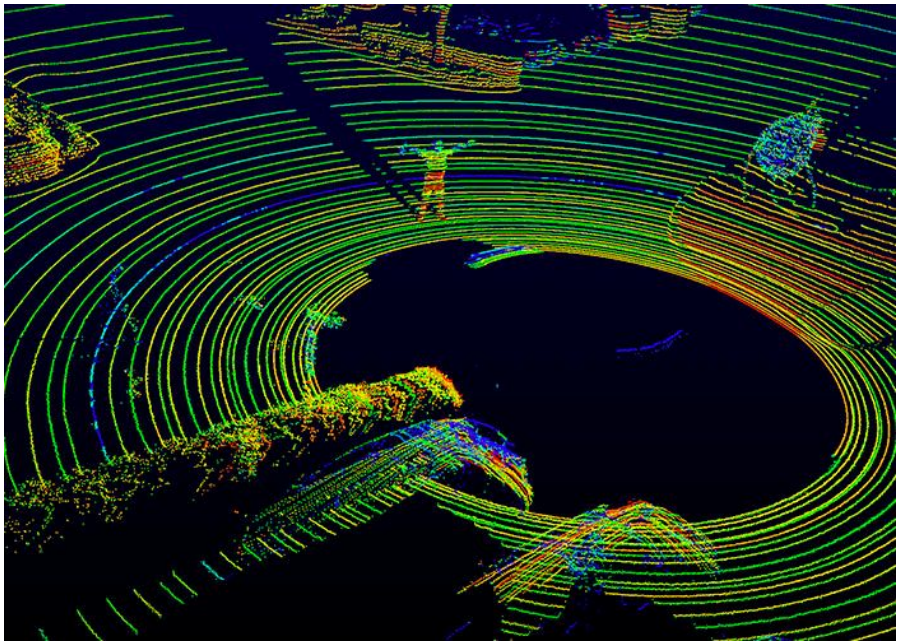
Parallel Nearest Neighbor Search for Velodyne Lidar Point Clouds

Nearest neighbor search is an important step in point cloud matching algorithms



Velodyne Lidar sensors are universally used in autonomous vehicles for mapping and localization



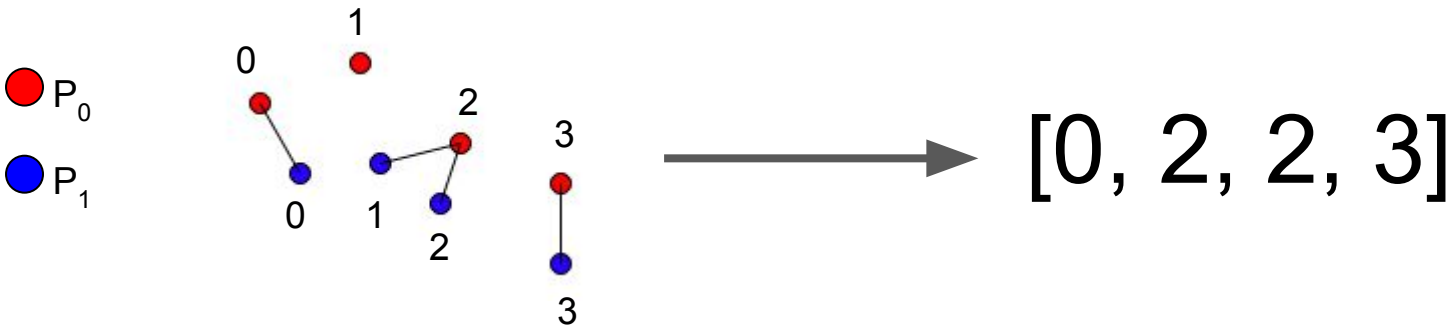
Every point in a point cloud is looking for its nearest neighbor independently: Parallelism!

- 120k points per point cloud
- CUDA

Formally defining the problem

Input: Point clouds P_0, P_1 of size 120k points each

Output: An array, nearestNeighbor, such that nearestNeighbor[i] is the index in P_0 of the nearest neighbor to P_1[i]



Our baseline is the Fast Library for Approximate Nearest Neighbors

- Sequential optimized implementation using k-d trees
- Time taken for 120k queries into a cloud of 120k points:

0.23s

(aim to beat this!)

We wrote a naive sequential implementation

```
Point* cloud0; // N0 points
Point* cloud1; // N1 points

int nearestNeighbor[N1]; // nearestNeighbor[i]:index of the nearest neighbor of cloud1[i] in cloud0
for (int i = 0; i < N1; i++) {
  int minIndex;
  float minDist = FLT_MAX;
  for (int j = 0; j < N0; j++) {
    float distance = computeDistance(cloud0[i], cloud1[j]);
    if (distance < minDist) {
      minDist = distance;
      minIndex = j;
    }
  }
  nearestNeighbor[i] = minIndex;
}
```

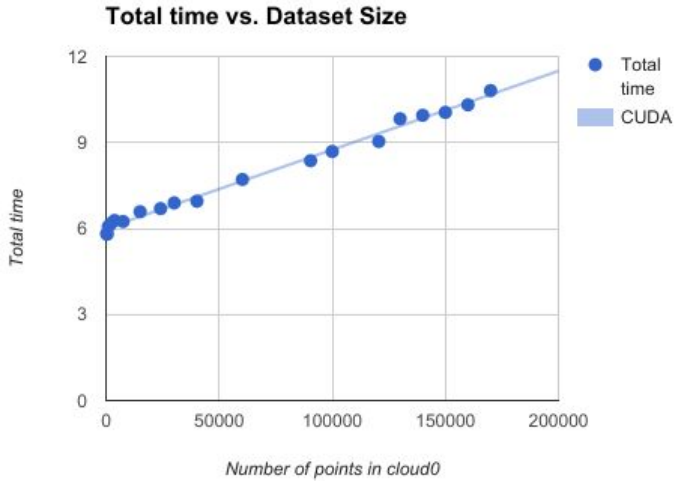
Running time: 62s

Attempt #1: Parallelize over points in cloud0

```
int findNearestNeighbor(Point* cloud0, Point X, int N) {
  __shared__ float distances[N];

  distances[threadID] = computeDistance(X, cloud0[threadID]);

  // Reduce to find smallest distance
  return minReduce(distances, N);
}
```

```
Point* cloud0; // N0 points
Point* cloud1; // N1 points

int nearestNeighbor[N1];
for (int i = 0; i < N1, i++) {
  nearestNeighbor[i] = findNearestNeighbor<<<N0>>>(cloud0, cloud1[i], N0);
}
```

Analysis



Running time: 9.4s

Conclusion: Overhead of kernel launches too high!

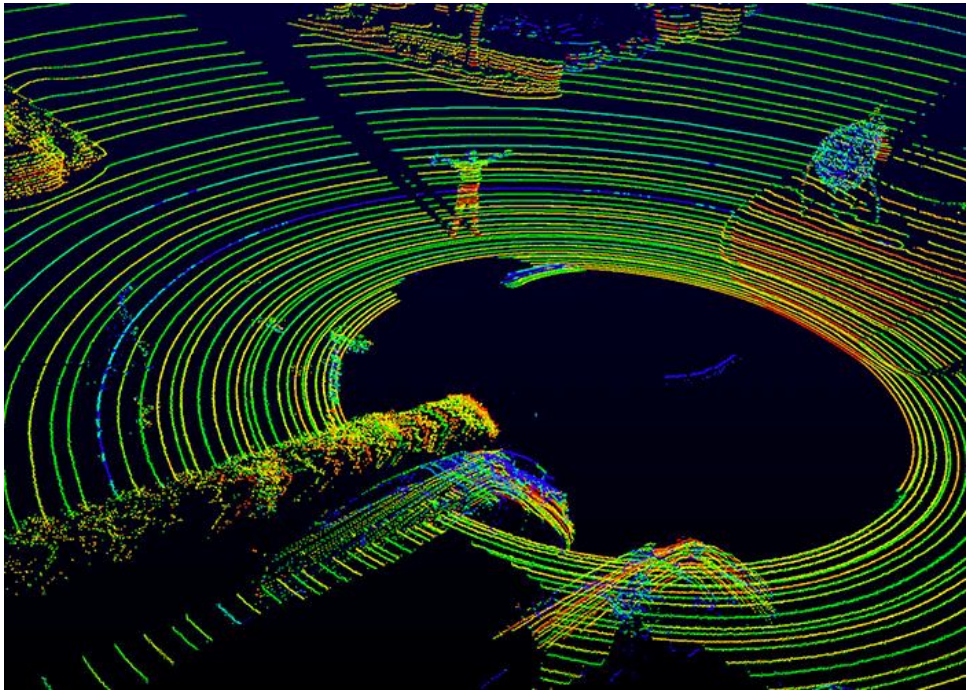
Attempt #2: Move loop inside kernel

```
void findNearestNeighbor(Point* cloud0, Point* cloud1, int N0, int N1, int* nearestNeighbor) {
  __shared__ float distances[N0];

  for (int i = 0; i < N1; i++) {
    distances[threadID] = computeDistance(cloud0[threadID], cloud1[i]);

    // Reduce to find smallest distance
    nearestNeighbor[i] = minReduce(distances, N0);
  }
}


Point* cloud0; // N0 points
Point* cloud1; // N1 points

int nearestNeighbor[N1];
findNearestNeighbor<<<N1>>>(cloud0, cloud1, N0, N1, nearestNeighbor);
```

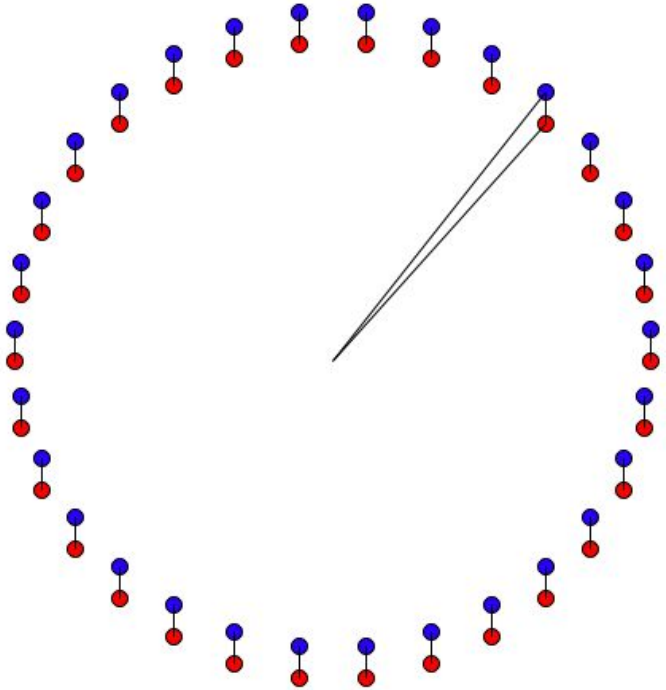
Running time: 2.0s

Can we do better?

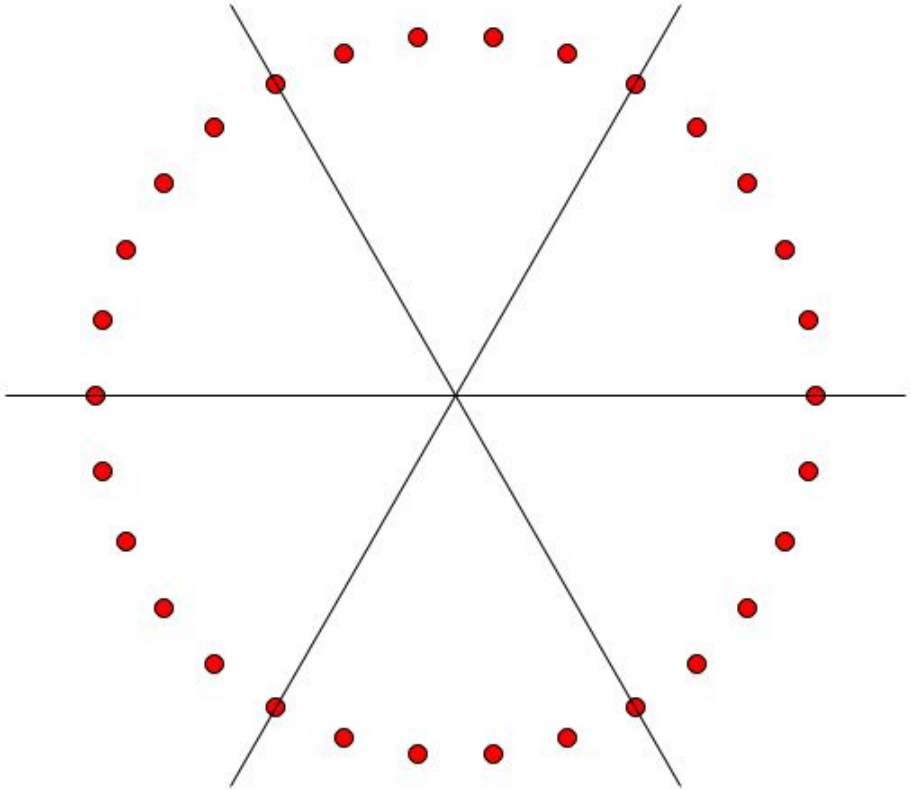
Velodyne lidar point clouds have a very specific 3D structure. Could we reduce the search space?



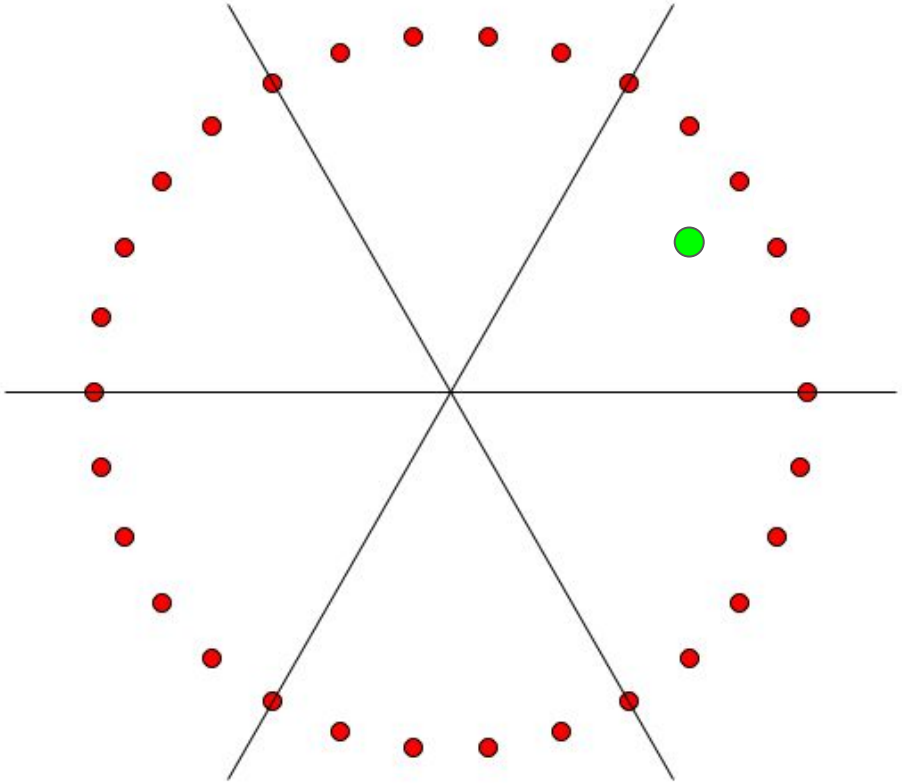
99% of nearest neighbors are within 2.5º in azimuth



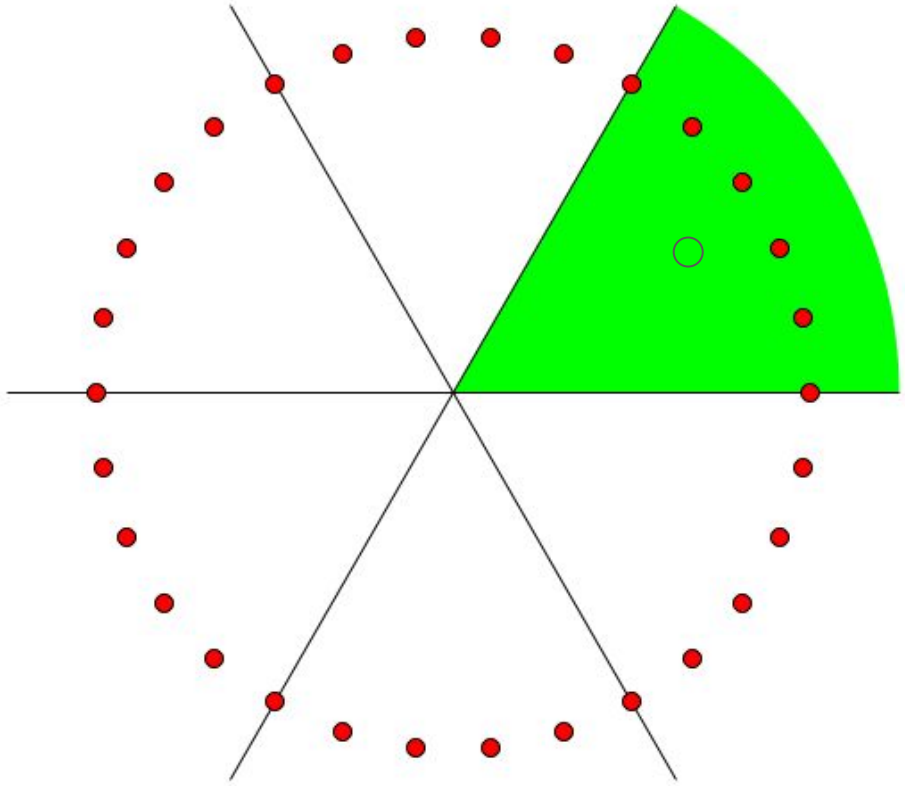
Idea: Divide point cloud up



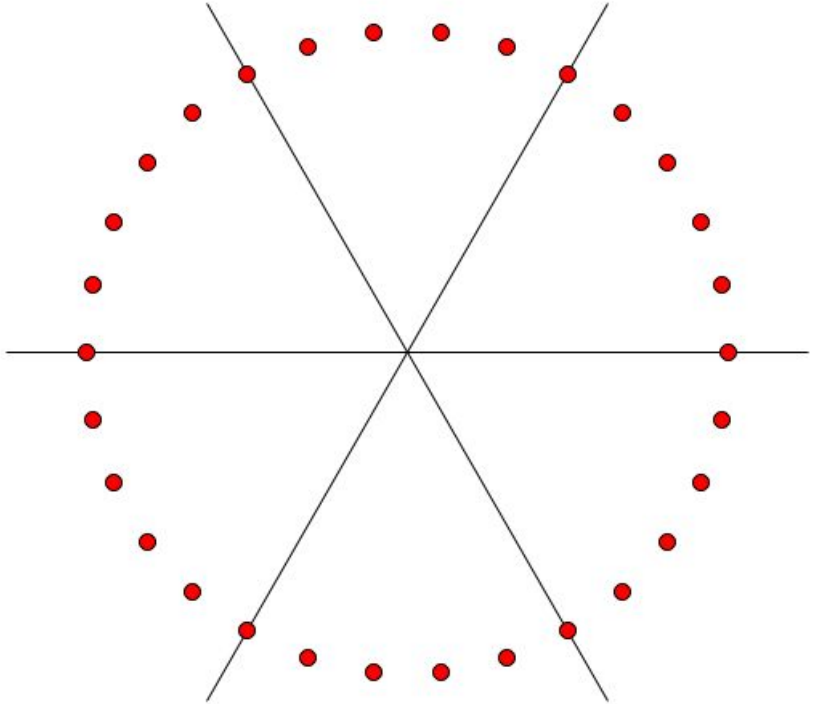
Idea: Divide point cloud up



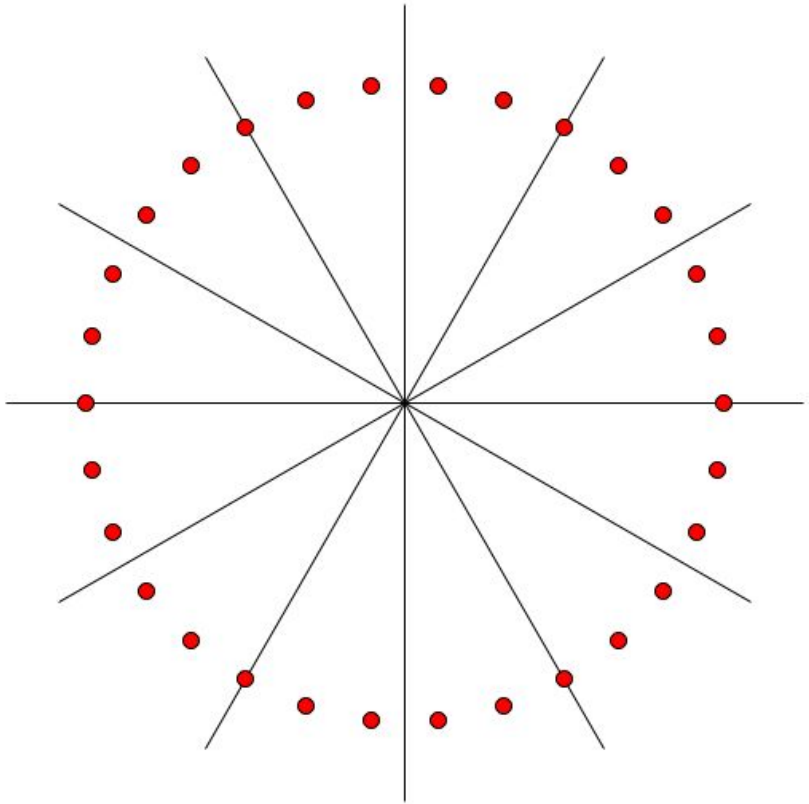
Idea: Divide point cloud up



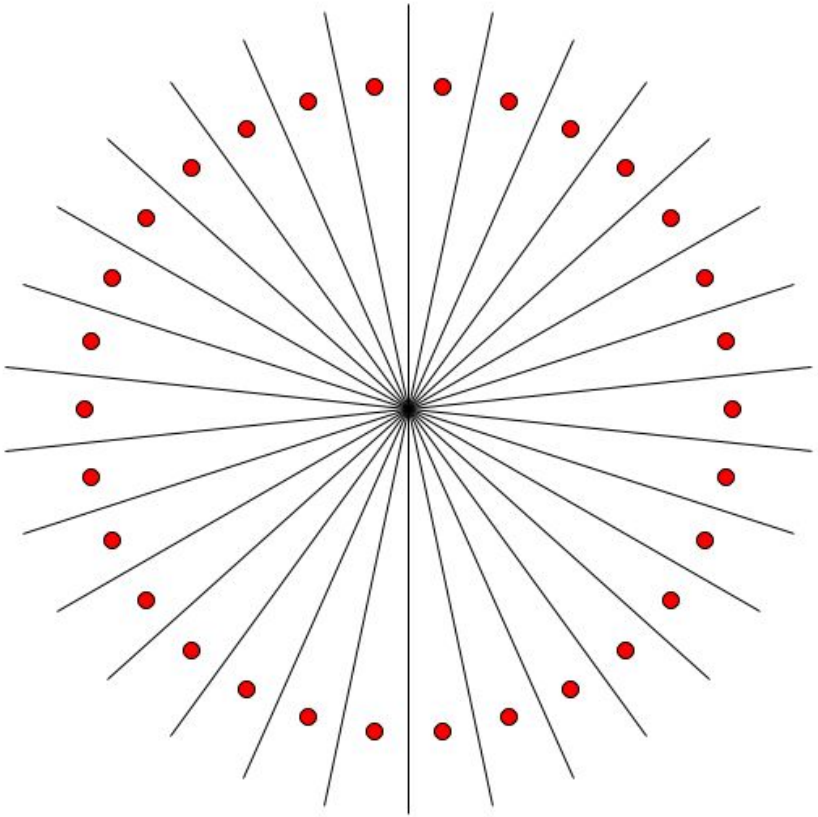
How much can we divide?



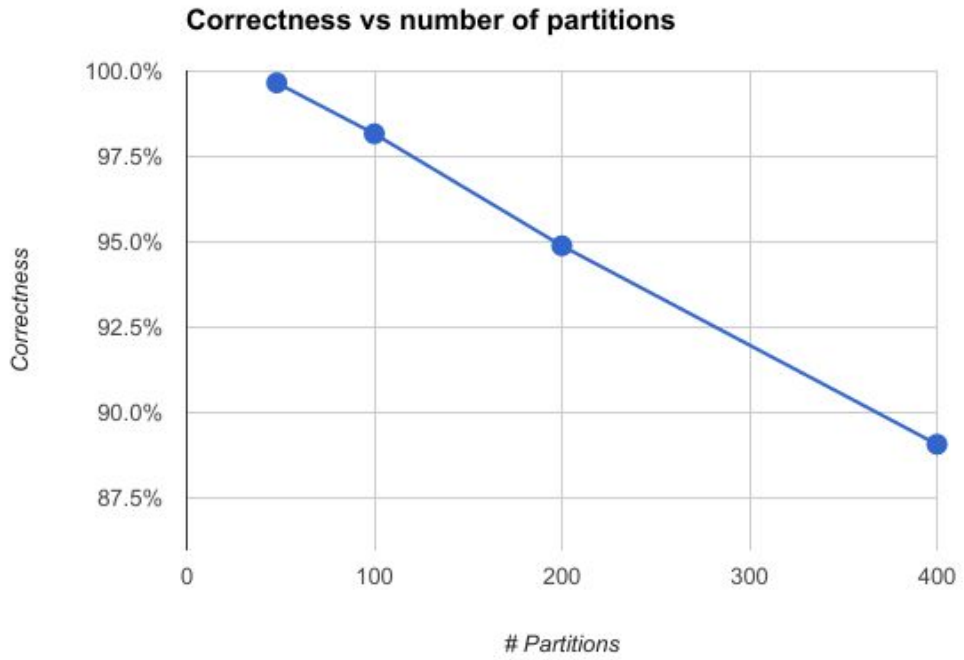
How much can we divide?



How much can we divide?



There is a tradeoff between the number of partitions and the correctness of the nearest neighbor search



Attempt #3: Reducing the search space and parallelizing over points in cloud1

```
void findNearestNeighbor(Point* cloud0, Point* cloud1, int N0, int N1, int* nearestNeighbor) {
  __shared__ float distances[512];

  // Each thread block is responsible for finding the nearest neighbor of one point in cloud1
  Point X = cloud1[blockID]
  int partitionOffset = computePartitionOffset(X); // index in cloud0 where X's partition begins
  distances[threadID] = computeDistance(X, cloud0[partitionOffset + threadID]);

  // Reduce to find smallest distance
  nearestNeighbor[blockID] = minReduce(distances, 512);
}
```

```
Point* cloud0; // N0 points
Point* cloud1; // N1 points

partition(cloud0, 300); // partitions cloud0 into 300 partitions

int nearestNeighbor[N1];
findNearestNeighbor<<<N1,512>>>(cloud0, cloud1, N0, N1, nearestNeighbor);
```

Running time: 0.36s
Total query time: 0.12s

Conclusion

- Parallelizing is hard
- We managed to beat the baseline's querying time, but through a lot of work
- Future work: parallel k-d tree construction and querying