Department of Information and Computer Science
Utrecht University

INFOB3CC: Assignment 3 Quickhull

Trevor L. McDonell & Ivo Gabe de Wolff

Deadline: Saturday, 1 February, 23:59

A shape is *convex* if it does not have any dents. Formally, for any two points on the shape, the line between those shapes must also be fully contained in the shape. In this assignment, you are given a set of points, of which you must compute the convex hull. The convex hull is the smallest polygon containing all those points. You can also phrase it as finding the set of vertices on the boundary of the input points.

Quickhull

One algorithm to compute the convex hull is *Quickhull*. As you might guess by its name, it has some similarities with Quicksort: it applies a partition and recurses on two segments. First, consider a description of the sequential Quickhull:

We start by finding the points p_1 and p_2 with lowest and highest x coordinates. Those points must be in the convex hull. Draw a line between those points, and partition the points into all points above, and all points below the line. The points p_1 and p_2 should be in both sides. Assume we're handling the upper half now.

Take the point p_f furthest from the line (p_1, p_2). All the points in the triangle between p_1, p_2 and p_3 cannot be in the convex hull, so those points can be ignored.

Let L be the set of points on the left of the line from p_1 to p_f and R the set of point on the left of line from p_f to p_2. Recurse on those sets.

In recursion, we will again search for the furthest point from respectively line (p_1, p_f) in L and (p_f, p_2) in R.

Accelerate

Accelerate is a DSL for array programming in Haskell. Using this DSL, you can write high performance code to be executed on multicore CPUs or GPUs. Accelerate exports many functions with the same name as functions from the Prelude. Hence we made the Prelude qualified, meaning that you need to write `P.` in front of many functions from the Prelude, like `P.show`. You can find documentation on Accelerate at `https://hackage.haskell.org/package/accelerate-1.2.0.1/docs/Data-Array-Accelerate.html`.

When working with tuples, you can use `T2` as a constructor or a pattern in pattern matching. This isn't yet documented on that page. If you have a value `x` of type `Exp (a, b)` you can use `T2 a b = x` to get values `a :: Exp a` and `b :: Exp b`. The same applies if you have a value of type `Acc (a, b)`. You can also use `fst`, `snd`, `afst` and `asnd`.

Multiple backends for Accelerate have been developed: one for the CPU, one for Nvidia GPUs and an interpreter. This allows Accelerate programs to be executed efficiently on different platforms. For this assignment it suffices to work with the interpreter. If you have an Nvidia GPU you can also try to execute your code on the GPU. To do so, you must uncomment the dependency on `accelerate-llvm` in `stack.yaml` and the dependency on `accelerate-llvm-ptx` in `package.yaml`. In `Quickhull.hs` you can then import `import Data.Array.Accelerate.LLVM.PTX` instead of the interpreter. You also need an installation of LLVM.

The CPU backend sadly does not work on Windows. You can use this backend on Linux or Mac if you want. The instructions are similar to the GPU instructions, you just have to replace PTX by Native.

Assignment

We will now make a parallel implementation of Quickhull. This assignment is split in small exercises to guide you through the algorithm. The template already has place-holders for all these exercises in `src/Quickhull.hs`.

Initial partition

We start by partitioning the array into two segments, one segment for the values above line (p_1, p_2) and one for the values below. To do so, we must first find that line and make a mapping of the indices, to be used for the permutation.

The desired values for these arrays, when executed on `input/1.in`, is shown at the last page of this document.

Exercise 1 Implement the functions `leftMostPoint, rightMostPoint :: Acc (Vector Point) -> Acc (Scalar Point)` which return the points with the minimum and maximum x coordinates. *Hint: use `fold`.*

Exercise 2 Using these points we construct the initial line on which we must partition the input. Construct arrays `isUpper, isLower :: Acc (Vector Bool)` which contain booleans denoting for each element if it is in the upper or the lower part. If a point is exactly on the line, you may choose in which part you place it. However, for points p_1 and p_2 the accompanying value in both arrays must be `False`. Use the function `pointIsLeftOfLine`, which is defined in the template.

Exercise 3 Build an array `lowerIndices :: Acc (Vector Int)` containing the relative indices of the points below the line. Field i must contain how many points in array `points`, with an index smaller than i, are below the line. *Hint: map the boolean values from `isLower` to integers and use `prescanl`.*

Exercise 4 Similarly, build an array `upperIndices :: Acc (Vector Int)` with the relative indices of the upper points. Also create a scalar `countUpper :: Acc (Scalar Int)` containing the total number of points above the line. *Hint: you can build both in a single scan using* `scanl'`*.*

Exercise 5 We will now use `lowerIndices`, `upperIndices` and `countUpper` to permute the array. Construct an array `permutation :: Acc (Vector (Z :. Int))` containing the new indices of all elements. Note that `Z :. Int` denotes a one dimensional index. You can extract the coordinate using `unindex1` and build an index with `index1`.

The indices must be choosen such that the resulting array would start with p_1, followed by all upper points, then p_2 and at last all lower points.

Exercise 6 To make the rest of the algorithm easier, we must place p_1 both at the start and the end of the array. To do so, create an array `empty` containing 1 more point than `points`, filled with p_1 on each position.

This array is used to partition `points` using `permutation`.

Exercise 7 The remainder of the algorithm uses head flags to distinguish the different segments in the array. At the start (or end) of a segment, the accompanying value in the head flags array should be `True`. After the initial partition, the head flag must be `True` for points p_1 and p_2. Construct this array `headFlags :: Acc (Vector Bool)`, which should have the same length as `newPoints`.

Utilities

As we use segments with head flags, we must write some utility functions.

Exercise 8 Implement `segmentedPostscanl, segmentedPostscanr :: Elt a => (Exp a -> Exp a -> Exp a) -> Acc (Vector Bool) -> Acc (Vector a) -> Acc (Vector a)` as segmented variants of `postscanl` and `postscanr`. They should compute a postfix scan per segment, e.g. the postfix should 'start' at a position with head flag `True`. *Hints: lift the operator from* `Exp a -> Exp a -> Exp a` *to* `Exp (Bool, a) -> Exp (Bool, a) -> Exp (Bool, a)`*. Define the functions using the standard* `postscanl` *and* `postscanr`*. You may assume that the head flag for the initial (first for a left scan, last for a right scan) value is* `True`*. You can thus pass an arbitrary value as initial value to* `postscanX` *with* `Unsafe.undef`*.*

Exercise 9 Implement `propagateL, propagateR :: Elt a => Acc (Vector Bool) -> Acc (Vector a) -> Acc (Vector a)` which propagate the value whose head flag is `True` to respectively the right or the left, until another head flag with value `True` is found. *Hint: Use* `segmentedPostscanl` *and* `segmentedPostscanr`*. What operator should you choose?*

Exercise 10 For each segment, the points at the start and end will form the line for the next step. Those points thus have head flag `True`. Write a function `propagateLine`

`:: Acc SegmentedPoints -> Acc (Vector Line)` which will for each point denote with which line it should operate. *Hint: use `propagateL`, `propagateR` and `zip`.*

Exercise 11 For some operations, we do not need the element with head flag `True`, but the element next to it. We can achieve that by shifting the head flags one position. Implement functions `shiftHeadFlagsL, shiftHeadFlagsR :: Acc (Vector Bool) -> Acc (Vector Bool)` which will respectively shift the flags one to the left or to the right. The last element in case of `shiftHeadFlagsL` and the first in case of `shiftHeadFlagsR` must be `False`. *Hint: use `generate`. Be carefull with the first or the last element, as you cannot read out of the bounds of an array.*

Partition

We can now start to work on the partition step, which will be executed repeatedly. Each segment will operate on a different line. We can find the matching line for each point with `propagateLine`. For each segment, we must find the point furthest to the line (p_1, p_2), and partition the other elements. After partitioning, the points must be ordered by first giving the points left to the line from p_1 to the furthest point, then the furthest point and then the points right from the line from p_2 to the furthest point.

Note that we already called `propagateLine`, `headFlagsL` and `headFlagsR` in the template, at the start of `partition`. You can find the desired results for these exercises at the end of this document.

Exercise 12 Construct an array `furthest :: Acc (Vector Point)` which contains for each point, the segment's furthest point. *Hint: First use `segmentedPostscanl` to put the point at maximal distance at the end of each segment. Then propagate that value to the left. Be carefull when choosing between `headFlagsL` and `headFlagsR`.*

Exercise 13 Build arrays `isLeft, isRight :: Acc (Vector Bool)`. For some point p in a segment with line (p_1, p_2) and furthest point p_f, the accompanying value in `isLeft` must be `True` if p is on the left of line from p_1 to p_f. Similarly, `isRight` must be true if p is on the right of the line from p_2 to p_f. Note that points within the triangle between p_1, p_2 and p_f will be false in both `isLeft` and `isRight`. *Hint: use `zipWithN` for some number N.*

Exercise 14 Create arrays `segmentIdxLeft, segmentIdxRight :: Acc (Vector Int)` with the relative index of the left and right elements per segment. This is similar to exercise 3 and 4, but now we need to count the number of respectively left or right elements before *and including* the current element. Implement these with a segmented postscan.

Exercise 15 Construct an array `countLeft :: Acc (Vector Int)`, containing for each point, the segment's number of left elements. *Hint: this value is already stored somewhere in `segmentIdxLeft`.*

Exercise 16 Define an array `segmentSize` containing the sizes of the segments. For an element whose head flag is true, this should be 1 and for the last element of a segment (e.g. before an element with head flag true) this should be the number of left elements plus the number of right elements plus one (for the furthest point). Using this array, build an array `segmentOffset :: Acc (Vector Int)` containing for each point the new offset of its segment. Also build a scalar `size :: Acc (Scalar Int)` containing the total size of new array, after the partitioning. *Hint: you can create `segmentOffset` and `size` with a single operation.*

Exercise 17 Using the arrays from the previous exercises, build the permutation as an array `permutation :: Acc (Vector (Z :. Int))`. If an element should not be in the resulting array, you can skip it with the magic index `ignore :: Exp (Z :. Int)`. Use `index1` to convert a number to an index. *Note: be carefull with off-by-one errors!*

Exercise 18 Create an array `newPoints :: Acc (Vector Point)` with the permuted points. To do so, first create an empty array `Acc (Vector Point)` of the desired size and use `permute const` to execute the permutation.

Exercise 19 Create the new head flags array `newHeadFlags :: Acc (Vector Bool)`, where you should mark the elements which whose head flag was previously already true and the furthest points. This can be done using `permute`, similar to the previous exercises.

Loop

To finish the algorithm, we must apply the partitioning step repeatedly.

Exercise 20 Write a function `condition :: Acc SegmentedPoints -> Acc (Scalar Bool)` which checks whether the loop should continue.

Exercise 21 Complete the algorithm by writing the function `quickhull' :: Acc (Vector Point) -> Acc (Vector Point)`. Note that you cannot use recursion to create the loop. Instead, you should use `awhile`.

Test input

You can find test input in the `input` folder. By running `stack run -- input/1.in input/2.in input/3.in input/4.in input/5.in input/6.in input/7.in` you will run the first seven inputs. Input eight is rather large, you should only run this if you have CPU or GPU backend.

General remarks

- Consider the case where three points are on a line. The middle point of these three cannot be part of the quick hull, as we search for a minimal set. Thus make sure that such point cannot be chosen as p_1 or p_2 in the initial partition nor as the furthest point. You can prevent this by implementing a tie braker which chooses the point with either the lowest or the highest coordinates.

- Make sure your program compiles using `stack build`.
- Include *useful* comments in your code. Do not paraphrase code but describe the structure of your program, special cases, preconditions, et cetera.
- Efficiency of your code may influence your grade.
- Copying solutions from the internet is not allowed.
- You may use external packages such as `containers` or `vector` which provide *non-concurrent* functions or data structures. Any concurrency related functions or data structures you must create yourself.
- We *strongly* suggest you work in a team of size two, but individual submissions are allowed. A team must submit a single assignment and put both names on it.

Submission

- Submission is done using the groups facility of Blackboard. Ensure that you and your partner sign up to a group as soon as possible; do not leave it until right before the deadline.
- To submit, run the command `stack clean --full` in the project directory (to remove the build artefacts). Make sure the directory contains the name of the group, for example `quickhull-group42`, and then create a zip file of that directory and upload it. The name of the .zip file must also include your group number, for example `quickhull-group42.zip`.
- The .zip file should contain everything the marker needs to build your project by running the command `stack build`.
- In the README.md include the group number as well as the name and student number of both group members.

Changelog

2020-01-16: Fix description of `shiftHeadFlagsL` and `shiftHeadFlagsR`

2020-01-16: Fix description of `propogateL` and `propogateR`

2020-01-07: Add description of tuples

2020-01-07: Initial release

Example output

The following tables contain the values of the intermediate arrays of the algorithm of the initial partition and the first partition of the loop. These origin from `input/1.in`.

Initial partition

This table shows the values of the arrays from exercises 1 - 7. Scalar `countUpper` is not shown in the table, its value is 7.

Index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Point	(1, 4)	(8, 19)	(5, 9)	(7, 9)	(4, 2)	(3, 9)	(9, 16)	(1, 5)	(9, 11)	(4, 0)	(8, 18)	(8, 7)	(7, 18)	(6, 18)	(4, 19)	
isUpper	False	True	False	False	False	True	False	True	False	False	True	False	True	True	True	
isLower	False	False	True	True	True	False	False	False	True	True	False	True	False	False	False	
lowerIndices	0	0	0	1	2	3	3	3	3	4	5	5	6	6	6	
upperIndices	0	0	1	1	1	1	2	2	3	3	3	4	4	5	6	
permutation	Z :. 0	Z :. 1	Z :. 9	Z :. 10	Z :. 11	Z :. 2	Z :. 8	Z :. 3	Z :. 12	Z :. 13	Z :. 4	Z :. 14	Z :. 5	Z :. 6	Z :. 7	
empty	(1, 4)	(1, 4)	(1, 4)	(1, 4)	(1, 4)	(1, 4)	(1, 4)	(1, 4)	(1, 4)	(1, 4)	(1, 4)	(1, 4)	(1, 4)	(1, 4)	(1, 4)	(1, 4)
newPoints	(1, 4)	(8, 19)	(3, 9)	(1,5)	(8, 18)	(7, 18)	(6, 18)	(4, 19)	(9, 16)	(5, 9)	(7, 9)	(4, 2)	(9, 11)	(4, 0)	(8, 7)	(1, 4)
headFlags	True	False	False	False	False	False	False	False	True	False	False	False	False	False	False	True

First partition of loop

The questionmark ? denotes an arbitrary value, $l_1 = ((1,4),(9,16))$ and $l_2 = ((9,16),(1,4))$. Furthermore, `size = Scalar Z [13]`.

Index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Point	(1, 4)	(8, 19)	(3, 9)	(1,5)	(8, 18)	(7, 18)	(6, 18)	(4, 19)	(9, 16)	(5, 9)	(7, 9)	(4, 2)	(9, 11)	(4, 0)	(8, 7)	(1, 4)
Head flag	True	False	False	False	False	False	False	False	True	False	False	False	False	False	False	True
Line	?	l_1	l_1	l_1	l_1	l_1	l_1	l_1	?	l_2	l_2	l_2	l_2	l_2	l_2	?
Furthest	?	(4,19)	(4,19)	(4,19)	(4,19)	(4,19)	(4,19)	(4,19)	?	(4,0)	(4,0)	(4,0)	(4,0)	(4,0)	(4,0)	?
isLeft	False	False	False	True	False	False	False	False	False	False	True	False	True	False	True	False
isRight	False	True	False	False	True	True	True	False	False	False	False	False	False	False	False	False
segmentIdxLeft	0	0	0	1	1	1	1	1	0	0	1	1	2	2	3	0
segmentIdxRight	0	1	1	1	2	3	4	4	0	0	0	0	0	0	0	0
countLeft	1	1	1	1	1	1	1	1	3	3	3	3	3	3	3	0
segmentSize	1	0	0	0	0	0	0	6	1	0	0	0	0	0	4	1
segmentOffset	0	1	1	1	1	1	1	1	7	8	8	8	8	8	8	12
permutation	Z :. 0	Z :. 3	ignore	Z :. 1	Z :. 4	Z :. 5	Z :. 6	Z :. 2	Z :. 7	ignore	Z :. 8	ignore	Z :. 9	Z :. 11	Z :. 10	Z :. 12