The Name of the Title is Hope

Abstract

A clear and well-documented LaTeX document is presented as an article formatted for publication by ACM in a conference proceedings or journal publication. Based on the "acmart" document class, this article presents and explains many of the common variations, as well as many of the formatting elements an author may use in the preparation of the documentation of their work.

CCS Concepts: • **Do Not Use This Code → Generate the Correct Terms for Your Paper**; *Generate the Correct Terms for Your Paper*; Generate the Correct Terms for Your Paper; Generate the Correct Terms for Your Paper.

Keywords: Do, Not, Us, This, Code, Put, the, Correct, Terms, for, Your, Paper

ACM Reference Format:
. 2018. The Name of the Title is Hope. In *Proceedings of Make sure to enter the correct conference title from your rights confirmation emai (Conference acronym 'XX)*. ACM, New York, NY, USA, 9 pages. https://doi.org/XXXXXXX.XXXXXXX

1 Motivation

Our main aim while designing TREEBEARD was to unify the diverse set of implementation strategies that have been used in existing systems for decision tree inference. Some differences in these systems are as follows:

- Decision tree inference is run on several platforms including CPUs and GPUs. The implementations used on each of these platforms are different and the techniques used to optimize them are also different.
- A diverse set techniques have been proposed for optimization of decision tree inference on CPUs and GPUs [1–3, 6? –8]. No system exists that unifies the disparate optimizations implemented in these systems.
- A very extensive design space of optimizations exists for decision tree inference outside the few that have been proposed in the literature. However, currently no system exists that is capable of exploring this space and identifying the best set of parameters to use for a given model and platform.
- Different systems use different in-memory representations for the model. For example, XGBoost uses a sparse representation, RAPIDs FIL uses what is called the reorg representation and Tahoe uses a variation of the reorg representation. Currently, systems implement inference kernels that are tied to a single representation of the model. Again, this means that no current system can explore different combinations of in-memory representations and optimizations.

At high-level, to make TREEBEARD capable of unifying these differences, we design 1) expressive intermediate representations that can represent and compose several proposed optimizations 2) a scheduling language that specifies the structure of the generated code and 3) a plugin mechanism with which different in-memory representations can be composed with different optimizations. Finally, we develop a heuristic to explore the extensive optimization space that TREEBEARD's design enables.

2 Compiler Overview

TREEBEARD takes a serialized decision tree ensemble as input (For example XGBoost JSON, ONNX etc.) and generates an optimized inference function. TREEBEARD automatically generates an optimized inference function from the serialized model and can either target CPUs or GPUs. Figure 1 shows the structure of the TREEBEARD compiler. The inference computation is lowered through three intermediate representations – high-level IR (HIR), mid-level IR (MIR) and low-level IR (LIR). The LIR is finally lowered to LLVM and then JIT'ed to the specified target processor.

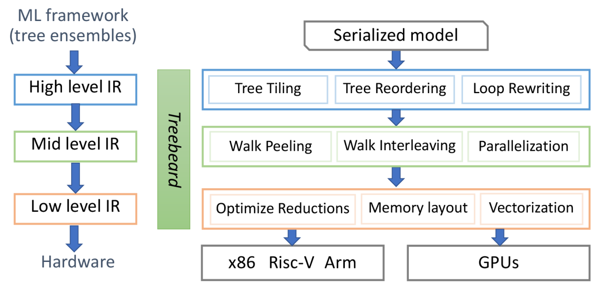
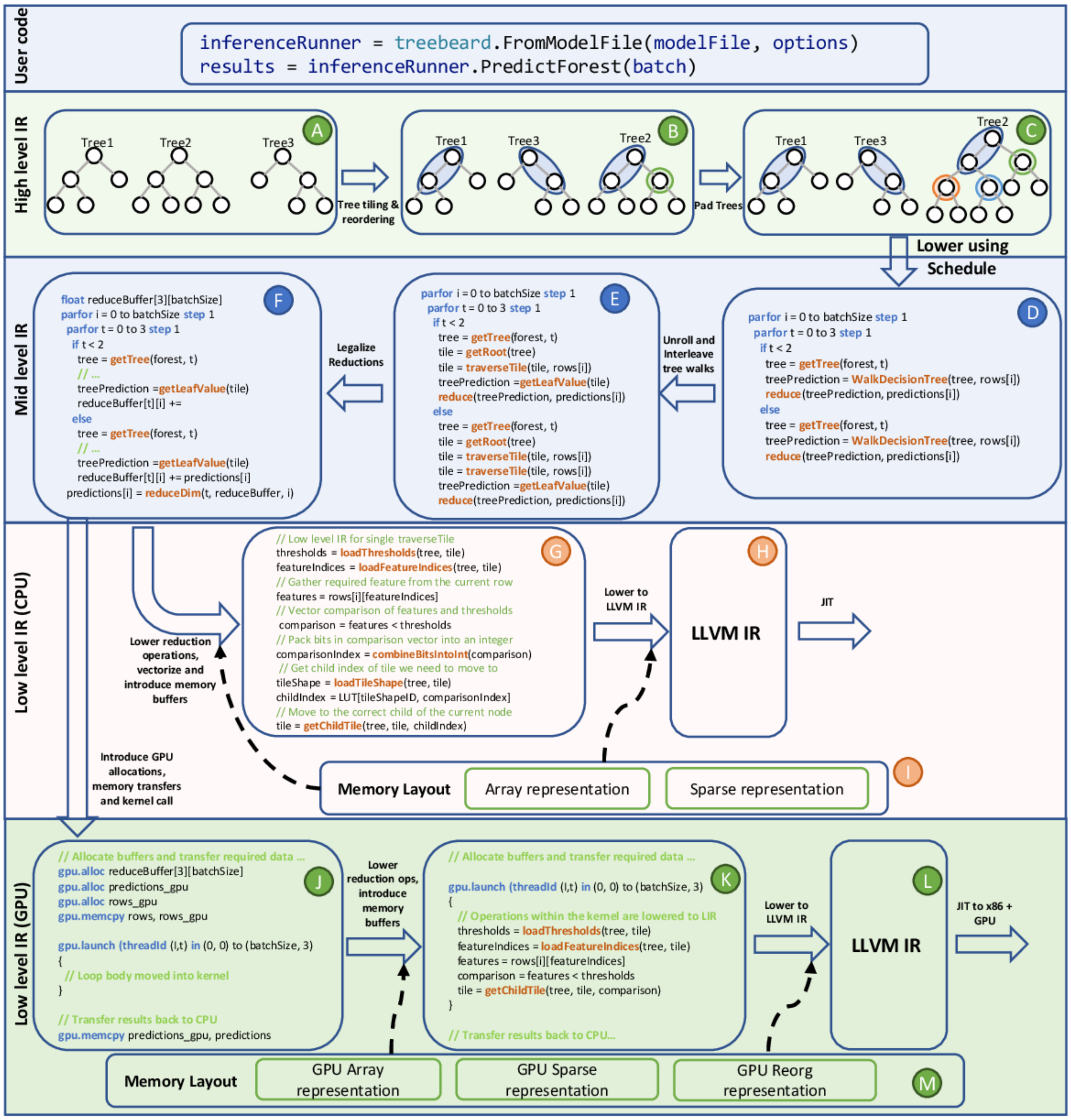


Figure 1. TREEBEARD compiler structure.

In HIR, the model is represented as a collection of binary trees. This abstraction allows the implementation of optimizations that require the manipulation of the model or its constituent trees. In Figure 2, A shows this representation



```
inferenceRunner = treebeard.FromModelFile(modelFile, options)
results = inferenceRunner.PredictForest(batch)
```

```
parfor i = 0 to batchSize step 1
 parfor t = 0 to 3 step 1
  if t < 2
    tree = getTree(forest, t)
    treePrediction = WalkDecisionTree(tree, rows[i])
    reduce(treePrediction, predictions[i])
  else
    tree = getTree(forest, t)
    treePrediction = WalkDecisionTree(tree, rows[i])
    reduce(treePrediction, predictions[i])
```

```
parfor i = 0 to batchSize step 1
 parfor t = 0 to 3 step 1
  if t < 2
    tree = getTree(forest, t)
    tile = getRoot(tree)
    tile = traverseTile(tile, rows[i])
    treePrediction =getLeafValue(tile)
    reduce(treePrediction, predictions[i])
  else
    tree = getTree(forest, t)
    tile = getRoot(tree)
    tile = traverseTile(tile, rows[i])
    tile = traverseTile(tile, rows[i])
    treePrediction =getLeafValue(tile)
    reduce(treePrediction, predictions[i])
```

```
float reduceBuffer[3][batchSize]
parfor i = 0 to batchSize step 1
 parfor t = 0 to 3 step 1
  if t < 2
   tree = getTree(forest, t)
   // ...
   treePrediction =getLeafValue(tile)
   reduceBuffer[t][i] +=
  else
   tree = getTree(forest, t)
   // ...
   treePrediction =getLeafValue(tile)
   reduceBuffer[t][i] += predictions[i]
 predictions[i] = reduceDim(t, reduceBuffer, i)
```

```
// Low level IR for single traverseTile
thresholds = loadThresholds(tree, tile)
featureIndices = loadFeatureIndices(tree, tile)
// Gather required feature from the current row
features = rows[i][featureIndices]
// Vector comparison of features and thresholds
 comparison = features < thresholds
// Pack bits in comparison vector into an integer
comparisonIndex = combineBitsIntoInt(comparison)
 // Get child index of tile we need to move to
tileShape = loadTileShape(tree, tile)
childIndex = LUT[tileShapeID, comparisonIndex]
// Move to the correct child of the current node
tile = getChildTile(tree, tile, childIndex)
```

```
// Allocate buffers and transfer required data ...
gpu.alloc reduceBuffer[3][batchSize]
gpu.alloc predictions_gpu
gpu.alloc rows_gpu
gpu.memcpy rows, rows_gpu

gpu.launch (threadId (I,t) in (0, 0) to (batchSize, 3)
{
  // Loop body moved into kernel
}

// Transfer results back to CPU
gpu.memcpy predictions_gpu, predictions
```

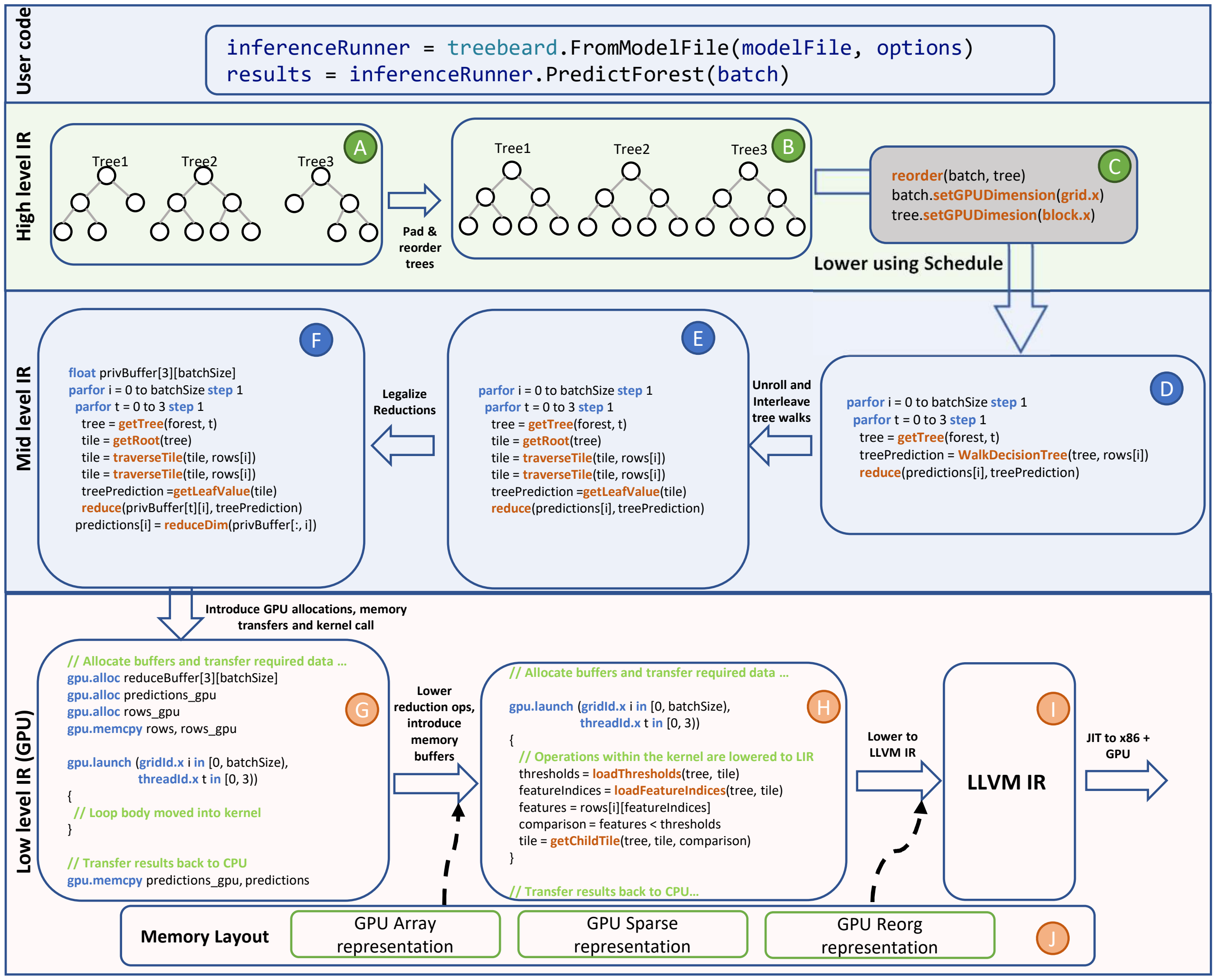
```
// Allocate buffers and transfer required data ...

gpu.launch (threadId (I,t) in (0, 0) to (batchSize, 3)
{
  // Operations within the kernel are lowered to LIR
  thresholds = loadThresholds(tree, tile)
  featureIndices = loadFeatureIndices(tree, tile)
  features = rows[i][featureIndices]
  comparison = features < thresholds
  tile = getChildTile(tree, tile, comparison)
}

// Transfer results back to CPU...
```

Figure 2. Treebeard IR lowering and optimization details: the three abstraction levels in Treebeard's IR are shown. The high level IR is a tree-based IR to perform model level optimization, the mid-level IR is for loop optimizations that are independent of memory layout and the low level IR allows us to perform vectorization and other memory layout dependent optimizations.

for a model with three trees. In HIR, Treebeard tiles tree nodes together to convert the binary tree to an n-ary tree as shown in B. Trees are also reordered to enable better code generation and padded to allow more efficient traversal as shown in C. While these are the optimizations currently implemented in Treebeard, these are by no means the only



```
inferenceRunner = treebeard.FromModelFile(modelFile, options)
results = inferenceRunner.PredictForest(batch)
```

```
reorder(batch, tree)
batch.setGPUDimension(grid.x)
tree.setGPUDimesion(block.x)
```

```
parfor i = 0 to batchSize step 1
  parfor t = 0 to 3 step 1
    tree = getTree(forest, t)
    treePrediction = WalkDecisionTree(tree, rows[i])
    reduce(predictions[i], treePrediction)
```

```
parfor i = 0 to batchSize step 1
  parfor t = 0 to 3 step 1
    tree = getTree(forest, t)
    tile = getRoot(tree)
    tile = traverseTile(tile, rows[i])
    tile = traverseTile(tile, rows[i])
    treePrediction =getLeafValue(tile)
    reduce(predictions[i], treePrediction)
```

```
float privBuffer[3][batchSize]
parfor i = 0 to batchSize step 1
  parfor t = 0 to 3 step 1
    tree = getTree(forest, t)
    tile = getRoot(tree)
    tile = traverseTile(tile, rows[i])
    tile = traverseTile(tile, rows[i])
    treePrediction =getLeafValue(tile)
    reduce(privBuffer[t][i], treePrediction)
  predictions[i] = reduceDim(privBuffer[:, i])
```

```
// Allocate buffers and transfer required data ...
gpu.alloc reduceBuffer[3][batchSize]
gpu.alloc predictions_gpu
gpu.alloc rows_gpu
gpu.memcpy rows, rows_gpu

gpu.launch (gridId.x i in [0, batchSize),
            threadId.x t in [0, 3))
{
  // Loop body moved into kernel
}

// Transfer results back to CPU
gpu.memcpy predictions_gpu, predictions
```

```
// Allocate buffers and transfer required data ...

gpu.launch (gridId.x i in [0, batchSize),
            threadId.x t in [0, 3))
{
  // Operations within the kernel are lowered to LIR
  thresholds = loadThresholds(tree, tile)
  featureIndices = loadFeatureIndices(tree, tile)
  features = rows[i][featureIndices]
  comparison = features < thresholds
  tile = getChildTile(tree, tile, comparison)
}

// Transfer results back to CPU...
```

Figure 3. Treebeard IR lowering and optimization details: the three abstraction levels in Treebeard's IR are shown. The high level IR is a tree-based IR to perform model level optimization, the mid-level IR is for loop optimizations that are independent of memory layout and the low level IR allows us to perform vectorization and other memory layout dependent optimizations.

ones that are enabled by HIR. It is not difficult to imagine optimizations such as tree pruning for specified accuracy levels for example. These optimizations and rewrites that are performed on the domain-specific HIR would have been much on a traditional loop based IR or even in other IRs within Treebeard.

After these model-level optimizations are performed on the HIR, the code is lowered to the mid-level IR (MIR) as dictated by a user specified schedule (C to D) in Figure 2). The schedule specifies how the iteration space that goes over the trees and input rows is to be traversed. It specifies how the iteration space is to be tiled, which loops are to be parallelized, which loops are to be mapped to GPU grid and block dimensions etc. (Details in Section ??). MIR is a loop-based IR that explicitly encodes details of the iteration space has to be traversed. However, it still abstracts details about the in-memory representation of the model. Optimizations such as tree-walk unrolling and interleaving are performed on the MIR (E). Subsequently, reduction operations are split and rewritten to correctly and explicitly implement reduction in the presence of parallel loops (F). More details of this process that we call *legalization* are in Section ??. Another important point to note is that MIR is independent of the target processor and therefore all optimizations on MIR can be reused across CPU and GPU compilation.

The MIR is then further lowered to a low-level IR (LIR). This is the level at which the compilation pipeline diverges for CPUs and GPUs. In the GPU compilation pipeline, the required memory transfers and kernel invocations are inserted into the LIR (J). Additionally, buffers to hold model values are inserted and tree operations are lowered to explicitly refer to these buffers. This lowering is controlled by a plugin mechanism where different in-memory representations can be added to the compiler by implementing an interface. These plugins provide information required for the lowering of MIR to LIR as well as the lowering to LLVM IR as shown in Figure 2. Again, a significant amount of code is shared between the CPU and GPU pipelines for representations that are common between them (Array and Sparse representations). Vectorization of tree traversals is also explicitly represented in LIR.

To reiterate, the following are the salient points of Treebeard's design.

1. The compiler uses three intermediate representations (IRs) to represent the inference computation at different levels of abstraction. This allows different optimizations to be performed and also allows us to share infrastructure between compilation pipelines for different target processors.
2. The specification of how the inference computation is to be lowered to loops is not encoded directly in the compiler. Instead, this is specified as an input to the compiler using a scheduling language that is specialized for decision tree inference computations (Section ??). This separation allows us to build optimizations and schedule exploration mechanisms independent of the core compiler (Section ??). The scheduling language also exposes details like parallelization, optimization of reductions (vectorization, use atomics, shared memory on GPUs etc.).
3. Treebeard has been designed to keep the optimization passes and code generator independent of the in-memory representation finally used for the model. To achieve this, Treebeard specifies an interface to implement that provides the necessary capabilities to the code generator as a plugin. This interface abstracts several details on how model values are stored. In particular, it abstracts the actual layout of the memory buffers, how to load model values like thresholds and feature indices, how to move from a node to its children and determining whether a node is a leaf. This design allows us to write each memory representation as a standalone plugin and reuse the rest of the compiler infrastructure.

3 Scheduling Language

The goal of the scheduling language is to express the following

- The order in which the iteration space over the batch of inputs (batch) and trees in the forest (tree) is to be traversed. Note that there is a reduction over the tree dimension.
- Intra or inter tree optimizations that are to be performed on a tree or set of trees (tree walk unrolling, pipelining, SIMDize etc).

The reasons to use a scheduling language rather than a hard-coded lowering are as follows

- Making the scheduling specification external to the compiler allows us to more easily build auto-schedulers and auto-tuners.
- It is very hard to come up with a template loop nest that works for all models (for example, tree sizes may vary across the model making it necessary to iterate different number of trees at different times).
- A scheduling language will make writing newer locality optimizations faster since no changes to the compiler infrastructure will be needed.
- Adding support for additional hardware targets (GPUs, FPGAs), will be much easier with a scheduling language.

There are some simplifying assumptions and limitations in the current design

- Tree traversals are considered atomic. There is no way to express partial tree traversals or schedule individual node/level computations.
- Accumulation of tree predictions is done immediately (as opposed to, for example, collecting all predictions and performing a reduction later).

3.1 Language Definition

We broadly have three classes of directives in the language. The first is a set of loop nest modifiers that are used to specify the structure of the loop nest to walk the iteration space. The second is a set of clauses that specify intra and inter tree optimizations. Finally, we have a class of attributes that control how reductions are performed.

3.1.1 Loop Modifiers. There are two special index variables – `batch` and `tree`. The clauses modify these index variables or index variables derived from these (through the application of clauses).

- **tile**: Tile the passed index variable using a fixed tile size.
- **split**: Split the range of the passed index variable into two parts. The range of the first part is specified by an argument.
- **unroll**: Unroll an index completely
- **reorder**: Reorder the specified indices. The specified indices must be successive indices in the current loop nest.

- **specialize**: Generate separate code for each iteration of the specified index variable. This is useful while parallelizing across trees and these trees have different depths.
- **gpuDimension**: Maps the specified index variable to represent a dimension in either the GPU kernel grid or thread block.

The default loop order (as currently generated by the compiler) is (batch, tree), i.e, for each row in the input batch, go over all trees.

The following are examples of how the loop modifiers can be used.

- The loop order used by XGBoost[3] is (tree, batch) – walk one tree for all inputs in the batch before moving to the next tree. The corresponding schedule would be

```
reorder(tree, batch)
```

- The below schedule computes 2 trees at a time over the whole batch.

```
tile(tree, t0, t1, 2)
reorder(t0, batch, t1)
```

- If we additionally only want to compute over 4 input rows (rather than the whole batch) for every 2 tree, and then move onto the next 2 trees for the same set of inputs, then the schedule is as follows.

```
tile(batch, b0, b1, 4)
tile(tree, t0, t1, 2)
reorder(b0, t0, b1, t1)
```

3.1.2 Optimizations.

The following clauses provide ways to optimize the inference routine being generated.

- **cache**: Cache the working set of one iteration of the specified loop corresponding to this index. This can be specified on either batch or tree loops. Specifying it on a batch loop leads to all rows accessed in a single iteration of the loop being cached. Similarly, specifying it on a tree loop leads to all trees accessed in one iteration of that loop being cached.
- **parallel**: Execute the loop corresponding to this index in parallel.
- **interleave**: Interleave the execution of the tree walks within the current index (must be applied on an inner most index).
- **unrollWalk**: Unroll tree walks at the current index.
- **peelWalk**: Peel the first n steps of the specified tree walk and don't check for leaves for that number of steps.

3.1.3 Reduction Optimization.

- **atomicReduce**: Use atomic memory operations to accumulate values across parallel iterations of the specified loop.
- **sharedReduce**: Only applies to GPU compilation. Specifies that intermediate results are to be stored in shared memory.
- **vectorReduce**: Use vector instructions with the specified vector width to reduce intermediate values across parallel iterations of the specified loop.

TODO Add examples for RAPIDS, Tahoe (Maybe show some strategies can be encoded?)

3.2 The XBoost Schedule

XGBoost[3] is a very popular gradient boosting library. It implements inference on the CPU by going over a fixed number of rows (64 in the previous version) for every tree and then moving to the next tree. When all trees have been walked for this set of rows, the next set of rows is taken up. Also, different sets of rows are processed in parallel.

The schedule used by XGBoost can be represented in TREEBEARD's scheduling language as follows.

```
tile(batch, b0, b1, CHUNK_SIZE)
reorder(b0, tree, b1)
parallel(b0)
```

3.3 Tahoe Schedules

Tahoe[8] has four strategies that it picks from for a given model. Each of these strategies can be encoded using TREEBEARD's scheduling language as we show below.

- **Direct Method**: In this strategy, a single GPU thread walks all trees for a given input row. The schedule for this strategy is as follows.

```
tile(batch, b0, b1, ROWS_PER_TB)
reorder(b0, b1, tree)
gpuDimension(b0, grid.x)
gpuDimension(b1, block.x)
```

  Here, ROWS_PER_TB is the number of rows that are processed by a single thread block.
- **Shared Data**: In this strategy, a thread block walks all the trees for a given row in parallel. If threads walk multiple trees, each thread accumulates partial results. Finally, a thread block wide reduction is performed to compute the prediction. The schedule for this strategy is as follows.

```
reorder(batch, tree)
gpuDimension(batch, grid.x)
gpuDimension(tree, block.x)
cache(batch)
```

- **Shared Forest**: In this strategy, the whole model is loaded into shared memory and subsequently, a single thread walks all trees for a particular row. The schedule for this strategy is as follows.

```
tile(batch, b0, b1, ROWS_PER_TB)
tile(tree, t0, t1, N_TREES)
reorder(b0, b1, t0, t1)
cache(t0)
gpuDimension(b0, grid.x)
gpuDimension(b1, block.x)
```

Here, we create a placeholder single iteration loop `t0` so that we can specify that all trees are to be cached.

- **Shared Partial Forest**: In case the model is too large to fit into shared memory, the model is split into chunks and each chunk is loaded into shared memory. Again, as in the previous strategy, one thread walks all trees assigned to a thread block for a row. The schedule for this strategy is as follows.

```
tile(batch, b0, b1, ROWS_PER_TB)

tile(tree, t0, t0Inner, TREES_PER_TB)
tile(t0Inner, t1, t2, TREES_PER_TB)
cache(t1)
reorder(b0, t0, b1, t1, t2)

gpuDimension(b0, grid.x)
gpuDimension(t0, grid.y)
gpuDimension(b1, block.x)
```

4 Reductions : Representation, Optimization and Lowering

Currently, existing reduction support in MLIR is insufficient to code generate and optimize the reductions Treebeard needs to perform while performing inference (sum up individual tree predictions to compute the prediction of the model). MLIR only supports reductions of value types and cannot directly represent and optimize inplace reductions of several elements of a memory buffer.

We design a mechanism to specify accumulating values into an element of a multi-dimensional array inplace. The accumulation is performed inside an arbitrary loop nest where several surrounding loops maybe parallel and the ultimate target machine maybe a CPU or a GPU. Because this problem is of general interest, we design this as an MLIR dialect.

The main abstraction we introduce is the `reduce` op. It models atomically accumulating values into an element of a multi-dimensional array (represented by a MLIR `memref`). The following example sums up the elements of the 1D memref `arr` into the first element of the memref `result`. It does this using in two concurrent iterations of a surrounding parallel loop.

```
builtin.func @ReduceVector(%arr: memref<num_elemsxf64>,
    %result: memref<1xf64>) -> void {
  par.for i0 in range(0 : num_elems/2 : num_elems) {
    for i1 in range(0 : num_elems/2)
      reduce(%result, 0, arr[i0 + i1]) <"+", 0.0>
  }
}
```

The semantics of the `reduce` op guarantee that all elements are correctly added and that there is no race between the parallel iterations of the loop.

The `reduce` op is defined for all associative and commutative reduction operations with a well-defined initial value. The reduction operator and the initial value are attributes applied on the `reduce` op.

The main differences between our `reduce` and the existing reductions in MLIR are the following: 1. Existing reductions only support scalar, by value reductions. It does not support accumulating inplace into memref elements. 2. Existing reduction support in MLIR does not provide a unified way to handle reductions in GPUs and CPUs. To the best of our knowledge, there is currently no way to model reductions on GPUs in MLIR. 2. Existing reductions have strict rules about where they can be written. For example, `scf.reduce` needs to be an immediate child `scf.parallel`. However, our `reduce` op can be generated anywhere in the loop nest.

Having modeled the reductions with an abstract op, the aim now is to lower this to a correct and optimized implementation on both CPU and GPU. In order to do this, we make the following observations. 1. The `reduce` op has a loop carried dependency on itself and loop carried dependences on other `reduce` ops that accumulate into the same target array. A simple lowering to a sequence of load-add-store instructions is incorrect if any of these dependences are carried by a parallel loop. We call any such surrounding parallel loop a **conflicting loop** (TODO Change the name!) for the reduction. 2. There is a race between the parallel iterations of such a loop when naively accumulating values into target memref elements. To avoid this race, the result memref can be **privatized** wrt each surrounding conflicting loop. Subsequently, each privatized dimension can be reduced at the end of the conflicting loop it was inserted for. TODO We cannot do better than this in terms of memory usage TODO Need a proof.

Definition 4.1. A parallel loop surrounding one or more `reduce` ops is a **conflicting loop** for a target multi-dimensional array if this loop has a non-zero dependence distance for the dependence between any of the contained `reduce` ops.

In the context of Treebeard, this set of loops is exactly the set of surrounding parallel loops that are iterating over trees. The results can be privatized for each conflicting loop iteratively and reductions along each privatized dimension can be inserted immediately following the loop the dimension was inserted due to.

We illustrate this process through the example above. The `i0` loop is a conflicting loop for the reduction into the `result` array. We would therefore privatize the `result` memref for each iteration of the `i0` loop.

```
builtin.func @ReduceVector(%arr: memref<num_elemsxf64>,
    %result: memref<1xf64>) -> void {
  results_1 = memref<2x1xf64>
  par.for i0 = range(0 : num_elems/2 : num_elems) {
    for i1 = range(0 : num_elems/2)
      reduce(%result_1[i0/(num_elems/2), 0], arr[i0 +
   i1]) <"+", 0.0>
  }
  results = reduce_dimension(results_1, 0) <"+", 0.0>
}
```

The op `reduce_dimension` reduces values across the specified dimension of an n-dimensional memref. In the above example, the `reduce_dimension` op is reducing across all elements of the first dimension (index 0). Therefore, in this case, it produces a result memref with a single element (the first dimension with size 2 is collapsed).

Definition 4.2. **`reduce_dimension(targetMemref, memref, dim, [indices], [rangeStart], [rangeEnd])`**: Computes the reduction over the dimension specified by `dimension` and stores the result in `targetMemref`. `[indices]` must be a vector of `dim` elements (or empty if the dimension being reduced is the first dimension). `[rangeStart]` and `[rangeEnd]` represent the range of indices following the reduction dimension and must have the same number of elements. If both are `null` (not passed), all elements of these dimensions are reduced. The computation performed by the op is as follows.

$$targetMemref[\vec{indices}, \vec{k}] = \sum_{i=0}^{shape[dim]} memref[\vec{indices}, i, \vec{k}] \quad \forall \vec{k} \in [[rangeStart_0, rangeEnd_0), ..., [rangeStart_n, rangeEnd_n)]$$

Consider the following code with nested parallel loops. (A situation where trees are split across both threads and thread blocks could result in such generated code in TREEBEARD.)

```
builtin.func @ReduceVector(%arr: memref<num_elemsxf64>,
    %result: memref<1xf64>) -> void {
  par.for i0 = range(0 : num_elems/2 : num_elems) {
    par.for i1 = range(0 : num_elems/4 : num_elems/2) {
      for i2 = range(0 : num_elems/4)
        reduce(%result, 0, arr[i0 + i1 + i2]) <"+",
  0.0>
    }
  }
}
```

Here, the `i0` and `i1` loops are conflicting loops wrt the `result` memref. We **legalize** the reduction by privatizing the `result` array wrt the `i0` and `i1` loops. However, there are now two privatized dimensions and therefore, two dimensions need to be reduced to compute the final result. This multi-stage reduction is what enables us to model hierarchical reductions.

The following code shows how the reduction above is legalized. We introduce a new op, `reduce_dimension_inplace` which reduces a dimension of the input memref and stores results in the same array. This helps saves memory by removing the need to create multiple intermediate arrays to store results. Only the final dimension reduction uses the `reduce_dimension` op.

```
builtin.func @ReduceVector(%arr: memref<num_elemsxf64>,
    %result: memref<1xf64>) -> void {
  results_1 = memref<2x2x1xf64>
  par.for i0 = range(0 : num_elems/2 : num_elems) {
    par.for i1 = range(0 : num_elems/4 : num_elems/2) {
      for i2 = range(0 : num_elems/4)
        index0 = i0/(num_elems/2)
        index1 = i1/(num_elems/4)
        reduce(%result_1[index0, index1, 0], arr[i0 +
  i1 + i2]) <"+", 0.0>
    }
    // result_1[i0/(num_elems/2), 0] = sum(result_1[i0
  /(num_elems/2), :])
      reduce_dimension_inplace(%result_1, 1, i0/(
    num_elems/2))
  }
  // result = sum(result[:, 0])
  %result = reduce_dimension(%result_1, 0)
}
```

The behavior of the `reduce_dimension_inplace` op is similar to the `reduce_dimension` op except that it updates the input array inplace rather than writing results to a target array. The definition of the op is as follows.

Definition 4.3. **`reduce_dimension_inplace(memref, dim, [indices], [rangeStart], [rangeEnd])`**: Computes the reduction over the dimension specified by `dimension` and stores the result at index 0 of that dimension. `[indices]` must be a vector of `dim` elements (or empty if the dimension being reduced is the first dimension). `[rangeStart]` and `[rangeEnd]` must have the same number of elements. If both are `null` (not passed), all elements of the corresponding dimension are reduced.

The computation performed by the op is defined by the following equation.

$$memref[\vec{indices}, 0, \vec{k}] = \sum_{i=0}^{shape[dim]} memref[\vec{indices}, i, \vec{k}] \quad \forall \vec{k} \in [[rangeStart_0, rangeEnd_0), ..., [rangeStart_n, rangeEnd_n)]$$

4.1 Lowering Reduction Operations

We implement lowering of the operations defined above to both the CPU and GPU. Since the compilation pipeline diverges after the reductions are legalized, we can implement lowering and optimization of our reduction dialect to CPUs and GPUs simply using different MLIR rewrite patterns. We now briefly describe how these operations are lowered to the CPU and GPU.

4.1.1 Lowering to CPU. The lowering of the reduction operations to CPU is fairly straightforward. We lower the two operations listed above, `reduce_dimension_inplace` and `reduce_dimension` to a simple loop nest that goes over the specified subset of the input array, performs the reduction and writes the result into the appropriate location of the target array. If the schedule specifies that the reduction is to be vectorized, then as many elements as specified by the vector width are read from the input array as a vector, accumulated as a vector, and finally written back to the target array. In general, this works well because reductions are typically being performed on dimensions other than the inner-most dimension and therefore, this strategy loads successive elements from memory maximizing memory bandwidth utilization.

TODO explain atomic reduction

4.1.2 Lowering to GPU. The lowering on GPU is slightly more involved than the lowering on CPUs. However, we can lower the same abstractions to efficient implementations and therefore simplify high-level code generation. The lowering for the inplace and non-inplace operations are essentially the same, except for the target array and we do not distinguish between them except for finally storing the result.

The lowering of the `reduce_dimension_*` ops is distinct from existing work on implementing reductions efficiently on GPUs [?] because our abstractions potentially represent several independent reductions (independent for different output elements). Therefore, we can either exploit parallelism across the independent reductions or the inherent parallelism in the reduction by performing a divide and conquer reduction.

The reduction pass for GPU can follow one of two paths. If the lowering pass determines that there are enough independent reductions to keep all threads in a thread block busy, then it simply generates code that performs one (or multiple) reductions completely in a thread. If however there are not enough independent reductions, then the lowering pass generates a tree style reduction where multiple threads cooperate to perform a single reduction using inter-thread shuffles.

Another feature specific to GPU reductions is the use of shared memory. If the schedule specifies that the reduction needs to be performed using shared memory, the privatized buffer is allocated in shared memory. Also, the compiler ensures that only as much shared memory is allocated as needed to hold values processed by a single thread-block and index offsets are appropriately rewritten to handle the differences between the indexing of the target memref and the shared memory array. Our abstractions allow our lowering passes to be written completely independent of whether we use shared memory and therefore allow us to enable or disable shared memory use independently from the other parts of the compiler.

4.2 Use in Treebeard

We now present an example specific to Treebeard. The schedule with which code is generated is as below. `N_t` is the number of trees and `batch_size` is the batch size. The schedule tiles both the batch loop and the tree loop and parallelizes the outer batch and tree loops.

```
IndexVar i0, i1, t0, t1;
auto& batch = schedule.GetBatchIndex();
auto& tree = schedule.GetTreeIndex();
schedule.Tile(batch, i0, i1, batch_size/2);
schedule.Tile(tree, t0, t1, N_t/2);
schedule.Reorder({i0, t0, t1, i1});
schedule.Parallel(t0);
schedule.Parallel(i0);
```

The loop-nest generated by Treebeard for the above schedule is as follows.

```
  builtin.func @Prediction_Function(%arg0: memref<
     batch_sizexnum_featuresxf64>) -> memref<
     batch_sizexf64> {
    %result = memref.alloc <batch_sizexf64>
    %0 = #decisionforest<ReductionType = 0, #Trees = N_t,
      resultType = memref<batch_sizexf64>>
    par.for i0 = range(0 : batch_size/2 : batch_size) {
      par.for t0 = range(0 : N_t/2 : N_t) {
        for t1 = range(0 : N_t/2) {
          for i1 = range(0 : batch_size/2) {
            %2 = GetTree(%0, t0 + t1)
            %3 = WalkDecisionTree(%2, %arg0[i0+i1])
            reduce(%result, i0+i1, %3)
          }
        }
      }
    }
  }
```

Treebeard determines that the `t0` loop is a conflicting loop for the `result` array and therefore legalizes the reduction by inserting a privatized array `result_1`. The privatized dimension of this array is reduced at the end of the `t0` loop.

```
  builtin.func @Prediction_Function(%arg0: memref<
     batch_sizexnum_featuresxf64>) -> memref<
     batch_sizexf64> {
    %result = memref.alloc <batch_sizexf64>
    %result_1 = memref.alloc <2xbatch_sizexf64>
    %0 = #decisionforest<ReductionType = 0, #Trees = N_t,
      resultType = memref<batch_sizexf64>>
    par.for i0 = range(0 : batch_size/2 : batch_size) {
      par.for t0 = range(0 : N_t/2 : N_t) {
        for t1 = range(0 : N_t/2) {
          for i1 = range(0 : batch_size/2) {
            %2 = GetTree(%0, t0 + t1)
            %3 = WalkDecisionTree(%2, %arg0[i0+i1])
            reduce(%result, i0+i1, %3)
          }
        }
      }
      %result[i0 : i0+step] = reduce_dimension(%result_1,
      0, i0 : i0+step)
    }
  }
```

While legalizing the reduction, the compiler determines that the `reduce_dimension` operation must only process a subset of the final result that is computed within the current parallel iteration of the `i0` loop. Once this process is complete, the `reduce` ops in the result IR can be lowered to a simple "read-accumulate-write" sequence of instructions

Finally, we note that in our experiments, we found that our current implementation of lowering the reduction operations was sufficient and reduction is not the bottleneck in our generated code. However, we believe this approach to enabling higher level code generators to easily generate reductions through simple abstractions and then having the compiler automatically lower them to efficient implementation is an important area for future work with applicability in several domains.

TODO Should we mention how we handle multi-class models?

5 Citations and Bibliographies

Artifacts: [5] and [4].

References

[1] [n. d.]. Treelite : model compiler for decision tree ensembles. https://treelite.readthedocs.io/en/latest/. Accessed: 2022-04-16.

[2] Nima Asadi, Jimmy Lin, and Arjen P. de Vries. 2014. Runtime Optimizations for Tree-Based Machine Learning Models. *IEEE Transactions on Knowledge and Data Engineering* 26, 9 (2014), 2281–2292. https://doi.org/10.1109/TKDE.2013.73

[3] Tianqi Chen and Carlos Guestrin. 2016. XGBoost: A Scalable Tree Boosting System. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* (San Francisco, California, USA) *(KDD '16)*. Association for Computing Machinery, New York, NY, USA, 785–794. https://doi.org/10.1145/2939672.2939785

[4] C. Lattner and V. Adve. 2004. LLVM: a compilation framework for lifelong program analysis amp; transformation. In *International Symposium on Code Generation and Optimization, 2004. CGO 2004.* 75–86. https://doi.org/10.1109/CGO.2004.1281665

[5] Chris Lattner, Mehdi Amini, Uday Bondhugula, Albert Cohen, Andy Davis, Jacques Pienaar, River Riddle, Tatiana Shpeisman, Nicolas Vasilache, and Oleksandr Zinenko. 2021. MLIR: Scaling Compiler Infrastructure for Domain Specific Computation. In *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. 2–14. https://doi.org/10.1109/CGO51591.2021.9370308

[6] Claudio Lucchese, Franco Maria Nardini, Salvatore Orlando, Raffaele Perego, Nicola Tonellotto, and Rossano Venturini. 2015. QuickScorer: A Fast Algorithm to Rank Documents with Additive Ensembles of Regression Trees. In *Proceedings of the 38th International ACM SIGIR Conference on Research and Development in Information Retrieval* (Santiago, Chile) *(SIGIR '15)*. Association for Computing Machinery, New York, NY, USA, 73–82. https://doi.org/10.1145/2766462.2767733

[7] Supun Nakandala, Karla Saur, Gyeong-In Yu, Konstantinos Karanasos, Carlo Curino, Markus Weimer, and Matteo Interlandi. 2020. A Tensor Compiler for Unified Machine Learning Prediction Serving. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. USENIX Association, 899–917. https://www.usenix.org/conference/osdi20/presentation/nakandala

[8] Zhen Xie, Wenqian Dong, Jiawen Liu, Hang Liu, and Dong Li. 2021. Tahoe: Tree Structure-Aware High Performance Inference Engine for Decision Tree Ensemble on GPU. In *Proceedings of the Sixteenth European Conference on Computer Systems* (Online Event, United Kingdom) *(EuroSys '21)*. Association for Computing Machinery, New York, NY, USA, 426–440. https://doi.org/10.1145/3447786.3456251

Received 20 February 2007; revised 12 March 2009; accepted 5 June 2009