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

*Shawn Brown & Jack Snoeyink*

*Department of Computer Science, University of North Carolina at Chapel Hill*

|  |  |
| --- | --- |
| **Abstract**  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 10-40 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.  **Search Types:** We support Query Nearest Neighbor (QNN), '' nearest neighbor (NN), and All nearest neighbor (All-NN or All-NN) searches using a Euclidean distance metric.  **Build & Search Algorithms:**  <*See full paper*>  Our NN search algorithm is adapted from Arya's (1993) approach. We use a minimal kd-tree, a search stack, and a trim optimization. We demonstrate this solution for 2D, 3D, and 4D points.  **Search 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 search stack is used to store overlapping *off-side* nodes for back-tracking*.*  **Hardware Limits and Design Choices**  The GTX 285 using CUDA 2.3 is our GPU platform.  **Coalescence:** The GPU can coalesce I/O requests if they are sequential, but spatial data structures like the kd-tree tend not to result in sequential reads, so we ignore this property for now.  **Memory Hierarchy:**  We focus on registers for local variables and shared memory for simple data structures such as arrays and stacks.  **Capacity:** We seek to minimize the size of our data structures. For example, our minimal kd-nodes consist of only the positional data, 2D point = <x,y>. | **Memory Alignment:** Data structures are aligned to 4,8 or 16 byte boundaries.  **Floats:** We focus only on 32-bit floating point data.  **Latency:** The GPU can hide I/O latency by scheduling many threads at once. We setup our NN searches to use one thread per query point.  **Thread Block Size:** Tests reveal that the optimal thread block size is 4-16 threads per block, depending on the NN search type and the size of the data.  **Divergence:**  Divergent branching degrades GPU performance. We eliminated as many branches as possible from our code. The remaining conditional logic is necessary, for which we accept the performance hit.  **kd-Tree Design Choices**  We seek to bound the kd-tree height and minimize the foot-print of data structures in memory.  **Bounded Height:** We bound the tree height to by building a balanced, static kd-tree stored as a left-balanced binary tree. The left-balanced median is used as our splitting heuristic.  **Minimal Foot-print:** We store a single point per kd-node. Using a left-balanced array allows us to find parent & children directly (*i*/2, 2*i*, 2*i*+1). A cyclic kd-tree allows the split axis & split value to be implicit. We obtain a fully minimal kd-tree where each kd-node contains just the original search points re-arranged into left-balanced kd-tree order.  **Results** <*See full paper*>  **2D Results:** 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** and **4D** results: Can handle up to 22+ million points. Searches are 6-30 times faster depending on the search type and size of data.  **References** <*See full paper*>  **Arya, S., and Mount. M, 1993, "Algorithms for Fast Vector Quantization", IEEE *Proceedings of Data Compression Conference*, IEEE Computer Society Press, pp. 381-390.** |