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*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

$dist(a,b)$ is a distance metric between two points

$$\mathbf{dist(a,b)} = \sqrt{(\mathbf{b_1} - \mathbf{a_1})^2 + (\mathbf{b_2} - \mathbf{a_2})^2 + \cdots + (\mathbf{b_d} - \mathbf{a_d})^2}$$

NN Search Types (part 1)

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

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

***k*NN:** *'k' Nearest Neighbors*
Find the k closest points in S for each point in Q by $\mathrm{dist}(p,q)$.

Input: S, Q
Output: List of km indices of closest points in S.

NN Search Types (part 2)

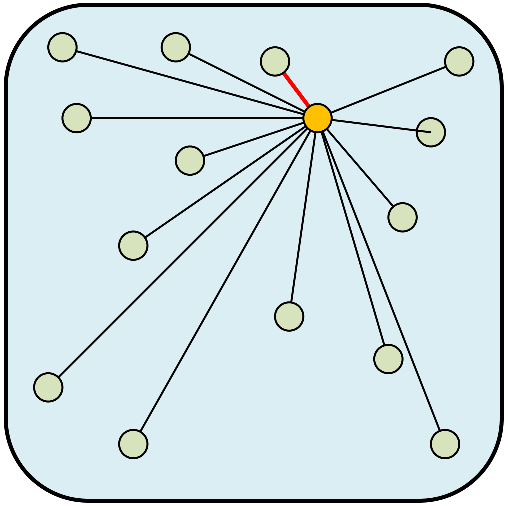
All-NN: *All Nearest Neighbor*
Find the closest point in S for each point in S by $\mathrm{dist}(p,q)$.
Input: S $(Q \leftrightarrow S)$
Output: List of n indices in S.
Note: Exclude zero distance results

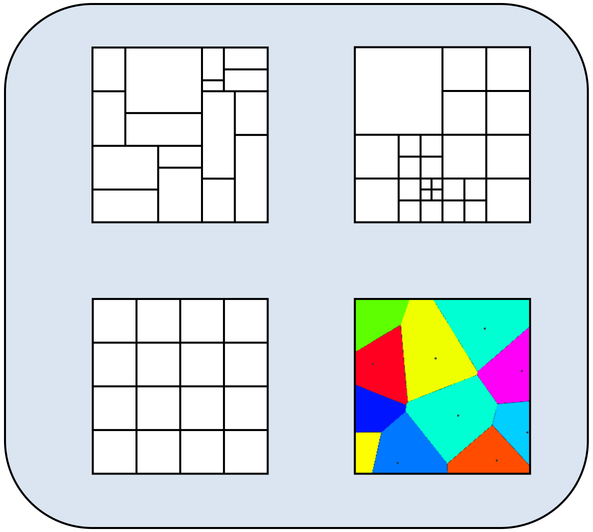
All-*k*NN: *All 'k' Nearest Neighbors*
Find the k closest points in S for each point in S by $\mathrm{dist}(p,q)$.
Input: S $(Q \leftrightarrow S)$
Output: List of km indices in S.
Note: Exclude zero distance results

RNN: *Range Query*

ANN: *Approximate Nearest Neighbor*

NN search Solutions

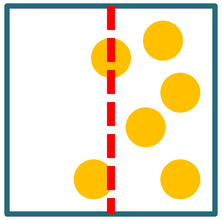




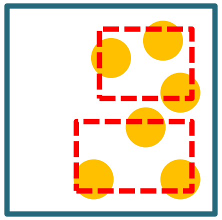
Linear Search:
Brute force solution, compare each query point to all search points

$$O(mn)$$

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

Search time: 100x faster vs. MATLAB

- Zhou 2008
  - Breadth first search kd-tree
  - Voxel Volume split heuristic

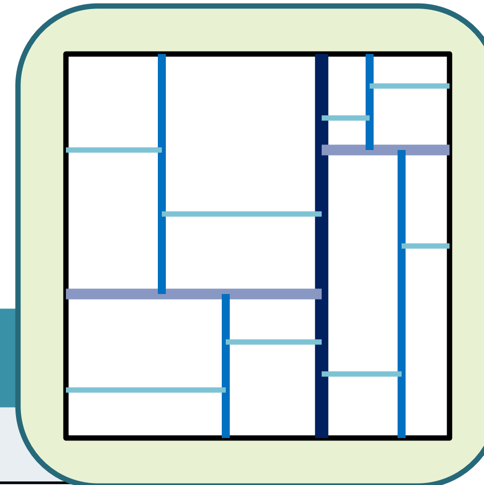
Build time: 9-13x faster vs. CPU

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

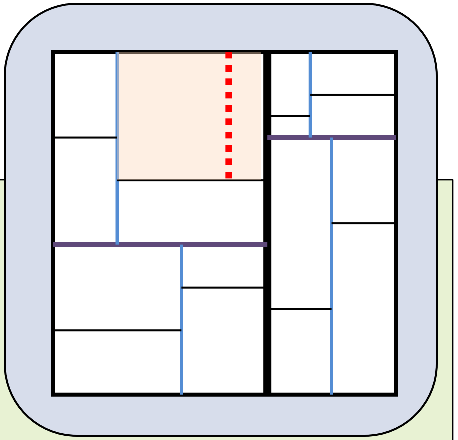
kd-tree



Invented by J.L. Bentley, 1975

Data Types	Points (more complicated objects)
Hierarchical	Corresponds to a **binary tree**
Axis aligned spatial **cells**	• Each **cell** ↔ **node** of the binary tree • The **root** cell contains the original bounds and all points
Recursively defined	• Divide each cell into **left** and **right** child cells starting from the root. • The points associated with each cell are also partitioned into the left and right child cells
Splitting Heuristics *Data Partitioning* *Space Partitioning*	Form a **cutting plane** (pick split axis & split value) **Median Split** Empty space maximization Surface Area, Voxel volume, etc.

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

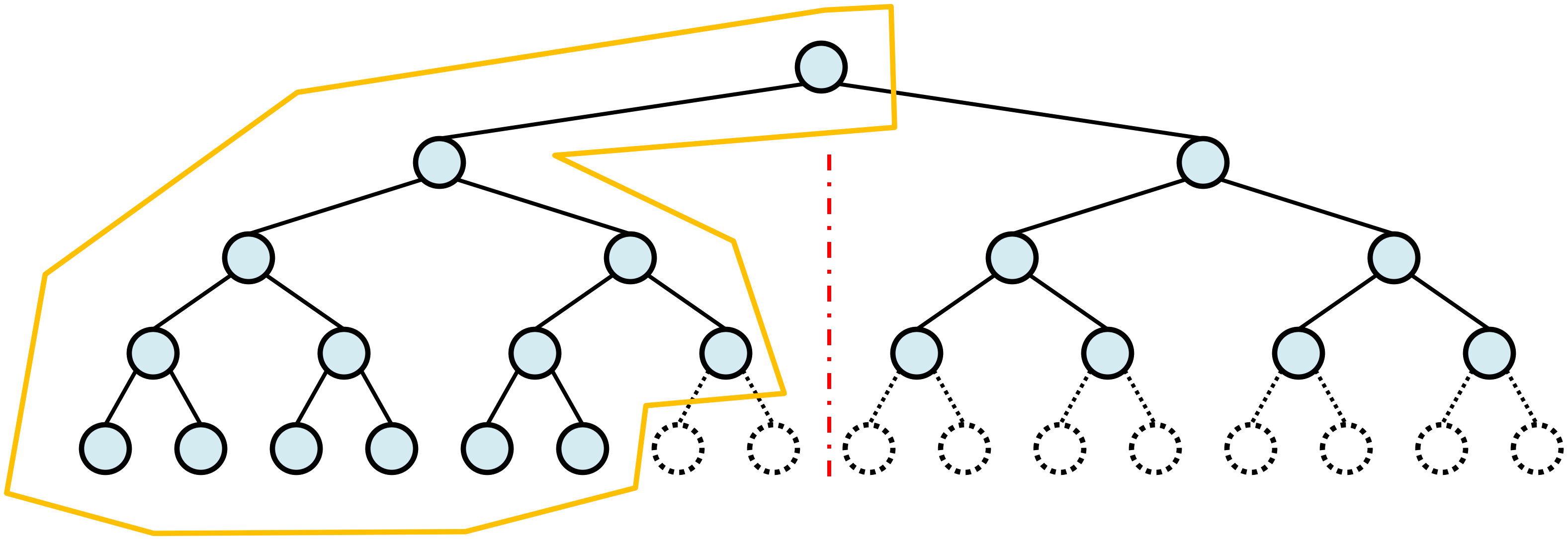
Storage:	O(dn)
Build Time:	O($dn \log n$)

More Build details

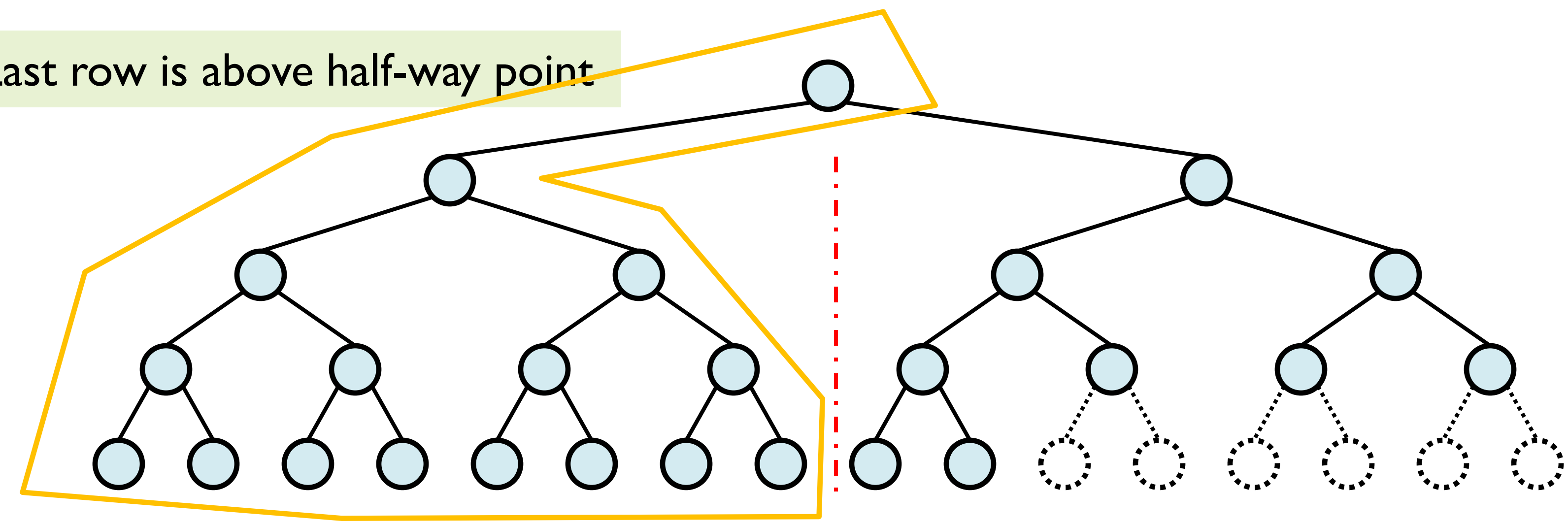
- Build kd-tree on CPU, transfer nodes to GPU
- **Splitting heuristic**
  - Use ***quickmedian*** selection algorithm for partitioning points in current range [*start*,*end*] on current axis <$x,y,z,\ldots$>. Root range = [1,n]
  - Use **LBM** median instead of true median
- Convert to Left-balanced 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



More Information

Left Balanced Tree

Left Balanced Median (LBM)

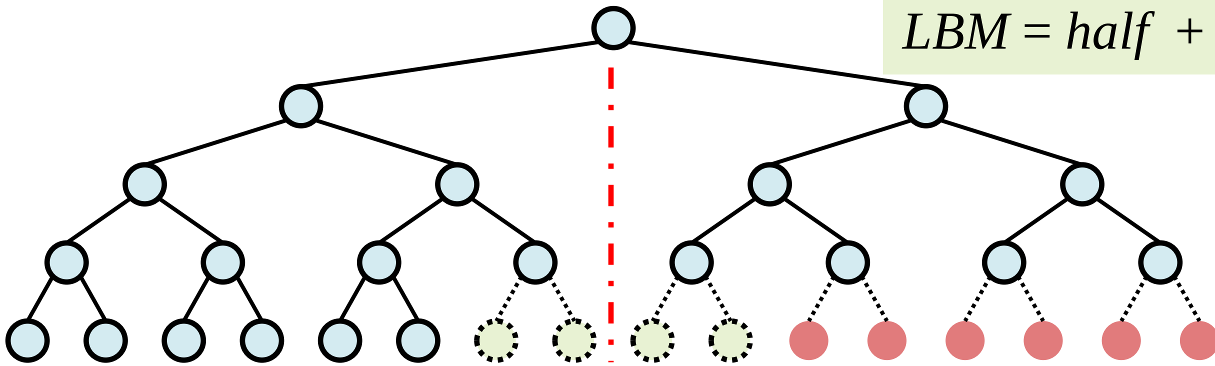
$h = \log_2(n+1)$

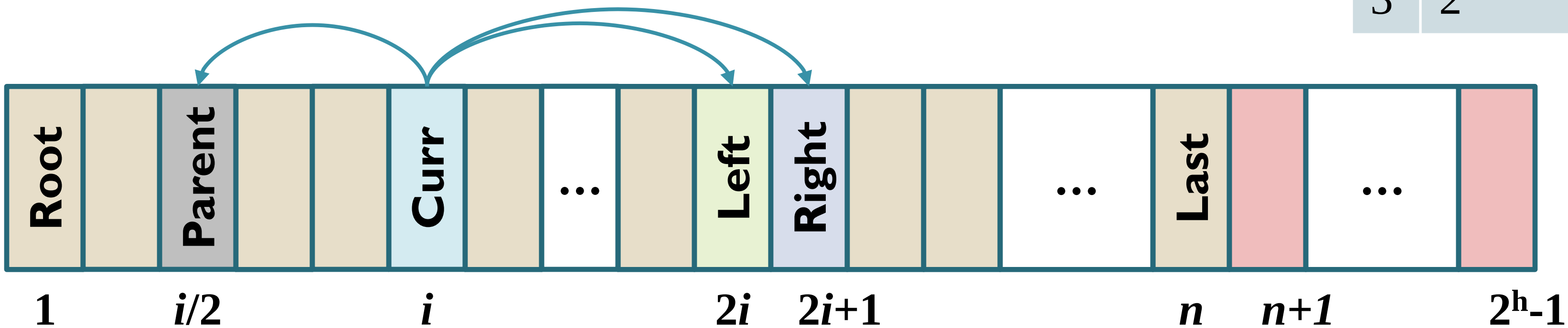
$half = 2^{h-2}$

$lastRow = n - (2 \bullet half) + 1$

$LBM = half + \mathbf{min}(half, lastRow)$

n	LBM
1	1
2	2
3	2





Links: *Given node @ i*

Parent = $i/2$

Left = $2i$

Right = $2i+1$

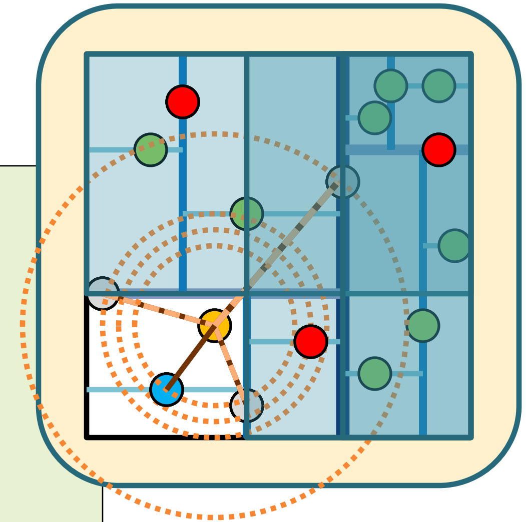
Tests:

isRoot $(i==1)$

isInvalid $(i > n)$

isLeaf $(2i > n)$ $\cancel{\& ((2i+1)>n)}$

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 = **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***

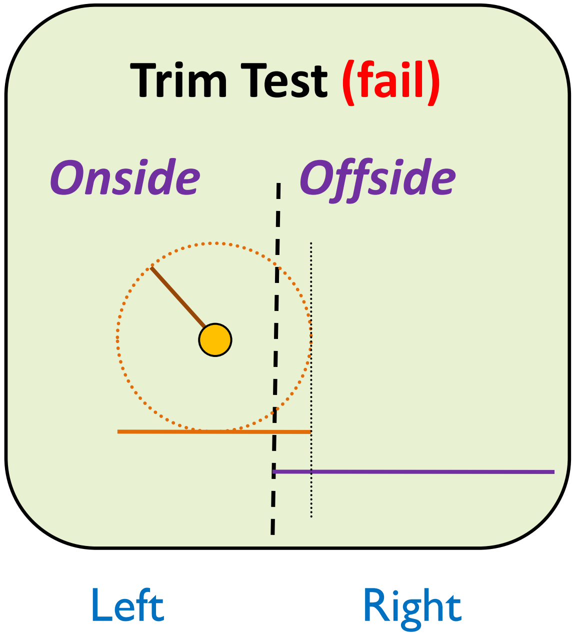
NN = **Best** distance (Best index)

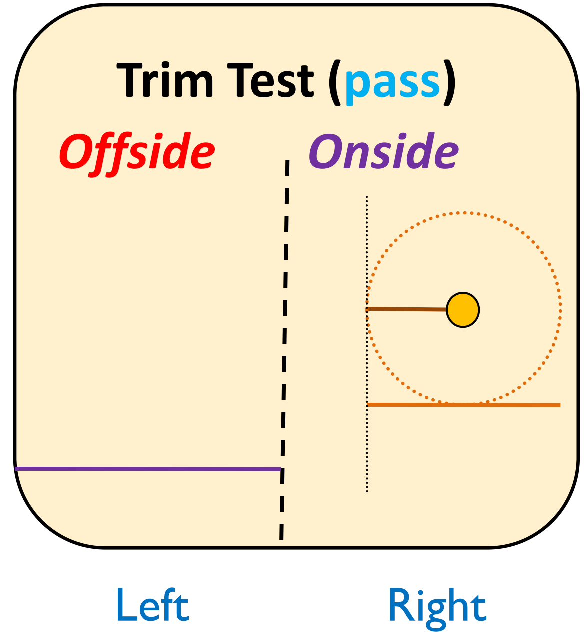
Search Times

Best:	$O\left(dm(\log n + t)\right)$
Expected:	$O\left(dm(n^{1-1/d} + t)\right)$

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

Trim Test Optimization



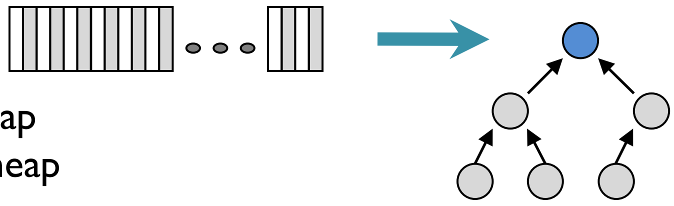


Onside = child cell containing query point
Offside = leftover child cell (without query point)

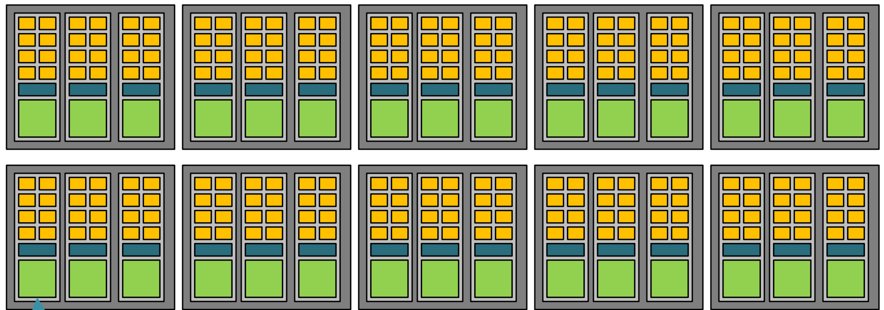
No 1D overlap → safe to discard the entire sub-tree.

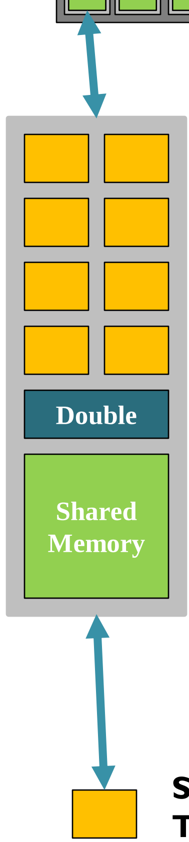
More Search Details

- Cyclic
  - start at root with x-axis
  - **nextAxis = (currAxis + 1) % d; prevAxis = (currAxis – 1) % d;**
- **Backtrack**ing 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
- ***k*NN 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

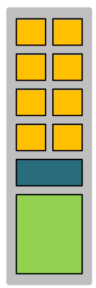
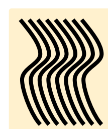
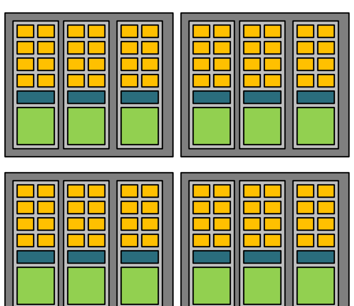
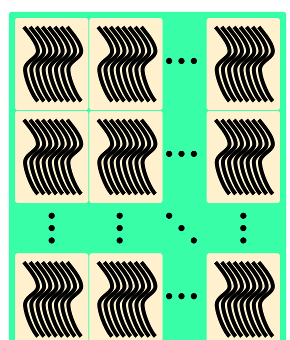
GPU Core

- 8 physical thread processers 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

Hardware	*Software*	*Notes*
Thread Processor 	 **Thread**	***Threads*** are executed by thread processors
GPU Core 	 **Thread Block**	***Threads blocks*** executed on GPU cores Supports syncing of threads within A block
GPU Device 	**Grid** 	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*

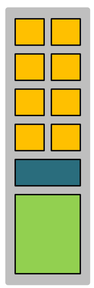
Execution

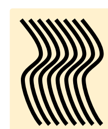
- •Thread blocks start & stay with initial core
- •Thread block finishes when all threads finish
- •Multiple blocks get mapped to each core
- •One GPU core can execute several blocks concurrently depending on resources
- •Maximum of 512 threads per thread block

Hardware

Thread Processor

GPU Core



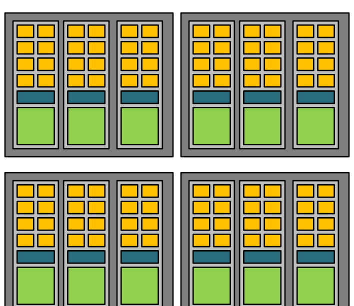


Thread Block

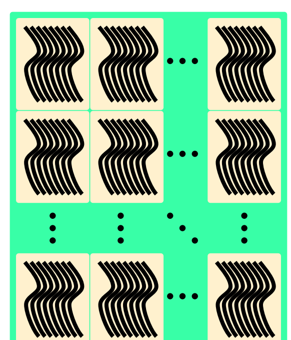
Threads blocks executed on GPU cores

Supports syncing of threads within A block

GPU Device



Grid

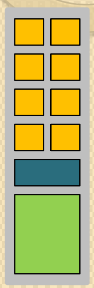


A kernel is launched as a **1D** or **2D** ***Grid*** of thread blocks

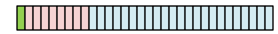
Only one kernel can execute on a GPU device at a time.

Syncing across blocks not supported*

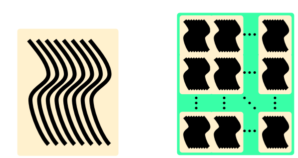
GPU Hardware Limits and Design Choices, part 1



- **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

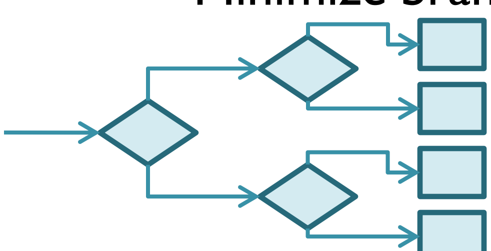


GPU Hardware Limits and Design Choices, part 2

- **Latency**

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

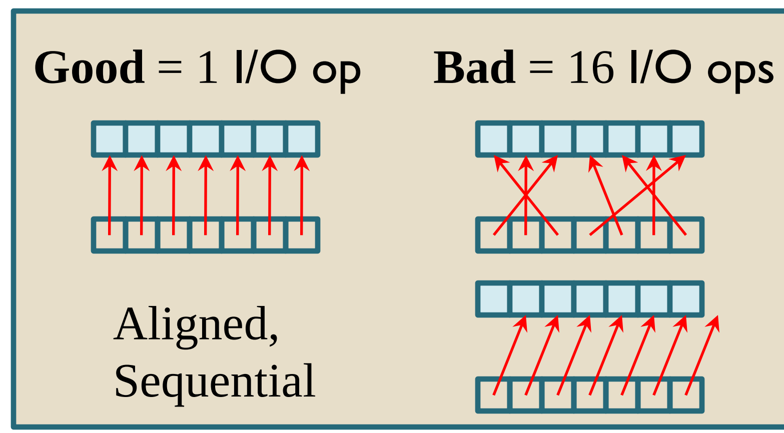
- **Divergence**
  - Divergent branching degrades performance
  - Minimize branching



- **Coalesence**

  GPU can coalesce aligned sequential I/O requests

  Unfortunately, kd-tree searches do not lend themselves to aligned I/O requests



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) → 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 → track via stack

kd-tree → **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)

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

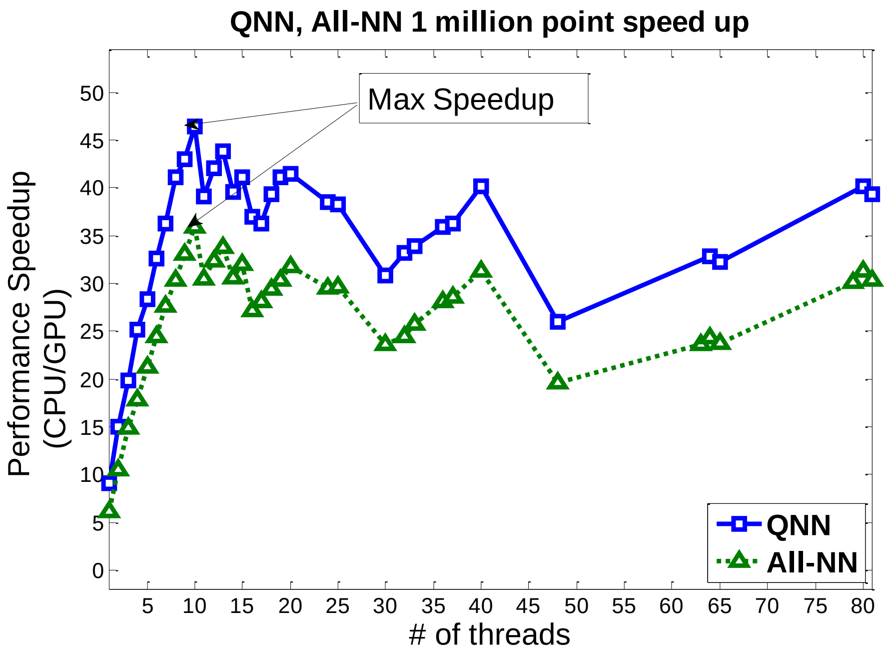
QNN search on GPU (CPU)

n	2D	3D	4D	
1,000	0.07 (0.10)	0.18	0.41	
10,000	0.42 (12.46)	1.02	2.12	
100,000	4.17 (156.20)	10.10	23.10	
1,000,000	45.62 (2,001.20)	111.34	247.47	
10,000,000	668.07 (26,971.21)	1,614.34	3,840.73	(85.54 s)

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

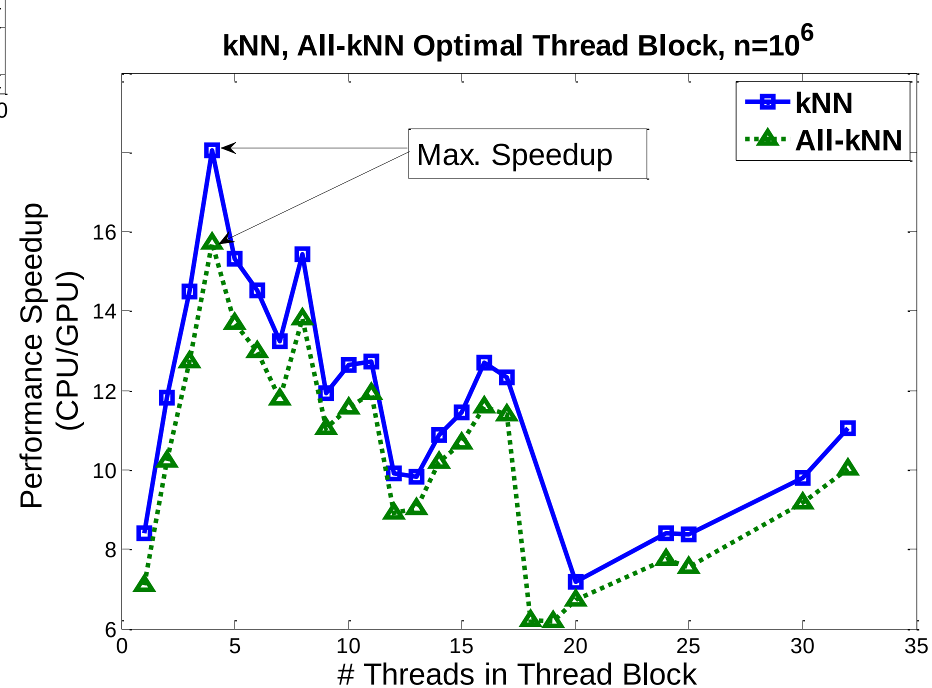
n	2D	3D	4D
1,000	1.01 (0.10)	1.64	2.64
10,000	5.88 (12.46)	12.57	28.73
100,000	57.04 (156.20)	123.74	291.26
1,000,000	579.57 (10,127.02)	1,270.45	2,9991.02

Optimal Thread Block Size

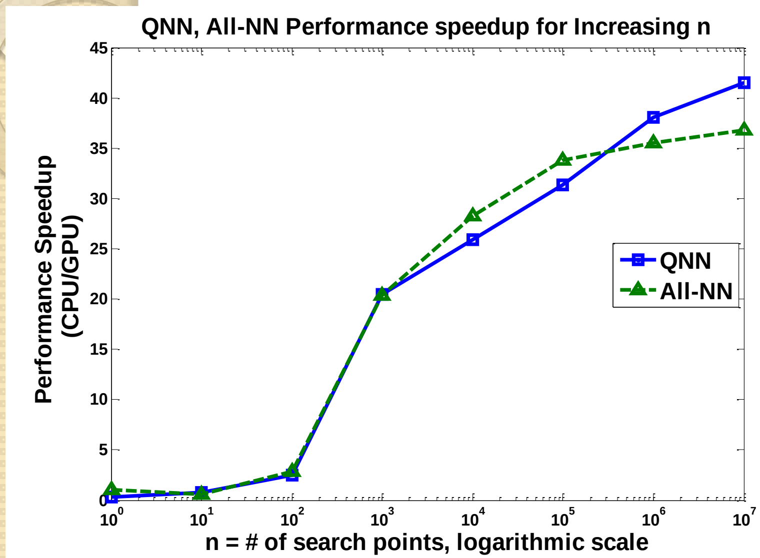


QNN, All-NN
The Optimal thread block
Is **10x1** for n,m=1 million points

kNN, All-kNN
The optimal thread block size is **4x1** for
n,m=1 million points, k=31

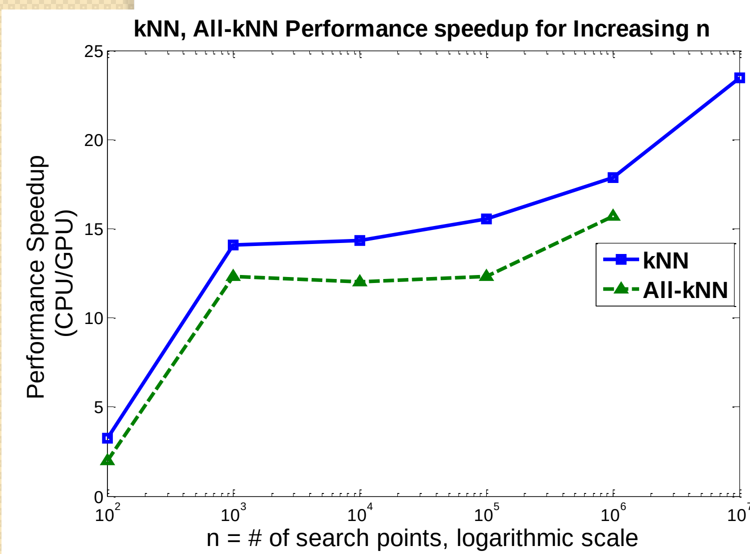


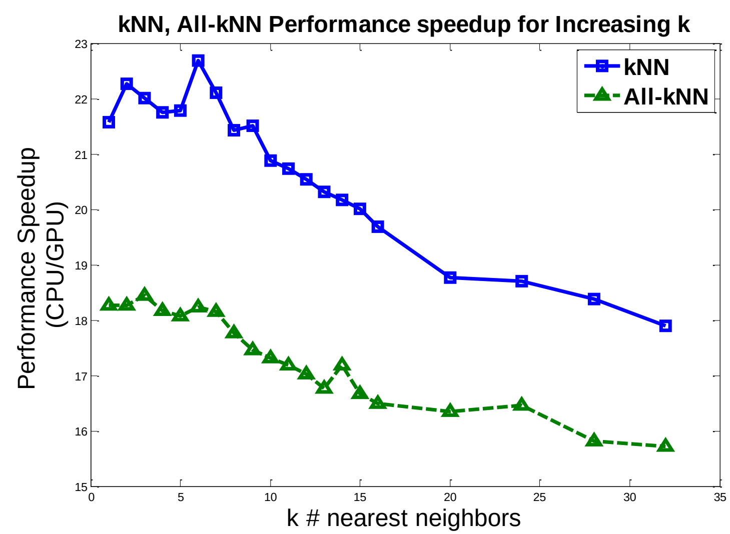
Increasing *n*,*m*; Increasing *k*



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
  *k***NN** up to **1** million, k=31
  GPU runs **8-44x** faster

- **3D Results: NN** up to 22 million points
  *k***NN** **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…

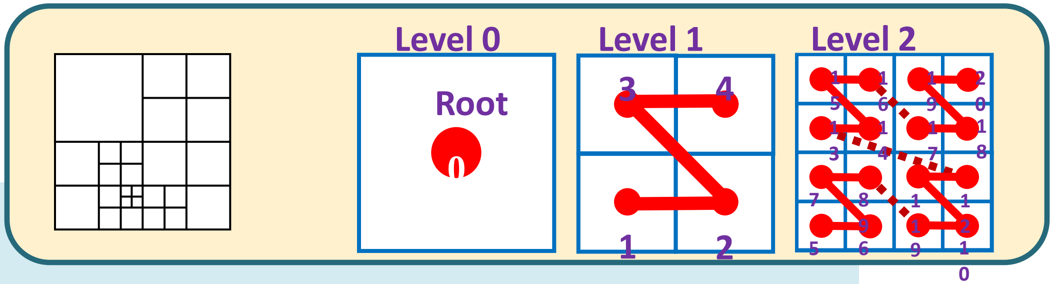
Limitations, part 2

- 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

Quadtree



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 leafs 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

More Information

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 *1x1* thread block and *1x1* grid and then improve to an *mxn* thread block and then to a *pxq* 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 *mxn* 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?

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

```
typedef struct __align__(16)
{
        float               pos[2];
        unsigned int Left;
        unsigned int Right;
} KDTreeNode2D_GPU;
```

Results

~Aligned Time (ms)	Aligned Time (ms)	Speedup
259.039	189.075	1.370

More Information

Locked Memory I/O

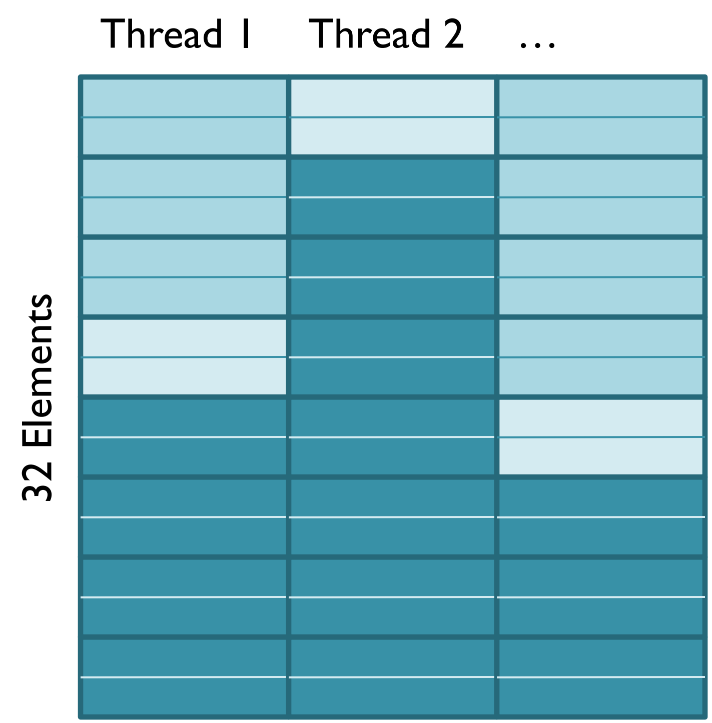
- Use locked memory instead of paged memory for CPU ↔ GPU transfers
- See CUDA API sample called "**BandwidthTest**"

Copy	Bytes	Paged Time (ms)	Pinned Time (ms)	Speedup
Onto	52 MB	22.938	16.073	**1.427**
From	8 MB	5.919	3.668	**1.614**
		BW (GB/s)	**BW(GB/s)**	
		2.267	3.235	
		1.352	2.181	

More Information

Try simple Data Structures

- 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.**

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

→

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

More Information

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