The quest for a fast KNN search

This document is a summary of our most recent (7 February 2014) findings, in the quest for a fast kNN search algorithm.

Our initial investigation led us to believe that a serial implementation could be as fast as the parallel brute-force solution, for point clouds with fewer than 1 000 000 points, given that both algorithms start with an unordered set of points. Reimplementing the brute-force algorithm with bitonic sort, and optimizing for three dimensions, has shown us that this initial belief was unsupported, and currently the brute force algorithm is faster when starting from a unorganized set of points. When considering repeated querying of the same point cloud, the k-d tree based solution pulls ahead, as most of its running time is spent building the k-d tree for querying. If building the k-d tree could be parallelized this could change. although documented in literature, such an parallelization is still elusive.

In order to make the document more readable, we have included short descriptions of the algorithms used, a short reference to theoretical time complexity. We then go on to list our current results, problematic areas and possible improvements.

The following papers, available in the resources folder, forms the literary basis for our current work.

Related to the brute force approach: * *Improving the k-Nearest Neighbour Algorithm with CUDA - Graham Nolan * Fast k Nearest Neighbor Search using GPU - Garcia et al. * K-nearest neighbor search: fast gpu-based implementations and application to high-dimensional feature matching - Garcia et al.*

Related to the k-d tree based approach: * *Real-Time KD-Tree Construction on Graphics Hardware - Kun Zhou et al.*

Brute force based effort

Garcia's base algorithm Garcia's algorithm is based on a naive brute-force approach. It consists if two steps:

1. Calculate the distance between all reference points and query points.
2. Sort the distances and pick the k smallest distances.

Garcias implementation supports any number of dimensions, reference points and query points (or up to ~65000, number of blocks in the GPU). Due to this feature the algorithm use a lot of extra computation power when only one query point and a small dimensions is selected.

Time complexity Steps:

1. O(n). Every reference point must be evaluated once. Since all calculations are independent, we have a large potential for parallelizing.
2. Insertion sort: O(nˆ2).

Our reimplementation Graham Nolan discusses the possibility of improving Garcia's algorithm by reimplementing step two with a bitonic sort. His source code has not been available to us, but he states that the run-time improvements was significant. As well as choosing bitonic sort for the sorting stage of our algorithm, our implementation supports up to 15 000 000 points before memory errors occur, and we have limited the number of dimensions to three.

Time complexity Steps:

1. O(n).
2. Bitonic sort: worst case = O(n*log^2(n)), average time (parallel) = O(log^2(n)).

Results Testing the different algorithms for a range of point cloud sizes and a fixed value for k, gave the following results.

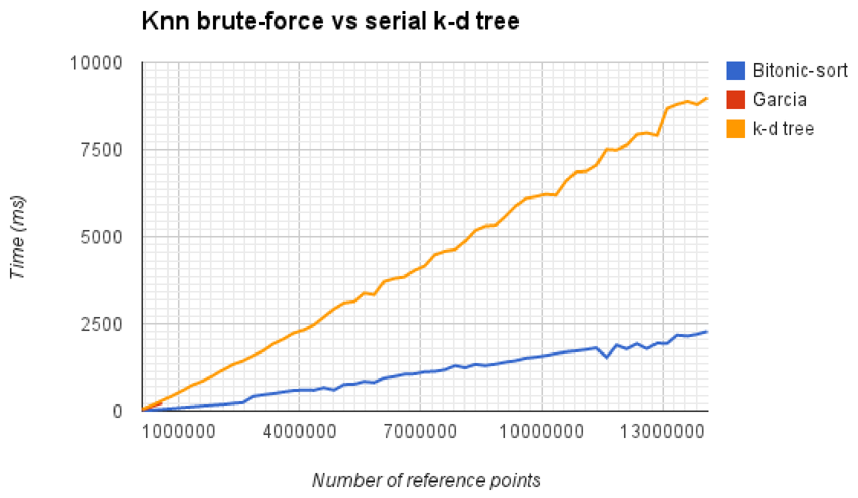

Figure 1: knn-brute-force-vs-serial-k-d-tree

We see that our reimplementation of the brute-force algorithm performs well overall, notably improving on Garcia's implementation (only visible as a short line in the beginning of the graph, due to the restricted number of points it is able to compute). Still more speed is desired before good interactive usage can be achieved.

The results for the serial k-d implementation will be discussed in the next section.

Possible improvements

- Memory improvements. Use shared memory and texture memory.
- Modify bitonic sort, so do not need to sort all points. We can split the distance array to fit into the GPU blocks, move the smallest values in each block, then sort the moved values. ~O((n/b)* b*log^2(b)) subsetof O(n/b), b = Number of threads in each block, n= number of reference points
- Replace bitonic sort with min reduce. O(k*log^2(n)).

KD-tree based effort

k-d trees A k-d tree can be thought of as a binary search tree for graphical data. A few different variations exist, but we will focus our explanation around a 2D example, storing point data in all nodes. The plane is split into two sub-planes along one of the axis (in our example the y-axis) and all the nodes are sorted as to whether they belong to the left or right of this split. To determine the left and right child of the root node, the two sub-planes are again split at an arbitrary point, this time cycling to the next axis (in our example the x-axis) and the

In order to build a k-d tree for 3D space, you simply cycle through the three dimensions, instead of two.

Given the previous splits and selection of nodes, the resulting binary tree would be as shown in the illustration under. (All illustrations gratuitously borrowed from Wikipedia)

Given that the resulting binary tree is balanced, we get an average search time for the closest neighbor in O(log^2 n) time. For values of k << n, the same average search time can be achieved, with minimal changes to the algorithm, when searching for the k closest neighbors. It is known from literature that balancing the tree can be achieved by always splitting on the meridian node. Building a k-d tree in this manner takes O(kn log^2 n) time.

The serial base algorithm

1. Build a balanced k-d tree from the point cloud.
2. Query the tree for different sets of neighbors.

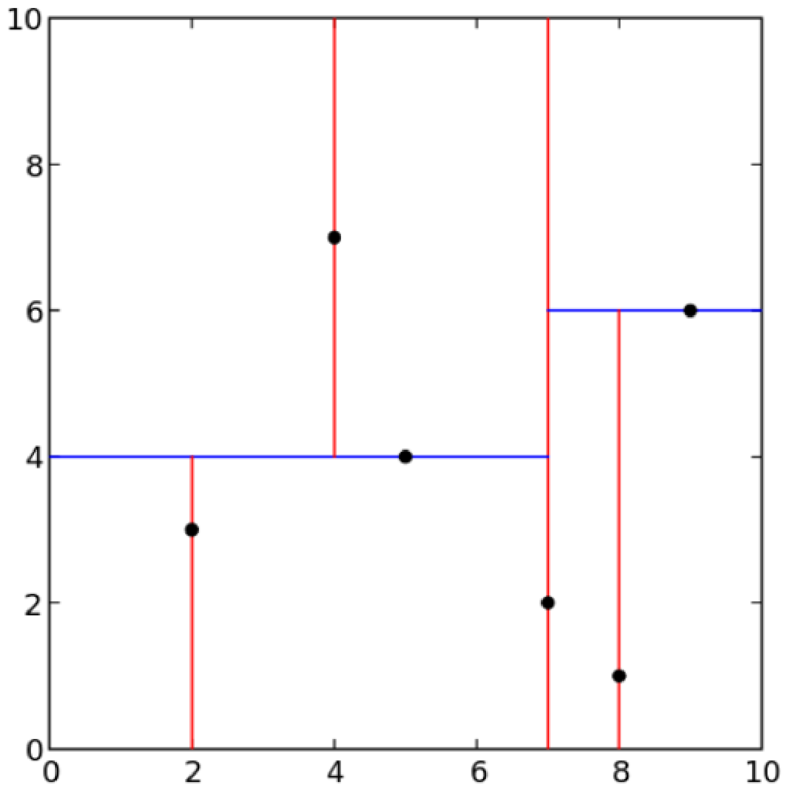

Figure 2: 2d-k-d-tree

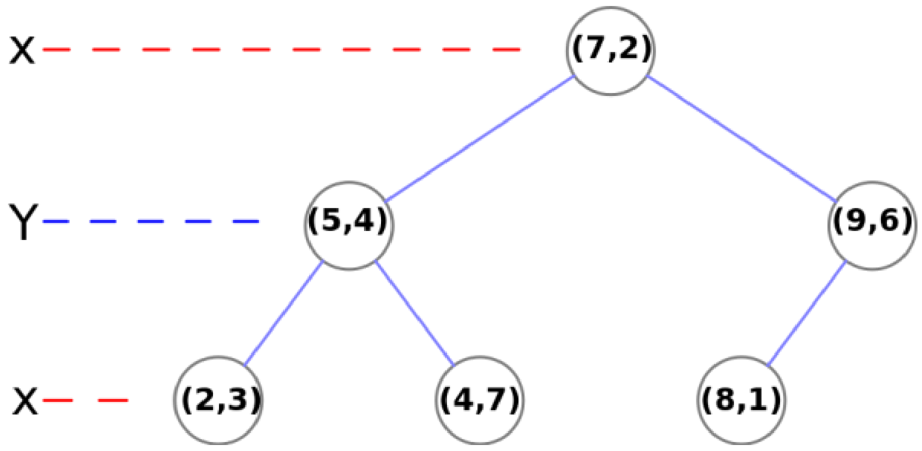

Figure 3: corresponding-binary-tree

Time complexity Steps:

1. O(kn $\log^2$ n). Achieving this speed is dependent on an efficient algorithm for finding the meridian.
2. Approximately O($\log^2$ n), but dependent on size of k.

Results Referring to the previous result graph, we are going to break down

As expected, almost all the time is spent building the tree. Querying for the closest neighbor in the largest tree took less than 0.0015 ms, but 9 seconds is a long time to wait for the tree to build.

The paper *Real-Time KD-Tree Construction on Graphics Hardware - Kun Zhou et al.* offers interesting, although slightly complex, ideas to an efficient parallelization of k-d tree construction. I order to save time, a good amount of time was spent searching for, and trying out, different open source implementations based on this paper. This search was unsuccessful. All the implementations we managed to find was problematic due to lack of updates, often not updated since 2011, and still running on CUDA 4.1, lack of documentation, lack of generalization or dubious source code.

A more uplifting find was several references to *Real-Time KD-Tree Construction on Graphics Hardware* in material published by NVIDIA, regarding their proprietary systems for ray tracing. A graphics rendering technique often reliant on k-d trees, and indeed dependent on high performance.

Our focus therefore turned to implementing and parallelizing the algorithm our self. This has proven to be a quite challenging task. One might think that you would just would split the workload over new processors every time a split occur

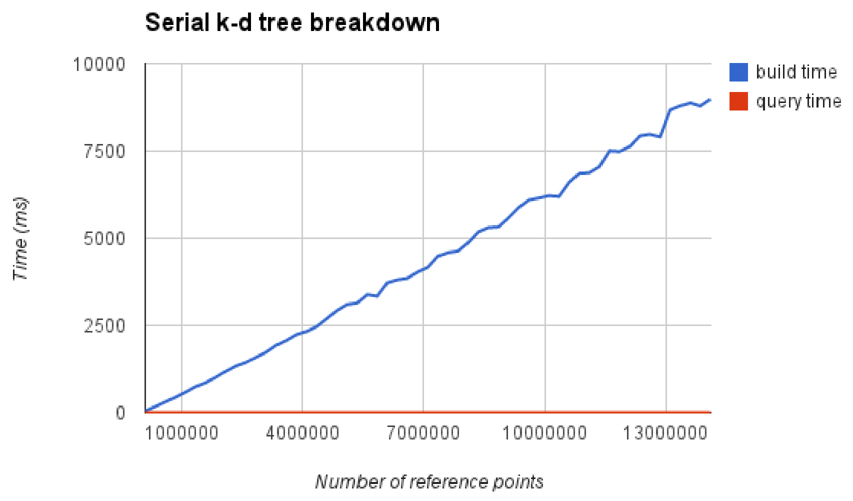

Figure 4: serial-k-d-tree-breakdown

in the recursive algorithm. This would give a speed increase, but you still have to process all the nodes in the root node, requiring at least O(n) time.

Another option could be to build several small trees on different processes, but then you would get an large time penalty when trying to combine the different sub-trees.

Further work

- Try out a heuristic and parallel method for determining the median point.
- Parallelize the code according to one of the simple strategies.
- Further investigate the strategies used in *Real-Time KD-Tree Construction on Graphics Hardware*

Final thoughts

We have discovered a possible relevant paper, CUKNN: A parallel implementation of K-nearest neighbor on CUDA-enabled GPU, behind a pay-wall we cannot access with our student accounts. Maybe you have access to this Ole Ivar?