

NearptD: A Parallel Implementation of Exact Nearest Neighbor Search using a Uniform Grid

David Hedin and W. Randolph Franklin, Rensselaer Polytechnic Institute, Troy NY USA

3 August 2016

Outline

Summary

- ▶ We preprocess millions of fixed points in E^2 to E^6 ,
- ▶ then perform nearest point queries against them.
- ▶ often much faster than previous solutions.
- ▶ exploit Nvidia GPU parallelism.
- ▶ data structure: uniform grid.
- ▶ broader lesson: simple data structures can beat sophisticated ones.

Applications

- ▶ collision detection
- ▶ computer vision
- ▶ machine learning
- ▶ DNA sequencing
- ▶ plagiarism detection

Prior art – FLANN

- ▶ machine learning
- ▶ approximate nearest neighbors
- ▶ included in OpenCV
- ▶ can distribute data to multiple processors with MPI and combine queries
- ▶ however, problems with CUDA.

Parallel computing is underappreciated.

- ▶ Almost all processors, even my smart phone, are parallel.
- ▶ Algorithms that don't parallelize are obsolete.
- ▶ One Xeon core is 20x more powerful than one CUDA core.
- ▶ Nvidia GPUs are almost ubiquitous.

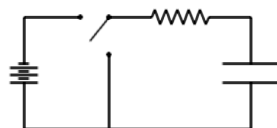
Example of current workstation

Just delivered to me for \$10K:

- ▶ dual 14-core Intel Xeon (= 56 hyperthreads).
- ▶ 256GB memory
- ▶ Intel Xeon Phi
- ▶ \$600 NVidia GTX 1080 GPU
 - ▶ 8GB memory,
 - ▶ 2560 CUDA cores
 - ▶ 9TFLOPS
 - ▶ is an export-controlled munition in USA!
- ▶ servers with many TB of memory are available.

Why use parallel HW?

- ▶ More processing \rightarrow faster clock speed.
- ▶ Faster \rightarrow more electrical power. Each bit flip (dis)charges a capacitor through a resistance.
- ▶ Faster \rightarrow requires smaller features on chip
- ▶ Smaller \rightarrow **greater** electrical resistance !
- ▶ $\Rightarrow \Leftarrow$.
- ▶ Serial processors have hit a wall.



Parallel HW features

- ▶ IBM Blue Gene / Intel / NVidia GPU / other
- ▶ Most laptops have NVidia GPUs.
- ▶ Thousands of cores / CPUs / GPUs
- ▶ Lower clock speed 750MHz vs 3.4GHz
- ▶ Hierarchy of memory: small/fast \rightarrow big/slow
- ▶ Communication cost \gg computation cost
- ▶ Efficient for blocks of threads to execute SIMD.
- ▶ OS, per 6/2013 <http://top500.org> :



runs on 187th fastest machine



& variants run on 1st through 186th.

Thrust

- ▶ C++ template library for CUDA based on STL.
- ▶ Functional paradigm: can make algorithms easier to express.
- ▶ Hides many CUDA details: good and bad.
- ▶ Powerful operators all parallelize: scatter/gather, reduction, reduction by key, permutation, transform iterator, zip iterator, sort, prefix sum.
- ▶ Surprisingly efficient algorithms like bucket sort.
- ▶ Possible back ends: CUDA, OpenMP, sequential on host.

NearptD Preprocessing step

- ▶ Given N_f fixed points in $[0, 1)^d$
- ▶ Choose scaling parameter G_r , often set $G_r = 1$.
- ▶ Choose grid resolution $G = G_r N_f^{1/d}$.
- ▶ Overlay grid of $G \times G \times \dots \times G$ cells on the data.
- ▶ Each grid cell will contain set of points in that range.
- ▶ E.g., if $N_f = 100$, $d = 2$, then $G = 10$ and one cell will contain points in $[.5, .6) \times [.1, .2)$.
- ▶ Store either
 - ▶ the point coordinates themselves or
 - ▶ the IDs, with coordinates in a separate array.

Statistics

- ▶ (This analysis is input sensitive.)
- ▶ Assume fixed points are i.i.d.
- ▶ Let λ be mean number of points per cell.
- ▶ If $G_r = 1$, then $\lambda = 1$.
- ▶ Number of points in a cell is a Poisson random variable.
- ▶ Experimentally the algorithm works, more slowly, when i.i.d. assumption is relaxed.
- ▶ Competing algorithms are also slower.

Uniform grid implementation

- ▶ Abstract data type is a G^d array of sets; operations:
 - ▶ insert, and then
 - ▶ after all insertions, output-all-elements.
- ▶ We want efficiency with millions of points, not just asymptotic rate-of-growth efficiency.
- ▶ STL array vector is wrong:
 - ▶ overhead of each vector is annoying.
 - ▶ memory heap allocation time is superlinear in number of objects
 - ▶ memory management doesn't parallelize well
- ▶ Linked lists are not excellent
 - ▶ overhead of each element is annoying.
 - ▶ managing many lists in parallel is annoying.
 - ▶ although complicated mitigations are possible.
- ▶ Ragged array is excellent.
- ▶ Thrust bucket sort is excellent.

Preprocessing for uniform grid

- ▶ Use Thrust to sort points by cell number.
- ▶ parallel sorting is fast.
- ▶ radix sort is *linear* time.
- ▶ GPU has fast scatter/gather.
- ▶ Access the cells with a ragged array dope vector.

Uniform Grid Qualities

- ▶ **Major disadvantage:** It's so simple that it apparently cannot work, especially for nonuniform data.
- ▶ **Major advantage:** For the operations I want to do (nearest points, intersection, containment, etc), it works very well for any real data I've ever tried.

Other applications

- ▶ volume of union of many cubes
- ▶ 2D planar graph overlay
- ▶ (in progress) 3D mesh overlay, using rational numbers, Simulation of Simplicity, uniform grid, OpenMP, local topological formulae.

Query – fast case

Given query point q , to find closest fixed point f_i :

- ▶ Find grid cell c containing q .
- ▶ Find closest point p in c to q (or c may be empty).
- ▶ However, another neighbor cell may contain a closer point.
- ▶ If c is not empty, then if a closer point exists, it will be in a cell adjacent to c .
- ▶ So, test all points in the $3^d - 1$ adjacent cells.
- ▶ $3^d - 1$ could be optimized.

Expected time: constant.

Query – slow case

- ▶ If c is empty, then spiral out from c until c' , a nonempty cell, is found.
- ▶ Continue spiraling out to cell c'' , the last cell whose closest point to q is no farther than the farthest point of c' .
- ▶ This spiral order, and the stopping cells, are precomputed, out to a large radius.
- ▶ Under fixed point i.i.d. assumption, expected time is constant.

Query – exhaustive case

- ▶ Perhaps all cells in spiral search order are empty.
- ▶ Then exhaustively test all points.

Real world data is not uniform

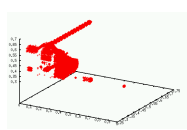
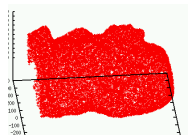
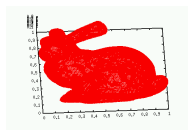
- ▶ e.g., scanned objects, CAD models.
- ▶ everyone is slower.
- ▶ empty uniform grid cells cheaper than empty k-d tree cells.
 - ▶ in both time and space.
- ▶ could get theoretical, e.g, data satisfies a Lipschitz condition
 - ▶ e.g., points do not get exponentially close as $n \rightarrow \infty$
- ▶ better just to try real data
- ▶ we used the worst data we could find.
- ▶ an adversary can make any algorithm slow, but perhaps not with real data.

Test data

3 types

- ▶ uniform random hypercube of points, $N_f \leq 10^8$.
- ▶ CAD model of powerplant from UNC-CH, $N_f = 5\,423\,053$.
- ▶ surface models of objects from

Stanford Digital Michelangelo Project Archive, Stanford Computer Graphics Laboratory, Clemson's Stereolithography Archive, Georgia Tech, Visible Human Project, $N_f \leq 184\,098\,599$.

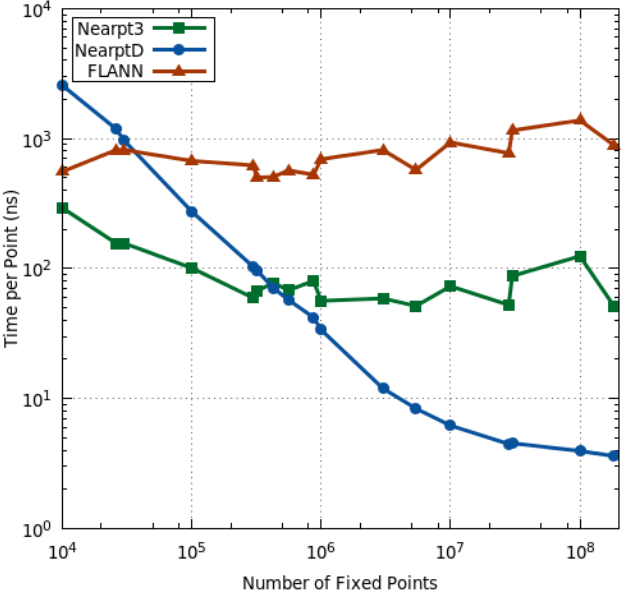


Note on times

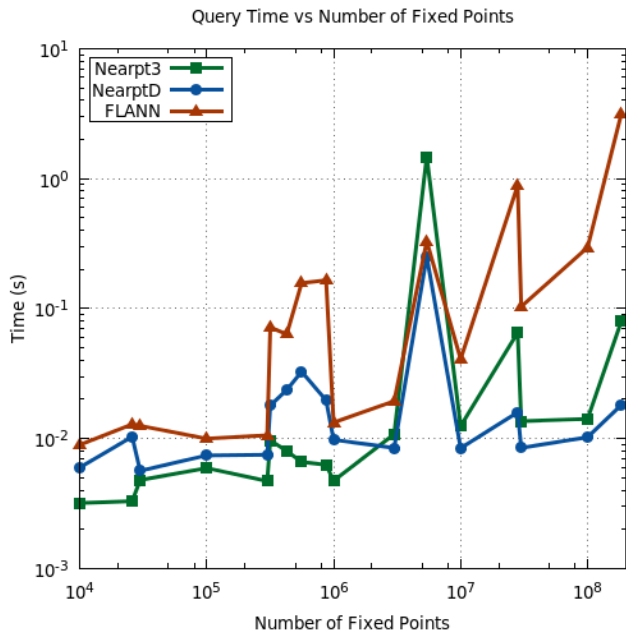
- ▶ Execution times can depend on
 - ▶ what you choose to time
 - ▶ biases in implementor
 - ▶ compiler
 - ▶ poorly documented HW properties
 - ▶ was the program just run earlier?
 - ▶ the data.
 - ▶ phase of the moon...
- ▶ However, large differences are meaningful.
- ▶ This is why asymptotic time analysis is popular
 - ▶ but it is too coarse; you have to test.

Preprocessing time, real data, 3D

Time per Fixed Point vs Number of Fixed Points

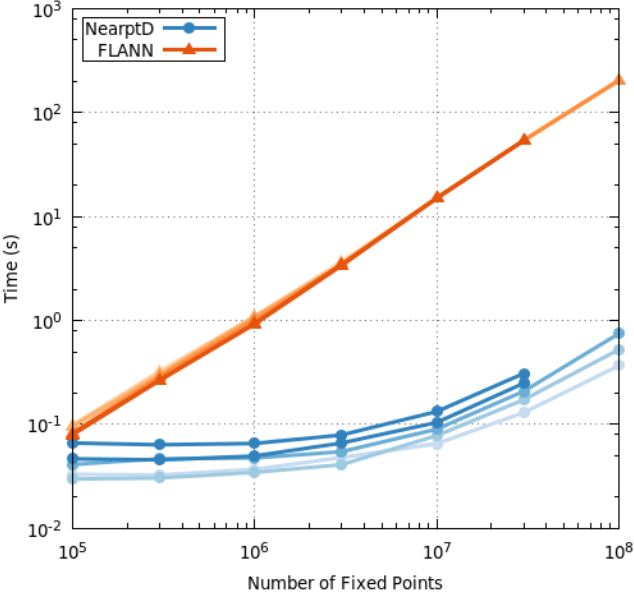


Query time, real data, 3D

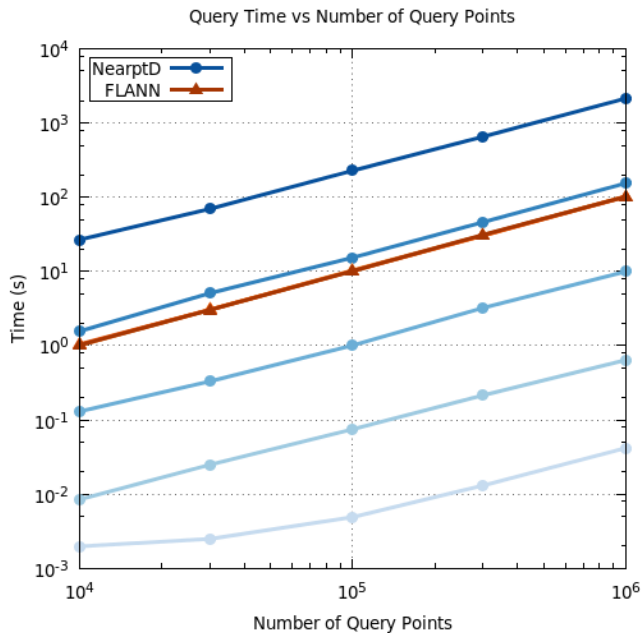


Preprocessing time, uniform points, 2D to 6D

Fixed Time vs Number of Fixed Points



Query time, uniform points, 2D to 6



Conclusions

- ▶ FLANN is better for hi dimensional machine learning queries.
- ▶ NearptD is better for preprocessing in any d and for querying real geometric data.
- ▶ Simple data structures can work well; other examples:
 - ▶ virtual memory page replacement
 - ▶ hashing
 - ▶ bin packing

Future

- ▶ optimize more, both FLANN and NearptD, esp in parallel.
- ▶ find more applications.
- ▶ SW is freely available on GitHub for nonprofit research and education.
 - ▶ please stress-test it.