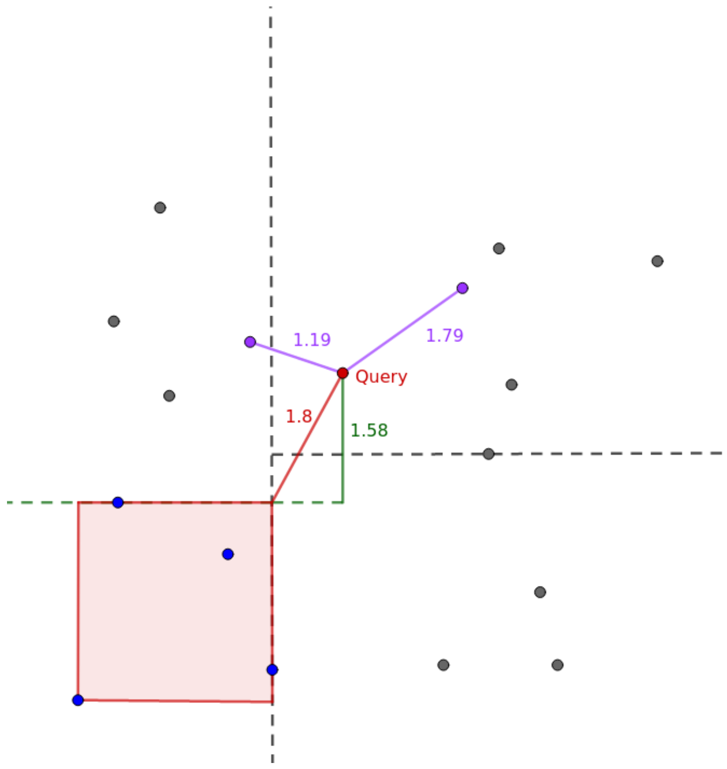
UNIVERSITY OF COPENHAGEN

FACULTY OF SCIENCE



Bachelor thesis

Ulrik Elmelund Petersen

Optimizing the kNN algorithm for GPGPUs in Futhark

using kd-trees, a novel boundary-test, and more!

Academic advisor: Cosmin Eugen Oancea

Handed in: June 29, 2020

Abstract

Fast computation of the `k` nearest neighbors problem (kNN) has important applications in fields such as machine learning, physics, astronomy. The brute-force approach compares each of the m queries to each of the n reference points and results in $O(mn)$ complexity, which is in many cases prohibitively expensive. Other solutions, such as kd-tree, split the reference set across the nodes of a tree according to a spatial topology that permits pruning the search space. They have been observed to reduce the average complexity to $O(m\,log(n))$, if the tree height is large enough.

This thesis presents a full data-parallel implementation of the kNN by means of a kd-tree search algorithm in the Futhark language—which is aimed at efficient parallel execution on GPUs. We start from a sequential, recursion-based implementation written in Python, and rewrite it as a nested composition of parallel constructs such as `map`, `reduce`, `scan` (and sequential loops). We then propose and test several optimizations: One of these corresponds to the kd-tree construction, where we use a padding technique to make parallel sizes regular—i.e., invariant to outer parallel constructs—that results in code that can be efficiently optimized by the compiler at the cost of negligible runtime overhead. Another improvement is designing a more accurate test (named boundary-test) for deciding whether a subtree can possibly contain candidates of interest or not. Our test reduces the number of false-positive visits, while maintaining perfect accuracy of the results. Finally, we improve temporal locality by sorting queries according to their last-visited leaves; this carries the same benefits as other solutions from literature, but in a simpler manner. We present a careful empirical evaluation, that tests points of various dimensions $d = 5, 7, 9, 12, 16$, and which reports high speedups in comparison with the brute force approach and demonstrates that the proposed optimizations have significant impact. As the best result, we report a $488\times$ speedup for 2^{21} (over two million) queries and the same number of reference points, with $d = 5$ and $k = 5$, while searching for k nearest neighbors on a `Geforce RTX 2080 Ti GPGPU`.

Contents

1 Acknowledgements

We acknowledge the ideas of Cosmin Oancea with regards to the various contributions of this paper, as elaborated in section 5. I also thank him for his continued interest and proactive support throughout the project.

2 Introduction

The exact k-nearest neighbor problem gives us two sets of points. One consists of references, and another of queries. It then asks us to find, for each query, which k references are nearest to it in euclidian space.

The kNN problem has relevance for many different domains, such as computer vision, deep learning, classification, and others. The brute force approach to solving this problem is straightforward, and simply tries all combinations. However, this has a very high computational complexity, making it prohibitively expensive for very large datasets, in terms of processingtime.

There are several alternative approaches, which deal with this issue, and some of these have been investigated. One of these is based on using a spacial search-structure, called a kd-tree, which recursively partitions the references according to their shared properties, and stores the information about these partitions in the internal nodes of a balanced binary tree. Each of these nodes corresponds to a 'box', containing the points within it, and each node splits these points in two parts, each of which is the basis for the nodes two children. The split is done based on their sorted order in the dimension of highest spread. Below the tree, the points are placed in a leaf-structure, which can be thought of as the last level of the tree.

This tree can then be traversed, based on this information, by the queries, in order to find their nearest neighbors. Depending on the shared spacial properties stored in the tree, various parts of it can be skipped during the traversal. In the traditional manner, as done in [5], this is done by comparing the distance from the query to the most distant of its current nearest neighbors, to the distance from the query to the median which the points were split through. In this paper, a newer method is investigated, in which the comparison is instead made with the distance from the query to the bounding box of a given node. This difference is illustrated on the front page of this paper, but explained more in detail later. While we are not comparing directly to [5], we are comparing indirectly, since it is state of the art, and widely used.

While the brute force approach has a computational complexity that is quadratic ie. $O(mn)$, a kd-tree traversal typically only visits $O(log(n))$ leaves, wherefore the complexity should be $O(m\, log(n))$, which is cheaper.

In this paper, many other improvement are also presented. The construction of the kd-tree is done in a data-parallel fashion, using a small amount of padding. The recursive traversal of the kd-tree is split into two stages, and the information nessisary is stored very compactly as bits in integers. The queries are sorted according to the leaves they are visiting. This improves temporal locality a lot, because queries that are going to process the same points in the same leaf, are going to do so at the same time.

In section 7 we evaluate speedups with respect to the brute force approach. We then compare the speedups between different versions of the algorithm, and assess the impacts of the main optimizations.

When processing a dataset with over two million queries and reference points, with dimensionality of 5, and searching for the 5 nearest neighbors, the fastest version of the algorithm gives a $488x$ speedup. However, when $d = 16$, the speedup is only about $2x$ faster than brute force, meaning that this approach is not free from the curse of dimensionality either. The benchmarks all start with a minimum dimensionality of 5, because more efficient methds are known for those below that, such as octo-trees etc.

3 Problem statement

There are many applications where one wishes to find the nearest neighbors of a number of points. These range from astronomy to deep learning and more. This paper deals with the process of finding these neighbors, and not with what application they are used for. Defined formally, the problem is: Given a set of n reference points P, and a set of m query points (queries) Q, both in $\mathbb{R}^d$, where $d \in \mathbb{N}^+$. For every $q \in Q$ find the k points in P that have the shortest euclidian distance to q. Informally, we want to find the nearest neighbors for every query, and the reference points are the candidates for this. The problem is a type of search problem, since we want to find a result within a solution-set.

4 Background

4.1 Brute force algorithm

There are many differnet ways of solving the problem. The simplest of these is by way of brute force, which can work as follows, shown in pseudocode:

```
BRUTE-FORCE-KNN($Q, P, k$)
1  // $knns$ is an array of arrays of tuples of int * float
2  Initialize empty $knns$ for Q
3  for every $q$ in $Q$
4      for every $p$ in $P$
5          $tmp = dist(q, p)$
6          if $tmp < knns[q.ind][k-1].dist$
7              $ind = p.ind$
8              for $i = 0, i < k, i++$
9                  if $dst$ < $knns[q.ind][i].dist$
10                     swap($tmp$, $knns[q.ind][i].dist$)
11                     swap($ind$, $knns[q.ind][i].ind$)
12 return $knns$
```

The algorithm goes through every possible combination of queries and references, and checks whether a given reference point is closer than the (worst) most distant of the current neighbors, of the given query. If it is, the new point gets inserted in sorted order into the knn candidates for the given query. Once every combination has been exhaustively tried, the exact results will have been found. This means that for every query, the nearest neighbors in P have been added to the query's set of knns.

When the queries are processed in parallel, this implementation exhibits good temporal (and spacial) locality, because the reference points are accessed in the same order in the computation of each query, and they will likely be reused from the L1 and/or L2 cache. However, this approach has $O(nm)$ work—because each query is compared to all reference points—making it very slow when the number of points in Q and P is large.

4.2 K-d tree

4.2.1 A spacial-search datastructure

In order to decrease computational complexity, we can reduce the search space in some manner, and not compute the distance between every single combination of queries and references. One way this is done, is by clustering the reference points together, according to their spacial properties. Those that are close to each other will be clustered together, and they will be handled based on their common properties.

A common way of doing is by using a k-d tree (k-dimensional binary tree), which is a spacial-search structure.

4.2.2 Recursive k-d tree construction

During the construction of the kd tree, the points are partitioned several times, once at each level of the tree.

Each node in the tree splits the points in in two equal parts, such that the points in each part fall on either side of a median in one of the d dimensions.

For every node, this is done by picking a dimension—typically the dimension exhibiting the highest spread—then sorting the points according to it, and then splitting them in half. The two clusters of points are each contained in a space that is about half the size. Each of these clusters then form the basis for continuing the process in the right and left children of the given node, in the next iteration.

The internal nodes of the tree contain information about which dimension the points were split along, and the median value that split them.

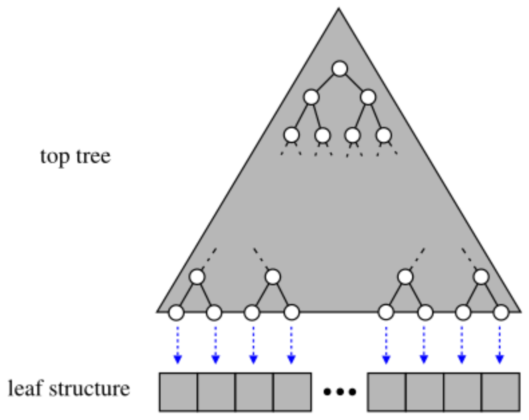


Figure 1: Illustration of how the top tree relates to the leaf structure. Adapted from one found in [5].

In this way, the points are progressively reordered and partitioned into smaller and smaller 'boxes', with each internal node symbolically corresponding to a box of points. (Have an illustration of this) When they are processed, this information is used to decide whether to process specific parts of the tree or not, enabling us to lessen the work to be performed. Hopefully only $O(log(n))$ nodes leaves will be visited by each query.

In practice, the brute force algorithm beats a kd-tree implementation when the datasets are small. For this reason, the splitting process stops at a certain treshhold. At that time, the points are stored in a leaf-structure that 'attaches' to the bottom of the kd-tree (also called the top-tree). Every node in the bottom layer of the top tree has its left and right children in this leaf structure.

In the classical kd-tree construction, each node in the tree stored the index of the dimension that was split by that node, together with the median value on that dimension. The tree information is stored into an array, in which the left and right children of a given node i are located at $2i + 1$ and $2i + 2$, respectively.

4.2.3 Example tree construction

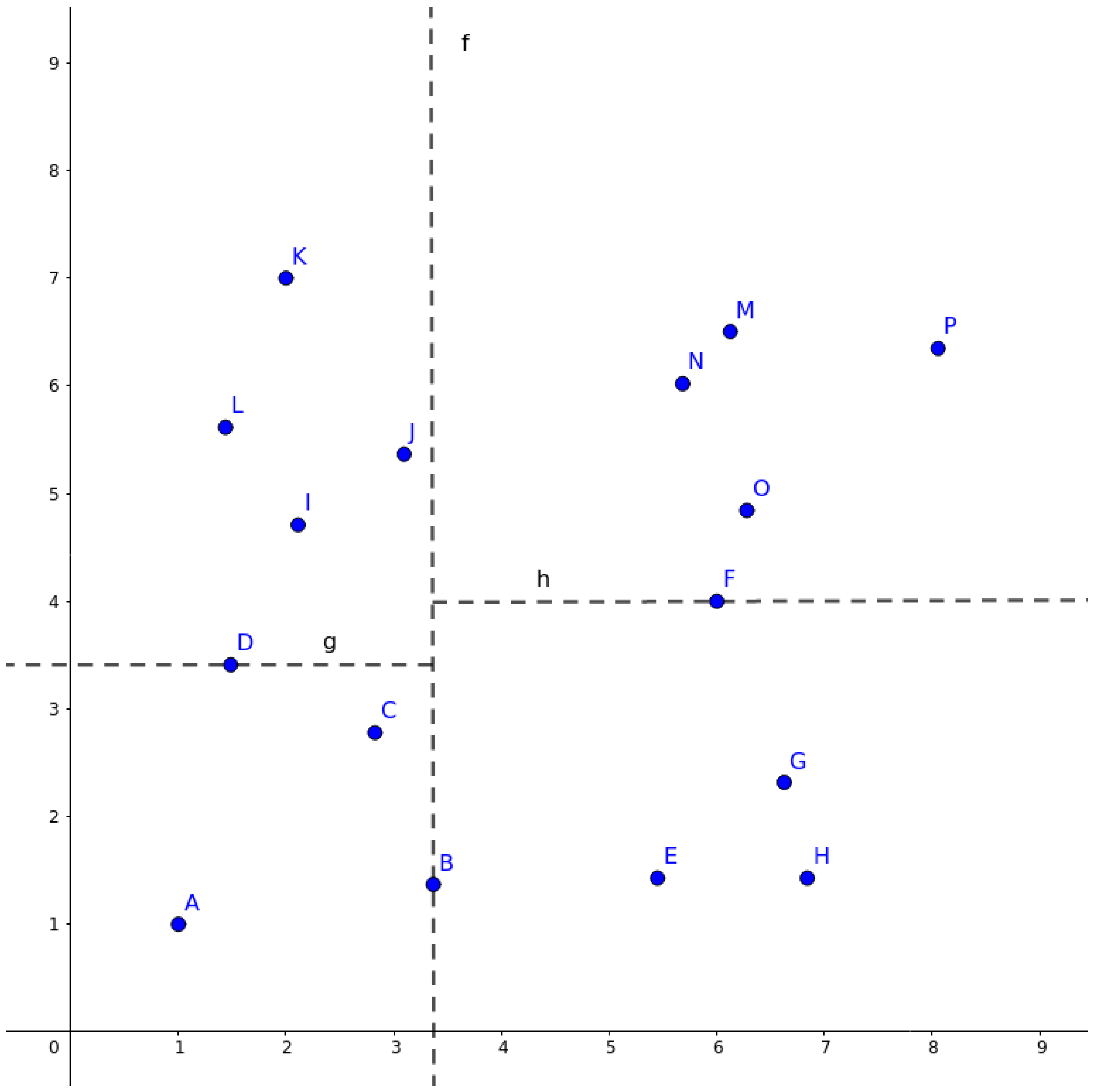


Figure 2: Consider the scenario of constructing the kd-tree using these 16 reference points. The first dimension to be chosen is the x-axis, because this has the highest variance of $\sim$ 7 compared with $\sim$ 6 for the y-axis. The points are split along this dimension, and the median value will be the x-dimension of the point B. This splits the points into two boxes, one in either side of the median, which is shown as the line f. Each of these two clusters are then split again, this time in the y-axis. The medians will be those of D and F, and the lines h and g go through these to show how the space is partitioned by this process. The references have now been split into several nested boxes, denoted by the lines drawn through the medians, with each pair of boxes 'fitting' into a parent box. The threshhold of four per leaf has been reached, so the process stops, and there are now four leaves in the leaf structure, each containing ABCD, EFGH, IJKL or MNOP.

4.2.4 Traversing the k-d tree with the queries

Processing is done by traversing every query through the tree, and visiting all the leaves that might contain better neighbors. The recursive process starts at the root. One way this can be done is the following:

- l. 2: Every recursive call first checks if the node-index is at a leaf in the leaf-structure.
- l. 3-5: If it is, recursion stops and it returns the result of running the brute force algorithm on the leaf.
- l. 6-12: If it is not a leaf, it must be an internal node in the tree. It finds out which child to visit first, by checking which side of the median the query naturally belongs on. We want to first visit the child that is on the same side of the current median as the query. Therefore, we name that index 'first', and 'second' is the index that corresponds to the child the query does not naturally belong to.
- l. 12: Recursion to the first child is not guarded by branch, meaning we will always visit that child. We do this because the query is on the same side of the median as that child, so chances are that it will find good neighbors there.
- l. 13-14: When it returns, it checks whether to visit the second child or not. This depends on whether there might be better neighbors in that child-node or not. In [5], this is decided, as shown, by checking if the distance from the query to the median, is less than the distance from the query to the "worst" of its currently known nearest neighbors. If so, then the algorithm conservatively decides that the "second" node might contain better nearest-neighbor candidates, and hence, needs to be (recursively) visited. Otherwise, if this condition does not hold, then it is guaranteed that no better candidate exists in the reference points enclosed by that node, and hence there is no need to compare to any of them.
- Because the decision is based on the median-value, this test is refered to as the median-test in this paper. It is a conservative test, as it only makes this decision based on one median value. The method used in this project is different. See section 5 for details. The first recursive call is always performed, the second is guarded by the test. In the best case, only one leaf is ever visited.
- l. 15: Finally, it returns the knn results from its own treeidx. The root call returns the result from processing the entire tree.

In this example, the treeidx is an integer representing the current node index into the top tree. As mention before, the tree consist of the medians and the dim (the dimension a node was split in). `knn` is the set of k nearest neighbors for that query, initialized correctly beforehand. It contains distances and indices for each neighbor. 'leaves' here is the leaf-structure.

4.2.5 Example traversal

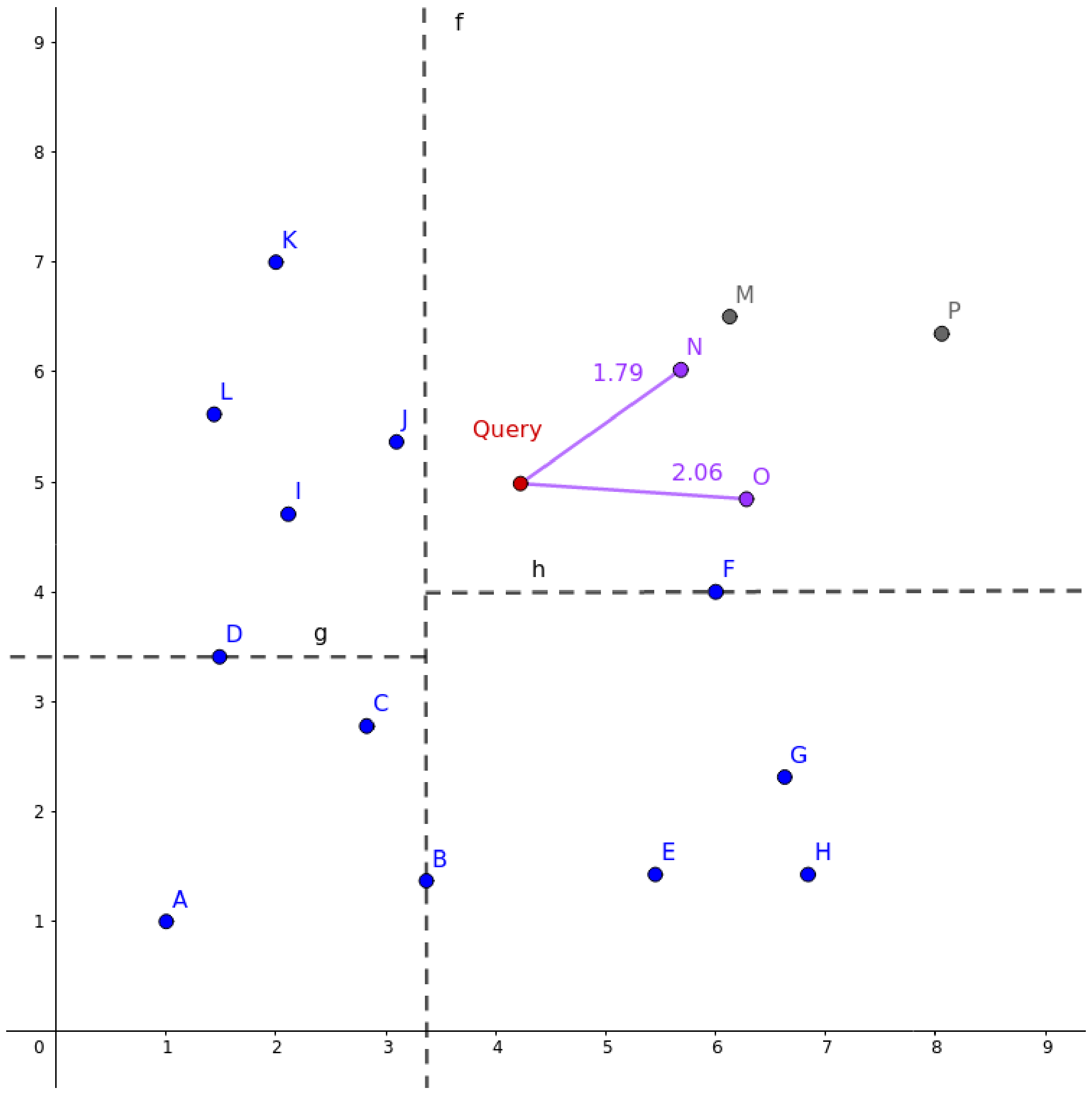


Figure 3: Consider the kd-tree from the previous example, and just one query, as marked with red here. Consider that we wish to find just the two nearest neighbors for this query. The first recursive call compares the query to the outer median f, and then h, and thereby finds the natural leaf of the query to be the one containing the points MNOP. The brute force algorithm runs on the leaf, and the two nearest neighbors are found to be N and O.

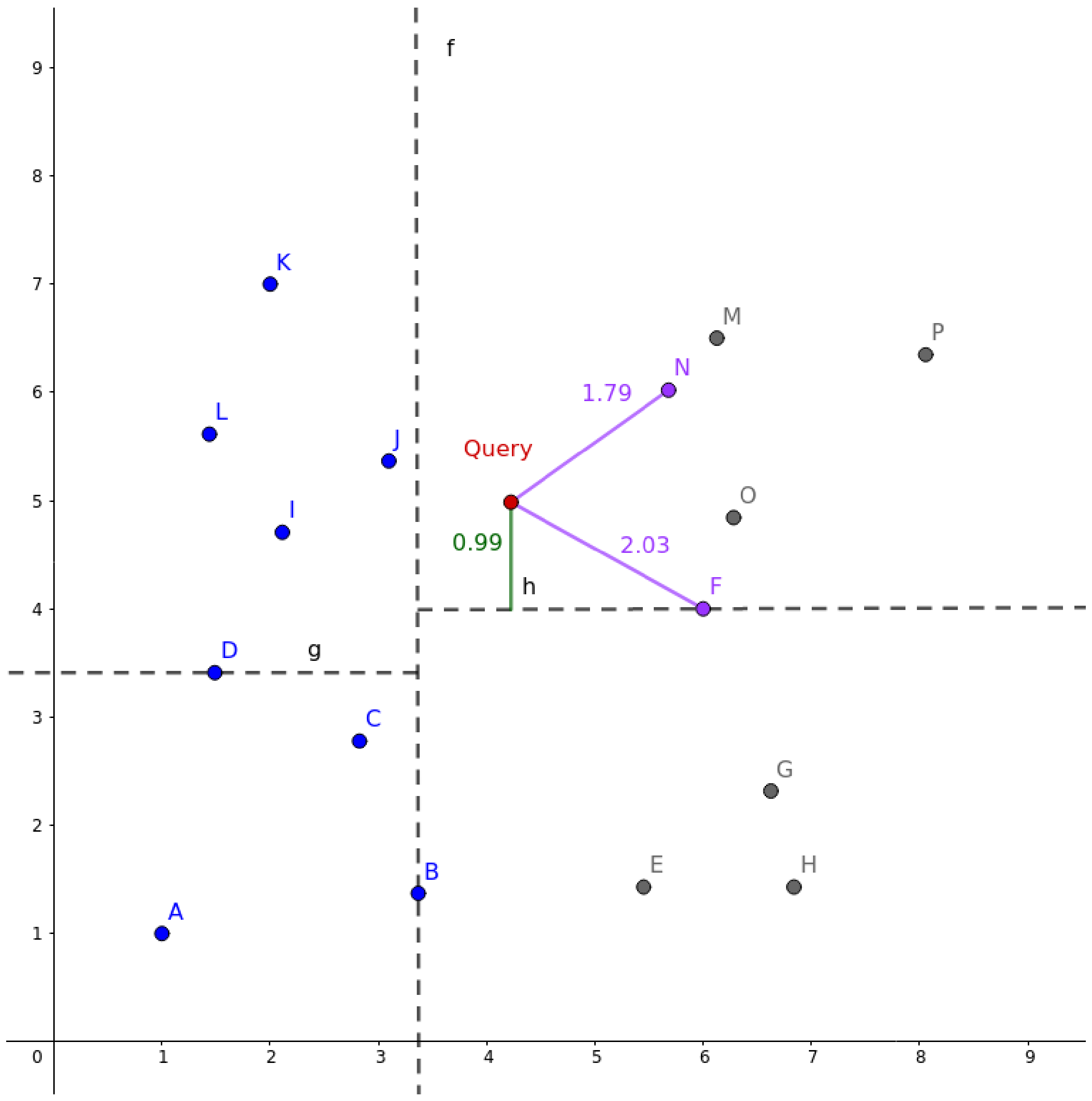


Figure 4: Having returned from the first leaf, the traversal considers whether to visit its sibling leaf. Since the distance to the sibling is less than that to its worst nearest neighbor, O, it visits the leaf, and updates its nearest neighbors to F and N.

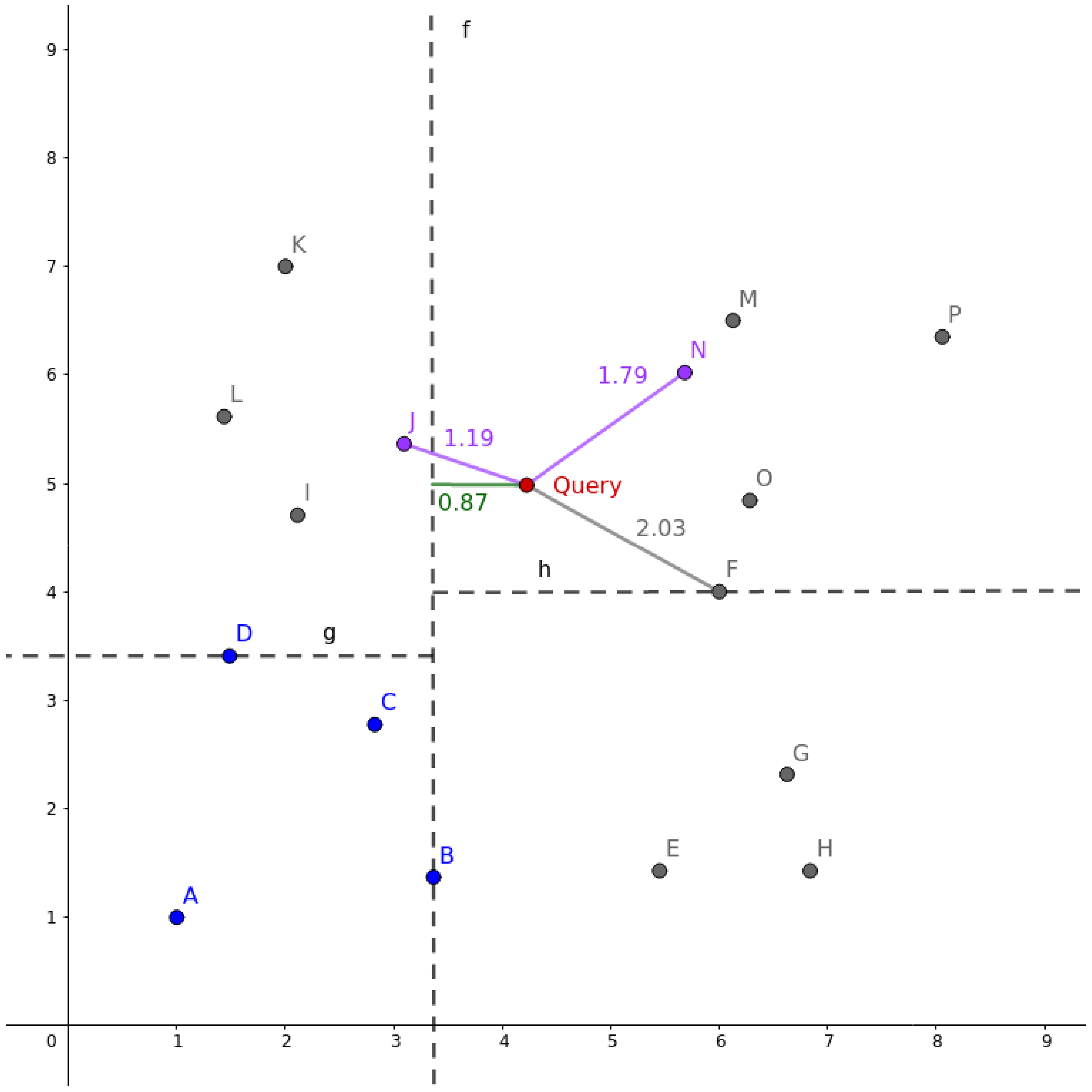


Figure 5: The traversal now comes back to the initial recursive call, and decides to enter the second child of the root node, because its distance to the median 0.87 is less than that to F. It then finds the natural leaf of the second child – of the root node – to be IJKL. It updates its nearest neighbors again. They are now J and N.

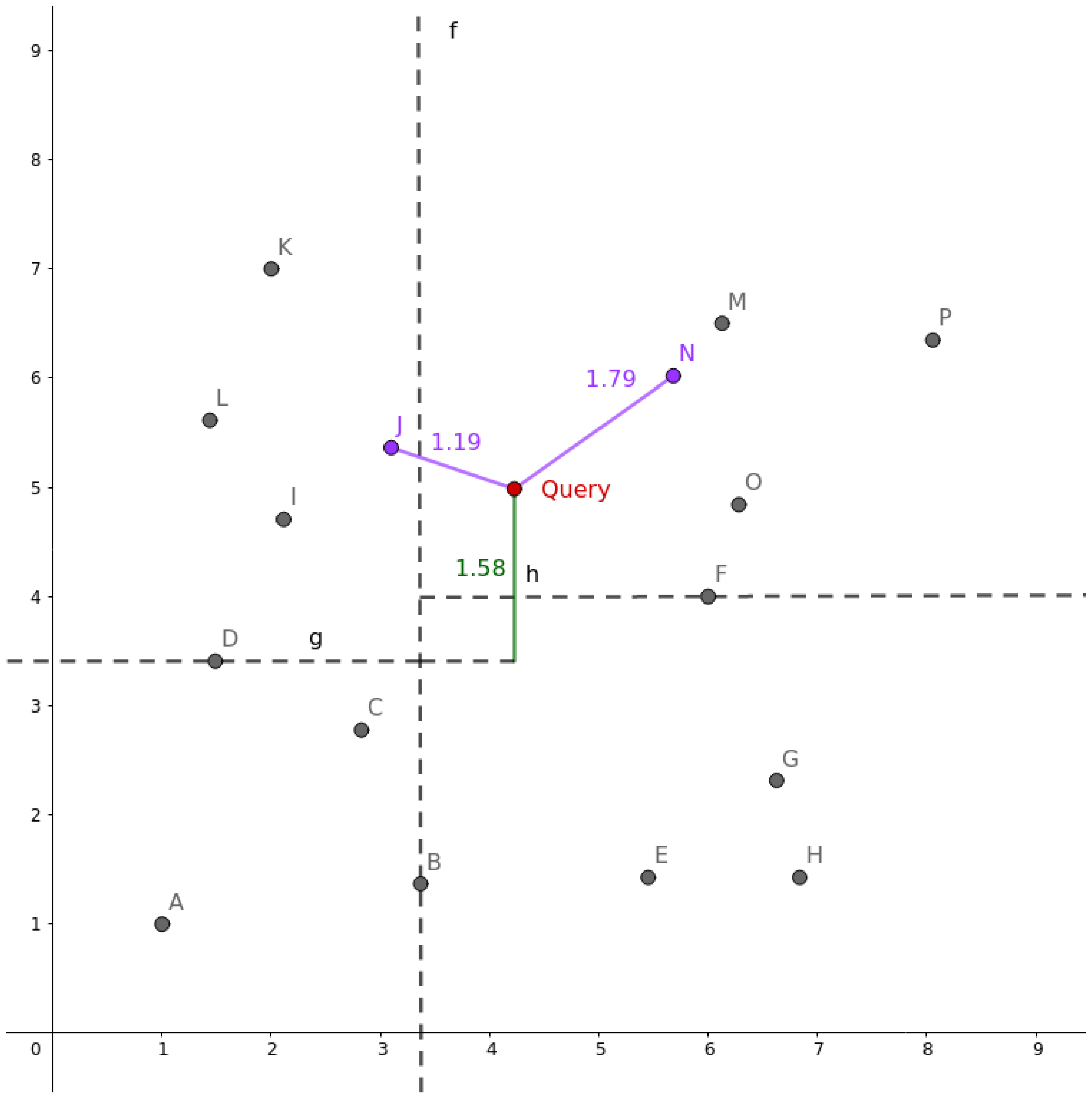


Figure 6: Finally, considering the last leaf, it decides to visit it, because its distance of 1.58 is less than 1.79 – he distannce to its most distant nearest neighbor N. But it turns out to be in vain, as none of the points in that leaf are any better than its current candidates. Every call then returns, and the nearest neighbors have been found.

4.3 Related work

4.3.1 Buffer kd-tree

One approach to using kd-trees is explained in [5]. This is the approach which this project sees as the standard, as it has been widely referenced and the library implementing it is widely used. Various queues and buffers are used to organize the order of processing for the queries, but in this project, all queries are processed simultaneously, eliminating the queues, and they are sorted according to their leaf, thus eliminating buffers. The dimension for splitting points, was chosen as a function of the depth in the tree, and therefore did not have to be stored. Choosing the dimension of highest spread improves the efficiency of the traversal, and that is the method employed in this project.

4.3.2 Recently published approaches

Another approach is described in [2], which was followed up by [1] later. These approaches use a more accurate test, which bears some similarity with the boundary-test described in this paper.

4.4 Futhark

Both alternative approaches mentioned in the previous subsections use either CUDA or OpenCL to develop their solution. This project uses the high-level language Futhark [4], which has two main advantages:

- it is much less tedious to use than low-level GPU APIs, and allows quick prototyping of parallelization strategies, see for example heuristics for breath-first search algorithm [8] and an application for monitoring changes in landscape by analysis of satellite images [6];
- it allows one to cleanly expresses the whole parallel structure of an application in terms of data-parallel building blocks such as `map`, `reduce`, `scan`, `scatter` (or parallel write).

As stated on Futhark's website[1], *Futhark is a small programming language designed to be compiled to efficient parallel code. It is a statically typed, data-parallel, and purely functional array language in the ML family, and comes with a heavily optimising ahead-of-time compiler that presently generates GPU code via CUDA and OpenCL.*

4.4.1 reduce

One example of how futhark code differs from regular C-like code can be seen in the following example (from the same website), which computes the average of an array of length `n` of 64-bits floating point numbers (a.k.a., double):

`reduce` takes a binary function, here addition, a neutral element and an input array. The result of the reduction is the sum of all the elements, which is then divided by the number of elements to get the average. As compared to equivalent C-code below, the above is very concise and arguably more readable:

`reduce` has the type:

$$reduce : (\alpha \rightarrow \alpha \rightarrow \alpha) \rightarrow \alpha \rightarrow [n]\alpha \rightarrow \alpha$$

4.4.2 map

Another kind of SOAC (second-order-array-combinator) is `map`, which takes a function and applies it to every element in a collection. For example the following code snippet adds one to each element of an array of doubles:

One of the important optimizations that our kd-tree implementation relies on is the fusion of parallel constructs [10, 9, 12], for example in the program below:
the `map` produces an array that is consumed by the reduce (sum) operation. A naive execution that manifests the result of the `map` in memory and then sums the resulted array will perform at least $3n$ accesses to global memory, where n is the length of the input array. However, when fusion is applied the result of the `map` is not manifested in memory, thus reducing the number of accesses to global memory to about n. `map` has the type:

$$map : (\alpha \rightarrow \beta) \rightarrow [n]\alpha \rightarrow [n]\beta$$

[1] `futhark-lang.org`

4.4.3 scan

`scan` is similar to `reduce`, in that they both apply a given binary-function to a neutral element and an array. In this sense they have the same type restrictions. However, the difference lies in what they return. While `reduce` returns just one value, `scan` returns an array of the same shape as the input array, where each element corresponds to the result of a `reduce`, up to that point. A `scan` is also known as a *prefix sum*. `scan` has the type:

$$scan : (\alpha \to \alpha \to \alpha) \to \alpha \to [n]\alpha \to [n]\alpha$$

4.4.4 Segmented scan

Because irregular multi-dimensional arrays can not be efficiently flattened, futhark does not support them. However, irregular arrays can still be represented in futhark, using a flat array containing the values, and a flag array. These have the same length, and the flag array contains either a `true` to denote the start of a segment, or a `false` otherwise. Each segment then corresponds to an inner array of the irregular multi-dimensional array being simulated.

In order to perform a scan on each segment sepparately, and not on the whole array, a function known as `segscan`, meaning segmented scan, is used. In addition to the arguments taken by `scan`, this function also receives the flag array.

`segscan` has the type:

$$segscan : (\alpha \to \alpha \to \alpha) \to \alpha \to [n]int \to [n]\alpha \to [n]\alpha$$

It's worth mentioning that since `segscan` is not a built-in function, but rather implemented as a library function, this definition is not a given. Following the common convention that any non-zero integer is `true`, and zero `false`, the flag array often has the flags be more useful than just booleans. For example, each flag can contain the length of the segment. This is sometimes useful in other segmented operations. An example implementation of `segscan` is shown here:

It is implemented as a regular scan, but wraps the application of the binary function 'op' in a check. The result is then extracted by pipelining it through and unzip, which turns the array of tuples into a tuple of arrays, and this is then piped further to extract the result.

4.4.5 scatter

Another important SOAC is `scatter`, which receives a destination array, a set of indices, and a source array. The indices indicate for each element in the source array, the index into the destination array that it should be written to. It has the signature:

$$scatter : [n]\alpha \to [m]int \to [m]\alpha \to [n]\alpha$$

4.4.6 partition

`partition` is used to split an array into two arrays, determined for each element by whether it fulfills a given predicate. The first array contains those elements that fulfill it, the other those that do not.

$$partition : (a \to bool) \to [n]\alpha \to ([m_1]\alpha * [m_2]\alpha)$$

4.4.7 gather1d

`gather1d` is used very frequently in the code shown in section 6, so we believe it is worth showing its implementation here. It is a simple map over an array of indices, which are used to get elements from a source array.

There are many other SOAC's, such as `filter`, `reduce_by_index`, and these can be easily combined together in a nested fashion to build code that describes all available parallelism (in a relatively concise notation).

4.4.8 iota, length, indices

`iota` is another commonly used futhark function. It is used to create arrays of indices of a given size. It receives an integer n, and returns an array of size n of integers, containing the numbers from 0 to $n-1$.

$$iota : int \rightarrow [n]int$$

`length` is another commonly used futhark function. It returns the length of a given array. `indices` is yet another similar function, which returns an array of indices corresponding to a given array. Applying `indices` is equivalent to applying `length` followed by `iota`.

While Futhark supports arbitrarily-nested parallelism, the low-level API for GPU programming morally supports only flat parallelism. An example of this would be a perfectly parallel nested loops. To brige this gap, the Futhark compiler performs a code transformation, which attempts to improve the degree of parallelism, by extracting such perfectly parallel nests – this is the flattening transformation. This is achieved by systematically interchanging `map` operations inside sequential loops, and by distributing `map` operations across the inner code containing parallel constructs. This kind of transformation requires that the inner parallel operations have sizes which are invariant to the enclosing `map` operations. This property is referred to as *regular parallelism*, and its achievement was central to the efficient parallelization of the kd-tree construction process, which is described in section 6.

However, exploiting parallelism in excess of what the hardware can support is not beneficial. For this reason, Futhark performs an analysis that creates multiple semantically equivalent versions [11] of the code that *incrementally* map to the hardware more and more of the application's levels of parallelism, and efficiently sequentializes the rest. The version of code to be executed, is determined at runtime, by comparing the amount of exploited parallelism of a certain code version with a threshold, which is found by autotuning. This multi-versioning technique draws inspiration from other hybrid (static and dynamic) analyses, which have been studied in the area of automatic parallelization [14, 13]. There, the code versions correspond to a cascade of sufficient conditions, which verify at runtime that the target loop can be safely executed in parallel.

Finally, Futhark is not meant to be general purpose, but rather domain-specific to accelerating the computational kernels of real-world applications, written in mainstream, productivity-oriented languages. To this end, Futhark provides code generators to, for example, Python+OpenCL [7], which allows such computational kernels generated in OpenCL to be easily imported and used as Python modules.[2] We expect that the parallel implementation presented in this thesis can be similarly used in larger projects written in mainstream language that require fast kNN computation.

5 Contributions of this paper

5.1 Parallelized kd-tree construction

One contribution of this paper, is that the construction of the kd-tree (and associated datastructures) has been parallelized. This means that it is also performed on the GPU, while every other published approach, as far as we are aware, constucts these data-structures on the CPU in a sequential fashion. In order for futhark to compile it to efficient parallel code, the inner parallelism

[2] Such code generators are very useful, but they do not currently support type-parameterized entry points, and they require significant maintenance. A potential solution in this sense can be the use of a interface-definition language supporting generic types [15, 3] that would automatically generate client wrappers across multiple languages for the required functionality, which would be implemented in Futhark.

had to be regular. If the number of points was not such that it could be evenly split between the leaves, this would not be possible. In order to regularize the inner dimensions of all the parallel operations, padding was nessisary. We use a padding heuristic which ensures that all leaves contain the same number of points (named leaf-size), while introducing at most one "dummy" point per leaf, and trying to keep the leaf size within a factor of two of the one specified (desired) by the user. The result is a tradeoff between having control over the leaf-size, and minimizing the amount of padding. For details, see section 6

The padding heuristic allows us to express the kd-tree construction by means of regular parallelism, because each node at the same level in the tree contains the same number of points, which is also the size of the inner-parallel constructs, e.g., the reductions necessary to find the dimension of highest spread, and the sorting needed to recursively split the points of a node to its sibling. The regular parallelism allows the Futhark compiler to generate efficient GPU code, at a negligible runtime cost of processing a small number of padded/dummy elements.

5.2 Sorting

The approach taken by [5], uses a very complicated approach in order to optimize temporal locality. We simplify that, and gain the same effect, by simply sorting the queries according to the leaves they fall into. This ensures that the number of consecutive queries are going to access the same reference points in the same leaf, at the same time, thus improving temporal locality of reference.

5.3 Two-stage iterative traversal

Since Futhark (and the GPU hardware) does not allow recursion, the traversal of the queries and the processing of the leaves have to be structured entirely differently, in an iterative way. Essentially, an initialization step is performed that propagates of each query from the root of the tree to the leaf the query naturally belongs to. Each recursive call then becomes an iteration of a loop, in which each of the in-progress queries starts from the previosuly visited leaf, finds the next leaf to be visited and performs a brute-force search on its points. We present how the traversal can be efficiently implemented by representing (and maintaining) the stack for each query as an integer whose bits record whether the "second" node[3] has been already visited for each of the ancestors of the current leaf (up to the root of the tree).

5.4 Bounds-based test

The last contribution of this project is the boundary-test, and the associated bounds-structure. This is a more precise test, which elimiates more false positives. This means that it rejects more subtrees, and traverses fewer nodes and processes fewer leaves. The only differences occur when it decides whether to visit a sibling node or not. Consider the following example, showing how the traversal shown in the background section changes when this new test is applied. One example is shown for each of the three situations where 'second' children are considered.

[3] In our notation the "second" node corresponds to the sibling on the other side of the current median with respect to the query.

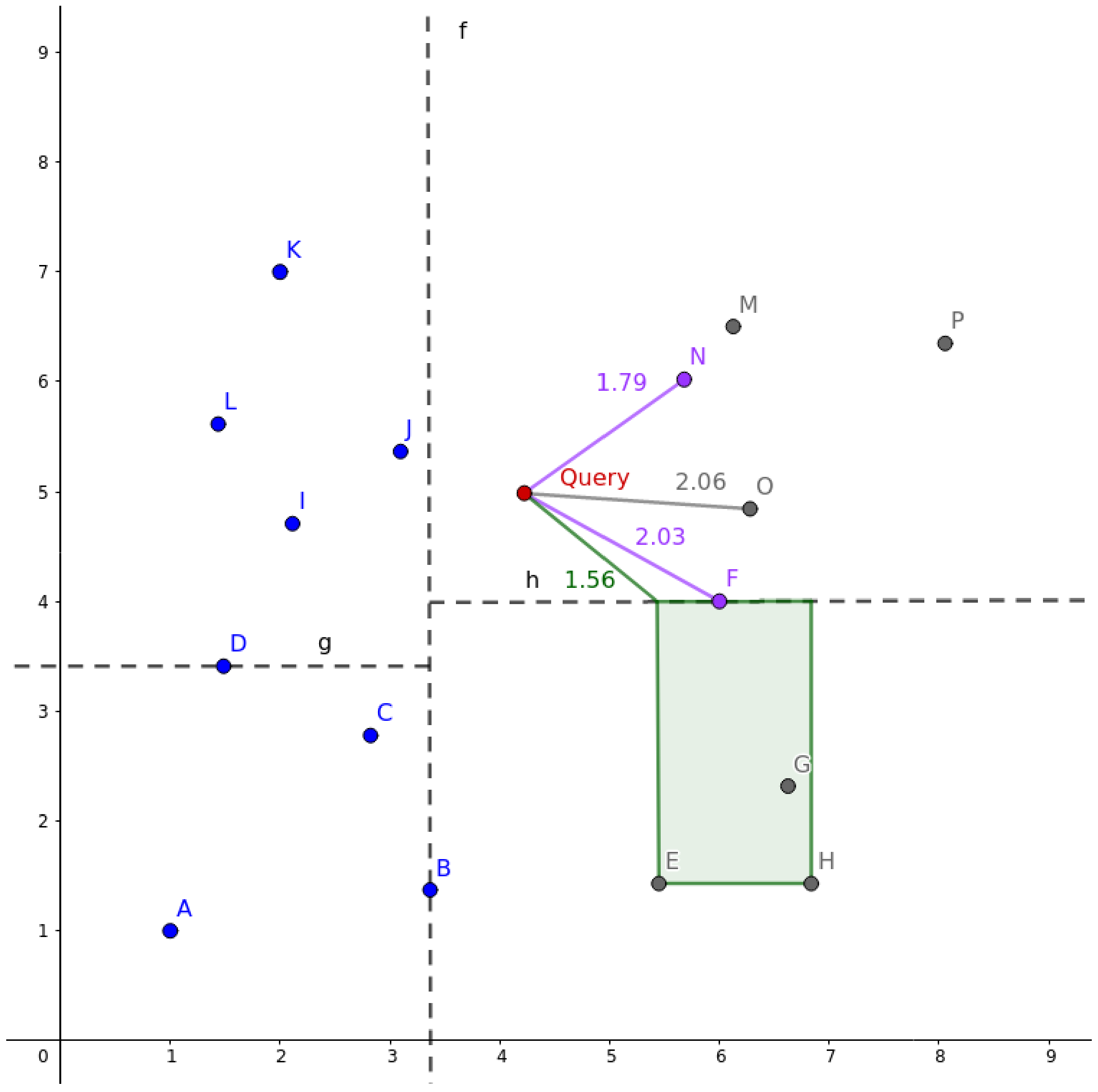


Figure 7: This example continues where figure 3 left of. The algorithm considers the neighbor of the query's natural leaf, which is the EFGH leaf, but this time, it compares its words neighbor candidate to the boundary-box of the leaf, rather than its median (h-line). It makes the decision to visit it, and O gets replaced with F as the new worst neighbor of the query.

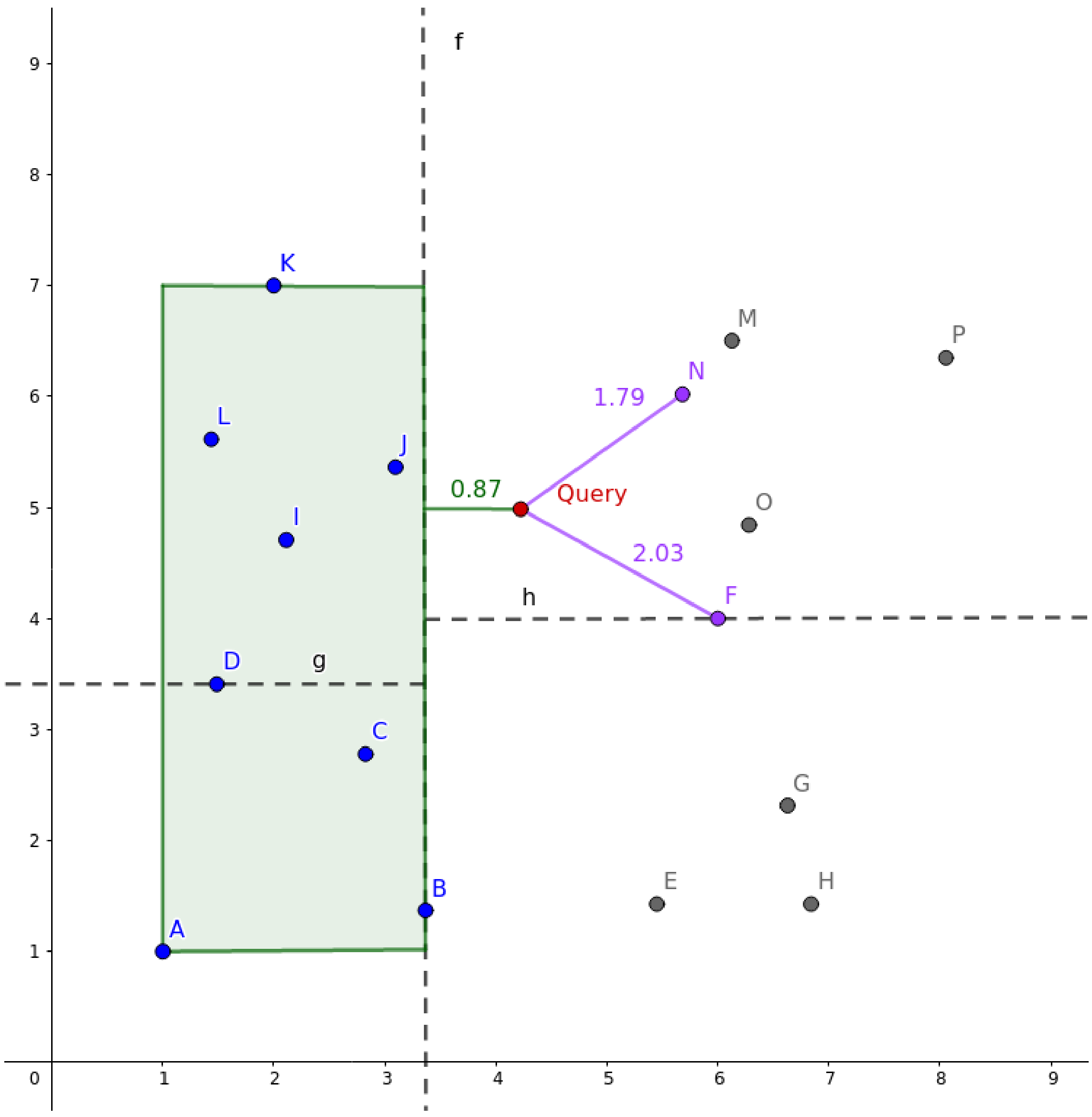


Figure 8: After having visited both right leaves, the traversal returns and considers other child of the root node. The distance to the box is the same as the distance to the median, because the query is 'within' the box, with regards to the y-dimension, though it is outside the box in the x-dimension. It chooses to visit the box, and goes to the natural leaf inside it, which is the IJKL leaf.

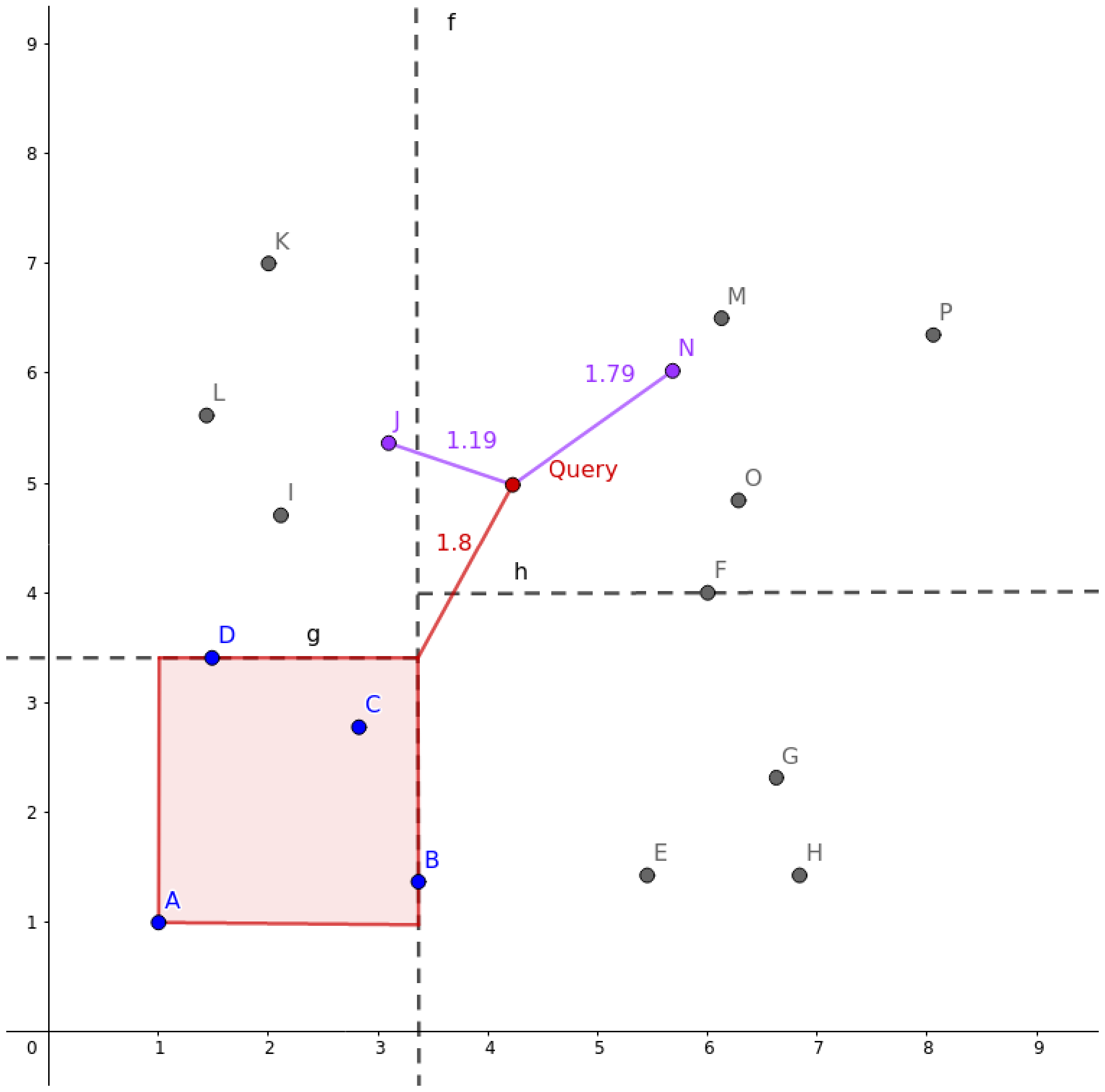


Figure 9: After having visited and processed the IJKL leaf, the query's nearest neighbors have been updated again. It now considers the sibling leaf of the IJKL leaf, which is the ABCD leaf. This is where it decides against visiting this leaf, because the distance to the box is greater than to its nearest neighbor. This means that no time is wasted on processing this leaf. Recall that this leaf was processed when the example was given using the median-test.

While testing only the medians in a given node forces us to compare the queries to half of $\mathbb{R}^d$ every time, thus giving many false positives, this new method compares with an increasingly shrinking subspace, as illustrated previously by the highlighted boxes. The median-test considers only one median, while bounds-test considers the distance from the query to the bounding box, hence the latter is more exact. To see the effect of this optimization in action, please see section 7.

6 Implementation

In this section, the implementation will be gone through in appropriate detail. The futhark-code will be trimmed a bit, to bring attention to the parts that use regular nested parallelism. For the full code, please refer to the futhark files in `src/` in the materials, and the appendix 11. We will start by explaining the big picture – the main algorithm– and then go into more details on the component parts. For the purpose of highlighting the most interesting parts of the code itself, some parts of the code are removed from the examples. This is denoted by '...'.

6.1 Main algorithm

```
Receives:
  Lower bound on leaf-size: leaf_size_lb
  Number of desired nearest-neighbors: k
  Set of testpoints: P
  Set of querypoints: Q
Returns:
  For every query q, k indices into P of nearest neighbors for q
Main algorithm:
  0. Pad P
  1. Build kd-tree
  2. Find the natural leaf for every query
  3: Sort queries according to the leaf indices
  4. While there are queries still being processed:
    a. Brute-force each query on its current leaf
    b. Traverse the tree once, with all queries, to find the next leaves
    c. Partition the queries, to separate those that are finished
    d. Update global knns for the queries that have finished processing
    e. Keep only the ongoing parts of knns, leaf_indices, stacks, Q_inds, Q
    f. Sort the queries according to the leaf indices
```

Figure 10: This pseudocode shows the overall structure of the main algorithm, corresponding to v5. This version was chosen because it gives the best impression of the overall algorithmic structure. More on versions section 7.

The final version, v7, is shown in full in section 11, in the Appendix.

The following goes through and explains the different sections of the code. Also not shown are initializations of arrays, and declarations of constants.

- l2: The set of points P is padded using the `pad` function. Not shown are various sizes that are determined during this process.
- l3: The padded set of points is used to construct the tree, the bounds-structures and the leaf-structure. Verbose details are hidden.
- l4: A `map` takes every query and applies the `find_natural_leaf` function to it, resulting in an array of the naural leaves for all queries.
- l5-6: The leaf-indices are sorted, and the ordering of the sort is recovered. This is done by zipping the leaf-indices with an array where each element is its own index, giving an array

of tuples. The function `iota` takes an integer `m` and returns an array of `m` integers from 0 to $m-1$. This is then sorted by the first element, which is the leaf-index. These are then unzipped, turning the array of tuples into a tuple of arrays.

By sorting this along with the leaf-indices, we get an array where each element effectively answers the question 'at what index were you originally'. This is a common futharkism to apply the same transformation to multiple arrays, without involving them in the sorting itself, thus improving performance.

- l7: The queries are sorted by `gather`ing them using the `sort_order` indices. `gather` takes an array of indices and an array of elements, and uses the indices to pick elements from the latter, effectively applying the transformation embodied in the indices.
- l8: `sort_order` now corresponds to the ordering of the queries, so they just need to be aliased.
- l9-11: The loop which processes the queries starts. It keeps going untill all the leaves are done, which is identified by the number of leafindices still in use.
- l12-15: `knns` are updated by using the `bruteForce` algorithm. Each query is paired with its knn and leaf-index by `map`ping over all three at the same time.
- l16-19: The tree is traversed by every query, and new leaf-indices are found. The stacks get updated in the process.
- l20-21: The indices of those queries that are done processing are identified from the rest, by partitioning the indices of the leaf-indices.
- l22-25: The indices of queries that are done, is used to write the final versions of their respective knn's to the result-collection.
- l26-33: The partitioning of the arrays, and the sorting of the elements that continue by their leafindex, is combined into one step.
- l26: We gather the leafindices of those queries that continue into the next iteration.
- l27-28: The trick used on lines 5-6 is repeated, recovering the sorting order.
- l29-33: The sorting order is used to reorder the indices of the elements that continue. This means that when they are gathered, not only are the elements that continue selected, but they are also placed into sorted order, effectively partitioning and sorting in one step.
- l34: The iteration finishes.
- l13-35: After the loop has finished processing, the construct 'res' contains the arrays that were looped over in their final state, and after line 35 the results are extracted from it and returned. This last part is not shown.

6.2 Padding the reference points

The padding-heuristic works as follows:

- Start with P and a given desired lower bound on how big leaves should be.
- Find out how many leaves, of that size, can be filled completely using P. Call this number `num_default_leaves`. This will be some arbitrary number, but we want the total number of leaves to be a power of two, because that is requried for the resulting tree to be a balanced, binary tree.

- Find the largest number, that is a power of two, and that is less than or equal to `num_default_leaves`. Call this number `num_leaves`. This will be the actual number of leaves in the leaf-structure, below the top tree.

- If we just used the initial leaf-size, some points would be in excess, so we increase the leaf-size untill they can all fit it. This will likely leave a gap in some leaves, because there is no guarantee that the excess points can be evenly distributed. Some leaves might have one less point than others, even if they are distributed as evenly as possible.

- Add however many dummy-points as is needed to balance the tree out. This number will be less than the number of leaves in total, since, if the actual points could evenly distributed, there would be no need for padding.

This heuristic ensures that (1) the number of padding-elements will be less than the number of leaves, and (2) the leaf-size will be at least the lower bound, and less than twice the the lower-bound, since if there were enough points to double the leaf size, they would instead have been distributed to twice as many points.

Padding is done with arrays of the highest `32-bit float`, that is a an actual floating-point number, and not infinity. This number is defined in `constants.fut`.

6.3 Constructing the kd-tree in parallel

This section will go into more detail about the construction of the kd-tree and the leaf structure in practice.

The bounds-structure is also constructed together with the top-tree. For every node in the top-tree *and* every leaf in the leaf-structure, it contains two pseudo-points, lb and ub.

These correspond to the lower and upper bounds for all the points in that node, for every dimension. For a node/leaf containing the points $(1,2),(9,3),(2,5)$, the lower-bound would be $(1,2)$, and the upper-bound would be $(9,5)$. This is then used while traversing, to decide whether to visit sibling-nodes, as shown in the contributions section.

In reality, the kd-tree is two arrays, one of medians and another of split-dimensions, and the bounds-structure is two matrices, one of upper-boundary points and another of lower-boundary points.

Parallelizing the kd-tree construction is one of the contributions of this project. At a high level, this is done in the following way:

```
Build kd tree:
  0 Initialize empty tree and bounds-structure
  1. While not all levels of the tree have been created:
    a. Determine the size and number and of segment/nodes in the current level
    b. For each node, determine the upper and lower bounds for each dimension
    c. Pick the split-dimension of highest spread
    d. Sort the points in the node according to that dimension
    e. Save in each node on the tree the median and the split
    f. Save in the bounds-structure the bounds for each node/segment
  2. Create the leaf-structure by reordering P in accordance with the tree
  3. For each leaf, determine the bounds and save these in the bounds-structur
```

The code for the function that does these things is given below in a reduced form, to highlight the parallel structure of the approach. To see the full code, please have a look in the `kd-tree-common.fut` file in the source folder. This is probably the most technically complicated part of the whole project.

Here follows an explaination of the shown part:

- l1: The function receives P (which has been padded), and a number of sizes from the pad function, which are not shown.
- l2: Initialization of the data structures is not shown.
- l3-5: The loop begins. It runs one iteration for every level in the tree, which is the height plus one, since the height is the number of edges on the path from the root to the deepest leaf node.
- l6-7: At the first iteration of the loop, all points are in the same 'segment'. Each segment should be thought of as the points contained in a given node. At each iteration, the segments are halved in size while the number of segments double, thus reflecting the way that each node has two children which each inherit half of its points. These sizes are calculated here.
- l8: The indices of P are shaped into a matrix, such that each inner vector corresponds to a segment of points.
- l9-l14: Each segment is gathered from P via its previously identified indices.
- l15-31: The split-dimensions (`dim_inds`) and lower and upper bounds are identified for each segment, by mapping over the indices:
- l17: The segment is transposed. This means that instead of each row containing a point, each row now contains a 'dimension' for all the points.
- l18-20: A vector containing the minimal values for each dimension of the points of the segment is found, by reducing each row (each of which contains the values of a given dimension, for all the points) with the binary function `f32.min`, which is self-explainatory. This is the lower bound for the segment.
- l21-23: The same is done with the `f32.max` function to find the upper bound for the segment.
- l24: An array containing the differences between the bounds, for every dimension, is found by mapping them with the subtraction function.
- l25-29: The index of the dimension of highest spread is found by reduction of the differences and their indices with a lambda function. This function takes two tuples of type `f32, i32`, and returns the tuple with the greatest f32 value. This way, we can identify the index of the dimension with the highest spread.
- l32-37: A matrix of value-index tuples is created. Each row in the matrix corresponds to a segment. Each tuple consists of the global index of one of the points in that segment, and its value in the dimension previously identified, with respect to the segment it is in, because each segment has its own split-dimension. This tuple-matrix is created so that the points can be sorted in the next step.
- l38-43: Here we use a library function `batch_merge_sort` to sort the rows of the tuple-matrix from before. Sorting the segments, according to their value in the split-dimension, in this way, is more efficient than sorting within a map. The lambda-function is an argument to the sorting function which tells it how to order the tuples.

- l44-48: This part finds two things: `t_inds`, which is the segment of infices with offset that we want to write elements to, in both the kd-tree (dims and meds) and the bounds structure (global lbs and ubs). The other part is recovery of the medians from the newly sorted segments. For each segment, the median that its future children will be split by, is found by line 45. The median is the last value of the first half of the segment, or the last value of the to-be left child of the given segment/node.

- l49-53: The new values for the tree and bounds-structure are scattered into it at the indices we just found. The segmentation of the indices of P is undone by 'flatten', so that it is ready to be reshaped in the next iteration. Arrays must have the same dimensions across iterations, so this is a necessity.

The parts after the main loop consist of a reordering of some elements, the filling of the leaf-structure, and the propagation of the bounds for the leaves into the bounds structures. This part is very similar to the way it is done within the loop. After these things, the function returns the structures.

6.4 Explaination of the implementation of the brute force algorithm

The brute force implementation consists of three functions that depend on each other. The second one, `bruteForce`, is designed such that it can be used both for the pure brute-force algorithm (`runBF`), and as an element of the kd-tree approach. These work as follows:

- l1: The `update_knn` function receives a collection `knn` corresponding to a querys current nearest neighbor candidates, which consist of a referencepoint index and a distance from the reference to the query. It also receives a new element for knn, which it is to insert sorted into the list.

- l2-3: The function iterates over each index in the knns. Paradoxically, the first line is the last thing that happens. It pipes the result of the loop, which is a tuple, into a function which picks out the first element, which is the result.

- l4-5: If the element is not closer to the query than that at the current index, the iteration ends with no effect.

- l6-9: If the element is closer, it is swapped into the current index in the knn array, and that which was there before is used in the next iteration. Whatever element is left over at the end is discarded.

- l11-15: The `bruteForce` function receives a collection of refrence-points and a query. It also receives a leafindex, which is used for an offset. This way, it can be used either on a leaf, or on the whole collection, as seen later.

- l16: It loops over every reference point, updating knn for each one.

- l17: It finds the distance from the referencepoint to the query.

- l18-19: If the distance is greater than or equal to, that from the worst of the nearest neighbors, it does not modify the neighbors.

- l20: If the distance is shorter than that of the worst of the nearest neighbors, it updates knn using this newly found better neighbor.

- l22: The `runBF` function receives the total set of references and queries, and this is the reference for the speedups of the main algorithm.
- l23: The knns are initialized like in the main algorithm (not shown).
- l24-25: `bruteForce` is ran on each query and on the whole collection of reference-points. The leaf-index given is 0, effectively treating the whole collection as if it were the first leaf. The result is then picked out afterwards.

6.5 Query propagation

6.5.1 Finding the leaf to which the query naturally belongs

This algorithm is part of the traversal process. It is used initially, before the main loop, to find the natural leaves of the queries. It is also used later on, as the downwards stage of the traversal, that is done inside the main loop of the main algorithm. This function is quite simple, and just traverses downwards for one query, by choosing the side of each median, that the query natually fits in. It is used inside a `map` to apply it to every active query at the same time in parallel.

It works in the following way:

- l1-5: The function receives a starting index into the tree. This is just 0 for the root in the pre-loop case in the main algorithm. It also receives the query to traverse, and the the kd-tree itself.
- l6: A loop is entered that continues while the index is still inside the tree. It will end when it has climbed downwards into a leaf.
- l7-9: If the value of the query is greater than the median in the node, with respect to the split-dimension of the node, the index for the next iteration is given by the `getRightChild` function. Otherwise, the `getLeftChild` function is used.
- l10: After the index 'exits out the bottom' of the tree, we adjust it so it becomes a leaf-index. Conceptually, the leaf-structure can be imagined as the last level of the tree, and if it was, the index would be directly useable. Since it isn't, this translation is necessary.

6.5.2 Completing one traversal of the tree, for one query

The function responsible for completing one traversal of the tree, in order to find the next leaf to be visited, is called `traverse_once` in the code. It is shown here in a slightly different structure, in pseudocode, to illustrate it.

An explaination of the parts is given here:

- l9: Translates the leafIndex into an equivalent treeIndex
- l10: Initializes the recursion node to -1.
- l11+17: While no new recursion node has been found, and the upwards traversal has not yet reached the root, it will keep climbing the tree.

- l12: Checks the stacks to see if it has been marked. This means it has been set to 'true'. This is the way we distinguished whether we are coming up from the natural or secondary child, so we can decide whether to perform the test on the sibling or not. As mentioned earlier, this saves some memory accesses.
- l13: If we have visited both siblings, the mark on the stack is cleared.
- l14: If we havent visited both children yet, we test whether the sibling of nodeIndex might contain better nearest neighbor candidates. This is where either of the tests is used.
- l15-16: If we perform the test and decide to visit the sibling, we mark that both siblings have been visited on the tree, and then we mark the sibling as the recursion node, so the loop halts.
- l17: No matter what, climb the index once up in the tree. In case we were going to the sibling, this wont have any effect, in the other two cases, this is what we want.
- l18-19+22: If no recursion node was found and the loop stopped, this means that we reached the root and that everything has been either processed or rejected by the test. We return -1 to the main algorithm to denote that this query is done processing.
- 20-22: If we found a recursion node, we call the same function as was used to initially propagate queries to their natural leaves, except the query is given the starting-point into the tree, which is this recursion node we have just identified. This function then returns a new leafIndex into this subtree, which is then returned.

Depending on whether we use the median-test or the boundary-test, the traversal will of course differ.

6.5.3 Explaining the bounds-test calculation

The construction of the kd-tree also returns a bounds-structure, as mentioned previously. This bounds-structure consists of two arrays, each containing points denoting lower and upper bounds, for every node in the tree and for every leaf in the leaf structure. These bounds mark out a box, which contains all the points in the node or leaf. We are going to use these bounds to find the shortest euclidian distance between the query and the box. This pseudocode computes the minimum distance from query to box.

```
acc = 0
for every i dimension in d:
  temp = 0
  if q[i] > ub[i]
    temp = q[i] - ub[i]
  else if lb[i] > q[i]
    temp = lb[i] - q[i]
  acc += temp*temp
if sqrt(acc) > wnnd
  visit the sibling
```

For every dimension, it checks whether the query is 'above' the upperbound, between them, or below the lower bound. If it is above or below, it adds the distance squared to the accumulator for the respective boundary, and if it is between them, nothing is added. Finally, the square root is taken, to get the euclidian distance, and this is compared with wnnd, i.e. the distance from the query to the worst (most distant) of its current nearest neighbor candidates.

7 Experiments

7.1 Setup

How to run various things is explained in the `README.md` file in the github repository. The server used to perform the tests was DIKU's `gpu04-diku-apl`, which has a *Geforce RTX 2080 Ti*.

7.2 Iterative optimizations

During the development of the code, various optimizations were discovered. Each version after v1 consists of an optimization to the previous version. This way, the contribution of each optimization can be assessed. Most of the optimizations consists of improvements to the way sorting is done. The following parts will highlight interesting performance impacts of each version.

7.3 Performance

In order to measure performance, various benchmark datasets were defined. See the files in the `benchmarks` folder for an example. Futhark has its own built-in way of defining and running benchmarks, and this was used. These datasets are randomly generated by Futhark and have a defined dimensionality and size. Each dataset consists of equally-sized Q and P, with some dimensionality. The size of k was set to 5 unless described otherwise. Performance is shown in terms of speedup versus brute force. If bruteforce took 10 seconds and a given version took 1, it would be a speedup of 10. The tables for all these can be seen in the file `benchmarks.ods`. The following parts will highlight the most interesting results. Varying k did not turn out to have very different impacts on the different versions, and for this reason, no graphs of it will be shown. As k moved away from 5, performance generally decreased for every version, in comparison to brute force. The results for this is also in the datasheet.

Bruteforce runtimes

n,m	d5	d7	d9	d12	d16
131072	1139526	1307229	1969961	2091046	3183972
262144	4738751	5522700	14213036	8101586	13985100
524288	18219441	22787605	42801905	29435870	59340592
1048576	73094720	90882295	113012009	142778495	226237680
2097152	289017747	387764743			

Table 1: This shows runtimes in μs for the brute-force implementation, for $k = 5$. These are the runtimes that the speedups are in relation to.

v1: Baseline

This is the algorithm as described in the pseudocode in the algorithm section, without sorting and with the old test.

v2: sorting

This step adds the sorting-optimization, as denoted in the algorithm section. More specifically, the array containing the leaf-indices are sorted while an array which denotes the transformation corresponding to the sorting is created. This is then used to gather the arrays containing the tentative knn's, the indirect array to Q, and the array containing the stacks. This does not sort Q itself, but only all the associated data. It turns out that this by itself does not improve performance. Sorting is done using radix-sort.

v3: complete sorting

This version also gathers Q itself using the sort-order array as described for v2. It only keeps the active queries in memory and discards those that are completed.

v1-3 comparison

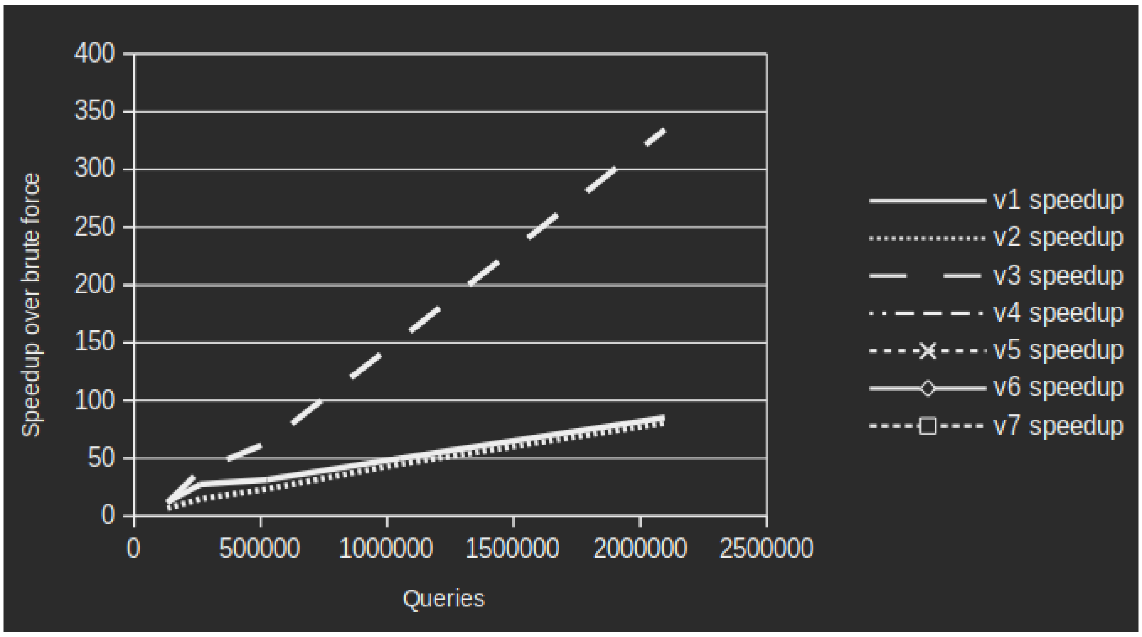


Figure 11: Results for d=5, k=5. v2 turns out to not be an improvement over v1, but rather a loss of performance, while v3 takes of from both of them. This shows that sorting Q itself is the central factor.

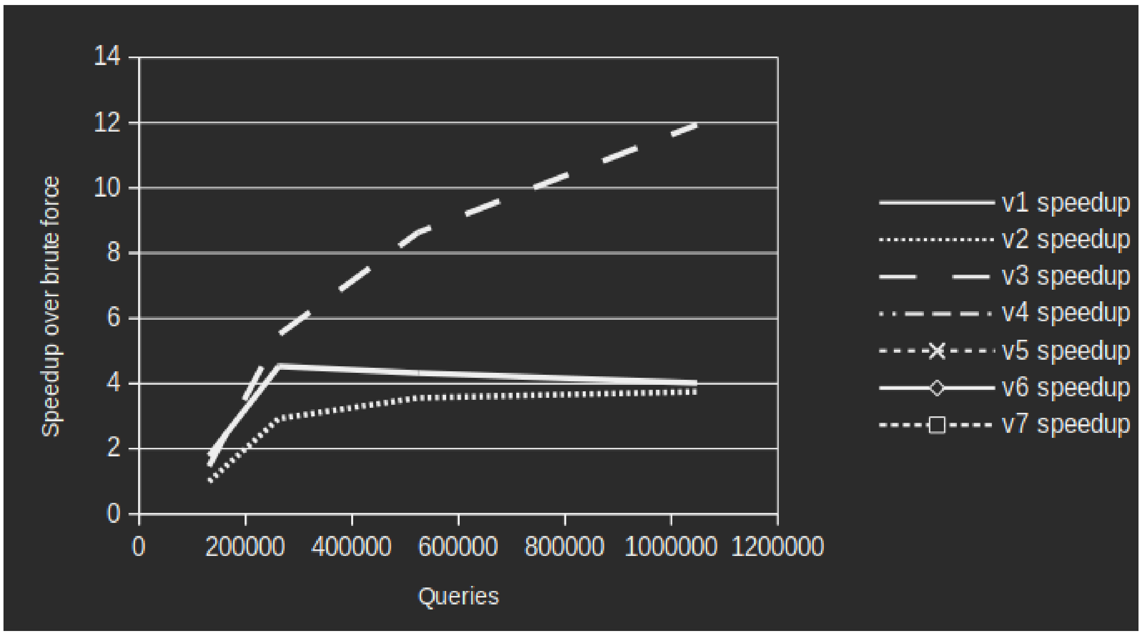


Figure 12: Results for d=9, k=5. As d increases, their speedup decreases drastically.

v4: optimizing radix-sort

Optimization of radix-sort is done in this version. Since a valid leaf-index can only be between 0 and `leaf_count-1`, performing radix-sort on all 32-bits is excessive. The number of bits used to represent all valid leaf-indices is `height+2`, so no leaf-index uses more bits than that. The optimization is for it to only sort this number of bits.

v5: using the more precise test

This version swaps out median-test with the boundary-test, as explained in section 5.

v3-5 comparison

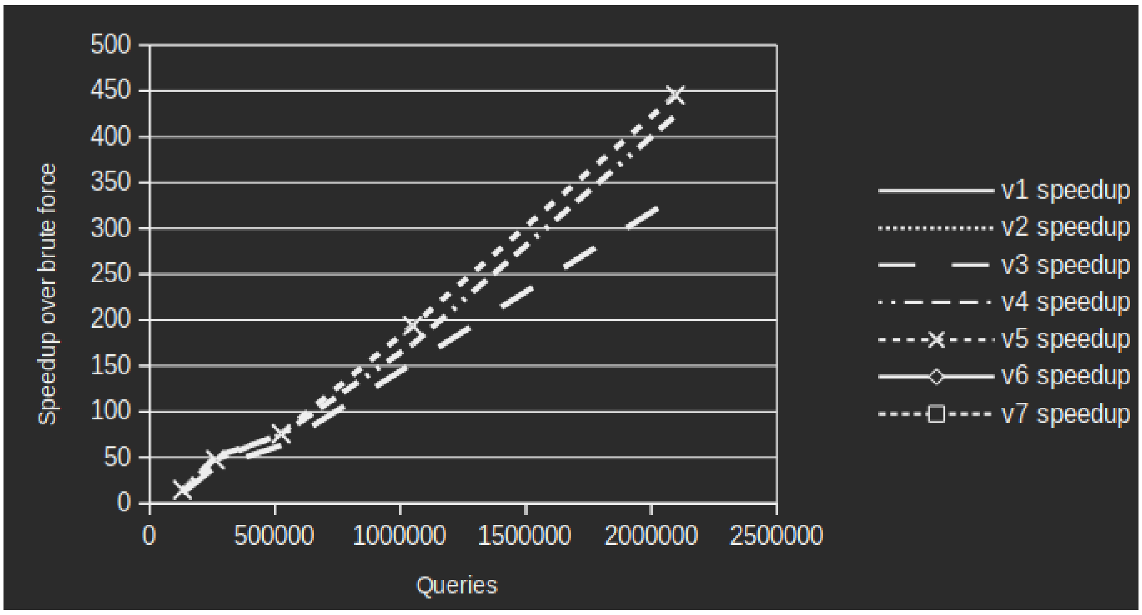


Figure 13: Results for d=5, k=5. Optimizing radix-sort in v4 gives a noticeable improvement over v3. The new bounds-test introduced in v5 also gives a speedup, but its not very significant.

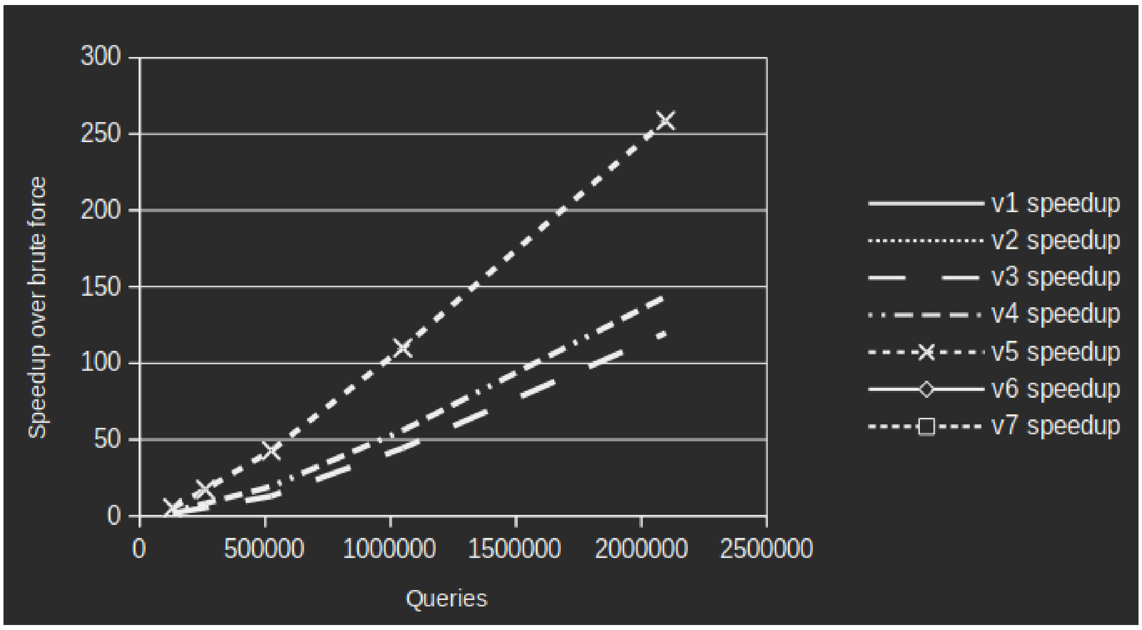


Figure 14: Results for d=7, k=5. As d increases to 7, less performance is lost for v5 than for v3 and v4, as indicated by the growing gap, suggesting a better scaling with dimensionality.

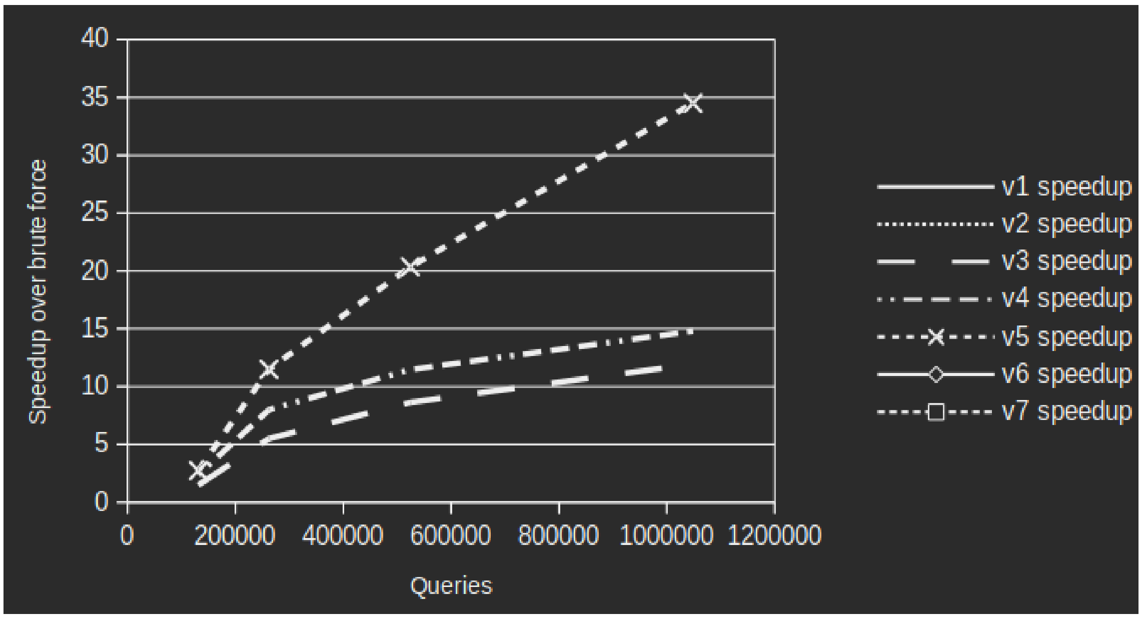


Figure 15: Results for d=9, k=5. Same tendency is seen as with d=7, of v5 gaining a lead.

v6: combining partitioning and sorting

The step of partitioning the queries after the traversal is combined with the sorting step. This is done by denoting a query as having finished processing by `leaf_count` rather than -1, and then sorting them. This way, those that are done collect in the upper end of the array. Radix-sort is modified to use `height + 3`, which allows sorting the done queries together with the rest. The queries that are done are then counted, and that number is used to determine where to split the arrays. This is a code-level optimization.

v7: using a treshhold to stop sorting

A threshhold-test is added to the main loop, to stop iterating when the number of active queries per leaf goes below a certain point. An identical loop – without sorting – is then started, to finish the last queries. The treshhold was found to be around 1. The point of this optimization is that when the number of queries per leaf is low, the benefit of having many queries process the same leaf goes away, and it becomes beneficial to stop sorting.

v5-7 comparison

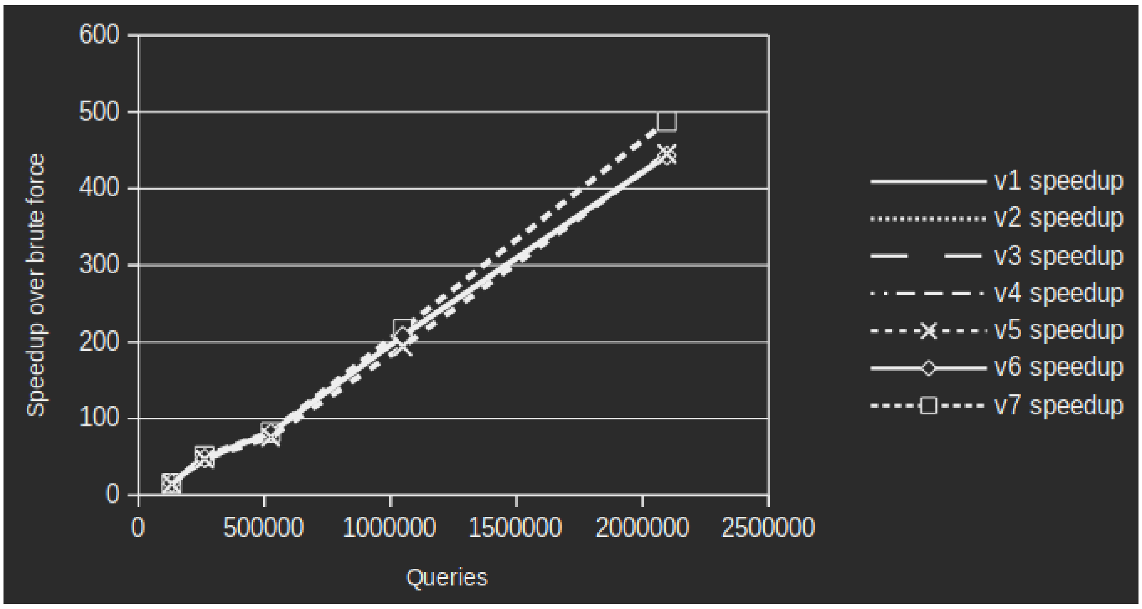


Figure 16: Results for d=5, k=5. No significant gain for v6 over v5. v7 gives a slight speedup. Note that they approach 500x speedup for around 2 million queries.

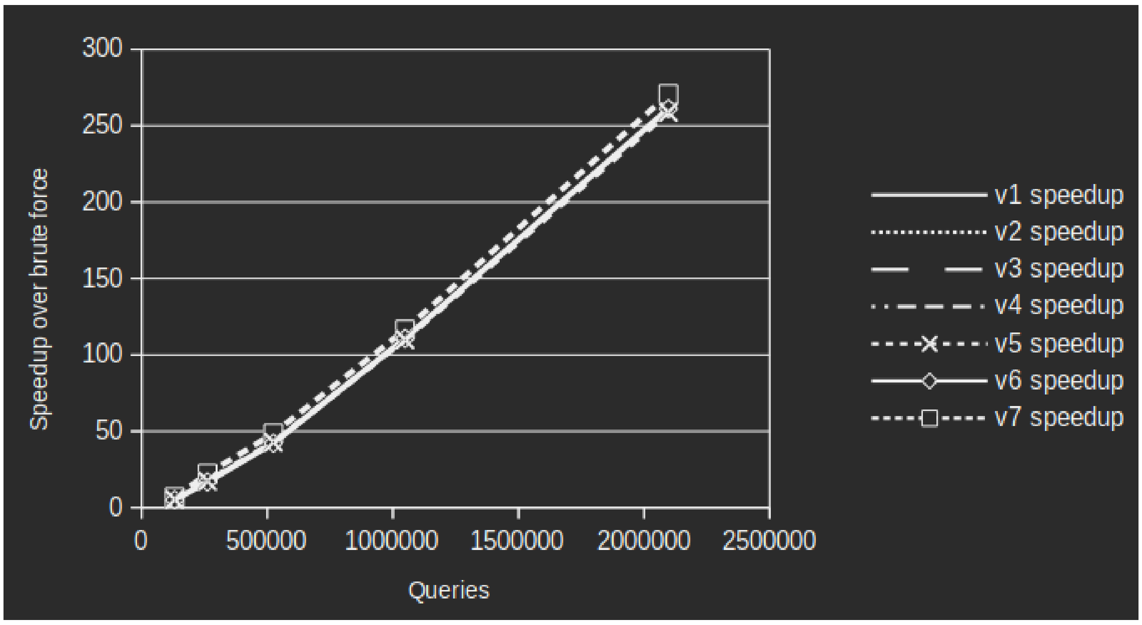


Figure 17: Results for d=7, k=5. Same as previous case.

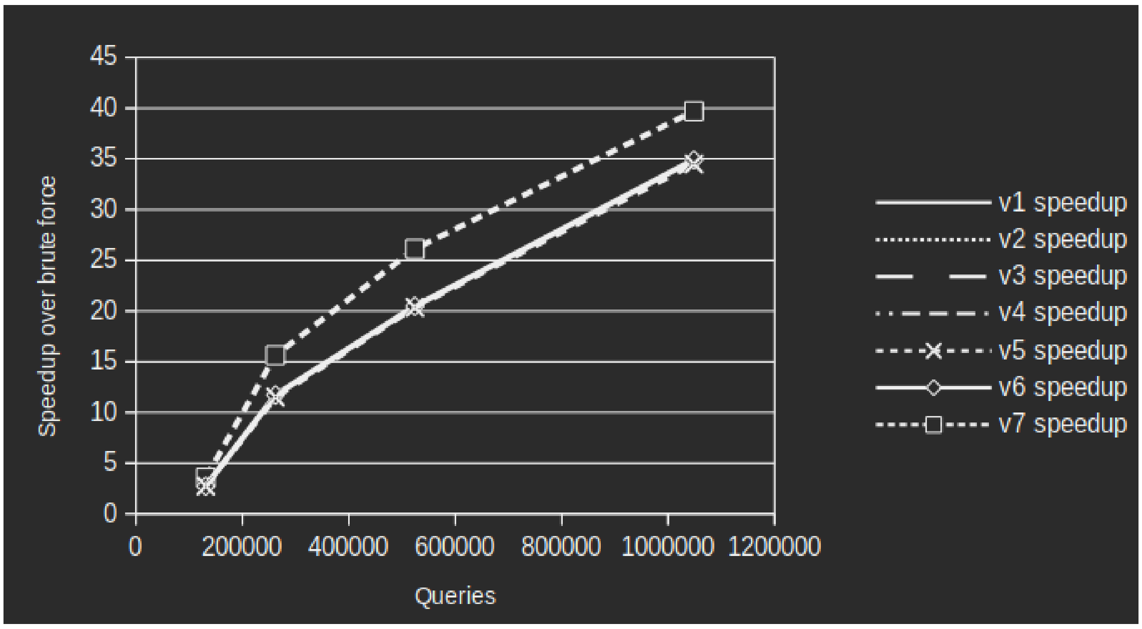


Figure 18: Results for d=9, k=5. As d grows, the benefit of v7's optimization starts to become more noticeable. This implies that v7's optimization helps the algorithm scale better with dimensionality.

7.4 Qualitative comparison of the median-test and boundary-test

In order to compare how the two tests influence the traversal, we want to compare how fast the queries finish under the two methods. A test was set up with one million points in Q and P, and 5, 10 and 20 dimensions. This meant that there were 2048 leaves in the leafstructure. Because every active query visits a new leaf in every iteration, there can be at most 2048 iterations.

The area under the graphs represents the computational resources occupied during the computation. The difference in area between the curves correspond to the resources freed up by using one over the other.

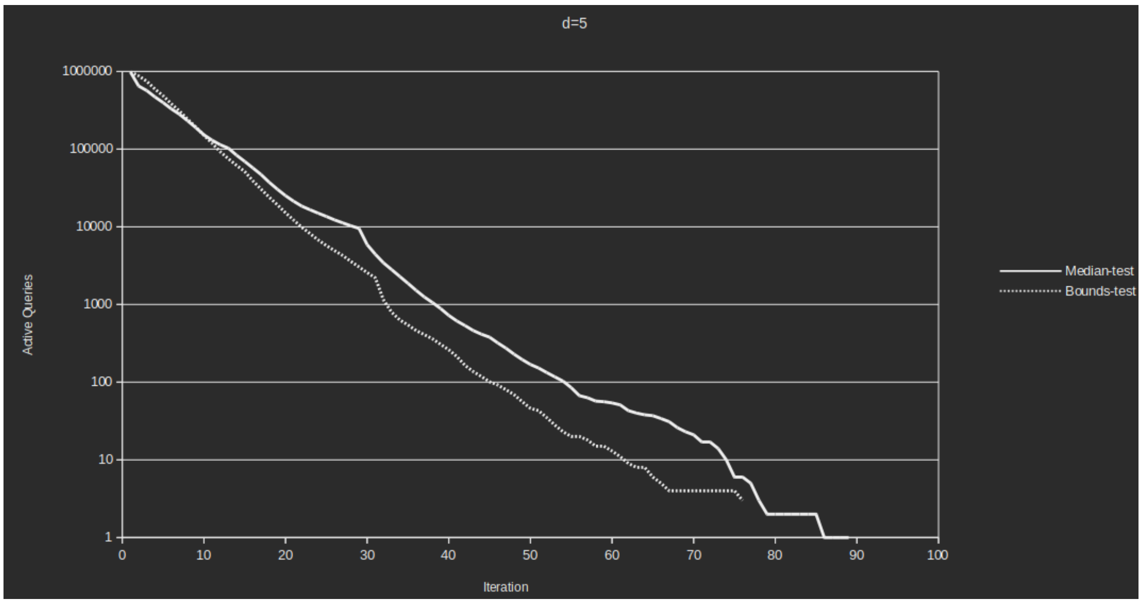


Figure 19: Results for d=5. The bounds-test does not improve significantly over the median-test. This is consistent with the speedups for d=5.

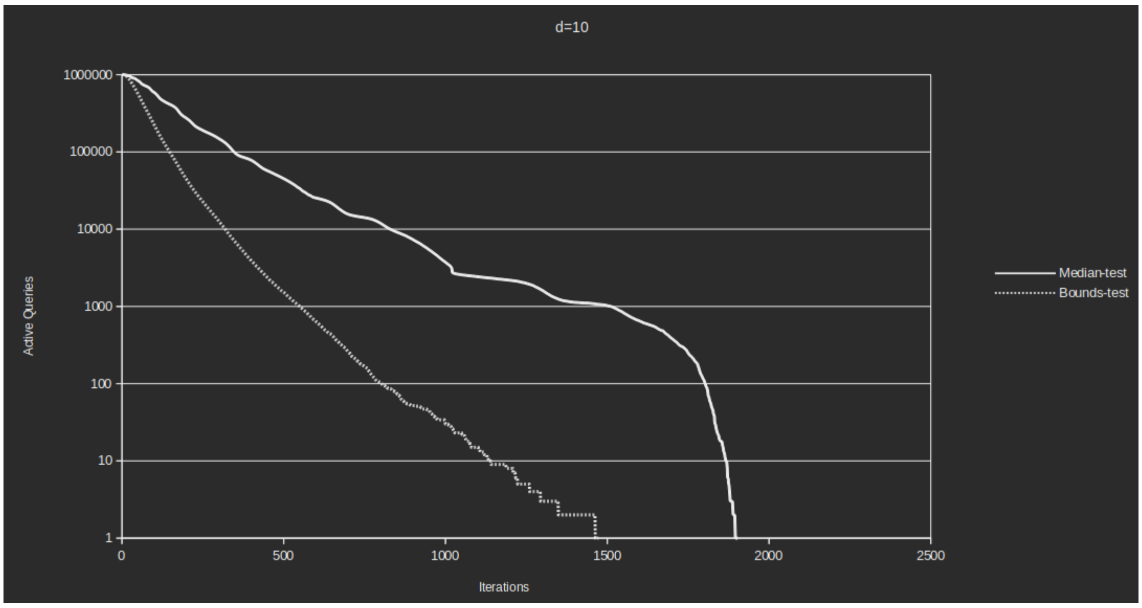


Figure 20: Results for d=10. At this point, a remarkable difference is noticeable, hinting that the bounds-test helps the algorithm scale better with d.

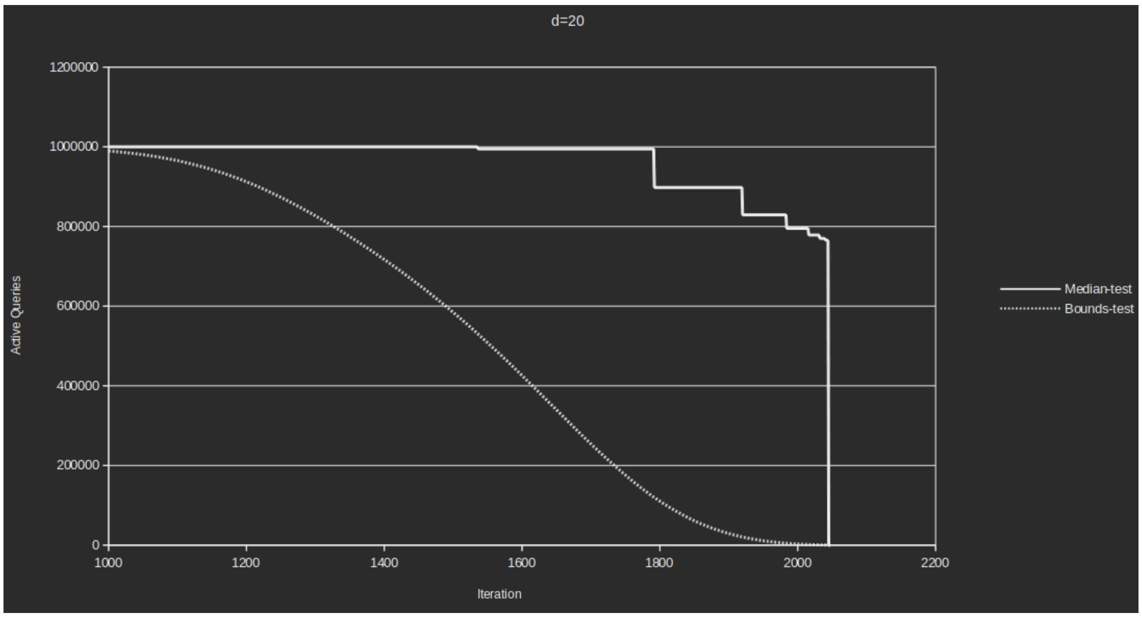


Figure 21: Results for d=20. The y-axis is not logarithmic in this case, and the x-axis starts at 1000. At this point, the median-test is hardly better than brute force.

All the benchmark runningtimes and associated speedups are in the datasheets in the materials submitted along with this report. For low d's, we only see a small improvement for one million queries, and they both finish in about eighty iterations. For d=10, however, there is a much larger difference, and it takes about 500 more iterations. For d=20, the old method only manages to finish about 20% of the queries before the last iteration, making it essentially solve those by brute force, while the new method releases about half of its total resources over the last half the iterations, if we conceptualize the resources as the area under the graph.

This directly implies that the new method scales better with dimensionality, because while they are very comparable for d=5, a gap develops as d increases, although it still suffers from the curse of dimensionality. In every case, the new method outperforms the old. The files associated with this test are all located in the `/qualitative_tests` folder.

In section 4, it was mentioned that hopefully only $O(log_2(n))$ reference-points would be processed by each query. The total number of 'leaf visits' is the sum of the histogram plus 1 million, because the number of active queries is recorded after processing occurs. There turns out to be about 170 leaf-visits on average for $d = 10$ using the median-test and 73 for the boundary-test. This seems to scale as expected in both cases, taking into account that each lefa contains 256 points. For $d = 5$ the average number of visited leaves is 6.8 and 5.6 respectively. This is not terrible, considering there are 2048 leaves, and it is in line with the $488x$ speedup observed.

7.5 Discussion

n	baseline	+ sorting	+ new test	+ final parts
131072	1.8	2.4	2.7	3.5
262144	4.5	8.0	11.5	15.6
524288	4.3	11.5	20.3	26.1
1048576	**4.0**	**14.8**	**34.5**	**39.7**

Table 2: This table shows speedups for d=9, k=5. Sorting has an overall impact of a factor of about three to four over the baseline. The new boundary-test has a futher impact of a factor of about two over the version with sorting. The final two optimizations give some futher speedup. If we extrapolate from these figures, sorting is the most impactful optimization overall, followed by the boundary-test. When d is smaller, sorting has an even larger impact relative to the new test.

7.6 Validation

Benchmarking itself does not prove correctness, but merely measures the runtime of the algorithm. In order to argue for the correctness of the results, we will use futharks built-in functionality for validation. `futhark test` and `futhark bench` both have a similar capacity for this, provided the data-sets have predefined reference-outputs to compare against. Because it was desired to test with as many points as possible and as high dimensionality as possible, validation for every benchmark was not done.

We decided to test against a wide set of smaller datasets, to check that various edge-cases would be handled correctly. This was done by using the output of the brute force implementation as a reference for tests of the other versions. All of these validate, and to run them, one must first enter `make setup_tests`, followed by `make tests`, to use the built-in `futhark test` for every algorithm, for every test-case. The datasets used can be seen in the Makefile where they are generated using the `futhark dataset` tool. These range from single queries, to many queries and very high dimensionalities.

8 Conclusion

Various contributions to how kd-trees are used to calculate knns have been described, including parallelizing the construction of the kd-tree and assocaited datastructures, sorting queries according to the leaves they last visited, and using a boundary-test to reject more false positives during the traversal. The implementation of these things have been described, and a number of iterative optimizations have been described and benchmarked. For datasets of about two million reference points and queries, for $k = 5$ the final version achieved a speedup, compared to brute force, for $d = 5$ of 487.6. In a case with a million references and queries, a speedup 40 for $d = 9$, and of 1.8 for $d = 16$. A qualitive test showed that for $d = 10$ and for a million queries, the boundary-based test made the queries visit about half as many leaves on average, as the median-based test. Finally, a test of the algorithms correctness was described, which compared its results with that of a pure brute force implementation, which was then assumed to be correct.

9 Future work

9.1 Code-level optimizations

Future work kan be done on discovering the a more calculation for the threshhold in `v7`, and how this is affected by `d`. The default lower bound for leaf-size is 256, but this could be optimized with more testing.

9.2 Algorithm-level optimizations

A further investigation of possible approaches to trim the search space as much as possible is warranted. It is likely that a convex hull would be both too irregular a datastructure, and too bulky in high dimensions, to amortize the cost of constructing and using it. It would be interesting to see if the semi-convex hull method employed in [1] would be an improvement over the boundary-test, and under what circumstances, or if some similar hybrid approach can be discovered.

10 References

[1] Y. Chen, L. Zhou, N. Bouguila, B. Zhong, F. Wu, Z. Lei, J. Du, and H. Li, Semi-convex hull tree: Fast nearest neighbor queries for large scale data on gpus (2018), 911–916.

[2] Y. Chen, L. Zhou, Y. Tang, J. P. Singh, N. Bouguila, C. Wang, H. Wang, and J. Du, Fast neighbor search by using revised k-d tree, *Information Sciences* **472** (2018).

[3] Y. Chicha, M. Lloyd, C. Oancea, and S. M. Watt, Parametric Polymorphism for Computer Algebra Software Components, *Proc. 6th International Symposium on Symbolic and Numeric Algorithms for Scientific Comput.*, Mirton Publishing House (2004), 119–130.

[4] M. Elsman, T. Henriksen, D. Annenkov, and C. E. Oancea, Static interpretation of higher-order modules in Futhark: Functional GPU programming in the large, *Proceedings of the ACM on Programming Languages* **2**, ICFP (2018), 97:1–97:30.

[5] F. Gieseke, J. Heinermann, C. Oancea, and C. Igel, Buffer k-d trees: Processing massive nearest neighbor queries on gpus, *Proceedings of the International Conference on Machine Learning, ICML* **1** (2014), 172–180.

[6] F. Gieseke, S. Rosca, T. Henriksen, J. Verbesselt, and C. Oancea, Massively-parallel change detection for satellite time series data with missing values, *Procs. of 36th IEEE International Conference on Data Engineering*, *ICDE'20*, IEEE (2020).

[7] T. Henriksen, M. Dybdal, H. Urms, A. S. Kiehn, D. Gavin, H. Abelskov, M. Elsman, and C. Oancea, APL on GPUs: A TAIL from the Past, Scribbled in Futhark, *Procs. of the 5th Int. Workshop on Functional High-Performance Computing*, *FHPC'16*, ACM, New York, NY, USA (2016), 38–43.

[8] T. Henriksen, M. Elsman, and C. E. Oancea, Modular acceleration: Tricky cases of functional high-performance computing, *Proceedings of the 7th ACM SIGPLAN International Workshop on Functional High-Performance Computing*, *FHPC '18*, ACM, New York, NY, USA (2018).

[9] T. Henriksen, K. F. Larsen, and C. E. Oancea, Design and GPGPU performance of futhark's redomap construct, *Proceedings of the 3rd ACM SIGPLAN International Workshop on Libraries, Languages, and Compilers for Array Programming*, *ARRAY 2016*, ACM, New York, NY, USA (2016), 17–24.

[10] T. Henriksen and C. E. Oancea, A T2 graph-reduction approach to fusion, *Proceedings of the 2Nd ACM SIGPLAN Workshop on Functional High-performance Computing*, *FHPC '13*, ACM, New York, NY, USA (2013), 47–58.

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

[12] R. W. Larsen and T. Henriksen, Strategies for regular segmented reductions on gpu, *Proceedings of the 6th ACM SIGPLAN International Workshop on Functional High-Performance Computing*, *FHPC 2017*, ACM, New York, NY, USA (2017), 42–52.

[13] C. E. Oancea and A. Mycroft, Set-congruence dynamic analysis for thread-level speculation (TLS), *Languages and Compilers for Parallel Computing*, Springer-Verlag, Berlin, Heidelberg (2008), 156–171.

[14] C. E. Oancea and L. Rauchwerger, A hybrid approach to proving memory reference monotonicity, *Languages and Compilers for Parallel Computing*, *LNCS*, Springer Berlin Heidelberg (2013), 61–75.

[15] C. E. Oancea and S. M. Watt, Parametric polymorphism for software component architectures, *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, *OOPSLA '05*, Association for Computing Machinery, New York, NY, USA (2005), 147–166.

11 Appendix

11.1 On the materials

The repository is available at https://github.com/Ulrik-dk/knn-fut. Any further development after the submission of this thesis, and before the defense, will be kept to a new branch, so that the master branch reflects the state of the project on submission. Submitted along with this `thesis.pdf` is a `materials.zip` archive. This contains the following:

- `src/` which corresponds to the repository at the Github page.
- `benchmarks.ods` contains the data from the benchmarks.
- `qualitative_test.ods` contains data from the qualitative test.

11.2 The final version and its dependent code files

Not including library files `batch_merge_sort.fut` and those in `lib/`.

11.2.1 `bf.fut`

11.2.2 `kd-tree-common.fut`

11.2.3 `util.fut`

11.2.4 `constants.fut`

11.2.5 `v7.fut`

11.2.6 `v7-test.fut`

11.2.7 `v7-bench.fut`