SilvanForge : A Schedule Guided Retargetable Compiler for Decision Tree Inference

Abstract

The rapid proliferation of machine learning coupled with accelerating evolution of the hardware ecosystem has led to a surge in the demand for model inference on a variety of hardware. This paper is motivated by the problems encountered when targeting inference of decision tree based models, the most popular models on tabular data, to run at peak performance on diverse CPU and GPU targets. We evaluated existing solutions and found that they do not provide portable performance across different hardware targets.

To address this we present the design of SilvanForge, a schedule guided compiler for decision tree based models that searches over a large design space to automatically generate high-performance inference routines. SilvanForge has two core components. A scheduling language that encapsulates the large optimization space for decision tree inference, and techniques to efficiently explore this space. TODO Change large optimization space. Second, an optimizing multi-level compiler that can generate code based on the schedule. For the latter, we re-architect the open-source Treebeard CPU compiler to support schedule guided compilation and add support for GPU code generation. GPU code generation required fundamental new optimization interfaces for caching, parallel reduction, and support for multiple in-memory representations of trees.

We evaluate SilvanForge on over seven hundred diverse models and demonstrate that the best schedule varies drastically with model, batch size, and target hardware. Our scheduling heuristic is able to quickly find near optimal schedules while searching over a small number ($\tilde{5}0$) of schedules. In terms of performance, SilvanForge generated code is an order of magnitude faster than XGBoost and about 2-3× faster on average than RAPIDs FIL and Tahoe. While these systems only target NVIDIA GPUs, SilvanForge achieves competent performance on AMD GPUs as well. On CPUs, SilvanForge achieves better scaling compared to Treebeard. For models where Treebeard was only able to achieve diminishing returns with an increasing number of threads, SilvanForge is able to scale linearly with the number of threads. TODO (numbers for CPU performance?)

ACM Reference Format:

. 2018. SilvanForge : A Schedule Guided Retargetable Compiler for Decision Tree Inference. In *Proceedings of Make sure to enter the correct conference title from your rights confirmation emai (Conference acronym 'XX)*. ACM, New York, NY, USA, 17 pages. https://doi.org/XXXXXXX.XXXXXXX

1 Introduction

We are in the midst of a hardware revolution, a new golden age for computer architecture [11]. The last decade has seen a shift in architectural paradigms, with the rise of GPUs and accelerators. This shift has been driven by the necessity to innovate in the post Moore's law and Dennard's scaling era. This transformation has also played a significant role in the success of modern deep learning models, as they enable scaling model training and inference to a massive number of threads. Such scalability would be essential for all performance critical applications, including other machine learning models that need to scale with increasing data sizes and model complexities.

Decision forest models remain the mainstay for machine learning over tabular data [10, 29]. Their robustness, interpretability, and ability to handle missing data make them a popular choice for a wide range of applications. TODO a few more lines, classification, regression, etc. A recent survey [**?**]. . . . The survey also finds that, the cost of inference is the most critical factor in the overall cost of deploying a machine learning model. This is because, in production settings, each model is trained once and often used for inference millions of times. However, inference is run on a variety of hardware platforms, ranging from low to high-end CPUs and GPUs. This paper is motivated by the need to accelerate decision tree inference to achieve portable performance on a variety of hardware.

Decision forest models are composed of a large collection of decision trees (100-1000), and inference involves traversing down each tree in the forest and aggregating the predictions. Inference is typically done in a batched setting, where multiple inputs are processed simultaneously. Despite the simplicity of the model and the availability of multiple sources of coarse grain parallelism (parallelism across inputs in a batch and parallelism across trees), existing systems do not consistently scale well across different models even on the limited set of targets they support.

Evaluation on a diverse set of models highlights that the best implementation often requires a careful composition of many optimization strategies like data layout optimizations, loop transformations, parallelization, and memory access optimizations. Existing systems today are mostly library based, and only support a predefined combination of optimizations. XGBoost [6] uses a sparse representation for the model and a loop structure that processes one tree for a block of rows before moving to the next tree. RAPIDS FIL [**?**] uses a reorg representation and partitions trees across a fixed number

of threads. Tahoe [35] uses a variation of the reorg representation and has four predefined inference strategies from which it picks one based on an analytical model. Treebeard, the state-of-the-art model compiler for CPUs, supports two fixed loop structures and does not scale well with increasing number of threads. Additionally, it lacks GPU specific optimizations that are critical to scale performance to massive number of threads.

This paper presents SilvanForge, a novel schedule guided compilation infrastructure for decision tree inference on multiple target hardware. SilvanForge is able to generate high-performance code for decision tree inference by exploring a large optimization space. The code generation is guided by a *schedule* written in a custom scheduling language that can represent a wide range of implementation strategies. We demonstrate that the language is sufficient to express the various optimizations proposed by prior work, and further generalizes them and incorporates several new optimizations. We also design and implement a heuristic that is able to quickly find high-performance schedules for the model being compiled. TODO Performance evaluation summary TODO Oganization

1.1 Contributions

- We present the design for a multi-target decision tree compiler infrastructure and implement several optimizations within this framework. We are also the first to implement an optimizing compiler for decision tree inference on GPUs.
- We identify that an extensive optimization space exists for the problem of decision tree inference. We design a scheduling language that allows us to effectively represent this solution space abstractly. This scheduling language is expressive enough to represent a wide range of implementation strategies proposed by prior work.
- To the best of our knowledge, we perform the first extensive characterization of the optimization space for decision tree inference on GPUs. Using some of the characteristics we identify, we design and implement a heuristic that is able to quickly find high-performance schedules for the model being compiled.
- We design and implement a general framework for expressing and optimizing reductions within MLIR. To the best of our knowledge, this is the first such framework.
- We evaluate our implementation by comparing it against RAPIDs and Tahoe, the state-of-the-art decision tree inference frameworks for GPU and report significant speedups. We also show that our compiler can effectively target different GPUs, including both NVIDIA and AMD GPUs.

2 Motivation

While a diverse set techniques have been proposed for optimization of decision tree inference on CPUs and GPUs [3, 4, 6, 19, 21, 35**?**], a very extensive design space of optimizations exists outside what has been proposed in the literature. Furthermore, 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 different. To make matters even more complicated, several in-memory representations have been proposed for decision tree models. For example, XGBoost[6] uses a sparse representation, RAPIDs FIL[**?**] uses what is called the reorg representation and Tahoe uses a variation of the reorg representation. TODO Can we add some numbers here to show that different models/batch sizes need different optimizations?

To solve the problems of exploring the design space of optimizations for decision tree inference and enabling portable performance, we build several techniques in SilvanForge, an open source compiler infrastructure for decision tree inference. To make SilvanForge capable of unifying these different techniques and targets, we do the following.

- We design a scheduling language that encapsulates various optimization techniques and controls the structure of the generated code.
- We design an MLIR dialect to represent and optimize reductions and use this dialect within SilvanForge to enable the generation of different variants of inference routines.
- We extend SilvanForge's intermediate representations to include operations like caching. We were able to easily reuse and extend SilvanForge's IR as it was built as an MLIR dialect.
- We design a plugin mechanism with which different in-memory representations can be composed with different optimizations.

3 Compiler Overview

SilvanForge takes a serialized decision tree ensemble as input (E.g.: XGBoost JSON, ONNX etc.) and automatically generates an optimized inference function. SilvanForge automatically generates an optimized inference function from the serialized model and can either target CPUs or GPUs. Figure 1 shows the structure of the SilvanForge 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.

Table 1 lists the operations in the three IRs. 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

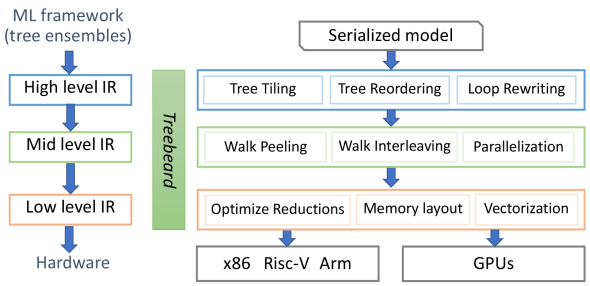


Figure 1. SilvanForge compiler structure.

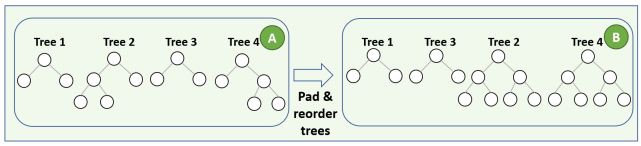


Figure 2. The representation of a model in high-level IR and the model after trees are padded and reordered.

trees. Figure 2 shows this representation for a model with four trees and how the model can be transformed. The model contains two trees of depth 1 and two trees of depth 2 (A). The right side of the diagram shows the models after padding and reordering (B). Padding makes all leaves in a tree the same depth (Trees 2 and 4 are padded). Trees are reordered so that trees of the same depth are together (Trees 2 and 3 are swapped). We use this model as a running example for the rest of the section. After these model-level optimizations are performed on the HIR, the code is lowered to the mid-level IR (MIR) as dictated by a ***schedule***. The schedule determines how to traverse the iteration space that goes over the trees and input rows by specifying how loops are to be tiled, parallelized, mapped to GPU grid and block dimensions etc. (Details in Section 4). While MIR is a loop-based IR that explicitly encodes details of the iteration space has to be traversed, it still abstracts details about the in-memory representation of the model.

In the lowered MIR, the compiler uses the `reduce` op from the `reduction` dialect we design (details in Section 6) to represent reduction operations. The lowering of the `reduce` operation involves introducing temporary buffers and splitting the reduction to correctly implement reduction in the presence of parallel loops. This process, that we call ***legalization***, is described in Section 6.

The MIR is then further lowered to a low-level IR (LIR). This is the level at which the compilation pipeline diverges for different targets. In the GPU compilation pipeline, the required memory transfers and kernel invocations are inserted into the LIR. Additionally, buffers to hold model values are inserted and abstract tree operations are lowered to explicitly refer to these buffers. This lowering is controlled by a plugin mechanism which enables different in-memory representations to be added to the compiler by implementing an interface (Section 7). Vectorization of tree traversals is also explicitly represented in LIR.

In the remainder of this section, we show using our running example from Figure 2 how different schedules can be used to lower the model to MIR. We also show how the `reduce` operation is lowered and legalized.

First, we consider the schedule that processes one tree at a time for all input rows and unrolls all tree walks. We assume that the trees have been reordered so that all trees with the same depth are together as shown in Figure 2. The schedule splits the loop over the trees into two loops – one that iterates over the first two trees (Trees 1 and 3 with depth 1) and the second that iterates over the last two trees (Trees 2 and 4 with depth 2). The schedule then unrolls the tree walks for each tree.

```
reorder(tree, batch)
split(tree, t0, t1, 2)
unrollWalk(t0, 1)
unrollWalk(t1, 2)
```

SilvanForge represents loops as nodes in a tree where the children of a node represent immediately contained loops. Each schedule primitive modifies this tree in some way. For example, the `split` primitive above duplicates the subtree rooted at the node that is being split. The compiler tracks the lineage of each of the loops. This allows the compiler to automatically infer the ranges for all loops. To lower the HIR to MIR, the compiler traverses the tree of loops and generates the appropriate loops. The MIR for the example model, generated using the above schedule is as follows.

```
model = ensemble(...)
for t0 = 0 to 2 step 1 {
  T = getTree(ensemble, t0)
  for batch = 0 to BATCH_SIZE step 1 {
    treePred = walkDecisionTree(T,
                  input[batch]) <unrollDepth = 1>
    reduce(result[batch], treePred)
  }
}
for t1 = 2 to 4 step 1 {
  T = getTree(ensemble, t0)
  for batch = 0 to BATCH_SIZE step 1 {
    treePred = walkDecisionTree(T,
                  input[batch]) <unrollDepth = 2>
    reduce(result[batch], treePred) <'+', 0.0>
  }
}
```

Subsequent lowering rewrites the `walkDecisionTree` operations to a series of `traverseTile` operations – one for the first instance and two for the second (as specified by the unroll depth attribute). SilvanForge also determines that the `reduce` operation can be implemented by a simple addition operation as there are no surrounding parallel loops (The legalization of reductions is described in detail in Section 6).

We now show how easily the same model can be compiled to target a GPU using SilvanForge's scheduling language. A possible schedule to accomplish this is as follows.

```
tile(tree, t0, t1, 2)
reorder(batch, t1, t0)
split(t0, t0_1, t0_2, 2)
unrollWalk(t0_1, 1)
unrollWalk(t0_2, 2)
gpuDimension(batch, grid.x)
gpuDimension(t1, block.x)
```

This schedule processes one input row per thread block (since the `batch` loop is mapped directly to `grid.x`). It also splits the trees into two sets by tiling the `tree` loop. Each of the two sets is processed in parallel. We unroll the tree walks for each tree. The MIR generated is as follows.

```
model = ensemble(...)
par.for batch = 0 to BATCH_SIZE step 1 <grid.x> {
  par.for t1 = 0 to 2 step 1 <block.x> {
    for t0_1 = 0 to 2 step 2 {
      T = getTree(ensemble, t0_1 + t1)
      treePred = walkDecisionTree(T,
                    input[batch]) <unrollDepth = 1>
      reduce(result[batch], treePred)
    }
    for t0_2 = 2 to 4 step 2 {
      T = getTree(ensemble, t0_2 + t1)
      treePred = walkDecisionTree(T,
                    input[batch]) <unrollDepth = 2>
      reduce(result[batch], treePred) <'+', 0.0>
    }
  }
}
```

In the case of this schedule, the lowering passes for the `reduce` operation determines that parallel iterations of the `t1` loop accumulate into the same element of the `result` array. Therefore, the reduction is rewritten so that each parallel iteration accumulates into a different array element by introducing a temporary buffer (`tempResults`) as follows.

```
float tempResults[2][BATCH_SIZE]
model = ensemble(...)
par.for batch = 0 to BATCH_SIZE step 1 <grid.x> {
  par.for t1 = 0 to 2 step 1 <block.x> {
    for t0_1 = 0 to 2 step 2 {
      T = getTree(ensemble, t0_1 + t1)
      treePred = walkDecisionTree(T,
                    input[batch]) <unrollDepth = 1>
      reduce(tempResults[t1][batch], treePred)
    }
    for t0_2 = 2 to 4 step 2 {
      T = getTree(ensemble, t0_2 + t1)
      treePred = walkDecisionTree(T,
                    input[batch]) <unrollDepth = 2>
      reduce(tempResults[t1][batch], treePred) <'+',
   0.0>
    }
  }
  result[batch] = reduce_dimension(tempResults[:][batch
   ], 0)
}
```

Here, partial results are accumulated into `tempResults` and then reduced across the `t1` dimension (represented by the `reduce_dimension` operation) to get the final result. Finally, since this computation is being targeted to the GPU, the parallel loops are rewritten into kernel calls and the required GPU memory allocations and memory transfers are introduced.

As is evident from these examples, the structure of the loop nest in the inference routine can get quite complex even for simple schedules. Writing these routines by hand is error-prone and time-consuming. Building a principled code generator to automatically generate these routines is the best way to explore the vast design space. The key features in the design of SilvanForge that enable us to build such a code generator are as follows.

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 4). This separation allows us to build optimizations and schedule exploration mechanisms independent of the core compiler (Section 9).
3. SilvanForge 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, SilvanForge 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 (Details in Section 7). This design allows us to write each in-memory representation as a standalone plugin and reuse the rest of the compiler infrastructure. TODO Should we also say it lets us easily search over multiple representations?

4 Scheduling Language

As we articulated in Section 1, there are several different configurations and optimizations strategies for decision tree inference. The best ones are significantly different across models, batch sizes and hardware platforms. Therefore, designing any one hard-coded lowering strategy is not feasible as this would make portable performance impossible. To address this problem, we design a scheduling language for SilvanForge. The scheduling language provides an abstract way to specify loop structure and other optimizations as an input to the compiler. Making the scheduling specification external to the compiler allows to build auto-schedulers and auto-tuners (Section 9). Using a scheduling language will also significantly simplify adding support for new hardware

Operation	Inputs	Outputs	Attributes	Description
`predictEnsemble`	**rows[]**	**result**	**ensemble** **predicate** **schedule**	Performs inference on the data in **rows[]** using the model specified by the **ensemble** attribute. The **schedule** attribute contains the schedule described in Section 4. **predicate** specifies the operator to use to evaluate nodes (Eg: $<$, $\leq$).
`walkDecisionTree`	**trees[]** **rows[]**	**results[]**	**predicate** **unrollDepth**	Represents an interleaved walk on all the element-wise pairs of **trees** and **rows**. **unrollDepth** specifies the number of hops to unroll. An array of tree walk results is returned.
`ensemble`		**ensemble**	**model**	Represents the forest of trees that constitute the model. The **model** attribute contains the actual trees model.
`getTree`	**ensemble** **treeIndex**	**tree**		Get the tree at the specified index (**treeIndex**) from the **ensemble**.
`getTreeClassId`	**ensemble** **treeIndex**	**classId**		Get the class ID for the tree at index **treeIndex** in the **ensemble**. This is used for multi-class models.
`getRoot`	**tree**	**rootNode**		Get the root node of the specified tree.
`isLeaf`	**tree** **node**	**bool**		Returns a boolean value indicating whether **node** is a leaf of **tree**.
`getLeafValue`	**tree** **node**	**value**		Returns the value of the leaf **node** in **tree**.
`traverseTreeTile`	**trees[]** **nodes[]** **rows[]**	**nodes[]**	**predicate**	Represents an interleaved traversal of the nodes in **nodes** based on the data in **rows**. **predicate** specifies the operator to use to evaluate nodes.
`cacheTrees`	**ensemble** **start** **end**	**ensemble**		Cache the trees in the **ensemble** between the specified **start** and **end** indices. The returned **ensemble** has the specified trees cached.
`cacheRows`	**rows[]** **start** **end**	**cachedRows[]**		Cache the rows in **rows[]** between the specified **start** and **end** indices. Returns an array of cached rows **cachedRows[]**.
`loadThreshold`	**buffer** **treeIndex** **nodeIndex**	**threshold**		Load the threshold value for the node specified by **nodeIndex** in the tree specified by **treeIndex** from **buffer**. Returns the loaded threshold.
`loadFeatureIndex`	**buffer** **treeIndex** **nodeIndex**	**threshold**		Load the feature index for the node specified by **nodeIndex** in the tree specified by **treeIndex** from **buffer**. Returns the loaded feature index.

Table 1. List of all the operations in the SilvanForge MLIR dialect. These operations are used in conjunction with operations from other MLIR dialects like `scf`, `arith`, gpu etc. to represent and optimize decision tree inference.

as this will likely require different locality optimizations and loop transformations.

The goal of SilvanForge's scheduling language is to declaratively express loop structures and the application of other optimizations (tree walk unrolling, tree walk interleaving etc.).

4.1 Language Definition

The core construct of SilvanForge's scheduling language is an ***index variable*** which abstractly represents a loop. The language then provides directives to manipulate these index variables. There are two special index variables – `batch` and `tree` that are used to represent the batch and tree loops and all other index variables are derived from these. A schedule derives new index variables from these root index variables by applying directives.

SilvanForge's scheduling language has three classes of directives. The first is a set of loop modifiers that are used to specify the structure of the loop nest to walk the iteration space (Table 2). The second is a set of directives that enable optimizations on a loop (Table 3). Finally, we have a class of attributes that enable reduction specific optimizations (Table 4).

Directive	Inputs	Description
`tile`	**indexVar outer inner tileSize**	Tile the loop corresponding to **indexVar** with the specified tile size. Resulting loops will be represented by **outer** and **inner**.
`split`	**indexVar first second splitIter**	Fiss the loop represented by **indexVar** at iteration **splitIter**. Resulting loops will be represented by **first** and **second**. Returns a maps from nested index variables to new ones created by splitting.
`reorder`	**indices[]**	Permute loops corresponding to the specified index variables. The loops must be perfectly nested in the current loop structure.
`specialize`	**indexVar**	Generate specialized code for each iteration of the loop. Useful when different iterations of a loop need to execute different code.
`gpuDimension`	**indexVar gpuDim**	Map the passed index variable to a dimension of either the grid or thread block.

Table 2. List of all the loop modifiers in SilvanForge's scheduling language. We use *index variable* and *loop* interchangeably in descriptions for clarity of exposition.

Directive	Inputs	Description
`cache`	**indexVar**	Cache the working set of one iteration of the specified loop. Cache rows for a batch loop and trees for a tree loop.
`parallel`	**indexVar**	Execute the iterations of the specified loop in parallel.
`interleave`	**indexVar**	Interleave tree walks within the specified loop (must be innermost loop).
`unrollWalk`	**indexVar unrollDepth**	Unroll tree walks at the specified loop for **unrollDepth** hops. Loop must be an innermost loop.

Table 3. List of optimization directives in SilvanForge's scheduling language. We use *index variable* and *loop* interchangeably in descriptions for clarity of exposition.

Directive	Inputs	Description
`atomicReduce`	**indexVar**	Use atomic memory operations to accumulate values across parallel iterations of the specified loop.
`sharedReduce`	**indexVar**	Specifies that intermediate results are to be stored in shared memory (GPU only).
`vectorReduce`	**indexVar width**	Use vector instructions with the specified vector width to reduce intermediate values across parallel iterations of the specified loop.

Table 4. List of reduction optimization directives in SilvanForge's scheduling language. We use *index variable* and *loop* interchangeably in descriptions for clarity of exposition.

We now show how the implementation strategies of several existing systems can be encoded in SilvanForge's scheduling language. First, we note that the default loop order is [`batch`, `tree`], i.e, for each row in the input batch, go over all trees.

XGBoost[6] 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. This schedule can be implemented in SilvanForge's scheduling language as follows.

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

Tahoe[35] has four strategies for inference on the GPU that it picks from for a given model. We show how two of these strategies can be encoded using SilvanForge's scheduling language. The rest can be encoded similarly.

- **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.

```
1 tile(batch, b0, b1, ROWS_PER_TB)
2 reorder(b0, b1, tree)
3 gpuDimension(b0, grid.x)
4 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)
```

There are some simplifying assumptions and limitations in the current design of the scheduling language. Mainly, tree traversals are considered atomic and accumulation of tree predictions is done immediately (as opposed to, for example, collecting all predictions and performing a reduction later). However, we find that these are not significant limitations in practice as the current design is able to express most strategies of interest.

5 HIR and MIR Optimizations

The Treebeard infrastructure was originally designed to target CPUs [24]. However, it implements several optimizations on the HIR and MIR that can be leveraged across target processors and we find that some these are beneficial for GPUs as well. This reuse of optimizations is possible because the intermediate representations on which these optimizations are performed are abstract and are designed to be target-independent. We briefly review these optimizations below.

5.1 Optimizations on High-Level IR

We augment the existing Treebeard infrastructure with loop rewrites on the HIR that are implemented through the scheduling language (Section 4). We use these to implement the automatic scheduling described in Section 9. Additionally, the Treebeard infrastructure implements HIR transformations to reorder and pad trees. It also implements tree tiling transformations on the HIR [24]. We reuse the reordering and padding transformations on the HIR for GPUs. However, we found that tiling trees was not beneficial for GPUs. This is because the tiling transformation introduces redundant computation inorder to vectorize computation on CPUs where all lanes need to follow the same control flow. However, on SIMT GPUs, we find that the benefits of tiling (coalescing memory accesses) do not outweigh the cost of redundant computation. We leave an investigation of this for future work.

5.2 Optimizations on Mid-Level IR

The original Treebeard infrastructure implements optimizations like tree-walk unrolling, tree-walk interleaving, and parallelization on the MIR. These optimizations are beneficial for GPUs as well. and the design of Treebeard allows us to reuse the tree-walk unrolling and tree-walk interleaving optimizations on the MIR for GPUs.

While building SilvanForge, we found that one of the performance bottlenecks on the GPU was that warps spent significant time being stalled. Since GPUs implement scoreboarding [**?**], we were able to alleviate this bottleneck by interleaving tree walks. This significantly improved performance of generated code. We found it surprising that the use of ILP could benefit performance on the GPU. TODO Should we talk about how interleaving is implemented as a statemachine and therefore it can be used across representations and tile traversal techniques?

5.3 A Note on Low-level IR

Significant changes to the original Treebeard design were required to get LIR to correctly lower to GPU code. The most important of these was the change to how the compiler implements support for in-memory representations of models (Section 7). With these design changes, we were able to reuse much of the CPU implementation while customizing some parts for GPUs (for example, buffers need to be allocated differently for CPU and GPU, caching is implemented differently etc.).

6 Reductions : Representation, Optimization and Lowering

Currently, existing reduction support in MLIR is insufficient to code generate and optimize the reductions SilvanForge 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 6.1. 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 SilvanForge, 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 6.2. **`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 SilvanForge.)

```
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 6.3. **`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.

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

6.1 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.

6.1.1 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

6.1.2 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.

6.2 Use in SilvanForge

We now present an example specific to SilvanForge. 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 SilvanForge 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)
          }
        }
      }
    }
  }
```

SilvanForge 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?

7 Representations

The design of the SilvanForge compiler allows the implementation of different strategies for the in-memory representation of the model. The compiler currently has implementations for the three representations shown in Figure 3. The array and sparse representations are the ones described in the SilvanForge paper[24]. The reorg representation is the representation used by the RAPIDs library[?]. The **array representation** is the simplest representation where the trees are stored in an array in level order. The **sparse representation** stores the trees in a sparse format where only the non-zero nodes are stored and nodes contain pointers to their children. The **reorg representation** interleaves the array representation of each tree in the model: all root nodes are stored first, then the left children of all the roots and so on.

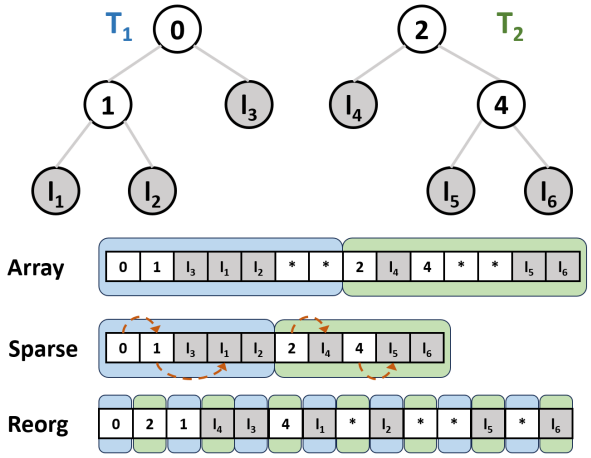


Figure 3. The three representations supported by SilvanForge.

One of the major changes we make to the original design of SilvanForge [24] is to separate the implementation of representations from the rest of the compiler. This allows us to implement representations as plugins to the compiler. We do this by defining an interface that each representation needs to implement. The code generator is implemented using methods on the representation interface thus hiding details of the actual representation from the core compiler. The interface abstracts the following details.

- **Buffer allocation:** The representation object exposes methods that generate buffer allocations in the IR.
- **Moving to child nodes:** Given the value of the predicate at a node (or tile), generate code to move to the appropriate child node of the current node.
- **Leaf representation:** The interface abstracts determining whether the current node is a leaf (which is needed to terminate walks) and how to get the value of the leaf (which is needed to get a tree's prediction).
- **Caching trees:** Since reading the trees into shared memory on GPU (or prefetching) them on CPU require knowledge of the buffer layout, the task of generating caching code for trees is delegated to the representation object.

- **Loading thresholds and feature indices:** The representation object provides MLIR rewrite patterns to lower operations that load thresholds and feature indices to LLVM IR. This is necessary because the details of how to load the thresholds and feature indices from the model buffers is representation-specific.

In summary, the representation interface abstracts the details of how the model is stored in memory and allows the compiler to generate code without having to explicitly know the details of the representation. This design allows us to implement new representations without changing the core compiler infrastructure. Implementing the representations as plugins also allows us to reuse the implementations across different lowering pipelines. For example, the code for the array and sparse representations is almost fully shared between the CPU and GPU lowering pipelines.

8 Caching Implementation

SilvanForge provides mechanisms to cache both trees and input rows on both the CPU and GPU. As described in Section 4, the user can specify that the working set of an iteration of an index variable needs to be cached using the `Cache()` directive. This provides a unified way to specify caching of both trees and input rows.

SilvanForge implements caching at the granularity of a tree or a row. Also, the semantics of caching depends on the target processor. For the CPU, caching is implemented as prefetching, while for the GPU, caching is implemented using shared memory.

8.1 IR Representation of Caching

Caching is encoded in the mid-level IR using the `cacheTrees` and `cacheRows` operations. These operations are generated when the HIR lowered to MIR and `Cache()` is specified on an index variable in the schedule. While the HIR is being lowered and a cached index variable is encountered, the compiler generates a `cacheTrees` or `cacheRows` operation depending on whether the index variable is a tree or a batch index variable. Additionally, as a part of the lowering process, SilvanForge determines the working set of the loop with caching enabled and generates a caching operation with the appropriate limits.

Each of the caching operations take parameters that specify the set of trees or rows that need to be cached. The caching operations are defined as follows.

- **`cacheTrees(forest, start, end)`:** This operation caches the trees in ensemble `forest` from `start` to `end`. The trees are cached in the order in which they are specified in the ensemble.
- **`cacheRows(data, start, end)`:** This operation caches the rows in the input array `data` from `start` to `end`. The rows are cached in the same order as in the input array.

8.2 Lowering of Caching Operations

When the MIR is lowered to LIR, the cache ops are lowered to target-specific code. Each of the two caching operations is lowered differently for the CPU and the GPU.

For the `cacheRows` operation, SilvanForge uses pre-implemented lowerings for both CPU and GPU. This is possible because the input is currently assumed to be a dense array format. Therefore, regardless of any other configuration choices (like what representation is used for the model itself), the lowering of the `cacheRows` operation is the same. For the CPU, the `cacheRows` operation is lowered to prefetches. For the GPU, a series of coalesced loads read the rows into shared memory.

For the `cacheTrees` operation, the lowering is representation-specific. Each representation provides a lowering to the target-specific code generator to lower the `cacheTrees` op when that representation is used. However, the SilvanForge infrastructure does provide helpers to generate caching code to cache contiguous regions of memory. These helpers are reused as required across different representations.

9 Exploring the Schedule Space

The set of schedules that can be constructed using the scheduling language described in Section 4 is vast. Searching this schedule space to find a schedule that provides good performance is a non-trivial task. To simplify this process, we identify a template schedule for GPUs that encompasses several strategies published in prior work. Our template schedule assigns a fixed number of rows to each thread block and to each thread. It distributes the trees across a fixed number of threads and can cache trees and input rows if required. Unrolling and interleaving of tree walks is also supported.

The template schedule exposes a number of parameters that can be tuned to find a high-performance schedule.

- **Number of rows per thread block (Integer):** The number of rows that are processed by each thread block.
- **Number of rows per thread (Integer):** The number of rows processed by each thread.
- **Number of tree threads (Integer):** The number of threads across which the trees are distributed.
- **Cache rows (Boolean):** Whether the input rows are cached in shared memory.
- **Cache trees (Boolean):** Whether the trees are cached in shared memory.
- **Unroll walks (Boolean):** Whether the tree walks are unrolled.
- **Tree walk interleave factor:** The number of tree walks that are interleaved.
- **Shared memory reduction:** Whether the reduction across tree threads is done in shared memory.

While the template schedule simplifies optimization of generated inference code, it is important to note that the

SilvanForge compiler itself does not place any restrictions on the schedule. The user is free to specify any schedule they wish. The auto-scheduler that implements the template schedule is implemented as a module outside the core SilvanForge compiler. Users are also free to implement other auto-schedulers that generate schedules different from the template schedule.

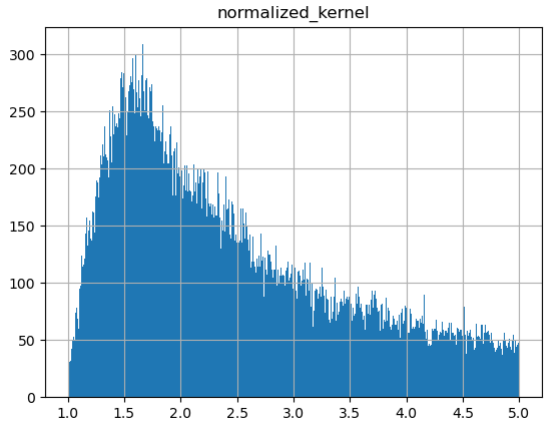


Figure 4. Distribution of normalized execution times for all benchmark models with different parameter values for the template schedule. TODO We need to describe exactly what the considered set of parameter values are.

Figure 4 shows the distribution of normalized execution times for all benchmark models with different parameter values for the template schedule. The normalized execution time is the inference time of the model with a given set of parameter values divided by the best inference time of the model. The histogram shows that even within the variants of the template schedule, there is a significant amount of variation in performance. Very few schedules perform close to the best while a vast majority of schedules perform poorly.

Exploring the schedule space extensively even for a reasonable set of parameter values is very expensive. We explored the set of parameter values listed in Table **??** for our benchmarks and found that it took anywhere between thirty minutes up to a few hours to explore the entire space. Performing this extensive search for every model being compiled is infeasible in practise. We therefore need a better mechanism to guide the search for a good schedule.

We design a heuristic to narrow down the set of schedules to explore based on the following observations on high-performance schedules.

- For small batch sizes, the best schedules tend to have a small number of rows per thread block and partition the trees across a larger number of threads. This is intuitive since the amount of data parallelism across the rows is limited for small batch sizes.
- Always cache rows in shared memory and never cache trees. We find that caching rows when possible (i.e., when the number of features is small enough to fit in shared memory) almost always improves performance. Caching trees on the other hand almost always degrades performance. This is because the one time cost of loading trees into shared memory is not sufficiently amortized when the whole of the tree is not accessed during inference.
- Models with a large number of features tend to benefit from partitioning the trees across more threads even at larger batch sizes. This is because processing fewer rows at a time allows us to keep them in shared memory. We empirically find that the threshold for when we should start partitioning the trees across more threads is when the number of features is greater than 100.
- We find that when a model prefers schedules with shared reduction, the same schedules without shared reduction are among the best performing schedules without shared reduction. We therefore are able to separate the evaluation of shared reduction by collecting the best schedules without shared reduction and only evaluating shared reduction on them. Evaluating the top 3 schedules for shared reduction is sufficient in practice.

SilvanForge uses these observations to narrow down the set of schedules to explore. The pseudo-code for the current heuristic is shown in Algorithm 1. The algorithm first computes a subset of thread block configurations in the function `TBConfigs`. A set of schedules based on these thread block configurations is then computed (`schedules`). The model is compiled with each of these schedules and then the resulting inference code is profiled. The three best performing schedules are collected and shared reduction is enabled on them and the resulting schedules evaluated. The best schedule among all the evaluated schedules is selected as the schedule to use. We find that this heuristic is able to find schedules that are close to the best schedules but improves the search time by two orders of magnitude as we show in Section **??**.

10 Experimental Evaluation

TODO

- CPU numbers for tree parallelization
- Tahoe on T400
- Sensitivity analysis for schedules. But for what specifically? Models, batch sizes?

11 Related Work

While several optimization strategies for decision tree based models have been studied in the literature, to the best of our knowledge, no systems that are capable of exploring the

Algorithm 1 Heuristic to guide the search for a good schedule

procedure TBConfigs(N_{batch}, N_f)
  $T_{batch} \leftarrow 2048, T_f \leftarrow 128$
  if $N_{batch} \leq T_{batch}$ **then**
    $rowsPerBlock \leftarrow \{8, 32\}$
    $treeThreads \leftarrow \{20, 50\}$
  else
    if $N_f \leq T_f$ **then**
      $rowsPerBlock \leftarrow \{32, 64\}$
      $treeThreads \leftarrow \{2, 10\}$
    else
      $rowsPerBlock \leftarrow \{8, 32\}$
      $treeThreads \leftarrow \{20, 50\}$
    end if
  end if
  return $rowsPerBlock, treeThreads$
end procedure

$bestSchedules \leftarrow PriorityQueue(3)$
$rowsPerTB, treeThreads \leftarrow TBConfigs(N_{batch}, N_f)$
$cacheRows \leftarrow$ True
$cacheTrees \leftarrow$ False
$interleaveFactors \leftarrow \{1, 2, 4\}$
$reps \leftarrow \{array, sparse, reorg\}$
$schedules \leftarrow (rowsPerTB, treeThreads,$
  $cacheRows, cacheTrees,$
  $interleaveFactors)$
for $(sched, rep) \in schedules \times reps$ **do**
  $time \leftarrow EvaluateSchedule(sched, rep)$
  $bestSchedules.insert(time, sched, rep)$
end for
$shMemSchedules \leftarrow \emptyset$
for $sched, rep \in bestSchedules$ **do**
  $EnableSharedReduction(sched)$
  $time \leftarrow EvaluateSchedule(sched, rep)$
  $shMemSchedules.insert(time, sched, rep)$
end for
return $min(shMemSchedules \cup bestSchedules)$

extensive optimization space exist. We describe related work and compare these systems to SilvanForge in this section.

Decision Tree Inference Systems for GPUs: Tahoe[35] is a system that implements high-performance library routines and a performance model for tree inference on GPUs. Tahoe is a library-based system that picks between four predefined strategies to implement decision tree inference on GPUs. In comparison, SilvanForge explores a much larger set of implementation options because it is a compiler. SilvanForge can also explore different in-memory representations for models. Also, SilvanForge generates code that is specific to a particular model, specializing both the parallelism (by deciding the thread block structure on a per model basis)

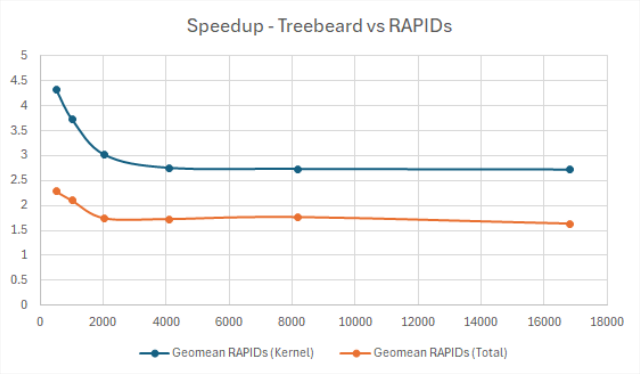


Figure 5. SilvanForge vs RAPIDs Speedup on NVIDIA RTX 4060.

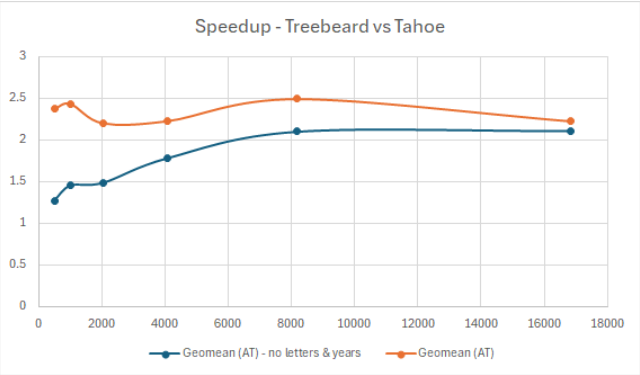


Figure 6. SilvanForge vs Tahoe Kernel Time Speedup on NVIDIA RTX 4060.

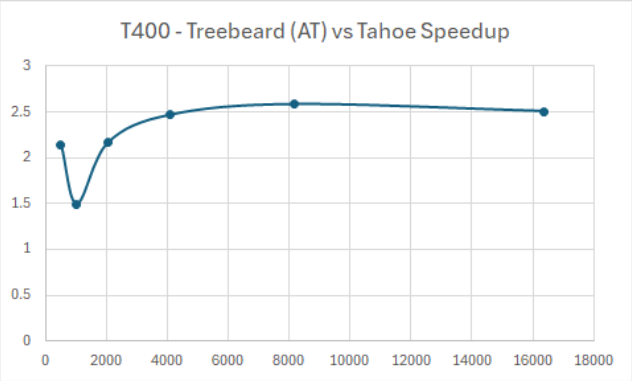


Figure 7. SilvanForge vs Tahoe Kernel Time Speedup on NVIDIA T400.

and the kernel code itself by performing optimizations like tree walk unrolling and interleaving. In contrast, Tahoe uses a library-based approach, and cannot generate code tailored to a model like SilvanForge does.

RAPIDS FIL[**?**] is a library that implements decision tree inference on GPUs and is the most widely used production

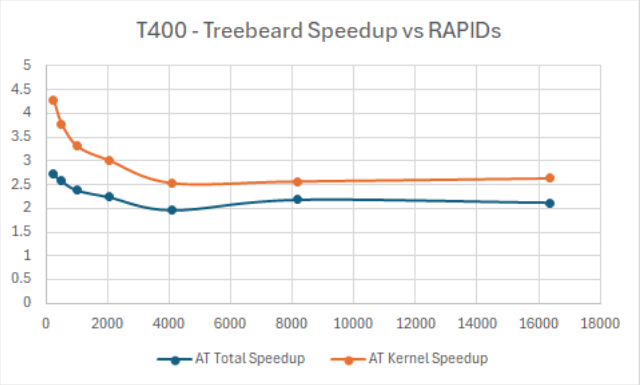


Figure 8. SilvanForge vs RAPIDs Speedup on T400.

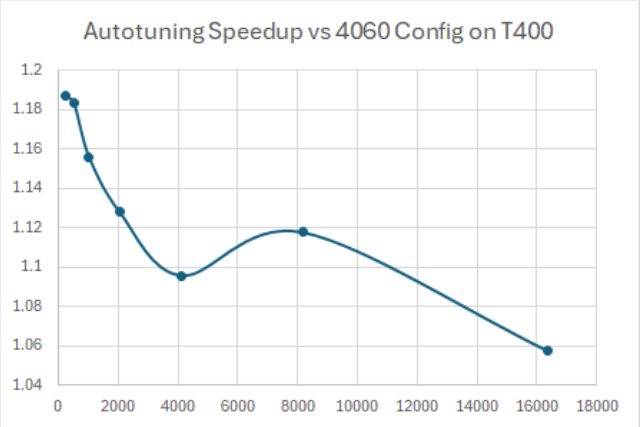


Figure 9. Autotuning heuristics speedup vs best 4060 schedule on T400.

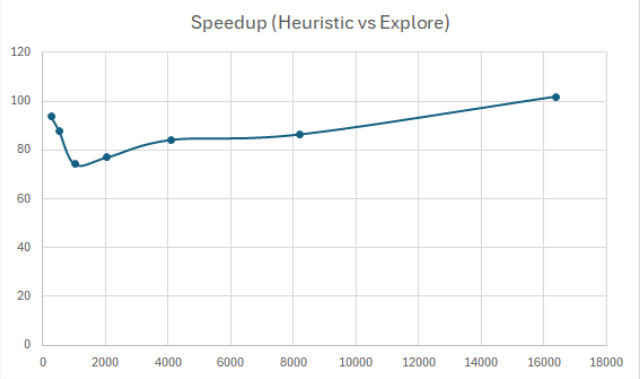


Figure 10. Autotuning heuristic compile time speedup vs full schedule exploration.

system for decision tree inference. While FIL does implement some heuristics to pick a good configuration for every model, these techniques are limited and the library essentially uses a single strategy and in-memory representation for all models. XGBoost [6] also implements GPU support[**?**]. Again,

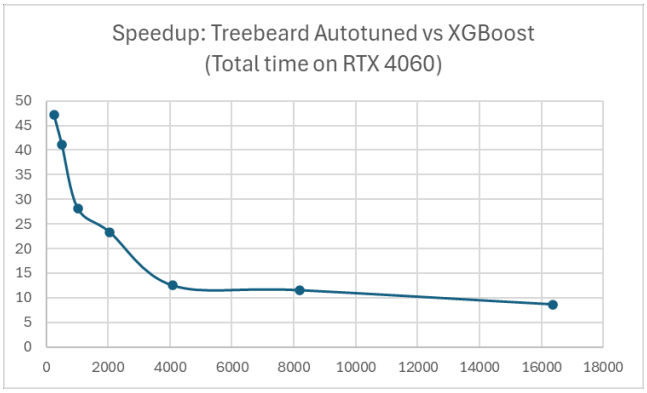


Figure 11. SilvanForge vs XGBoost Speedup on RTX 4060.

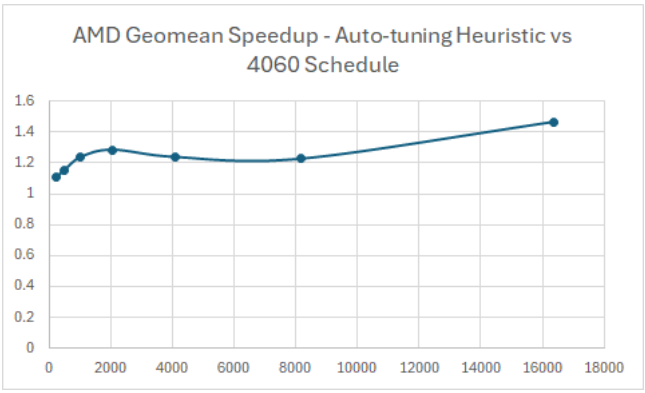
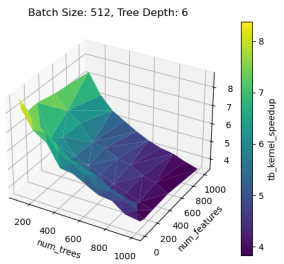


Figure 12. Autotuning heuristics speedup vs best 4060 schedule on MI210.

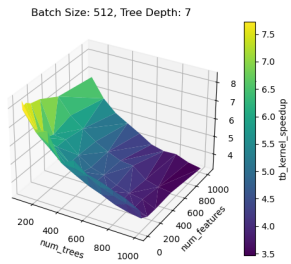
XGBoost uses a single strategy and in-memory representation for all models. In contrast, SilvanForge is a compiler that can explore a much larger optimization space and can generate code that is tailored to a specific model.

Decision Tree Ensemble Compilers: Several compilers for decision tree ensembles have been proposed in the literature [3, 21, 24]. Treebeard and Treelite exclusively target CPUs and all their optimizations are designed purely for performance on CPUs. Treelite[3] is a model compiler that only supports generation of simple `if-else` style code. Its code generator is hard-coded to expand trees in an ensemble into a series of `if-else` statements (one for each node in a tree). Due to this, extending Treelite and reusing it to target GPUs is not possible.

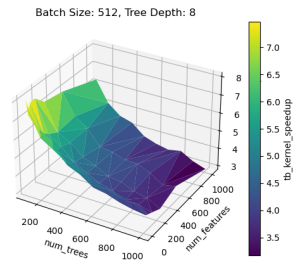
Treebeard is the work most closely related to SilvanForge. While we build on top of Treebeard, SilvanForge is a significant enhancement over Treebeard. Most importantly, we re-architect Treebeard in order to generate code for multiple target processors. Specifically, we introduce the scheduling language and schedule exploration while also enhancing the intermediate representations and support for



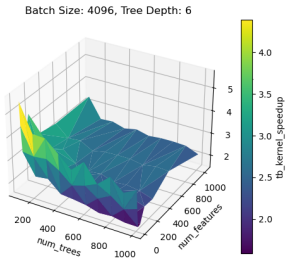
(a) Kernel Speedup for batch size 512, depth 6.



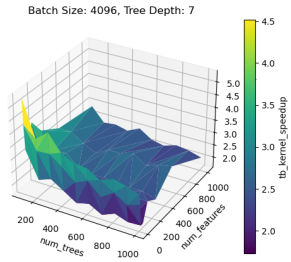
(b) Kernel Speedup for batch size 512, depth 7.



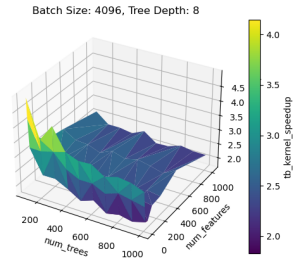
(c) Kernel Speedup for batch size 512, depth 8.



(d) Kernel Speedup for batch size 4096, depth 6.



(e) Kernel Speedup for batch size 4096, depth 7.



(f) Kernel Speedup for batch size 4096, depth 8.

parallelizing across trees through the implementation of a novel MLIR reduction dialect.

Hummingbird[21] is a compiler that compiles traditional ML models to tensor operations thereby enabling them to be run on tensor-based frameworks like TensorFlow. Hummingbird can target both CPUs and GPUs. Its stated aim is to allow traditional ML based applications to easily leverage progress in tensor compilers. However, as was shown earlier [24], tensor operations are not the most efficient way to implement decision tree inference and the performance of code generated by Hummingbird is significantly lower than the code generated by other frameworks.

Libraries: The most popular systems for decision tree-based models are libraries. On CPUs, XGBoost[6], LightGBM[15] and scikit-learn[2] are extremely popular. These libraries implement both training and inference. On GPUs, RAPIDs FIL[**?**] is the most commonly used library. However, as mentioned in Section 1, none of these systems provide portable performance across different target machines. TODO Write about the PACT paper and whether our scheduling language can represent all the schedules they propose. Other systems that hide dependency stalls by interleaving tree walks[4], implement optimized algorithms for tree inference[19, 20] and predict cache performance of decision tree ensembles on CPUs[13, 31] have been proposed in prior work. However, these systems are limited to CPUs. Some systems have been proposed to parallelize decision tree training on CPUs and GPUs[12, 22].

Other Systems and Techniques: Ren et. al. [28] design an intermediate language and a virtual machine to enable vector execution of decision tree inference. In contrast with SilvanForge, the virtual machine that performs the SIMD execution is itself implemented by hand on different target processors. This is clearly more expensive than SilvanForge's approach since the VM needs to be reimplemented for every supported target architecture. Jo et. al.[14] describe code transformations and runtime techniques that help vectorize tree-based applications. However, they do not study optimizations specific to decision trees. Inspector-executor systems [18, 23] have been developed to parallelize tree walks. However, these techniques are not a good fit for decision tree inference. In decision trees, the individual node predicates are very simple and the overhead of an inspector-executor system would be prohibitive.

Code Generation Systems from Other Domains: Several optimizing compilers and code generation techniques have been developed for other domains. TVM[7], Tiramisu[5], and Tensor Comprehensions[33] are optimizing compilers for DNNs that can target a variety of processors. Similarly, Halide[26] is a DSL and compiler primarily designed for image processing applications. The concept of separating the computation from the schedule was pioneered by Halide and has since

been adopted by several other systems [5, 7, 36]. However, to the best of our knowledge, SilvanForge is the first system to design a scheduling language for decision tree inference optimization and to build a system capable of state-of-the-art performance across different processors.

Libraries that compose or generate optimized implementations for BLAS[1, 32, 34] and signal processing[8, 25].

Reductions: CUB[**?**] and Thrust[**?**] are libraries that implement high-performance parallel reductions on GPUs. While they provide highly-tuned implementations to perform large reductions, it is not possible to fuse these functions with other computations as required in SilvanForge. Reddy et. al. [27] describe language constructs in Pencil [**?**] to express reductions and to represent and optimize them using the polyhedral framework. It is not clear how these techniques can be fused with other computations in arbitrary loop nests as required in SilvanForge. Additionally, their system does not express the hierarchical nature of reductions and also only targets GPUs. Suriana et. al. [30] extend Halide to add support for factoring reductions in the Halide scheduling language and to synthesize reduction operators. De Gonzalo et. al. [9] describe a system based on Tangram that composes several partial reduction implementations into different reduction implementations for GPUs and then searches through these alternate implementations to find the best ones. In summary, none of these systems provide abstractions and a general framework to generate and optimize reductions across different target processors as SilvanForge does.

12 Citations and Bibliographies

Artifacts: [17] and [16].

References

[1] [n. d.]. NVIDIA CUTLASS. https://github.com/NVIDIA/cutlass. Accessed: 2022-04-16.

[2] [n. d.]. scikit-learn : Machine Learning in Python. https://scikit-learn.org/stable/. Accessed: 2022-04-16.

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

[4] 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

[5] Riyadh Baghdadi, Jessica Ray, Malek Ben Romdhane, Emanuele Del Sozzo, Abdurrahman Akkas, Yunming Zhang, Patricia Suriana, Shoaib Kamil, and Saman Amarasinghe. 2019. Tiramisu: A Polyhedral Compiler for Expressing Fast and Portable Code. In *Proceedings of the 2019 IEEE/ACM International Symposium on Code Generation and Optimization* (Washington, DC, USA) *(CGO 2019)*. IEEE Press, 193–205.

[6] 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

[7] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. 2018. TVM: An Automated End-to-End Optimizing Compiler for Deep Learning. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. USENIX Association, Carlsbad, CA, 578–594. https://www.usenix.org/conference/osdi18/presentation/chen

[8] Matteo Frigo. 1999. A Fast Fourier Transform Compiler. In *Proceedings of the ACM SIGPLAN 1999 Conference on Programming Language Design and Implementation* (Atlanta, Georgia, USA) *(PLDI '99)*. Association for Computing Machinery, New York, NY, USA, 169–180. https://doi.org/10.1145/301618.301661

[9] Simon Garcia De Gonzalo, Sitao Huang, Juan Gómez-Luna, Simon Hammond, Onur Mutlu, and Wen-mei Hwu. 2019. Automatic Generation of Warp-Level Primitives and Atomic Instructions for Fast and Portable Parallel Reduction on GPUs. In *2019 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. 73–84. https://doi.org/10.1109/CGO.2019.8661187

[10] Léo Grinsztajn, Edouard Oyallon, and Gaël Varoquaux. 2022. Why do tree-based models still outperform deep learning on tabular data? arXiv:2207.08815 [cs.LG]

[11] John L. Hennessy and David A. Patterson. 2019. A new golden age for computer architecture. *Commun. ACM* 62, 2 (jan 2019), 48–60. https://doi.org/10.1145/3282307

[12] Karl Jansson, Håkan Sundell, and Henrik Boström. 2014. gpuRF and gpuERT: Efficient and Scalable GPU Algorithms for Decision Tree Ensembles. *2014 IEEE International Parallel & Distributed Processing Symposium Workshops* (2014), 1612–1621.

[13] Xin Jin, Tao Yang, and Xun Tang. 2016. A Comparison of Cache Blocking Methods for Fast Execution of Ensemble-Based Score Computation. In *Proceedings of the 39th International ACM SIGIR Conference on Research and Development in Information Retrieval* (Pisa, Italy) *(SIGIR '16)*. Association for Computing Machinery, New York, NY, USA, 629–638. https://doi.org/10.1145/2911451.2911520

[14] Youngjoon Jo, Michael Goldfarb, and Milind Kulkarni. 2013. Automatic Vectorization of Tree Traversals. In *Proceedings of the 22nd International Conference on Parallel Architectures and Compilation Techniques* (Edinburgh, Scotland, UK) *(PACT '13)*. IEEE Press, 363–374.

[15] Guolin Ke, Qi Meng, Thomas Finley, Taifeng Wang, Wei Chen, Weidong Ma, Qiwei Ye, and Tie-Yan Liu. 2017. LightGBM: A Highly Efficient Gradient Boosting Decision Tree. In *Proceedings of the 31st International Conference on Neural Information Processing Systems* (Long Beach, California, USA) *(NIPS'17)*. Curran Associates Inc., Red Hook, NY, USA, 3149–3157.

[16] 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

[17] 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

[18] Jianqiao Liu, Nikhil Hegde, and Milind Kulkarni. 2016. Hybrid CPU-GPU Scheduling and Execution of Tree Traversals. In *Proceedings of the 2016 International Conference on Supercomputing* (Istanbul, Turkey) *(ICS '16)*. Association for Computing Machinery, New York, NY, USA, Article 2, 12 pages. https://doi.org/10.1145/2925426.2926261

[19] 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

[20] Claudio Lucchese, Franco Maria Nardini, Salvatore Orlando, Raffaele Perego, Nicola Tonellotto, and Rossano Venturini. 2016. Exploiting CPU SIMD Extensions to Speed-up Document Scoring with Tree Ensembles. In *Proceedings of the 39th International ACM SIGIR Conference on Research and Development in Information Retrieval* (Pisa, Italy) *(SIGIR '16)*. Association for Computing Machinery, New York, NY, USA, 833–836. https://doi.org/10.1145/2911451.2914758

[21] 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

[22] Aziz Nasridinov, Yangsun Lee, and Young-Ho Park. 2013. Decision tree construction on GPU: ubiquitous parallel computing approach. *Computing* 96 (2013), 403–413.

[23] Keshav Pingali, Donald Nguyen, Milind Kulkarni, Martin Burtscher, M. Amber Hassaan, Rashid Kaleem, Tsung-Hsien Lee, Andrew Lenharth, Roman Manevich, Mario Méndez-Lojo, Dimitrios Prountzos, and Xin Sui. 2011. The Tao of Parallelism in Algorithms. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation* (San Jose, California, USA) *(PLDI '11)*. Association for Computing Machinery, New York, NY, USA, 12–25. https://doi.org/10.1145/1993498.1993501

[24] Ashwin Prasad, Sampath Rajendra, Kaushik Rajan, R Govindarajan, and Uday Bondhugula. 2022. Treebeard: An Optimizing Compiler for Decision Tree Based ML Inference. In *2022 55th IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 494–511. https://doi.org/10.1109/MICRO56248.2022.00043

[25] M. Puschel, J.M.F. Moura, J.R. Johnson, D. Padua, M.M. Veloso, B.W. Singer, Jianxin Xiong, F. Franchetti, A. Gacic, Y. Voronenko, K. Chen, R.W. Johnson, and N. Rizzolo. 2005. SPIRAL: Code Generation for DSP Transforms. *Proc. IEEE* 93, 2 (2005), 232–275. https://doi.org/10.1109/JPROC.2004.840306

[26] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. 2013. Halide: A Language and Compiler for Optimizing Parallelism, Locality, and Recomputation in Image Processing Pipelines. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Seattle, Washington, USA) *(PLDI '13)*. Association for Computing Machinery, New York, NY, USA, 519–530. https://doi.org/10.1145/2491956.2462176

[27] Chandan Reddy, Michael Kruse, and Albert Cohen. 2016. Reduction Drawing: Language Constructs and Polyhedral Compilation for Reductions on GPU. In *Proceedings of the 2016 International Conference on Parallel Architectures and Compilation* (Haifa, Israel) *(PACT '16)*. Association for Computing Machinery, New York, NY, USA, 87–97. https://doi.org/10.1145/2967938.2967950

[28] Bin Ren, Todd Mytkowicz, and Gagan Agrawal. 2014. A Portable Optimization Engine for Accelerating Irregular Data-Traversal Applications on SIMD Architectures. *ACM Trans. Archit. Code Optim.* 11, 2, Article 16 (jun 2014), 31 pages. https://doi.org/10.1145/2632215

[29] Ravid Shwartz-Ziv and Amitai Armon. 2022. Tabular data: Deep learning is not all you need. *Inf. Fusion* 81, C (may 2022), 84–90. https://doi.org/10.1016/j.inffus.2021.11.011

[30] Patricia Suriana, Andrew Adams, and Shoaib Kamil. 2017. Parallel associative reductions in halide. In *Proceedings of the 2017 International Symposium on Code Generation and Optimization* (Austin, USA) *(CGO '17)*. IEEE Press, 281–291.

[31] Xun Tang, Xin Jin, and Tao Yang. 2014. Cache-Conscious Runtime Optimization for Ranking Ensembles. In *Proceedings of the 37th International ACM SIGIR Conference on Research amp; Development in Information Retrieval* (Gold Coast, Queensland, Australia) *(SIGIR '14)*. Association for Computing Machinery, New York, NY, USA, 1123–1126. https://doi.org/10.1145/2600428.2609525

[32] Field G. Van Zee and Robert A. van de Geijn. 2015. BLIS: A Framework for Rapidly Instantiating BLAS Functionality. *ACM Trans. Math. Softw.* 41, 3, Article 14 (jun 2015), 33 pages. https://doi.org/10.1145/2764454

[33] Nicolas Vasilache, Oleksandr Zinenko, Theodoros Theodoridis, Priya Goyal, Zachary DeVito, William S. Moses, Sven Verdoolaege, Andrew Adams, and Albert Cohen. 2018. Tensor Comprehensions: Framework-Agnostic High-Performance Machine Learning Abstractions. https://doi.org/10.48550/ARXIV.1802.04730

[34] R. Clint Whaley and Jack Dongarra. 1998. Automatically Tuned Linear Algebra Software. In *SuperComputing 1998: High Performance Networking and Computing*.

[35] 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

[36] Yunming Zhang, Mengjiao Yang, Riyadh Baghdadi, Shoaib Kamil, Julian Shun, and Saman Amarasinghe. 2018. GraphIt: a high-performance graph DSL. *Proc. ACM Program. Lang.* 2, OOPSLA, Article 121 (oct 2018), 30 pages. https://doi.org/10.1145/3276491