SilvanForge : A Schedule Guided Retargetable Compiler for Decision Tree Inference

Abstract

This paper is motivated by the growing demand for the increased performance of machine learning applications on different hardware platforms including CPUs and GPUs. We focus on accelerating the inference of decision tree based models, which are the most popular models for tabular data. Existing solutions do not achieve the highest possible performance because they do not explore different optimization configurations. And since these systems are hand-written, they are not portable either.

We address these problems by designing SilvanForge, a *schedule-guided*, *retargetable* compiler infrastructure for decision tree based models. SilvanForge has two core components. The first is 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. The second is an optimizing retargetable multi-level compiler that can generate code for any specified schedule. SilvanForge's retargetability is based not only on being able to generate code for different target architectures (CPU vs. GPU), but also on its ability to use different data layouts, caching strategies, parallel reduction schemes etc. To accomplish this level of configurability, we re-architect and significantly extend the open-source Treebeard CPU compiler to support (i) schedule-guided compilation, (ii) retargetable GPU code generation, and (iii) GPU-specific optimizations.

We demonstrate that SilvanForge can generate high-performance inference code, for several hundred decision tree models across different batch sizes and target architectures. Our scheduling heuristic is able to quickly find near-optimal schedules TODO [how do we argue that the schedule is near-optimal? Do we have some exptl. evidence that we can show in the results section?] schedule while searching over a small number (~50) 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, 16 pages. https://doi.org/XXXXXXX.XXXXXXX

1 Introduction

We are in the midst of a hardware revolution, a new golden age for computer architecture [21]. 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 training and inference to models with billions of parameters across 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 [20, 42]. Their robustness, interpretability, and ability to handle missing data make them a popular choice for a wide range of applications[12, 17, 26, 27, 34, 43]. A recent survey [1] found that about three quarters of data scientists use decision tree based models. An analysis of ML workloads at a large scale web company found that these models are most widely used [37]. Recent work has noted that the cost of inference is the most critical factor in the overall cost of deploying a machine learning model [6, 32]. This is because, in production settings, each model is trained once and often used for inference millions of times. Further, 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 commodity platforms with CPUs and GPUs.

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 and typically only target a single platform. XGBoost [15] 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 [5] uses a reorg representation and partitions trees across a fixed number of threads. Tahoe [49] uses a variation of the reorg representation and has four predefined inference strategies from which it picks one based on an analytical model. All these systems are CUDA based and only work on NVIDIA based GPUs. Treebeard, the state-of-the-art decision tree model compiler for CPUs built using the MLIR infrastructure [28], 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. TODO Shouldn't we reverse this? First say no GPU support and then the scaling issue.

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. This is achieved by a compilation framework consisting of a custom scheduling language that can represent a wide range of implementation strategies and techniques to efficiently explore the optimization space. We demonstrate that the language is sufficient to express the various optimizations proposed by prior work and that our schedule exploration heuristic can quickly find a near optimal schedule for the model being compiled. TODO RG: it would be good to expand on this and also talk about the retargetable component. We could also say that the schedule framework and the retargetable compiler work in an interwined manner, each benefitting from the other. The second component of SilvanForge is a *retargetable* multi-level compiler that can generate efficient code for any specified schedule for both CPUs and GPUs. For this purpose, we re-architect Treebeard to support schedule-guided code generation, and incorporate several new optimizations. These two components of the proposed SilvanForge compiler infrastructure are interwined, each benefitting from the other. TODO Performance evaluation summary

1.1 Contributions

- We present the design of a multi-target compiler infrastructure for decision tree inferencing and implement several optimizations within this framework. We are also the first to implement an optimizing compiler for decision tree inference on GPUs. TODO AP: Given that there is Hummingbird, can we really say this?
- 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, such as different data layouts, caching strategies, parallel reduction schemes, that work across CPUs and GPUs.
- 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

In this section, we first motivate the need for a scheduling language by showing how a model can be compiled in different ways and subsequently, we show how drastically performance can vary across these variants for real benchmarks.

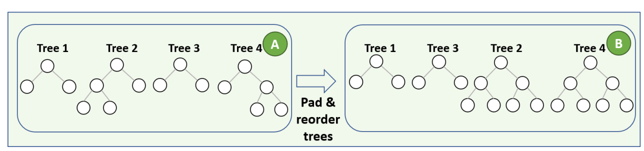


Figure 1. The representation of a model in high-level IR and the model after trees are padded and reordered.

As an example, consider a model with four trees, two complete trees of depth 1 and two of depth 2 (Figure 1). We first describe a simple strategy that processes one tree at a time for all input rows and unrolls all tree walks. The loop over the trees is split into two loops – one that iterates over the first two trees (Trees 1 and 2 with depth 1) and the second that iterates over the last two trees (Trees 3 and 4 with depth 2). The SilvanForge schedule then unrolls the tree walks for each tree.

```
reorder(tree, batch)
// Fiss tree loop so trees with equal depth
// are processed together
split(tree, t_depth1, t_depth2, 2)
// Unroll the tree walks
```

```
unrollWalk(t_depth1, 1)
unrollWalk(t_depth2, 2)
```

The concrete implementation of this schedule (in one of SilvanForge's IRs) is as follows.

```
model = ensemble(...)
for t_depth1 = 0 to 2 step 1 {
  T = getTree(ensemble, t_depth1)
  for batch = 0 to BATCH_SIZE step 1 {
    treePred = walkDecisionTree(T,
                    input[batch]) <unrollDepth = 1>
    reduce(result[batch], treePred)
  }
}
for t_depth2 = 2 to 4 step 1 {
  T = getTree(ensemble, t_depth2)
  for batch = 0 to BATCH_SIZE step 1 {
    treePred = walkDecisionTree(T,
                    input[batch]) <unrollDepth = 2>
    reduce(result[batch], treePred) <'+', 0.0>
  }
}
```

This schedule is ideally suited for a single-core CPU. It maximizes the reuse of trees in the L1 cache and also minimizes the amount of branching by unrolling tree walks. However, it doesn't exploit any parallelism and is therefore ill-suited for parallel processors.

One form of parallelism that can be exploited is to process rows in parallel. However, with massively parallel processors like GPUs, this may not yield sufficient parallel work. Another option is to also parallelize across trees. A possible strategy to accomplish this is encoded in the following schedule.

```
// Split the trees into two sets
tile(tree, t0, t1, 2)
reorder(batch, t1, t0)
// Fiss loop so that trees with equal
// depth are processed together
split(t0, t0_depth1, t0_depth2, 2)
unrollWalk(t0_depth1, 1)
unrollWalk(t0_depth2, 2)
// Configure the GPU kernel dimensions
gpuDimension(batch, grid.x)
gpuDimension(t1, block.x)
```

This schedule generates an inference function that runs on the GPU. The inference routine 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 IR 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_depth1 = 0 to 2 step 2 {
      T = getTree(ensemble, t0_depth1 + t1)
      treePred = walkDecisionTree(T,
                      input[batch]) <unrollDepth = 1>
      reduce(result[batch], treePred)
    }
    for t0_depth2 = 2 to 4 step 2 {
      T = getTree(ensemble, t0_depth2 + t1)
      treePred = walkDecisionTree(T,
                      input[batch]) <unrollDepth = 2>
      reduce(result[batch], treePred) <'+', 0.0>
    }
  }
}
```

In the case of this schedule, the `reduce` operation needs special consideration. In order to correctly generate code for this schedule, the compiler needs to determine that parallel iterations of the `t1` loop accumulate into the same element of the `result` array. One possible solution is to rewrite the reduction so that each parallel iteration accumulates into a different array element by introducing a temporary buffer (`temp`) as follows.

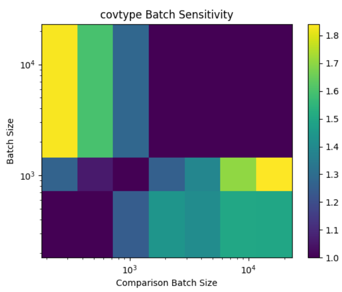
```
float temp[2][BATCH_SIZE]
model = ensemble(...)
par.for batch = 0 to BATCH_SIZE step 1 <grid.x> {
  par.for t1 = 0 to 2 step 1 <block.x> {
    for t0_depth1 = 0 to 2 step 2 {
      T = getTree(ensemble, t0_depth1 + t1)
      treePred = walkDecisionTree(T,
                      input[batch]) <unrollDepth = 1>
      reduce(temp[t1][batch], treePred)
    }
    for t0_depth2 = 2 to 4 step 2 {
      T = getTree(ensemble, t0_depth2 + t1)
      treePred = walkDecisionTree(T,
                      input[batch]) <unrollDepth = 2>
      reduce(temp[t1][batch], treePred) <'+', 0.0>
    }
  }
  result[batch] = reduce_dimension(temp[:][batch], 0)
}
```

Here, partial results are accumulated into `temp` and then reduced across the `t1` dimension to get the final result.

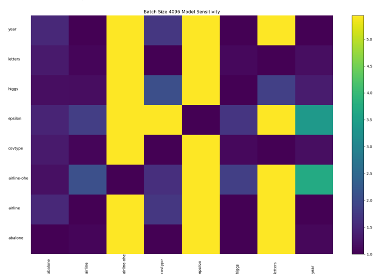
As is evident from these examples, it is possible to optimize the inference routine in different ways. Also, the structure of the loop nest in the inference routine can get quite complex even for simple schedules. Writing these routines by hand is error-prone and time-consuming. We believe that designing a scheduling language to encapsulate these strategies and a principled code generator to automatically generate code based on the schedule is the best approach.

To further complicate matters, we find that these different strategies can have significantly different performance. Figures 2a and 2b show the variation in performance when the same schedules are used across batch sizes and models respectively. These diagrams show that the best schedules to use vary across batch sizes and models. The largest slowdown is 5× when schedules are used across different batch sizes and 6× when schedules are used across different models. Therefore, no one strategy can be used across all models and batch sizes. A system that is capable of specializing generated code for both batch size and model is required for the best performance.

Building such a configurable compiler and supporting code generation for CPUs and GPUs required us to solve several fundamental problems. We had to enable the compiler to represent and optimize reductions, deal uniformly with



(a) Batch sensitivity for `covtype`. Each point shows the slowdown when best schedule for the x-axis batch size is used for the y-axis batch size.



(b) Model sensitivity for batch size 4096. Each point shows the slowdown when best schedule for the x-axis model is used for the y-axis model.

different in-memory representations of the model, design optimizations to effectively use the memory hierarchy of the target processor (shared memory on GPUs and cache on the CPU) and finally be able to generate target specific code. The rest of the paper describes these challenges in detail and how we solved them in SilvanForge.

3 Scheduling Language

As established in Section 2, there are several different configurations and optimizations strategies for decision tree inference. The best ones vary significantly across models, batch sizes and hardware platforms. Therefore, designing any one hard-coded strategy is not feasible as this would make portable performance impossible. To address this, 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. The specified schedule controls the lowering of model inference to a set of loop nests. This configurability provided by the schedule allows us to build auto-schedulers and auto-tuners (Section 8).

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 1). The second is a set of directives that enable optimizations on a loop (Table 2). Finally, we have a class of attributes that enable reduction specific optimizations (Table 3).

The set of loops (index variables) are internally represented as nodes in a tree where the children of a node represent immediately contained loops. Each schedule primitive modifies this tree in some way. The compiler tracks the lineage of each of the loops. This allows the compiler to automatically infer the ranges for all loops.

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.
`gpuDimension`	**indexVar** **gpuDim**	Map the passed index variable to a dimension of either the grid or thread block.

Table 1. List of all the loop modifiers in SilvanForge's scheduling language. We use *index variable* and *loop* interchangeably in descriptions for clarity of exposition.

SilvanForge's scheduling language is expressive enough to represent a wide range of strategies used in existing systems. We show examples of how it can be used to represent XGBoost and Tahoe's strategies. Before presenting the examples, we note that SilvanForge's default loop order is [`batch`, `tree`], i.e, for each row in the input batch, go over all trees.

XGBoost[15] implements inference on the CPU by going over a fixed number of rows (64 in the previous version) for

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

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. Different sets of rows are processed in parallel. The following schedule expresses XGBoost's strategy.

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

Tahoe[49] 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.

- In the *direct method*, 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.
- In the *shared data* strategy, a thread block walks all the trees for a given row in parallel. Then, 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)
```

4 SilvanForge Multi-Level Compiler

SilvanForge takes a serialized decision tree ensemble as input (XGBoost JSON, ONNX etc.) and automatically generates an optimized inference function that can either target CPUs or GPUs. Figure 3 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. SilvanForge is built using the open-source Treebeard infrastructure [36]. Since the Treebeard infrastructure was originally designed to target CPUs, significant extensions (shown as dark boxes in Figure 3) were required to support schedule guided compilation for CPUs and GPUs.

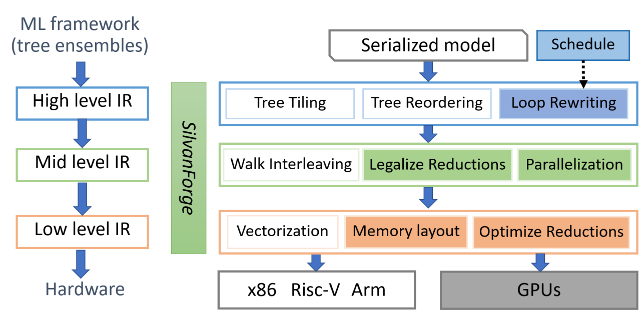


Figure 3. SilvanForge compiler structure.

Table 4 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 trees. We extend the Treebeard infrastructure with loop rewrites on the HIR that are implemented through the scheduling language (Section 3). We use these to implement the automatic scheduling described in Section 8. Additionally, the Treebeard infrastructure implements HIR transformations to reorder and pad trees.

The HIR is lowered to MIR as dictated by the *schedule*. In the lowered MIR, the compiler uses the `reduce` op from the `reduction` dialect we design (details in Section 5) to represent reduction operations. The lowering of the `reduce` operation involves introducing temporary buffers and splitting the operation to correctly implement reduction in the presence of parallel loops. This process, that we call ***legalization***, is described in Section 5.

Treebeard 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. The fact that exploting ILP improved performance on GPUs was surprising.

The MIR is then further lowered to a low-level IR (LIR). 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 6). Also, 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. Subsequently, the LIR is lowered to LLVM and then JIT'ed to the specified target processor.

5 Reductions : Representation, Optimization and Lowering

SilvanForge needs to sum up individual tree predictions to compute the prediction of the model while performing inference. However, generating fused reductions within arbitrary loop nests specified using SilvanForge's scheduling language is non-trivial. We found that existing reduction support in MLIR is insufficient to code generate and optimize these reductions. MLIR only supports reductions of value types and does not provide ways to lower reductions to GPUs. To address this gap, we design an MLIR dialect that allows us to specify accumulating values into an element of a multi-dimensional array and can be lowered to CPU or GPU.

The main abstraction we introduce is the `reduce` op. It models atomically accumulating values into an element of a multi-dimensional array (represented by an MLIR `memref`). The following example shows how the `reduce` op can be used to sum up the elements of an array in parallel.

```
float arr[10], result[1]
par.for i0 = 0  to 10 step 5 {
  for i1 = 0 to 5
    reduce(result[0], arr[i0 + i1]) <"+", 0.0>
}
```

The semantics of the `reduce` op is exactly the semantics of an atomic accumulation, i.e. it guarantees that all accumulations are correctly performed even in the presence of parallel loops. 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.

Having modeled the reductions with an abstract operation, the aim now is to lower this to a correct and optimized implementation on both CPU and GPU. In order to do this, we first determine if any parallel loop iterations can accumulate into the same array element. We call such loops ***reduction loops***. If such loops exist, we ***privatize*** the array for each iteration of the loop. We call this process ***legalization***. Subsequently, each privatized dimension can be reduced at the end of the reduction loop it was inserted for. TODO We cannot do better than this in terms of memory usage TODO Need a proof.

In our example above, parallel iterations of the `i0` loop accumulate into the same element of the `result` array. We would therefore privatize the `result` array for each iteration of the `i0` loop as follows.

```
float arr[10], result[1]
float resultPriv[2][1]
par.for i0 = 0  to 10 step 5 {
  for i1 = 0 to 5
    reduce(resultPriv[i0/5][0], arr[i0 + i1]) <"+",
   0.0>
}
result = reduce_dimension(resultPriv, 0)
```

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

To reduce the amount of memory used by arrays introduced for reduction, we introduce the `reduce_dimension_inplace` operation. It is similar to the `reduce_dimension` op except that it updates the input array inplace rather than writing results to a target array. It writes results to the zeroth index of the dimension being reduced.

5.1 Lowering Reduction Operations

We implement lowering of the operations defined above to both the CPU and GPU. Since the lowering pipeline from MIR to LIR are different for CPU and GPU compilation, we implement lowering and optimization of our reduction dialect to CPUs and GPUs simply using different MLIR rewrite patterns. In this section, we briefly describe how these operations are lowered to the CPU and GPU.

5.1.1 Lowering to CPU. The lowering of the reduction operations to CPU is fairly straightforward. We lower `reduce_dimension_in`

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 3. **predicate** specifies the operator to use to evaluate nodes (Eg: <, ≤).
`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 4. 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.

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.

5.1.2 Lowering to GPU.

The same abstractions can be lowered to efficient GPU implementations and therefore, simplify higher-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 can either exploit parallelism across the independent reductions or the inherent parallelism in the reduction by performing a divide and conquer reduction. If there are enough independent reductions to keep all threads in a thread block busy, then the lowering pass can generate code that performs one (or multiple) reductions in each thread. If, however, there are not enough independent reductions, then the lowering pass generates a tree style reduction where multiple threads co-operate 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. The compiler only allocates as much shared memory as needed to hold values processed by a single thread-block. 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.

5.2 Use in SilvanForge

We now show how SilvanForge uses the reduction dialect to generate code for decision tree inference using an example. In our example, `N_t` is the number of trees and `batch_size` is the batch size. The schedule tiles both the batch and tree loops and parallelizes the outer batch and tree loops. The schedule with which code is generated is as follows.

```
tile(batch, i0, i1, batch_size/2);
tile(tree, t0, t1, N_t/2);
reorder({i0, t0, t1, i1});
parallel(t0);
parallel(i0);
```

The MIR generated by SilvanForge for the above schedule is as follows.

```
float result[batch_size]
model = ensemble(...)
par.for i0 = 0 to batch_size step batch_size/2 {
  par.for t0 = 0 to N_t step N_t/2 {
    for t1 = 0 to N_t/2 {
      for i1 = 0 to batch_size/2 {
        t = getTree(model, t0 + t1)
        p = walkDecisionTree(t, rows[i0+i1])
        reduce(result[i0+i1], p)
      }
    }
  }
}
```

SilvanForge determines that the `t0` loop is a reduction loop w.r.t the `result` array and therefore legalizes the reduction by inserting a privatized array `partResults`. The privatized dimension of this array is reduced at the end of the `t0` loop.

```
float result[batch_size], partResults[2][batch_size]
model = ensemble(...)
par.for i0 = 0 to batch_size step batch_size/2 {
  par.for t0 = 0 to N_t step N_t/2 {
    for t1 = 0 to N_t/2 {
      for i1 = 0 to batch_size/2 {
        t = getTree(model, t0 + t1)
        p = walkDecisionTree(t, rows[i0+i1])
        reduce(result[i0+i1], p)
      }
    }
  }
  results[i0:i0+batch_size/2] = reduce_dimension(
    partResults[:, i0:i0+batch_size/2], 0)
}
```

While legalizing the reduction, the compiler determines that the `reduce_dimension` operation can only compute a subset of the final result (the subset that is computed within the current parallel iteration of the `i0` loop).

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.

6 Model 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 4. The array and sparse representations are the ones described in the Treebeard paper[36]. The reorg representation is the representation used by the RAPIDs library[5]. 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 memory is allocated only for nodes present in the tree 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. This representation was designed to improve memory coalescing when tree nodes are being loaded.

One of the major changes we make to the original design of Treebeard [36] 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 define an interface that representations implement. The code generator is implemented using this interface thus hiding details of the actual representation from the core compiler. Curcially, the interface abstracts how and what buffers are allocated, how to move from a node to its child, how trees are cached, reading the value of leaves and now threshold and feature indices are read from the allocated buffers.

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

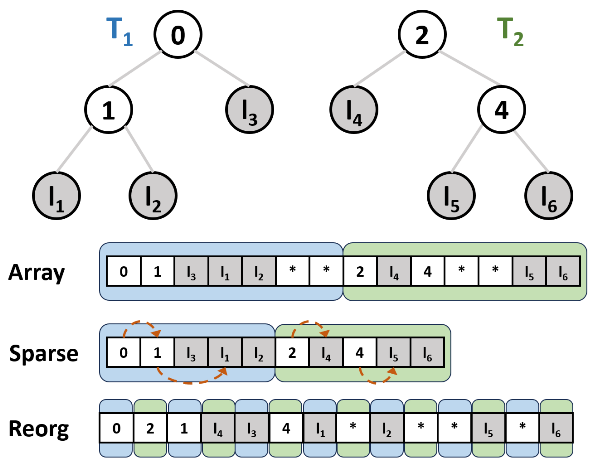


Figure 4. The three representations supported by SilvanForge.

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.

7 Caching

SilvanForge provides mechanisms to cache both trees and input rows on both the CPU and GPU. As described in Section 3, the user can specify that the working set of an iteration of a loop 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.

Caching is encoded in the mid-level IR using the `cacheTrees` and `cacheRows` operations (Table 4). 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. SilvanForge also determines the working set of the loop and generates a caching operation with the appropriate limits.

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. On CPU, the cahce operations are lowered to preteches while on the GPU they are lowered to reads into shared memory.

Lowering the `cacheRows` operation is straightforward because the input is currently assumed to be a dense array format. The lowering for the `cacheRows` operation is implemented directly in the SilvanForge compiler.

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.

8 Exploring the Schedule Space

The set of schedules that can be constructed using the scheduling language described in Section 3 is unbounded. Searching this schedule space to find a high-performance schedule is a non-trivial task. To simplify this process, we design a template schedule for GPUs that encompasses several strategies published in prior work. Our template schedule assigns a configurable number of rows to each thread block and to each thread. It distributes the trees across a specified number of threads and can cache trees and input rows if required while unrolling and interleaving of tree walks. Table 5 lists the parameter values we tried.

Parameter	Values
Rows per thread block	$\{8, 32, 64\}$
Rows per thread	$\{1, 2, 4\}$
Number of tree threads	$\{2, 10, 20, 50\}$
Cache rows	$\{True, False\}$
Cache trees	$\{True, False\}$
Unroll walks	$\{True, False\}$
Tree walk interleave factor	$\{1, 2, 4\}$
Shared memory reduction	$\{True, False\}$

Table 5. List of parameter values we explored for the template GPU schedule.

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 compiler pass that implements the template schedule is also implemented as a module outside the core SilvanForge compiler.

While the template schedule simplifies code generation, finding a good set of parameter values is still hard. Figure 5 shows the distribution of normalized execution times for all benchmark models with different parameter values for the template schedule (inference times normalized w.r.t fastest time for that model). There is a significant amount of variation in performance even within the variants of the template schedule,. 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 5 for our benchmarks and found that it took anywhere between thirty minutes up to a few hours to explore the entire space for each model. We therefore design a heuristic to narrow down the set of schedules to explore based on the following observations.

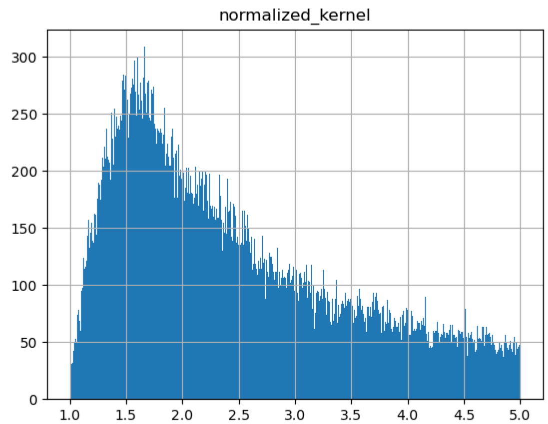


Figure 5. Distribution of normalized execution times for all benchmark models with the template schedule using parameter values as shown in Table 5

.

- 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 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 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. 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 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 9.

Algorithm 1 Heuristic to find a good schedule

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

$bestSchedules \leftarrow shMemSchedules \leftarrow \emptyset$
$rowsPerTB, treeThds \leftarrow TBConfigs(N_{batch}, N_f)$
$cacheRows \leftarrow$ True, $cacheTrees \leftarrow$ False
$interleave \leftarrow \{1, 2, 4\}$
$schedules \leftarrow (rowsPerTB, treeThds, cacheRows,$
            $cacheTrees, interleave)$
for $(sched, rep) \in schedules \times \{array, sparse, reorg\}$ **do**
    $time \leftarrow EvaluateSchedule(sched, rep)$
    $bestSchedules.insert(time, sched, rep)$
end for

for $sched, rep \in Top3(bestSchedules)$ **do**
    $EnableSharedReduction(sched)$
    $time \leftarrow EvaluateSchedule(sched, rep)$
    $shMemSchedules.insert(time, sched, rep)$
end for
return $min(shMemSchedules \cup bestSchedules)$

9 Experimental Evaluation

We evaluate SilvanForge on several machines. The first machine has an AMD Ryzen 9 7950X 16-Core processor, 128 GB of RAM, an NVIDIA RTX 4060 GPU with 8 GB of RAM and runs Ubuntu 22.04.2 LTS with CUDA 11.5. The second has an Intel Core i9-11900K (Rocket Lake) processor with 8 physical cores, 128 GB of RAM, an NVIDIA T400 with 2GB of RAM and runs Ubuntu 20.04.3 LTS with CUDA 11.8. We also evaluate SilvanForge on an AMD MI210 GPU. To establish the efficacy of SilvanForge, we compare it with NVIDIA RAPIDs v23.10, Tahoe, and XGBoost v1.7.6. We use the same benchmark models as in the Treebeard paper[36]. We also evaluate SilvanForge on a set of randomly generated models.

9.1 Performance on Random Models

- Generated a set of over 700 random models with varying depths (6, 7, 8), number of trees (100 - 1000 in steps of 100), and number of features (powers of 2 between 8 to 1024 inclusive).
- Compared the kernel time speedup of SilvanForge vs RAPIDs on the RTX 4060. The schedule used was the one picked by the auto-tuner.
- Speedups range between 1.5× and 8×. SilvanForge outperforms RAPIDs on all models and batch sizes tested.
- Figure 6 shows the speedup for models of depth 8 and batch size 512 and depth 6 and batch size 4096. Trends for other batch sizes and depths are similar.

9.2 Comparison with RAPIDs, Tahoe and XGBoost

- Measured both the kernel time and total time (time including transferring data to the GPU and results back) for RAPIDs and SilvanForge.
- Tahoe only allows us to measure the kernel time since it is written as an executable that performs inference repeatedly on the same data that is transferred to the GPU once.
- Tahoe does not support multiclass models. We just ran the multiclass models (`covtype` and `letters`) as regression models for the comparison. Tahoe also gives wrong results (as reported by its own tests) for `letters` and `year`. In these cases, we pick the time of the fastest variant that gives the correct results.
- Compared the kernel time and total time speedup of SilvanForge vs RAPIDs on the RTX 4060 and T400.
- Compared the total time speedup of SilvanForge vs XGBoost on the RTX 4060.
- The schedule used was the one picked by the auto-tuner.
- SilvanForge outperforms RAPIDs at all batch sizes as shown in Figure 7.
- SilvanForge outperforms Tahoe on all models and batch sizes tested. Individual benchmark speedups range between 1.1× and 16×.
- SilvanForge is faster than XGBoost by more than an order of magnitude. Results are not shown because the speedups don't fit on the same graph.
- Figure 8 shows that SilvanForge offers substantial speedup over RAPIDs even when data needs to be transferred to the GPU and results need to be transferred back.
- Figure 10 shows that these speedups are also observed on the T400 thus showing that SilvanForge offers portable performance across different GPUs.
- Figure 9 shows that SilvanForge outperforms RAPIDs and Tahoe on individual benchmarks on the RTX 4060 at small (1024) and large (8192) batch sizes. These batch sizes require different schedules, but the SilvanForge auto-tuning heuristic is able to find them.
- Figure 11 compares the total time of SilvanForge vs RAPIDs on individual benchmarks. It shows that even with the overhead of data transfers, SilvanForge offers significant speedup. The graph in Figure 11 shows the breakup of the total time into kernel time and data transfer time.

9.3 Autotuning Heuristic

9.4 AMD GPU

9.5 CPU Improvements

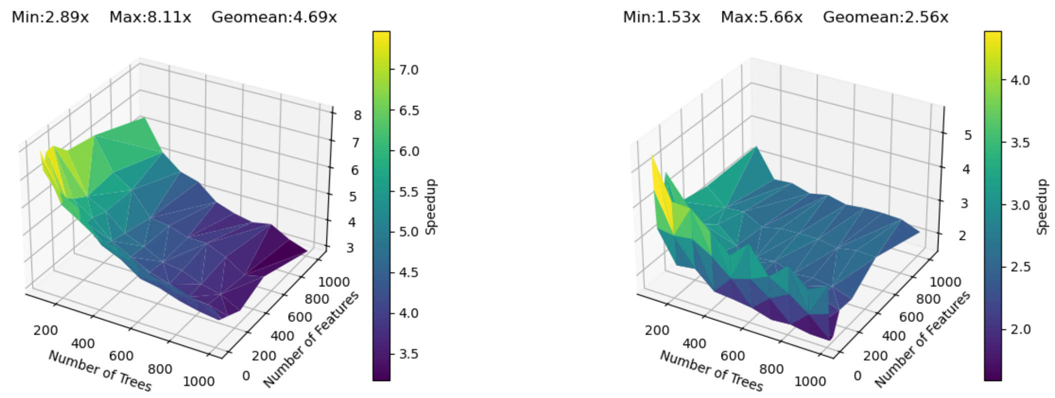
10 Related Work

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

Decision Tree Inference Systems: Tahoe[49] 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) and the kernel code itself by performing optimizations like tree walk unrolling and interleaving.

RAPIDS FIL[5] is a library that implements decision tree inference on GPUs and is the most widely used production 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 [15] also implements GPU support[9] but uses a single strategy and in-memory representation.

On CPUs, XGBoost[15], LightGBM[25] and scikit-learn[3] are extremely popular. 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[11], implement optimized algorithms for tree inference[30, 31] and improve cache performance of decision tree ensembles on CPUs[23, 45] 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[22, 33].



(a) Batch size 512, depth 8

(b) Batch size 4096, depth 6

Figure 6. SilvanForge vs RAPIDs Kernel Time Speedup on NVIDIA RTX 4060 for several randomly generated models.

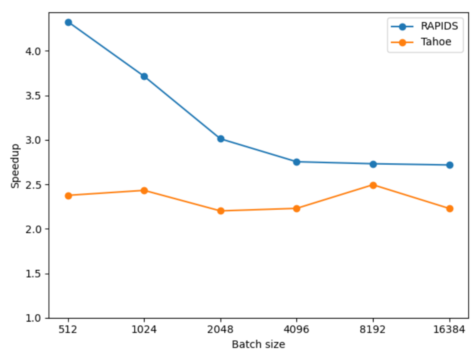


Figure 7. SilvanForge vs RAPIDs and Tahoe Kernel Time Speedup on NVIDIA RTX 4060

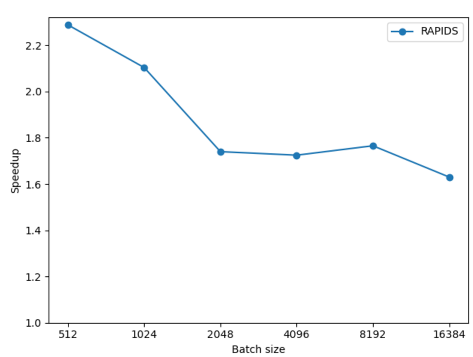


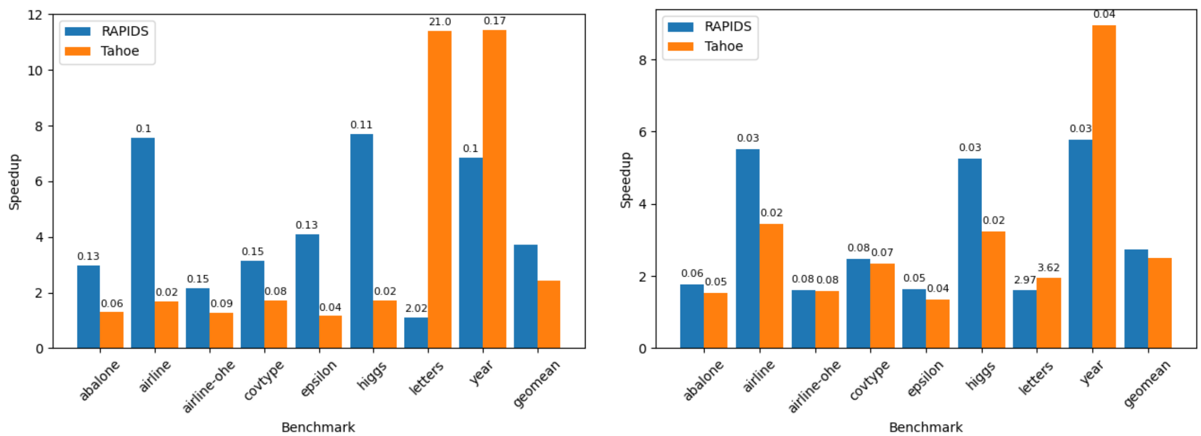
Figure 8. SilvanForge vs RAPIDs Total Time Speedup on NVIDIA RTX 4060.

Decision Tree Ensemble Compilers: Several compilers for decision tree ensembles have been proposed in the literature [4, 32, 36]. Treebeard and Treelite exclusively target CPUs and all their optimizations are designed purely for performance on CPUs. Treelite[4] is a model compiler that only generates `if-else` code for each tree in the model.

Treebeard is the work most closely related to SilvanForge. While we build on top of Treebeard, SilvanForge is a significant enhancement over Treebeard. Specifically, we introduce the scheduling language and schedule exploration while also enhancing the IRs and support for parallelizing across trees through the implementation of a novel MLIR reduction dialect.

Hummingbird[32] is a compiler that compiles traditional ML models to tensor operations, thereby enabling them to be run on tensor-based frameworks like TensorFlow[10]. Hummingbird can target both CPUs and GPUs, but, as was shown earlier [36], tensor operations are not the most efficient way to implement decision tree inference and the performance of Hummingbird is significantly lower than that of other frameworks.

Other Systems and Techniques: Ren et. al. [41] design an intermediate language and a virtual machine to enable vector execution of decision tree inference. However, this virtual machine is itself implemented by hand on different target processors. This is clearly more expensive than SilvanForge's approach. Jo et. al.[24] 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 [29, 35] have been developed to parallelize tree walks but are not a good fit for decision tree inference as the individual node predicates are simple and the overhead of an inspector-executor system would be prohibitive.



(a) Batch size 1024

(b) Batch size 8192

Figure 9. Kernel time speedup of SilvanForge vs RAPIDs on NVIDIA RTX 4060. Numbers on the bars are inference times per sample in μs for RAPIDs and Tahoe.

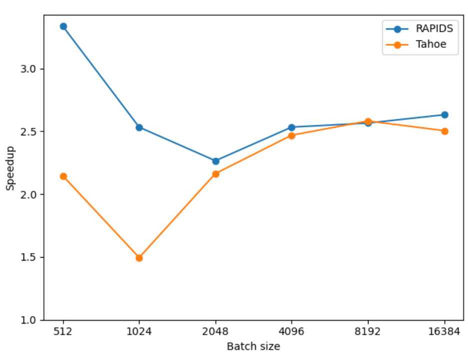


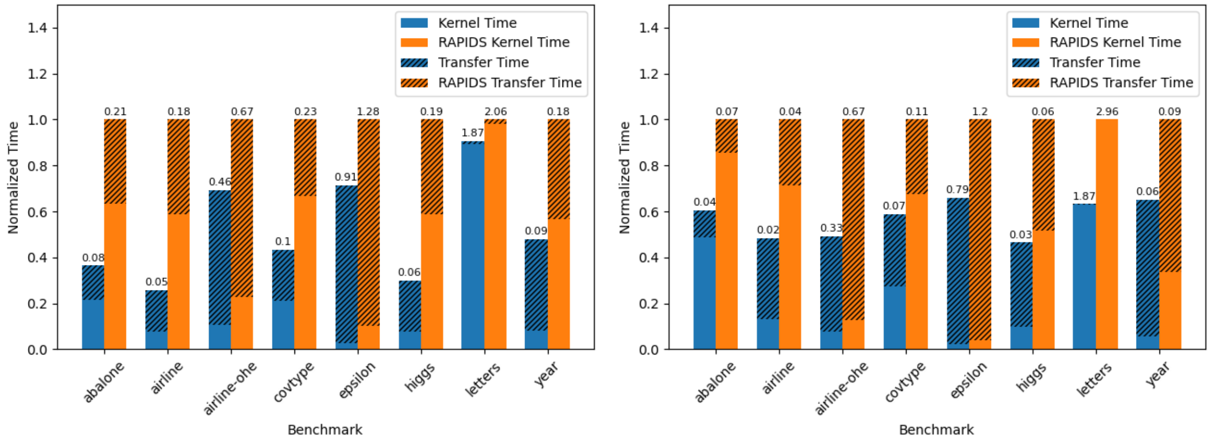
Figure 10. SilvanForge vs RAPIDs and Tahoe Kernel Time Speedup on NVIDIA T400.

Code Generation Systems from Other Domains: Several optimizing compilers and code generation techniques have been developed for other domains. TVM[16], Tiramisu[14], and Tensor Comprehensions[47] are optimizing compilers for DNNs that can target a variety of processors. Similarly, Halide[39] 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 [14, 16, 50]. 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[2, 46, 48] and signal processing[18, 38] have also been developed.

Reductions: CUB[7] and Thrust[8] 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. [40] describe language constructs in Pencil [13] 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. [44] extend Halide to add support for factoring reductions in the Halide scheduling language and to synthesize reduction operators. De Gonzalo et. al. [19] 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.

References

[1] [n. d.]. Kaggle State of Data Science and Machine Learning 2021. https://www.kaggle.com/kaggle-survey-2021. Accessed: 2022-04-16.
[2] [n. d.]. NVIDIA CUTLASS. https://github.com/NVIDIA/cutlass. Accessed: 2022-04-16.
[3] [n. d.]. scikit-learn : Machine Learning in Python. https://scikit-learn.org/stable/. Accessed: 2022-04-16.
[4] [n. d.]. Treelite : model compiler for decision tree ensembles. https://treelite.readthedocs.io/en/latest/. Accessed: 2022-04-16.
[5] 2019. RAPIDS Forest Inference Library: Prediction at 100 million rows per second. https://medium.com/rapids-ai/rapids-forest-inference-



(a) Batch size 1024.

(b) Batch size 8192.

Figure 11. SilvanForge vs RAPIDs total time comparison on NVIDIA RTX 4060. Numbers on the bars are the times per sample in μs for SilvanForge and RAPIDs. Times for each benchmark are normalized w.r.t the RAPIDs time for that benchmark.

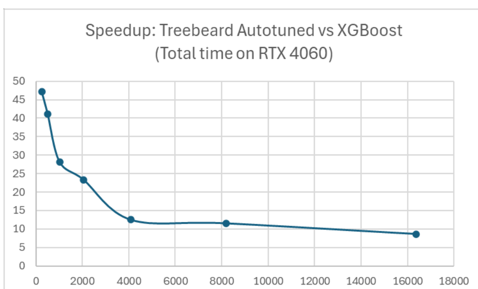


Figure 12. SilvanForge vs XGBoost Speedup on RTX 4060.

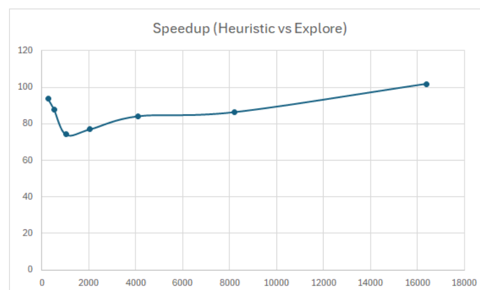


Figure 14. Autotuning heuristic compile time speedup vs full schedule exploration.

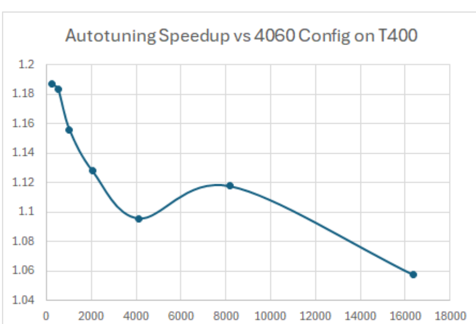


Figure 13. Autotuning heuristics speedup vs best 4060 schedule on T400.

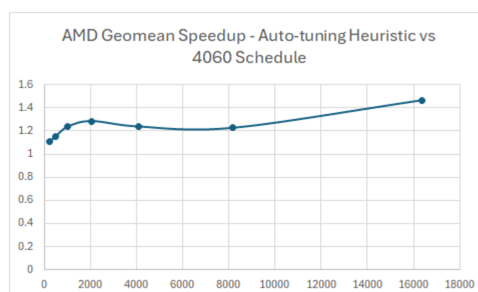


Figure 15. Autotuning heuristics speedup vs best 4060 schedule on MI210.

library-prediction-at-100-million-rows-per-second-19558890bc35. Accessed: 2024-04-15.

[6] 2020. The total cost of ownership of Amazon SageMaker. https://pages.awscloud.com/rs/112-TZM-766/images/Amazon_SageMaker_TCO_uf.pdf.

[7] 2024. CUB: API Reference for CUB. https://docs.nvidia.com/cuda/cub/index.html. Accessed: 2024-04-15.

[8] 2024. Thrust. https://developer.nvidia.com/thrust. Accessed: 2024-04-15.

[9] 2024. XGBoost GPU Support. https://xgboost.readthedocs.io/en/stable/gpu/index.html. Accessed: 2024-04-15.

[10] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry

Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. 2016. TensorFlow: A System for Large-Scale Machine Learning. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation* (Savannah, GA, USA) *(OSDI'16)*. USENIX Association, USA, 265–283.

[11] 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

[12] Ahmad Azar and Shereen El-Metwally. 2013. Decision tree classifiers for automated medical diagnosis. *Neural Computing and Applications* 23 (11 2013), 2387–2403. https://doi.org/10.1007/s00521-012-1196-7

[13] Riyadh Baghdadi, Ulysse Beaugnon, Albert Cohen, Tobias Grosser, Michael Kruse, Chandan Reddy, Sven Verdoolaege, Adam Betts, Alastair F. Donaldson, Jeroen Ketema, Javed Absar, Sven Van Haastregt, Alexey Kravets, Anton Lokhmotov, Robert David, and Elnar Hajiyev. 2015. PENCIL: A Platform-Neutral Compute Intermediate Language for Accelerator Programming. In *2015 International Conference on Parallel Architecture and Compilation (PACT)*. 138–149. https://doi.org/10.1109/PACT.2015.17

[14] 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.

[15] 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

[16] 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

[17] Dursun Delen, Cemil Kuzey, and Ali Uyar. 2013. Measuring firm performance using financial ratios: A decision tree approach. *Expert Systems with Applications* 40 (08 2013), 3970–3983. https://doi.org/10.1016/j.eswa.2013.01.012

[18] 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

[19] 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

[20] 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]

[21] 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

[22] 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.

[23] 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

[24] 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.

[25] 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.

[26] Sotiris Kotsiantis. 2013. Decision trees: A recent overview. *Artificial Intelligence Review* (04 2013), 1–23. https://doi.org/10.1007/s10462-011-9272-4

[27] Vidhi Lalchand. 2020. Extracting more from boosted decision trees: A high energy physics case study. https://doi.org/10.48550/ARXIV.2001.06033

[28] 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

[29] 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

[30] 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

[31] 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

[32] 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

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

[34] Jongsoo Park, Maxim Naumov, Protonu Basu, Summer Deng, Aravind Kalaiah, Daya Shanker Khudia, James Law, Parth Malani, Andrey Malevich, Nadathur Satish, Juan Miguel Pino, Martin Schatz, Alexander Sidorov, Viswanath Sivakumar, Andrew Tulloch, Xiaodong Wang, Yiming Wu, Hector Yuen, Utku Diril, Dmytro Dzhulgakov, Kim M. Hazelwood, Bill Jia, Yangqing Jia, Lin Qiao, Vijay Rao, Nadav Rotem, Sungjoo Yoo, and Mikhail Smelyanskiy. 2018. Deep Learning Inference in Facebook Data Centers: Characterization, Performance Optimizations and

Hardware Implications. *CoRR* abs/1811.09886 (2018). arXiv:1811.09886 http://arxiv.org/abs/1811.09886

[35] 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

[36] 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

[37] Fotis Psallidas, Yiwen Zhu, Bojan Karlas, Matteo Interlandi, Avrilia Floratou, Konstantinos Karanasos, Wentao Wu, Ce Zhang, Subru Krishnan, Carlo Curino, and Markus Weimer. 2019. Data Science through the looking glass and what we found there. https://doi.org/10.48550/ARXIV.1912.09536

[38] 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

[39] 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

[40] 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

[41] 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

[42] 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

[43] Jyoti Soni, Ujma Ansari, Dipesh Sharma, and Sunita Soni. 2011. Predictive Data Mining for Medical Diagnosis: An Overview of Heart Disease Prediction. *International Journal of Computer Applications* 17 (03 2011), 43–48. https://doi.org/10.5120/2237-2860

[44] 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.

[45] 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

[46] 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

[47] 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

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

[49] 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

[50] 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