

THE FLORIDA STATE UNIVERSITY  
COLLEGE OF ARTS AND SCIENCE

MULTI-GPU SOLUTIONS OF GEOPHYSICAL PDES WITH RADIAL BASIS  
FUNCTION-GENERATED FINITE DIFFERENCES

By

EVAN F. BOLLIG

A Dissertation submitted to the  
Department of Scientific Computing  
in partial fulfillment of the  
requirements for the degree of  
Doctor of Philosophy

Degree Awarded:  
Fall Semester, 2012

Evan F. Bollig defended this dissertation on October 1, 2012.

The members of the supervisory committee were:

Gordon Erlebacher  
Professor Directing Thesis

Natasha Flyer  
Outside University Representative

Mark Sussman  
University Representative

Dennis Slice  
Committee Member

Ming Ye  
Committee Member

Janet Peterson  
Committee Member

The Graduate School has verified and approved the above-named committee members, and certifies that the dissertation has been approved in accordance with the university requirements.

# TABLE OF CONTENTS

List of Tables . . . . .	iv
List of Figures . . . . .	v
Abstract . . . . .	vii
<b>1 An Alternative Stencil Generation Algorithm for RBF-FD</b>	<b>1</b>
1.1 $k$ -D Tree . . . . .	2
1.2 A Fixed-Grid Hashing Algorithm . . . . .	5
1.2.1 $k$ -NN with Fixed-Grid . . . . .	7
1.2.2 Performance . . . . .	7
1.2.3 ANN . . . . .	10
1.2.4 $k$ -NN vs $k$ -ANN . . . . .	11
1.2.5 Integer Dilation and Node Reordering . . . . .	12
1.3 Performance Comparison . . . . .	13
1.3.1 Hashing . . . . .	14
1.4 On Space Filling Curves and Other Orderings . . . . .	20
1.4.1 Integer Dilation . . . . .	20
1.5 Conclusions on Stencil Generation . . . . .	21

## **LIST OF TABLES**

## LIST OF FIGURES

<p>1.1 A stencil center in green finds neighboring stencil nodes in blue. Two ball queries are shown as dashed and dash-dot circles to demonstrate the added difficulty of finding the right query radius to obtain the <math>k</math>-nearest neighbors. . . . .</p> <p>1.2 An example <math>k</math>-D Tree in 2-Dimensions. Nodes are partitioned with a cyclic dimension splitting rule (i.e., splits occur first in <math>X</math>, then <math>Y</math>, then <math>X</math>, etc.); all splits occur at the median node in each dimension. . . . .</p> <p>1.3 Two example space filling curves to linearize the same fixed-grid. Left: Raster-ordering (<math>ijk</math>); Right: Morton-/Z-ordering. . . . .</p> <p>1.4 High level overview of the <i>Fixed-Grid Bucket</i> (a.k.a. Hash) Algorithm. A coarse regular grid is overlaid on the domain. Nodes coordinates are hashed to containing cell indices and pushed onto a list for the appropriate cell. The cells are reordered in memory according to a space filling curve (i.e., Raster (IJK), Morton (Z), Graycode (U), etc.). Stencil queries start search with the cell containing the stencil center and expand to neighboring cells until at least <math>n</math> candidate nodes are found. The candidate list is truncated to the <math>n</math> closest neighbors. . . . .</p> <p>1.5 A stencil generated with <math>k</math>-ANN satisfies the required stencil size, but is not guaranteed to choose the true nearest neighbors. . . . .</p> <p>1.6 Example effects of node reordering within neighbor query algorithm for MD node set <math>N = 6400</math> with <math>n = 50</math>. Matrix is 0.78% full. k-D Tree maintains original ordering of the nodes and deceptively appears nearly dense. LSH algorithm reorders nodes according to raster ordering and reveals sparsity of the problem. . . . .</p> <p>1.7 In order: a) node ordering test cases; b) original ordering of regular grid (raster); c) coarse grid overlay for hash functions (<math>hnx = 6</math>); d) example stencil (<math>n = 31</math>) spanning multiple Z's; e) spy of DM after orderings. Author's Note: <u>REGENERATE FIGURES WITH RANDOM/HALTON NODES</u> . . . . .</p> <p>1.8 In order: a) node ordering test cases; b) original ordering of regular grid (raster); c) coarse grid overlay for hash functions (<math>hnx = 6</math>); d) example stencil (<math>n = 31</math>) spanning multiple Z's; e) spy of DM after orderings. . . . .</p>	<p style="text-align: right;">2</p> <p style="text-align: right;">4</p> <p style="text-align: right;">7</p> <p style="text-align: right;">10</p> <p style="text-align: right;">12</p> <p style="text-align: right;">13</p> <p style="text-align: right;">15</p> <p style="text-align: right;">15</p>
--	--

1.9	Querying the $n = 50$ nearest neighbors on a regular grid up to $N = 160^3$ demonstrates the significant gains achieved by our spatially binned neighbor query. While KDTree queries grow as $O(N \log N)$ . . . . .	16
1.10	Querying the $n = 50$ nearest neighbors on a regular grid up to $N = 160^3$ demonstrates the significant gains achieved by our spatially binned neighbor query. While KDTree queries grow as $O(N \log N)$ . . . . .	17
1.11	Querying the $n = 50$ nearest neighbors on a regular grid up to $N = 160^3$ demonstrates the significant gains achieved by our spatially binned neighbor query. While KDTree queries grow as $O(N \log N)$ . . . . .	18
1.12	Generating stencils for increasing subsets of the $N = 1e6$ CVT nodes mesh. . . . .	18
1.13	Based on the proper choice of overlay resolution, the hash stencil query can accelerate stencil generation, but the sophistication of the algorithm is low enough that negative impact is more likely. On the other hand, the impact on SpMV performance is always positive with the routine accelerated up to 4.9x faster. . . . .	19
1.14	As the coarse grid resolution increases the hashing algorithm achieves both 2x faster than KDTree in stencil generation, with greater than 4x gain in SpMV performance (for free). . . . .	19
1.15	In order: a) node ordering test cases; b) original ordering of regular grid (raster); c) coarse grid overlay for hash functions ( $hnx = 6$ ); d) example stencil ( $n = 31$ ) spanning multiple Z's; e) spy of DM after orderings. . . . .	20

# ABSTRACT

Many numerical methods based on Radial Basis Functions (RBFs) are gaining popularity in the geosciences due to their competitive accuracy, functionality on unstructured meshes, and natural extension into higher dimensions. One method in particular, the Radial Basis Function-generated Finite Differences (RBF-FD), has drawn significant attention due to its comparatively low computational complexity versus other RBF methods, high-order accuracy (6th to 10th order is common), and parallel nature.

Similar to classical Finite Differences (FD), RBF-FD computes weighted differences of stencil node values to approximate derivatives at stencil centers. The method differs from classical FD in that the test functions used to calculate the differentiation weights are  $n$ -dimensional RBFs rather than one-dimensional polynomials. This allows for generalization to  $n$ -dimensional space on completely scattered node layouts.

Although RBF-FD was first proposed nearly a decade ago, it is only now gaining a critical mass to compete against well known competitors in modeling like FD, Finite Volume and Finite Element. To truly contend, RBF-FD must transition from single threaded MATLAB environments to large-scale parallel architectures.

Many HPC systems around the world have made the transition to Graphics Processing Unit (GPU) accelerators as a solution for added parallelism and higher throughput. Some systems offer significantly more GPUs than CPUs. As the problem size,  $N$ , grows larger, it behooves us to work on parallel architectures, be it CPUs or GPUs. In addition to demonstrating the ability to scale to hundreds or thousands of compute nodes, this work introduces parallelization strategies that span RBF-FD across multi-GPU clusters. The stability and accuracy of the parallel implementations are tested in explicit and implicit modes to verify correctness.

This work establishes RBF-FD as a contender in the arena of distributed HPC numerical methods.

# CHAPTER 1

## AN ALTERNATIVE STENCIL GENERATION ALGORITHM FOR RBF-FD

Like all RBF methods, RBF-FD is designed to handle irregular node distributions, and the emphasis in literature focuses on how the method manages point clouds. While nothing prevents implementations of RBF-FD from utilizing existing meshes/lattices, most work in the field concentrates on simple geometries like the sphere [6, 7, 8, 9] and quadrilateral/prism [?] to better understand properties of the method and develop extensions.

Without mesh/lattice connectivity available, stencils are generated by choosing the  $n$ -nearest neighbors to each node, including the query node. This is known more formally as a *k-nearest neighbor (k-NN)* problem [24] (a.k.a.  $\ell$ -nearest neighbor search [27]). Here “nearest” is defined with the Euclidean distance metric, although it is possible to generalize to other metrics (see e.g., [1]).

In comparison to the RBF-FD method, global RBF methods with infinite support connect all nodes to all other nodes, so there is no need for neighbor queries. On the other hand, compact RBF methods require all nodes—with no limit on the count—that lie within the support/radius of the RBF centered at each node. This type of neighbor query is referred to as a *ball query* (a.k.a. range query [27]) due to the closed ball created by the radius of support in Equation ??.

The *k*-NN and ball query share many similarities, but the former is harder to solve. Consider, for example, the scenario in Figure 1.1. Two ball queries around a green stencil center are represented as dashed and dash-dot circles. The inner query returns four neighbors, and the outer returns six. If a stencil of size  $n = 6$  is desired, then the outer query can be truncated to give the five required neighbors shown in blue. In this example the red node and the farthest blue node are equidistant from the center, and ties are broken arbitrarily. Although *k*-NN is simply a truncated ball query, the real challenge lies in finding the proper search radius to enclose at least the  $k$  desired neighbors. To find the radius in practice depends on the choice of data structure used to access node locations.

A naïve approach to neighbor queries would be a brute-force search that checks distances from all nodes to every other node. Obviously the cost of such a method is high:  $O(N^2)$  for all stencils. Multi-dimensional data structures can limit the scope of searching and reduce the cost of stencil generation to  $O(N \log N)$ .

For the most part, investigations in RBF communities that delve into efficient neighbor queries are limited methods that depend on ball queries. For example, the Partition of Unity



Figure 1.1: A stencil center in green finds neighboring stencil nodes in blue. Two ball queries are shown as dashed and dash-dot circles to demonstrate the added difficulty of finding the right query radius to obtain the  $k$ -nearest neighbors.

method for approximation (e.g., [26, 27]), and particle methods like the Fast Multipole Method (e.g., [14, 28]) or Smoothed Particle Hydrodynamics (e.g., [16]). Examples of fast algorithms employed in these fields include the fixed-grid method [16, 27],  $k$ -D Trees [27], Range Trees [26, 27], and  $2^d$ -Trees (i.e., Quad- and Octrees) [14, 28]. Surprisingly, while other communities continue the quest for fast neighbor queries, RBF collocation and RBF-FD communities have been slow on the uptake. For many years, the standard in the community has been to use  $k$ -D Trees (see e.g., [5, 7, 10]).

This chapter considers the use of an alternative approach to generate stencils for RBF-FD. It is based loosely on the fixed-grid method from [16]. [22] would classify the algorithm as a *fixed grid “bucket” method with one-dimensional spatial ordering*. However, due to the use of spatial hashing to order nodes in memory it is referred to here as the *Hash Algorithm*.

The Hash Algorithm loosens the requirements for  $k$ -nearest neighbors ( $k$ -NN) stencils to allow  $k$ -“approximately nearest” neighbors ( $k$ -ANN). It also reorders nodes according to space-filling curves. In what follows, the Hash Algorithm is compared to an efficient implementation of  $k$ -D Tree ([24]). Benchmarks demonstrate that with the proper choice of parameters the Hash Algorithm is up to 2x faster than  $k$ -D Tree, and it comes with a free bonus: up to 5x faster SpMV due to the impact of spatial reordering that occurs during stencil generation.

## 1.1 $k$ -D Tree

A  $k$ -D Tree is a spatial data structure that generally decomposes a space/volume into a small number of cells. All  $k$ -D Trees are binary and iteratively subdivide volumes and sub-volumes at each level into two parts. The “ $k$ ” in  $k$ -D Tree refers to the dimensionality of the data/volume partitioned—that is  $k \equiv d$ .

Given a set of points bounded by a  $d$ -dimensional volume, a  $k$ -D Tree applies a hierarchy of  $(d - 1)$ -dimensional axis aligned *splitting planes* to cut the space. At each level of the hierarchy the splitting planes result in two new *half-planes* [23]. Consecutive splits intercept

one another at a *splitting value*.  $k$ -D Trees do not require that half-planes equally subdivide a volume; more often it is the data contained within the volume that is equally partitioned. The choice of dimension for the splitting plane, in conjunction with a variety of methods for choosing the splitting values allows for many flavors of  $k$ -D Trees (see e.g., [4, 22, 23] for comprehensive lists). *Point k-D Trees*,  $2^d$  *Trees* (i.e., quad-/octrees), *BSP-Trees*, and *R-trees* are all members of the general  $k$ -D Tree class [23, 28].

This work considers *Point k-D Trees* [22], which partition a set of discrete points/nodes as outlined by the recursive procedure in Algorithm 1.1. Point  $k$ -D Trees assume that splitting planes intercept nodes rather than occur arbitrarily along the half-plane. The splitting value at each level of the tree is set to the *median coordinate* of the points in the half-plane, which ensures the tree is well balanced on initial construction. All nodes with coordinate (in the current dimension) less than or equal to the splitting value are contained by the left half-plane, and all nodes with coordinate greater than the splitting value are contained by the right. Half-planes containing only one element correspond to leaves of the tree. The median coordinate of a half-plane is found by sorting the  $n$  node coordinates contained by the partition and selecting the  $\lceil \frac{n}{2} \rceil$ -th element [4].

---

**Algorithm 1.1** BuildKDTree( $P$ ,  $depth$ )

---

- 1: **Input:** A set of  $d$ -dimensional points  $P$  and the current  $depth$ .
- 2: **Output:** The root of the  $k$ -D Tree for  $P$ .
- 3:
- 4: **if**  $\text{size}(P) = 1$  **then**
- 5:     **return** a new leaf storing  $P$
- 6: **end if**
- 7:  $L_i := \text{median}(\text{coord}(P, depth))$
- 8:  $v_l := \text{BuildKDTree}(\text{coord}(P, depth) \leq L_i, (\text{depth} + 1) \text{ modulo } d)$
- 9:  $v_r := \text{BuildKDTree}(\text{coord}(P, depth) > L_i, (\text{depth} + 1) \text{ modulo } d)$
- 10: **return** A new node  $v := \begin{pmatrix} \text{value} := L_i \\ \text{left} := v_l \\ \text{right} := v_r \end{pmatrix}$

---

The  $k$ -D Tree in Figure 1.2 is an example of a Point  $k$ -D Tree. Given a set of eight nodes in two dimensions, the tree is constructed by applying one-dimensional cuts along the  $x$ -dimension, then the  $y$ -dimension, then back to the  $x$ -dimension. This approach is referred to as *cyclic splitting*, as consecutive cuts are applied by iterating dimensions in a round-robin fashion [22]. The first cut,  $L_1$ , shown in green, splits the nodes into two sets on either side of  $A$ . The corresponding tree in the center of Figure 1.2 shows  $L_1$  as the tree root with all nodes having  $x$ -coordinates less than or equal to  $A$  to the left, and all nodes having  $x$ -coordinates greater than  $A$  to the right. The second level of the tree,  $L_2$  and  $L_3$  (in blue), splits the half-planes on either side of  $A$  at nodes  $B$  and  $C$ . The axis parallel splits for each half-plane intercept  $L_1$  independently to partition half-planes along the  $y$ -dimension; once again, nodes with coordinates less than or equal (i.e., below the cut) to the splitting value branch left in the tree, and  $y$ -coordinates greater than (i.e., above) the value branch right. The third level (red) returns to splitting half-planes in the  $x$ -dimension. Nodes  $D$  and  $H$  are not intersected by a splitting plane; their half-planes contain only one

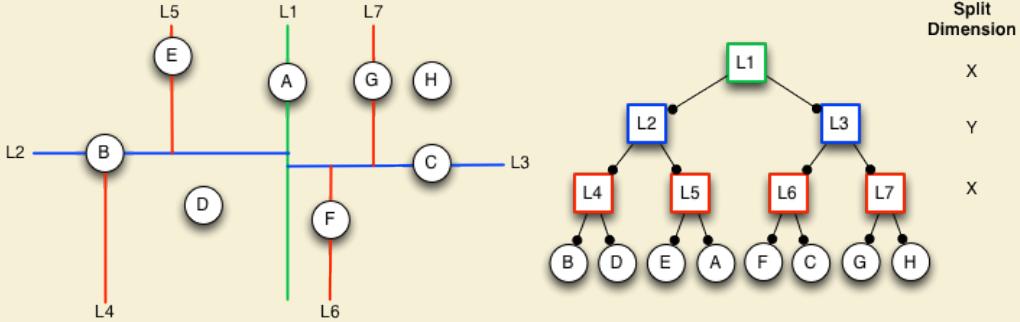


Figure 1.2: An example  $k$ -D Tree in 2-Dimensions. Nodes are partitioned with a cyclic dimension splitting rule (i.e., splits occur first in  $X$ , then  $Y$ , then  $X$ , etc.); all splits occur at the median node in each dimension.

node so they immediately become leaves of the tree. This process to build a Point  $k$ -D Tree has a complexity of  $O(N \log N)$  with  $O(N)$  storage required [4, 22].

Frequently, the terms  *$k$ -D Tree* and *Point  $k$ -D Tree* are used synonymously by the RBF community (see e.g., [5, 7, 10]); the same is convention adopted here.

Generating an RBF-FD stencil with a  $k$ -D Tree can be efficiently accomplished in  $O(n \log N)$  time—where  $n$  is the stencil size—following an approach introduced in [11], and presented in Algorithm 1.2. The  $k$ -NN search starts a depth-first recursive search of the  $k$ -D Tree to find the nearest neighbor to a query point,  $X_q$ . Traversal of the tree occurs by following branches left or right based on comparison of  $X_q$  coordinates to the splitting value stored at each node of the tree, with the objective to find the smallest half-plane containing  $X_q$ . The search traverses the height of the tree in  $O(\log N)$  steps to find the leaf that stores the nearest neighbor to  $X_q$ . The neighbor point and its distance from  $X_q$  are inserted into a global priority queue,  $pq$ . Points in the priority queue are sorted in descending order according to distance.

After finding the nearest neighbor the algorithm returns to the previous node in the tree and onto the opposing half-plane (i.e., down the far branch) to look for other leaves. So long as the size of  $pq$  is at less than capacity ( $n$ ) the search automatically adds points to the priority queue. If  $pq$  reaches capacity the algorithm starts to pop off excess points with the understanding that the action removes those points farthest from  $X_q$ .

In order to prune branches from the search and reduce complexity, Algorithm 1.2 makes use of a routine called “BoundsOverlapBall”, which checks if any boundaries of the current level half-plane intersect/overlap with a closed ball centered at  $X_q$ . The ball is given a radius equal to the maximum distance in  $pq$ . Then, if the ball and a boundary intersect, the search will continue onto the half-plane on the opposite side of that boundary. This step handles the possibility that nearer nodes occur within the overlapped region in the other half-plane. If the ball and boundary do not intersect, the opposing half-plane and its related subtree are pruned from the search. Additional details on the implementation of “BoundsOverlapBall” can be found in [11, 25].

The authors of [11] find Algorithm 1.2 capable of efficiently querying the  $n$ -nearest neighbors with a complexity proportional to  $O(\log N)$  (dominated by the cost of tree traversal).

---

**Algorithm 1.2** KNNSearchKDTree( $X_q, n, root, depth$ )

---

- 1: **Input:** A query node  $X_q$ , number of desired neighbors ( $n$ ), the current *root* of the  $k$ -D Tree, and the current *depth* of traversal.
- 2: **Output:** A global priority queue,  $pq$ , containing the  $n$ -nearest neighbors to  $X_q$  sorted by distance from  $X_q$  in descending order.
- 3: **Assume:** A routine named “BoundsOverlapBall” exists to determine if the boundaries of the current half-plane are intersected by the ball centered at  $X_q$  with radius equal to the maximum distance in  $pq$ . As long as  $pq.size < n$ , “BoundsOverlapBall” defaults to true.

- 4:
- 5: **if** *root* is leaf **then**
- 6:     Insert  $\{root, \text{dist}(X_q, root)\}$  into  $pq$
- 7:     **if**  $pq.size > n$  **then**
- 8:          $pq.pop$  ▷ Keep only  $n$ -nearest neighbors
- 9:     **end if**
- 10:    **return**
- 11: **end if**
- 12:
- 13: **if**  $\text{coord}(X_q, depth) \leq root.value$  **then**
- 14:     KNNSearchKDTree( $X_q, n, root.left, (depth + 1) \% d$ )
- 15: **else**
- 16:     KNNSearchKDTree( $X_q, n, root.right, (depth + 1) \% d$ )
- 17: **end if**
- 18:
- 19: **if**  $\text{coord}(X_q, depth) \leq root.value$  **then**
- 20:     **if** BoundsOverlapBall( $X_q$ ) **then**
- 21:         KNNSearchKDTree( $X_q, n, root.right, (depth + 1) \% d$ )
- 22:     **end if**
- 23: **else**
- 24:     **if** BoundsOverlapBall( $X_q$ ) **then**
- 25:         KNNSearchKDTree( $X_q, n, root.left, (depth + 1) \% d$ )
- 26:     **end if**
- 27: **end if**
- 28: **return**

---

The relationship between stencil size  $n$ , and grid size,  $N$ , is better expressed as  $O(n \log N)$  for one stencil.

RBF-FD only needs to generate stencils once, so the overall time for the step subsumes the cost of tree construction and  $N$  queries. The resulting overall complexity is proportional to  $O(N \log N)$ .

## 1.2 A Fixed-Grid Hashing Algorithm

While a  $k$ -D Tree functions well for queries, the cost to build the tree structure is unnecessary overhead. Among the many data-structures that exist for nearest neighbor

queries, alternatives like fixed-grid methods [22, 26, 27] (a.k.a. uniform grid [13, 16]) bypass much of the cost in construction with an assumption that only lower spatial dimensions (e.g., 2-D or 3-D) are significant. This discards the need to build a tree and shifts focus onto querying neighbors.

Fixed-grid methods get their name from a coarse 2-D or 3-D regular grid that is overlaid on the domain. The  $d$ -dimensional grid divides the domain’s axis aligned bounding box (AABB)—that is, the minimum bounding box containing the entire domain with edges parallel to axes—into  $(h_n)^d$  cells. Subdivisions are uniform, so one can easily identify the cell containing any sample point,  $p$ , given the coordinates of the AABB and  $(h_n)^d$ . For example, let  $(c_x, c_y, c_z)$  be the desired containing cell in 3-D, and  $(x_{min}, y_{min}, z_{min})$  and  $(x_{max}, y_{max}, z_{max})$  be the minimum and maximum coordinates of the AABB (resp.). Then the cell coordinates are found by:

$$(dx, dy, dz) = \left( \frac{(x_{max} - x_{min})}{h_n}, \frac{(y_{max} - y_{min})}{h_n}, \frac{(z_{max} - z_{min})}{h_n} \right)$$

$$(c_x, c_y, c_z) = \left( \left\lfloor \frac{(p_x - x_{min})}{dx} \right\rfloor, \left\lfloor \frac{(p_y - y_{min})}{dy} \right\rfloor, \left\lfloor \frac{(p_z - z_{min})}{dz} \right\rfloor \right). \quad (1.1)$$

Cells neighboring  $(c_x, c_y, c_z)$  are trivial to find by adding positive and negative offsets to each coordinate. Note that the fixed-grid in this method is virtual. Nodes are matched with cell coordinates, but the full grid is never explicitly constructed.

Fixed grid methods also make use of *space filling curves*. Space filling curves pass through every point in  $d$ -dimensional space, and through each point only once. Equivalently, space filling curves map  $d$ -dimensional space down to 1-D, where every point is converted to a unique index or traversal order based on its spatial coordinates. These mapping properties make space filling curves ideal for use as hash functions. Traversing the  $d$ -dimensional points (i.e., playing “connect the dots”) draws the space filling curve. Figure 1.3 presents two common orderings of a 2-D fixed-grid. Note that one-dimensional orderings are not unique. On the left is a *Raster-ordering* (a.k.a. Scanline- or *ijk*-ordering):  $f(c_x, c_y, c_z) = ((c_x * h_n) + c_y) * h_n + c_z$ . The right half of Figure 1.3 shows an ordering known as Morton- or *Z*-ordering; construction of this ordering is discussed later in this chapter. The lower left corner of each cell in Figure 1.3 indicates the mapped index. Traversing the cells in order produces the curves superimposed in red.

At a high level, fixed-grid methods have the following preprocessing steps [16]:

1. Subdivide the domain with the overlay grid.
2. For each node, identify the containing cell coordinates.
3. For each node, use the cell coordinates as input to a spatial hash function (i.e., a space-filling curve).
4. Sort the nodes according to their spatial hash.

Particular details of how nodes are sorted, the choice of hashing function, etc. determine the specific class of fixed-grid method and corresponding complexity. A comprehensive list of options and classifications can be found in [22].

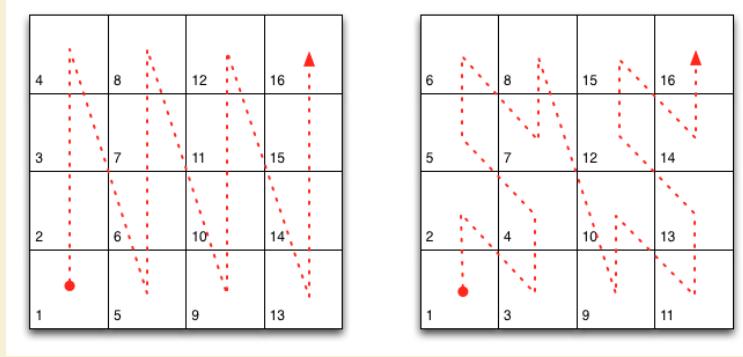


Figure 1.3: Two example space filling curves to linearize the same fixed-grid. Left: Raster-ordering ( $ijk$ ); Right: Morton-/Z-ordering.

The implementation in this work is inspired by fixed-grid approaches for GPU particle simulations [13, 15, 16]. Particle methods require a ball query at each time-step. With time-steps often dominated by the cost of querying neighbors, the community understandably devotes significant effort to finding the most efficient algorithms possible. The fixed-grid method is competitive for at least two reasons: a) by bypassing the need to build a tree, half the cost in querying neighbors is avoided; and b) nodes sorted according to a space filling curve are typically closer in memory to their neighbors, resulting in a higher likelihood that data will be cached when required.

A space filling curve is any path

map multidimensional data into 1-D. The curves traverse all data. When used to index data, space filling curves give a variety of solutions for ordering data. Given the cells of a fixed-grid, a space filling curve assigns a unique index to each cell

Example space filling curves are presented in Figure 1.3

When the fixed-grid is mapped into linear memory, cell coordinates are passed through a hash function to produce a unique index for each cell. Alternatively, one might use another

Space filling curves like the options shown in Figure 1.3

In what follows,

as a *spatial hash function*

For now assume the spatial hash function is the standard Raster- or  $ijk$ -ordering (with  $k$  varying fastest):  $ind = ((c_x * h_n) + c_y) * h_n + c_z$ . To get the coordinates of the cell, simply swap the two  $h_n$  terms with  $d_x$  and  $d_y$ .

The choice of spatial hash function certainly impacts memory access patterns.

### 1.2.1 $k$ -NN with Fixed-Grid

### 1.2.2 Performance

Although RBF-FD only requires neighbor queries once, the results that follow reveal a long lasting positive impact on memory with a fixed-grid method, which is sufficient to justify its use. Investigations into moving node coordinates and/or local refinements for RBF methods (e.g., [6]) would find the fixed-grid method significantly more beneficial. As of this writing no known applications of RBF-FD consider moving nodes

---

**Algorithm 1.3** BuildFixedGrid( $P, h_n$ )

---

- 1: **Input:** A set of  $d$ -dimensional points  $P$ , and the desired resolution of the overlay grid,  $h_{nx}$ .
- 2: **Output:** The sorted set of points  $\hat{P}$ .
- 3:
- 4: Create  $Q$ : an  $(h_n)^d$  array of empty vectors.
- 5: **for** point  $p_i$  in  $P$  **do**
- 6:    $c := \text{CellCoords}(p_i)$
- 7:    $ind := \text{SpatialHash}(c)$
- 8:   Append  $i$  to  $Q[ind]$
- 9: **end for**
- 10: **for** each  $Q[j], j = 0, 1, \dots, (h_n)^d$  **do**
- 11:   **if**  $Q[j]$  is not empty **then**
- 12:     Append  $P[Q[j]]$  to  $\hat{P}$
- 13:     Overwrite  $Q[j]$  with new indices in  $\hat{P}$
- 14:   **end if**
- 15: **end for**
- 16: **return**  $\hat{P}, Q$

---

---

**Algorithm 1.4** QueryFixedGrid( $\hat{P}, Q, h_n$ )

---

- 1: **Input:** A set of  $d$ -dimensional points  $P$ , and the desired resolution of the overlay grid,  $h_{nx}$ .
- 2: **Output:** The sorted set of points  $\hat{P}$ .
- 3: Get  $min$  and  $max$  coordinates from  $P$
- 4: Create a vector of  $(h_n)^d$  empty vectors,  $Q$ .
- 5: **for** point  $p_i$  in  $P$  **do**
- 6:    $c := \text{CellCoords}(p_i)$
- 7:    $ind := \text{SpatialHash}(c)$
- 8:   Append  $i$  to  $Q[ind]$
- 9: **end for**
- 10: **for** each  $Q[j], j = 0, 1, \dots, (h_n)^d$  **do**
- 11:   **if**  $Q[j]$  is not empty **then**
- 12:     Append all (unsorted) nodes in  $P[Q[j]]$  to  $\hat{P}$
- 13:     Overwrite  $Q[j]$  with new indices from  $\hat{P}$
- 14:   **end if**
- 15: **end for**
- 16: **return**  $\hat{P}, Q$

---

Due to the limited significance of stencil generation under RBF-FD, the overhead in implementing and debugging the fixed-grid method on the GPU is difficult to justify. The implementation tested here was developed as a pure CPU prototype with minimal attention to optimization. The added complexity in reproducing the efficient fixed-grid method on the GPU could be the subject of future work for moving nodes.

[16] assumes that at each time-step a new virtual fixed grid is generated

Implementations of fixed-grid often assume that no more than two nodes reside in any one cell.

This implementation is CPU only. Porting the one time stencil generation to the GPU was seen as

use of fixed-grid methods originated due to low overhead before performing queries. Although all three related works focus on offloading simulations to the GPU, the method is also popular for CPU implementations

In [26] and later [27], Wendland presents the fixed-grid method as a low-cost alternative for RBF methods based on Partition of Unity.

[20] adopt the a similar approach that incrementally grows the

Following [22], the method used in this work is classified as a fixed-grid *bucket* method with one dimensional ordering. The emphasis on bucket arises from the application of a coarse grid that allows

implementations of the method leverage radix sort to directly sort nodes based on their spatial hash. The radix sort completely reorders nodes based on their Space filling curves are . Examples include *Z*- or *Morton* ordering, *U*- or *Greycodes*-ordering, *X*-ordering, etc. Hilbert curves have been demonstrated to be the most efficient [? ].

The algorithm chosen for this work is provided in Algorithm ???. The algorithm starts by creating a list of lists representing the  $(h_n)^d$  cells. As nodes are matched with cell indices, the node index is appended to the

These tests assume  $dx = dy = dz$  to ensure spherical stencils.

With regularly spacing it is possible to directly access the  $(i, j, k)$ -th cell in the domain with the formula  $cell = ((h_n * i) + j) * h_n + k$ .

Each cell of the overlay queries are popular in particle-based methods like Smoothed Particle Hydrodynamics [16? ], where neighbors are determined based on spatial proximity in lower dimensions (i.e., 2-D or 3-D). Since RBF-FD stencils are also based on spatial proximity it is possible to

The low cost of both construction and query phases allow methods with moving nodes to perform updated nearest neighbor queries at each iteration without significant overhead.

Since stencil are rarely based on high dimensional data. It suffices to use a

Having witnessed the success of the fixed-grid method in particle-based simulations at the onset of this work ([16, 20? ]), a prototype

The algorithm used in this work is considered a particular

The fixed-grid algorithm

Figure 1.4 clarifies this algorithm presents the variant of the fixed-grid method used in this work from a high-level perspective. Starting with an input set of points, steps 1 through 3 are

[?] introduce a minimal *k*-D Tree for the GPU which

Fixed grid method from a high level. Works great for quasi-regular nodes. Avoids construction with overlaid grid of known cell indices.

The fixed-grid method

An implementation of the method used in [26], discussed in [27] and

For moving nodes, such as in Lagrangian schemes, this cost is prohibitively high. In an attempt to reduce the cost, lagrangian schemes introduced approximate nearest neighbor queries based on

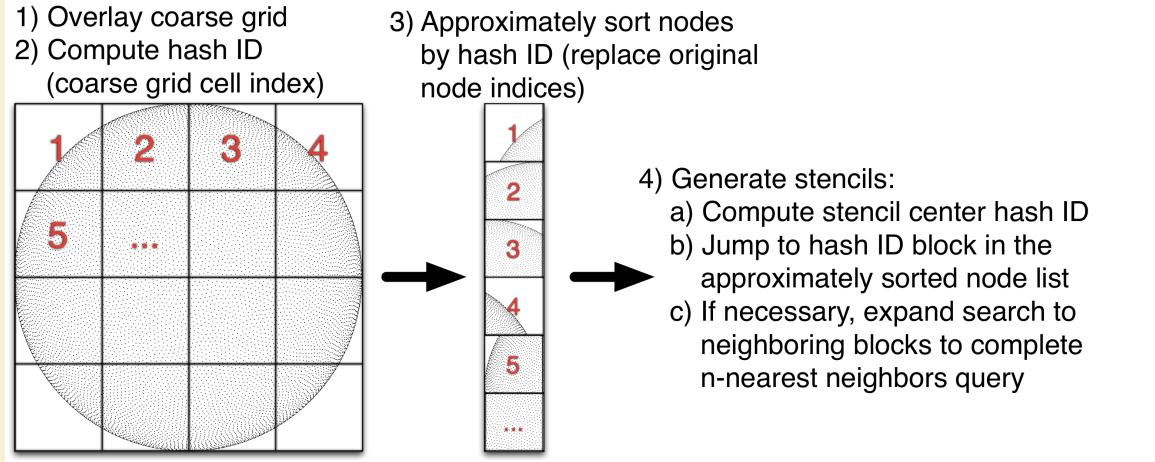


Figure 1.4: High level overview of the *Fixed-Grid Bucket* (a.k.a. Hash) Algorithm. A coarse regular grid is overlaid on the domain. Nodes coordinates are hashed to containing cell indices and pushed onto a list for the appropriate cell. The cells are reordered in memory according to a space filling curve (i.e., Raster (IJK), Morton (Z), Graycode (U), etc.). Stencil queries start search with the cell containing the stencil center and expand to neighboring cells until at least  $n$  candidate nodes are found. The candidate list is truncated to the  $n$  closest neighbors.

The algorithm introduced here is the same as the Fixed-Grid Method in [27].

### 1.2.3 ANN

Our benchmarks and testing focus on the more challenging problem of  $k$ -NN

As part of the preprocessing stage for RBF-FD, the scattered point cloud must be analyzed to generate stencils. To generate a stencil, any collection of nodes can be selected. However, by choosing nodes close to the stencil center and well balanced around it, we stand to get the best possible approximations to derivatives.

Why? well, the approximations are based on differences. Similar to classic FD, draw a secant connecting two nodes of a stencil. The slope of the secant determines the gradient at either point or a point in between. In the limit as the points are moved closer to the same spot, the approximation to the derivative at that point becomes exact.

So ideally every RBF-FD stencil will operate on nearest neighbors.

can be described as a *fixed grid “bucket” method with one-dimensional spatial ordering* as classified by [22].

The implementation runs entirely on the CPU ([16] runs on the GPU).

The  $k$ -D tree implementation compared in this work is the *kd-Tree Matlab* library from Tagliasacchi [24, 25]. Originally posted to the Matlab FileExchange and now maintained as a Google Code project. Until the addition of *KNNsearcher* in Matlab

Tagliasacchi [25] uses the standard split by median in each dimension, with alternating chosen by incrementing the dimension by one, modulo the number of dimensions (e.g., in 2D the order is  $X, Y, X, Y, \dots$ ). Wendland [27] suggests a number of more advanced median cuts such as

[24]. [1]

Hashing, shown in Figure ?? overlays a regular grid. This is equivalent to an axis aligned bounding box AABB, with refinement. In other words, we form a quad-tree in 2D, an octree in 3D. The neighbor query starts with the cell in which  $x_c$  resides. Since we use an axis aligned bounding box, this cell index is easily calculated given the coordinate and number of subdivisions in each dimension. Once the cell index is resolved, the stencil is populated by taking the  $n$  nearest neighbors from within the current cell. If the cell does not contain sufficient number of nodes to fill the stencil, the search for neighbors expands to include the cells immediately adjoining the center cell, taking only the nearest nodes in the provided cells. The search continues to expand outward in a rasterized circle/sphere until  $n$  is satisfied. This search is considered approximate because it can happen that a true nearest neighbor would lie in a cell that is not included in the rasterized circle, and other nodes are substituted from the far reaches of the discretized grid.

The complexity of the method is still higher than the more efficient implementations used by Lagrangian methods, but as demonstrated in Figure ?? the savings are significant. Generating stencils for RBF-FD is a preprocessing cost, so we do not dedicate an excessive amount of attention to this algorithm. However, a few ideas that would improve: hilbert ordering, choose AABB resolution based on  $N$  not user parameters, faster sorting, GPU implementation

In the benchmarks below, a  $k$ -D Tree implementation by Tagliasacchi [24, 25]. The implementation follows the construction process and

it was a popular package offered on the MATLAB FileExchange and garnered attention due to its MEX compiled interface that allowed use as both a C++ and MATLAB library.

Most of the RBF community leverages the  $k$ -D tree, due to its low computational complexity for querying neighbors and its wide availability as standalone software in the public domain (e.g., matlab central has a few implementations for download, and the MATLAB Statistics Toolbox includes an efficient k-D Tree).

The Matlab central  $k$ -D Tree is MEX compiled and efficient. We integrated the standalone C++ code into our library.

#### 1.2.4 $k$ -NN vs $k$ -ANN

Start at the center cell and add the 8 cells immediately surrounding it. Add to this the 16 cells in the second halo for a total of 25 cells.

So while a  $k$ -NN stencil of size  $n = 8$  in Figure 1.1 would contain the black node in the right-most column of the grid. The  $k$ -ANN generated a stencil of size  $n = 8$  would opt for the black node to the top right because it is in the second halo and accepted as “closer” because the stencil can be satisfied without querying the extra 24 cells on the outermost halo.

Therefore, it is not essential that stencils contain only nearest neighbors. Instead, one can acquire the *approximate nearest neighbors*. Figure ?? demonstrates a case where a does not contain all nearest neighbors. As illustrated in the Figure, the ANN stencil and true nearest neighbor stencil differ by one node. THis is not dire

The query is approximate because nodes such as the white node in Figure 1.1 could be included in the stencil before

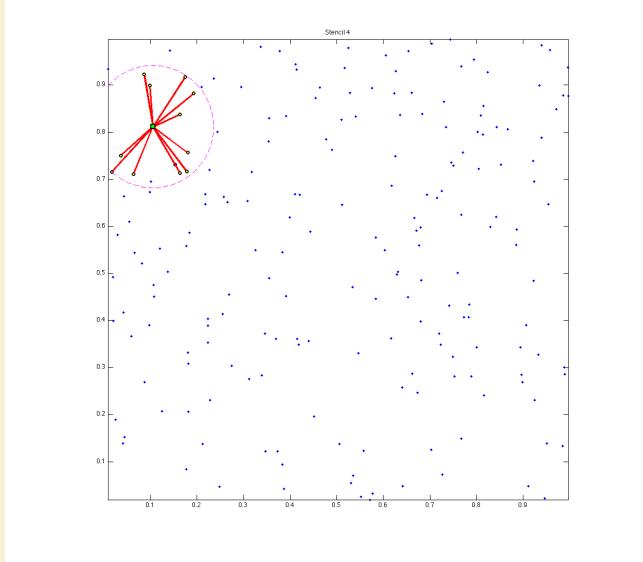


Figure 1.5: A stencil generated with  $k$ -ANN satisfies the required stencil size, but is not guaranteed to choose the true nearest neighbors.

RBF-FD operates on general node distributions. Historically, stencils are uniform in size ( $n$ ) and generated by selecting the  $(n - 1)$  true nearest neighbors to a node  $x_c$ . This is a  $k$ -NN query.

Alternative queries are possible: ball query and approximate nearest neighbor. The approximate is of particular interest because nodes closest to the stencil will always be selected, whereas the nodes further away have minimal influence so swapping out can't hurt. The justification in altering the selection is for reduced complexity in neighbor queries.

### 1.2.5 Integer Dilation and Node Reordering

[18] found that reordering nodes via RCM and space filling curves offer similar benefits in terms of reduced TLB misses and better cache coherency.

[22] labels this type of algorithm as a *fixed grid bucket algorithm* with the twist that it also has *one dimensional ordering*

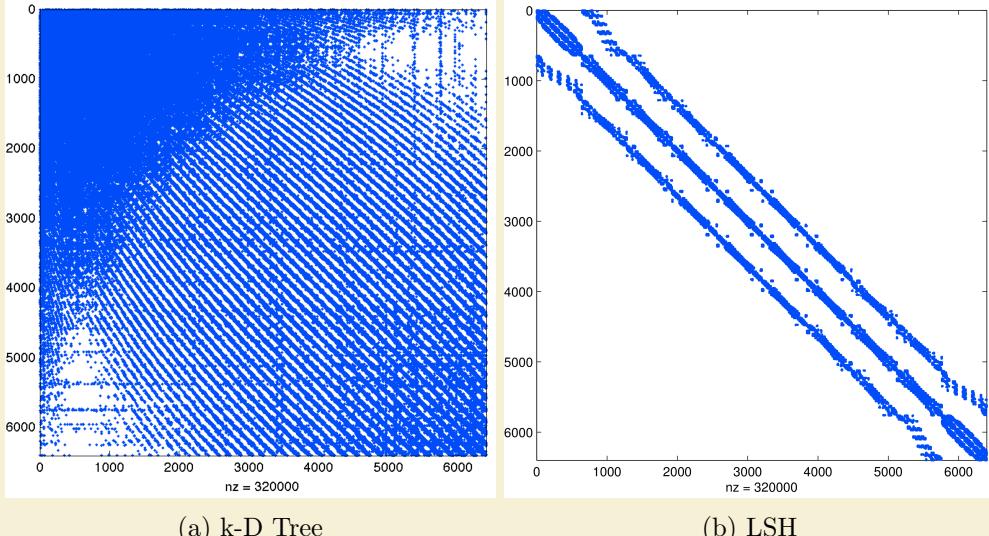


Figure 1.6: Example effects of node reordering within neighbor query algorithm for MD node set  $N = 6400$  with  $n = 50$ . Matrix is 0.78% full. k-D Tree maintains original ordering of the nodes and deceptively appears nearly dense. LSH algorithm reorders nodes according to raster ordering and reveals sparsity of the problem.

### 1.3 Performance Comparison

At the start of this dissertation, the most widely used  $k$ -D Tree implementation in the RBF community was [24]. As a MEX-compiled

- Performance improvement
- SpMV impact
- impact from space filling curves

Many algorithms exist to query the  $k$ -nearest neighbors (equivalently all nodes in the minimum/smallest enclosing circle). Some algorithms overlay a grid similar to Locality Sensitive Hashing and query such as... [? ].

For the purpose of generating RBF-FD stencils the  $k$ D-Tree is built and all queries are made once. There is no reuse of the tree so the build and query times are combined into one.

impact from ordering on matrix sparsity. Bandwidth impact. Bandwidth impact on condition considered in future chapter.

what is best overlay resolution? based on time to generate. choose resolution as  $n/2$ ,  $n/9$ ? etc?

perform neighbor queries on raster order because its easiest to jump cells. then each cell can be reordered according to  $z,x,u$ , etc. for better memory locality. Matlab script to do this (can be ported to C)

[http://www.vincentgarcia.org/data/Garcia\\_2010\\_ICIP.pdf](http://www.vincentgarcia.org/data/Garcia_2010_ICIP.pdf)

GPU version of Locality Sensitive Hashing could reduce complexity further [19]

This can be done efficiently using neighbor query algorithms or spatial partitioning data-structures such as Locality Sensitive Hashing (LSH) and  $k$ D-Tree. Different query algorithms often have a profound impact on the DM structure and memory access patterns. We choose a Raster ( $ijk$ ) ordering LSH algorithm [?] leading to the matrix structure in Figures ?? and ?. While querying neighbors for each stencil is an embarrassingly parallel operation, the node sets used here are stationary and require stencil generation only once. Efficiency and parallelism for this task has little impact on the overall run-time of tests, which is dominated by the time-stepping. We preprocess node sets and generate stencils serially, then load stencils and nodes from disk at run-time. In contrast to the RBF-FD view of a static grid, Lagrangian/particle based PDE algorithms promote efficient parallel variants of LSH in order to accelerate querying neighbors at each time-step [12, 19].

At the onset of our work on RBF-FD, the most commonly used KDTree implementation used by the RBF community was [? ]. Recently, improvements were made to the KDTree algorithm to reduce the cost of building the KDTree to  $O(N \log^2 N)$ .

Figure ?? compares the total time to generate  $N$  stencils of size  $n = 50$  with three methods: [? ], our hash-based neighbor query, and the improved KDTree from [? ]. Until the new release of KDTree, our algorithm was a major improvement to the performance of stencil generation. The hash-based approach achieved greater than

Our work in [2] leveraged an alternative to  $k$ -D tree, based loosely on space-filling curve orderings common in Lagrangian schemes like Smoothed Particle Hydrodynamics (e.g., [? ], [? ]).

### 1.3.1 Hashing

[3] provide a fast parallel

[? ] is working on parallel generation. [? ] has OpenCL neighbor queries

We started with a CPU implementation to test appropriateness.

Approximate nearest neighbors will be nearly balanced. We observe that RBF-FD functions as well on stencils of true nearest neighbors as it does on approximate nearest neighbors.

To demonstrate the savings in choice of stencil generation method, we provide Figure ??.

The impact of our neighbor query also extends influence on the structure of the RBF-FD DMs. To quantify the sparsity of a Differentiation Matrix we consider the ratio of non-zeros ( $N * n$ ) to total elements in the matrix ( $N^2$ ). For example, a problem of size  $N = 10,000$  with stencil size  $n = 31$  has a ratio of 0.0031 and is 99.69% empty.

Querying neighbors requires searching at least the immediate cell one layer of neighbors. by including one extra layer we ensure that small stencils near the border of the immediate cell can pick up neighbors in adjacent cells.

A prototype implementation of this method allowed for a variety of space filling curves to reorder the cells. The curves are created through integer dilation [? ]

The KDTree implementation used in this work is from

Original data showed our algorithm as wildly successful against a version

The Cuthill McKee algorithms can be equated to a breadth-first search. The algorithm queues nodes in order of degree at each level of the search and traverses the lowest degree

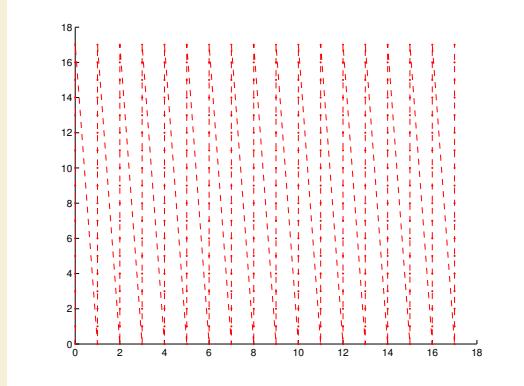


Figure 1.7: In order: a) node ordering test cases; b) original ordering of regular grid (raster); c) coarse grid overlay for hash functions ( $hnx = 6$ ); d) example stencil ( $n = 31$ ) spanning multiple Z's; e) spy of DM after orderings. Author's Note: [REGENERATE FIGURES WITH RANDOM/HALTON NODES](#)

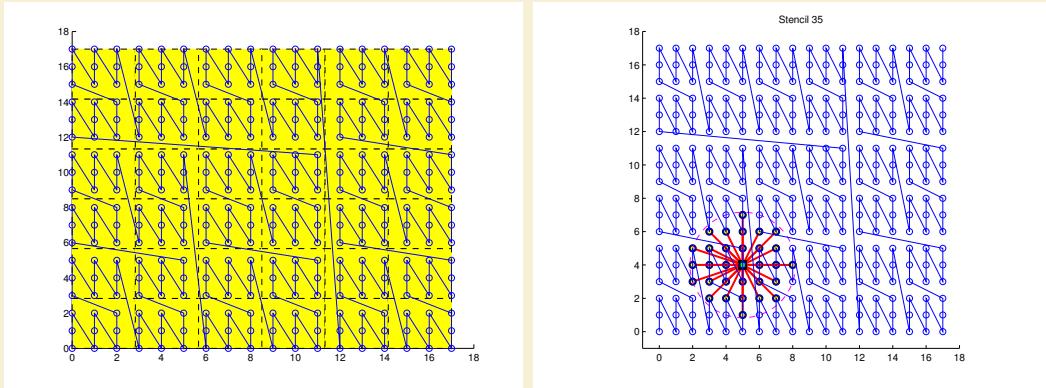


Figure 1.8: In order: a) node ordering test cases; b) original ordering of regular grid (raster); c) coarse grid overlay for hash functions ( $hnx = 6$ ); d) example stencil ( $n = 31$ ) spanning multiple Z's; e) spy of DM after orderings.

priority. The Reverse variant of Cuthill-McKee inverts the node order so that the lowest degree and top level node are at the end of the matrix rather than the beginning. Aside from ordering, the Reverse and Standard Cuthill McKee algorithms are identical processes. RCM is the more popular of the variants though, due to storage savings and reduced fill-in for some decompositions [17]. .

Obviously, the ideal case for bandwidth is when all rows contain the  $\frac{n}{2}$  nodes corresponding to solution value to either side of  $u_j$ . In 1-D this corresponds to every node containing the  $\frac{n}{2}$  nodes to the left and right of  $x_j$ . In 2-D this is only possible if the nodes in the domain are properly indexed such that stencils contain the proper set of neighbors—a stringent requirement that will

The early implementation of  $k$ -D tree had an  $O(n^2)$  growth in complexity. This algorithm was developed to alleviate that cost. It took a few hours to implement but has had

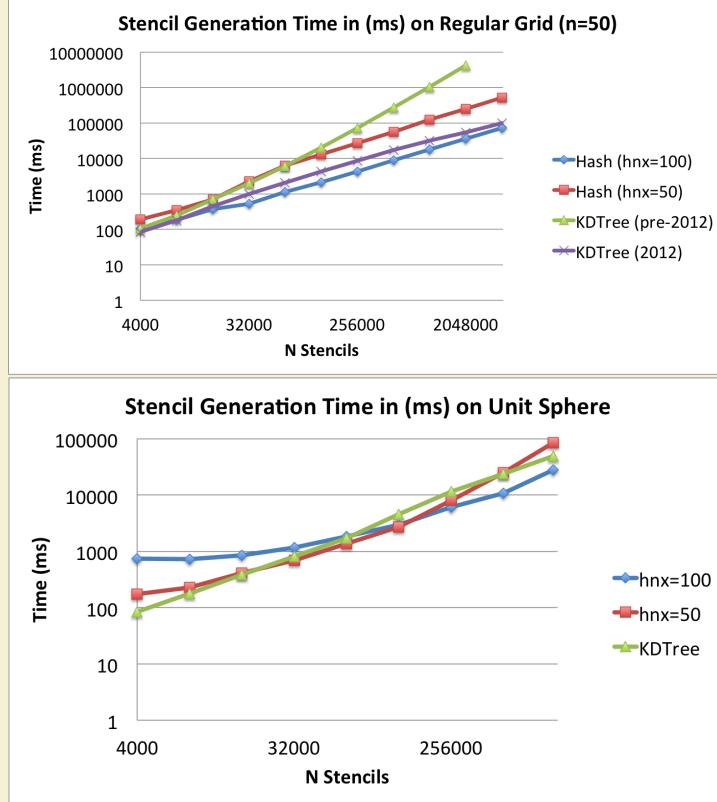


Figure 1.9: Querying the  $n = 50$  nearest neighbors on a regular grid up to  $N = 160^3$  demonstrates the significant gains achieved by our spatially binned neighbor query. While KDTree queries grow as  $O(N \log N)$

some surprising impacts. Note that the complexity of  $k$ -D tree was reduced to  $O(N \log^2 N)$  in 2012. On a regular grid (generated with raster/IJK ordering), the cost of  $k$ -D tree grows at the same rate as the hashing method.

At  $N = 32000$  the cost of hashing drops below  $k$ -D Tree due to the decreasing number of empty hash cells. Likewise, at  $N = 1000000$  and beyond, the gap between hashing and  $k$ -D Tree begins to close as cells contain more than one

Q: why does the curve drop for  $hnx = 100$ ? Q: the complexity of the algorithm? Q: the sphere I understand: its localizing the search to small patches on the sphere, and

For every  $N$  there is an optimal  $hnx$ . This is depicted for  $N = 500000$  CVT and  $hnx = 100$ .

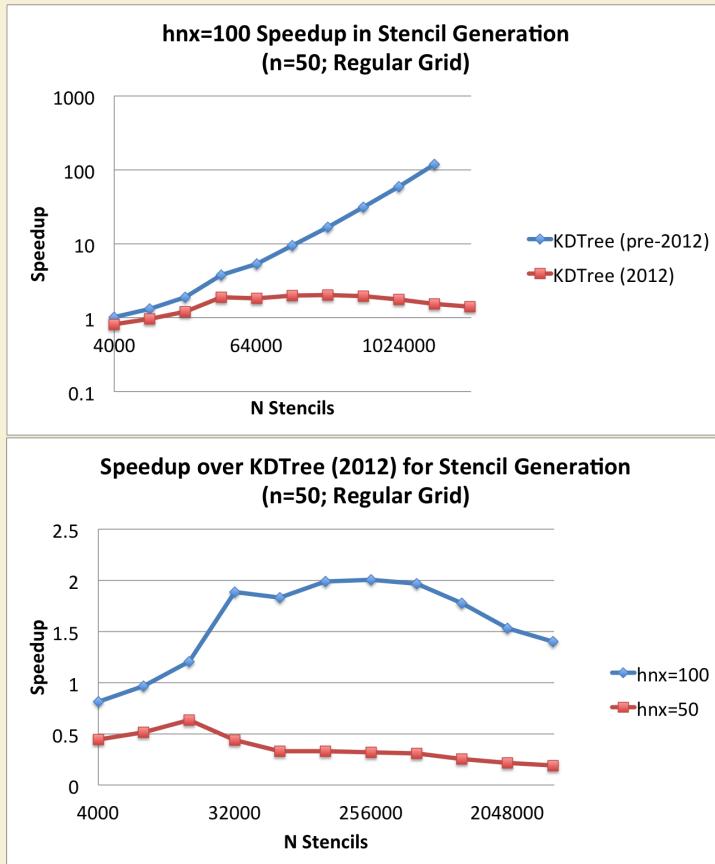


Figure 1.10: Querying the  $n = 50$  nearest neighbors on a regular grid up to  $N = 160^3$  demonstrates the significant gains achieved by our spatially binned neighbor query. While KDTree queries grow as  $O(N \log N)$

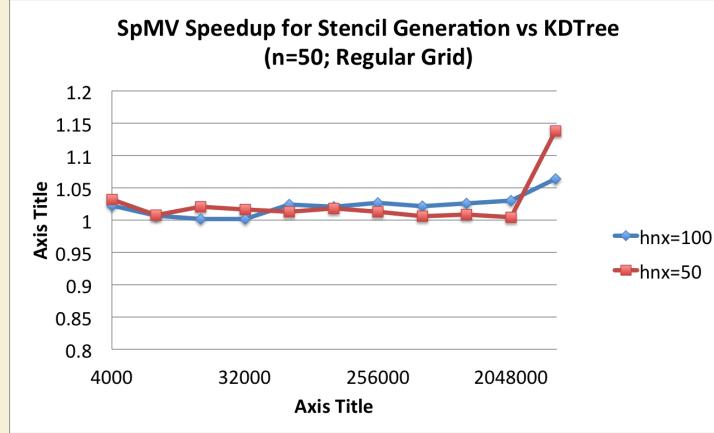


Figure 1.11: Querying the  $n = 50$  nearest neighbors on a regular grid up to  $N = 160^3$  demonstrates the significant gains achieved by our spatially binned neighbor query. While KDTree queries grow as  $O(N \log N)$

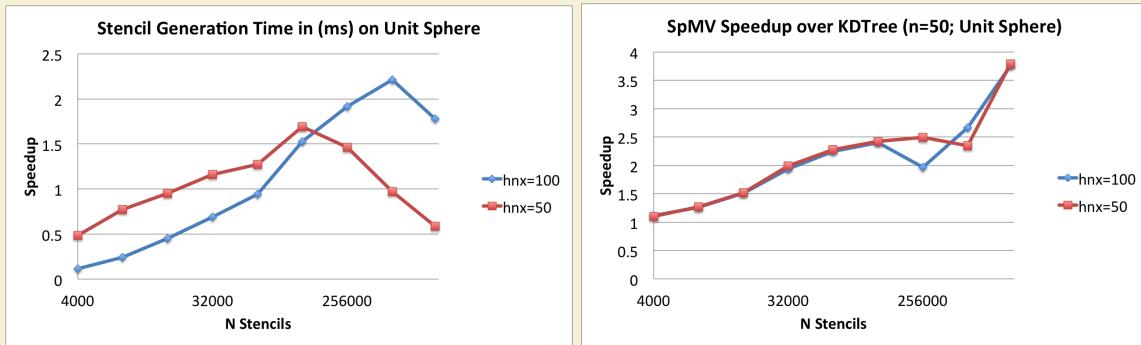


Figure 1.12: Generating stencils for increasing subsets of the  $N = 1e6$  CVT nodes mesh.

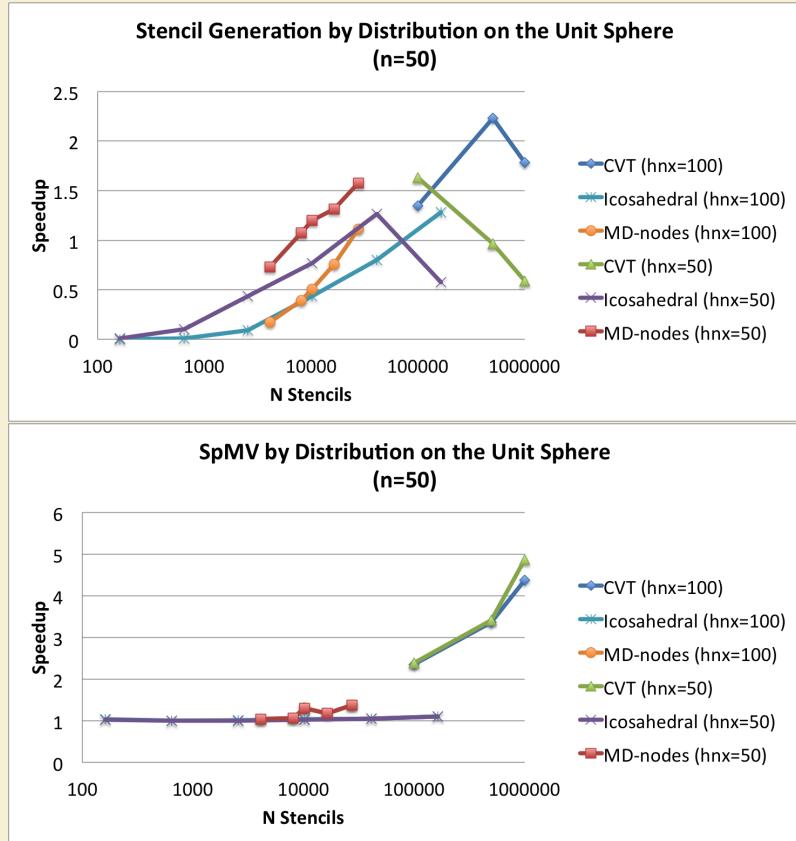


Figure 1.13: Based on the proper choice of overlay resolution, the hash stencil query can accelerate stencil generation, but the sophistication of the algorithm is low enough that negative impact is more likely. On the other hand, the impact on SpMV performance is always positive with the routine accelerated up to 4.9x faster.

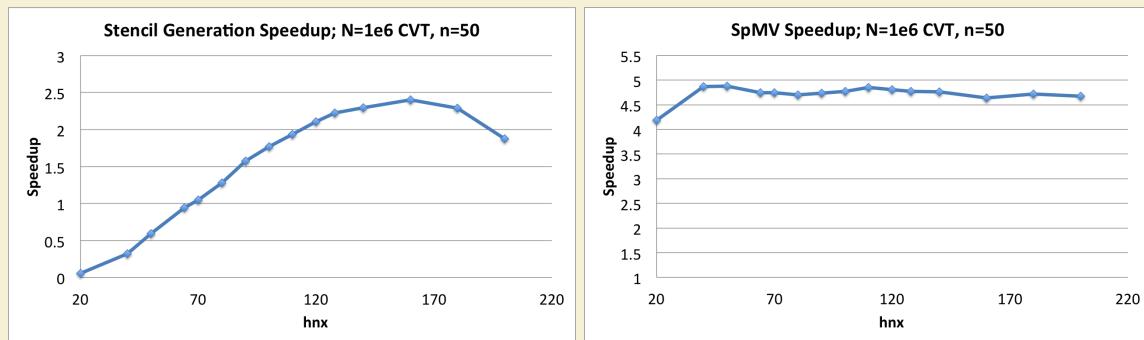


Figure 1.14: As the coarse grid resolution increases the hashing algorithm achieves both 2x faster than KDTree in stencil generation, with greater than 4x gain in SpMV performance (for free).

## 1.4 On Space Filling Curves and Other Orderings

### 1.4.1 Integer Dilation

One frequently hears that ordering via space filling curves like Morton Ordering and/or gray codes can benefit memory access patterns.

(Related? <http://publish.uwo.ca/~shaque4/presentationSONAD.pdf> <http://www.cs.duke.edu/~alvy/papers/sc98/> <http://www.cs.indiana.edu/~dswise/Arcee/Papers/medea06.pdf> <http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.118.7720&rep=rep1&type=pdf> <http://stackoverflow.com/questions/4260002/benefits-of-nearest-neighbor-search>

[21] mentions the impact of ordering on conditioning.

Algorithms like Reverse Cuthill McKee and Approximate Minimum Degree ordering allow general restructuring of matrices.

**Author's Note:** NEed to compare conditioning of LSH and other algorithms in Matlab

Q: what is an ideal ordering? Q: what is the best conditioning from ordering? Q: what is the relative cost of ordering?

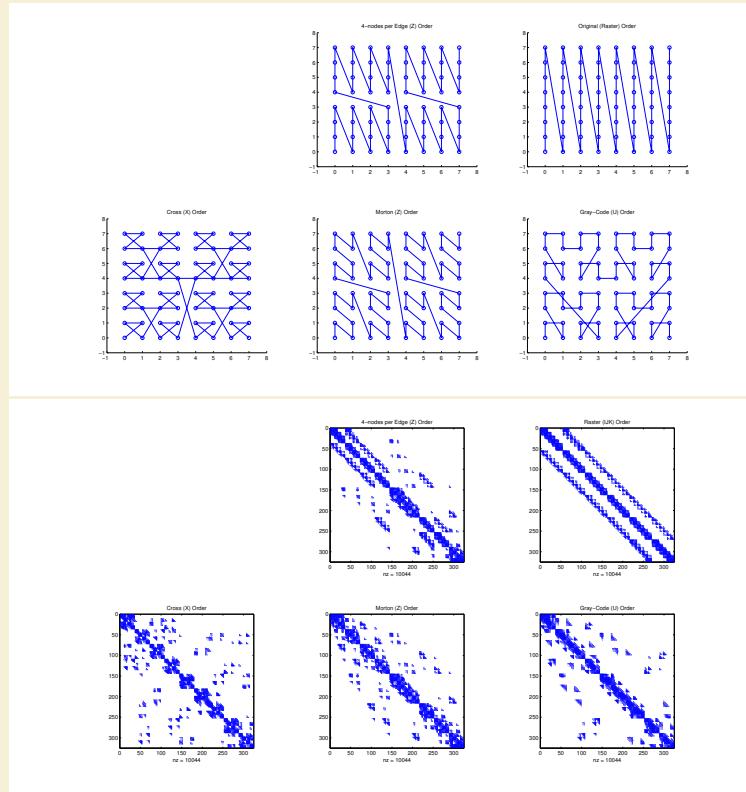


Figure 1.15: In order: a) node ordering test cases; b) original ordering of regular grid (raster); c) coarse grid overlay for hash functions ( $hnx = 6$ ); d) example stencil ( $n = 31$ ) spanning multiple Z's; e) spy of DM after orderings.

## 1.5 Conclusions on Stencil Generation

## BIBLIOGRAPHY

- [1] Kdtreesearcher class. MATLAB R2013a Documentation (<http://www.mathworks.com/help/stats/kdtreesearcherclass.html>), Aug 2013. 1, 11
- [2] Evan F. Bollig, Natasha Flyer, and Gordon Erlebacher. Solution to PDEs using radial basis function finite-differences (rbf-fd) on multiple GPUs. *Journal of Computational Physics*, 231(21):7133 – 7151, 2012. 14
- [3] Michael Connor and Piyush Kumar. Fast construction of k-nearest neighbor graphs for point clouds. *IEEE Transactions on Visualization and Computer Graphics*, 16:599–608, 2010. 14
- [4] Mark de Berg, Otfried Cheong, Marc van Kreveld, and Mark Overmars. *Computational Geometry: Algorithms and Applications*. Springer-Verlag, 3rd edition, 2008. 3, 4
- [5] Gregory E. Fasshauer. *Meshfree Approximation Methods with MATLAB*, volume 6 of *Interdisciplinary Mathematical Sciences*. World Scientific Publishing Co. Pte. Ltd., 5 Toh Tuck Link, Singapore 596224, 2007. 2, 4
- [6] Natasha Flyer and Erik Lehto. Rotational transport on a sphere: Local node refinement with radial basis functions. *Journal of Computational Physics*, 229(6):1954–1969, March 2010. 1, 7
- [7] Natasha Flyer, Erik Lehto, Sebastien Blaise, Grady B. Wright, and Amik St-Cyr. Rbf-generated finite differences for nonlinear transport on a sphere: shallow water simulations. *Submitted to Elsevier*, pages 1–29, 2011. 1, 2, 4
- [8] Natasha Flyer and Grady B. Wright. Transport schemes on a sphere using radial basis functions. *Journal of Computational Physics*, 226(1):1059 – 1084, 2007. 1
- [9] Natasha Flyer and Grady B. Wright. A Radial Basis Function Method for the Shallow Water Equations on a Sphere. In *Proc. R. Soc. A*, volume 465, pages 1949–1976, December 2009. 1
- [10] Bengt Fornberg and Erik Lehto. Stabilization of RBF-generated finite difference methods for convective PDEs. *Journal of Computational Physics*, 230(6):2270–2285, March 2011. 2, 4
- [11] Jerome H. Friedman, Jon Louis Bentley, and Raphael Ari Finkel. An algorithm for finding best matches in logarithmic expected time. *ACM Trans. Math. Softw.*, 3(3):209–226, September 1977. 4

- [12] Prashant Goswami, Philipp Schlegel, Barbara Solenthaler, and Renato Pajarola. Interactive sph simulation and rendering on the gpu. In *Proceedings of the 2010 ACM SIGGRAPH/Eurographics Symposium on Computer Animation*, SCA ’10, pages 55–64, Aire-la-Ville, Switzerland, Switzerland, 2010. Eurographics Association. 14
- [13] S. Green. Cuda particles. NVidia Whitepaper, 2010. 6, 7
- [14] N. A. Gumerov, R. Duraiswami, and E. A. Borovikov. Data structures, optimal choice of parameters, and complexity results for generalized multilevel fast multipole methods in  $d$  dimensions. Technical Report UMIACS-TR-2003-28, University of Maryland (College Park, Md.), Apr 2003. 2
- [15] I. Johnson. Real-time particle systems in the blender game engine. Master’s thesis, Florida State University, November 2011. 7
- [16] Øystein E. Krog. GPU-based Real-Time Snow Avalanche Simulations. Master’s thesis, Norwegian University of Science and Technology, June 2010. 2, 6, 7, 8, 9, 10
- [17] Wai-Hung Liu and Andrew H. Sherman. Comparative analysis of the cuthill-mckee and the reverse cuthill-mckee ordering algorithms for sparse matrices. *SIAM Journal on Numerical Analysis*, 13(2):pp. 198–213, Apr 1976. 15
- [18] John Mellor-crummey, David Whalley, and Ken Kennedy. Improving memory hierarchy performance for irregular applications using data and computation reorderings. In *International Journal of Parallel Programming*, pages 425–433, 2001. 12
- [19] Jia Pan and Dinesh Manocha. Fast GPU-based Locality Sensitive Hashing for K-Nearest Neighbor Computation. *Proceedings of the 19th ACM SIGSPATIAL GIS ’11*, 2011. 14
- [20] Timothy J. Purcell, Craig Donner, Mike Cammarano, Henrik Wann Jensen, and Pat Hanrahan. Photon mapping on programmable graphics hardware. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Conference on Graphics Hardware*, pages 41–50. Eurographics Association, 2003. 9
- [21] Yousef Saad. *Iterative Methods for Sparse Linear Systems*. Society for Industrial Mathematics, second edition, 2003. 20
- [22] Hanan Samet. *Foundations of Multidimensional and Metric Data Structures (The Morgan Kaufmann Series in Computer Graphics and Geometric Modeling)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2005. 2, 3, 4, 6, 9, 10, 12
- [23] Steven Skiena. *The Algorithm Design Manual (2. ed.)*. Springer, 2008. 2, 3
- [24] Andrea Tagliasacchi. kd-tree for matlab. <http://www.mathworks.com/matlabcentral/fileexchange/21512-kd-tree-for-matlab>, Sep 2010. 1, 2, 10, 11, 13
- [25] Andrea Tagliasacchi. kd-tree matlab. <https://code.google.com/p/kdtree-matlab>, Jun 2012. 4, 10, 11

- [26] Holger Wendland. Fast evaluation of radial basis functions: Methods based on partition of unity. In *Approximation Theory X: Wavelets, Splines, and Applications*, pages 473–483. Vanderbilt University Press, 2002. [2](#), [6](#), [9](#)
- [27] Holger Wendland. *Scattered Data Approximation*. Cambridge Monographs on Applied and Computational Mathematics. Cambridge University Press, 2005. [1](#), [2](#), [6](#), [9](#), [10](#)
- [28] Lexing Ying. A kernel independent fast multipole algorithm for radial basis functions. *Journal of Computational Physics*, 213(2):451 – 457, 2006. [2](#), [3](#)