Bachelor thesis

Tjørn Lynghus

Data-parallel Implementation of Randomized Approximate Nearest Neighbours

Supervisor: Cosmin Eugen Oancea

Datalogisk Institut. June 11, 2023

Contents

1 Abstract

Finding the k nearest neighbours is a problem of important applicability in many different areas of computer science. Solving the k nearest neighbours problem is however computationally expensive. This has increased the interest in approximate nearest neighbour algorithms that, instead of finding the exact nearest neighbours, tries to find good approximations and in turn lower the runtime. One such algorithm is called Randomized Approximate Nearest Neighbours (RANN) which makes use of fast random rotations and k-d trees to lower runtime. In recent years parallel implementations of machine learning algorithms using graphics processing units have been shown to decrease the runtime significantly when compared to the sequential implementations.
In this thesis a parallel implementation of RANN[1] is proposed. The implementation is shown to provide a significant speed up when compared to a parallel implementation of an exact k nearest neighbours algorithm.

2 Introduction

The nearest neighbour search problem is of significance in computer science. A variation of nearest neighbour search is the k nearest neighbours (knn) problem which can be described in the following way: Given a point set $\mathbf{A}$ in $\mathbb{R}^d$ and a query point $\mathbf{x}$. Find the k points in $\mathbf{A}$ that are closest to $\mathbf{x}$. The brute force solution for this problem has computational complexity $O(nm)$ (where n is the amount of queries and m is the amount of points in $\mathbf{A}$).
This has many applications across computer science including vector compression, computational statistics, computer vision and many more (see e.g. reference [1] for more applications). However as time has gone on the datasets for these applications have become larger and larger and finding the nearest neighbours in these datasets is in many instances computationally too expensive.
One way to decrease the computational complexity is by using k-d trees [13]. This can yield good results on datasets with low dimensionality. This is because that it can be expected that only $\log m$ leaves needs to be searched for each query for a data set with low dimensionality. This results in an asymptotic time of $O(n \log m)$. However as the dimensionality increases, more and more leaves need to be searched due to the curse of dimensionality [6]. For higher dimensionality all the leaves need to be searched. This results a computational complexity of $O(nm)$ which is the same complexity the brute force approach achieves.
This has spurred on approximate nearest neighbour algorithms that find a good approximation of the nearest neighbours at a lower computational cost.
One approach is that of locality sensitive hashing (LSH) [4]. LSH is an algorithm that tries to hash similar objects or points into the same hash value. LSH thus tries to maximize hash collisions.
Approximate nearest neighbour solutions that are domain specific also exist. These include propagation-assisted k-d trees [7] and coherency sensitive hashing [11] which are used in computer vision. These algorithms were inspired by PatchMatch [3] which uses propagation. The propagation exploits the coherency of images such that good matches can be propagated to their surroundings.
All of these approaches tries to exploit the locality of the given dataset.
This thesis will focus on a context independent knn solution which uses a combination of fast random rotations and k-d trees, namely randomized approximate nearest neighbours (RANN), which was proposed in [10]. Here the rotations make it possible to transform the dataset in such a way

[1]The gihub repository https://github.com/Ritobusk/RANN

that you can create the kd-tree from the transformed dataset that reorders it in a desirable way. When searching the kd-tree you first find each query's natural leaf. After that you use the path to the natural leaves to find $\log m$ (where m is the number of points in the reference set) other leaves for each query. This is looped **T** times to improve the accuracy. Which means **T** kd-trees are build and searched. The computational complexity for RANN is $n \cdot \log m \cdot T$.
All of these approaches improve the computational cost of exact knn methods. However as datasets continue to increase further improvements are needed. The above mentioned algorithms' implementations are all originally designed to be run on a CPU. However one can exploit modern highly-parallel hardware such as graphics processing units (GPUs) to achieve improved runtime. By implementing solutions in a highly parallel manner efficient use of the GPU can be achieved. This has been seen to improve the runtime significantly in comparison with implementations designed for the CPU. [5]
A tool to create such parallel implementations is Futhark [8] [9]. Futhark has already been used to parallelize the previously mentioned domain specific solutions which improved the runtime significantly. Namely propagation-assisted k-d trees [12] and coherency sensitive hashing [2].
For this thesis a parallel implementation of RANN was implemented with the help of Futhark. The implementation improved the runtime when compared to a parallel implementation of a brute force exact knn solution. The parallel implementation of RANN with T = 1 was between 20 and 140 times faster than the parallel brute force algorithm. The parallel implementation of RANN can achieve almost 100% accuracy when increasing T. (This is explored in depth in section 7.7)
The thesis includes the following: A presentation of related algorithms in section 3. Preliminaries for the programming language Futhark in section 4. A description of RANN in section 5. A description of the parallel implementation of RANN in section 6. Experimentation and evaluation of the parallel implementation of RANN is presented in section 7. Section 8 mentions some improvements that can be made to the implementation and evaluation.

3 Related work

This section contains brief explanations of 2 kd-tree based solutions for the knn problem.

3.1 Exact Nearest Neighbours via kd-Trees

The nearest neighbours via kd-trees algorithm [13] is a way of solving the knn problem by creating a kd-tree and then traversing it. This solution computes the true nearest neighbours.
A k-d tree is a balanced tree with the root being a set of points $\mathbf{P} = \{x_1, x_2, ..., x_m\}$ where each point has d dimensions. For a given node $\mathbf{v}_j$ with a point set $\mathbf{P}_{\mathbf{v_j}}$ in the tree at level **i** you split $\mathbf{P}_{\mathbf{v_j}}$ at the median in dimension **i**. Thus obtaining two new almost equally sized subsets. These nodes will be part of level **i** + 1. This process of splitting continues recursively on the new nodes until the tree has the desired height. A k-d tree is thus a datastructure that reorders the dataset **P**.
When the kd-tree is constructed it needs to be searched for each query to find the query's knns. Initially the natural leaf is found and the points within are used to update the knns. The natural leaf is found by traversing the kd-tree and comparing the medians to the query's values. Once the natural leaf is found the knns are updated by finding the true nearest neighbours within the leaf to the query. Now neighbouring leaves are checked to see if they can contain better nearest neighbour candidates. This is done by comparing the medians of a neighbouring leaf to the current knn candidate that has the largest distance to the query. If this comparison shows that there could be better candidates inside, it is visited and searched. Otherwise the leaf is ignored and others are checked. In this way the reference set is pruned.
This algorithm performs very well on datasets with low dimensionality which can reduce the complexity to $O(n \log(m))$, where n is the size of the query set and m the size of the reference set. However as the dimensionality increases the computational complexity is closer to $O(mn)$, which means that most times every leaf is searched.

3.2 Approximate Nearest-Neighbour Fields via Massively-Parallel Propagation-Assisted K-D Trees

Propagation-Assisted K-D Trees is an algorithm specialized in finding similarities in 2 images via approximate nearest-neighbour fields. The goal is to find for every patch in an image A a patch in image B which approximates it. It does so with the combination of k-d trees and a coherency property found in images which is used to propagate good nearest-neighbour candidates.
The coherency property is that of locality. The intuition for this is that patches whose nearest neighbour was previously computed, have good candidates for the nearest neighbours of close by patches who are yet to be computed. To ensure that what is propagated has a good basis, you first compute the exact nearest neighbours for the first row of patches with a k-d tree traversal. Then you start propagating the information to the next row.
The algorithm works in the following way: First you reduce the dimensionality with principal component analysis. This is done to reduce the computational complexity. Next you build the k-d tree from the patch set with reduced dimensionality. Then you compute the exact nearest neighbours of the first row of patches by traversing the k-d tree. The result of this is then propagated to the next row to find good candidates which is then propagated to the row after that and so on. When you have computed the candidates for all patches you select for each patch the k best nearest neighbour candidates by taking into account the original patch set.

Propagation-assisted k-d trees were first proposed by He and Sun [7]. Later on a parallel implementation was created by Oancea et al. [12]

4 Preliminaries for Futhark

Since the implementation of RANN makes use of Futhark, this section will present the functions from Futhark that the implementation is build on. Futhark is statically typed and purely-functional. Furthermore it is build to produce efficient parallel code when compiled. The compiled code can be either multi-threaded CPU code or GPU code via either the CUDA or OpenCL backends.

4.1 Functions

4.1.1 Map

Map has type signature: `map : (f: a -> b) -> (as: [n]a) -> [n]b`. It takes as arguments a function *f* with type `a -> b` and an array *as* with *n* elements of type *a*. Map takes the function and applies it to every element of the array thus producing a new array of similar length with type `[n]b`. An example of map: `map (+1) [3, 5, 1] = [4, 6, 2]`

4.1.2 Iota

Iota has type signature: `iota : (n : int) -> [n]int`. It takes as argument an integer *n* denoting the length. It then produces an array of consecutive integers starting at 0 with length *n*. An example is: `iota 4 = [0, 1, 2, 3]`

4.1.3 Reduce

Reduce has type signature: `reduce : (op: a -> a -> a) -> (ne: a) -> (as: [n]a) -> a`.
It takes as arguments an operator, op, that takes two arguments of similar type and returns the result of applying the operator to the arguments. An example with where op is (+): `op 2 3 = 5`.
Reduce then takes a neutral element of type *a* and finally an array `as: [n]a`.
Reduce applies the operator in the following way:`(as[n-1] op (... (as[2] op (as[1] op (as[0] op ne)))))`.
An example is: `reduce (+) 0 [3, 6, 2] = (2 + (6 +(3 + 0))) = 11`.
An important requirement for reduce in Futhark is that the operator is associative
(i.e.:`(a op (b op c) = (a op b) op c)`). Since it would otherwise create unspecified behavior when you compute reduce in parallel.

4.1.4 Scan

Scan has type signature: `scan : (op: a -> a -> a) -> (ne: a) -> (as: [n]a) -> [n]a`.
Scan takes arguments similar to reduce however it returns an array. Scan also works in a similar way to reduce. But instead of just returning the application of op in the same way as reduce, scan saves each intermediate application. This is best illustrated with an example:
`scan (+) 0 [3, 6, 2] = [(3 + 0), (6 + (3 + 0)), (2 + (6 + (3 + 0)))] = [3, 9, 11]`
As with reduce, op needs to be associative.

4.1.5 Scatter

Scatter has type signature: `scatter : (dest: [k]t) -> (is: [n]i64) -> (vs: [n]t) -> [k]t`
The idea of scatter is to alter *n* values in an array *dest*. As arguments scatter takes an array *dest* which is the array that is altered. An array *is* which is an array of the zero-indexed indices of *dest*

that will be altered. And *vs* which is the values that are written to the *dest* array. Here is an example: `scatter [1.0, 2.0, 3.0, 4.0] [3, 0, -1] [5.4, 6.5, 8.0] = [6.5, 2.0, 3.0, 5.4]` In the example index 3 and 0 of *dest* have been altered. If an index which is out of bounds is encountered the index and corresponding value is ignored. It is important that scatter does not try to write different values to the same indices since it will result in unspecified behavior.

4.2 Time complexity

To measure the asymptotical time of using one of the above functions their work and span is provided in table 4.2. Work describes the total amount of operations that is performed by the function. The span describes the total amount of work a single thread will potentially do. Since

Function	**Work**	**Span**
map	O($n \cdot W(f)$)	O(S(f))
iota	O(n)	O(1)
reduce	O($n \cdot W(op)$)	O($\log(n) \cdot W(op)$)
scan	O($n \cdot W(op)$)	O($\log(n) \cdot W(op)$)
scatter	O(n)	O(1)

map takes a function as an argument and reduce and scan take an operator it is more difficult to determine their asymptotic bounds. W(f) and S(f) in table 4.2 is therefore the work of applying *f* once and the span of applying *f* once respectively. W(op) is the work of applying the operator *op* once.

5 Randomized approximate nearest neighbours

The randomized approximate nearest neighbours algorithm (RANN, created by P. W. Jones, A. Osipov, and V. Rokhlin and introduced by them in [10]) makes use of random orthogonal linear transformations and kd-trees to find an approximation of the k nearest neighbours of each point $\mathbf{x}_i$ with dimensionality d inside a set of points **A** also with dimensionality d. Where $\mathbf{x}_i$ belongs to the set of query points

$$\mathbf{B} = \{x_0, x_1, ..., x_{n-1}\} \subseteq \mathbb{R}^d$$

As opposed to the nearest neighbours via kd-trees algorithm [13] that searched all leaves that could contain nearest neighbours, this algorithm only searches a fraction of the leaves. The set of leaves that is searched for each query is called $\mathbf{V}_i$.
The central idea of the algorithm is the following observation: If it is possible to find a subset of **A**; $\mathbf{V}_i$, such that it is more likely for the nearest neighbours of point $\mathbf{x}_i$ to be in $\mathbf{V}_i$, then searching $\mathbf{V}_i$ for the nearest neighbours of $\mathbf{x}_i$ should yield good nearest neighbour candidates.
Since there might be many true nearest neighbours not included in $\mathbf{V}_i$ it makes sense to reselect $\mathbf{V}_i$ such that it contains new points that are also likely to be good candidates.

Here is an overview of the algorithm:

1. Shift all points by subtracting the center of mass of the collection of **A**
2. Perform a random orthogonal linear transformation Θ on all points obtaining for each point $\mathbf{x}_i \in B$ and each point $\mathbf{y}_i \in A$: $\Theta(\mathbf{x}_i)$ and $\Theta(\mathbf{y}_i)$ respectively.
3. Build a kd-tree from the transformed points in **A**
4. Search the kd-tree to obtain a $\mathbf{V}_i$ for each $\Theta(\mathbf{x}_i)$
5. Update the k nearest neighbours of each $\mathbf{x}_i$ with the best candidates found in $\mathbf{V}_i$
6. Perform steps 2 to 5 **T** times
7. Do a depth one search, dubbed 'supercharging', on the candidates of the k nearest neighbours of each point $\mathbf{x}_i$

Each of these steps are explained below.

5.1 Shifting the points

Subtracting the center of mass of the collection of **A** from each point is fairly simple. The center of mass of a collection is defined by summing up all points of the collection and then dividing by the number of points in the collection.
The summation can be done with vector addition since the d dimensional points can be treated as vectors.
Then the sum of points can be divided by the number of points inside the collection and you have the center of mass of the collection.

$$\frac{\sum_{i=0}^{n-1} x_i}{n}$$

To shift the points you subtract the center of mass from each point with vector subtraction.

5.2 Pseudorandom orthogonal linear transformation

The pseudorandom orthogonal transformation $\mathbf{\Theta}$ consists of 3 operators. This transformation needs to be performed on all points such that they are all transformed in a similar way.
The first operator is **P** which permutes the coordinates of a vector. The second operator is **Q** which rotates a vector. And the third operator, **F**, is a fast discrete Fourier transform.

P operates by generating a permutation of the numbers $\{0, 1, ..., d-1\}$, π, and then permuting a d dimensional vector, **v**, with π such that

$$P(\mathbf{v}[i]) = \mathbf{v}[\pi[i]], \qquad for\ i = 0, 1, \dots, d-1 \tag{1}$$

Q operates by first generating $d-1$ independent pseudo random numbers, θ, uniformly distributed in $(0, 2\pi)$. Then $\mathbf{Q}_k$ rotates the coordinates k and $k+1$ by the angle $\theta[k]$ of some vector **v** with dimensionality d:

$$\mathbf{Q}_k(v)\begin{pmatrix} k \\ k+1 \end{pmatrix} = \begin{pmatrix} \cos(\theta(k)) & \sin(\theta(k)) \\ -\sin(\theta(k)) & \cos(\theta(k)) \end{pmatrix} \cdot \begin{pmatrix} v[k] \\ v[k+1] \end{pmatrix} \tag{2}$$

Q is then calculated with the compositions:

$$\mathbf{Q} = \mathbf{Q}_{d-2} \circ \mathbf{Q}_{d-1} \circ ... \circ \mathbf{Q}_0 \tag{3}$$

F has two components. A fast Fourier transform matrix $d_2 \times d_2$ T $\mathbb{C}^{d_2} \to \mathbb{C}^{d_2}$, where $d_2 = \frac{d}{2}$ and a one-to-one linear operator Z $\mathbb{R}^d \to \mathbb{C}^{d_2}$. T is defined by

$$T[k, l] = \frac{1}{\sqrt{d_2}} \cdot \exp[-\frac{2\pi i(k-1)(l-1)}{d_2}], \tag{4}$$

where $k, l = 1, ..., d_2$ and $i = \sqrt{-1}$. Z operates on a vector in the following way

$$Z(v) = \begin{pmatrix} v[0] + i \cdot v[1] \\ v[2] + i \cdot v[3] \\ \vdots \\ v[2d_2 - 2] + i \cdot v[2d_2 - 1] \end{pmatrix} \tag{5}$$

F is finally computed with

$$\mathbf{F} = Z^{-1} \circ T \circ Z(v), \tag{6}$$

when d is even. If d is odd you only apply **F** on the first $d-1$ coordinates of v and leave the last coordinate as is.

The pseudorandom orthogonal transformation $\mathbf{\Theta}$ is finally defined as

$$\mathbf{\Theta}(v) = (\prod_{j=0}^{M_1-1} Q_j \circ P_j) \circ F \circ (\prod_{j=M_1}^{M_1+M_2-1} Q_j \circ P_j) \circ v \tag{7}$$

$M_1 + M_2$ are chosen to be $\approx \log d$. Since Q_j and P_j are applied multiple times this means that in total $M_1 + M_2$ permutations for **P** and $(M_1 + M_2) \cdot (d-1)$ random numbers for **Q** are needed to compute $\mathbf{\Theta}$.

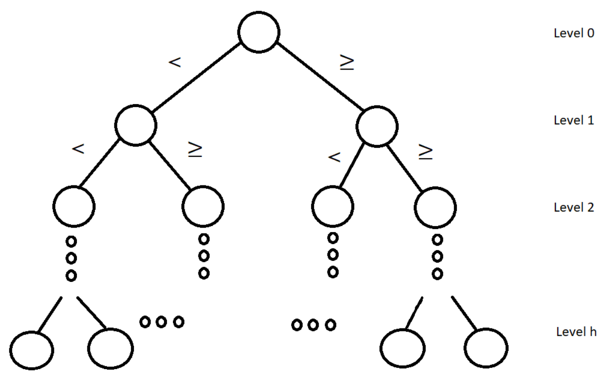

Figure 1: kd-tree

5.3 Building the kd-tree

The kd-tree is constructed by first considering the root node which contains all the new transformed points in $\mathbf{A}$. Then the median value of the first coordinate of the points in the node is found. Now two new nodes are created. One containing the points whose first coordinate value is $<$ than the median and one containing the ones that are $\geq$ than the median. In this way the next level of the tree (level 1) is been computed, consisting of these 2 new nodes.
To compute level 2 you perform the same procedure on each node in level 1, creating the 4 nodes level 2 will consist of. The coordinate that is chosen to be compared is however changed between the levels. It is computed with

$$coordinate = level \mod d,$$

to increase the spread.
This continues until the tree has reached a specified height resulting in a binary tree of height h. The kd-tree is thus constructed iteratively as opposed to the recursively constructed kd-tree of the nearest neighbors via kd-trees algorithm (section 3.1). A visual representation is seen in figure 1. The following pseudocode describes the process of building the tree:

```
Build_kdtree (reference_set A, height h) =
    nodes_in_level[0] = A
    for (i = 0; i < h; i++)
        coordinate = i mod d
        nodes_in_level[i+1] =
            foreach node in nodes_in_level[i]
                median = find median of coordinate in node
                new_nodes = partition the node according to median
```

5.4 Searching the kd-tree

For each new transformed point $\mathbf{x}_i$ in $\mathbf{B}$ a set $\mathbf{V}_i$ is needed to be found. $\mathbf{V}_i$ consists of a union of two sets. The first is the points in the leaf of the kd-tree that naturally belongs to a point $\mathbf{x}_i$.
To find this leaf you compare the coordinate value of $\mathbf{x}_i$ corresponding to the current level with the

median of the current node. If the value is < than the median you travel to the left, down the path where the points of A also are < than the median at the coordinate of the level. Otherwise you travel to the right. This continues until you have a path that leads the natural leaf at level h. This path can be depicted with an h long word of bits, where a zero denotes that the path travels to the left and a 1 denotes travels to the right. E.g $[1, 0, 0]$ is the path of a tree of height 3 the travels to the right and then twice left to reach the leaf.

The second set $\mathbf{V}_i$ consists of is the points inside the leaves that have the same path as the natural leaf from the root but differ at one level. E.g. the path $[1, 0, 1]$ is a path to such a leaf if the example path from above is the path to the natural leaf.
$\mathbf{V}_i$ will thus consist of $h + 1$ points. Pseudocode for finding each $\mathbf{V}_i$ is shown below:

```
Compute_V_is (k-d_tree kd_t, queries qs, height h) =
  foreach x_i in qs
    (natural_leaf, natrual_path_to_leaf) =
                find (natural_leaf, natural_path_to_leaf) of x_i in kd_t
    leaves_with_changed_path =
      V_i_tmp[h]
      for (i = 0; i < h; i++)
        new_path = change bit i of natrual_path_to_leaf
        V_i_tmp[i] = extract leaf from kd_t with path new_path
    V_i = natural_leaf ++ leaves_with_changed_path
```

5.5 Searching the subsets V_i (the leaves)

Searching each $\mathbf{V}_i$ to find nearest neighbour candidates for the corresponding transformed point $\mathbf{x}_i$ is done by finding the true knns of $\mathbf{x}_i$ inside of $\mathbf{V}_i$. This is determined with a brute force method that finds the k points in $\mathbf{V}_i$ with the lowest L2 distance. This process can be described with the following pseudocode:

```
Search_leaves (queries B, all_V_is V) =
    new_knn_candidates =
        foreach query in queries and V_i in V
            find k closest points to query in V_i
```

5.6 The loop

This step simply repeats 2 to 5 mentioned above to find new nearest neighbour candidates. When the loop is done the best candidates from each iteration of the loop are selected as k nearest neighbour candidates.

5.7 Supercharging

The depth one search (called 'supercharging' in the article) is done to improve the accuracy of the candidates for knns of each point $\mathbf{x}_i$. The idea is that good candidates that were not found by the other steps might be found among the k candidates of $\mathbf{x}_i$'s candidates. The candidates are in $\mathbf{A}$. So this means that supercharging necessitates that one knows the knns of each point $\mathbf{y}_i \in \mathbf{A}$ among $\mathbf{A}$

or at least good candidates for them.
For each $\mathbf{x}_i$ you search the neighbours of $\mathbf{x}_i$'s candidate nearest neighbours found in step 6 for better candidates. Each point of course has k neighbours. This means that each $\mathbf{x}_i$ will search through k^2 candidates. Supercharging can be described with the following pseudocode:

```
Supercharging (queries B, reference_set A, queries_knn_candidates q_knns,
                                           reference_knn_candidates r_knns) =
    for (i = 0; i < length of B; i++)
        new_candidates =
            foreach nn in q_knns[i]
                r_knns[nn]
        find exact knns in new_candidates
```

The fact that supercharging necessitates good nearest neighbour candidates for the points in **A** amongst the points in **A** means the use of supercharging dictates how the algorithm is structured. It therefore also makes most sense to use supercharging when **A** is equal to **B** since you then do not have to find knns for the points in **A** among **A**. Here are the ways the algorithm can be structured:

- If supercharging is not used, a similar structure as the one outlined in the overview of the algorithm from the start of this section can be used. Notably without step 7: The supercharging.
- If you use supercharging and already know the points in **A**'s nearest neighbours within **A**, you can follow the outlined algorithm from the start of the section.
- If you use supercharging and **A** is equal to **B** then you can follow the outlined algorithm.
- If you use supercharging and do not know the points in **A**'s nearest neighbours within **A**, you need to find either the true or approximate k nearest neighbours for these points. If you use RANN to find approximate knns you can modify the searching of the tree (step 4) and the finding of the nearest neighbours within the result of step 4 (step 5). Step 4 now needs to also search the tree for a $\mathbf{U}_i$ for each $\Theta(\mathbf{y}_i)$. Where $\mathbf{U}_i$ is a subset of **A** with good candidates for $\Theta(\mathbf{y}_i)$ found in a similar fashion as $\mathbf{V}_i$ is for $\Theta(\mathbf{x}_i)$. Step 5 needs to be modified so you also search for each $\Theta(\mathbf{y}_i)$ its $\mathbf{U}_i$ for true nearest neighbours.

6 Parallel implementation

This section will explain how each step of RANN has been implemented to be data parallel in Futhark. The explanations will often be accompanied by some code snippets. These snippets are most often presented and then explained in more detail with text below the snippet.

6.1 Shifting the points

```
1  def shiftPoints [n] [m] [d]
2           (A : [m][d]f32) (B : [n][d]f32) : ([m][d]f32, [n][d]f32)  =
3     let com_acc       = replicate (d) 0.0f32
4     let com_sum       = reduce (\acc p -> map2 (+) acc p) com_acc A
5     let com           = map (\i -> i / m) com_sum
6     let shifted_A = map (\p -> map2 (\pv mv -> pv - mv) p com) A
7     let shifted_B = map (\p -> map2 (\pv mv -> pv - mv) p com) B
8     in (shifted_A, shifted_B)
```

The goal is to subtract the center of mass of the collection of **A** from each point. First the center of mass needs to found which can be done with a reduce over **A** where the operator is (+). This results in a vector sum of all the points in **A**. To get the center of mass you divide the result of the reduce with the number of points in **A**. This can be seen in line 3-5 in the code above.
To shift the points in **A** and **B** you then subtract the mass from the points as can be seen in line 6 and 7.

6.2 Pseudorandom orthogonal linear transformation

This step contains 4 functions that needs to be implemented. Namely **P**, **Q**, **F** and **Θ**. Where **Θ** is the pseudorandom orthogonal linear transformation applied to one point. In the code snippet below is an overview of how **Θ**, defined by equation 7, is calculated. This can be seen as an overview of what happens in step 2:

```
let Theta [d] (point :  [d]f32) (permutations : [][]i64)
              (random_numbers : [][]f32) (m1 : i64) (m2 : i64) =
  let m2_PQ =
      loop acc = point for i < m2 do
        let pointP = calculate_Pj acc permutations[i]
        in  calculate_Qj pointP random_numbers[i]
  let d2 = d / 2
  let Fd_m2_PQ = Fd m2_PQ d2
  in loop acc =
        Fd_m2_PQ for i < m1 do
          let pointP = calculate_Pj acc permutations[i + m2]
          in  calculate_Qj pointP random_numbers[i]
```

There are 2 loops. In the first **P** and **Q** is applied to a point *m2* times. When the loop is finished the fast Fourier transform **F** is applied to the result of the first loop. Then the second loop applies

P and **Q** to the result of line 8. **Θ** is applied to each point in the dataset with a map. How **P**, **Q** and **F** are computed is described in greater detail below.

To get random numbers and use complex numbers 3 modules that have not been created for this thesis will be used: Random[2], shuffle[3], and complex[4]. The 'random' module will allow for generating random numbers. The 'shuffle' module can create permutations for **P**. The 'complex' module will allow for the use of complex numbers.

6.2.1 P

P permutes a d dimensional vector. To accomplish this a random number generator engine from the 'random' module is used. And a shuffler from the 'shuffle' module is used to create permutations of $[0, 1, ..., d-1]$. The permutations are created as seen in the code snippet:

```
1      let rng_P = rng_engine.rng_from_seed [t]
2      let rngs = rng_engine.split_rng (M1+M2) rng_P
3      let permutations = map (\r -> shuffle.shuffle r [0...d-1]) rngs
```

Line one creates an `rng` that can generate random numbers. Here `rng_engine.split_rng` is used to allow for parallelism since it creates many `rng`s with different seeds that can be used in parallel. The permutations are calculated in line 3 with the shuffle module which uses the `rng`s to create random permutations of $[0, 1, ..., d-1]$. These are applied as described by equation 1 to a point in the following way:

```
let calculate_Pj [d] (p: [d]f32) (permutation : [d]i64) : *[d]f32 =
    map (\i -> p[i]) permutation
```

6.2.2 Q

Q rotates a d dimensional vector. To generate the $(d-1) \cdot (M_1 + M_2)$ random numbers used for the rotation as defined by equation 2, a similar approach of splitting the `rng`s as in **P** is used. Once the random numbers have been generated they need to be normalized to lie in the range $(0, 2\pi)$. This is done by first normalizing to 1 and the multiplying the result with 2π. Bellow is the implementation of **Q** applied on a single point.

```
let calculate_Qj [d] [d_minus1] (point: *[d]f32)
                     (rand_numbers: [d_minus1]f32) : *[d]f32 =
    loop acc = point for i < d_minus1  do
        let tmp = acc[i]
        let acc[i]   = ((f32.cos rand_numbers[i]) * tmp  +
                        (f32.sin rand_numbers[i]) * acc[i+1])
        let acc[i+1] = ((-f32.sin rand_numbers[i]) * tmp +
                        (f32.cos rand_numbers[i]) * acc[i+1])
        in  acc
```

[2][Link to random module]
[3][Link to shuffle module]
[4][Link to complex numbers module]

Instead of using function composition as in equation 3, a loop is used. Here the rotation of coordinates of a point are computed in place. The matrix vector multiplication of equation 2 is calculated in line 5 to 8. The use of a loop means that applying **Q** on a single point utilizes no parallelism. However since **Q** is mapped across all points in the dataset, this is not a big issue.

6.2.3 F

F uses 3 functions. **Z** to have a single point go from $\mathbb{R}^d \to \mathbb{C}^{d_2}$. **T** a fast Fourier transform. And then $\mathbf{Z}^{-1}$ to get the point back to $\mathbb{R}^d$ i.e. $\mathbb{C}^{d_2} \to \mathbb{R}^d$. To do this the 'complex' module is used to represent complex numbers.
Z is calculated as follows:

```
1  let Z [d] (point: [d]f32) (d2 : i64) : [d2](f32,f32)  =
2      map (\i -> c32.mk (point[i*2]) (point[(i*2) + 1])) (iota d2)
```

`c32.mk` in line 2 makes a complex number from 2 consecutive coordinate values from a point as seen in equation 5. `d2` is $\frac{d}{2}$. A map is used to create the complex numbers as described in equation 5.
$\mathbf{Z}^{-1}$ simply extracts the real number and then the imaginary number to get the point back into $\mathbb{R}^d$:

```
1  let Z_inv [d2] [d] (TZ: [d2](f32,f32))  (point_orignal : [d]f32)
2                                                       : [d]f32 =
3    map (\i -> if (i%2==0 && i == d-1) then point_orignal[d-1]
4               else if (i%2 == 0)  then c32.re TZ[i/2]
5               else c32.im TZ[i/2]
6        ) (iota (d))
```

Here `TZ` is a point after **Z** and **T** has been applied to it where `point_orignal` is the point before they have been applied. Notice the conditional in line 2. Since if d is odd you only apply **F** on the first $d-1$ coordinates. This means you extract the final coordinate from the original point if the conditional in line 2 holds. In this way **F** is not applied to the last coordinate. `c32.re` and `c32.im` extract the real and imaginary part of a given complex number.
Now only **T** remains. The $d2 \times d2$ matrix specified by equation 4 can be computed with 2 maps as shown in the code snippet for calculating **F**:

```
1   let Fd [d] (point: [d]f32) (d2 : i64) : [d]f32  =
2     let pZ = Z point d2
3     let one_over_d2sqr = c32.mk_re (1.0 / (f32.sqrt d2) )
4     let T = map (\k -> map (\l ->
5                     let exponent_im = c32.exp (c32.mk_im
6                         (-(2.0 * f32.pi * (k) * (l)) / (d2)))
7                     in one_over_d2sqr c32.* exponent_im
8                 ) (iota d2) ) (iota d2)
9     let pTZ = mat_vec_complex T pZ
10    in Z_inv pTZ point
```

As seen in line 9 the function composition of equation 6 ($\mathbf{F} = Z^{-1} \circ T \circ Z(v)$) is actually a matrix vector multiplication for the application of $T \circ Z(v)$. The matrix vector multiplication is done with a map over the matrix **T**, then a map that multiplies each element in the row of **T** with the coordinate values of **Z** applied to the point, `pZ`, then finally a reduce with the (+) operator.

6.3 Building the kd-tree

Building the kd-tree can be described with the pseudocode from section 4.3:

```
Build_kdtree (reference_set A, height h) =
    nodes_in_level[0] = A
    for (i = 0; i < h; i++)
        coordinate = i mod d
        nodes_in_level[i+1] =
            foreach node in nodes_in_level[i]
                median = find median of coordinate in node
                new_nodes = partition the node according to median
```

To implement this one needs to know the height, how to construct the loop, how to find the medians and how to partition/reorder the nodes. Another thing that is important to note is that the tree will be represented with a flat array. This is needed to avoid a bug that happens if you just partition at the index in the middle. The bug can be described as follows: Imagine a node that at level **i** contains many values equal to the median of level *i*. If the method of splitting at the middle is used, you end up with two nodes which can contain values that are equal to those of the other node. This causes problems when trying to find the natural leaf of query points.
Here is an example of a kd-tree of height 1 where you split the node in the middle:

```
query: [1,900]    A: [[1,700],[1,990],[1,3],[1,2]]
kd-tree:
level 0 [[1,700],[1,990],[1,3],[1,2]] Median: 1
level 1 [[1,700],[1,990]]  [[1,3],[1,2]]
```

When finding the natural leaf for the query since the first coordinate of the query is $\geq$ the median ($1 \geq 1$), it will visit the child to the right. However the query is much closer to the points in the left child. By allowing the leaves to have a flat representation you can move all the points with the same value as the median to the same child:

```
kd-tree:
level 0 [[1,700],[1,990],[1,3],[1,2]] Median: 1
level 1 []    [[1,700],[1,990],[1,3],[1,2]]
```

Now the query will find its natural leaf correctly when comparing its coordinate values to the medians.
To represent a flat array you need a shape array. A shape array shows the sizes of each segment of the flat array. E.g. the shape array `[3,4]` contains 2 segments where the first has 3 elements and the second has 4. The example from before also shows that some nodes can have a shape value of 0. This can be seen in level 1, where the shape of the level is $[0, 4]$. Special care needs to be taken to accommodate such 0 element nodes.

6.3.1 The height and loop

The height is determined by the leaf size and is calculated like this: `height = log (m / leaf_size)`, where `m` is the number of points in **A**. Since the tree is represented with a flat array so that the sizes of different leaves can be different, the `leaf_size` is just the average leaf size. For this implementation the average leaf size is chosen to be 256. The effect of different leaf sizes is explored in section 7.6.
As for the loop it is sequential and iterates through the levels starting at level 0. For iteration i the following is calculated for level i: The medians of the nodes in level i, the partitioning of the nodes in level i, the shape of the next level's nodes and an indirect array for the points. The indirect array are the original indices of the reordered points. The indirect array is therefore how you keep track of how the points have been reordered.
The next level, being $\mathbf{i}+1$, is then computed by using the information from computing level **i**. This continues until **i** is equal to the `height`.

6.3.2 Computing the medians

This section shows how the medians are computed for 1 iteration of the loop. I have been given a rank-k search solution that operates on a flat array and finds the **k**th largest value of each segment/node in the flat array. This is used to find the median. The solution needs: an approximation of the means of each node, the **k** of each node you want to find, the shape of the nodes this level, the II1 of the flat array and finally the values of the flat array.
The II1 can be described as follows: If there are **o** segments, the II1 will have the same length as **A**, and each element will indicate the segment number (plus one) in which the current element of **A** resides. An example of the II1 can be seen at the end of this section (section 6.3.2)
The shape of this level was already calculated by the computation of the previous level.
The **k**s can be easily calculated with a map over the shape. Since the value you want is the median of the node, you can just divide the shape values by 2.
Since only one of the dimensions of the points in the flat array is used only the values in that dimension is extracted with a map. This representation of the values in the flat array is called '`chosen_dim`' in the following code snippets. The dimension in question is determined by the current level as with the *coordinate* value from the pseudocode at the start of section 6.3.
The approximation of the means are calculated with 2 scans to find the minimum and the maximum value of each segment/node. Then adding these values together and dividing the result by 2. However since the representation is flat using a normal scan will not work. This is because it is the minimum and maximum values of each segment that is needed and not the minimum and maximum values of the whole array.
The scan that is needed is a segmented scan: `sgmscan`. This works like a regular scan, but the accumulator value is reset to the neutral element when a new segment begins. An example of how a segmented scan works:

```
Shape array: [2,3], flat array [1,2,3,4,5], neutral element: 0
sgmscan of flat array with the operator (+) = [1,3,3,7,12]
```

To know where the segments end and begin a flag array of the shape is created. A flag array indicates the start of a segment with a value while having all other values be a 'zero' element. It's length is the sum of the shape array used to create it. Furthermore it also uses a 'value array' of the same length as the shape array that indicates the value of each new segment start. Here

is an example of a flag array of the shape array [2,3] a zero value 0 and a value array [1,2]:
mkFlagArray [2,3] 0 [1,2] = [1,0,2,0,0]
Here is the code for finding the approximate means:

```
1 let shp_flag_arr =  mkFlagArray shp_this_lvl (0i32)
2                  (replicate nodes_this_lvl 1i32) :> [m]i32
3 let mins = sgmscan f32.min f32.highest shp_flag_arr chosen_dim
4 let maxs = sgmscan f32.max f32.lowest shp_flag_arr chosen_dim
5 let means = map (\i -> if i == 0 then 0
6             else (mins[i-1] + maxs[i-1])/2.0f32 ) scan_shp_this_lvl
```

shp_this_lvl describes the shape of the nodes at the current level. nodes_this_lvl describes how many nodes there are at the current level. Because of the sgmscans, the minimum and maximum values will be at the end of each segment so they need to be extracted. This can be done as seen in line 5 and 6 with the scanned shape array of this level. The scanned shape array will indicate where each segment ends plus 1. The scanned shape array is computed with a regular scan with the (+) operator over the shape array.

The II1 and the medians are computed with the following code snippet:

```
1 let flag_iotap1 = [0...nodes_this_lvl - 1] |> map (+1)
2                 |> mkFlagArray shp_this_lvl (0i32)) :> [m]i32
3 let II1 = sgmscan (+) 0 flag_iotap1 flag_iotap1
4 let medians_this_lvl = rankSearchBatch means ks
5                          shp_this_lvl (copy II1) (copy chosen_dim)
```

flag_iotap1 is the flag array with the shape array being the shape array for this level and the value array being [1...nodes_this_lvl]. flag_iotap1 can be illustrated with this example:
flag_iotap1 = mkFlagArray [3,0,0,3] 0 [1,2,3,4] = [1,0,0,4,0,0]
As can be seen in the example some segments can potentially contain 0 elements and shp_this_lvl contains this information. The shape array from the example; [3,0,0,3] has 2 segments/nodes with 0 elments in them. By using the shp_this_lvl for the flag array these 0-element segments are ignored.
By using a sgmscan as show in line 3 the II1 is created. If this was done on the example it would result in: [1,1,1,4,4,4].
The medians are computed by providing the supplied rank-k search solution with the arguments mentioned above (as seen in line 4 and 5).

6.3.3 Partition the nodes

Now that the medians for the nodes at the current level have been calculated and stored in medians_this_lvl the nodes can be partitioned.
When partitioning the nodes a mask is needed. This mask array shows with a Boolean if the values of chosen_dim is $<$ or $\geq$ than the median of the segment the value belongs to:

```
let mask = map2 (\v i -> v < medians_this_lvl[i-1]) chosen_dim II1
```

Partitioning the nodes with the implementation below accomplishes two things: It reorders the nodes and returns where the nodes 'split'. Where the nodes split is important to know when determining the shape of the next level's nodes. E.g. if you have a node with shape [7] and it's split is 5, then there are 5 elements in the node that are true under the mask. This means that the node will be partitioned into 2 nodes whose flat shape array would be [5,2].

```
def partition2L 't [n] [p] (mask : [n]bool) (shp_flag_arr: [n]i32)
                (sc_shp: [p]i32) (shp : [p]i32, flat_arr : [n]t)
                                      (dummy : t) : ([n]t, [p]i32)
```

The *partition2L* function takes as input the mask, shape information for this level and a flat array (in this case the indirect representation of it). The shape information for the current level is the flag array of the shape called `shp_flag_arr`, the scanned shape called `sc_shp`, and the shape called `shp`.
First the splits are computed. This can be done by counting how many values in a given segment are less than the median. The splits are computed in the code snippet along with an example:

```
1  let ffs = map (\f -> if f then 0 else 1) mask
2  let tfs = map (\f -> if f then 1 else 0) mask
3  let split_tmp = sgmscan (+) 0 shp_flag_arr tfs
4  let splits = map2 (\s off -> if s == 0 then 0 else split_tmp[off-1])
5                                              shp sc_shp :> [p]i32
6  Example:
7  --ffs =        [0,1,1,0,0,1,1,0,1]       shp    = [2,3,4]
8  --tfs =        [1,0,0,1,1,0,0,1,0]       sc_shp = [2,5,9]
9  --split_tmp = [1,1,0,1,2,0,0,1,1]
10 --splits = [1,2,1]
```

Here *tfs* converts the mask Booleans into 1s and 0s. *ffs* is the same but where false values equates to 1s and true values to 0s. The `sgmscan` in line 3 counts how many true values that are in each segment such that the end of each segment in `split_tmp` will contain the number of true values for that segment. These are then extracted.

Now that the splits have been calculated how the nodes should be partitioned is computed. The partitioning of a given node will place the elements that are $<$ than its median to the left and the elements that are $\geq$ than its median to the right. The partitioning can be described as follows:

- Find the indices of where the elements that hold under the mask should be placed.
- Do the same for elements that do not hold.
- Combine them such that the new indices for all elements are known
- Use the new indices to reorder the indirect array.

To do this 2 arrays needs to be created: `isT_segments` and `isF_segments`. `isT_segments` will contain the indices of where the elements that hold under the mask should be placed within the flat array. `isF_segments` will do the same but for the elements that do not hold under the mask. Here is an example that shows the whole process of reordering the points by showing how `isT_segments`, `isF_segments`, their combination and a reordered flat array will look like with the following shape, `tfs`, `ffs` and flat array:

```
Flat array values     = [0,1,  2,3,4,  5,6,7,8]
Shape: [2,3,4]  tfs   = [1,0,  0,1,1,  0,0,1,0]
                ffs   = [0,1,  1,0,0,  1,1,0,1]

isT_segments          = [1,1,  2,3,4,  5,5,6,6]
isF_segments          = [1,2,  5,5,5,  7,8,8,9]
combination           = [1,2,  5,3,4,  7,8,6,9]

Reordered flat array = [0,1,  4,2,3,  6,7,5,8]
```

As can be seen in the example `isT_segments` and `isF_segments` has a value for each element in the flat array. `isT_segments` uses `tfs` to determine its indices by increasing the indices every time a 1 is encountered in `tfs`. Notice that at the 3rd, and 6th entry in `isT_segments` increases in value even though even though `tfs` contains a 0 at these entries. This is because new nodes are entered (see the shape array). Since elements of one node should not be placed in the node left of it, you need to insert the total amount of elements encountered so far when entering a new node.
`isF_segments` has the same relation to `ffs` as `isT_segments` has to `tfs`. Since the elements that are false under the mask are placed to the right of the ones that are true, `isF_segments` has to insert the last value of each segment in `isT_segments` at the start of each node/segment. This can be seen in the 1st, 3rd and 6th entry in `isF_segments`. In the 3rd and 6th entry `ffs` also contains a 1 so the value is also increased by one here.
For computing the combination (the new indices of the flat array) one iterates through the mask or tfs. If a 1 is encountered you read the value from `isT_segments`. If a 0 is encountered you read from `isF_segments`. The reordered flat array is calculated with a scatter, where the indices array is `combination` and the value array is `Flat array values`.

`isT_segments` can almost be computed with a single `sgmscan`. However, you still need to insert the amount of elements encountered so far when entering a new node. A `sgmscan` will ignore this. You therefore add the exclusive scanned shape array to the start of each segment of `tfs` before you use the `sgmscan` like so:

```
1 let isT_segments =
2   let tfs_add_shp = map (\i -> tfs[i] + i) exc_sc_shp
3   let tfs_with_inds = scatter (copy tfs) exc_sc_shp tfs_add_shp
4   in sgmscan (+) 0 shp_flag_arr tfs_with_inds
```

Here line 2 and 3 add the exclusive scanned shape array to the start of each segment of `tfs` and line 4 does the segmented scan.
Now `isF_segments` can be computed. First the last element of each segment of `isT_segments` need to be inserted to `ffs`.

```
let isTs_last_elem = map2 (\ind s -> if s == 0 then -1
                              else isT_segments[ind-1]) sc_shp shp
let isT_ind = map2 (\off s -> if s == -1 then -1 else off)
                                      exc_sc_shp isTs_last_elem
```

For `isTs_last_elem` the last elements of each segment of `isT_segments` are accessed with the scanned shape. However since there can be nodes of length 0 you need to check the shape to not extract the wrong information for those segments/nodes. `isT_ind` is the indices of where the `isTs_last_elem` elements should be added in `ffs`. `isF_segments` is calculated in a similar way as `isT_segments`:

```
let isF_segments =
  let ffs_add_Ts     = map2 (\i t_val -> ffs[i] + t_val)
                              exc_sc_shp isTs_last_elem
  let ffs_with_inds = scatter (copy ffs) (isT_ind) ffs_add_Ts
  in sgmscan (+) 0 shp_flag_arr ffs_with_inds
```

Now that `isT_segments` and `isF_segments` are calculated you can finally calculate the final indices (the `combination`):

```
1 let inds = map3 (\c iT iF -> if c then i64.i32(iT -1)
2                                     else i64.i32(iF -1)
3                ) mask isT_segments isF_segments
4 let result =  scatter (replicate n dummy) inds flat_arr
```

The `inds` array is computed with a map where the mask is checked. If the mask value is true you read from the `isT_segments` array. If it is false you read from the `isF_segments` array. Now how the values of `flat_arr` should be reordered is know. They can thus be scattered with `inds` as is seen in line 4.

Now the medians of the level, the new splits for the shape and how the elements should be partitioned have been calculated. This information is scattered into some arrays which can be accessed by the next level of the kd-tree. This process continues until the kd-tree reaches the desired height.

6.4 Searching the kd-tree

As was seen in section 5.4 there are 2 steps to searching the kd-tree and finding a $\mathbf{V}_i$ for each query $\mathbf{x}_i$.

- First: The natural leaf and the path to it is found for each query.
- Then finding all the leaves whose path differ at 1 level from the natural path.

6.4.1 Finding the natural leaf

After building the kd-tree the leaves, shape of the leaves, indirect array of the leaves and medians of the kd-tree are now known. The natural leaf is found with the following map over each query:

```
let queries_init_leafs = map (findLeaf median_dims median_vals
                                                height) queries
```

`findLeaf` uses the median values and dimensions of the kd-tree to find where each query naturally belongs. This is computed by having a loop iterates through each level of the kd-tree. Here the median's value and query's value are compared. If the query value is less than the median for a given node, you travel to the left child of the node. Otherwise you travel to the right child. The result is a leaf number (an integer). The number also doubles as the natural path. This is because you can just flip a bit **j** of the leaf number and thus get a new leaf number with a path that differs at 1 level from the natural path.

An optimization of the memory accesses to the leaves is performed by sorting the queries according to their natural leaves. The effect of this optimization is explored in section 7.5. This is done with the following code:

```
1 let (sorted_query_with_ind, sorted_query_leaf) =
2   let q_with_ind = zip queries (iota n)
3   let q_ind_with_leaf = zip q_with_ind queries_init_leafs
4   in (radix_sort_int_by_key (\(_,l) -> l) (log2Int (length leaves))
5                              i64.get_bit q_ind_with_leaf) |> unzip
```

Here the queries and their initial index is zipped together and then together with the queries' natural leaves in line 2 and 3. The initial index is needed such that after the sort, each query is still identifiable. Radix sort from the radix sort module[5] is then used in line 4 and 5 to sort this according to the natural leaf number of each query. The result is a tuple where the first element contains the sorted queries with their initial index and the second element is the sorted natural leaves of the queries.

6.4.2 Finding the rest of $\mathbf{V}_i$

A loop is used to iterate through all the paths that differ at 1 level from the natural path. In each iteration the **j**th bit of the leaf number is flipped. This is done with the following:

```
let new_leaves = map (\l_num -> reverseBit l_num j) sorted_query_leaf
```

Here the sorted natural leaf numbers of the queries have their **j**th bit flipped by `reverseBit`. By doing this for *j = 0* to *j = height - 1*, all the leaf numbers whose path differ at 1 level from the natural path of each query is found. Now both the natural leaf and all the other leaves of $\mathbf{V}_i$ for each query $\mathbf{x}_i$ have been found.

6.5 Searching the subsets $\mathbf{V}_i$ (the leaves)

$\mathbf{V}_i$ is used to update the knns of a query by first searching the natural leaf for new knn candidates and then the rest of the leaves in $\mathbf{V}_i$ one at a time. This is done since the natural leaf is found first and then the rest of the leaves are found one at a time in a loop afterwards as described in the previous section. The true nearest neighbours within the leaves for each query is searched with the following function:

[5][Radix sort module link]

```
 1  def bruteForce [m][d][k] (query: [d]f32) (knns0: [k](i32,f32))
 2                           (refs: [m](i32,[d]f32)) : [k](i32,f32) =
 3    loop (knns) = (copy knns0)
 4      for i <  m do
 5        let dist = sumSqrsSeq query (refs[i].1) in
 6        if dist >= knns[k-1].1 then knns -- early exit
 7        else let ref_ind = refs[i].0 in
 8          let (_, _, knns') =
 9            loop (dist, ref_ind, knns) for j < k do
10              let cur_nn = knns[j] in
11              if cur_nn.0 == ref_ind
12                then (f32.inf, -1, knns) -- already encountered
13              else if dist >= cur_nn.1
14                then (dist, ref_ind, knns)
15              else let tmp_ind = cur_nn.0
16                let knns[j] = (ref_ind, dist)
17                let ref_ind = tmp_ind
18                in  (cur_nn.1, ref_ind, knns)
19          in  knns'
```

`bruteForce` updates the knn candidates of one query. It uses a query, the queries current knns (`knns0`) and some reference points (`refs`) which are the points inside the leaves of $\mathbf{V}_i$. Each nearest neighbour in `knns0` is stored with a tuple where the first element is the index of the nearest neighbour candidate inside of **A**, and the second element is the distance to this nearest neighbour candidate.

`bruteForce` iterates through the `refs` one by one with the for loop in line 3 and 4. It then uses the `sumSqrsSeq` function in line 5 to find the sum of the squares for the distance. `sumSqrsSeq` is sufficient for comparing the 'distance' between points. To optimize `bruteForce`, the square root of the sum is therefore not computed to find the true L2 distance. Line 6 then does an important optimization by exiting the iteration early if the distance is greater than the largest distance in `knns0`.

However if the distance is found to be better than the worst knn candidate the function enters a loop in line 9 that iterates through each nearest neighbour in the knns and updates the knns. The first thing that is checked is that the reference point has not been encountered before (line 11 and 12). It is possible to encounter reference points that have already been processed when **T** (the amount of times the algorithm loops in step 6) is greater than 1. This is because the pseudorandom orthogonal linear transformation of step 2 does not ensure that the $\mathbf{V}_i$ obtained from searching the kd-tree build from step 2's transformation contains only different points as opposed to the other $\mathbf{V}_i$s obtained in the other iterations of step 6. When a reference point that has already been encountered appears; no knn candidates are updated. This is done by setting the distance to infinity.

If the distance to the reference point is greater than to the current nearest neighbour in `knns` then the loop continues checking the other nearest neighbours in `knns` (line 13 and 14). However if the distance is smaller then `knns` is updated. The value that was updated is then propagated to the next iteration of the loop that runs through the `knns` of a query. This updates the rest of the `knns` array. This also ensures that the `knns` array is sorted. The fact that the `knns` are sorted is what enables the early exit of line 6.

Now that $\mathbf{V}_i$ has been searched for each query, good candidates for knns of each query have been obtained.

6.6 The loop: Overview of the implementation so far

```
def RANN [m] [n] [d] (T: i32) (k: i64) (reference_set: [m][d]f32)
                                       (queries: [n][d]f32) =
  -- Step 1: shift points
  let (ref_shifted, q_shifted) =
            shiftTandQPoints reference_set queries
  -- Setup for loop
  let init_knns = replicate n (replicate k (-1i32, f32.inf))
  let height =  ( log2Int (m / 256))

  -- Step 2-6 The loop:
  let new_knns =
    loop curr_nns = init_knns for t < T do
      -- Step 2 transformation
      let M1M2 = log2Int d
      let transformed_reference_set =
            pseudoRandomOrthogonalTransformation M1M2 t ref_shifted
      let transformed_queries  =
            pseudoRandomOrthogonalTransformation M1M2 t q_shifted
      -- Step 3 Build the kd-tree
      let (leaves, indir, median_dims, median_vals, shp_arr) =
                buildKdTree height transformed_reference_set
      -- Step 4 & 5 Search the tree and find new candidates
      in searchForKnns transformed_queries curr_nns leaves indir
                       median_dims median_vals shp_arr height

  let (knns_inds, _) =  unzip <| map (\i_knn -> unzip i_knn) new_knns
  in knns_inds
```

Above is the implementation so far. First all points are shifted towards the center of mass of the reference set. Then the loop is prepared. The loop has **T** iterations. On each iteration the reference set and query set is transformed with step 2. Then the kd-tree is build from the reference set. Finally the kd-tree is searched and a $\mathbf{V}_i$ is found for each query and then searched through to update the nearest neighbour candidates. After the loop it then returns the indices of the best knn candidates that were found.
The above implementation is what will be tested in the experiments section 7 when RANN is tested without supercharging enabled.

6.7 Supercharging

As mentioned in section 5.7 supercharging searches through k^2 new candidate points to obtain better knn candidates. These points are the knns of each of the knns of the queries. Supercharging also necessitates that good candidates for knns for each point in the reference set **A** within **A** are known or found. It is implemented in the code below where the current knn candidates for the queries are `knn_inds_q` and the candidates for knns of the reference set is `knn_inds_r`:

```
 1  let supercharging =
 2    loop (curr_knns_q) = (new_knns_q) for i < k do
 3      let ref_k_inds =
 4        map (\knn_ind_q ->
 5          map (\q -> knn_inds_r[knn_ind_q[i],q]) (iota k)) knn_inds_q
 6      let super_points =
 7        map (\inds ->
 8          map (ind ->
 9            let point_tmp =
10              map (\k_ind -> reference_set[ind, k_ind]) (iota d)
11            in (ind, point_tmp) ) inds) ref_k_inds
12      in  map2 (\refs ind ->
13                  bruteForce queries[ind] curr_knns_q[ind] refs)
14                  super_points (iota n)
```

Since the total amount of points which supercharging extracts from **A** is very high ($n \cdot k^2$), a loop is used to only extract $n \cdot k$ points in each iteration.
First the k indices of the k^2 supercharging candidates are extracted with a map in line 3 - 5. Then these indices are used to extract the k points called `super_points` from the reference set with a map in line 6 to 11. In both the extraction of `ref_k_inds` and `super_points` a second map is used to achieve coalesced access for the threads. Coalesced access means that consecutive threads access consecutive memory locations. Once the `super_points` are extracted they are used as the `refs` of the `bruteForce` function presented in section 6.5 to refine the knn candidates of each query.
When the loop has finished, `supercharging` will return better knn candidates for each query.

6.8 Computational complexity

For analysing the high level computational complexity of the implementation, the leaf size and the dimensionality d will be treated as constants. The work and span of each step except step 6 (the loop) is analyzed. Since steps 2 to 5 are inside the sequential loop of step 6 their work and span can be multiplied with **T**. **T** is usually a small value.

- Step 1's (shifting the points) complexity is dominated by a reduce over the reference set and maps over the query set with n points and the reference set with m. The work is therefore $O(n + m)$. The span is $O(\log m)$.
- Step 2's (pseudorandom orthogonal linear transformations) complexity is dominated by the maps over the query set and reference set with the functions of **P**, **Q** and **F**.
  P, **Q** and **F** are $O(1)$ since they all make operation over the values of a point with dimensionality d. The work of step 2 is therefore $O(n + m)$ and the span $O(1)$.
- Step 3's (building the tree) complexity has a loop over the height of the tree with $\log m$ iterations. All the prerequisites for computing the medians at level **i** are done with scans over coordinate **i** for all points in the reference set. The medians themselves can be computed in

linear time. The partitioning in level **i** is done with scans and maps over all the points in the reference set. The work is therefore $O(\log m \cdot \log m \cdot m)$ and the span is $O(\log m \cdot \log m)$.

- Step 4's (searching the tree) complexity depends on finding the natural leaf of each query, sorting the queries according to these leaves and finding the rest of the leaves that make up $\mathbf{V}_i$ for each query. Finding the natural leaf for a single point is done by traversing through the kd-tree with height $\log m$ with a sequential loop. This is mapped across each query.
  Sorting the leaves is done with radix sort which takes $O(l \cdot n)$ work and $O(l \cdot \log n)$ span, where l is the number of bits in each element that is sorted. The bits in each element is $\log(number\_of\_leaves)$. Since the leaf size is a constant the number of leaves is considered to be $O(m)$. Radix sort therefore has work: $O(\log m \cdot n)$ and span: $O(\log m \cdot \log n)$
  The rest of the $\log m$ leaves are found within a sequential loop by flipping a bit in the natural leaf of every query. The total work of this step is $O(n \cdot \log m)$ and the span is $O(\log m + \log m \cdot \log n)$

- Step 5's (searching the leaves) complexity is dependant on the bruteForce function that is applied to each query. For a single point the function has a loop that iterates over the $\log m + 1$ leaves of V_i's points. Since the leaf size is treated as a constant the number of leaves in V_i just equates to $O(\log m)$. In each iteration it can potentially loop over all the knns ($O(k)$). The work of this step is therefore $O(n \cdot (\log m \cdot k))$ and the span is $O(\log m \cdot k)$

- Step 7's (supercharging) complexity is also dependant on the bruteForce function. This time the number of points bruteForce iterates over for a query is k^2. The work is therefore $O(n \cdot k^2 \cdot k)$ and the span is $O(k^2 \cdot k)$.

7 Experiments

To measure the performance of the data parallel versions of RANN the accuracy and runtime is measured. This is done on a machine with an AMD EPYC 7352 24-Core Processor with 2 threads per core, 256 GBs of DRAM and an Nvidia A100-PCIE GPU with 40 GBs of memory.
The algorithms used are implemented in Futhark version 0.25 and compiled with the cuda backend. These include the data-parallel version of RANN, the data-parallel version of RANN with supercharging and a brute force exact implementation, used to measure accuracy and runtime against. The brute force exact implementation uses a similar function as the one used by RANN when a point $\mathbf{x}_i$ searches for better candidates in $\mathbf{V}_i$. However instead of $\mathbf{V}_i$ being the reference set obtained by traversing a k-d tree, the reference set is the whole reference set $\mathbf{A}$. This ensures that the true nearest neighbours in the reference set are found for each $\mathbf{x}_i$. The parallelism for the exact knn implementation is achieved by mapping `bruteForce` (shown in section 6.5) across all queries.

7.1 Testing methodology

In the following experiments the default leaf size will be 256 unless otherwise specified. The reference set and query set will each contain 500,000 unless otherwise specified. When testing for accuracy only 100 queries will be selected from the query set to compare the accuracy of a given configuration of RANN to the true nearest neighbours obtained by the exact knn implementation.
The accuracy is computed in the following way: Accuracy is the percentage of candidates found by RANN that are true nearest neighbours. This is checked by seeing if the exact knn implementation found the same nearest neighbours as RANN. Accuracy is thus calculated by simply dividing the number of true nearest neighbours with the total amount of neighbours:

$$Accuracy = \frac{True\ nns\ found}{k \cdot n}$$

This also means that an accuracy of e.g. 50% is extremely good since this means that half of the candidates found by RANN are true nearest neighbours while the rest are good approximations.
The fact that the rest are good approximations can be checked by telling RANN to e.g. find the 5 nearest neighbours (k = 5) of all queries and checking against the exact knn implementation. And then afterwards checking the result of RANN from before against exact knn implementation k = 10. If the accuracy increases by doing this, it shows that the candidates found by RANN that were not true nearest neighbours were still good candidates.
Here is an example of this looser interpretation of accuracy: The RANN configuration with k = 5, t = 5 and no supercharging on a dataset as the one described at the start of this section with d = 50 had accuracy: 49.2% . However when you compare the above RANN configuration to when the exact knn implementation finds k = 10, it has accuracy: 84%. This means that 84% of RANNs candidates were among the 10 nearest neighbours and 49.2% were among the 5 nearest neighbours in a reference set of 500,000 points.

7.2 RANN vs Brute force

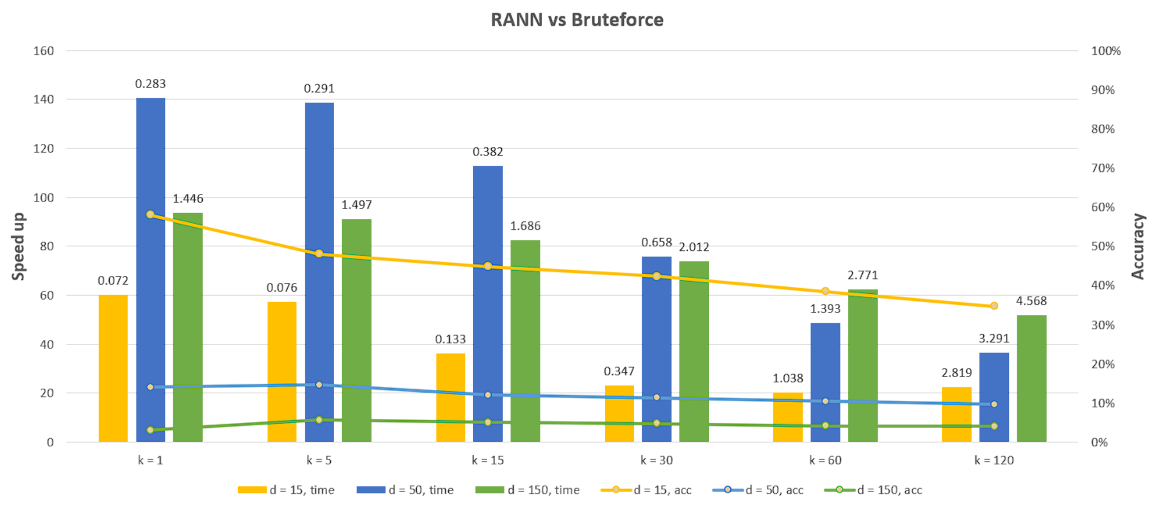

Figure 2: The performance of RANN compared to the parallel exact knn implementation. The RANN configuration is: T = 1 without supercharging. The bars represent the speedup (how many times faster RANN is than the parallel exact knn implementation. The y-value on the left). The numbers above the bars show the actual time RANN took in seconds. The lines show the accuracy (the y-values on the right). The bar and line of the same color was run on a dataset with the same d.

First RANN and the parallel exact knn implementation are compared. For this comparison the configuration of RANN are T = 1. The results can be seen in figure 2. Here speed up on the left y axis is measured by: $\frac{Bruteforce\_time}{RANN\_time} = Speed\ up$. E.g. the left most yellow bar shows that RANN with k = 1, d = 15 is 60 times faster than the parallel brute force with k = 1, d = 15. The accuracy of RANN can be seen on the right y-axis.
From the results RANN is between 20.4 and 140.7 times faster than the parallel exact knn implementation. The greatest speed ups are seen when d = 50 and k is relatively low. The falling speed up as k gets larger makes sense since when k is larger, RANN has to use more time on finding the true nearest neighbours with the `bruteForce` function in the $\mathbf{V}_i$ sets.
In terms of the accuracy of RANN, the most important factor is the dimensionality. As d increases, the accuracy decreases. k has the same relationship with accuracy. Since the accuracy is higher when k is small, it indicates that RANN is best at finding the absolute closest nearest neighbours first.

7.3 The impact of T

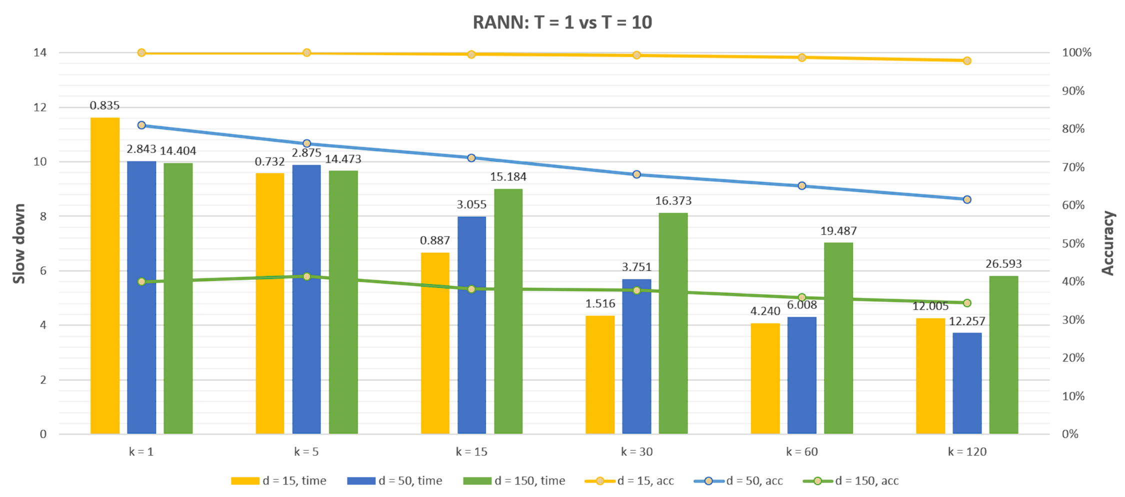

Figure 3: The performance impact on RANN of an increased T value. The diagram layout is similar to figure 2. However it now shows slow down instead of speed up on the left y-axis. Slow down shows how many times slower RANN with T = 10 was to RANN T = 1. The values above the bars are the time it took RANN with T = 10 in seconds.

Now the impact of increasing T is measured. The results can be seen in figure 3. The slow down is measured similarly to how speed up was measured in figure 2: $\frac{T10\_time}{T1\_time} = slow\ down$. Thus this shows how many times RANN with T = 1 is faster than RANN with T = 10.
As T = 10 means that the algorithm loops 10 times more than when T = 1, one would assume that this means that it would generally be 10 times slower. This is true for low values of k, however when k increases the slow down also becomes less severe. E.g. when k = 120, T = 10 it is only about 4 times slower than T = 1 for d = 15 and 50. This could be because of the early exit functionality of the `bruteForce` function. k has a large impact on how many comparisons `bruteForce` needs to make. When the algorithm has looped a couple of times the k nearest neighbours of each point will already have some good candidates. It then becomes increasingly more rare that a better closer candidate is found on the later iterations resulting in many early exits.
The accuracy is also greatly improved if it is compared to the accuracy of T = 1 show in figure 2. The impact of T on accuracy is shown in more detail in section 7.7.

7.4 Supercharging

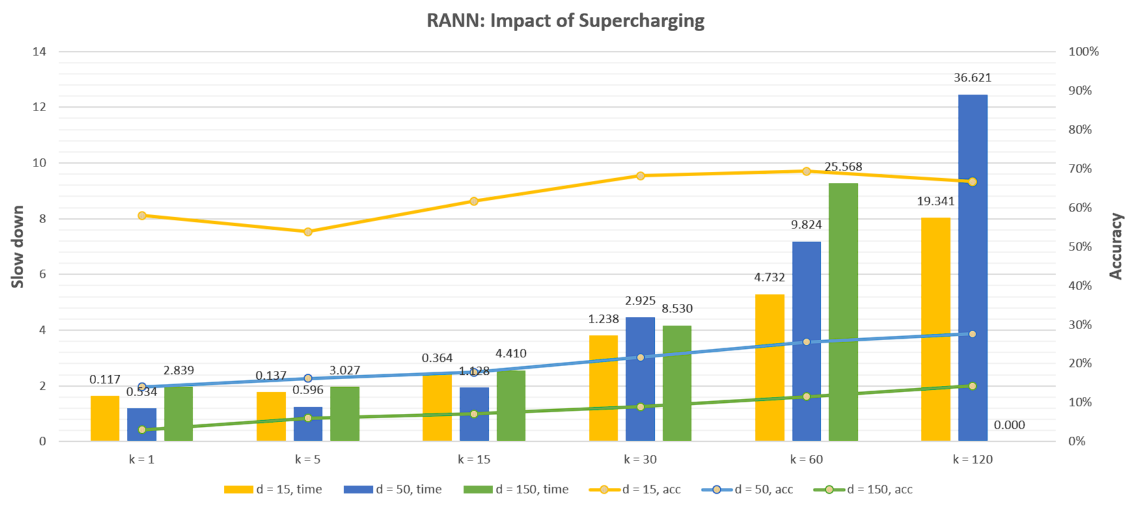

Figure 4: The performance impact on RANN with supercharging enabled. Here RANN with supercharging and T = 1 is compared to RANN without supercharging and T = 1. The diagram layout is identical to figure 3. The numbers above the bars show how long RANN with supercharging took in seconds.

The results of supercharging with T = 1 can be seen in figure 4. Here the slow down is compared to RANN with T = 1 and no supercharging. Note that the GPU runs out of memory for k = 120 and d = 150 and therefore this has no slowdown value. The largest k-value for this dimensionality (d = 150) is k = 63. This is because the implementation of supercharging's memory footprint becomes very large as k increases. Another thing to note is that both the query set and reference set were searched for knns. This means that this configuration of RANN with supercharging should have at minimum a *slow down* ≈ 2.
The run time impact of supercharging remains relatively stable for lower values of k. However when k is above 30 the slow down becomes more severe. This is because supercharging searches through k^2 candidates with bruteforce. The stable value of a 2 times slow down for the lower k values indicates that the dominant reason for the slow down is the fact that this implementation of supercharging necessitates that nearest neighbour candidates for the points in the reference set are also found.
The accuracy generally improves however for large k values the slow down is too severe. Improving the T value would therefore yield better performance over all. The accuracy of supercharging is also explored in section 7.7.

7.5 The impact of sorting the leaves

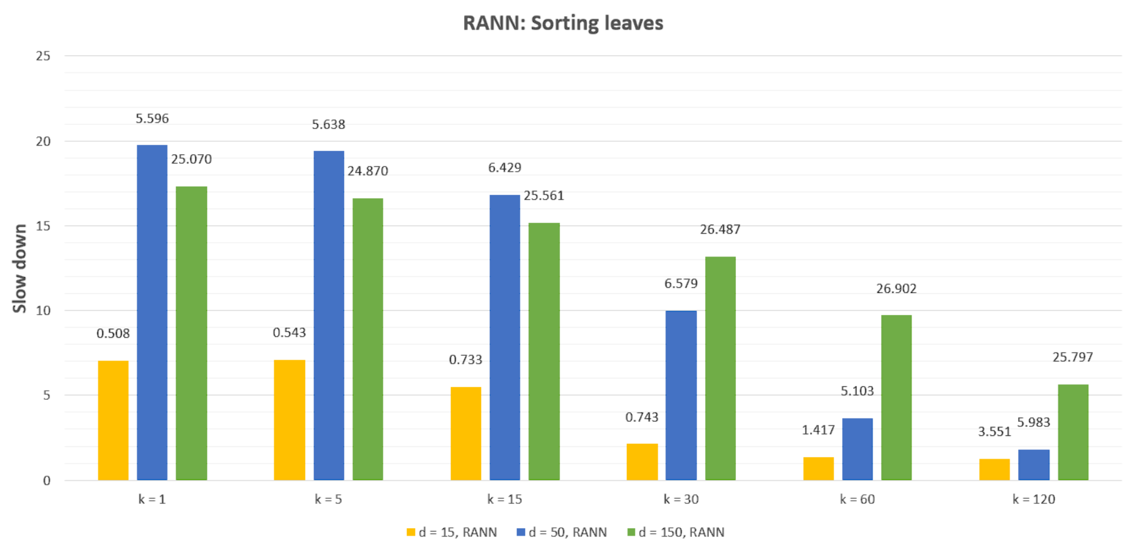

Figure 5: The runtime impact on RANN when the leaves are sorted vs when they are not. The diagram has the same layout as the previous ones, minus the accuracy data. The numbers above the bars show how long the not sorting approach took in seconds.

Whether or not you sort the points by their natural leaf has no impact on accuracy. Sorting the points should however improve memory accesses. The runtime impact of sorting the points vs not sorting them with T = 1 can be seen in figure 5.
What is worth noting is that not sorting the points is a major bottleneck when d = 50 and 150 and also to a lesser extend d = 15. Here the actual runtime stays almost the same across all the tested k values (see the values above the bars) for a given dimensionality. The slowdown will thus decrease as k increases. The results show that sorting the points by their natural leaf provides a huge performance boost.

7.6 The impact of leaf size

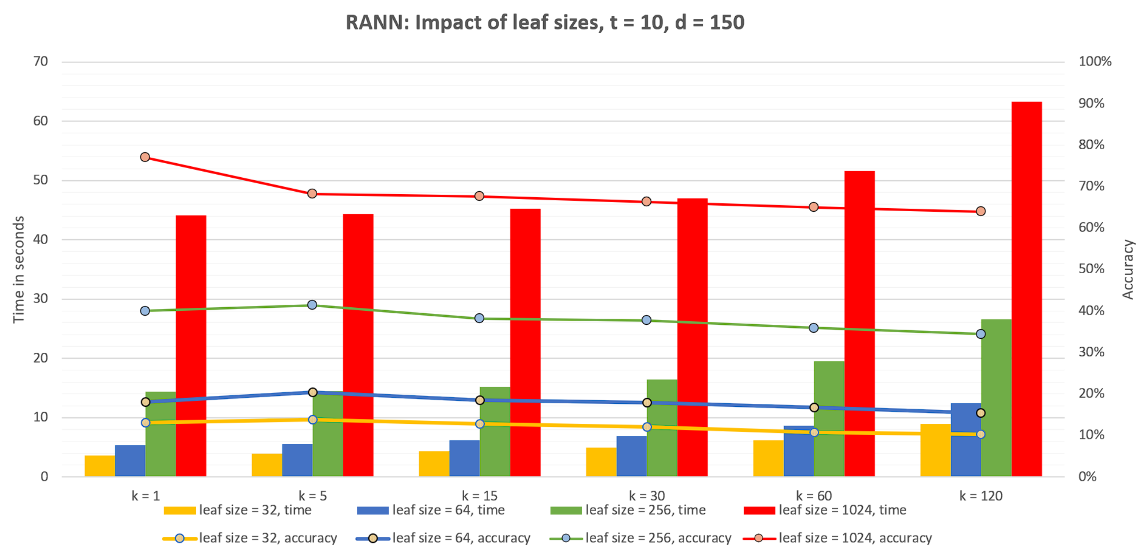

Figure 6: The performance impact on RANN of different leaf size values. Here the RANN configuration is $\mathbf{T} = 10$ and no supercharging. The layout of the diagram is similar to the previous ones. However the left y-axis just depicts the time it took the various configurations of leaf sizes in seconds.

As the average size of the leaves increases so does the amount of points in $\mathbf{V}_i$. The average leaf size contributes to the computation of the height of the kd-tree as seen in section 6.3.1. This means the `bruteForce` function has to search through more points. Since more points are searched one would expect the accuracy and runtime to increase. The performance of different values of average leaf size (32, 64, 256, 1024) can be seen in figure 6.
The figure makes it evident that the accuracy and runtime do indeed increase when the leaf size is increased. The data set is run on points with d = 150. This results in the smaller leaf size values becoming very inaccurate. If the dimensionality was lower you could imagine that these smaller leaf sizes would begin to have more acceptable accuracy levels. Adjusting the leaf size to the given dataset to customize the desired amount of accuracy/runtime is therefore an option. Below is also a figure that is identical to the one above, however the accuracy has been loosened up in the same way as it was described in section 7.1.

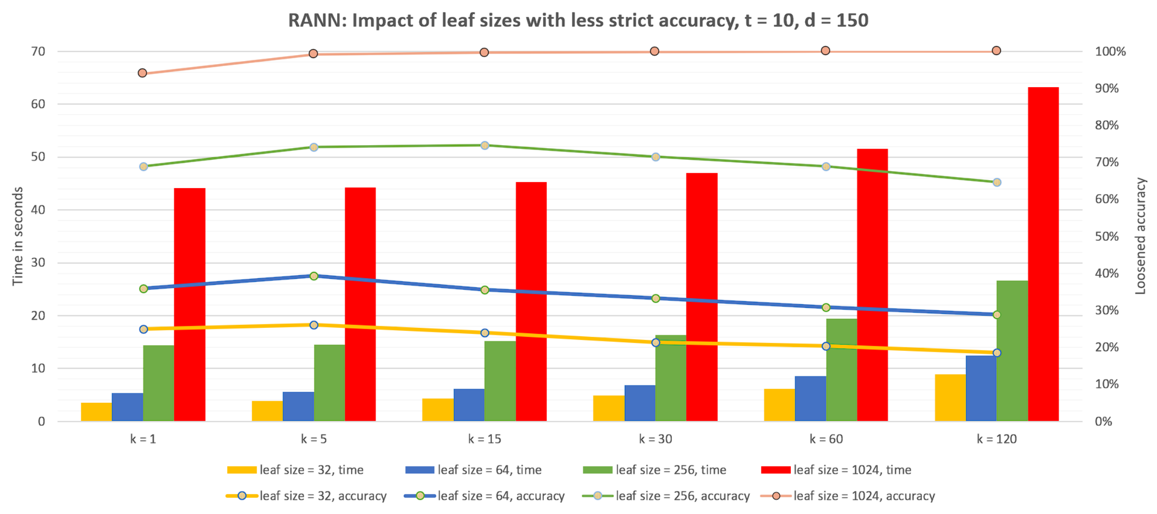

Figure 7: This is the same chart as figure 6, however the accuracy has been loosened in the same way as was described in the testing methodology section 7.1

As can be seen in figure 7 the accuracy is generally much higher when measured in this way. When the leaf size is 1024 the loosened accuracy is almost 100% for all k values. The default leaf size of 256 also has a very respectable accuracy while the loosened accuracy of the leaf sizes 32 and 64 is still a little low.
RANN with leaf size 256 is generally 2.5 to 3 times faster than RANN with leaf size 1024. Since the default leaf size of 256 already provides a good accuracy it is debatable whether or not the increased runtime cost is worth the extra accuracy. The worsening accuracy when d increases as shown in the previous figures does however indicate that the increased leaf size will become necessary for datasets with a larger dimensionality than d = 150.

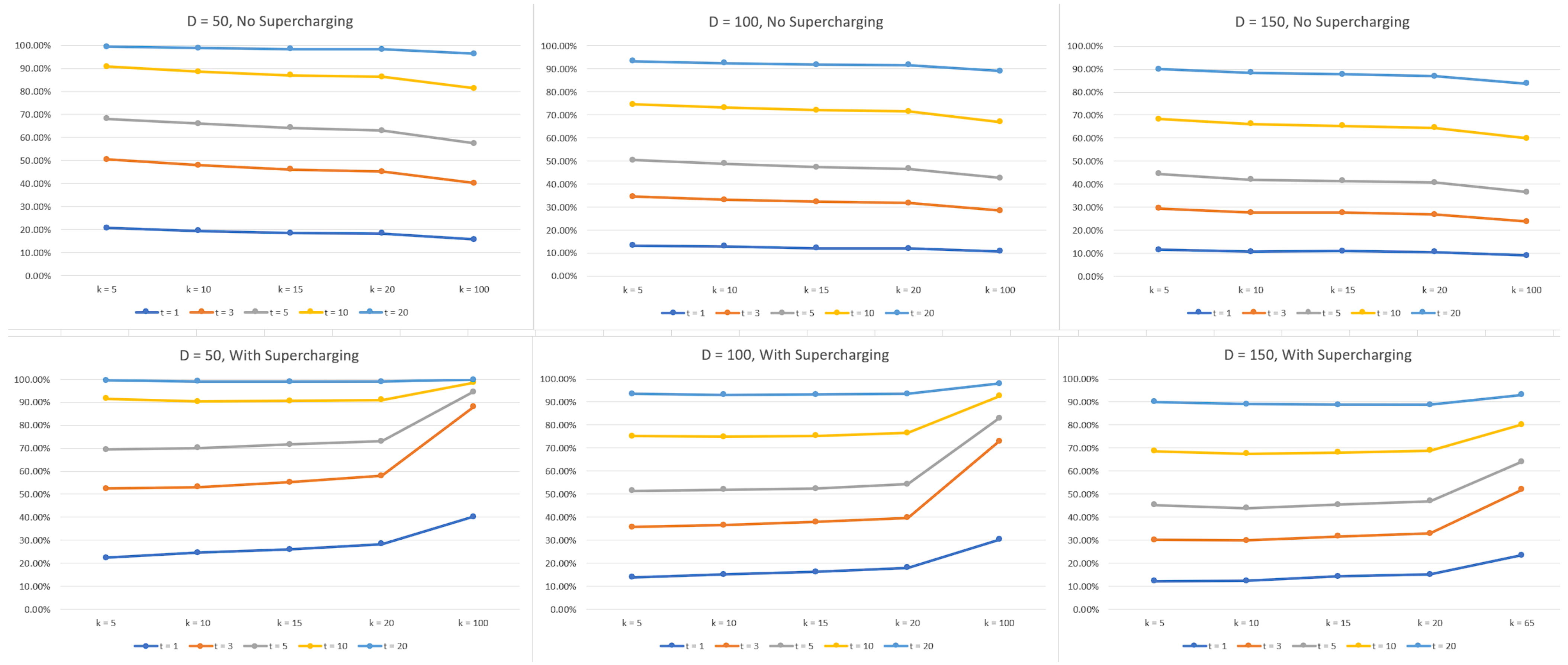

Figure 8: The accuracy of different configurations of RANN measured by seeing how many of the candidates found by RANN were in the true nearest neighbours found by the exact knn implementation. The lines indicate RANN with different T values. The reference set, **A**, is smaller than the previous ones and consists of 100.000 points while the query set, **B**, consists of 500.

7.7 Accuracy

Finally a closer look on how some of the previous variables impact accuracy. The measurements of different configurations of RANN can be seen in figure 8. This was run on a smaller dataset than the previous experiments.
Here it is evident that the most impactful parameter is **T**. This indicates that the random transformation is good at transforming the points of **A** such that the $\mathbf{V}_i$s that are obtained by creating and searching the kd-tree contain good candidates. And that for each new transformation the resulting $\mathbf{V}_i$s contain many different good candidates from the ones already found. It was also the main intuition behind RANN (as mentioned in section 5), that such $\mathbf{V}_i$s could be found. So it is perhaps not surprising **T** is the most impactful parameter.
d also has a significant impact. As **d** grows the accuracy shrinks. **k** behaves similarly if supercharging is not taken into account.
When supercharging is used it neutralizes the negative impact increasing k has on accuracy when k is low. However when k reaches some threshold the accuracy is improved substantially. Since supercharging searches through $\mathbf{k}^2$ candidates for each $\mathbf{x}_i$ it makes sense that the accuracy increases for larger **k**s since a lot more candidates are searched through.
The number of points in the reference set also has an effect. The smaller reference set size of figure 8 benefits accuracy as compared with the results from the previous sections. This is because as the reference set grows from e.g. 100.000 to 200.000 there are 100.000 more points in the reference set. However the kd-tree's height will only be 1 level taller. $\mathbf{V}_i$ will therefore only contain 1 more leaf where the default leaf size is 256. $\mathbf{V}_i$ thus becomes proportionally smaller and smaller since it grows logarithmically as compared to the reference set that grows linearly.
As seen in figure 8 RANN has the potential to be highly accurate even when the dimensionality d and k value is high. This can be achieved with an increased **T** value and enabling supercharging. Here it is best to increase the **T** value since it provides the most amount of accuracy gain for the least amount of runtime impact.

8 Improvements

Here are some improvements to the data-parallel implementation of the algorithm and the testing of the algorithm that can be made:

- The sorting of the native leaves of each query point currently uses radix sort. It would be faster to use counting sort and make a cuda kernel that handles this. This could be done by having the current implementation put into some framework that allows for communication between the futhark generated code/kernels and the counting sort cuda kernel.
- When calculating the pseudorandom orthogonal linear transformations, the transformation **Q** is sequentially. This could be parallelized. The run time gain would however be small since **Q** is used as a function mapped across all points which is already a lot of parallelism.
- When measuring the accuracy it would have been good to also measure the error rate: How much the candidates differ from the true nearest neighbours. This would give an indication of the quality of the candidates that are not found among the true nearest neighbours. The only indication the previous experiments have shown is the fact that the early exit functionality of the bruteforce function seems to make the algorithm faster when T is high (see figure 3 and section 7.3). And that the loosened measuring of the accuracy provides a higher percentage.
- The implementation could be integrated into something like the annfmp package. (A github repository [14] made for the 'Approximate nearest-neighbour fields via massively-parallel propagation-assisted k-d trees' algorithm [12] by Oancea et al.) This would allow a comparision of RANN with other data parallel solutions for the k nearest neighbours problem. Mainly the ones using the properties of images mentioned in the related works section.

9 Conclusion

In this thesis a parallel implementation of the RANN algorithm was implemented and evaluated. Here the various variables of the algorithm were tested to measure their impact on both accuracy and runtime. The parallel implementation of RANN when T = 1 was found to be between 20 to 140 times faster than a parallel implementation of an exact brute force knn algorithm. As the dimensionality the dataset grows, the accuracy becomes worse. This decrease in accuracy is most effectively combated with increasing T.

References

[1] A. Andoni and P. Indyk. Nearest neighbors in high-dimensional spaces. In C. Tóth, J. Goodman, and J. O'Rourke, editors, *Handbook of Discrete and Computational Geometry*, pages 1135 – 1155. CRC Press, 2017.

[2] K. B. Andreasen. Data-parallel coherency sensitive hashing for approximate nearest neighbour fields. 2021. `https://futhark-lang.org/student-projects/kristian-b%C3%B8jer-andreasen-msc-thesis.pdf`.

[3] C. Barnes, E. Shechtman, A. Finkelstein, and D. B. Goldman. PatchMatch: A randomized correspondence algorithm for structural image editing. *ACM Transactions on Graphics (Proc. SIGGRAPH)*, 28(3), Aug. 2009.

[4] M. Datar, N. Immorlica, P. Indyk, and V. S. Mirrokni. Locality-sensitive hashing scheme based on p-stable distributions. In *Proceedings of the Twentieth Annual Symposium on Computational Geometry*, SCG '04, page 253–262, New York, NY, USA, 2004. Association for Computing Machinery.

[5] F. Gieseke, S. Rosca, T. Henriksen, J. Verbesselt, and C. E. Oancea. Massively-parallel change detection for satellite time series data with missing values. In *2020 IEEE 36th International Conference on Data Engineering (ICDE)*, pages 385–396, 2020.

[6] T. Hastie, R. Tibshirani, and J. Friedman. *The elements of statistical learning: data mining, inference and prediction*. Springer, 2 edition, 2009.

[7] K. He and J. Sun. Computing nearest-neighbor fields via propagation-assisted kd-trees. In *2012 IEEE Conference on Computer Vision and Pattern Recognition*, pages 111–118, 2012.

[8] T. Henriksen, S. Hellfritzsch, P. Sadayappan, and C. Oancea. Compiling generalized histograms for gpu. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–14, 2020.

[9] T. Henriksen, F. Thorøe, M. Elsman, and C. Oancea. Incremental flattening for nested data parallelism. In *Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming*, PPoPP '19, page 53–67, New York, NY, USA, 2019. Association for Computing Machinery.

[10] P. W. Jones, A. Osipov, and V. Rokhlin. Randomized approximate nearest neighbors algorithm. *Proceedings of the National Academy of Sciences*, 108:15679 – 15686, 2011.

[11] S. Korman and S. Avidan. Coherency sensitive hashing. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 38:1099–1112, 06 2016.

[12] C. E. Oancea, T. Robroek, and F. Gieseke. Approximate nearest-neighbour fields via massively-parallel propagation-assisted k-d trees. In *2020 IEEE International Conference on Big Data (Big Data)*, pages 5172–5181, 2020.

[13] U. E. Petersen. Optimizing the knn algorithm for gpgpus in futhark. 2020. `https://futhark-lang.org/student-projects/ulrik-bsc-thesis.pdf`.

[14] T. Robroek and C. E. Oancea. annfmp. `https://github.com/diku-dk/annfmp`, 2021.