**GPU Nearest Neighbor Searches using a Minimal kd-tree**

Shawn Brown Jack Snoeyink

Department of Computer Science   
University of North Carolina at Chapel Hill

The kd-tree is a spatial partitioning data structure that supports efficient nearest neighbor (NN) and k-nearest neighbor (NN) searches on a CPU. Although the kd-tree is not a natural fit for GPU implementation, it can still be effective with the right engineering decisions. In our implementation, by bounding the maximum height of the kd-tree, minimizing the memory footprint of data structures, and optimizing the GPU kernel code, multi-core GPU NN searches with tens of thousands to tens of millions of points run 20-40 times faster than the equivalent single-core CPU NN searches, even after we rewrote the CPU code with the knowledge gained in optimizing the GPU code.

The GPU NN searches presented in this paper are motivated by a larger goal of being able to extract meaningful structure from massive terrain datasets containing billions of LIDAR points. We have used spatial streaming with finalization tags to support computational tasks that work in mesh neighborhoods or spatial neighborhoods in point clouds. Following Pajarola and co-authors (2005), we want to extend the notion of spatial streaming to tasks that use k-nearest neighbors as their neighborhoods, such as interpolation and surface fitting.

**1. Background**

The nearest neighbor (NN) problem, which finds the closest point in a point cloud to a specified query point, is important in many areas of computer science including: computer graphics, machine learning, pattern recognition, statistics, and data mining (Shakhnarovich 2005).

Actually, there are several NN search problems. In each, given are sets of searched points, , and query points, , and a distance metric in dimensions, such as Euclidean, Manhattan, Chebyshev, or Mahlanobis distance. For example, the Euclidean distance between search point and query point is .

We define six nearest neighbor search problems, the first four of which we will use in testing our implementation.

***Query Nearest Neighbor Problem (*QNN*):*** Find the closest point in for each point in .

*Input:* - a set of search points; - a set of query points

*Output:* list of *m* indices of closest points in *S*, one for each of the *m* query points*.*

***'k' Nearest Neighbor Problem (*NN*):*** Find the closest points in for each point in

*Input:* - a set of search points; - a set of query points; - the number of nearest neighbor points in to find for each query point in . We assume .

*Output:* list of indices of closest points in , with for each of the query points.

***All Nearest Neighbor Problem (*All-NN *or* All-NN*):*** The above problems for , except that zero distance results are excluded, otherwise each point would return itself.

***Range Query (RNN):*** Find all points from contained in each query region belonging to a set . Each individual query region is typically a -dimensional hyper-box or hyper-ball of radius . Although a kd-tree can support RNN searches, We will not discuss them here.

***Approximate Nearest Neighbor (*ANN*):*** Find the approximate closest neighbor to each point in from . The answer is approximately correct with high probability. There is always a small chance that another point (the true solution) is closer. kd-trees can support ANN searches, but we will not discuss them; see Mount & Arya (2010).

**NN Solution Approaches:** A brute force QNN search could directly compare the query point to all points in the search set. Solving the All-NN problem this way takes quadratic time.

However, one can do much better using spatial data structures, such as fixed grids, Quad-trees, BSP Trees, R-Trees, Voronoi Diagrams, and kd-trees. Most of these structures subdivide the original space containing all the points into smaller spatial regions, called cells, and partition the original points into these cells. Many also impose a spatial hierarchy on the cells. NN searches on these data structures use "branch and bound", which focuses the effort on the small set of nearby cells that are likely to contain neighbors and trims away large groups of cells that are too distant. The box tree of Vaidya (1989) is an early example of a data structure to solve the All-NN problem in time. For a review of spatial data structures, please refer to Samet (2006).

We will focus on the kd-tree, a generalized binary tree invented by Bentley (1975) and improved by many in the years since. Arya (1993) detailed an efficient nearest neighbor (NN) algorithm using a balanced kd-tree, a priority queue, and a trim optimization to avoid unproductive search paths, resulting in expected search times for well distributed point sets. Jensen (2001) implemented Arya's NN search in his book on photon mapping. Jensen's implementation is the basis for our CPU kd-tree algorithm.

**The kd-tree:** The kd-tree is a hierarchical spatial partitioning data structure used to organize objects in -dimensional space. The kd-tree partitions points and more complicated objects into axis-aligned cells called nodes. For each internal node of the tree, we pick an axis and split value to form a cutting plane. This cutting plane partitions all points at each parent node into left and right child nodes. One or more points contained in the cutting plane may be stored at each node. kd-trees differ in how the cutting plane is picked. Splitting heuristics include median split, empty space maximization, surface area, voxel volume, etc.

A kd-tree for a search set of -dimensional points takes storage and can be built in time. We build the kd-tree on the CPU, then transfer the kd-nodes onto the GPU.

To perform a nearest neighbor search in a kd tree, one can imagine traversing the entire tree, computing the distance of the query to the search points stored at each node and keeping track of the nearest neighbor point found thus far. If we reach a node whose box extent is farther from the query point than our current best candidate, we can skip that cell and all of its children. If we do the traversal by first visiting the child on the same side of the split as the query point, and the other child only if necessary, then we are likely to prune many nodes. Search queries (QNN, NN and RNN) that return results have been shown to take worst-case time for all search point sets and expected time for well distributed search point sets. For the 2D All-NN and All-NN searches, we multiply the theoretical cost of a single point query by the number of points in our search set, giving worst-case time and expected time using a balanced kd-tree implementation (Samet 2006).

In addition to NN searches, kd-trees can solve point location, range search, and partial key retrieval (Skiena 2008).

**NN on GPU:** For GPU solutions, the first NN searches were implemented brute force, with each query/search point distance calculation computed in parallel, followed by a parallel reduction to find the minimal distance for each query point. The expected performance is , where is the number of parallel cores. Purcell et al (2003) approximated a NN search for photon gathering using a multi-pass algorithm involving a uniform grid and an incrementally growing radius. Bustos et al (2006) stored data as textures and used three fragment programs to compute Manhattan distances and then minimize those distances by reduction. Rozen et al (*2008*) implemented a bucket sort to partition 3D points into cells, then searched brute force in the 3x3x3 cell neighborhood of each query point. Garcia et al (2008) implemented a brute force NNalgorithm in CUDA with a 100+ to 1 speedup compared to the equivalent algorithm in MATLAB. All these authors mention that Arya's kd-tree approach is more efficient but difficult to implement on the GPU due to hardware and software limits.

Zhou et al (2008) built a breadth-first search GPU kd-tree in CUDA with a splitting metric that combines empty space splitting and median splitting. This splitting metric approximates either the surface area heuristic (SAH) or the voxel volume heuristic (VVH). The SAH kd-tree accelerated ray-tracing, while the VVH kd-tree accelerated NN searches. The VVH NNsearch was iterated using a range region search and by increasing the fixed radius of the search region on each iteration. The GPU kd-tree built about 9-13 times faster than the CPU kd-tree. The GPU NN search reportedly ran 7-10 times faster than the CPU NN search.

Qiu et al (2008) developed a GPU ANN search based on Arya's approach with a kd-tree to assist in solving a 3D registration problem on the GPU. The kd-tree is built on the host CPU and then transferred to the GPU before running ANN. The ANN search back-tracks to candidate nodes using a small fixed length queue. If the queue is full, new candidate nodes are discarded. Thus final query results are approximate. According to Qui, GPU registration is 88 times faster than CPU registration. Unfortunately, the performance comparison between the GPU and CPU ANN searches are not broken out from the overall results.

**2. The kd-tree Data Structure**

Our NN search algorithm is adapted from Arya's (1993). It uses a minimal kd-tree, a search stack, and the trim optimization. We demonstrate this solution for 2D, 3D, and 4D points.

**kd-tree Search Concepts:** To understand this approach, we briefly enumerate the following concepts: (1) Each kd-node contains a *search point* <x,y,...>. (2) A *best distance* variable tracks the closest solution found so far. (3) A 1D interval trim test eliminates non-overlapping sub-trees. (4) At any level of our search path, the *onside* node is the left or right child containing the query point and the *offside* node is the remaining node. (5) A depth first search (DFS) first explores *onside* nodes while storing overlapping *off-side* nodes in a *search stack.* (6) Each element stored on the search stack contains a kd-node index, *onside/offside* status, split axis and split value.

***Onside***

***Offside***

**Trim Test**

**kd-tree NN Search:** Our search algorithm works as follows. The root search element (root index, *onside*, x-axis, ∞) is pushed onto the stack. While the stack is not empty, the top search element is popped off the stack, from which is extracted the current node index, *onside/offside* status, split axis and split value. If the node is marked as *offside*, a trim test is applied to accept/reject the entire offside sub-tree. The trim test, illustrated at right, is a 1D interval overlap test of a ball with the offside half-plane, where the ball is centered at the query point with radius equal to the best distance, and the half-plane is defined by the current split axis and split value. If the node is onside (or offside and accepted), the current kd-node is loaded from the node index. Next, if the distance between the query point and the current node’s search point is smaller than the current best, we update the *best distance* and *best index*. The current nodes split axis and value are used to form left and right 1D intervals. The interval containing the query point is the *onside* node, the remaining interval is the *offside* node. The trim test is applied to the *offside* node to keep or reject it. If kept, an *offside* search element is pushed onto the stack. An *onside* search element is always pushed onto the stack. When the stack becomes empty, the *best distance* and *best index* indicate the nearest neighbor.

**3. Hardware Limits and Design Choices**

**GPU Hardware Considerations**

In this section, we discuss GPU hardware limits and the engineering decisions made in response. All our NN solutions were implemented on the Nvidia GTX285 GPU using the CUDA 2.3 API.

**Floats:** The GPU supports both 32-bit and 64-bit floating point data. We focus only on 32-bit floating point data, since 64-bit doubles take twice the space and are a factor of 8 slower. Floating point support is not fully IEEE 754 compliant, and in a handful of queries our GPU and CPU NN searches returned slightly different neighbors. In all cases that we investigated, the neighbor distances turned out to be identical, so both results were valid.

**Memory Hierarchy:** On the GPU, the memory hierarchy contains, from fastest to slowest, registers, shared memory, constant memory, and RAM. For performance, we aim to put local variables in registers, simple data structures in shared memory, and keep points and nodes in main memory. (We cannot put indexed structures in registers; if we try, then the system puts them in main memory, increasing the search time by a factor of 3.).

We minimize the number of data transfers from slower RAM into faster shared memory. The NN search code contains a single read per loop. The total reads per query is expected or worst case. For example: In a QNN search containing 1 million search and query points, each query visits about 40-80 kd-nodes (1 read per node) to find the exact answer.

**Memory Capacity:** The GPU has 1 GB of fixed memory limiting data storage, we seek to minimize the size of our data structures in GPU memory. We compress kd-nodes from 8 down to 2 fields for 2D points, so the 2D QNN search needs only 7 32-bit elements per 2D point to store query points, kd-nodes, and final results. This allows the search to process up to 36+ million 2D points on the GTX 285 GPU.

**Memory Alignment:** Data structures aligned on 4, 8, or 16 byte memory boundaries perform faster than unaligned data. We saw a *37%* speed improvement by aligning our data structures.

**Coalescence:** The GPU can coalesce read requests if they are sequential, but spatial data structures like the kd-tree tend not to result in sequential reads, so we ignore this GPU hardware property at this time.

**Latency:**  The GPU hides latency by scheduling a grid of thread blocks; block performance is limited by the slowest thread in the block. Both grids and thread blocks can be 1D or 2D in shape. A 1D or 2D thread block shape has little effect on performance, so we excluded the 2D thread block results. The grid can support a maximum of 65,535 thread blocks in any dimension. Each thread block can contain a maximum of 512 threads. For the GTX 285, The thread manager maps thread blocks onto 15 GPU multi-cores each containing 16 SIMD cores. We setup our NN searches to use one thread per query point. We pad queries up to the next multiple of the thread block size by repeating the first query as needed; this avoids a comparison that would increase divergence.

**Thread Block Size:** Each GPU core is limited to 16 KB of shared memory and 8,192 32-bit registers. Our current NN searches require about 24-32 registers for temporary variables, which limits the maximum number of threads to at most 256. The QNN and All-NN searches require 192–240 bytes of shared memory for data structures including a 20–28 element deep stack. This limits the maximum number of threads to at most 64–80. Tests reveal that the optimal thread block size is 4–16 threads per block, depending on the search type and the size of the data.

**Divergence:** On the GPU, divergent branching degrades performance. If at least 2 threads in a thread block diverge at a conditional branch, then both the "if" and "else" branches must be executed by all threads in the thread block. We eliminated as many branches as possible from the code. The remaining conditional logic is necessary for correct behavior, for which we accept the performance hit due to divergence. We process the All-NN and All-NN searches in sequential kd-tree order to increase the coherence of all threads in the thread block. This results in a modest *4*–*5%* time improvement for All-NN over QNN, but All-*k*NN is slightly worse than NN.

**kd-tree Design Choices**

Based on the GPU hardware limits, we seek to efficiently use GPU memory resources. This suggests bounding the kd-tree height and reducing the size of data structures in memory.

**Bounding KD-Tree Height:** Shared memory is the target for our NN search stack. There is only 16KB of shared memory across all threads in each GPU core. If we use 64 threads per thread-block then we have at most 256 bytes available for all data structures including the search stack, bounding the stack to at most 20-28 elements. We need to bound the length of any kd-tree search path to avoid overflowing the stack. One way to do this is to bound the height of the kd-tree, bounding the height implies that the kd-tree should be both balanced and static.

**Balanced kd-tree:** A balanced kd-tree of maximum height , with a difference of at most 1 level across all leaf nodes, is built by setting the cutting plane through the median point of each sub-tree.

**Static kd-tree:** A dynamic kd-tree that can handle insertions, deletions, and modifications, but can quickly become unbalanced, and exceed our bound on height. When we know all points *a priori* we can build the kd-tree all at once and never change it. A static tree also enables a *left-balanced binary tree layout order* for the kd-tree.

**Array Layout:** We store the kd-nodes in an array as a left-balanced binary tree. kd-nodes are stored in the range [1,n] using one-based indexing. The root is always stored at index one. Given a node at index , it's parent is found at and it's left and right children are found at and respectively. Child indices greater than are invalid. Leaf nodes have both invalid left and right child indices. The kd-nodes are first built as a left-balanced median kd-tree and then converted into a left balanced binary tree as part of the build process. The left-balanced median position () for splitting a partition containing nodes can be found in 3 steps as: (1) (2) and (3) . Basis Step: The for are respectively.

**Reducing Memory Foot-print:** To maximize the number of points in GPU RAM memory, we minimize the size of the kd-tree data structure. A maximal set of kd-node fields might be: child pointers, parent pointer, split axis, split value, cell bounding box, and stored point.

**-Dimensionality:** We use points with 2-4 dimensions <x,y,...> in these NN searches which reduces the data stored on the GPU. The searches can be extended to higher dimensions as well, however, since kd-tree worst-case performance is no better than a brute-force search for higher dimensions, We recommend not using a kd-tree based NN searches for points with high dimensionality, say .

**Eliminating fields:** The parent pointer can be avoided by using the search stack in the NN search to backtrack. The split axis can be implicit in a cyclic kd-tree that splits *<x,y,x,y,...>* and the split value implied by the stored -dimensional point. Cell bounding boxes are not needed forNN search. Child pointers can be eliminated by computing them directly from the binary tree layout (, ). This results in a fully minimal kd-tree where each kd-node contains just the original points re-arranged into a left-balanced binary tree order. Note: We also need to store a remapping array of size for converting node indices back into the original point indices for final search results.

**Final Design:** We implement the kd-tree as a 2D, 3D, or 4D static balanced cyclical kd-tree with a single left-balanced median point stored at each node (internal or leaf). The nodes of the kd-tree are stored as a left-balanced binary tree array. The NN search is implemented using a depth first search using a stack for back-tracking. This design bounds the height of the kd-tree for predictable stack sizes, minimizes the foot-print of the kd-tree and search nodes in memory, and reduces the number of transfers to/from slower RAM memory.

**4. Building the kd-tree**

In this section, we describe how to build the minimal kd-tree from the search points. The kd-tree is constructed on the CPU and then transferred to the GPU for the GPU NN search. A high level overview is found in figure 1.A.

First, we compute the minimum and maximum bounds of the search points. The root of the kd-tree is conceptually associated with these min-max bounds and the sequence [1*,n*] of original points. A split value is picked along one of the dimensional axes. The split value is chosen to optimize the results according to some measure. All points are partitioned into the two smaller left and right boxes based on the splitting value. Each child node is associated with its bounding box and partitioned sequence of points. The kd-tree is recursively refined by splitting each child sub-tree, the associated boxes, and associated point sequences until we reach some stopping criteria. In general, there are many possible ways in which the splitting plane can be chosen. For our kd-tree, we always choose the left-balanced median point as our splitting plane along the current cyclical axis <*x,y,x,y,...>*, as the tree is descended. The left-balanced median point is found using the **quickmedian** selection algorithm. Recursion is converted into iteration by means of a stack for tracking work yet to be done.

***Quickmedian* Selection Algorithm:** This algorithm is similar to ***quicksort*** and uses the same ***partition*** sub-routine. Each selection iteration runs in 2 phases, pivoting and partitioning. *Pivot phase:* Pick a candidate pivot value using the median of 3 technique. *Partition Phase:* The pivot is used to partition the points into 3 data sets {*Left*: points less than , the singleton pivot value, and *Right*: all points greater than or equal to .}. If the true median position is equal to the current pivot position, the algorithm stops and returns the pivot point as the median. Otherwise, the algorithm iterates into the child data set (left or right) which contains the true median position. This approach takes quadratic time in the worst case but it's expected performance is linear ; see Sedgewick 1998; In practice this approach is fast and reliable.

**5. Searching the kd-tree**

All these NN search solution are based on the approach described in section 4.

**Point Location Problem:** We can quickly find items in a kd-tree by traversing down the tree until we find the cell of interest and then finding the point of interest within that cell. This is easily implemented using the search algorithm described in figure 1.B by simply eliminating the stack and back-tracking. While traversing the kd-tree, we always choose the *onside* node (left or right sub-tree) whose 1D interval contains the query point until arriving at the leaf node which locates the query point.

**QNN Search:** To solve the **QNN** problem, we directly convert the kd-tree NN search algorithm into code. One minor change is also required. The algorithm actually returns the index of the nearest kd-node in median array layout. However, we need the index of the nearest point in the original search set. We solve this problem via an additional remap array of original indices stored in median array order.

**QNN Search Algorithm:**

Here is a brief high level overview of the QNN search algorithm. Both the GPU and CPU code are implemented using this algorithm for a fair comparison. see figure 1.B for more details. For the CPU code, we call the CPU NNsearch algorithm for each query point in turn. For the GPU kernel, we create a host CPU scaffolding function which does the following: (1) Allocates host and device memory for the search points, query points, kd-tree nodes, and final results. (2) Sets up the thread blocks for the GPU. (3) Transfers the inputs onto the GPU. (4) Invokes the parallel GPU NN Kernel. (5) Transfers the output results back onto the CPU.

|  |  |
| --- | --- |
| **procedure** BuildKDTree( *d*, *points, lbm kd-nodes* )  // Initialize kd-tree nodes  n ←  Allocate memory for *n* *median* kd-nodes  Allocate memory for *n* *left balanced median (lbm)* kd-nodes  **for** all in *points*  *medianNodes*[*idx*].*xy ← points*[*idx*].*xy*  *medianNodes*[*idx*]*.pointIdx ← idx*  *medianNodes*[*idx*]*.nodeIdx ← INVALID*  // Add root build item  *top* ← 0  *build* ← { [0,*n*-1], *x-axis*, 1 }  *buildStack*[*top*++] ← *build*;  // Build kd-Tree  **while** *buildStack* not empty **do**  // Get current build item  *currItem* ← *buildStack*[*top*--]  [*low,high*] ← *currItem.sequence*  *currAxis* ← *currItem.splitAxis*  *currIdx* ← *currItem.location*  *N* ← (*high-low*)+1  *M* ← low + **LBMpos**(*N*) // Left balanced Median  *L* ← (*low*+*M*-1)/2  *R* ← (*M*+1+*high*)/2  *left* ← 2\**currIdx*;  *right* ← *left*+1;  *nextAxis* = (*currAxis*+1) % *d*  // Partition via Median Selection  **Partition**( *medianNodes*, *M* ) into sub-sequences  Left{*low*,*M*-1}, Median{*M*}, and Right {*M*+1,*high*}  *mNode* ← *medianNodes*[*M*]  *lbmNode*[*currIdx*] ← { *mNode*.*xy, mNode.pointIdx, M*}  // Add right build item to stack  *rightItem* ← {[*low*+*M*+1, *high*], *nextAxis*, *right* }  *buildStack*[*top*++] ← *rightItem*  // Add left build item to stack  *leftItem* ← {[*low*, *low*+*M*-1], *nextAxis*, *left* }  *buildStack*[*top*++] ← *rightItem*  **end while**  // Cleanup  Free memory associated with *median* kd-nodes  **return** *lbm kd-nodes* | **procedure** QNNsearch( *d*, *qp, kd-nodes, remap* )  // Initialize search  *root* ← *kd-nodes*[1]  *bestIdx* ← 1  *bestDist* ← *Huge Value (Infinity)*  // Add root search element  *top* ←0  *rootElem* ← { 1, *onside*, *x-axis*, *Huge Value (Infinity)* }  *searchStack*[*top*++] = *rootElem*;  // Find Nearest Neighbor  **while** *searchStack* not empty **do**  *currElem* ← *searchStack*[*top*--];  **if** *currElem.state* == *offside*,  *result* ← **trimtest(** *bestDist, currElem.splitValue* **)**  **if** *result == rejected*, return to top of loop  // Update Best Distance  *currNode* ← *kd-nodes*[*currElem.nodeIdx*]  *left* ← 2\* *currElem.nodeIdx*  *right* ← *left*+1  *currDist* ← **distance**( *qp.xy*, *currNode.xy* )  **if** *currDist* < *bestDist*  *bestDist* ← *currDist*  *bestIdx* ← *nodeIdx*  *currAxis* ← *currElem.splitAxis*  *nextAxis* ← (*currAxis* + 1) % *d*  *splitValue* ← *currElem.xy*[*currAxis*];  determine *onside* and *offside* nodes  from *qp*, s*plitAxis*, *splitValue*  // Add *offside* node  *diff2* ← (*qp.xy*[*currAxis*] - *currNode.xy*[*currAxis*])^2  *result* ← **trimtest(** *bestDist, diff2* **)**  **if** *result* == *accepted*  *offElem* ← { *offIdx*, *offside*, *nextAxis*, *splitValue* }  *searchStack*[*top*++] ← *offElem*  // Add *onside* node  *onElem* ← { *onIdx*, *onside*, *nextAxis*, *splitValue* }  *searchStack*[*top*++] ← *onElem*;  **end while**  *bestIdx* ← *remap*[*bestIdx*] // turn *nodeIdx* into *pointIdx*  **return** *bestIdx* and *bestDist*. |
| **Figure 1. a)** On the left is the algorithm used for building the kd-tree from a list of search points. b) On the Right is the algorithm used for searching the kd-tree from a single query point (qp). | |

**All-NN Search:** The All-NNSearch is implemented in exactly the same manner as the QNN search except the search and query points are one and the same and zero-distance results must be excluded.

**NN and All-NN Search:**

The NN and All-NN searches are based on the QNN and All-NN searches. Two simple changes enable these searches. (1) the neighbors are tracked by a *closest heap*. (2) the trim distance is changed to work with points.

**Closest Heap Data Structure:** The nearest neighbors are stored in a *closest heap* data structure which acts first like an array then a heap. The first - nodes visited are appended to the end of the array. Each append takes time. After adding the th search point into the array, the array is converted into a max-distance heap using the **heapify** method. The **heapify** method takes time to run. Each subsequent search point is compared to the top element of the *closest heap*. If the distance from the new point to the query point is less than the distance on top of the heap, the top is replaced with the new point. The correct heap ordering is restored via the **demote** operation. The demote operation takes worst case time to run. During the processing of a QNN search we expect to visit nodes. The worst case time to process will be the time to append the first points, the time to heapify when the closest point heap becomes full , and the time to compare and insert the last nodes into the heap. Summing these operations, we arrive at worst case time, for which the linear term dominates. Each individual NN search thus takes time. Given cores, the All-NN search algorithm should take time.

**Adjusting Trim Distance:** The behavior of the current trim distance is adjusted as follows: The initial huge or infinite best distance value is not allowed to change for the first insertions into the closest point heap data structure. After the th insertion, The trim distance is changed to match the maximum best distance from the top of the *closest heap* data structure in time. Any time, the top of the heap changes, we also need to update the trim distance to correspond to this new maximum best distance.

**GPU Resource constraints for NN and All-NN search**

The same two memory constraints apply with the NN and All-NN searches -- RAM memory and Shared memory. First, we need to save the search results for each of the query points. This means our final result structure takes space. We have a maximum of 1 GB of RAM memory on our GTX 285 card. This limits us to about = one million and = 32 in the worst case. Another possible configuration is = 10 million and = 8. Second, In order to support NN on the GPU, we need to carve out space for the *closest heap* data structure from the same shared memory that we use for our search stack. We assume will never be larger than our maximum stack size of 32 elements. This implies that the NN and All-NN searches can be done using about half as many threads as we used for the singleton QNN and All-NN searches.

**6. Performance Results**

In this section, We compare NN search performance on GPU vs. CPU.

**Test Environment:** All performance tests conducted on a desktop computer: ***Hardware:*** Intel i7 920 CPU (4 cores, 8 threads) each running at 2.67 GHz, 12 GB of CPU RAM memory, 2 NVidia GTX 285 video display cards (1 used for display, 1 used for CUDA computing) each with 1GB GPU RAM; ***OS:*** Windows 7, 64 bit; ***Software:*** CUDA 2.3, Visual Studio 2008 (all apps built as 64-bit).

**Building the kd-tree on CPU**

The table below shows the CPU cost of building the kd-tree for different numbers of 2D points. The amortized time per point to build the kd-tree initially decreases and then surprisingly levels off after 100 points. We expected the time per point to increase, matching the theoretical performance, but some CPU caching effects are perhaps coming into play.

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **, # of points** | **1** | **10** |  |  |  |  |  |  |
| **Build Time (in ms)** | 0.019 | 0.045 | 0.151 | 2.43 | 22.74 | 192.52 | 2,165.31 | 24,491.28 |
| **Time/Pnt (ms/pnt)** | .014 | .0043 | .00165 | .00165 | .00214 | .00163 | .00179 | .00202 |

**Finding the optimal thread block size:** A complicated set of trade-offs determine the optimal number of threads per thread block. One the one hand, more threads means more work gets done, and more opportunities to hide latency, On the other hand, more threads means more competition for resources, increased chances for slow threads to stall entire thread block, and more branch conflicts.

|  |  |
| --- | --- |
| **a) 2D QNN, All-NN searches ( = 106 points)** | **b) 2D kNN, All-NN searches ( = 106 points, =32)** |
| QNN_ALL_1MIL_OPT_TB.emf | k_ALL_NN_Opt_TB_1mil.emf |
| **c) 2D QNN, All-NN searches ( = 107 points)** | **d) 2D NN, All-NN, Increasing (TB=4x1, =32)** |
| Q_ALL_Opt_TB_10mil.emf | k_All_NN_Increase_N.emf |
| **e) 2D QNN, All-NN, Increasing (TB = 10x1)** | **f) 2D NN, All-NN, Increasing (n=106,TB=4x1)** |
| Q_ALL_NN_Increase_N.emf | k_All_NN_Increase_k.emf |
| **Figure 2:** a) This chart plots the GPU/CPU speedup for 2D QNN, All-NN searches for increasing thread block sizes with a fixed size search and query data set of 1 million points. b) This chart is the same but for the 2D NN and All-*k*NN searches. c) this chart plots the 2D QNN, All-NN speedups for 10 million points. d) This chart tracks 2D NN and All-*k*NN speedups for increasing values of . e) This chart tracks 2D QNN and All-NN speedups for increasing values of . f) This chart tracks 2D NN and All-*k*NN speedups for increasing values of from 1-32. | |

To find the optimal thread block size on the GTX 285 GPU, We manually tried thread blocks sizes containing between 1 and 80 threads for each of the NN searches. Shown in Figure 2.a-c are 2D QNN, All-NN, NN, All-NNresults for data sets containing 1 million and 10 million search points, respectively. For QNN of 1 million search points, the optimal thread block was 10x1 with a speedup of 46.4. For 10 million points, it was 7x1 with a speedup of 43.6. For All-NN of 1 million points the optimal thread block was 10x1 with a speedup of 35.9. For 10 million points, it was 10x1 with a speedup of 36.8. For NN using 1 million search points and = 32, the optimal thread block was 4x1 with a speedup of 18.1. For All-NN search using 1 million points and = 32, the optimal thread block was 4x1 with a speedup of 15.7.

**Performance for increasing and :** In figures 2.d-f, We increase, , the total number of search points across several orders of magnitude using the optimal thread block size for each type of NN search. We keep the number of query points equal to the number of search points in these tests. We plot the 4 2D searches in two pairs (QNN and All-NN; NN and All-NN) as they have similar algorithms and results. For the NN searches, We also test performance for increasing values of from 1-32.

**Increasing:** For 2D QNN, we see speed-ups in the range [20 - 41.5], the maximum speed-up occurs for = 10 million. For All-NN, we see similar results: the speedups in the range [20 - 36.8], maximum again at 10 million points. For both searches, if n <= 100 points, it is better to use a CPU or brute force solution.

For 2D NN and All-NN, we fix = 32. For NN, we see speedups in the range [14 - 18] with the maximum at 1 million points. We can also run a query with 10 million points but have to reduce the = 8 in order for the search stack and closest heap to fit in shared memory. In this case, We see a speedup of 23.4. For All-NN, We see speed-ups in the range [12 - 15.7] with the maximum again at 1 million points. Again, for n <= 100, it is better to use a CPU or brute force solution.

**Increasing :** for the 2D searches, We fix n = 106 and vary from 1 - 32. In both cases, the speed-ups appear to follow a shallow inverse quadratic curve. For the kNN search, all the speed-ups are in the range [17.9 - 22.7] with the maximum at = 6. For the All-kNN search, the results are similar with speedups in the range [15.7 - 18.4] with the maximum at = 3.

**7. Conclusion**

The demonstrated QNN, NN, All-NN, and All-NN search algorithms are based on a minimal kd-tree. The minimal kd-tree design is static, balanced, cyclical, using all nodes (internal and leaf) each storing a single point corresponding to the left-balanced median split along the current axis. This kd-tree design allows us to handle more points with higher performance by efficient memory utilization. Not only is it possible to support nearest neighbor searches on the GPU using a minimal kd-tree but there is a large performance gain from doing so.

**2D Summary:** The GPU NNsearches can handle up to 36+ million 2D points. The multi-core GPUQNN search runs 20-44 times faster than the equivalent single core CPU search QNN. The GPU All-NN search runs 10-40 times faster than the CPU All-NN search. The GPU NN search runs 13-18 times faster than the CPU NN search. The GPU All-NN search runs 8 - 17 times faster than the CPU ALL-NN search.

**3D Summary:** The GPU NNsearches can handle up to 22+ million 3D points. The multi-core GPUQNN search runs 10-30 times faster than the equivalent single core CPU search QNN. The GPU All-NN search runs 10-29 times faster than the CPU All-NN search. The GPU NN search runs 7-16 times faster than the CPU NN search. The GPU All-NN search runs 7 - 14 times faster than the CPU ALL-NN search.

**4D Summary:** The GPU NNsearches can also handle up to 22+ million 4D points. The multi-core GPUQNN search runs 8-22 times faster than the equivalent single core CPU search QNN. The GPU All-NN search runs 11-21 times faster than the CPU All-NN search. The GPU NN search runs 6-14 times faster than the CPU NN search. The GPU All-NN search runs 6 - 13 times faster than the CPU ALL-NN search.

**Future Directions:** We are exploring how to best build the kd-tree directly on the GPU. We plan to compare our performance against the CGAL library NN search. Finally, we plan to implement a streaming NN algorithm on top of our GPU kd-tree kernels.

A brief sketch of our plan for the framework for streaming *k*NN is as follows: (1) We will spatially finalize the terrain data into quad-tree cells containing up to at most 1 million points. (2) We will read the finalized steam of points into memory. Upon seeing the finalization tag for a given quad-tree cell, we will stream all points in that cell onto the GPU, build a kd-tree on the GPU, find the *k* nearest neighbors for all points in that cell. We will also test for *interior*/*boundary* points. Imagine centering a hyper-ball at each point with radius equal to the distance to its farthest neighbor. We define an *interior* as a point where the associated hyper-ball is completely contained in the current quad-tree cell bounds. We define a *boundary* as a point whose hyper-ball overlaps one or more neighboring quad-tree cells implying the need for additional *k*NN searches in those overlapping cells before we have the true *k*NN results for these boundary points. (3) We will document the outgoing point stream with 2 new finalization tags to represent when we have seen all *k*NN neighbor points for all points in the current quad-cell (KNN\_INTERIOR , KNN\_BOUNDARY) . (4) The streaming *k*NN framework will be a two-pass solution through the point stream. The first pass will generate and process all interior points and some of the boundary points. All boundary points will be added to work queues for those cells that they overlap. We will also process all boundary points in the work queues associated with the current cell. Then throw all points associated with the just processed cell out of memory. We will need a second pass through the file for those remaining boundary points that overlap cells that have already been seen, processed, and kicked out of memory in the first pass.

**8. References**

**Arya, S., and Mount. M, 1993, "Algorithms for Fast Vector Quantization", IEEE *Proceedings of Data Compression Conference*, IEEE Computer Society Press, pp. 381-390.**

**Bentley, J. L., 1975, "Multidimensional binary search trees used for associative searching", *Communications of the ACM*, Sept. 1975, Vol 18(9), pp. 509-517**

Berg, M. de et al, 2000, *Computational Geometry, Algorithms and Applications, 2nd Edition*, Springer Verlag, New York NY

Bustos, B., Deussen, O., Hiller, S., Keim, D., 2006, "A Graphics Hardware Accelerated Algorithm for Nearest Neighbor Search", *Proceedings of the 6th International Conference on Computational Science*, May 2006, pp. 196-199

Cuccuru G., Gobbetti E., Marton F., Pajarola R., Pintus R., 2009, “Fast low-memory streaming MLS reconstruction of point-sampled surfaces,” *Graphics Interface,* pp. 15-22.

Garcia V., Debreuve E., and Barlaud, M., 2008, "Fast k Nearest Neighbor Search using GPU*." Proceedings of Computer Vision and Pattern Recognition Workshops*, June 2008, pp. 1-6.

Jensen H., 2001, *Realistic Image Synthesis Using Photon Mapping*, A K Peters, Natick MA

**Mount, D. and Arya, S., 2010, "ANN: A Library for Approximate Nearest Neighbor Searching", URL:** <http://www.cs.umd.edu/~mount/ANN/> **ANN Version 1.1.2 (Release date: Jan 27,2010)**

Pajarola, R., 2005, “Stream-Processing Points.” *IEEE Visualization,* p. 31.

Purcell, T.J, Donner, C, Cammarano, M., Jensen, H.W., and Hanrahan, P., 2003, "Photon Mapping on Programmable Graphics Hardware". *ACM SIGGRAPH / EUROGRAPHICS Conference on Graphics Hardware*, ACM Press

**Qiu, D., May, S., and Nuchter, A., 2009, "GPU-accelerated Nearest Neighbor Search for 3D Registration", *7th International Conference on Computer Vision Systems*, Oct 2009, pp. 194-203**

**Rozen, T., Boryczko, Alda, W., 2008, "GPU Bucket Sort Algorithm with Applications to Nearest-Neighbor Search", *Journal of the 16th International Conference in Central Europe on Computer Graphics, Visualization and Computer Vision*, Feb. 2008**

**Samet, Hanan, 2006 *Foundations of Multidimensional and Metric Data Structures*, Morgan Kaufmann, San Francisco CA**

**Sedgewick, R., 1998, *Algorithms in C++, 3rd Edition, Parts 1-4: Fundamentals, Data Structures, Sorting, Searching*, chapter 7 pp. 315-346, Addison-Wesley, Reading MA**

**Shakhnarovich, G., Darrell, T, and Indyk P., Editors, 2005, *Nearest-Neighbor Methods in Learning and Vision, Theory and Practice*, The MIT Press, Cambridge MA**

**Skiena, S. S., 2008, *The Algorithm Design Manual, 2nd Edition*, Springer-Verlag, London UK**

Vaidya, P. M., 1989, "An O(n log n) Algorithm for the All-Nearest Neighbors Problem.", *Discrete and Computational Geometry*, (4):101-115, 1989

**Zhou, K., Hou, Q., Wang, R., and Guo B., 2008, "Real-time KD-Tree Construction on Graphics Hardware", *ACM SIGGRAPH Asia 2008*, April 2008, page 10.**