FACULTY OF SCIENCE
UNIVERSITY OF COPENHAGEN

Accelerating Nearest-Neighbours via KD-Trees on GPUs

Author:

Sarah Maria Hyatt

zfj900@alumni.ku.dk

Supervisor:

Cosmin Eugen Oancea

cosmin.oancea@di.ku.dk

Project Outside Course Scope

Department of Computer Science, University of Copenhagen

May 1, 2020

Contents

1 Introduction

The original direction of this project was inspired by (He & Sun, 2012) who propose a solution for efficiently computing approximate nearest-neighbours (ANN) computations using propagation-assisted k-d trees. The performance bottleneck is the search for K-nearest neighbours (KNN) to improve accuracy, and while various other methods such as the k-means tree and the bbd-tree improve accuracy over the k-d tree, they spend extra effort in building the trees, e.g. the k-means tree uses 10-20 more time (He & Sun, 2012). However, we found that in order to implement such algorithms we would still need all the components of the exact K-Nearest Neighbour (KNN) computations by k-d trees as we would for the approximate solution, thus, for this project, it was commonly agreed to concentrate on the exact KNN computation solution.

KNN is the core of many applications in various fields such as Computer Vision and Machine Learning, for instance, finding similarities between two images as the original direction proposed.

The naive approach of computing KNN is using brute force, and while brute force has been empirically observed to be efficient than the k-d tree traversal when dealing with a small number of reference points, such as in the hundreds, it is infeasible time-wise for larger datasets due to its complexity $O(n \times m)$.

The idea of KNN by k-d trees is to construct a spatial data-structure, which is a binary tree where each internal node semantically corresponds to a subset of the reference points that can be reached through them, such that each query does not have to be compared with each reference point, as it would in the naive approach, keeping the number of comparisons asymptotically cheaper than the O(n*m). In our approach, the leaves contain a set of points and we apply brute force at the leaf level due to because it has been empirically observed that brute force is more efficient than k-d tree traversal when the number of reference points is small such as in the hundred for GPU computation.

In order to compute the KNN for a query, the tree is traversed in which each visit to each node determines whether it is necessary to go into a recursion to visit the next node or to skip it. Typically we reason about whether to visit the next node by computing the distance to the worst nearest-neighbour with the distance from the query to the box that bounds that region represented by that internal node. The approach typically compares the distance between the query to the current median against the distance to the worst nearest-neighbour. However, we improve this test to be more accurate by considering the upper and lower bounds of all dimensions corresponding to that.

In principal this possibly reduces the asymptotic work to a factor of $O(n\ lg\ n)$, but in practice, the efficiency of the solution depends fundamentally on the dimensionality and the size of K. While this offers a cheaper traversal it still suffers from the curse of high dimensionality. Thus, the trade-off is accuracy for lower dimensionality and reduced algorithmic complexity. In practice there are two main problems that may affect the performance of the solution in asymptotic terms (1) dimensionality, if it is too high, then you might have to compare against most of the reference points (2) and similarly K if K is too high and the distance to the worst neighbour becomes big then similarly you are going to visit more leaves.

One of the advantages of the k-d tree, compared to various other similar structures, such as octal trees - is that is can in principal conduct NN computations over high dimensionality spaces. Octal trees are limited to three dimensions.

To address the curse of high dimensionality, we present a more accurate test that, in several cases, allows to efficiently process queries of higher dimensionality than the classical approach. While the typical solution computes the distance between the query to the current median against the distance to the worst nearest-neighbour, this solution considers the upper and lower bounds of all dimensions. Which, in return, improves temporal locality by sorting the leaves by accessing the same reference points in the same order. See more in section **ADD REFERENCE**.

Additional optimisation, such as improving temporal locality, have been done by sorting the queries such that they access the same reference points in the same order whenever we do brute force of the leaves.

Summarise the conclusions at a higher level.

for this dimensionality we obtain this amazing speedups and for small k's we increase the k's for a small dimensionality and this is roughly how the speed ups is xxx. (OBS the summary should back my previous defined claims)

The summarise the results : here we want to validate the points that we have just made : does sorting improve things or not? and more insight like - give people a magical pill so they can understand things - even if it is not that precise. for example, if i'm having a uniformly distributed dataset of 16 points then if i'm choosing k it's reasonable to compute this knn like by 5, over 5 I see the performance reduces drastically.

- what are the downsides of the solution - significant overhead and uses contains much more data - so that is why experiments are important to show which ds and ks run best. if you visit most of the leaves then you are going to be much much slower, due to the significant overhead. That's why experimental evaluation is important because it shows on what d's and k's it is beneficial - does it make sense to run the k-d tree this way.

1.1 Acknowledgement

2 Background

KNN is used find the K nearest neighbours in some dimensionality D, and it is widely applied due to its simplicity and excellent empirical performance, it has no training phase, it works well handling multi-dimensional datasets. We look at a D dimensional space, where we have a set of reference points of size n, and a set of queries of size m, and we aim to compute the nearest neighbours according to a measured distance for each query.

The naive approach is using brute force, in which each reference point is compared with each query, meaning the complexity of the search will be $O(m \times n)$. The brute force approach is explained further in section 3. Although brute force is a simple solution, there exists several alternative techniques for computing KNN. One commonly applied method is using a generalisation of the binary tree, known as the k-d tree.

The k-d tree consists of a root, internal nodes and leaves. The reference points are fully partitioned, such that all the internal nodes on each level represent all reference points. Two nodes on the same level do not intersect in terms of the reference points they represent. The leaves do not necessarily contain one reference point each, they may contain several reference points, and in our approach we apply brute force on the leaves.

Using a k-d tree improves search performance, in principal the average work complexity is $O(m \ \log \ n)$ (Friedman, Bentley, & Finkel, 1977).

2.1 Pseudo-code for the Computing KNN using k-d Trees

Algorithms 1, 2 and 3 below represent an implementation of the k-d tree construction and the tree traversal as high-level pseudo-code. Both Algorithms 2 and 3 use divide and conquer recursion. Algorithm 1 is the `main` function in which a k-d tree is created of the reference points, and iterations through the queries call TRAVERSE to find the best neighbours for query.

Algorithm 1 Main

```
procedure MAIN(k, queries, refs)
    $tree \leftarrow$ BUILDTREE($refs$, $refs.idxs$, 0, 0, NONE)
    $neighbours \leftarrow$ NONE
foreach query in queries do:
    $neighbours \leftarrow$ TRAVERSE($tree$, $query$, $0$, $neighbours$, $k$)
return neighbours
```

2.2 Construction the Tree

Algorithm 2 represents the construction of the k-d tree. The tree construction is divided into two stages; one the processes the internal nodes (lines 3-18) and one the handles the leaves (lines 19-20). The condition is implemented by comparing the current level with the tree height, and if the two are equal, it means that we have reached the leaves.

For a given node, at any given node level the reference points are partitioned by finding the dimension with the widest spread across the points represented by that node, and choosing the median value of that dimension. The left child of that node will represent the first half of the reference points in sorted order, and the right child of that node will represent the rest. The nodes, therefore, contain a dimension and a median value. Line 4 uses the function GetSplitDimension to find the dimension of widest spread, lines 5-6 sort the indices and patches w.r.t the chosen dimension and lines 8-9 pick out the median index and value. Once done, the median and the dimension are ready to be stored, which essentially creates that particular node of the tree. Lines 12-18 compute the indices of the left and right children of the current node for the next iteration, and recursively call the BuildTree function again, in order to complete the creation of nodes in the tree.

Last, lines 20-21 sorts the leaves w.r.t the indices that were sorted and partitioned throughout the recursive node creation.

Algorithm 2 Building the Tree

```
procedure BuildTree(refs, indices, depth, index, tree)

    if IsLeaf(depth) then
        $dim \leftarrow \text{GetSplitDimension}(refs)$
        $indices \leftarrow \text{SortByDim}(indices, dim)$
        $points \leftarrow refs[indices]$

        $medianIdx \leftarrow \text{Length}(indices)/2$
        $median \leftarrow points[medianIdx]$

        $tree[index] \leftarrow (median, dim)$
        $depth \leftarrow depth + 1$

        $leftIdx \leftarrow \text{GetLeftChild}(index)$
        $rightIdx \leftarrow \text{GetRightChild}(index)$

        $\text{BuildTree}(points[: medianIdx], indices[: medianIdx], depth, leftIdx, tree)$
        $\text{BuildTree}(points[medianIdx :], indices[medianIdx :], depth, rightIdx, tree)$
    else
        $leaves[index].refs \leftarrow refs$
        $leaves[index].idxs \leftarrow indices$
```

2.3 Tree Traversal

The traversal is presented in Algorithm 3 below. Each query is going to traverse the k-d tree, and at each internal node in the tree, it is going to decide whether it has to visit a certain child by making a conservative test to whether the KNN can possibly be improved in the regions that that node represents. If a leaf is reached, a brute force is performed w.r.t the query and all the reference points represented in that leaf, this is shown in line 2-4. The rest of the code corresponds to the recursive traversal, lines 6-21.

Algorithm 3 The Tree Traversal

```
 1: procedure TRAVERSE(tree, query, nodeIndex, neighbours, k)
 2:     if IsLeaf($nodeIndex$) then
 3:         $neighbours \leftarrow$ BRUTEFORCE($tree, query, nodeIndex, neighbours, k$)
 4:         return neighbours
 5:
 6:     $dimens \leftarrow tree[nodeIndex].dims$
 7:     $median \leftarrow tree[nodeIndex].meds$
 8:     $queryV \leftarrow query[dimens]$
 9:
10:     if queryV $<=$ median then
11:         $first \leftarrow$ GOLEFT($nodeIndex$)
12:         $second \leftarrow$ GORIGHT($nodeIndex$)
13:     else
14:         $first \leftarrow$ GORIGHT($nodeIndex$)
15:         $second \leftarrow$ GOLEFT($nodeIndex$)
16:
17:     $neighbours \leftarrow$ TRAVERSE($tree$, $query$, $first$, $neighbours$, $k$)
18:     $worstNeighbour \leftarrow$ LAST OF $neighbours$
19:
20:     if (median - queryV) $<$ worstNeighbour then
21:         $neighbours \leftarrow$ TRAVERSE($tree$, $query$, $second$, $neighbours$, $k$)
22:  return neighbours
```

The code denotes by first, the child of the current node that is on the same side as median w.r.t the query, similarly second denotes the child of the current node that is on the opposite side of the median. This can be observed between the computations on lines 10-15.

Ret så det lyder bedre

If the query value of the current dimension is less than the median of the current node, then first is the left child, and right otherwise.

If we reach this node, then since the query is on the same side as the current median as the node denoted first, then one recursive call is always going to be made, because it means that we can always reach a leaf that potentially has a neighbour better than the worst one found so far.

The recursive call that follows the second node is not necessarily made, since it is guarded by a conservative condition that tests whether the distance from the query to the current median is better than the worst found neighbour so far. If this condition does not hold, it is safe not to visit the left child, because it is ensured that you cannot find any better neighbours than the ones we currently have, which is visible in lines 20-21.

hertil

The goal is to implement the solution using Futhark since Futhark is a pure functional data-parallel array language, that works efficiently on the GPU. It is a great advantage to write fully parallel code in a high-level functional language. However, since Futhark utilises regular parallelism it does not support recursion. With this in mind, the pseudo-code, demonstrated above, will need to be rewritten without divide-and-conquer recursion, which is demonstrated in the sections k-d Tree Construction and Tree Traversal.

2.4 Related Work

2.4.1 PatchMatch

PatchMatch uses a randomised algorithm for quickly finding approximate nearest neighbour fields (ANNF) between image patches.

While previous solutions use KD-Trees with dimensionality reduction, PatchMatch instead searches through a 2D space of possible patch offset. The initial step is choosing random patches followed by 4-5 iterations containing two processes: (1) propagation applying a statistic that can be used to examine the relation between two signals or data sets, known as coherence, (2) random search in the concentric neighbourhood to seek better matches by multiple scales of random offsets. The argument is that one random choice when assigning a patch is non-optimal, however applying a large field of random assignments is likely a good choice because it populates a larger domain.

The gains are particularly performance enabling interactive image editing and sparse memory usage, resulting in runtime O(mM log M) and memory usage O(M), where m is the number of pixels and M the number of patches. The downside is that PatchMatch depends on similar images because it searches in nearby regions in few iterations, which in turn also affects its accuracy.

2.4.2 Coherency Sensitive Hashing

Coherency Sensitive Hashing (CSH) is inspired by the techniques of Locality Sensitive Hashing (LSH) and PatchMatch. CSH uses a likewise hashing scheme of LSH and is built on two overall stages, indexing and searching.

The indexing stage applies 2D Walsh-Hadamard kernels in which the projections of each patch in image A and B are initially computed onto the WH kernels. Subsequently the hash tables are created by the following. First, it takes a random line defined by a patch and divides it into bins of a constant width while shifting the division by a random offset. Second, the patch is projected onto the most significant 2D Walsh-Hadamard kernels. Last, a hash value is assigned, being the index of the bin it falls into.

Applying hash tables has the benefit that similar patches are likely to be hashed into the same entry.

The searching stage starts by initialising an arbitrary candidate map of ANNF. This is followed by iterations through each patch in image A where: (1) the nearest neighbour candidates of image B are found using the current ANNF and the current hash table, (2) the current ANNF mapping is updated with approximate distances.

Selecting candidates follows the appearance-based techniques of LSH and as well as the coherence-based techniques of PatchMatch. Choosing among the candidates is done in an approximate manner where the WH kernels rejection scheme for pattern matching is applied beneficially.

CSH has ‘46%‘ better performance than PatchMatch and a higher level of accuracy with error rates that are ‘22%‘ lower in comparison

3 Implementation

3.1 Brute Force

The pseudo-code for the brute force implementation is found in Listing 1. It is constructed with an outer parallel map on line 2, in which the body of the map is sequential, consisting of a loop nest on lines 4 and 9.

The first for-loop iterates through the reference points, while computing the distance between the query and each reference point, in line 6. Lines 8-18 choose the KNN for each reference point by iterating through a list of size K, which contains the current nearest neighbours. If a better candidate is found, it will be interchanged to its correct position in the list, and the rest of the list will be forwarded.

The advantages of implementing the body of the map sequentially, is that the number of queries is large enough to saturate all hardware threads on the GPU, see more in section 4 for a specialisation of the hardware used. Meaning that the Futhark compiler will spawn all active threads, such that each thread will compute the KNN for a query given to that thread. Figure 15 in section 4 shows the performance of brute force with various size of dimensions D and Ks.

```
 1 entry nnk [m][n][d] (qs:[m][d]real) (refs:[n][d]real) : [m][k](int,real) =
 2     map (\q ->
 3         let  nn = replicate k (-1i32, real_inf)
 4         in loop nn for j < n do
 5             let ref = refs[j]
 6             let dist = seq_euclidean q ref
 7             let r_idx = j in
 8             let (_, _, nn') =
 9                 loop (dist, r_idx, nn) for i < k do
10                     let cur_nn = nn[i].1  in
11                     if dist <= cur_nn then
12                         let tmp_ind = nn[i].0
13                         let nn[i] = (r_idx, dist)
14                         let r_idx = tmp_ind
15                         let dist  = cur_nn
16                         in  (dist, r_idx, nn)
17                     else    (dist, r_idx, nn)
18             in  nn'
19     ) qs
```

Listing 1: Futhark implementation of the Brute Force.

While this pseudo-code shows the full brute force of all queries and reference points, it is conceptually similar to the brute force performed on each leaf-level in the k-d tree implementation.

3.1.1 Difficulties and Shortcomings

The distance computed on line 6 in Listing 1 was initially implemented as a map-reduce composition while still called inside the first loop on line 4. The result of this was poor performance because the Futhark compiler would see that there was some inner parallelism, which it would exploit, even though it is inefficient. It was, therefore, more efficient to compute the distance sequentially, as described previously.

3.2 k-d Tree Construction

Listing 2 represents Futhark-like pseudo-code to demonstrate the overall steps and parallel constructs that are applied in constructing the k-d tree. Note that the structure is not equivalent to the final code, due to some shortcoming of Futhark that are discussed in subsection 3.2.1.

Recall the tree construction Algorithm 2 from section 2 using divide-and-conquer recursion, which is not supported by Futhark. The way we account for the divide-and-conquer recursion in Listing 2, is by iterating through a sequential loop from level 0 to $h+1$, in line 8, and computing for each level the number of nodes per level and number of points per node per level, lines 9-10, in order to unflatten the arrays containing the reference point data into sections that fit the current number of nodes for each level. This is illustrated in figure 1. Once the reference points are sorted within the bounds of each node, the arrays with reference data will be re-flattened, ready for the next iteration of the for-loop.

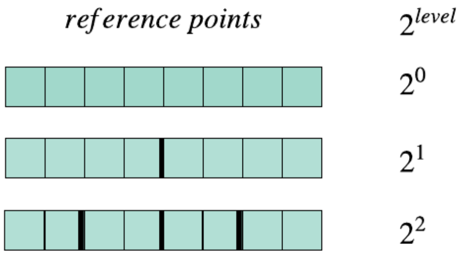


Fig. 1: A demonstration of dividing the reference points at each node level.

The completed tree will consist of the reorganised array of reference points, the original indices of the reference points and two arrays containing the dimension with the widest spread and the median value, that together represent the splitting values of each node. In our optimised solution, the tree also consists of two 2D arrays containing the upper and lower bounds for each dimension at each current node. This idea is elaborated further in section 3.3.

The dimension with the widest spread is computed on lines 16-20, and subsequently, the values in that dimension are sorted, and the reference points and indices are gathered accordingly, on lines 21-24. The median value is picked out on line 25, and both the median and dimension are scattered into the tree arrays containing all medians and dimensions in lines 34-35. Since the upper and lower bounds are computed when finding the dimension with the widest spread, they too are scattered into the tree arrays containing all upper and lower bounds, in lines 36-37.

```
entry buildTree [m][d] (points : [m][d]f32) (h: i32) =
    let num_pads =     (...)   -- computing the padding needed
    let padding  = map (...)   -- creating the padding array
    let reference = points ++ padding

    let (ref_idxs, reference, median_vals, median_dims, lower_bounds, upper_bounds) =
            (...) -- initalising the loop variables
        for level < (h+1) do
            let num_nodes_pr_lvl = 1 << level
            let num_points_pr_node_pr_lvl = m // num_nodes_pr_lvl
            let reference = unflatten num_nodes_pr_lvl num_points_pr_node_pr_lvl reference
            let ref_inds  = unflatten num_nodes_pr_lvl num_points_pr_node_pr_lvl ref_idxs

            let (indices', reference', node_info', lower, upper) = unzip5 <|
                map3 (\i node_arr inds ->
                        let dim_arrs = transpose node_arr   |> intrinsics.opaque
                        let smallest = map o reduce         |> intrinsics.opaque
                        let largest  = map o reduce         -- max numbers for each dim
                        let differences = map         (...) -- computing differences
                        let (dim,_)     = reduce_comm (...) -- dimension w/ widest spread
                        let extract_dim = map         (...) -- extract dimension values
                        let d_sort_idxs = extract_dim |> sort |> map (.0)
                        let indices     = gather d_sort_idxs inds
                        let node_arrp   = gather2D d_sort_idxs node_arr
                        let median      = node_arrp[num_points_per_node_per_lvl // 2, dim]
                        let node_info   = (median, dim)
                        in (indices, node_arrp, node_info, smallest, largest)
                    ) (iota num_nodes_per_lvl) reference ref_inds

            let (medians, dims) = unzip node_info'
            let inds = map (...)   -- computing indices for scatter
            in (flatten ref_idxs',
                flatten reference',
                scatter median_vals inds medians,
                scatter median_dims inds dims,
                scatter2D lower_bounds inds lower,
                scatter2D upper_bounds inds upper)

    in (ref_idxs, reference, median_vals, median_dims, lower_bounds, upper_bounds)
```

Listing 2: Futhark implementation of the tree creation.

One essential optimisation of the k-d tree construction is the use of padding. Since the datasets might be of sizes that are not dividable with the number of leaves, then padding the dataset to match the number of leaves would mean the parallel dimension will have the same size overall the elements of the map, allowing regular parallelism and flattening which is much more efficient.

Arguably, the proposed solution uses at most one padded element per leaf, accounting to less than 1% overhead. Assuming that we choose a height that fits that dataset size, such that the number of points per leaf is in the hundreds.

The upper bound of the number of pads should not exceed the number of leaves, thus, we want to see how many pads that amount per leaf. In Equation 1 and 2 we denote n to be the total number of reference points, 2^{h+1} computes the total number of leaves, where h is the height of the tree excluding the leaves. Thus, $\left\lceil \frac{n}{2^{h+1}} \right\rceil$ denotes the number of points per leaf. In Equation 1 we multiply the number of leaves with the number of points per leaf, we get the number of elements that are dividable with the total number of leaves, and by subtracting n, we will have the number of pads needed. In Equation 2 we divide with the number of leaves on each side, resulting in less than or equal to 1 pad per leaf.

$$\left\lceil \frac{n}{2^{h+1}} \right\rceil \times 2^{h+1} - n \leq 2^{h+1} \tag{1}$$

$$\left\lceil \frac{n}{2^{h+1}} \right\rceil - \frac{n}{2^{h+1}} \leq 1 \tag{2}$$

3.2.1 Difficulties and Shortcomings

In the construct above in Listing 2, the original solution was to use a merge sort on line 19. However, although it was safe for Futhark to perform a loop distribution and loop interchange between the outer map inside both for loops, it did not realise, in which case the interchange was not performed. Thus, causing poor performance for the tree construction. The solution was to (1) use batched merge sort, which operates on 2D arrays rather than 1D arrays, (2) distribute the map from line 12, across the body of the map function, such that the sorting is outside the map.

Another shortcoming shows by the use of `intrinsics.opaque` at lines 14-15 in Listing 2, where the purpose is to prevent the Futhark compiler from fusing the two map-reduce compositions on lines 15-16. The reasoning behind this is that both map-reduce compositions call on the same size array, namely m.

Since the Futhark compiler uses moderate flattening, a map-reduce composition will execute in parallel. However, in a map-reduce composition inside a map, as we have in lines 13 and 15-16, the inner map-reduce composition will be sequentialised, and thus resulting in inefficient code. By adding `intrinsics.opaque` and preventing the fusion, the compiler will create two reduce inside a map, which in return will exploit all parallelism.

3.3 Tree Traversal

In section 2 the tree traversal Algorithm 2, is created using recursion, and as in section 3.2 we must re-write the traversal without recursion. The proposed solution uses a stack, and the stack is represented as an integer, since the height of the tree never exceeds 32.

The traversal has two stages overall stages: (1) the first traversal finding the unique leaf in which the query naturally belongs, (2) backtracking the tree to find additional or better KNNs.

3.3.1 The First Traversal

As described in section 2, first denotes the child of the current node that is on the same side as median w.r.t the query, similarly second denotes the child of the current node that is on the opposite side of the median. Figure 2 demonstrates how the tree is traversed to the leaf in which the query belongs, note that the leaf marked with red is denoted first and its sibling leaf is denoted second.

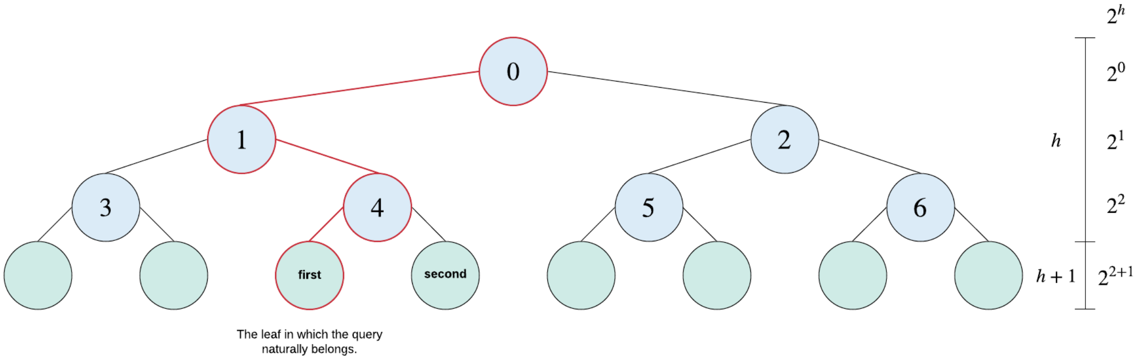


Fig. 2: The first traversal finding the leaf in which the query naturally belongs.

3.3.2 The Traversal for Additional Leaves

In this demonstration, we look at a fictive traversal in which we always check the second node. The validation technique as to whether to check the second node or not, is elaborated in section 3.3.3.

Figure 3 is a continuation of figure 2, in which the initial leaf is reached and the stack is popped to see if the second node has been visited or not. Since it had not been visited, the traversal also visits the second node.

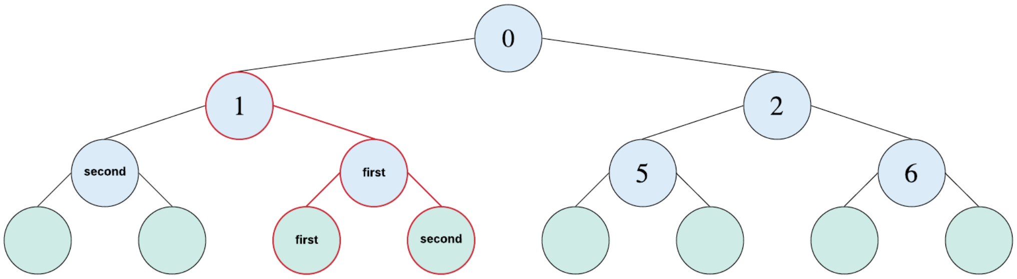


Fig. 3: Visiting the second node of the initial leaf.

In figure 4 the traversal looks to the parents second child, and since that has not yet been visited, it traverses to the first leaf of the parents second child.

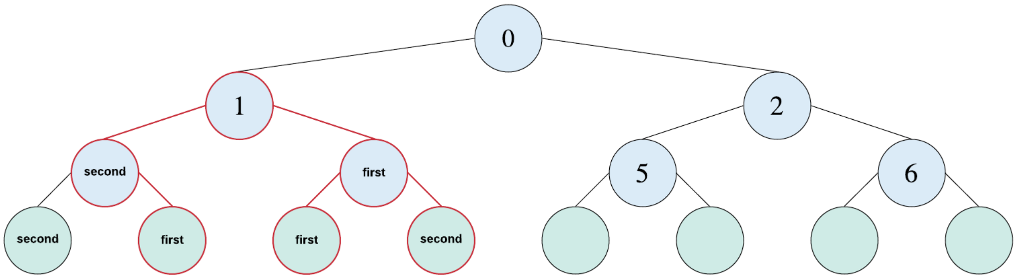


Fig. 4: Visiting first node of the parents second child.

Pretending that we still have not found the KNN, the traversal will continue into the second leaf of the parents second child, as in figure 5. If the KNN is still not found, the traversal will continue to the second child of the root until the whole tree has been traversed, or the KNN is found.

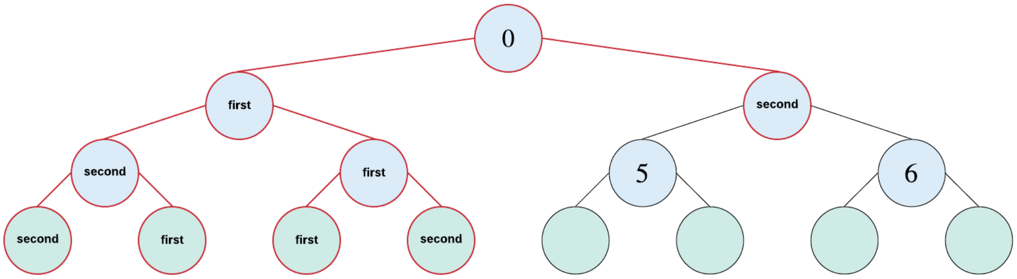


Fig. 5

3.3.3 Validating Whether to Look at the `second`

We propose an optimised solution for checking whether to visit the second node or not. *wknn* denotes the worst found neighbour so far, R denotes our reference points, Q denotes our queries, *lb* and *ub* denote the arrays containing the upper and lower bounds and M denotes the array containing the median values for each node.

$$\forall i.[lb_i, ub_i].\ \forall q \in Q_m.\ \forall p \in R_n.\ \forall m \in M_{nn}.$$

The original solution determines whether to visit the second node by checking if the distance from the query to the current median is smaller than the worst found neighbour, as in Equation 3. However, the solution looks only at one dimension when checking whether to visit or not, meaning that it is likely that another dimensionality could contain information that would otherwise have prevented the second visit. Especially if the number of dimensions is, because then a lot of values will be overlooked.

$$\mid q_i - m_i \mid >\ wknn \tag{3}$$

The naive approach would be to compute all distances, as in Equation 4. Although this has a higher level of accuracy compared to the original solution, it also inefficient.

$$\sqrt{\sum_i (p_i - q_i)^2}\ \geq\ wknn \tag{4}$$

Instead, we propose a solution that achieves a higher level of accuracy compared to the original solution, while still running efficiently. The idea is that the upper and lower bounds for each dimension is saved, for each node, and then used to compute the distance between the query to the upper and lower bounds, respectively. Thus, instead of checking one dimension, we check all dimensions, but only against the upper and lower bounds. This is formulated in Equation 5 and it corresponds to lines 11-26 in Listing 3.

$$\sqrt{\sum_i (p_i - q_i)^2}\ \geq\ \sqrt{\sum_i \begin{matrix} (q_i - lb_i)^2 \text{ , if } & q_i \leq lb_i \\ (q_i - ub_i)^2 \text{, if } & q_i \geq ub_i \\ 0 \quad , & \text{otherwise} \end{matrix}}\ \geq\ wknn \tag{5}$$

Section 4.2 demonstrates the increased performance by using this solution. On a dataset of size roughly 4 mio., with dimensionality D=12 and K=5, we obtain a speed-up of 5.6. The reason for this is elaborated further in Section 4.2.1, which demonstrates how the checking all the upper and lower bound on all dimensions reduces the number of visits to additional node, due to its increased accuracy. Thus, reducing the steps made in the traversal and improving the performance.

```
let (parent_rec, stack, count, rec_node) =
    loop (node_index, stack, count, rec_node) =
         (last_leaf, stack, height, -1)
          while (node_index != 0) && (rec_node < 0) do
              let parent = getParent node_index
              let second = node_index + addToSecond node_index in

              if isVisited stack count
              then (parent, stack, count-1, -1)
              else
                let ack = loop ack = 0.0f32
                    for i < d do
                        let cur_q = query[i]
                        let lower = lower_bounds[second,i]
                        let upper = upper_bounds[second,i] in

                        if cur_q <= lower then
                            let res = (cur_q-lower)*(cur_q-lower)
                            in (ack + res)
                        else if cur_q >= upper then
                            let res = (cur_q-upper)*(cur_q-upper)
                            in (ack + res)
                        else (ack + 0.0)

                let to_visit = (f32.sqrt ack) < wknn in
                if !to_visit
                then (parent, stack, count-1, -1)
                else
                  let second = node_index + addToSecond node_index
                  let stack  = setVisited stack count in
                  (parent, stack, count, second)
```

Listing 3: A part of the tree traversal demonstrating checking upper and lower bound to determine visits.

3.4 The Full Implementation

At each step you reason about tradeoffs/alternative design choices and justify why you did it in the way you did it (advantages/shortcommings, and why your approach is reasonable). Of course mostly related to performance.

Whenever possible, support your reasoning with (as in point to) experimental evaluation results (next).

4 Experimental Evaluation

This section describes and demonstrates the performance of each optimisation described in sections 3.2 and 3.3, as well as the overall performance gains achieved of the k-d tree solution compared to the brute force from section 3.

Each solution has been validated against the result of the brute force solution, using Futhark benching.

4.1 Optimisation using Sorting over Partition

The following subsection demonstrates the results for solutions using partition and sorting, respectively, described in section 3.2.

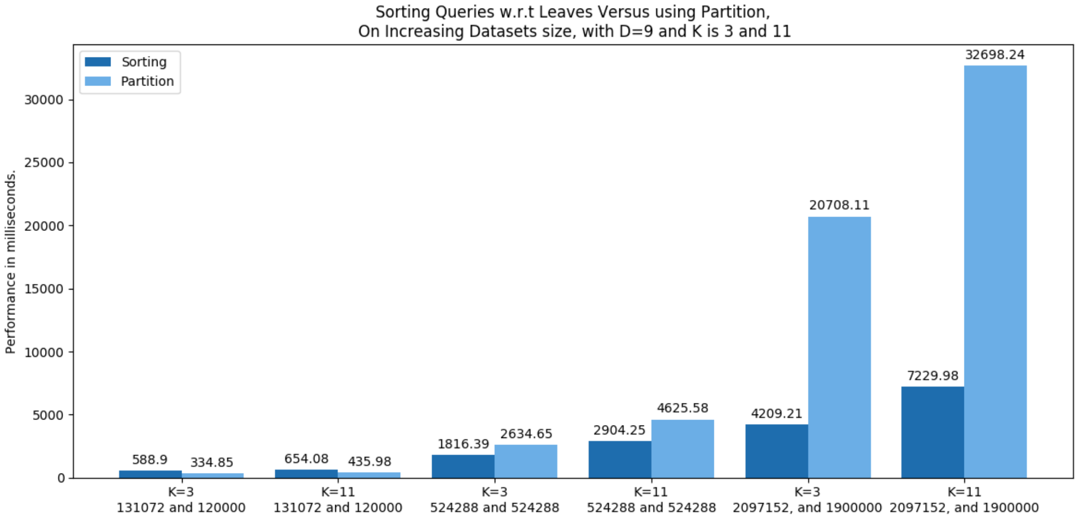


Fig. 6: Testing for D=9 with datasets of sizes, 2097152 and 1900000, 524288 and 524288, and 131072 and 120000.

In figures 6 and 7, we compare the performances between sorting and partition. It shows that sorting works faster for datasets larger than 524288, with higher dimensionality such as D=9, or lower dimensionality such as D=4 with a high K, such as K=11.

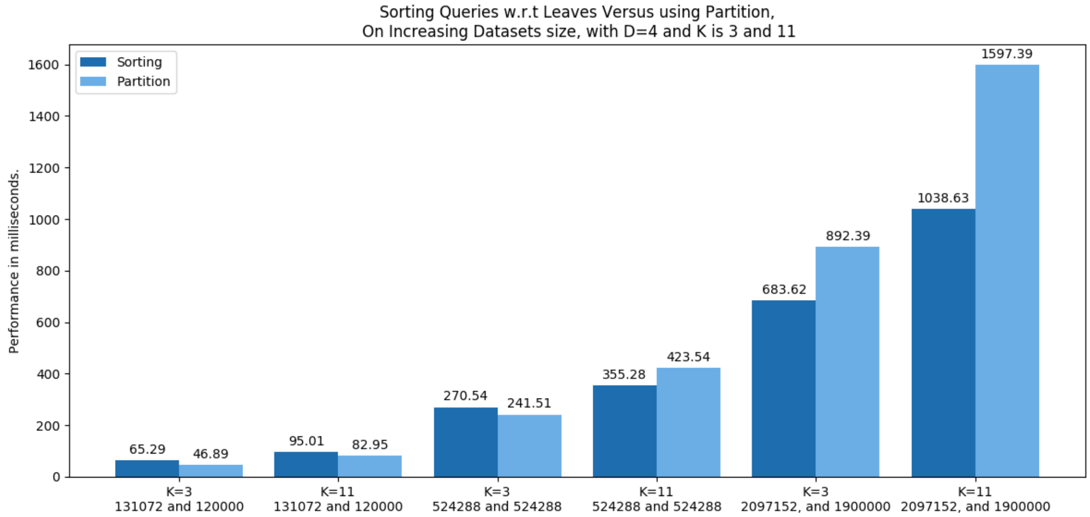


Fig. 7: Testing for D=4 with datasets of sizes, 2097152 and 1900000, 524288 and 524288, and 131072 and 120000.

However, by looking at the speed-ups presented in table 1, we see that K does not have any remarkable effect, since the speed-up is higher for K=3 on the dataset of size 524288, and the speed-up is higher for K=11 on the dataset of size 2097152. We also see that increasing D has a positive effect on the speed-ups. The datasets of sizes 2097152 and 1900000 have a speed-ups that are 3.61 and 3.99 times higher when using a dimension of size 9 over 4.

D	K=3	K=11	K=3	K=11	K=3	K=11
4	0.71	0.87	0.89	1.19	1.3	1.53
9	0.56	0.66	1.45	1.59	4.91	4.52

Table 1: Speed-ups gained by increasing the dataset sizes, for sorting against partition.

In conclusion, we see that sorting works incrementally better for increasing sizes in D and datasets.

4.2 Optimising Tree Traversal with Full Dimensionality Checking

The following demonstrates the results for the optimisation described in section 3.3.

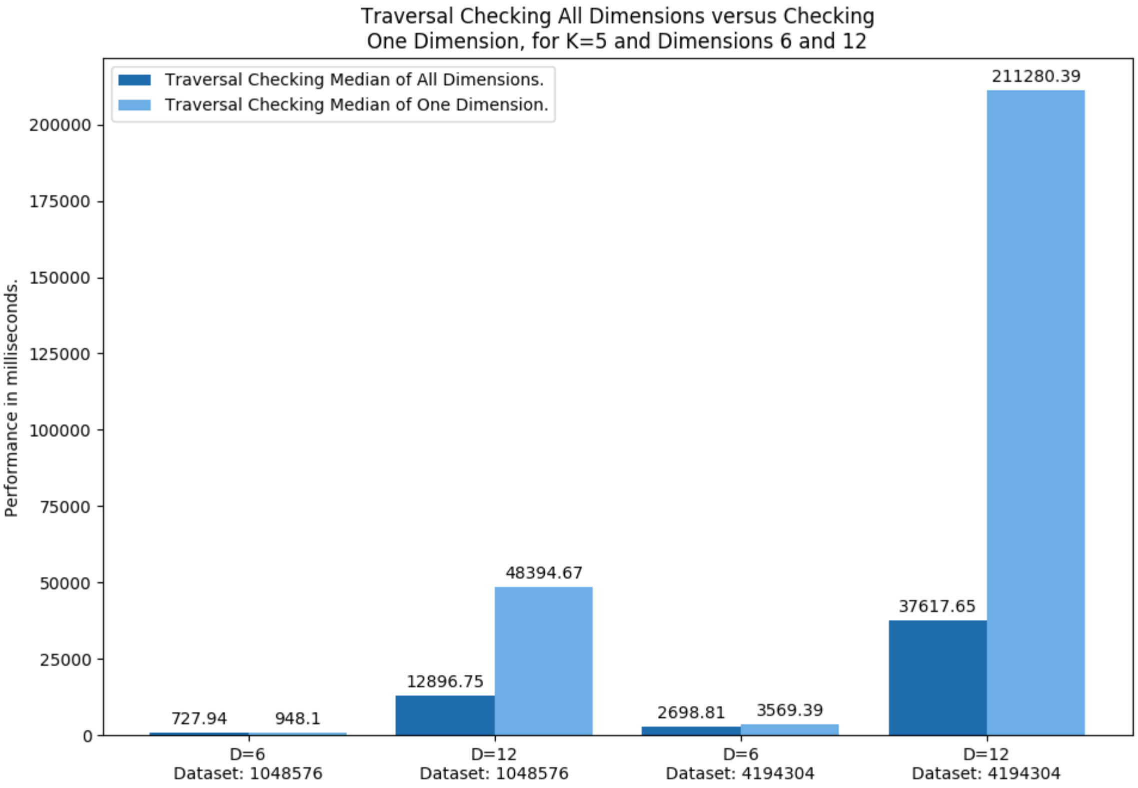


Fig. 8: Traversal comparing results w.r.t varying sizes for D on a smaller and larger dataset.

Figure 8 shows that increasing the size of D as well as the dataset sizes will give a better performance for the optimised solution that checks at all dimensions. The test uses a static size K=5 and tests on D=6 and D=12 for both small and large datasets, where the light blue is the traversal checking one dimension and the dark blue is the traversal checking all dimensions. In comparison, the speed-ups for D=12 is 2.45 and 4.3 times faster than D=6, respectively.

Datasize	*K=1*	*K=5*	*K=17*
Small	2.47	4.47	4.3
Large	3.14	3.37	2.65

Table 2: Speed-ups gained by increasing the size of K for both small and large datasets.

Ret med friske øjne

The next illustration, figure 9, we analyse whether increasing or decreasing K amounts to a speed-up. Testing against a D=12 for both small and large datasets the speed-ups visible in table 2 show that the performance in both cases are best with a K=5, however, it shows no correlation between a small or large K amounting

and more so for larger datasets–which mores visible in table 2, which shows the actual speed-ups from the graph in figure 9.

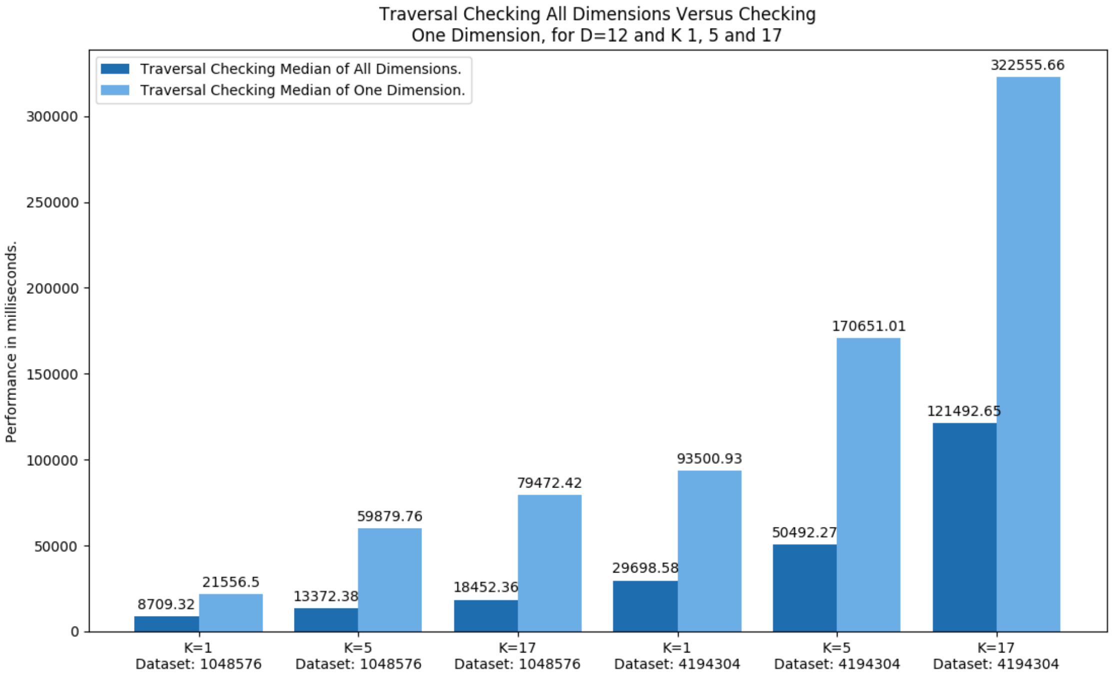


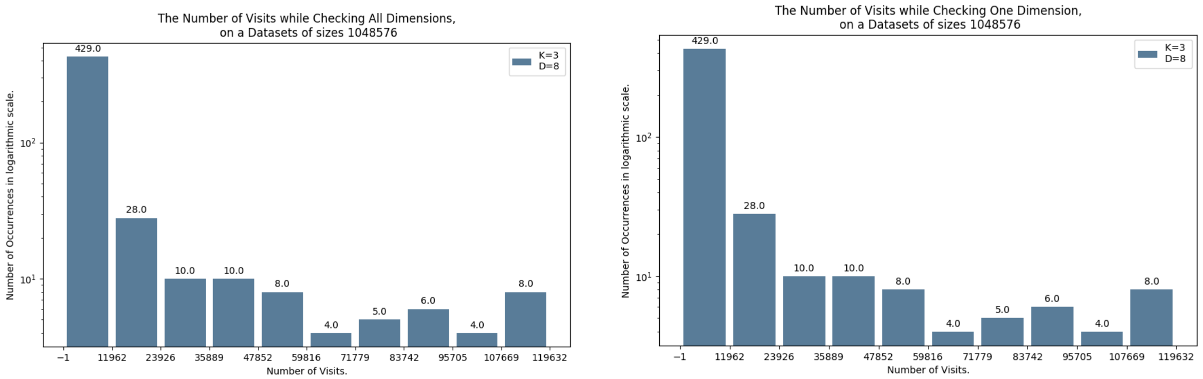
Fig. 9: Traversal comparing results w.r.t varying sizes for K on a smaller and larger dataset.

4.2.1 Demonstrating a Comparison of the Number of Leaf Visits

The following histograms demonstrate the number of visits to leaves performed when searching for the exact KNN. The size of the visits-array is initialised to the total number of leaves, after which each search iteration will store the number of new visits in the array. The small example below shows that the program would finish finding the exact KNN after six iterations of searches through leaves. The three -1's indicating that no further leaves are visited.

$$[100, 99, 85, 62, 33, 18, -1, -1, -1]$$

Thus, in the histograms below, a large number of -1's to the left will indicate a few visits, resulting in better performance, while a large number to the right will indicate many visits and poor performance.

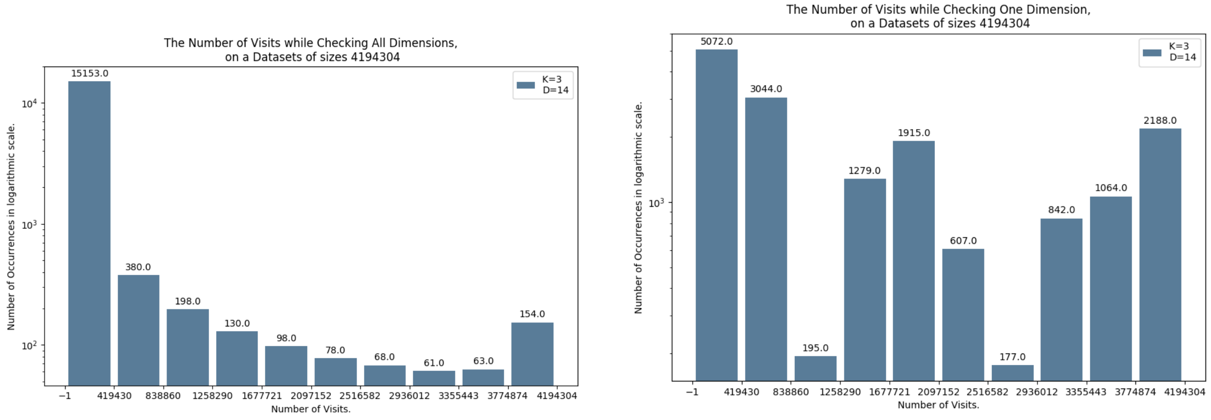


(a) Checking all dimensions.

(b) Checking one dimension.

Fig. 10: Figure a and b show the same result for K=3 and D=8 on both traversal solutions, on datasets of sizes 1048576.

As figures, 10a and 10b demonstrate, for a small K of size 3 and a relatively small D of size 8, we see that the number of visits to leaves in both solutions is equal. However, figures 11a and 11b show increasing D to 16 drastically changes the outcome, such that the method of checking all dimensions excels.

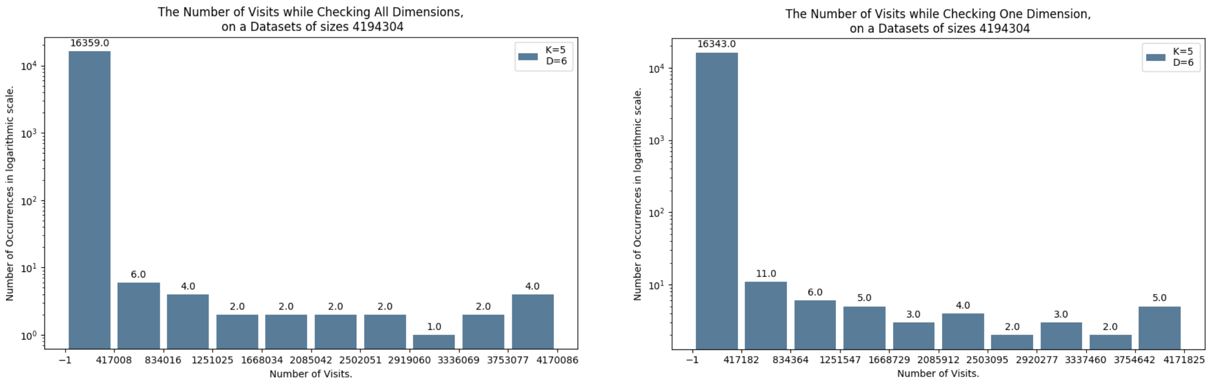


(a) Checking all dimensions.

(b) Checking one dimension.

Fig. 11: Figure a and b show the same result for K=3 and D=14 on both traversal solutions, on datasets of sizes 4194304.

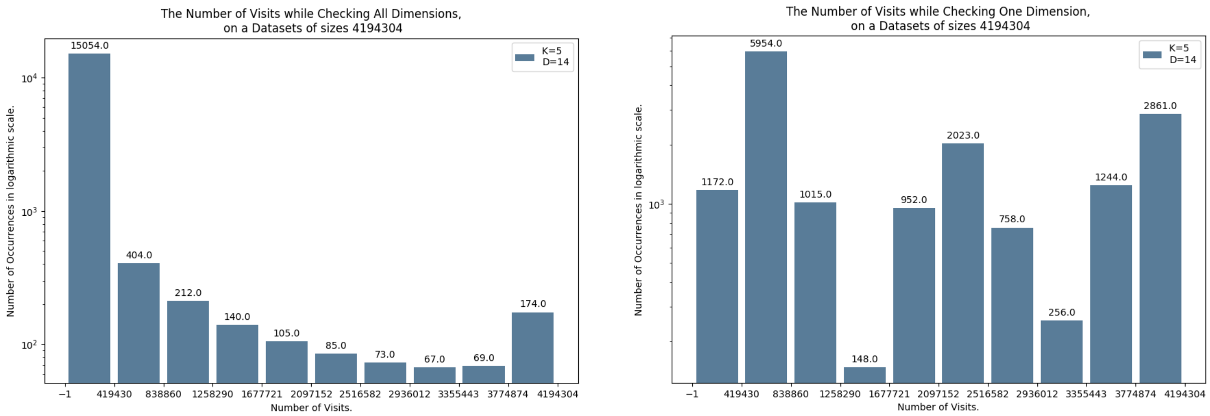
If we experiment further by increasing K to 5 while testing against both a small and large size D, namely 6 and 14, we see in figure 12 that even for a small D there is a slight decrease in the number of visits when checking all dimensions in 12a compared to checking one dimension in 12b. The figures in 13 show that for a K=5 and a D=14, the difference is incredibly in favour of the solution checking all dimensions. Thus, all histograms show increasing sizes for K and D will likewise have increasingly better performance on the solution checking all dimensions compared to checking one dimension.



(a) Checking all dimensions.

(b) Checking one dimension.

Fig. 12: Figure a and b show the same result for K=5 and D=6 on both traversal solutions, on datasets of sizes 4194304.



(a) Checking all dimensions.

(b) Checking one dimension.

Fig. 13: Figure a and b show the same result for K=5 and D=14 on both traversal solutions, on datasets of sizes 4194304.

4.3 Brute Force versus k-d Trees for Computing KNN

Table 3 summarises the speed-ups for testing the fully optimised k-d tree implementation against the brute force implementation, where we look at D sizes ranging from 2-14 and K sizes 2, 6 and 12. Figures showing the performance for K=2 and K=6, referenced in the table, can be found in the Appendix 6 figures 17, 18 and 19.

K	*D=2*	*D=4*	*D=6*	*D=8*	*D=10*	*D=12*
12	78.7	58.57	40.04	17.6	8.26	4.77
6	65.49	53.96	38.16	14.41	6.0	3.08
2	34.64	33.42	27.47	13.17	5.72	2.81

Table 3: Speed-ups achieved by the k-d tree solution compared to the brute force solution, based on datasets of sizes 1048576.

The results show that any given size of K and D results in a speed-up in favour of the k-d tree solution. This increased performance applies particularly for small dimensions and large K, for instance, K=12 and D=2 result in a speed-up of almost 79. Figure 14 below illustrates the performance for a static K=12.

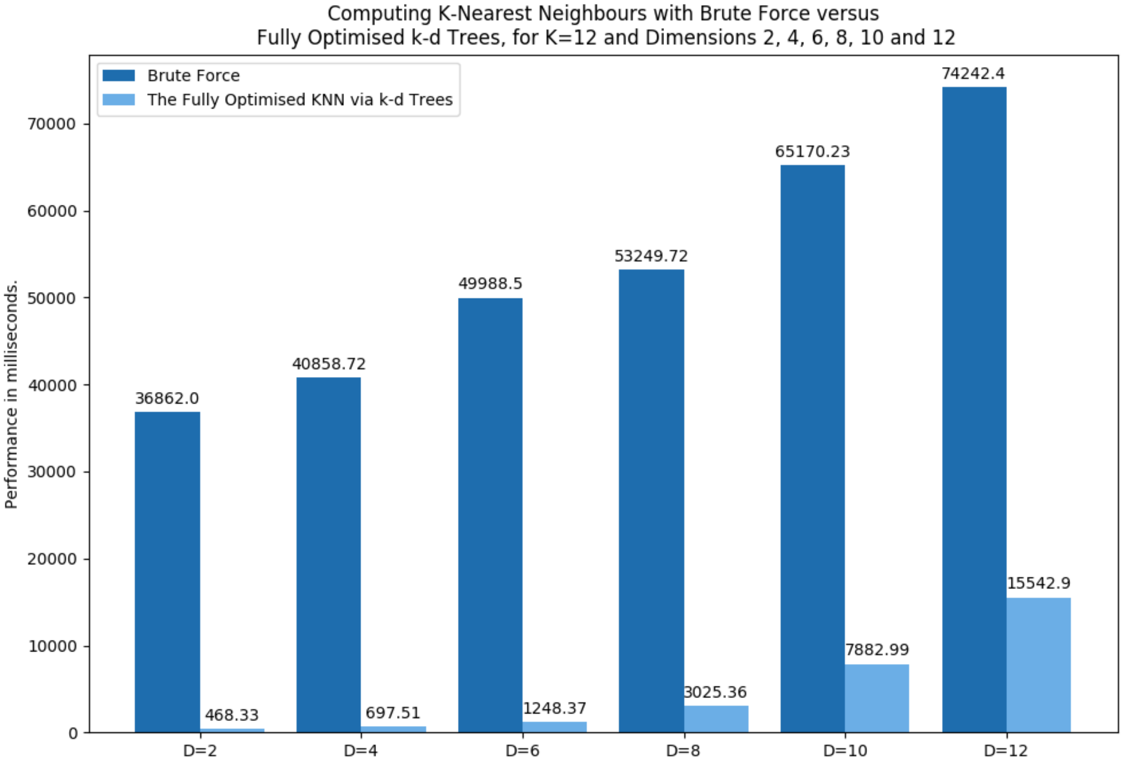


Fig. 14: Performance comparison of k-d tree versus brute force, with a static K=12 on datasets of sizes 1048576.

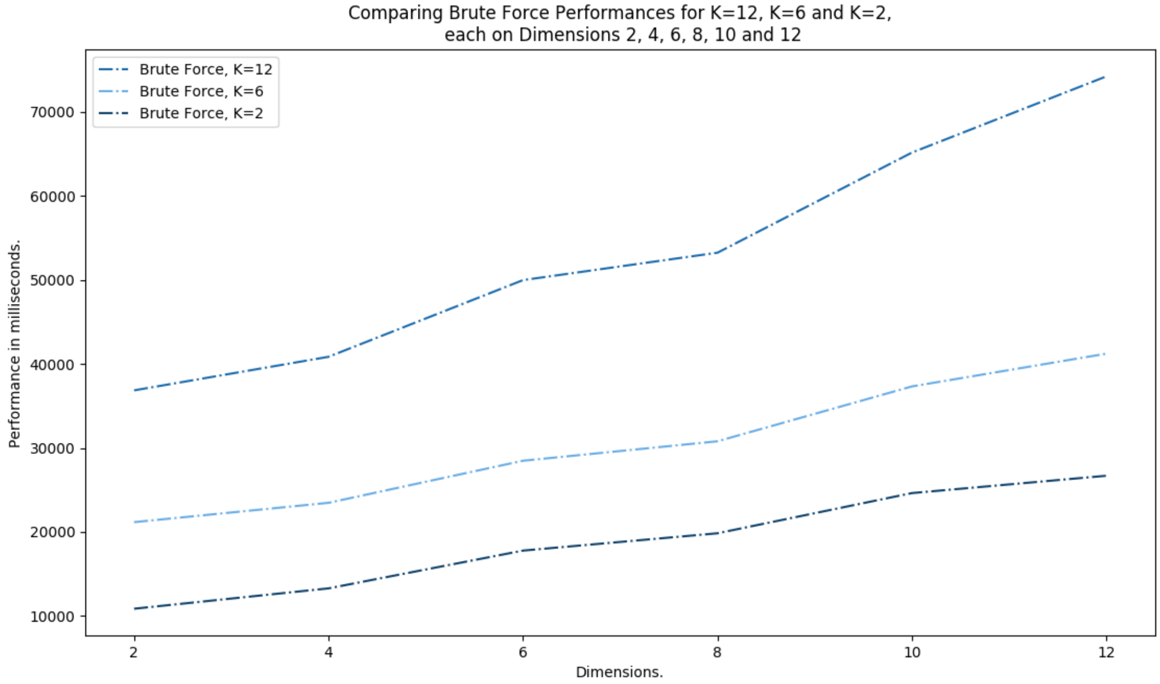


Fig. 15: Performance comparison of brute force, with K=12, K=6 and K=2, on datasets of sizes 1048576.

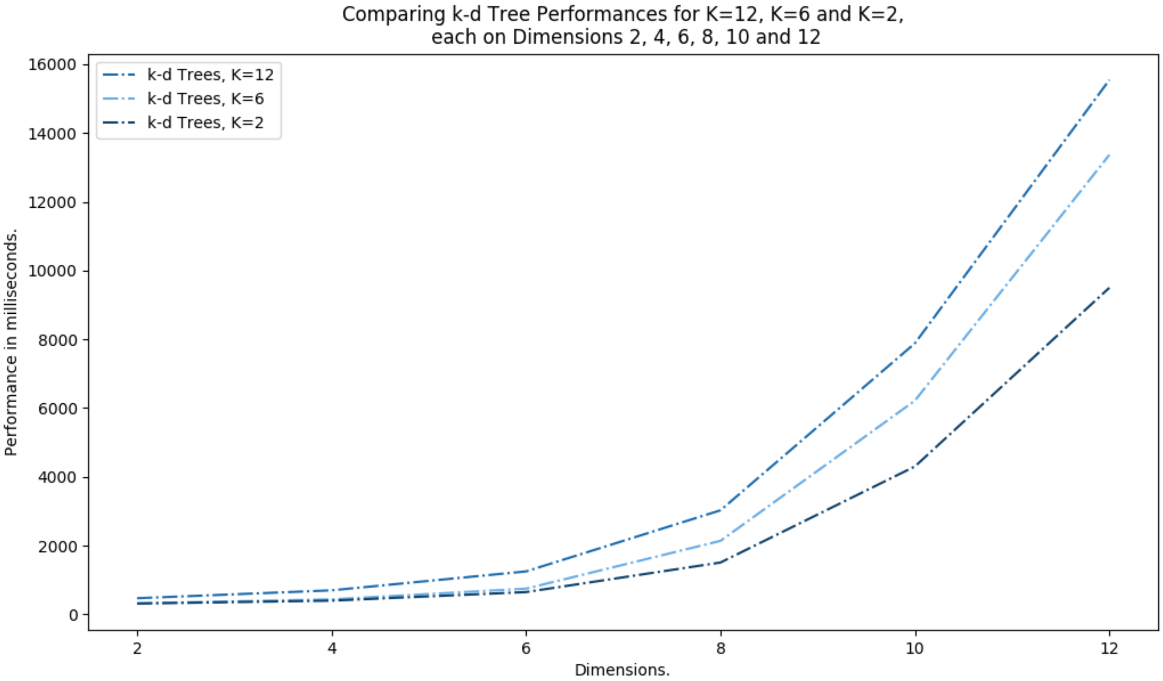


Fig. 16: Performance comparison of k-d tree, with K=12, K=6 and K=2, on datasets of sizes 1048576.

5 Conclusion

6 Appendix

6.1 Additional Graphs for Performance of Brute Force versus k-d Trees

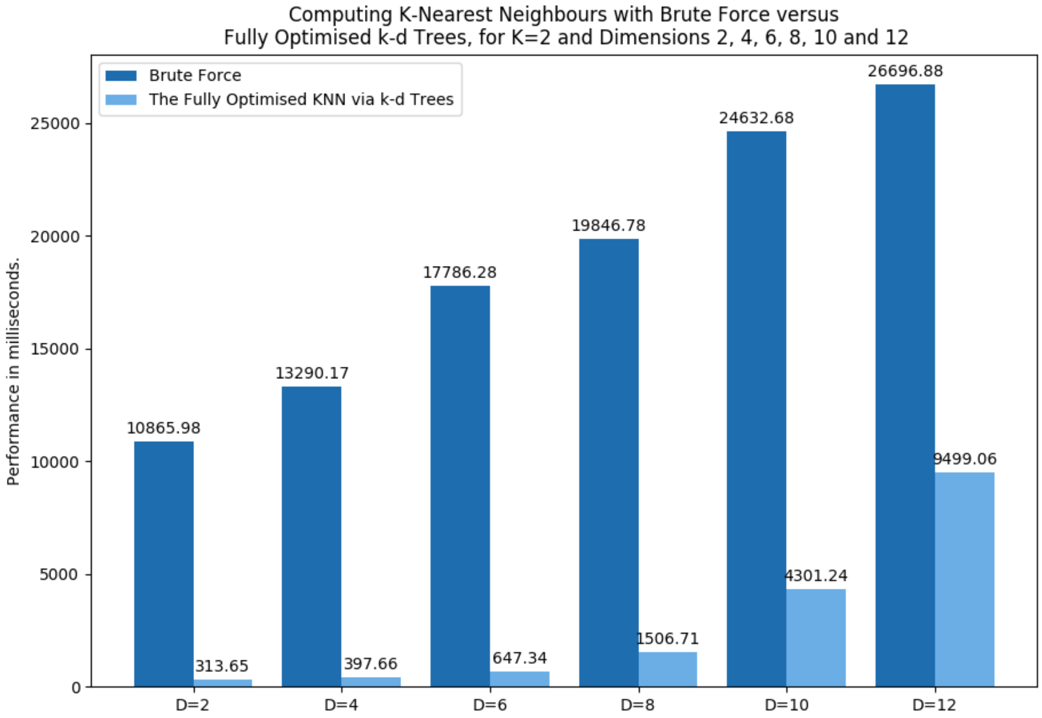


Fig. 17: Performance comparison of k-d tree versus brute force, with a static K=2 on datasets of sizes 1048576.

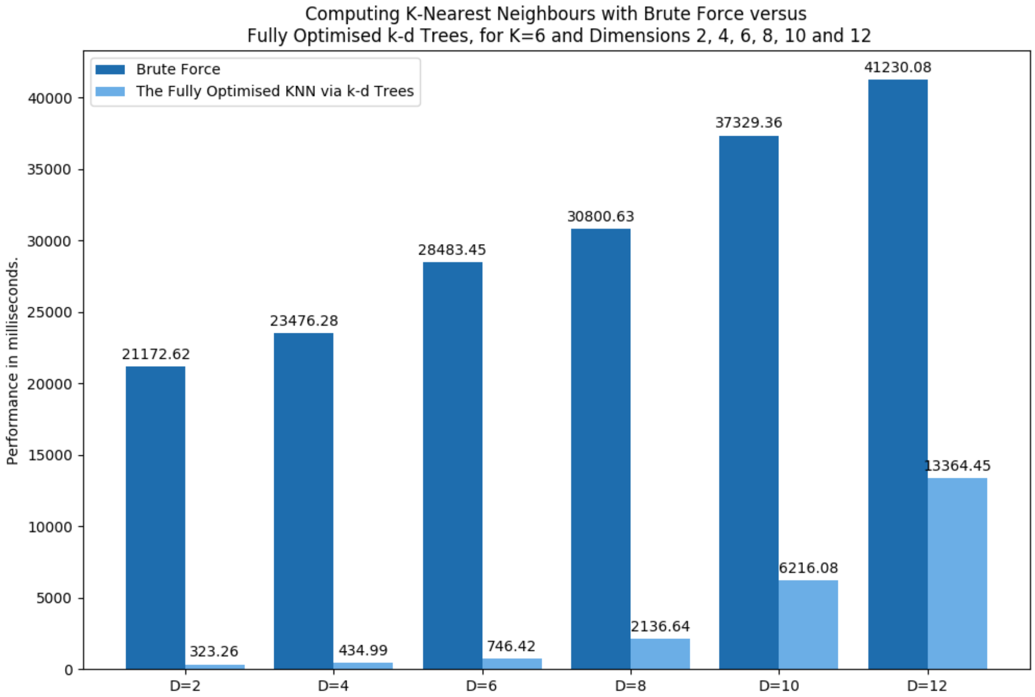


Fig. 18: Performance comparison of k-d tree versus brute force, with a static K=6 on datasets of sizes 1048576.

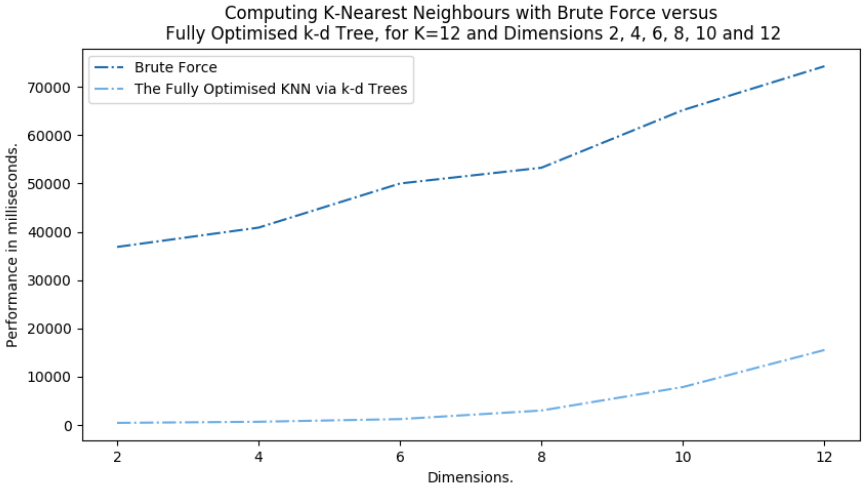


Fig. 19: Performance comparison of k-d tree versus brute force, with a static K=12 on datasets of sizes 1048576.

References

Friedman, J. H., Bentley, J. L., & Finkel, R. A. (1977). An algorithm for finding best matches in logarithmic expected time. *ACM Transactions on Mathematical Software*.

He, K., & Sun, J. (2012). Computing nearest-neighbor fields via propagation-assisted kd-trees. *Microsoft Research Asia*.