Making Woody Parallel

Programming Massively Parallel Hardware
Department of Computer Science
University of Copenhagen

Hugh McGrade <wbr412@alumni.ku.dk>
Mads Obitsø <scr411@alumni.ku.dk>
Titus Robroek <robroek@di.ku.dk>

November 2, 2018

1 Introduction

Random forests have long been a staple of machine learning. They offer a fast, reliable way of tackling classification and regression problems. Their practical ability, however, is not without bounds. Scaling up trees to huge sizes, having massive numbers of trees and processing gigantic numbers of queries all show the limits of current processing technology. This report is about utilising GPU computing to scale up decision tree processing.

In order to utilise GPU computing for decision tree processing we implemented the key part of random forest prediction - the evaluation of a single decision tree with test data - in Futhark. This implementation is intended for use as a replacement for the existing decision tree processing in the Woody random forest framework [2].

2 Motivation

This research aims to investigate ways to scale the evaluation of large decision trees on large data. Larger decision trees may be very useful for some areas of research. The field of remote sensing, for example, requires the evaluation of decision trees on large and potentially complex datasets such as satelite images [1]. The speed improvements which could be achieved by parallelising this evaluation would make it possible to apply random forest models to larger problems.

3 Woody Random Forest Prediction in Futhark

3.1 Decision tree evaluation

The evaluation of a single decision tree in a random forest prediction takes a single decision tree from the forest and a matrix of test data. From this a vector of predictions is produced, each of which corresponds to the decision made from evaluating the decision tree for a given row of the test data. These rows can either be specified by an indices array mapping decision indices to row indices, or simply by taking each row in order.

Each node of the decision tree specifies a threshold for proceeding on its left or right branch and the feature (column of the test data) which should be used to evaluate this threshold. In evaluation this continues until a leaf node is reached which is then the result.

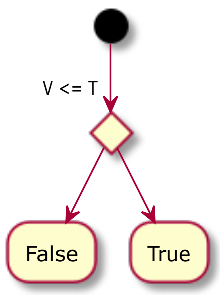

Figure 1: A non-leaf node within a tree. The query is directed based on one of it's data elements (V) tested against a specific threshold(T).

3.2 Interoperation of Woody and Futhark

The Futhark compiler features code generation for the PyOpenCL library, allowing Futhark code compiled for OpenCL to be called from Python. This generated code accepts NumPy arrays as Futhark arrays. As Woody is written in Python, we are able to compile our Futhark library for PyOpenCL and use it as a library in Woody's Python code.

3.3 Tree and test data encoding for Futhark

In order to pass the tree and test data from Woody to Futhark, we encoded each as a series of flat arrays.

In the case of the tree, this took the form of four arrays giving the properties of each node by index: `treeLeftid`, `treeRightid`, `treeFeature` and `treeThres_or_leaf`. These give the left child ID, right child ID, feature index and threshold respectively.

The test data matrix is described by a single array containing each of the test values (`Xtest`) and values `nXtest` and `dXtest` which give the shape of the matrix.

3.4 Basic Futhark implementation

We started by producing a translation of Woody's single tree query into Futhark. Woody's version is parallelised using OpenMP over the trees in the forest. The Futhark for this is as follows:

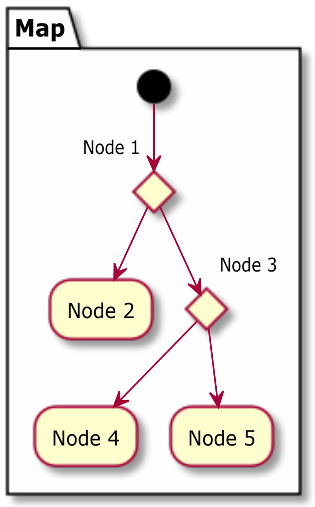

Figure 2: Our basic futhark implementation. Every query is tested on a tree by mapping over all queries.

```
unsafe map (\ i ->
let idx = if dindices > 0 then indices[i] else i
let row_start = idx * dXtest
in loop node_id = TREE_ROOT_ID
      while treeLeftid[node_id] != TREE_CHILD_ID_NOT_SET do
            if Xtest[row_start + treeFeature[node_id]] <=
                treeThres_or_leaf[node_id]
              then treeLeftid[node_id]
              else treeRightid[node_id]
           ) (iota n_preds)
```

This implementation parallelises the evaluation over the rows of the test data using a map over each prediction. The function to be mapped uses a Futhark while loop which assigns the root of the tree to `node_id` and subsequently the result of the loop body which gives the next node. The loop terminates when a leaf node is reached and thus the function returns the ID of the leaf node.

This is clearly not an idiomatic Futhark implementation and is included here only as a Futhark translation of the original C. The parellelisation approach is useful in that it distributes the trees, all of which have the same size.

3.5 Layered Futhark implementation

A different approach to parallelising tree querying in Futhark is to combine the basic implementation's `map` and `while` loop into a single `map` and repeat this sequentially for each layer of the tree. This way we can remove the `while` loop out of the program.

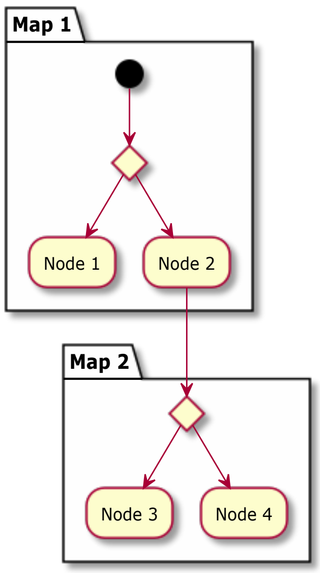

Figure 3: Our layered Futhark implementation. Every layer of the tree is executed over the queries as a map, resulting in as many maps as there are layers in the tree.

This is accomplished with an outer loop over every layer of the tree and an inner map over each node in the row. For the map, the nodes array (`node_array`) and the data rows array (`data_row_starts`) are joined using a zip and mapped with a function producing the next node for each. This new array of node IDs is then used for the next iteration.

```
loop node_array for row in iota(depth) do
    unsafe map (\ (node_id, data_row_start) ->
                 if (treeLeftid[node_id] != 0)
                    then (if Xtest[data_row_start + treeFeature
                        [node_id]] <= treeThres_or_leaf[node_id]
                        then treeLeftid[node_id]
                        else treeRightid[node_id])
                    else node_id)
                    (zip node_array data_row_starts)
```

3.6 Filtering Futhark implementation

Our layered Futhark implementation applies a map for every layer of the tree. Some queries, however, will have hit a leaf at some point. The current implementation, however, still attempts to apply a map for every query, even if it has stopped at a leaf. Our next implementation attempts to solve this issue. We want to avoid repeatedly processing those paths which have reached a leaf node. We do this by applying a filter operation on our iterations. We filter the queries based on them have reached a leaf node or not. This causes the map to only be applied to "live" paths.

Following the tree traversal a sort is applied based on the test data rows to produce a prediction for each row.

```
let nodes = zip node_array data_row_starts
let leaves = []
let (_, leaves) = loop (nodes, leaves) for row in iota(depth)
     do
                   let new_nodes = (unsafe map next_node nodes
                       )
                   in (unsafe filter is_not_leaf new_nodes,
                       leaves ++ (unsafe filter is_leaf
                           new_nodes))
let result = map (\ (a, _) -> a)
                   (radix_sort_by_key (\ (_, a) -> a)
                                       i32.num_bits
                                       i32.get_bit
                                       leaves)
in result
```

This implementation again parallelises the queries by applying a map to each row. As it prunes the tree we can expect it to perform better on larger, deeper and less balanced trees.

3.7 Treesolver Precompute

A further approach we took to implement decision tree querying in Futhark was to compute the outcome of each tree node prior to traversing the tree. This greatly increases the scope for parallelism in the evaluation of tree thresholds as every node can now be computed independently. To do this we map the function `make_next_tree` (with the tree) to every row of the data. This further maps the function `next_node` (with the row of the data) to every node of the tree, creating a direction matrix which for each node (by index) gives the following node's index. This is shown in figure 4. The tree is then trivially traversed by following these pointers from the root node.

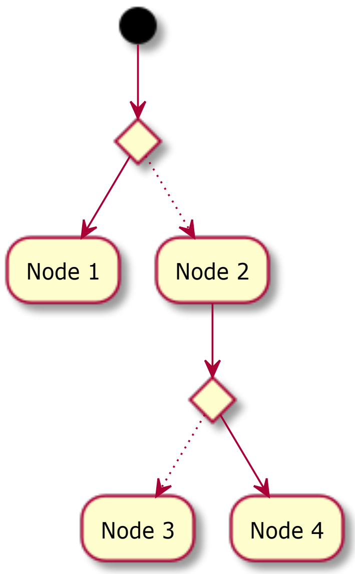

Figure 4: A visualisation of a traverse tree/matrix. Node 0 points to node 1 and node 2 points to node 4 for this specific query. Walking through the tree will thus result in this query be classified as node 1.

```
let next_node
  (row: []f64)
  ((left, right, feature, thres) : (i32, i32, i32, f64)) : i32
     =
  if row[feature] <= thres then left else right

let make_next_tree
   (tree: [](i32, i32, i32, f64))
   (row : []f64) : []i32 =
   map (next_node row) tree

let traverse
   (next_nodes: []i32): i32 =
   let (last, current) = (0, next_nodes[0])
   let (result, _) = loop (last, current) while current != 0 do
        (current, next_nodes[current])
   in result

let nodes = zip4 treeLeftid treeRightid treeFeature
    treeThres_or_leaf
let rows = unflatten nXtest dXtest Xtest

let next_nodes = unsafe map (make_next_tree nodes) rows

in unsafe map traverse next_nodes
```

We found this approach was far better suited to using an idiomatic Futhark style and made the potential for parallelisation clearer by separating tree travsersal. Future work could consider better flattening this approach.

3.8 Treesolve Matrix

A similar approach is to again parallelise by mapping over all possible nodes and their corresponding data using a series of maps. This produces a series of matrices representing properties of the tree and results in the matrix `direction` which gives the following node from each node. As in the previous implementation, this is then trivially traversed to obtain the result.

```
let repeated_criteria = flatten (replicate nXtest treeFeature)
  let repeated_offsets = flatten (map (\ i -> replicate
      treelength i) (steps 0 nXtest dXtest))
  let flcr = map2 (+) repeated_offsets repeated_criteria
  let scattered_features = unsafe map (\ i -> Xtest[i]) flcr
  let threshold_result = map2 (<=) scattered_features (flatten
      (replicate nXtest treeThres_or_leaf))
  let left_or_right = (\ b l r -> if b then l else r)
  let repeatedLeft = flatten (replicate nXtest treeLeftid)
  let repeatedRight = flatten (replicate nXtest treeRightid)
  let directions = map3 left_or_right threshold_result
      repeatedLeft repeatedRight

  in unsafe map traverse (unflatten nXtest treelength
      directions)
```

4 Experimental Setup

To evaluate our Futhark implementation, we performed various comparisons with the woody implementation of `cpu_query_tree`.

All experiments were performed on a GPU node with two Intel(R) Xeon(R) CPU E5-2650 v2 processors at 2.60GHz (32 cores, 2 threads per core) and two GeForce GTX 780 Ti graphics cards.

We wanted to both compare our predictions from Futhark to those from Woody and to measure and compare the runtimes with varying tree sizes and amount of predictions. To accomplish this, we modified parts of the Woody codebase and wrote a set of python scripts[1]. that orchestrates the testing and measuring. The scripts will generate and fit a forest based on the given training size and optional testing size, where testing size determines the number of predictions to be made. It will then do a large number of predictions with the first tree of the forest, using in turn woody and all of our Futhark pro-

[1]The scripts can be run to recreate our comparisons and measuring. They can be found here: https://github.com/HughMcGrade/woody/tree/master/experiments/parallel. Run `python launch.py` from the directory `experiments/parallel`

grams. After measuring running times, all predictions from Futhark programs are compared to those from Woody.

We measure the runtime of the Woody implementation directly in Python. The runtimes of the Futhark programs are measured using the `-t` argument to Futhark programs. This might give an unfair overhead to the woody implementation and make the result skewed towards the Futhark implementations. We call Futhark from Python through `os.system()` instead of compiling our Futhark programs to Python modules and calling them directly from Python. Using modules might give a more practical measurement for comparisons.

All programs are run 10 times and the average of those runtimes is used for the comparisons. The dataset used is the covtype dataset, already available in woody.

5 Results and Evaluation

We have included two different comparisons. Both figures show runtime comparisons of predictions on single tree, between the Woody implementation and our 5 Futhark implementations. Two of the programs, matrix and prune in the plots, fail with larger data but are included where they do run.

Figure 5 shows runtime comparisons for varying tree sizes. The training size affects the size of the tree, with small training sizes giving very small trees. All measurements in 5 were made with 116203 predictions, the size of the standard test set.

Figure 6 shows runtime comparisons for increasing amounts of predictions. The number of queries is set equal to the training size.

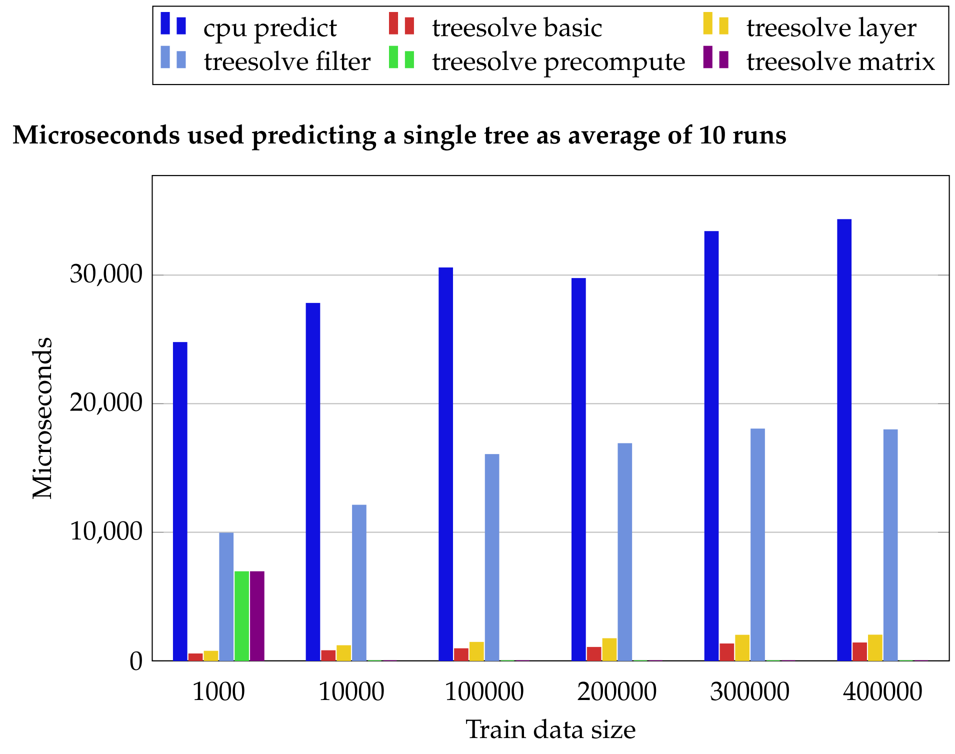

Figure 5: Measuring time to make 116203 predictions with increasing tree sizes. Two of the programs, precompute and matrix in the plots, fail with larger data but are included where they do run.

The comparison shown in 5 gives the runtimes for each of our five Futhark implementations and Woody's C implementation when varying the size of the training data producing the decision trees. Due to memory issues, the matrix and precompute implementations fail on larger data. These results show that the basic and layer implementations give substantial improvements in performance over Woody. The runtime of all programs grow slowly as the size of the

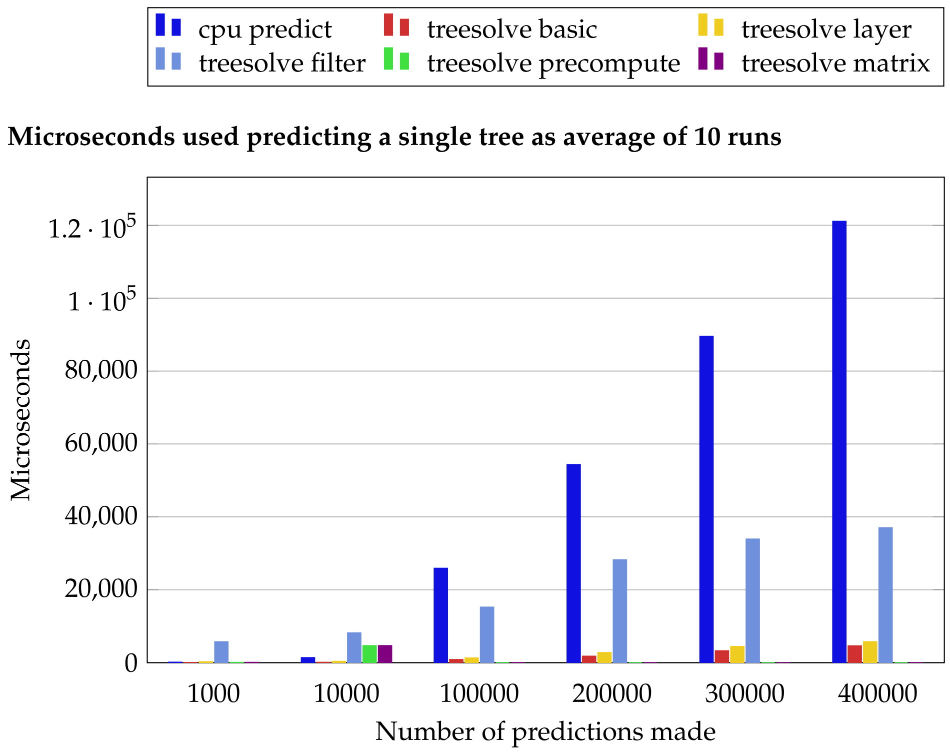

Figure 6: Measuring time to make increasing amounts of predictions with increasing tree sizes. Two of the programs, precompute and matrix in the plots, fail with larger data but are included where they do run.

training date increases.

The comparison shown in 6 gives the runtimes for each of our five Futhark implementations and Woody's C implementation when varying the number of predictions being made. These results show that the number of predictions has a far greater impact on the Woody implementation than on the Futhark implementation. The increase in Woody's runtime from 10.000 to 100.000 is quite large, where Futhark has only a marginal increase. Futhark seems to handle large tree sizes much faster than woody.

In the future, it may be interesting to future investigate these parallel implementations. For example, more sophisticated techniques may be able to further improve the performance of the algorithms. Furthermore, it may be possible to streamline the precompute and matrix implementation.

6 Conclusion

We have proposed a number of approaches to parallelising the evaluation of decision trees using Futhark. Our findings show that as a whole this parallelisation is promising for the performance of decision tree evaluation on large datasets.

References

[1] Belgiu, Mariana, and Lucian Drăguţ. "Random forest in remote sensing: A review of applications and future directions." ISPRS Journal of Photogrammetry and Remote Sensing 114 (2016): 24-31. 1

[2] Gieseke, Fabian, and Christian Igel. "Training Big Random Forests with Little Resources." arXiv preprint arXiv:1802.06394 (2018). 1