GPU Nearest Neighbor Searches using a Minimal kd-tree

Shawn Brown    Jack Snoeyink
Department of Computer Science
University of North Carolina at Chapel Hill
shawndb@cs.unc.edu    snoeyink@cs.unc.edu
My Goals

**Primary:** Write spatial streaming tool to process billions of points by applying operators to local neighborhoods.

**Survey:** Compare & contrast kd-Tree, Quad-Tree, and Morton Z-order nearest neighbor search algorithms for GPUs.

**Current:** GPU kd-Tree NN search

**Result:** 15 million 2D queries per second
NN Search Definitions

Vocabulary:
NN - Nearest Neighbor
\( k \text{NN} \) – ‘k’ nearest neighbors

Definitions:
d is the number of dimensions
\( S \) is a search set containing ‘\( n \)’ points
\( Q \) is a query set containing ‘\( m \)’ points
\( \text{dist}(a,b) \) is a distance metric between two points

\[
\text{dist}(a,b) = \sqrt{(b_1 - a_1)^2 + (b_2 - a_2)^2 + \cdots + (b_d - a_d)^2}
\]
**NN Search Types (part 1)**

**QNN:** Query Nearest Neighbor
Find the closest point in $S$ for each point in $Q$ by $\text{dist}(p,q)$.

*Input:* $S, Q$
*Output:* List of $m$ indices of closest points in $S$.

**kNN:** ‘$k$’ Nearest Neighbors
Find the $k$ closest points in $S$ for each point in $Q$ by $\text{dist}(p,q)$.

*Input:* $S, Q$
*Output:* List of $km$ indices of closest points in $S$. 
NN Search Types (part 2)

**All-NN:** All Nearest Neighbor
Find the closest point in $S$ for each point in $S$ by $\text{dist}(p,q)$.

*Input:* $S$ ($Q \leftrightarrow S$)

*Output:* List of $n$ indices in $S$.

*Note:* Exclude zero distance results

**All-$k$NN:** All ‘$k$’ Nearest Neighbors
Find the $k$ closest points in $S$ for each point in $S$ by $\text{dist}(p,q)$.

*Input:* $S$ ($Q \leftrightarrow S$)

*Output:* List of $km$ indices in $S$.

*Note:* Exclude zero distance results

**RNN:** Range Query

**ANN:** Approximate Nearest Neighbor
NN search Solutions

**Linear Search:**
Brute force solution, compare each query point to all search points

\[ O(mn) \]

**Spatial Partitioning Data Structures:**
Divide space into smaller spatial cells. Use “branch and bound” to focus on productive cells.

**Examples:** kd-tree, Quad-tree, Grid, Voronoi Diagram, …

---

**Spatial Partitioning:**
subdivide space

**Data Partitioning:**
subdivide data into sets
NN Searches on GPU

- **Purcell 2003**
  - Multi-pass using uniform grid
  - Approximate

- **Bustos 2006**
  - Trick video card into finding Manhattan distance by texture operations

- **Rozen 2008**
  - Bucket points into 3D cells then brute force search on 3x3x3 neighborhoods

- **Garcia 2008**
  - Brute force algorithm

- **Zhou 2008**
  - Breadth first search kd-tree
  - Voxel Volume split heuristic
  - Build time: 9-13x faster vs. CPU
  - Search time: 7-10x faster vs. CPU

- **Qiu 2008**
  - Depth first search kd-tree
  - Median split heuristic
  - Approximate results
  - Registration time: 100x faster vs. CPU

Search time: 100x faster vs. MATLAB
**kd-tree**

Invented by J.L. Bentley, 1975

<table>
<thead>
<tr>
<th>Data Types</th>
<th>Points (more complicated objects)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Hierarchical</strong></td>
<td>Corresponds to a binary tree</td>
</tr>
<tr>
<td>Axis aligned spatial <strong>cells</strong></td>
<td>• Each cell ↔ node of the binary tree</td>
</tr>
<tr>
<td></td>
<td>• The root cell contains the original bounds and all points</td>
</tr>
<tr>
<td><strong>Recursively defined</strong></td>
<td>• Divide each cell into left and right child cells starting from the root.</td>
</tr>
<tr>
<td></td>
<td>• The points associated with each cell are also partitioned into the left and right child cells</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>Splitting Heuristics</strong></th>
<th>Form a cutting plane (pick split axis &amp; split value)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Data Partitioning</strong></td>
<td><strong>Median Split</strong></td>
</tr>
<tr>
<td><strong>Space Partitioning</strong></td>
<td>Empty space maximization</td>
</tr>
<tr>
<td></td>
<td>Surface Area, Voxel volume, etc.</td>
</tr>
</tbody>
</table>
Building a kd-tree

Add root cell to build queue
While build queue not empty
  • grab current cell from build queue
  • Pick a cutting plane (via median split)
  • **Subdivide** current cell
    • **Termination** “Do nothing” < m points in cell
    • Split parent bounds into left & right cells
    • Partition parent points into left & right cells
    • Add left & right cells to build queue

**Storage:** $O(dn)$
**Build Time:** $O(dn \log n)$
More Build details

- Build kd-tree on CPU, transfer nodes to GPU

**Splitting heuristic**

- Use *quickmedian* selection algorithm for partitioning points in current range \([start, end]\) on current axis \(<x, y, z, \ldots>\). Root range = \([1, n]\)

- Use **LBM** median instead of true median

- Convert to **Left-balanced median array layout**

  - Move node at median array position to targeted position in Left-balanced median array

- Also create **remapping array** during build

  - Convert kd-node indices back into original point indices
Left Balanced Median (LBM)
Nearly complete binary tree

Case 1: Last row is below half-way point

Case 2: Last row is above half-way point
Left Balanced Tree

Links: Given node @ $i$
- Parent = $i/2$
- Left = $2i$
- Right = $2i+1$

Tests:
- isRoot $(i==1)$
- isInvalid $(i > n)$
- isLeaf $(2i > n)$
  & $(2i+1) > n$
Searching a kd-tree

Push **root** node onto stack

Recursively search children**

- Pop current search node off stack
- **Trim Test** current node, if offside
- currDist = \text{dist}(qp, currNode.point),
- Update **Best** distance, if currDist is closer
- Map left/right nodes onto onside/offside
- **Trim Test** & Push offside node onto stack
- Push onside node → **Point Location**

NN = **Best** distance (Best index)

**Search Times**

Best: \( O(dm(\log n + t)) \)

Expected: \( O(dm(n^{1-1/d} + t)) \)

**Based on Arya's Efficient NN kd-tree Method**
Trim Test Optimization

Trim Test (fail)

Onside

Offside

Trim Test (pass)

Offside

Onside

Onside = child cell containing query point
Offside = leftover child cell (without query point)

No 1D overlap → safe to discard the entire sub-tree.
More Search Details

- **Cyclic**
  - start at root with x-axis
  - \( \text{nextAxis} = (\text{currAxis} + 1) \mod d; \text{prevAxis} = (\text{currAxis} - 1) \mod d; \)

- **Backtracking via DFS stack, not BFS queue**
  - **Less storage** → shared memory: \( O(\log n) \) stack vs. \( O(n) \) queue
  - **Better trim behavior**: 40-80 iterations per query point using stack vs. 200-500 iterations using queue

- **12 GPU kernels**
  - NN types (QNN, All-NN, kNN, All-kNN) * (2D, 3D, 4D) = 12 kernels
  - Could be rewritten to one kernel using templating

- **One thread per query point**
  - I/O Latency overcome through thread scheduling
  - Thread block must wait on slowest thread to finish

- **Avoid slow I/O operations (RAM)**
  - 1 I/O (load point) per search loop
  - extra trim test → continue loop before doing unnecessary I/O
  - Remap once from node index to point index at end of search

- **kNN search**
  - **Closest heap** data structure
  - Acts like array (k-1 inserts) then acts like max-heap
  - Trim distance kept equal to point at top of max-heap
**GTX 285 Architecture**

**Device Resources**
- 30 GPU Cores
  - 240 total thread processors
- 1 GB on-board RAM
- 32 KB constant memory

**GPU Core**
- 8 physical thread processors per core
- 1 double precision unit per core
- 16 KB shared memory
- 8,192 shared 32-bit registers

**Thread Processor**
- Shares resources (memory, registers) in same GPU core
## Execution Model

<table>
<thead>
<tr>
<th>Hardware</th>
<th>Software</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Thread Processor</td>
<td>Thread</td>
<td>Threads are executed by thread processors</td>
</tr>
<tr>
<td>GPU Core</td>
<td>Thread Block</td>
<td>Threads blocks executed on GPU cores</td>
</tr>
<tr>
<td>GPU Device</td>
<td>Grid</td>
<td>A kernel is launched as a 1D or 2D Grid of thread blocks</td>
</tr>
</tbody>
</table>

- Supports syncing of threads within a block
- A kernel can execute on a GPU device at a time.
- Syncing across blocks not supported*
**Execution Model**

**Hardware**

- **Thread Processor**
  - Threads are executed by thread processors

- **GPU Core**
  - Threads blocks executed on GPU cores
  - Supports syncing of threads within a block

- **GPU Device**
  - A kernel is launched as a 1D or 2D Grid of thread blocks
  - Only one kernel can execute on a GPU device at a time.
  - Syncing across blocks not supported*

**Notes**

- Thread blocks start & stay with initial core
- Thread block finishes when all threads finish
- Multiple blocks get mapped to each core
- One GPU core can execute several blocks concurrently depending on resources
- Maximum of 512 threads per thread block

*Syncing across blocks not supported.
GPU Hardware Limits and Design Choices, part 1

- **Memory**
  - Aligned data (4, 8, 16 bytes) → better performance
  - limited capacity → use minimal data structures

- **Memory Hierarchy**
  - registers » shared » constant » RAM
    - Local variables → registers
    - stacks/arrays → shared

- **Floats (IEEE 754 compliant)**
  - Focus on singles (32-bit)
  - Doubles (64-bit) are 8x slower on GTX 285

- **Thread Block Size**
  - 4-16 threads per block is optimal based on testing
  - 1 thread per query point
GPU Hardware Limits and Design Choices, part 2

- **Latency**
  - Waiting on I/Os impacts performance
  - Hide I/O latency by massive scheduling of threads
  - 1 thread per query point

- **Divergence**
  - Divergent branching degrades performance
  - Minimize branching

- **Coalescence**
  - GPU can coalesce aligned sequential I/O requests
  - Unfortunately, kd-tree searches do not lend themselves to aligned I/O requests

Good = 1 I/O op

Bad = 16 I/O ops

Aligned, Sequential
kd-tree Design Choices

Bound kd-tree Height
Bound height to $\text{ceil}[\log_2 n]$
Build a balanced static kd-tree
Store as left-balanced binary array

Minimal Foot-print
Store one point per node $O(dn)$
Eliminate fields
  No pointers (parent, child) $\rightarrow$ Compute directly
  No cell min/max bounds
    Single split plane per cell is sufficient
    Split plane (value, axis) is implicit
  Cyclic kd-tree axis access $\rightarrow$ track via stack
kd-tree $\rightarrow$ inplace reorder of search points

Final kd-tree Design:
  • Static
  • Balanced
  • Median Split
  • Minimal (Inplace)
  • Cyclic
Storage:
  • one point per node
  • left balanced array
    $i/2, 2i, 2i+1$
### Timings (in ms)

#### QNN search on GPU (CPU)

<table>
<thead>
<tr>
<th>$n$</th>
<th>2D (in ms)</th>
<th>3D (in ms)</th>
<th>4D (in ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1,000</td>
<td>0.07 (0.10)</td>
<td>0.18</td>
<td>0.41</td>
</tr>
<tr>
<td>10,000</td>
<td>0.42 (12.46)</td>
<td>1.02</td>
<td>2.12</td>
</tr>
<tr>
<td>100,000</td>
<td>4.17 (156.20)</td>
<td>10.10</td>
<td>23.10</td>
</tr>
<tr>
<td>1,000,000</td>
<td>45.62 (2,001.20)</td>
<td>111.34</td>
<td>247.47</td>
</tr>
<tr>
<td>10,000,000</td>
<td>668.07 (26,971.21)</td>
<td>1,614.34</td>
<td>3,840.73</td>
</tr>
</tbody>
</table>

#### All-$k$NN search on GPU (CPU), $k = 31$

<table>
<thead>
<tr>
<th>$n$</th>
<th>2D (in ms)</th>
<th>3D (in ms)</th>
<th>4D (in ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1,000</td>
<td>1.01 (0.10)</td>
<td>1.64</td>
<td>2.64</td>
</tr>
<tr>
<td>10,000</td>
<td>5.88 (12.46)</td>
<td>12.57</td>
<td>28.73</td>
</tr>
<tr>
<td>100,000</td>
<td>57.04 (156.20)</td>
<td>123.74</td>
<td>291.26</td>
</tr>
<tr>
<td>1,000,000</td>
<td>579.57 (10,127.02)</td>
<td>1,270.45</td>
<td>2,9991.02</td>
</tr>
</tbody>
</table>

Uniform Random data
On range $[0,1]$ for each axis

(85.54 s)
**Optimal Thread Block Size**

### QNN, All-NN
- The optimal thread block size is **10x1** for \( n,m=1 \) million points.

### kNN, All-kNN
- The optimal thread block size is **4x1** for \( n,m=1 \) million points, \( k=31 \).
Increasing $n,m$; Increasing $k$

Increasing $n,m$; $n \leq 100$, use CPU  
$n \geq 1000$, use GPU

Increasing $k$ ($k$NN, All-$k$NN)  
Divergence on GPU gradually hurts performance
Results

GPU: GTX 285 using CUDA 2.3
CPU: Intel I7-920 @ 2.4 Ghz

- **2D Results:** NN up to 36 million points
  \( kNN \) up to 1 million, \( k=31 \)
  GPU runs \( 8-44x \) faster

- **3D Results:** NN up to 22 million points
  \( kNN \) 1 million, \( k=31 \)
  3D: Runs \( 7-29x \) faster
  4D: Runs \( 6-22x \) faster
Limitations, part 1

- Under utilization of GPU
  - Scan, 13 Billion 32-bit elements per second
  - Radix Sort, 480 Million 32-bit key/value pairs per second
  - Kd-tree NN Search, 15 Million queries (2D points) per second against a 15 million element kd-tree.
  - **Solution:** Use another approach that maps onto GPU better

- Low Occupancy
  - Lots of shared memory for per thread stacks
  - QNN 2D Kernel (Max Occupancy = 32,
    - 10 threads per block, 12 registers, 2,136 bytes shared memory
    - 19% occupancy

- Divergence
  - almost guaranteed → serialized code access
  - More threads → more opportunities for divergence
  - Entire thread block doesn’t finish until slowest thread finishes

- Bank conflicts
  - Haven’t done any analysis yet…
Limitations, part 2

- No coalescence
  - Access pattern of each search is effectively random
  - Up to a 10x improvement in performance if we could leverage this feature somehow …
  - Possible Solution: Spatially pre-sort search keys

- Shared memory constraints
  - Lots of shared memory pressure from per thread stacks
  - Few threads per thread block
  - Solution #1: More shared memory → better overall performance
  - Solution #2: Reduce stack size (1 32-bit word instead of 2)
  - Solution #3: Move all or part of stack into registers
Future Directions

• Streaming Neighborhood Tool
  • Apply operators on local neighborhoods (billions of points)

• Build on GPU
  • Attempted works but is slower than CPU solution
  • Use coalescence, Increase # of threads
  • Need different approaches for startup, middle, and wind-down phases to get enough parallelism

• Compare & contrast against other NN solutions
  • CGAL, GPU Quadtree, GPU Morton Z-order sort

• Improve Search performance
  • Store top 5-10 levels of tree in constant memory
  • All-NN, All-kNN rewrite search to be bottom-up

• Improve code
  • Use ‘Templates’ to reduce total amount of code
**Build**

- Radix sort the search points using their Morton ID’s as keys
  - Fixed depth (4096 bins implies depth 2D = 6, 3D = 4, & 4D = 3)
- Accumulate results from leaves back up to root
- Recursively split and partition any cell with more than ‘m’ points (m = 64, 256, 1024)

**Search**

- Lookup *start cell* corresponding to query point’s Morton ID from search bounds at same fixed depth.
- Traverse down (or up) search stack from *start cell* until current cell contains fewer than ‘m’ points.
  - Brute force compare the ‘m’ points in current cell to query point to get initial ‘k’ closest points list.
- Traverse back up search stack…
  - Branch and bound using overlap trim test.
  - Update list of ‘k’ closest points as closer points are found.
- Should be possible to compress stack into just 2-4 32-bit integers
Thank You

The paper, more detailed results, & the source code are stored at ...

http://cs.unc.edu/~shawndb/
GPU TIPS & Tricks

- Develop methodically
- Minimize I/O’s
- Tweak kernels for better performance
- Use aligned data structures (4,8,16)
- Use Locked Memory I/O
- Compress Data Structures
- Structure of Arrays (SOA) vs. Array of Structures (AOS)
More Information:

**CPU Host Scaffolding**

- Computes Thread Block Grid Layout
  - Pads n,m to block grid layout
- Allocates memory resources
- Initializes search, query lists
- Builds kd-tree
- Transfers inputs onto GPU
  - kd-tree, search, query data
- Invokes GPU Kernel
- Transfers NN results back onto CPU
- Validates GPU results against CPU search,
  - if requested
- Cleanup memory resources
Develop Methodically

- Plan out resource usage (shared, registers)
  - 16K / 32 threads = 512 bytes per thread
- Get the GPU kernel working correctly first
  - Write working function(s) on CPU first
    - Use these function(s) as check on the GPU Kernel(s)
  - Get the GPU Kernel(s) working first on a $1 \times 1$ thread block and $1 \times 1$ grid and then improve to an $m \times n$ thread block and then to a $p \times q$ grid.
- Then focus on improving GPU performance
  - Look for algorithmic improvements
  - Look to minimize memory I/Os
  - Add profiling code (or use a GPU profiler)
  - Find optimal $m \times n$ thread block size for best performance
  - Tweak GPU Kernel (see next slide deck)
- If you improve the GPU code algorithmically, then update the matching CPU algorithm as well for a fair comparison.
More Information

Tweak GPU Kernel

- Is there a better overall algorithm?
- Can I reduce the number of memory I/Os?
  - Combine multiple kernel(s) that can work on data simultaneously
- Can I reduce the size of objects/structures?
  - Combine fields in less space
- Can I re-order the code to be more efficient?
  - More calculations for fewer I/O’s
  - Avoid waits, Insert non-dependent calculations after I/Os
- Can I reduce register usage
  - by reducing or reusing temporary variables?
typedef struct __align__(16) {
    float pos[2];
    unsigned int Left;
    unsigned int Right;
} KDTreeNode2D_GPU;

More Information

Align Data Structures

- CUDA compiler is capable of moving 4, 8, 16 byte chunks around in a single atomic operation
- More efficient to align to one of these boundaries
- May result in some wasted space

Results

<table>
<thead>
<tr>
<th>~Aligned</th>
<th>Aligned</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time (ms)</td>
<td>Time (ms)</td>
</tr>
<tr>
<td>259.039</td>
<td>189.075</td>
</tr>
</tbody>
</table>
More Information

Locked Memory I/O

- Use locked memory instead of paged memory for CPU ↔ GPU transfers
- See CUDA API sample called “*BandwidthTest*”

<table>
<thead>
<tr>
<th>Copy</th>
<th>Bytes</th>
<th>Paged Time (ms)</th>
<th>Pinned Time (ms)</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>Onto</td>
<td>52 MB</td>
<td>22.938</td>
<td>16.073</td>
<td>1.427</td>
</tr>
<tr>
<td>From</td>
<td>8 MB</td>
<td>5.919</td>
<td>3.668</td>
<td>1.614</td>
</tr>
</tbody>
</table>

**BW (GB/s)**
- From 2.267 3.235
- From 1.352 2.181
Consider the lowly Stack
- 16K of **shared** memory
- 16K/32 = 512 bytes per thread
- 32 * 8 bytes = 256 bytes
- We have just enough room for a simple 32 element stack with two 32-bit fields per stack object on each thread
- This is enough to handle a binary tree of $2^{32} = 4$ gig elements
More Information

Compress DATA Structures

• Memory accesses are slow
• Local calculations are fast
• Paying the cost of compression/decompression calculations to reduce memory I/O can increase performance.

```c
typedef struct __align__(16) {
    unsigned int nodeIdx;
    unsigned int splitAxis;
    unsigned int InOut;
    float splitValue;
} KDSearch_CPU;
```

```c
typedef struct __align__(8) {
    unsigned int nodeFlags;
    // Node Idx (29 bits)
    // split Axis (2 bits)
    // InOut (1 bit)
    float splitValue;
} KDSearch_GPU;
```
Break Apart Data Structures

- Structure of Arrays vs. Array of Structures
  - Try both and use which ever gives you better performance
- 8 field (64 byte) KDNode structure
- Managed to compress it to 5 fields (40 bytes) but couldn’t compress further.
- Broke it into 2 data structures
  - KDNode: 4 fields __align 16___ (pos[x,y], left, right)
  - IDNode: 1 field __align 4__ (ID)
- Surprising Result:
  - The algorithm had a **3x-5x speed increase** as a result of this one change alone
More Information

Other Possibilities

- Take advantage of different memory models
  - Use `__shared__` memory
    - Read/Write, 16K, shared by all threads on GPU core
  - Use `__constant__` memory
    - Read only, 64K, 8K cache, watch out for serialized access
  - Use Texture Memory
    - Read only, 8K cache, optimized for 2D, addressing modes

- Use table lookup instead of conditionals

- Use fast math operations
  - FMAD, `__mul24`, `__fdividef(x, y)`, etc.
  - Avoid division, modulus for integers
  - Floating Point arithmetic is actually faster than integer