Data-parallel Construction of Interval Trees and Range Trees

Author: Victor Alexander Schmidt `rqc908`
Supervisor: Troels Henriksen

Submitted: June 10th, 2024

Abstract

An interpretation of interval trees and range trees in the programming language Futhark. Through the theory of centered interval trees and k-dimensional range trees, this report will explore and conclude on a data-parallel approach, an evaluate how efficient such an implementaion performs against a naive approach.

Contents

1 Introduction

This report documents the theory and the data-parallel implementations of interval trees and range trees, or more specifically, **centered interval trees** and **k-dimensional range trees** (we are mostly looking at the 2-dimensional case, the dimensional generalization should be considered a novelty).

Futhermore, we will cover the performance of the before mentioned trees and attempt to analyze when a tree performs better than a brute force solution. In other words, how many queries does it take to justify the amount of time spent constructing a tree compared to the amount of queries the speedup would save?

In the real world, range trees are famous for being a structure that supports k-dimensional queries on databases. Interval trees do have use cases, though they may be more of a niche.

The full implementation can be found on GitHub[1].

2 Background

2.1 Futhark

To understand the why the challenges presented in this project pose complications, we first have to understand the purpose of the programming language of Futhark. Futhark is a statically typed, data-parallel, and purely funcitonal array language in the ML family, and comes with a heavily optimising ahead-of-time compiler that presently generates either GPU code via CUDA and OpenCL, or multi-threaded CPU code [4].

The strengths of Futhark lies in its simplicity, its speed, and its ability to generate, normally tedious to write, parallel code, using primitive operations that promise such parallelization. Within Futhark, we call these SOACs: Second-Order Array Combinators, which the compiler will know how to exploit to generate parallel code.

The limitations of Futhark lies in what it lacks. Futhark is quite simple, which leads to some problems as the need for more complex programs arise. Mainly, in the scope of this project, Futhark's lack of support for recursion, for pointer structures and for irregular nested parallelism. These limitations make way for new techniques to substitute these lacking funcitonalities. Two such techniques, which we will be making use of in this project, are *function-lifting* and *array-flattening*.

2.1.1 SOACs

All complex operations must be carried out using a combination of primitive functions, which promise some parallel execution. Primitive functions being built-in functions in the Futhark compiler, that has some form of possible parallelization. For instance, the SOAC `map` has the identity `map` : $(\alpha \to \beta) \to$

[1] `https://github.com/Victoriast8/futhark-range-trees`

$[n]\alpha \to [n]\beta$, transforming each element individually using a passed function. This is an example of parallelization using SOACs, as the compiler *knows* that each `map` call operates on each element individually, making it possible to assign each index its own thread (or rather allocates chunks/blocks, but this is besides the point) and in this manner assign n threads to an array of size n when `map` is called. In this way, the function completes in very little time, as long as it has the resources to parallelize efficiently.

Every function built inside of the Futhark compiler is exposed in the documentation[2] together with its asymototic work and span.

2.1.2 Flattening Arrays

Why do we need to flatten? The answer is simple - Futhark does not support irregular nested arrays (and by extension, irregular nested parallelism). For this reason, we are forced to use flattened arrays, when nested irregular arrays become unavoidable. Presenting trees as nested arrays would not be an option either, as irregular nested parallelism is not supported (nested array might also exhibit bad locality, but that is besides the point).

Using the flattening technique, a single array can represent the whole data set, as long as we keep track of the segment sizes or indices of where the flat array splits into subarrays. It is then possible to predefine new functions based on primitives, that operate on segmented data. When flattening, one usually has some nested, recusive or pointer structure to start with to flatten from, and then uses this version to flatten; this could be represented as a morphing of `[n][m] -> [n*m]`, assuming `[n][m]` is regular.

$$\texttt{[n][m]}\alpha \quad \xrightarrow{flat} \quad \texttt{[n*m]}\alpha$$

This expression hides one important detail, which is that flat arrays *can be irregular*. This means that each segment does not have to be of size `m`, but can have its own size, completely independent from neighboring segments.

To keep track of each segment inside of a flat array, we make use of a `shape` array, which contains the *size* of each segment in order. Here's an example:

```
array:       [[0], [1, 2], [3, 4, 5], [], [6]]
flat array:  [0, 1, 2, 3, 4, 5, 6]
shape array: [1, 2, 3, 0, 1]
```

2.1.3 Function-lifting and flattening

When it comes to flattening, functions also play a role. Imagine having a nested irregular array and presenting it as flat data. A **flattened function** would be the equivalent of instead of applying `map` with some function f on some nested array `arr`, then apply some flat function f_{flat} on the flat representation of `arr`. When handling flat arrays, we are theoretically dealing with nested arrays, or

[2]specifically SOACs can be found here: `https://futhark-lang.org/docs/prelude/doc/prelude/soacs.html`

in other words, arrays of multiple dimensions, but represented as a single array. This means that to define a segmented operation, a function that operates on segmented data/flat arrays, we first have to define a version of said function, such that it operates on nested arrays. This is what is called **function lifting**:

$$f : \texttt{[n]}\alpha \to \texttt{[n]}\beta \quad \xrightarrow{lift} \quad f^L : \texttt{[m][k]}\alpha \to \texttt{[m][k]}\beta$$
$$f^L : \texttt{[m][k]}\alpha \to \texttt{[m][k]}\beta \quad \xrightarrow{flat} \quad f^L_{flat} : \texttt{[m*k]}\alpha \to \texttt{[m*k]}\beta$$

The second step is then to flatten the lifted function, such that it operates on flat arrays instead of nested arrays.

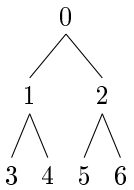
2.1.4 Pointer-Structures in Futhark

As mentioned, Futhark does not support pointer-structures. This means that to keep track of references, we have to get creative.

One way to keep track of references between data is using an array, which keeps track of reference indexes, as mentioned by Troels Henriksen in [5]. The general idea looks something like this:

```
value:     [0,1,2,3,4,5,6]

reference: [0,0,0,1,1,2,2]
```



This idea is what is solely used as pointer-references in this project, as this also bundles very nicely with flat arrays, as one can simply used a combined type consisting of both information and index-referencing, where one element points to the index of another element in the same array.

The example from before conveys the idea of parent-pointers. In this project, we concern ourselves with child-pointers; where parent-pointers are used by nodes to refer to which node it itself belongs to, child-pointers denote which nodes belong to a node. Child-pointers can represent the previous example as follows:

```
value: [0,1,2,3,4,5,6]

lc:    [1,3,5,-1,-1,-1,-1]

rc:    [2,4,6,-1,-1,-1,-1]
```

Where `lc` denotes a node's left-child index and `rc` denotes a node's right-child index. A value of -1 means there exists no child. If a node has no children, it is considered a **leaf**.

3 Interval trees

This section breifly explains the theory behind interval trees, how they are defined, how they are constructed and queried. The implementation of interval trees in Futhark will also be explained and analysed.

3.1 Theory

3.1.1 What is an Interval Tree?

An interval tree is a datastructure which represents a collection of 1-dimensional intervals. Interval trees have different ways of being expressed: As explained in the Wikipedia article for interval trees [8], there is:

- the **naive approach** (basically just two **BST**s, no better than brute force),
- the **augmented map**, which is an augmented, slightly modified, **BST**. This is usually what is referred to as interval trees, e.g. in [7] [3, chapter 17.3],
- the **centered interval tree**.

Going forward, we will only be discussing the **centered interval tree**, as this is the type of interval tree that has been used throughout this project. With these clarifications in place, we can move on discover how an interval tree is constructed and queried.

Say we have a set of intervals I, each consisting of a smaller start value x_{start} and bigger end value x_{end} and we would like to query a point p to see how many intervals in I where p lies within. In other words, count every interval where $x_{start} \leq p \leq x_{end}$ holds. Figure 1 visualizes a simple example of this.

But then, how do one define a datastructure using intervals? **Idea:** Create a **BST**, which stores two sets of overlapping intervals, one set by sorted x_{start} and one set sorted by x_{end} and a point x_{mid} that overlaps the intervals in each node. Intervals where $x_{end} < x_{mid}$ or $x_{start} > x_{mid}$ are then recusively put in separate nodes, left or right according to which direction the interval is located with respect to x_{mid}. This leads to the **centered interval tree** as described by Wikipedia in [8], resulting in a binary tree where each node contains:

- A center point (x_{mid})
- A pointer to a subtree that contains all intervals less than x_{mid} ($x_{mid} > x_{end}$ holds for all elements in that subtree)
- A pointer to a subtree that contains all intervals greater than x_{mid} ($x_{mid} < x_{start}$ holds for all elements in that subtree)
- All intervals overlapping x_{mid} sorted by x_{start}
- All intervals overlapping x_{mid} sorted by x_{end}

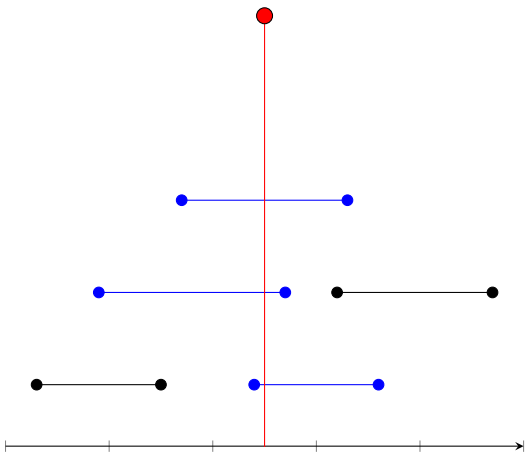


Figure 1: Visualization: Intervals in I and point p (red). Note that the intervals lie on a line, but has been shifted up/down, so we can distinguish between start- and end-points. One can observe the vertical line drawn from point p, slicing through intervals, indicating overlaps (blue intervals overlap with p).

3.1.2 Construction

Constructing an interval tree, requires the calculation of x_{mid}, the creation of two subtrees for non-overlapping interval if they exist, and the storage of $2k$ intervals, k being the number of intervals overlapping x_{mid}. x_{mid} should be chosen as some sort the midpoint of the evaluated intervals. The way we chose x_{mid}, is to take the smallest x_{start} in the sample of I_i and the biggest x_{end} in the sample of I_i and find the midpoint between these two points, where I_i are the intervals considered for any node n_i. x_{mid} can then be written as:

$$x_{mid} = \min(I_i) + \frac{\max(I_i) - \min(I_i)}{2}$$

Taking the average of I_i would be just as valid. For reference, Figure 2 shows a visualization of a **centered interval tree**.

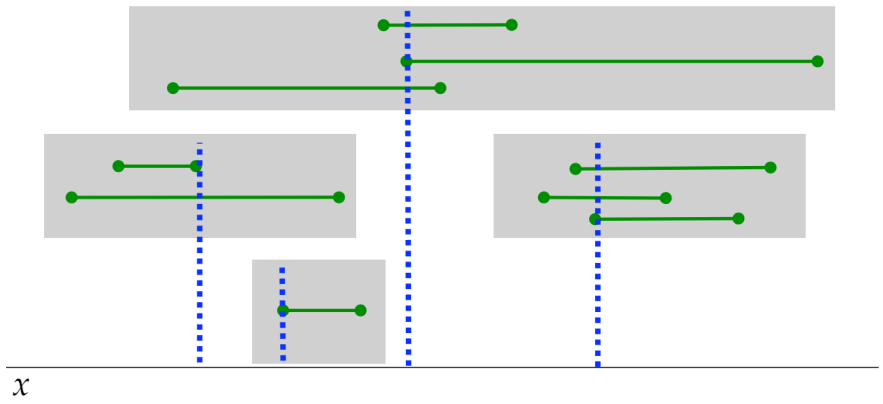


Figure 2: Visualization of the conecpt of a **centered interval tree** from the lecture slides of Carnegie Mellon University on *Interval Trees* [2]. The blue lines represent some overlapping point x_{mid} of each gray box, each gray box represents a node.

Algorithm 1 Construct (I_i)

Input. A set of intervals I_i.
Output. The root of an interval tree.

1: find x_{mid} using I_i
2: define (`left`, `center`, `right`) using x_{mid} to partition I_i for intervals, where `left` $= x_{end} < x_{mid} \in I_i$, `center` $= x_{start} \leq x_{mid} \leq x_{end} \in I_i$, `right` $= x_{mid} < x_{start} \in I_i$
3: `lc` = Construct(`left`)
4: `rc` = Construct(`right`)
5: (`sbs`,`sbe`) = sort `center` by x_{start} and x_{end} for efficient queries
6: return node $n_i = \{x_{mid}, \texttt{lc}, \texttt{rc}, \texttt{sbs}, \texttt{sbe}\}$

In Algorithm 1 we see the overall idea behind "Construct," the algorithm for constructing a **centered interval tree** similarly to what is described by Wikipedia [8].

3.1.3 Querying

When querying a **centered interval tree**, one can do it in more than one way, but they all share the same general approach: As described in Algorithm 2, at any node n_i we do a check for our query-point $p < x_{mid}$, as this implies that $\forall x_{end} \in n_i, \quad p < x_{end}$, or $\forall x_{start} \in n_i, \quad p > x_{start}$ if $p > x_{mid}$. Simply put, we only have to check start- or end-points, based on the difference between p and

x_{mid} of n_i. Knowing this, and knowing we have intervals sorted by start- and end-point stored in n_i, compare p to x_{mid}: If $p < x_{mid}$ then choose `sbs` (sorted by start x_{start}), else if $p \geq x_{mid}$ then choose `sbe` (sorted by end x_{end}). Then, for this node n_i, we go through the chosen set sorted on x_{start} or x_{end} furthest from the midpoint x_{mid} reporting overlapping intervals, until p falls outside of an interval or all intervals have been reported. Finally, run recursively on left-child (`lc`) if $p < x_{mid}$, or on right-child (`rc`) if $p > x_{mid}$, or stop if the child does not exist.

Algorithm 2 Query (p,n_i)

Input. A query-point p and a node n_i.
Output. Intervals that overlap with p.

1: remember: $n_i = \{x_{mid}, \texttt{lc}, \texttt{rc}, \texttt{sbs}, \texttt{sbe}\}$
2: find direction; on which side of x_{mid} does p lie?
3: `acc` = assume $p < x_{mid}$ is true; run through and report intervals sorted by x_{start} (`sbs`): Else, report intervals sorted by x_{end} (`sbe`).
4: `subquery` = assume $p < x_{mid}$ is true; Query (p,`lc`) if `lc` exists: Else, Query (p,`rc`) if `rc` exists
5: return `acc` + `subquery`

3.2 Implementation

The implementation of **centered interval trees** in Futhark can be seen in full on Github[3].

This section will cover *some* of the code concerning interval trees, but not all. The most non-trivial steps have been included and discussed in the report.

3.2.1 Construction

Constructing a **centered interval trees** is rather simple:

```
1 -- the keyword 'Maybe' in Haskell
2 type opt 'v = #some v
3             | #none
4
5 -- representing tree nodes with the following types
6 type point = f64
7 type child = opt i64
8 type interval = (point,point)
9 type node = {m: point, slice: (i64,i64), left: child, right: child}
```

Listing 1: Type definitions for interval trees.

Listing 1 shows type definitions that forego the implementation of interval trees. These remind us, how a `node` is represented in an interval tree: A midpoint `m` (x_{mid}), a `slice` as intervals stored in a node will almost always be irregular and

[3] https://github.com/Victoriast8/futhark-range-trees/blob/main/trees/interval_tree.fut

have to be stored separately, a `left` child index pointer and a `right` child index pointer.

An `interval` is defined as two points, x_{start} and x_{end}, and in this implementation, `point`s are defined as `f64`s, but this type could be arbitrary with a `lt` operator.

```
-- parameters for loop:
-- 1. work; dictates what data needs processing
-- 2. shape of work; dictates the sizes of each subarray in 'work'
-- 3. accumulator; accumulates the resulting nodes and sorted
    intervals
-- 4. number of nodes; simply a constant. Could be replaced with '
    length acc.0' where 'non' is used
-- 5. initial offsets; keeps track of each subarray's offset from
    previous iterations
loop (wrk, wrk_shp, acc, non, iv_off)
  =  (iv, [(i32.i64 n)], ([],[],[]), 0, 0)
  while !(null wrk) do
```

Listing 2: `loop` call and parameters for interval tree.

In Listing 2, one can observe the use of a `loop`. By why is this the case? Are `loops` not notoriously slow and fully sequential in Futhark? Yes, indeed they are. Recall that recursion is not supported in Futhark - the way to go about this is to parallelize the implementation in a different way. Instead of recursion we find another way to construct trees by using the SOACs defined within Futhark to achieve some parallelization.

The way we construct an interval tree by the use of a `loop`, is in our case with a top-down, breadth-first approach, creating one level of nodes at a time. Looking back at Listing 2, the first parameter `wrk` is the information for all nodes that is to be created represented as flat data at a depth h, where h is the current iteration of the `while-loop`. Keeping track of a flat data array, also requires keeping track of its shape, hence another `loop` parameter `wrk_shp`. `acc` is the accumulating value, accumulating nodes and intervals of work that is finished processing. `non` stands for number of nodes, `iv_off` stands for interval offset and are both omissible, as they represent the amount of stored nodes and intervals respectively. The length of `acc` is equally valid and is probably better to use, as to prevent unnecessarily many `loop` parameters.

```
-- step 1. create mid values for partition
let min = sgmScan (\x y -> if x < y then x else y)
            f64.highest flags (map (.0) wrk)
let max = sgmScan (\x y -> if x > y then x else y)
            f64.lowest flags (map (.1) wrk)
let mid = map (\i -> min[i] + 0.5 * (max[i] - min[i])) ends
```

Listing 3: The calculation of segmented x_{mid}.

Listing 3 shows the calculation of x_{mid} values for each segment; for each new node. Making use of a segmented scan, the `min` and `max` is located at the end of each segment, and afterwards a `map` over the `ends` allows for direct access to these values and simple indexing for the calculation of each segment's midpoint.

The segmented scan makes use of a previously calculated array `flags`, which is an array with the same size as `wrk` with non-zero values at the start index of each segment.

```
1 -- step 2. do the partition2L
2 let flg_scn = map (+(-1)) (sgmScan (+) 0 flags flags) -- in other
      words, the index of the segment
3 let ((split1,split2),(_,pwrk)) = flat_res_partition2L
4         (\t -> t.1.1 >= mid[t.0] && mid[t.0] >= t.1.0)
5         (\t -> mid[t.0] > t.1.1)
6         (0,(0.0,0.0)) (wrk_shp,(zip flg_scn wrk))
7 let (_,pwrk) = unzip pwrk
```

Listing 4: Segmented partition2 (splits each segment up into three parts, based on two comparison functions).

In Listing 4 observe the use of `flat_res_partition2L`. This is as the name suggests, a lifted `partition2` that works on flat arrays. In the context of creating interval trees, the flat partitioning partitions each segment (each queued node) into three parts: One part where all intervals have both its x_{start} and x_{end} to the left of x_{mid} (λ on line 5); one part where all intervals overlap with x_{mid} (λ on line 4); one part where all intervals lie to the right of x_{mid}, both x_{start} and x_{end} is $> x_{mid}$ (here is no λ - these intervals implicitly failed both previous partitions).

The definition of `flat_res_partition2L` can be seen in Listing 14, from line 85 to line 91. The flat partition2 call is defined as two flat partitions (`partitionL`), which can be found right above on line 58 to line 78.

```
1 -- accumulate results
2 let sbs_acc = sort_by_key (.0) (f64.<=) done -- sorted by start
3 let sbe_acc = sort_by_key (.1) (f64.<=) done -- sorted by end
4 let cent_offsets = scanExcl (+) 0 cent_length
5 let islice = map2 (\off len ->
6                     ((i64.i32 (off + iv_off)), i64.i32 len)
7                   ) cent_offsets cent_length
8 let new_acc =
9     (concat acc.0
10        (map3 (\p i (l,r) -> create_node p i l r) mid islice
      children),
11      concat acc.1 sbs_acc,
12      concat acc.2 sbe_acc)
13 let new_off = iv_off + (last cent_offsets) + (last cent_length)
```

Listing 5: Accumulation of results at the end of a depth iteration of the interval tree `loop`.

Creation of nodes and sorted intervals is shown in Listing 5. Line 8-12 shows the accumulation of nodes and stored intervals, sorted by x_{start} and x_{end} respectively, with variable names `sbs` (sorted by start) and `sbe` (sorted by end). The calculation of the interval slice (indexing of stored intervals, belonging to a given node) happens on line 5-7, using the segmented offset, the segment length and

the `loop` parameter, interval offset `iv_off`. Remember that segments were previously partitioned, so here there are only intervals satisfying $x_{start} \leq p \leq x_{end}$, which in the code is also referred to as center intervals.

The tree `loop` runs at most $\lg n$ times and at least 1 time, if either no intervals overlap or if all intervals overlap at a single point. The work here is then very simple to analyze, as sorting finished intervals is the worst-case work when we have a single node with $O(n \lg n)$ run time, and otherwise the SOACs dominate the work with $O(n)$. Remember $\lg n$ for the `loop`, and we get a work of:

$$W(\text{Construction}) = O(n \lg n)$$

Which matches [8] and [2].

The span uses the same line of reasoning, but with $O(\lg n)$ for both sorting and SOACs, times the $\lg n$ for the worst-case `loop`, giving of a span of:

$$S(\text{Construction}) = O(\lg n \cdot \lg n) = O(\lg^2 n)$$

3.2.2 Querying

```
def count (p : point) (t : tree) : i64 =
  let new_child_idx (n : child) : i64 =
      match n
      case #some idx -> idx
      case #none     -> -1
  -- loop traverses tree
  let (_,cnt) = loop (i,acc) = (0,0) while i >= 0 do
      let current = t.tNodes[i]
      let (istart, ilen) = (current.slice.0, current.slice.1)
      let dir = p <= current.m
      let (new_i, ivs, startidx, ldir) =
          if dir then (new_child_idx current.left,
                       t.tStartSortedIntervals, istart, 1)
                 else (new_child_idx current.right,
                       t.tEndSortedIntervals, istart+ilen-1, (-1))
      in if ilen <= 0 then (new_i,acc) else
          -- loop counts # of hit intervals
          let (_,sum) = loop (idx,iacc) = (startidx,0)
              while ilen > iacc
                  && (if idx >= 0 && idx < istart+ilen then
                      p >= ivs[idx].0 && p <= ivs[idx].1
                          else false) do
                              (idx+ldir,iacc+1)
          in (new_i,acc+sum)
  in cnt
```

Listing 6: `count` counts the number of overlaps between a given point p and a **centered interval tree** t.

The entire code for interval tree queries is shown in Listing 6. The query acts exactly as described by CMU and Wikipedia [8] [2], while being fully sequential. This query algorithm is impossible to parallelize, as it consists of a single tree traversal plus counting intervals. A `loop` is necessary for the traversal of the tree (outer loop). However, one may wonder: Why *ever* use a `loop` for counting

node intervals? It is inefficient and can be parallelized using a `map`. The counterargument is to imagine querying a point p that lies outside of most intervals. Using a `loop` in this case, completely skips the inner loop, as the algorithm counts from one of the sorted ends (as described in Algorithm 2), which also means as soon as p does not overlap one intervals, it will lie outside of the rest of the considered intervals, due to them being sorted. This is precisely why we wanted to sort the centered intervals in the first place, by both x_{start} and x_{end}, as to avoid querying as many intervals as possible.

Taking a look at the complexity, reveals a complexity very much like a **BST**: For the outer `loop` we always get $\lg n$ iterations (one full traversal, assuming $\lg n$ depth). Then for the inner `loop`, we get exactly k repetitions, where k is the amount of intervals our query-point p overlaps with. This gives us a work and span (as this query is fully sequential they are the same) of:

$$W(\text{Query}) = S(\text{Query}) = O(\lg n + k)$$

And this matches both the logic and complexity described in [8] [2].

4 Range trees

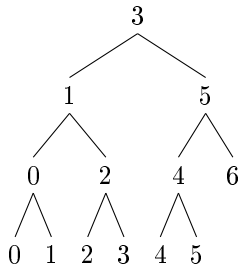
4.1 Theory

Problem: Given a k-dimensional box and a set ps k-dimensional points, how do we in an efficient way find the points contained in the box? **Idea:** Create a k-dimensional range tree for optimal queries!

How do we define a *range tree*? To start with, let us consider the simple 1-dimensional case. What is a one dimensional range tree? Consider the set of one-dimensional points:

$$ps = \{0, 1, 2, 3, 4, 5, 6\}$$

The conventions of a *range tree* are, that all points in a set of points, let us say ps, are represented as leaves in a **BST**, and internal nodes represents the largest value in the left subtree (why this is the case, will become apparent in the query algorithm). Here is a 1D range tree, build on ps:



In a *range tree* an internal node has the following properties:

$$n_{rangetree} = \{p, P(n), \texttt{lc}, \texttt{rc}, (\texttt{associated})\}$$

Where p is the smallest value of the left subtree, $P(n)$ denotes the canonical subset of node n and `lc`/`rc` denotes the left- and right-child respectively. `associated` is the associated structure, which is not relevant for the one-dimensional case. But what is the canonical subset $P(n)$ of a node n? The canonical subset $P(n)$ is the set of points (leaves) that can be found in both subtrees `lc` and `rc`. The canonical subset is also used to quickly report hits when querying, and for building associated structures, when dealing with higher dimensional *range trees*

When adding dimensions to the *range tree*, it becomes slightly more complicated. Let us say we need to represent a 2-dimensional *range tree*. The general idea is to construct a 1-dimensional *range tree* on the x-coordinate and simultaniously construct an associated structure in each node sorted on the y-coordinate, which is also a 1-dimensional *range tree*.

If we were to add more dimensions, let us say we would like to construct a k-dimensional *range tree*, simply extend the 2-dimensional range tree, such that each node in the associated structure of the first tree also contains an associated

structure, but this structure is sorted on the z-coordinate. Continue this kind of tree-nesting, until you have k nested associated structures sorted on all k dimensions, then you have a k-dimensional *range tree*.

4.1.1 Construction

Algorithm 3 ConstructKDRangeTree (ps,cd)

Input. A set ps of k-dimensional points and a current dimension cd.
Output. The root of a k-dimensional range tree.

1: if $cd < k$ then `subtree` = ConstructKDRangeTree(ps,$cd+1$)
2: `sps` = sort ps by cd
3: find median x_{mid} in `sps`
4: ($\texttt{sps}_{left}$, $\texttt{sps}_{right}$) = split `sps` at x_{mid}
5: `lc` = ConstructKDRangeTree($\texttt{sps}_{left}$,cd)
6: `rc` = ConstructKDRangeTree($\texttt{sps}_{right}$,cd)
7: return node $n_i = \{x_{mid}, \texttt{sps}, \texttt{lc}, \texttt{rc}\}$

The construction algorithm 3 is actually rather simple. The algorithm for k-dimensional range trees simply requires recursive construction of a right and left subtree, in addition to the creation of an associated structure, if the current dimension cd is not equal to the total dimension k of the points. The split point x_{mid} is quickly found as the median index rounded up.

There is one step in algorithm 3 which is not optimal: As mentioned by Mark de Berg et al. in *Computational Geometry* [1, p. 108, l. 5–16] presorting on the y-coordinate prevents sorting ps for each in line 2, and makes the construction of the associated structure take linear time. This means that presorting makes the construction take $O(n \lg n)$ time ($\lg n$ nodes times n for the construction of a subtree), instead of $O(n \lg n \cdot \lg n)$ time (sorting takes $O(n \lg n)$ time, do this for $\lg n$ nodes). [6, video 4/6] also suggests that using the merge-step of the merge-sort, one can sort by the y-coordinate by merging `lc` and `rc` element-by-element, obtaining the same construction time of $O(n \lg n)$. Please note, this paragraph is only true for 2D range trees, kD range trees have the construction time of $O(n \lg^{k-1} n)$ with presorting on every dimension and $O(n \lg^k n)$ without presorting.

4.1.2 Querying

Algorithm 4 KDRangeQuery (b,t,cd)

Input. A k-dimensional box b, a k-dimensional range tree t and the current operating dimension cd.
Output. The reported points of tree t that lie in box b

1: find v_{split}
2: **if** v_{split} is a leaf, check if the stored point should be reported
3: **else**
4: $v = v_{split}$.lc
5: **while** v is not a leaf **do**
6: **if** $b.0_{cd} \leq v.p_{cd}$ **then**
7: $v = v$.lc
8: **if** Equals(cd, k) **then** ReportCanonicals(v.rc)
9: **if** $cd < k$ **then** acc = acc+ KDRangeQuery (b,$v.\texttt{rc}_{asso}$,$cd + 1$)
10: **else** $v = v$.rc
11: **if** v is a leaf, check if store point should be reported
12: Similarly, repeat the while-loop for the right path, and report canonical subsets/run recursively on associated structures in left subtrees, when the algorithm moves to a right subtree.

*This algorithm is based on the 2DRangeQuery algorithm mentioned in [1, p. 108]

v_{split} is the point where a vertex v has the property $b.0_{cd} \leq v.p_{cd} \leq b.1_{cd}$, which means when the point of the vertex $v.p_{cd}$ is inbetween the points of the box b in the current dimension cd.

Important note for algorithm 4; a **box** is a representation of two k-dimensional points; the convention for these two points is the first point **must** contain the smallest value in **every** dimension. For instance, in a 2-dimensional window (box), the first point is defined as the lower left corner, while the second point is the upper right corner of the box, as the lower x- and y-value usually is presented this way on a graph.

How do *k*d-trees and range trees differ? One may find it important to note the distinction between a kd-tree and a range tree, as the ideas could seem identical. A k*d-tree* stores points, by separating points using k-dimensional axes to partition points, making use of linear storage, but has a $O(\sqrt{n} + k)$ query time. In contrast, *range trees* requires $O(n \lg^{d-1} n)$ storage for a k-dimensional range tree, where querying takes $O(\lg^d n + k)$ time. For further explanation, [1, chapter 5] [6, video 1-5] describes *kd-trees* and *range trees* in more detail.

4.2 Implementation

The implementation of k-dimensional range trees in Futhark follows algorithm 3 strictly. The full implementation of construction and querying can be found on GitHub[4].

4.2.1 Construction

To elaborate on some of the design choices that were made as a consequence of implemeting data-parallel construction of range trees: Range trees in Futhark are actually by nature really simple to implement, due to the flat data interacting nicely with the working sets of child-recursion. Remember that there is no recursion in Futhark - this means that the recursive behaviour has to be accomodated by `loops`. This is in practice also where the data-parallelization comes into play, as we can group recursive calls as segmented work, such that multiple nodes can be constructed at the same time in parallel, by calling SOACs which the compiler know how to optimize and parallelize efficiently.

In short, the implementation for construction was chosen as a top-down, breadth-first (depth-first) style. To elaborate, this means that for each level of depth in a range tree t, we create every node at that depth at the same time, while also preparing for the next iteration of nodes, as we are of course making use of a `loop`. This means that a for construction, the range tree `loop` run $O(\lg n)$ times. And this holds true, even when constructing k-dimensional associated structures, as this implementation simply adds associated subtrees to the working set, creating associated structures in parallel with its parent structures.

```
type opt 'v = #some v
            | #none
type child = opt i64
type point [d] = [d]f64
type box   [d] = (point [d],point [d])
type node  [d] = {m : point [d],
                  slice : (i64,i64),
                  subtree : child,
                  left : child,
                  right : child}
```

Listing 7: Custom types in range tree implementation.

Listing 7 shows type definitions for the k-dimensional range tree. A point is defined as an array, due to this being the only way to represent a k-/d-dimensional point consistently, while supporting more than a single dimension. Also, the dimension constructor `[d]` allows for the definition of a particular dimension, which is favorable in the `range_tree` module, as the compiler will allow us to only accept input of the same, arbitrary dimension, which is quite satisfying.

The type `opt 'v` is used as the keyword "Maybe" in haskell. This allows the representation of "Nothing" in Haskell or `#none` in Futhark, comparable to using

[4] `https://github.com/Victoriast8/futhark-range-trees/blob/main/trees/range_tree.fut`

integers where for instance a negativ value represents the absence of a value.

A `node` is represented as a point `m`, with the canonical subset `slice`, an associated structure root-index `subtree`, and `left`/`right` child-indexes. In the module `k_range_tree`, a node n in a k-dimensional range tree will have `subtree` = `#none` when the node n is part of is the kth (last) associated structure, and n will have `left` = `#none` and `right` = `#none` if and only if n is a leaf.

```
1 loop (wrk, wrk_shp, wrk_dim, acc) =
2   (ps', [(i32.i64 n)], [1i32], ([],[]))
3   while !(null wrk) do ...
```

Listing 8: The range tree loop and its parameters.

Listing 8 shows the `loop` and its parameters between iterations, in other words which properties that goes between the calculation of nodes of different depths. This includes the `wrk` array, which stands for work and is the flat data that needs processing. `wrk_shp` is simply the shape (size of each segment) of `wrk`. `wrk_dim` assigns each segment a working dimension, as to keep track of when we should stop adding $d+1$ dimensional associated structures. And lastly `acc` is of course the accumulated result. Make note that `acc` contains two values, as we not only need to accumulate `treeNodes`, but also `treeCanonicals`, as Futhark does not support irregular arrays we need to represent the canonical subset in a separate array. This is also why `slice = (i64,i64)` in the definition of `node` in Listing 7, as we indicate the segment of canonicals this node has ownership of. `slice` can be seen as `slice = (start-index, length-of-canonical-segment)`.

```
1 let medians =
2           map (\shp ->
3                   if shp < 2 then 1 else shp - (shp)/2
4               ) wrk_shp
5 let (new_shp,new_dim) = zip3 wrk_shp medians wrk_dim
6     |> filter (\(shp,_,_) -> shp > 1)
7     |> map (\(s,i,d) -> [(i,d),(s-i,d)])
8     |> flatten
9     |> unzip
```

Listing 9: Calculation of new work (not associated structures).

Listing 9 shows the calculation of median indexes: In Futhark, division of integers always round down. Line 3 then always lets the median be bigger, if the shape is odd, which is how the implementation is described in [1]; if the number of nodes is odd, let the left subtree have the additional element.

The new shape `new_shp` is then calculated by first removing leaves and halving the sizes of the shape, as seen in the map in line 7. This is particularly nice, as we do not have to modify the actual work except for removing leaves; the shape dictates which trees to build, so for creating children, we simply split each segment into two equally sized segments. The dimension `new_dim` follows, as each segment still needs a dimension to accompany it.

```
let new_sub_wrk =
  zip3 wrk seg seg_dim
  |> filter (\(_,_,dim) -> (i64.i32 dim) < d)
let (sub_wrk,_,_) = sort_by_key (\e -> e)
                    (\x y -> if x.1 == y.1
                             then x.0[x.2] <= y.0[y.2]
                             else x.1 < y.1
                    ) new_sub_wrk |> unzip3
```

Listing 10: Segmented sort of spawned associated structures.

In Listing 10 we see the queueing of associated work. This is done by having the work with its corresponding shape index, together with its corresponding dimension. This makes it possible to `filter` off kth dimensional work, as this it is the final layer of associated structures, and then duplicating every working set of points, using a segmented sort to sort by segment, or if the segment is equal, then sort by coordinate in the working dimension.

Other parts of construction includes storing canonical subsets and predicting child indexes and associated structure indexes.

The work of the construction algorithm is $O(\lg n)$ iterations for the loop, with the most expensive call being the call to `sort_by_key`, which in this case is a call to `merge_sort`. `merge_sort` has a work of $O(n \lg n)$, which gives us a total work of $W(\text{Construction}) = O(n \lg^2 n)$. But we forgot one thing - `merge_sort` is only called when associated structures are to be queued, which means that the real work depend on the amount of dimensions we queue work for. This end up giving us work identical to the theoretical asymptotics, $W(\text{Construction}) = O(n \lg^k n)$, as each iteration queues n nodes per size of the previous dimension, taking $\lg n$ iterations means having $\lg n$ sorted work with 2 dimensions, $\lg n \cdot \lg n = \lg^2 n$ for three dimensions, and so on. Multply this with the work of sorting, and we get the work as mentioned before; $W(\text{Construction}) = O(n \lg n \cdot \lg^{k-1} n) = O(n \lg^k n)$

The span of data-parallel construction of k-dimensional range trees, comes out to be the span of number of iterations, which again is $O(\lg n)$ times the most costly operation. `merge_sort` should have a span of $O(\lg n)$, which matches the span of other operations, such as `filter`; however again we are creating a lot of work per dimension. This means that `merge_sort` again should have the highest span, using the same argumentation as for the work, the amount of work we produce ends up as $\lg^{k-1} n$. The span in total comes out to be iterations times `merge_sort` complexity: $S(\text{Construction}) = O(\lg n \cdot \lg^{k-1} n) = O(\lg^k n)$. One important note to this; if we are only dealing with a 1-dimensional range tree (just a **BST**) the span will actually be $S(\text{Construction}) = O(\lg^2 n)$, as the most expensive operation will no longer be `merge_sort`. We can then write the *actual* span as: $S(\text{Construction}) = O(\lg^k n + \lg^2 n)$

4.2.2 Querying

There is not much to say about the query algorithm which has not already been said. The main idea is to find a v_{split} that splits the path of a querying win-

dow/box, such that the smaller value of the window is smaller than the point in v_{split} and the bigger value is bigger than the point in v_{split}. Then report left and right canonical subsets or the associated structure for further querying, depending on the dimension. Repeat until no more v_{split}s are left and all left-/right reporting has finished.

```
1 loop (queue,typ,dim,acc) =
2   ([t.tNodes[0]],[0],[1],0)
3   while !(null queue) do ...
```

Listing 11: Query `loop` and parameters.

In Listing 11, we see the loop parameters. The query is fully sequential and consists of a `queue`, which dictates upcoming work. `typ` and `dim` contains information about the type of work and dimension to operate in, of the same index in `queue`. `acc` accumulates once again (in this case, reporting the number of points inside the query window. A query reporting the actual point coordinates has also been implemented in exactly the same fashion, just accumulating points instead of incrementing a counter).

Note that `typ`; 0 is finding v_{split}; 1 is reporting on the left subtree; 2 is reporting on the right subtree. When handling finding a v_{split}, a right search and left search is queued, unless v_{split} is a leaf node, which in that case queues nothing. A left and right search can queue $\lg n$ associated structures each, or none at all, which makes it possible to queue $\lg n \cdot v_{split}$.

The work of querying vary vastly, depending on the size of the querying window. With a very small window, we have the work of $W(\text{Query}) = O(\lg n)$, no matter the dimension of the queried range tree. As described in [1, chapter 5.4, Theorem 5.9, p.110] th recurrence for querying a d-dimensional range tree involves seaching in a first-level tree, which as mentioned before takes $O(\lg n)$ time, and querying a logarithmic number of $(d-1)$-dimenional range trees. Hence,

$$Q_d(n) = O(\lg n) + O(\lg n) \cdot Q_{d-1}(n)$$

As our query algorithm is both identical to the one in [1] and their logic hold for our implementation, we can copy their complexity analysis. Remember: our query is fully sequential, making both work and span: $W(\text{Query}) = S(\text{Query}) = O(\lg^d n + k)$, where k is the number of reported points.

4.2.3 Shortcomings (possible improvements)

Duplicate coordinates was mentioned in [1] as a problem, thus using general sets of points was an imposed solution. This project has not revealed exactly why this is problematic, and duplicate, randomly generated 64-bit numbers did not seem to pose any problems for the implementation. However, we find it appropriate to inform that this may/may not cause some unforeseen incidents, as duplicate dimensional points have been completely ignored throughout the range tree implementation, so one may need to keep this in mind going forward.

The algorithm asymptotics are as mentioned in the theory section for range trees, not optimal. As we are neither presorting on every coordinate nor merging the canonical subsets of a node's children, we retain the construction time of $O(n \lg^k n)$ instead of the possible $O(n \lg^{k-1} n)$.

The original plan was to also do also do a bottom-up implementation of the range tree construction in Futhark, as to see if this concern of worse asymptotics made any difference in practice. Due to the complexity of a bottom-up implementation, this was unfortunately not done in time.

A simpler alternative may have been to duplicate and presort on each dimension of each point for construction, and `zip`/represent each presort as flat data with a constant offset. This is purely a trail of thought, if there was one thing we wish we had asked Troels to share his thoughts on, this would probably have been it.

Data-parallel query As this project's focus has been on data-parallel construction of interval trees (and by this point, also range trees), the data-parallel query was never a requirement for this project. However, it should be quite a simple implementation and might be interesting, especially for higher dimensions (as higher dimensions creates more work, it gives way for more parallelism).

5 Validation testing

Validation testing plan The general testing plan has been the following:

1. Create the program and have it pass the compiler.
2. Test on small datasets and inspect values manually. Usually inspecting values manually is a little overkill as the program usually can be set up to do this for you. However using the `opaque` types that come with the Futhark `modules`, one does not simply reference or match on intrinsic values of these types within a program. So this was the solution; using the `$ futhark repl` to inspect output.
3. Test on large (and numerous) datasets. If this fails, return to step 2 with the knowlegde of where the big dataset failed.

But how do we validation test? How can we be sure that our so called `opaque` types actually represents a tree as described in this report? The trick is, that we can't. A tree is an abstract representation, as long as the subfunctions return the expected results (in our instance, just `Query`), the tree construction should be valid as long as it uphold the expected asymptotics for run time and storage.

So the validation plan is to **compare to a brute force query**, or in other words compare the result of running an interval tree and range tree query against a simple implementation, which goes through and counts all overlapping points/intervals on the same set of data which the tree is built upon.

```
1  def brute_count [n] (p : point) (iv : [n]interval) : i64 =
2  map (\(l,h) -> if p >= l && h >= p then 1 else 0) iv |> reduce (+)
       0
3  -- ==
4  -- entry: validate_itree1D_count
5  -- nobench random input { [100000]f64 [100000]f64 } output { true }
6  -- nobench input
7  -- { [0f64,100f64,100f64,42f64]
8  --   [93f64,100f64,10000f64,73f64]
9  -- }
10 -- output { true }
11 entry validate_itree1D_count [n] (iv1 : [n]f64) (iv2 : [n]f64) :
       bool =
12     let iv = zip iv1 iv2
13     let p = ((last iv).0 + (last iv).1)*0.5 -- mostly targeted at
       random datasets
14     let t = itree1D.many iv
15     let traversal = itree1D.count p t
16     let brute = brute_count p iv
17     in brute == traversal
```

Listing 12: Validation of interval trees.

```
def brute_query [n][d] (b : box [d]) (ps : [n]point [d]) : i64 =
  map (\p ->
          if (map3 (\p lo hi ->
              p >= lo && p <= hi
          ) p b.0 b.1
          |> all (\t -> t)) then 1 else 0
      ) ps |> reduce (+) 0
-- ==
-- entry: validate_rtree2D_query
-- nobench input { [1.0,1.0] [5.7,5.8]
    [[1.1,2.2],[3.3,4.4],[5.5,6.6],[7.7,8.8]] }
-- output { true }
-- nobench random input { [2]f64 [2]f64 [64][2]f64 } output { true
    }
entry validate_rtree2D_query [n][d] (b1 : point [d]) (b2 : point [d
    ]) (ps : [n]point [d]) : bool =
    let b = fix_box b1 b2
    let brute = brute_query b ps
    let t = k_range_tree.build ps
    let traversal = k_range_tree.count b t
    in (brute == traversal)
```

Listing 13: Validation of range trees.

Both validation tests make use of the `brute_query` or `brute_count`. This brute force implementation makes use of a `map` to check each element and a `reduce` to collect all hits. The Futhark compiler was assumed to know how to fuse these together, as the `map` $\rightarrow$ `reduce` was tested to be either faster or equally fast to `looping` over each element sequentially. Also note the small datasets on line 3-10 in Listing 12 and line 8-12 in Listing 13. To run the small datasets, clone the GitHub repository[5] and navigate to the `/trees` subdirectory and run `$ futhark test interval_tree_tests.fut` for interval tree testing and `$ futhark test range_tree_tests.fut` for range tree testing.

If you wish to run the validation of big datasets, these have also been set up with a `Makefile` in the same directory:

- `make validate_1D_count` generates dataset of sizes $2^{16}, 2^{17}..., 2^{20}$ and validates them by printing a boolean, stating if the brute force solution was equal to the tree function. If you are unsure of what this means, a output of `true` means success.

- `make validate_KD_query` generates windows of $2, 3, 4$ dimensions and datasets (points) of $2^{16}, 2^{17}$ 2-dimensional points and $2, 3, 4$-dimensional points of size 2^{10}. The semantics are the same as for the big dataset test for the interval tree.

At the time of writing this, one thing to be vary of is the `KD_query` failing validation from the syntax of calling the entry point. But this only happens sometimes, know this may be an issue if you decide to try validation yourself.

[5] `https://github.com/Victoriast8/futhark-range-trees`

6 Benchmarking

This section will cover the benchmarking regarding interval trees and range trees.

6.1 Benchmarking plan

The plan for benchmarking will be the following:

1. Benchmark the creation of trees when created sequentially, on multiple CPUs and on the GPU.
2. Benchmark sequential querying trees of different sizes.
3. Benchmark sequential brute force counting/querying.

The interesting thing here will be the difference in querying time on different input sizes and the difference between sequential and parallel tree-construction.

Input sizes have been chosen as a result of previous benchmarking - for optimal query comparison and to observe the change in tree construction, both relatively small and somewhat big datasets could prove interesting. For range trees, sizes $2^{10}, \ldots, 2^{18}$ was chosen, as any bigger value makes the sequential tree construction take incredibly long. Interval tree input size is slightly bigger; $2^{10}, \ldots, 2^{20}$ as these are generally quicker to build. Please note that not all inputs are used for tree construction, only the big datasets. Small datasets (such as 2^{10}) are used for query comparison.

If you wish to run these benchmarks yourself, the `Makefile` in the `trees` subdirectory contains the commands used for benchmarking (and generating datasets, but those are run automatically before benchmarking). These commands are:

- `make bench_1D_count` benchmarks interval tree creation with backends C, multicore and cuda, while benchmarking interval overlap count with backend C.
- `make bench_range_tree` benchmarks range tree creation with backends C, multicore and cuda, while benchmarking 2D range querying with backend C.

6.2 Benchmarks

Benchmarks were performed on the hendrixgate GPU cluster on the machine `hendrixgpu05fl`, using a Titan RTX with 48+ CPUs and 8 GPUs.

#intervals	sequential	multicore (CPU)	cuda (GPU)
2^{16}	132.3	57.8	8.8
2^{17}	256.5	99.2	11.0
2^{18}	557.9	179.9	12.8
2^{19}	1275.3	354.1	17.1
2^{20}	2743.2	718.0	27.4

Table 1: Benchmark results for interval tree creation in milliseconds ms.

#intervals	interval tree count	brute force count
2^{10}	1	1
2^{11}	3	2
2^{12}	6	4
2^{13}	9	8
2^{14}	15	16
2^{15}	27	31
2^{16}	53	65
2^{17}	108	136
2^{18}	209	261
2^{19}	421	558
2^{20}	896	1556

Table 2: Benchmark results for counting overlapping intervals in microseconds μs.

#points	sequential	multicore (CPU)	cuda (GPU)
2^{16}	4.768	1.925	0.043
2^{17}	7.997	3.110	0.079
2^{18}	17.599	6.359	0.173
2^{10}	0.020	0.020	0.009
2^{10} (3-dim)	0.154	0.116	0.014
2^{10} (4-dim)	0.893	0.374	0.021

Table 3: Benchmark results for range tree creation in seconds. Notice that the last three trees has the same number of input points, but differ in dimensions.

#points	range tree query	brute force query
2^{10}	47	5
2^{11}	52	8
2^{12}	57	13
2^{13}	65	21
2^{14}	68	41
2^{15}	77	83
2^{16}	83	168
2^{17}	89	330
2^{18}	95	675

Table 4: Benchmark results for 2D range queries in microseconds μs.

6.3 Discussion

Range tree benchmarks exhibits a good speedup, when constructed in parallel. The following can be conducted from observing Table 3 and Table 4:

- During tree creation, the multicore backend exhibits a $\sim 2.4 - 2.8$x times speedup over the sequential backend, which seems to grow as the datasets get bigger.
- During tree creation, the cuda backend exhibits an excellent speedup of $\sim$ 100x times speedup compared to the sequential backend, and a $\sim 36 - 45$x speedup compared to the multicore backend, with speedups decreasing as datasets get larger.
- When querying, using a range tree for querying becomes faster than brute force approximately when the amount of points that needs querying is $\geq 2^{15}$.
- Brute queries scales linearly as expected (datasets increase in powers of two and so does brute force query time), while range tree queries becomes slightly longer ($5 - 10\mu s$) when the dataset doubles in size.
- Tree creation seem to take approximately double the time, as the dataset doubles in size.
- To amortize the cost of creating a tree of size 2^{18} using cuda, one would have to query $(675 - 95)x = 173060 \Rightarrow x = 298.38$ times, rounding up to 299 sequential queries, in order to justify creating a data-parallel range tree.

Were these results expected? We would argue yes, a speedup was definitely expected, though not of this magnitude. One reason for this speedup being so big,

is probably the work of this particular range tree implementation not being very optimized, while the implementation has a good span (meaning a lot of parallelization is possible). This would explain the massive speedup, as the sequential backend would be slowed drastically, while the cuda backend has the opportunity to cut a lot of corners (initialize a lot of parallelism). This speculation is based mostly off of the work-span analysis in the range tree implementation section.

The range query does seem a little fishy, as it does not seem to slow down. However one thing the range query does particularly well, is sort away points that lie outside of a query-window. Combine this with the fact that the query-window is chosen as a constant to prevent calculations during the benchmark, and we may realize that maybe this **query-window** was **poorly chosen**. In hindsight, a more thought through approach should have been taken. Choosing both complete misses and complete overlaps would accomodate worst and best case queries.

Interval tree benchmarks exhibits some speedup, when constructed in parallel. Some derivations of Table 1 and Table 2 are:

- During tree creation, the multicore backend exhibits a $\sim 2.3 - 3.8$x times speedup compared to the sequential backend.
- During tree creation, the cuda backend exhibits a $\sim 15 - 100$x times speedup compared to the sequential backend and a $\sim 6.5 - 26.2$x times speedup compared to the multicore backend.
- Interval tree queries seem to run slightly faster than brute force counts. Supplying a little extra data here - running query benchmarking on a local machine yields approximately exactly $\sim$ 2x speedup when using interval trees. This benchmark might contain some noise.
- To amortize interval tree construction, going off of a data-parallel construction using the GPU of input size 2^{20}, we get that we need to conduct $(1556 - 896)x = 27400 \approx 41.5$ queries, or rounded up, after 42 queries it would be more beneficial to create and query an interval tree, rather than use naive query.

Were these results expected? The construction speedup was expected, but why the query has a speedup of $\sim$ 2x is not clear. Maybe it just so happens that $\lg n + k$ tends toward $\frac{n}{2}$? Like in the case with the range tree, perhaps the huge speedup mostly comes from the fact that tree construction takes quite a bit of work, while having a good span, allowing for a good amount of parallelism. However one can not deny, that the interval tree runs quite a lot faster on the GPU and on the CPU. If you would like to observe the speed difference, I reccommend running the benchmarks yourself, as described in the benchmarking plan section.

7 Conclusion

Despite somewhat deficient benchmarking, and implementation shortcomings, we have managed to create the datastructures of interval trees and range trees in a data-parallel fashion. We have made use of flat arrays, function lifting, `loops` accomodating for recursion, and more, to achieve a program that runs, validates, and reacts well to running on the GPU.

8 Future work

What is the next step in the direction of this project? Here are some suggestions, based on the path this project shed light upon.

Data-parallel query As mentioned in the shortcomings of the range tree implementation, a data-parallel query could be interesting to implement. Higher dimensional range trees would benefit from a data-parallel query, as "spawning more work" that can be parallelized would be the key to achieving a speedup, and the higher the dimension, the more v_{split} and in turn the more left and right searches are created. For two dimensions however, this would probably not have a significant impact.

Improvements to range trees Once again, as mentioned in the range tree shortcomings section, our range tree implementation is not optimal. Too much sorting is the current implementation's issue, creating a "bottom-up" version or a "presorted on all dimensions" version should prove faster. Naturally, this would be the first step if there were more time in this project. Do however note that such an implementation is theorized by us, to be quite the bit more challenging to implement in Futhark than the current version, due to differing logic (the implementations would look very different).

Fractional cascading would surely be the second step to improve range trees. As mentioned by both [6] [1], fractional cascading is a technique to speed up query time or storage complexity, and as far as we recall, does not come with a cost.

9 Code listings

A section for additional code listings that are too big to comfortably weave into the report.

```
-- This listing is ALL of helper.fut except for THIS comment.
import "../lib/github.com/diku-dk/sorts/merge_sort"
-- helper functions

-- intertwines two arrays - as = [1,2,3], bs = [4,5,6]; intertwine as bs = [1,4,2,5,3,6]
def intertwine [n] 't (as : [n]t) (bs : [n]t) : [n*2]t =
    map2 (\x y -> [x,y]) as bs |> flatten

def sort_by_key [n] 't 'k (key : t -> k) (dir : k -> k -> bool) (xs : [n]t) : [n]t =
    merge_sort_by_key key dir xs

-- Typical exclusive scan
def scanExcl [n] 't (op : t -> t -> t) (ne: t) (arr : [n]t) : [n]t =
    scan op ne (map (\i -> if i > 0 then arr[i-1] else ne) (iota n))

-- Typical (inclusive) segmented scan
def sgmScan [n] 't (op : t -> t -> t)
                   (ne : t) (flg : [n]i32) (arr : [n]t)
                   : [n]t =
    let flgs_vals =
        scan (\(f1,x1) (f2,x2) ->
            let f = f1 | f2 in
            if f2 > 0 then (f, x2)
            else (f, x1 `op` x2)
        ) (0,ne) (zip flg arr)
    let (_, vals) = unzip flgs_vals
    in vals

-- Segmented (inclusive) scan on type i32 with operator '+'
def sgmSumInt [n] (flg : [n]i32) (arr : [n]i32) : [n]i32 =
    let flgs_vals =
        scan ( \ (f1, x1) (f2,x2) ->
            let f = f1 | f2 in
            if f2 > 0 then (f, x2)
            else (f, x1 + x2))
            (0,0) (zip flg arr)
    let (_, vals) = unzip flgs_vals
    in vals

-- Makes a flag array: Given a shape and values to insert as the flag-value.
def mkFlagArray 't [m]
            (aoa_shp: [m]i32) (zero: t)       --aoa_shp =[0,3,1,0,4,2,0]
            (aoa_val: [m]t  ) : []t =         --aoa_val =[1,1,1,1,1,1,1]
```

```
    let shp_rot = map (\i->if i==0 then 0        --shp_rot
    =[0,0,3,1,0,4,2]
                           else aoa_shp[i-1]
                      ) (map i32.i64 (iota m))
    let shp_scn = scan (+) 0 shp_rot             --shp_scn
    =[0,0,3,4,4,8,10]
    let aoa_len = shp_scn[m-1]+aoa_shp[m-1]      --aoa_len= 10
    let shp_ind = map2 (\shp ind ->              --shp_ind=
                           if shp==0 then -1        --
    [-1,0,3,-1,4,8,-1]
                           else ind                 --scatter
                       ) aoa_shp shp_scn         --
    [0,0,0,0,0,0,0,0,0,0]
    in scatter(replicate (i64.i32 aoa_len) zero)--
    [-1,0,3,-1,4,8,-1]
            (map i64.i32 shp_ind) aoa_val      --    [1,1,1,1,1,1,1,1]
    -- result: [1,0,0,1,1,0,0,0,1,0]

-- Lifted partition
def partitionL 't [n] [m]
                  (condsL: [n]bool) (dummy: t)
                  (shp: [m]i32, arr: [n]t) :
                  ([m]i32, ([m]i32, [n]t)) =
    let begs   = scan (+) 0 shp
    let flags  = mkFlagArray shp 0i32 (map (+1) (map i32.i64 (iota
    m)))
    let outinds= map (\f -> if f==0 then 0 else f-1) flags |>
    sgmSumInt flags :> [n]i32

    let tflgs = map (\ c -> if c then 1 else 0) condsL
    let fflgs = map (\ b -> 1 - b) tflgs

    let indsT = sgmSumInt (flags :> [n]i32) tflgs
    let tmp   = sgmSumInt (flags :> [n]i32) fflgs
    let lst   = map2 (\b s -> if s > 0 then indsT[b-1] else -1i32)
    begs shp
    let indsF = map2 (\i v -> lst[i]+v) outinds tmp

    let inds  =  map4(\c indT indF sgmind->
                        let offs = if sgmind > 0 then (i64.i32 begs
    [sgmind-1]) else 0i64
                        in if c then offs + (i64.i32 indT) - 1i64
                                else offs + (i64.i32 indF) - 1i64
                      ) condsL indsT indsF outinds

    let fltarr = scatter (replicate n dummy) inds arr

    in  (lst, (shp,fltarr))

-- Lifted partition2. There are some shortcomings, namely the
    repeated partition on an aldready partitioned segment
def flat_res_partition2L 't [n] [m]
                    (p1 : t -> bool) (p2 : t -> bool)
                    (dummy : t) (shp : [m]i32, arr : [n]t)
                    : (([m]i32,[m]i32), ([m]i32, [n]t)) =
    let (split1, (_,ps))  = partitionL (map p1 arr) dummy (shp, arr
    )
```

```
90      let (split2, (_,ps')) = partitionL (map p2 ps)  dummy (shp, ps)
91      in ((split1,split2),(shp,ps'))
92
93  -- Flat/segmented replicate of bools. Never use this function,
        unless 'ns' reduces to >=1i64:
94  -- ns - is a shape array. Convention: foreach segment: replicate ns
        [i] ms[i]
95  def flat_replicate_bools [n] (ns : [n]i64) (ms : [n]bool) : []bool
        =
96      let scn = scanExcl (+) 0 ns
97      let inds = map2 (\n i -> if n>0 then i else -1) ns scn
98      let size = (last scn) + (last ns) -- DPP slides use '(last inds
        )'. This is wrong. Consider the last element of 'ns' is 0.
99      let vals = scatter (replicate size false) inds ms
100     let flgs = scatter (replicate size 0) inds ns
101     in sgmScan (||) false (map i32.i64 flgs) vals
102
103 -- Flat/segmented replicate for i32
104 def flat_replicate_i32 [n] (ns : [n]i64) (ms : [n]i32) : []i32 =
105     let scn = scanExcl (+) 0 ns
106     let inds = map2 (\n i -> if n>0 then i else -1) ns scn
107     let size = (last scn) + (last ns)
108     let vals = scatter (replicate size 0) inds ms
109     let flgs = scatter (replicate size 0) inds ns
110     in sgmScan (+) 0 (map i32.i64 flgs) vals
```

Listing 14: All helper functions. If in doubt, refer to this listing with a line number. The most recent version of `helper.fut` can be found here.

References

[1] Mark de Berg et al. *Computational Geometry Algorithms and Applications Third Edition*. Springer, 2008.

[2] CMU. *Interval Trees*. URL: `https://www.cs.cmu.edu/~ckingsf/bioinfo-lectures/intervaltrees.pdf` (visited on 06/01/2024).

[3] Thomas H. Cormen et al. *Introduction to Algorithms, Fourth Edition*. The MIT Press, 2022.

[4] DIKU. *Why Futhark?* URL: `https://futhark-lang.org/` (visited on 05/15/2024).

[5] Troels Henriksen. *Pointer Structures*. `https://github.com/diku-dk/dpp-e2023-pub/blob/main/slides/L4-pointer-structures.pdf`. 2023.

[6] Phillip Kindermann. *Lecture 5; Computational Geometry; Orthogonal Range Queries: Range Trees and Kd-Trees*. `https://www.youtube.com/watch?v=XjfwknwOqHg&list=PLubYOWSl9mIsOJW-u2ZusOJzZvhNi0xks&ab_channel=PhilippKindermann`. 2020.

[7] Yihan Sun, Daniel Ferizovic, and Guy E. Belloch. "PAM: Parallel Augmented Maps". In: (2018). URL: `https://dl.acm.org/doi/pdf/10.1145/3200691.3178509`.

[8] Wikipedia. *Interval Tree*. URL: https://en.wikipedia.org/wiki/Interval_tree (visited on 05/31/2024).