QuickNN
Memory and Performance Optimization of k-d Tree Based Nearest Neighbor Search for 3D Point Clouds

HPCA 2020, San Diego
Reid Pinkham\(^\ddagger\)*, Shuqing Zeng\(^\wedge\), Zhengya Zhang\(^\ddagger\)
+University of Michigan, Ann Arbor
\(^\wedge\)General Motors
*Presenting
What is QuickNN?

• Quick k-Nearest Neighbor search across point clouds for autonomous driving
  • Approximate kNN search
  • Optimized to reduce memory bandwidth
  • 19x and 7.3x speedup over CPU/GPU
Outline

• Motivation
• k-d tree data structure for kNN search
• Architectural design challenges
• QuickNN architecture
• Result
Outline

• Motivation
  • k-d tree data structure for kNN search
  • Architectural design challenges
  • QuickNN architecture
• Result
LiDARs for Autonomous Driving

• Modern LiDAR sensors
  • Produce >100k points per frame
  • 10-20 FPS

• Typical processing
  • Removes redundant ground points first
  • 20-30k points per frame
The Need for kNN Acceleration

• kNN search accounts for ~75% of point cloud processing time
k-Nearest Neighbor

**Given:** a frame of 3D points, a single query point

**Find:** the $k$-nearest frame points to the query point

- Often repeated across a full frame of points

$k = 3$

- Query Point
- Frame Point
## kNN Algorithm Choices

<table>
<thead>
<tr>
<th></th>
<th>Accuracy</th>
<th>Search Complexity</th>
<th>Memory Ops</th>
</tr>
</thead>
<tbody>
<tr>
<td>Linear</td>
<td>100%</td>
<td>☹️</td>
<td>☹️</td>
</tr>
<tr>
<td>Approx. k-means</td>
<td>99%</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Approx. k-d Tree</td>
<td>91%</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Approx. LSH</td>
<td>☹️ 18%</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- k-d tree construction is simpler than k-means cluster finding
  - 2x faster than k-means across dataset (FLANN)

- k-d tree’s slightly lower accuracy is tolerable as kNN is often used in a loop with correction capability

- LSH: Locality Sensitive Hashing
## kNN Algorithm Choices

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Accuracy</th>
<th>Search Complexity</th>
<th>Memory Ops</th>
</tr>
</thead>
<tbody>
<tr>
<td>Linear</td>
<td>100%</td>
<td>☹</td>
<td>☹</td>
</tr>
<tr>
<td>Approx. k-means</td>
<td>99%</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Approx. k-d Tree</td>
<td>91%</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Approx. LSH</td>
<td>18%</td>
<td>☹</td>
<td>☹</td>
</tr>
</tbody>
</table>

- Fast and scalable with acceptable accuracy
Outline

• Motivation
• k-d tree data structure for kNN search
  • k-d tree construction
  • k-d tree search
• Architectural design challenges
• QuickNN architecture
• Result
k-d Tree Construction Process

**Goal:** Partition point cloud into buckets of nearby points

**Example:** a 2D point cloud
k-d Tree Construction Process

Broken into sort and split steps

1. Sort reference points along one of \{x,y\} dimension
2. Split at midpoint, form node
3. Repeat until leaf size meets a minimum threshold
4. Form a bucket at each leaf
k-d Tree Construction Process

Broken into sort and split steps

1. Sort reference points along one of \{x,y\} dimension
2. Split at midpoint, form node
3. Repeat until leaf size meets a minimum threshold
4. Form a bucket at each leaf
k-d Tree Construction Process

Broken into sort and split steps

1. Sort reference points along one of \{x,y\} dimension
2. Split at midpoint, form node
3. Repeat until leaf size meets a minimum threshold
4. Form a bucket at each leaf
k-d Tree Construction Process

Broken into sort and split steps

1. Sort reference points along one of \{x,y\} dimension
2. Split at midpoint, form node
3. Repeat until leaf size meets a minimum threshold
4. Form a bucket at each leaf
k-d Tree Construction Process

Broken into sort and split steps

1. Sort reference points along one of \{x,y\} dimension
2. Split at midpoint, form node
3. Repeat until leaf size meets a minimum threshold
4. Form a bucket at each leaf
k-d Tree Construction Process

Broken into sort and split steps

1. Sort reference points along one of \{x,y\} dimension
2. Split at midpoint, form node
3. Repeat until leaf size meets a minimum threshold
4. Form a bucket at each leaf
k-d Tree Construction Process

Broken into sort and split steps

1. Sort reference points along one of \{x, y\} dimension
2. Split at midpoint, form node
3. Repeat until leaf size meets a minimum threshold
4. Form a bucket at each leaf
**k-d Tree Construction Process**

Broken into sort and split steps

1. Sort reference points along one of \{x,y\} dimension
2. Split at midpoint, form node
3. Repeat until leaf size meets a minimum threshold
4. Form a bucket at each leaf
Outline

• Motivation
• k-d tree data structure for kNN search
• Architectural design challenges
  • k-d tree building (TBuild)
  • k-d tree search (TSearch)
  • Compute kernels
• QuickNN architecture
• Result
TBuild: Memory Read and Write

TBuild
1. Read points and sort them to form the tree
2. Place points into buckets and write them back
TBuild: Memory Read and Write

Challenges:
• Large data
  • Single Frame Size: >360 kB
  • Off-chip storage is required
• Random access
  • Points are placed in random buckets
  • Use write-gather cache to collect random access
TBuild: Memory Read and Write

- Observation:
  - The tree is accessed many times
  - Opportunity for reuse!
Solutions: Data Structure and Caching

- $N_{tree\ nodes} \ll N_{points}$
- Tree nodes have high reuse
- Cache the tree nodes to cut DRAM access

Buckets stored in chunks for efficient burst access
Frame Processing Pipeline

<table>
<thead>
<tr>
<th>Time</th>
<th>Frame 1</th>
<th>Frame 2</th>
<th>Frame 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Round 1</td>
<td>TBuild</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Round 2</td>
<td>TSearch</td>
<td>TBuild TSearch</td>
<td></td>
</tr>
<tr>
<td>Round 3</td>
<td></td>
<td>TSearch</td>
<td>TBuild TSearch</td>
</tr>
</tbody>
</table>
Frame Processing Pipeline

- **TSearch**: Search Frame 2 (query) against Frame 1 (ref)
- **TBuild**: Store Frame 2 as the ref for next step
TSearch: Read and Write

3. Read query points
   • *snoop* query points, cut duplicated access

4. Read ref points

5. Write back NN results
Challenge:
• Rd3 is fragmented

Solution:
• Use read-gather cache
FU for kNN Distance Calculation

Stream in bucket points

kNN Functional Unit

\[ \text{Dist}^2 \]

Calc

A

B

<

Largest Distance

NN List

NNs to Mem
FU for kNN Distance Calculation

Stream in bucket points
Outline

• Motivation
• k-d tree data structure for kNN search
• Architectural design challenges
• QuickNN architecture
• Performance and Benchmarking
QuickNN: A Split Architecture
QuickNN: TBuild

Step 1: Make Tree

Step 2: Fill Buckets
QuickNN: TSearch
Outline

• Motivation
• k-d tree data structure for kNN search
• Architectural design challenges
• QuickNN architecture
• Result
Reduction in Memory Ops

- Linear: 21.39 M
- Simple kD: 7.84 M
- QuickNN: 0.59 M

36x reduction
13x reduction
Speedup Comparison to CPU/GPU

![QuickNN Speedup Chart](chart1)

![QuickNN Perf/Watt Chart](chart2)
Scaling with frame size

QuickNN can effectively scale maintaining an order of magnitude faster execution than GPU
QuickNN: Summary

• Approximate kNN search is a key kernel step for autonomous driving
• QuickNN has multiple memory optimizations to reduce DRAM bandwidth
• Prototyped on FPGA
• Outperforms CPU/GPU by an order of magnitude
• Efficiently scales for future kNN workloads