SilvanForge: A Schedule Guided Retargetable Compiler for Decision Tree Inference

Ashwin Prasad
Indian Institute of Science and NI R&D
Bangalore, India
ashwinprasad@iisc.ac.in

Sampath Rajendra
Microsoft Research
Bangalore, India
srajendra@microsoft.com

Kaushik Rajan
Microsoft Research
Redmond, USA
krajan@microsoft.com

R Govindarajan
Indian Institute of Science
Bangalore, India
govind@iisc.ac.in

Uday Bondhugula
Indian Institute of Science and PolyMage Labs
Bangalore, India
udayb@iisc.ac.in

Abstract

The proliferation of machine learning together with the rapid evolution of the hardware ecosystem has led to a surge in the demand for model inference on a variety of hardware. Decision tree based models are the most popular models on tabular data. This paper is motivated by the problems encountered when targeting inference of these models to run at peak performance on CPU and GPU targets. Existing solutions are neither portable nor achieve the best possible performance for the specific hardware they target. This is because they do not explore and customize optimization configurations to the target processor and the model being used.

We present SilvanForge, a *schedule-guided*, *retargetable* compiler for decision tree based models that searches over several optimization choices and automatically generates high-performance inference routines for CPUs and GPUs. SilvanForge has two core components. The first is a scheduling language that encapsulates the optimization space, and techniques to efficiently explore this space. The second is an optimizing retargetable compiler that can generate code for any specified schedule. SilvanForge's ability to use different data layouts, loop structures and caching strategies enables it to achieve portable performance across a range of targets.

We evaluate SilvanForge on several hundred decision tree models across different batch sizes and target architectures. We find that our schedule exploration strategy is able to quickly find near-optimal schedules. In terms of performance, SilvanForge generated code is an order of magnitude faster than XGBoost, about 2–4× faster on average than RAPIDS FIL and Tahoe, and 2.5–3× faster than Hummingbird over several batch sizes. While most systems only target NVIDIA GPUs, SilvanForge achieves competent performance on AMD GPUs as well. On CPUs, SilvanForge is able to outperform Treebeard by up to 5× by utilizing additional sources of parallelism at small batch sizes.

CCS Concepts: • **Software and its engineering → Compilers**; **Domain specific languages**.

Keywords: Optimizing Compiler, Decision Tree Ensemble, Decision Tree Inference, Parallelization, Machine Learning, GPU

ACM Reference Format:
Ashwin Prasad, Sampath Rajendra, Kaushik Rajan, R Govindarajan, and Uday Bondhugula. 2024. SilvanForge: A Schedule Guided Retargetable Compiler for Decision Tree Inference. In *ACM SIGOPS 30th Symposium on Operating Systems Principles (SOSP '24), November 4–6, 2024, Austin, TX, USA.* ACM, New York, NY, USA, 17 pages. https://doi.org/10.1145/3694715.3695958

1 Introduction

We are in the midst of a hardware revolution, a new golden age for computer architecture [26]. 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 [24, 49]. Their robustness, interpretability, and ability to handle missing data make them a popular choice for a wide range of applications [14, 20, 31, 32, 39, 50]. The Kaggle AI report for 2023 [1] highlights the continuing *dominance of gradient boosted trees as the algorithm of choice* for tabular data. It also estimates that

SOSP '24, November 4–6, 2024, Austin, TX, USA

ACM ISBN 979-8-4007-1251-7/24/11
https://doi.org/10.1145/3694715.3695958

between 50% and 90% of practicing data scientists use tabular data as the primary type of data in their professional setting. This has also been observed in other surveys [2, 43]. Recent work has noted that the cost of inference is the most critical factor in the overall cost of deploying a machine learning model [8, 37]. 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 (typically 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 perform 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, only support a predefined combination of optimizations, and typically only target a single platform. On the CPU, XGBoost [18] 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. On the GPU, RAPIDS FIL [6] uses a reorg representation [6] and partitions trees across a fixed number of threads. Tahoe [57] uses a variation of the reorg representation and has four predefined inference strategies from which it picks one based on an analytical model. The GPU systems are CUDA based and only work on NVIDIA GPUs. Hummingbird [37], a compiler that compiles decision tree inference to tensor operations, performs all tree walks in parallel and generates a separate kernel for reduction. It is restricted to tensor based representations and supports two that are similar to the array and sparse representation (Section 6).

This paper presents SilvanForge, a novel schedule-guided compiler for decision tree inference on multiple hardware targets. 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 capable of expressing all optimizations proposed by prior work and more. Further, our schedule exploration heuristic can quickly find a near optimal schedule for the model being compiled. 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 [42] to support schedule-guided code generation, and incorporate several new optimizations that are critical to target GPUs. These two components of SilvanForge are intertwined, each benefitting from the other.

We evaluated SilvanForge on a large number of synthetic and real-world benchmark models and compared against state-of-the-art systems RAPIDS [10], Tahoe [57], Hummingbird [37], XGBoost [18] and Treebeard [42]. SilvanForge's geomean speedup (across all benchmarks) over several batch sizes is 2.5–4× over RAPIDS, 2–2.5× over Tahoe, and 2.5–3× over Hummingbird. The geomean speedup over XGBoost is more than 10×. While most systems [5, 10, 17, 18, 30, 42, 57] only run on specific hardware, SilvanForge can provide portable performance across a range of target hardware including AMD GPUs and CPUs. SilvanForge also enables better scalability on CPUs by parallelizing across multiple dimensions. We also demonstrate that SilvanForge's scheduling heuristic is able to find near-optimal schedules in less than 30 seconds on average.

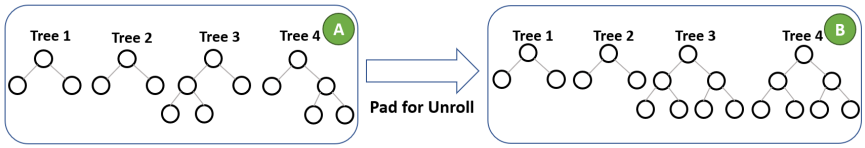
In summary, this paper makes the following important contributions.

- We identify that there are many different ways to implement decision tree inference. Often, the best implementation depends both on the model and the target processor. We design a scheduling language that allows us to represent this solution space.
- We propose a retargetable compiler that can generate code as specified by the schedule. The compiler also includes new abstractions for caching, reduction and representing models in memory. These are critical to generate efficient GPU code.
- Since the scheduling language can represent an unbounded number of schedules, we design and implement techniques to search over this space and find high-performance schedules in tens of seconds.
- We exhaustively evaluate SilvanForge and report that it achieves portable performance across a range of hardware targets. It consistently outperforms state-of-the-art decision tree inference frameworks for both CPU and GPU.

2 Motivation

In this section, we first motivate our work by showing how a model can be compiled in different ways. Subsequently, we show the drastic performance difference across these variants for real benchmarks.

Consider a model with four trees, two trees of depth 1 and two of depth 2 (A in Figure 1a). We first describe the simple schedule shown in Figure 1b that processes one tree



(a) A sample model and an illustration of the effect of the schedule construct `unroll`

```
reorder(tree, batch)
// Split 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)
```

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

(b) A serial CPU schedule

```
// Split the trees into two sets
tile(tree, t0, t1, 2)
reorder(batch, t1, t0)
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)
```

```
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(model, t0_depth1 + t1)
      treePred = walkDecisionTree(T,
                    input[batch]) <unrollDepth = 1>
      reduce(result[batch], treePred) <'+', 0.0>
    for t0_depth2 = 2 to 4 step 2:
      T = getTree(model, t0_depth2 + t1)
      treePred = walkDecisionTree(T,
                    input[batch]) <unrollDepth = 2>
      reduce(result[batch], treePred) <'+', 0.0>
```

(c) A GPU schedule

Figure 1. Figure 1a shows the model for the motivating example. Figure 1b and 1c show some possible schedules to compile the motivating example and the generated SilvanForge IR. The arrows connect entities in the schedule to related entities in the IR. Indices that go over trees are shown in **orange** and those that go over input rows are shown in **green**.

at a time for all input rows.[1] In this schedule, the loop over the trees is split into two – 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). Straightforward traversal of trees requires a `while` loop and involves branching to check if a leaf has been reached. One way to avoid this, as done in this schedule, is to unroll the tree walks for each tree. When the schedule specifies that the tree walks should be unrolled, SilvanForge pads the trees so that all leaves of a tree are at the same depth (B in Figure 1a). The concrete implementation of this schedule (in SilvanForge's IR) is as shown in Figure 1b.[2] This schedule is ideally suited for a single-core CPU. It maximizes the reuse of trees in the L1 cache. However, it doesn't exploit any parallelism.

One form of parallelism that can be exploited is to process rows in parallel. While this may work for multi-core CPUs,[3] with massively parallel processors like GPUs, this strategy may not yield sufficient parallel work. To expose more parallelism, we can additionally parallelize across trees as done in the schedule in Figure 1c. The corresponding GPU inference routine is shown in the same figure. This schedule generates a routine that processes one input row per thread block (`batch` loop is mapped to `grid.x`). The schedule also tiles the `tree` loop into a nest of two loops with indices $t0$ and $t1$. It then additionally parallelizes across the $t1$ loop. Finally, it splits $t0$ and unrolls the walks for each tree depth.

Note that while unrolling helps avoid branching, it increases the total amount of computation. Another option is to not unroll and let the GPU manage the branching. The schedules with and without unrolling place different constraints on the target processor, and the best choice depends on the characteristics of the model and micro-architectural features like register file size and handling of branch divergence [22, 48].

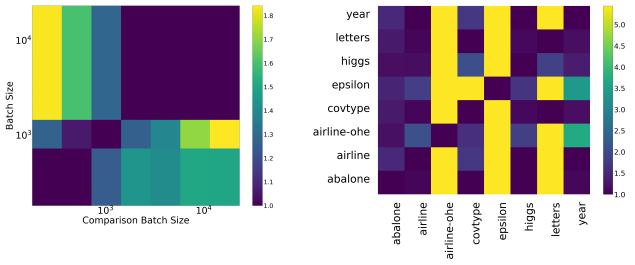
These SilvanForge schedules just use a combination of 4-8 constructs and already generate strategies that are different from what existing systems like Hummingbird, XGBoost, Tahoe and RAPIDs FIL use. As one can imagine, several other schedules with different trade-offs can be generated using these constructs.

2.1 Performance of Different Schedules

To establish the importance of choosing the right schedule, we compare the performance of the schedules generated by SilvanForge on several real-world benchmarks. Figure 2a shows the variation in performance when the best schedule for a given batch size is used across different batch sizes for one of the models. Figure 2b shows the variation when schedules are used across models at a fixed batch size. As can be seen, performance degrades by 2× when the best schedule for a smaller batch size is used for a larger batch size and vice-versa. Across all our benchmarks, the largest slowdown is 5×. The degradation is much worse when schedules are used across different models. In many (∼ 20%) instances, using the best schedule for one model on another results in a 5× slowdown. As we report in Section 9, reusing schedules

[1] Since the scheduling constructs are fairly intuitive, we defer a detailed explanation of the scheduling language to Section 3.
[2] We do not show the full listing of the IR. Instead, we present inference routines as pseudo-code for the sake of clarity.
[3] As we report in Section 9, other ways to parallelize can benefit inference on CPUs as well.

across different architecture also leads to significant slowdowns. Clearly, using a single strategy across models, batch sizes and targets leaves significant performance on the table.



(a) `covtype` batch sensitivity **(b)** Model sensitivity at batch size 4096

Figure 2. Batch and model sensitivity plots. Each point shows the slowdown when the best schedule for the x-axis batch size (model) is used for the y-axis batch size (model).

These performance considerations, coupled with the importance of running ML applications on a diverse set of hardware targets, motivates the need for a retargetable compiler for decision tree inference. Building such a configurable compiler and supporting code generation for CPUs and GPUs required us to solve several fundamental problems. The rest of the paper describes these challenges in detail and how we solved them in SilvanForge.

3 SilvanForge's Scheduling Language

SilvanForge's 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. The configurability provided by the schedule allows us to build automatic schedulers (Section 8).

3.1 Language Definition

The core construct of SilvanForge's scheduling language is an ***index variable*** which abstractly represents a loop. Each index variable has a range of values that it can take along with a step. The language 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.[4] 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 (Table 2). Finally, we have a class of attributes that enable reduction specific optimizations (Table 3).

[4]The canonical inference routine has a batch loop that goes over all rows and a loop that goes over all trees nested within it.

The compiler internally represents loops (index variables) as nodes in a tree where the children of a node represent immediately contained loops. Each schedule primitive modifies this tree in some way. Also, the compiler automatically infers the ranges of all loops by tracking its lineage.

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**	Perform loop fission on the loop represented by **indexVar** at iteration **splitIter**. Resulting loops will be represented by **first** and **second**.
`reorder`	**indices[]**	Permute the specified loops. The loops must be *perfectly nested*, i.e. only the innermost loop contains program statements.
`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.

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). Walk multiple trees simultaneously by interleaving instructions at a fine granularity [42].
`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.

3.2 Expressiveness of the Scheduling Language

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 [18] and Tahoe's strategies [57].

XGBoost [18] implements inference on the CPU by going over a fixed number of rows (64 in the current version) for every tree and then moving to the next tree. It moves to the next set of rows when all trees have been walked for

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.

the current set of rows. Different sets of rows are processed in parallel. This is expressed in SilvanForge's scheduling language as:

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

Tahoe [57] 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.

In the *direct method* [57], a single GPU thread walks all trees for a given input row. The schedule for this strategy is:

```
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 [57], a thread block walks all the trees for a given row in parallel. Input rows are cached in shared memory. The schedule for this strategy is:

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

In summary, SilvanForge's scheduling language provides a convenient way to encode a wide range of strategies and to control how the compiler lowers the inference routine to optimized target code. The next section discusses how the rest of the compiler is structured.

4 Overview of SilvanForge 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 extends the open-source Treebeard compiler [42]. The Treebeard compiler is built using MLIR [33] and generates efficient CPU code for decision tree inference. Treebeard lacks a scheduling language and does not support code generation for different implementation strategies. We enhance Treebeard to support schedule-guided compilation for CPUs and GPUs. The parts of SilvanForge that are new or significantly different are shown as shaded boxes in Figure 3.

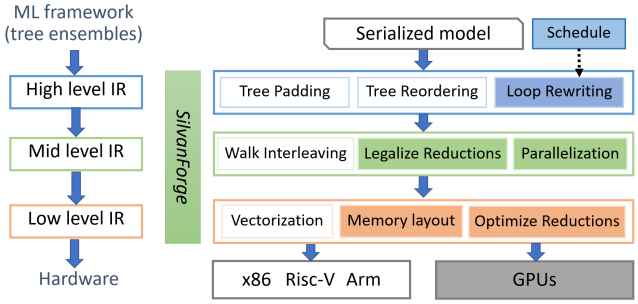


Figure 3. SilvanForge compiler structure.

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 Treebeard with loop rewrites on the HIR that are implemented through the scheduling language (Section 3). The *schedule* is implemented as an MLIR attribute on the `predictEnsemble` operation. We use this object to implement the automatic scheduling described in Section 8. We reuse HIR transformations to reorder and pad trees from Treebeard. The tree transformations that reorder and pad trees are used in conjunction with loop transformations like splitting to specialize inference code as the example in Section 2 shows.

The HIR is lowered to MIR as dictated by the *schedule*. Optimizations like tree-walk unrolling and tree-walk interleaving are performed on the MIR. One surprising thing we found while developing SilvanForge was that we could use ILP to improve performance on GPUs. One of the performance bottlenecks in inference code targeted to GPUs was that warps spent significant time being stalled. We were able to alleviate this bottleneck by interleaving tree walks.

In the generated 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.

The MIR is further lowered to a low-level IR (LIR). Significant changes to Treebeard were required to get LIR to correctly lower to GPU code. The most important of these was changing how the compiler implements support for in-memory representations of models (Section 6). Also, when the target processor is a GPU, the required memory transfers and kernel invocations are inserted into the LIR. 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.

4.1 Portability and Reuse

From a user's perspective, there is no difference between targeting CPUs and GPUs using SilvanForge. The same scheduling language is used to specify the structure of inference code on CPUs and GPUs. When a schedule does not map any loops to a GPU dimension, CPU code is generated.

The implementations of the scheduling language, HIR and MIR, and optimizations on them are fully shared between the CPU and GPU compilation pipelines. The compilation pipelines diverge starting at the point where MIR is lowered to LIR. Even so, we are able to share a significant amount of code between the CPU and GPU lowering because of how in-memory representations are abstracted (Section 6).

5 Representing and Optimizing Reductions

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 currently supports parallel reductions for value types (`scf.reduce`) and reducing dimensions of tensors (`linalg.reduce`). Neither of these meet the requirements for generating code for decision tree inference. Lowering reductions of arbitrary value types to GPUs is non-trivial and MLIR currently does not have support for it. With `linalg.reduce`, the reduction cannot be fused with another loop as the tensor of values to be reduced needs to be materialized first. To address this gap, we design an MLIR dialect that allows us to specify accumulating values into an element of a multidimensional 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 multidimensional array (represented by an MLIR `memref`). Consider the following SilvanForge schedule. In our example, `N_t` is the number of trees and `batch_size` is the batch size. The schedule tiles the tree loop and parallelizes the resulting outer loop.

```
tile(tree, t0, t1, N_t/2);
reorder(t0, t1, batch);
parallel(t0);
```

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

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

The compiler simply generates a `reduce` op to perform the required parallel reduction. 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 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.

In our example, 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]
float partResults[2][batch_size]
model = ensemble(...)
par.for t0 = 0 to N_t step N_t/2:
  for t1 = 0 to N_t/2:
    for batch = 0 to batch_size:
      t = getTree(model, t0 + t1)
      p = walkDecisionTree(t, rows[batch])
      reduce(partResults[t0/(N_t/2)][batch], p)

results = reduceDimension(partResults[:, :], 0)
```

The op `reduceDimension` reduces values across the specified dimension of an n-dimensional array. Here, it reduces all elements of the first dimension (dimension 0). and produces a result memref with a single dimension of size `batch_size`.

To reduce the amount of memory used by arrays introduced for reduction, we introduce the `reduceDimInplace` operation. It is similar to the `reduceDimension` 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. We use this op to compute intermediate results when several reduction loops are identified.

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 compare feature values and thresholds (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 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. Different IR levels (HIR, MIR and LIR) are separated by double lines.

5.1 Lowering Reduction Operations

For both CPU and GPU, `reduce` is lowered to a sequence of load, compute and write operations. This is possible because legalization ensures that parallel threads do not write to the same array element.

For CPUs, we lower `reduceDimInplace` and `reduceDimension` 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, the lowering passes generate vector (LLVM IR) instructions for the reduction.

The same reduction abstractions can be lowered to efficient GPU implementations and simplify higher-level code generation. SilvanForge can lower these ops to exploit either the parallelism across independent reductions or the inherent parallelism in a single reduction. It does this by using a divide and conquer reduction strategy when there aren't enough independent reductions to keep all threads in a thread block busy.

Additionally, if the schedule specifies that the reduction needs to be performed using shared memory, the privatized buffer is allocated in shared memory. Our abstractions for reductions allow our lowering passes to be agnostic 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.

SilvanForge's reduction support, including the lowering implementations for ops in the reduction dialect, is fairly general. At present, only the process of identifying reduction loops is specific to decision tree inference.[5]

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 as proposed in Treebeard [42]. The reorg representation is the representation used by the RAPIDs library [6]. 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.

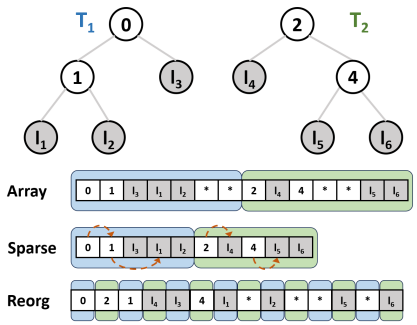


Figure 4. The three representations supported by SilvanForge.

We change the design of Treebeard [42] 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. Crucially, 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 how threshold and feature indices are read from the allocated buffers.

[5]The compiler uses annotations to mark loops as reduction loops. Implementing a different analysis to annotate reduction loops is straightforward.

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.

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. 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). When 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 one iteration 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 cache operations are lowered to prefetches 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. The lowering for the `cacheRows` operation is implemented directly in the SilvanForge compiler as a series of coalesced loads 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.

8 Exploring the Schedule Space

The set of schedules that can be constructed using the scheduling language described in Section 3 is unbounded. Additional tools are required to help search for a high-performance schedule. Like with other schedule-guided ML compilers [19], we expect schedule exploration will be performed offline (before model deployment), but spending more than a few minutes on this search is likely not acceptable. We propose a set of heuristics to guide the search for a good schedule for GPUs. These heuristics work by first defining a bounded search space and then pruning this search space further based on the batch size and model characteristics.

8.1 Bounding the search space

We bound the space using a few meta-parameters that together define the primitives used to construct a template schedule. We do so while making sure that the space (atleast) covers the known strategies published in prior work. Specifically, we use the 9 parameters listed in Table 5. We picked the specific values listed through experimentation. The first three numeric parameters assign a configurable number of rows to each thread block, to each thread, and determine how trees are distributed across a specified number of threads. These parameters together define the loop schedule primitives to use (including the arguments to pass) from Table 1 and the parallelization strategy. The next four parameters determine the caching strategy, unrolling strategy and how many trees to traverse simultaneously within a single thread (the remaining primitives from Table 2). The last two parameters determine reduction optimizations (Table 3) and the layout[6] to use.

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	$\{True, False\}$
Representation	$\{array, sparse, reorg\}$

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.

We evaluated all the schedules in this space on several real world models and observed that there is huge variation in performance with different parameter values. Figure 5 shows the distribution of normalized execution times for our real-world benchmark models with different parameter values (inference times normalized w.r.t fastest time for that model). As can be seen, very few schedules perform close to the best while a vast majority perform poorly.

Exhaustive exploration over this bounded space (5184 schedules) is still very expensive, taking between 30 – 300 minutes to explore the entire space for a given model.

[6]This does not have a schedule primitive. We always explore all three layout options.

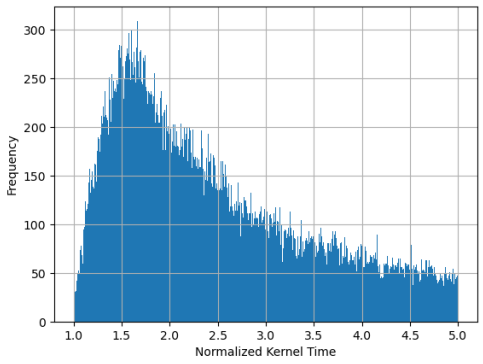


Figure 5. Distribution of normalized execution times for real-world benchmark models with different combinations of parameter values shown in Table 5

.

8.2 Pruning the search space

A careful analysis of the search space reveals that certain schedules are not likely to perform well as they either do not exploit the parallelism available in the model or do not take advantage of the hardware features. We use a combination of four strategies to further prune the search space. Algorithm 1 presents our final heuristic to find a good schedule.

Insufficient parallelism. Some configurations do not expose sufficient parallelism, we prune these out. For example when the batch size is small, it does not make sense to have many rows per thread block or to partition the trees across a few threads. We therefore limit the combinations used based on batch size (see lines 3—8).

Utilizing shared memory. Shared memory is a critical resource on GPUs and it helps to only bring objects that would be reused into it. We observe that tree nodes have limited reuse and the one time cost of loading trees is not sufficiently amortized when the whole tree is not accessed during inference. On the other hand caching rows almost always improves performance. Further when the inputs have many features, it may not be possible to cache all rows. In this scenario it helps to retain a few rows in memory, and pick schedules similar to the small batch size case (lines 15,3—5).

Unrolling walks. Unrolling tree walks completely when multiple walks are being interleaved leads to performance improvements, while unrolling non-interleaved walks is sometimes detrimental to performance without any significant gains in other cases. We therefore choose to unroll walks only when they are interleaved (line 17).

Orthogonal parameters. We find that some parameters like reduction type are orthogonal to the rest. We therefore break the exploration into two phases, first we pick the best

schedules without reduction optimizations and then evaluate reduction options on the best schedules. Evaluating the top 3 schedules for reduction is sufficient in practice.

SilvanForge performs an exhaustive search over the pruned schedules to find the best one. The model is compiled with each of these schedules and evaluated on a few input batches. The best schedule among all the evaluated schedules is selected as the schedule to use. We report that this heuristic is able to find schedules that are close to the best schedules while improving the search time by two orders of magnitude (see Section 9).

Algorithm 1 Heuristic to find a good schedule

```
procedure TBConfigs(N_batch, N_f)
    T_batch ← 2048, T_f ← 128
    if N_batch ≤ T_batch or N_f > T_f then
        rowsPerBlock ← {8, 32}
        treeThreads ← {20, 50}
    else
        rowsPerBlock ← {32, 64}
        treeThreads ← {2, 10}
    end if
    return rowsPerBlock, treeThreads
end procedure

bestSchedules ← shMemSchedules ← ∅
rowsPerTB, treeThds ← TBConfigs(N_batch, N_f)
cacheRows ← True, cacheTrees ← False
interleave ← {1, 2, 4}
unroll ← interleave ≠ 1
schedules ← (rowsPerTB, treeThds, cacheRows,
                cacheTrees, interleave, unroll)
for (sched, rep) ∈ schedules × {array, sparse, reorg} do
    time ← EvaluateSchedule(sched, rep)
    bestSchedules.insert(time, sched, rep)
end for

for sched, rep ∈ Top3(bestSchedules) do
    EnableSharedReduction(sched)
    time ← EvaluateSchedule(sched, rep)
    shMemSchedules.insert(time, sched, rep)
end for
return min(shMemSchedules ∪ bestSchedules)
```

8.3 CPU Schedule Exploration

For CPUs, searching over a small set of pre-defined schedules with different parallelization strategies (parallelize across rows, parallelize across trees, parallelize across both) and interleaving factors works well. As the set of schedules to explore is small, exhaustive search is sufficient.

9 Experimental Evaluation

We evaluate SilvanForge on four different target processors, a low resource NVIDIA T400 GPU with 2GB RAM, a medium-tier NVIDIA RTX 4060 GPU with 8GB RAM, a large AMD MI210 GPU with 64GB RAM, and an Intel Core i9-11900K (Rocket Lake) CPU with 16 virtual cores and 128 GB RAM. We compare SilvanForge with five other systems: NVIDIA RAPIDS [10] v23.10, Tahoe [57], Hummingbird [37] v0.4.11[7] running on PyTorch v2.4.0, XGBoost [18] v1.7.6 and Treebeard [42] CPU. We measure both kernel time and total time (including time for data transfers to the GPU and back) for RAPIDS, only kernel time for Tahoe[8] and total time for Hummingbird.[9] For SilvanForge, we use schedules found using the schedule exploration heuristic (Section 8) unless otherwise specified.

We use two sets of benchmarks to perform the comparison. We use 8 real-world models trained on data from the Intel Machine Learning Benchmark suite [7]. These models were also used to evaluate Treebeard. To enable more exhaustive evaluation we generated 700 random models with varying number of trees (100—1000) and features (powers of two in the range 8—1024). Each tree has leaves at depths 2 to a maximum depth of 6, 7 or 8.

9.1 Performance comparisons on NVIDIA GPUs

Real-world Models. Figure 6a shows the geomean speedup of SilvanForge over RAPIDS and Tahoe at different batch sizes on RTX 4060. We do not show results for XGBoost since the speedups are an order of magnitude higher (9×—40×) and too large to fit on the same graph.

Figure 6a has lines for kernel time and total time speedup over RAPIDS and kernel time speedup over Tahoe. As can been seen SilvanForge is uniformly faster than Tahoe[10] by 2 – 2.5× at all batch sizes. Compared to RAPIDS, SilvanForge is about 4× faster at batch size 512. The relative performance of RAPIDS improves with batch size from 512—4096 as RAPIDS is optimized for larger batches. SilvanForge is still consistently faster by 1.5 – 2× all the way up to a very large batch size of 16k. The plot also shows that the speedups are significant even when the overhead of data transfer to and from the GPU is included. They are lower than kernel time speedup as both systems have a constant additional transfer overhead.

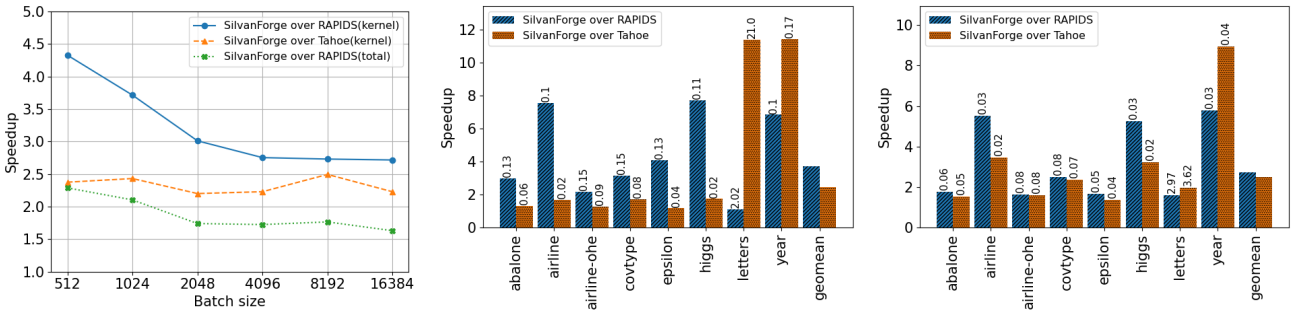
Figures 6b and 6c show per benchmark results at two different batch sizes. SilvanForge is able to find better schedules than both RAPIDS and Tahoe on each of the benchmarks. It

[7] This is the current version of Hummingbird as of September 2024, and has several improvements compared to the system used in the Hummingbird paper.

[8] 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.

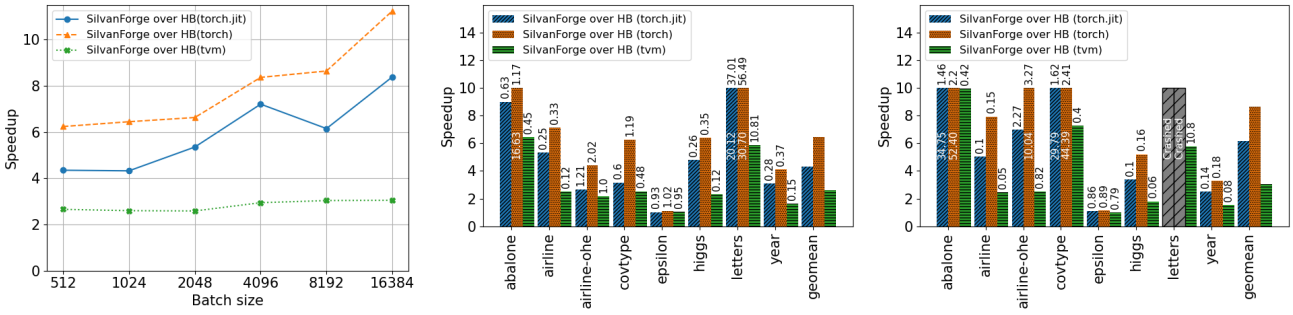
[9] We report only total time comparisons with Hummingbird because we could not find a reliable method to measure the kernel times for Hummingbird.

[10] Tahoe does not support multiclass models. To enable comparison, we ran the multiclass models (`covtype` and `letters`) as regression models with Tahoe. We also noticed that some variants of Tahoe produce 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.



(a) Speedup vs batch size. **(b)** Batch size 1024. **(c)** Batch size 8192.

Figure 6. The first graph shows kernel and total time speedup of SilvanForge over RAPIDS and Tahoe (geomean over real-world benchmarks) across batch sizes on NVIDIA RTX 4060. The bar graphs show the kernel time speedup of SilvanForge per benchmark. Numbers on the bars are inference times per sample in μs for RAPIDS and Tahoe.



(a) Speedup vs batch size. **(b)** Batch size 1024. **(c)** Batch size 8192.

Figure 7. The first graph shows the total time speedup of SilvanForge over different Hummingbird backends (geomean over real-world benchmarks) across batch sizes on NVIDIA RTX 4060. The bar graphs show the total time speedup of SilvanForge per benchmark (clipped to a maximum of 10×). Numbers on the bars are inference times per sample in μs for Hummingbird.

consistently outperforms both systems, achieving a speedup of upto 12×. For about half the benchmarks, the speedup is 2× or more over both baselines at batch size 8192.

Figure 7 shows the result of our comparison with Hummingbird. Figure 7a shows that SilvanForge is significantly faster than all supported Hummingbird backends, even when the transfer time is taken into account. As noted in the Hummingbird paper, the TVM backend is the fastest. SilvanForge is $2.5-3\times$ faster than the TVM backend. It is $4-8\times$ faster than the `torchscript` backend and $6-10\times$ faster than the `torch` backend.

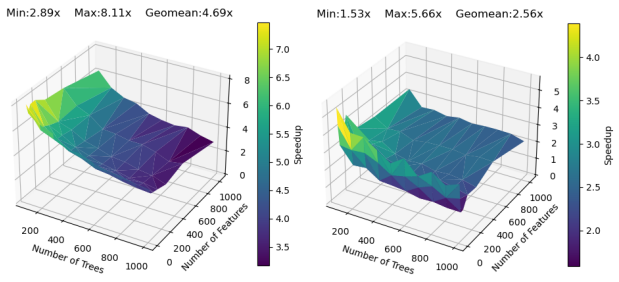
Figures 7b and 7c show that SilvanForge is able to find schedules that out-perform all Hummingbird backends. Total time speedups are greater than 2× for most benchmarks,[11] which implies that SilvanForge is able to generate significantly faster kernels.

[11]Performance for `epsilon` is the same for both SilvanForge and Hummingbird as this benchmark's total time is dominated by transfer times due to the large number of features (2000).

Synthetic Models. To establish that SilvanForge can consistently find better schedules, we compared SilvanForge to RAPIDS and Hummingbird on the synthetic models. Figure 8 and Figure 9 plot the results at batch sizes 512 and 4096 on RTX 4060. The plots show the speedup of SilvanForge over RAPIDS and Hummingbird along the z-axis, with the x-axis and y-axis representing the number of trees and the number of features.

As Figure 8 shows, while the exact trends at different batch sizes vary, SilvanForge consistently outperforms RAPIDS by $1.5-8\times$. Along the tree dimension we find that the speedup is very high with fewer trees and stabilizes at around 2× from 300 onwards. We note that SilvanForge continues to scale well when the number of trees are increased even further. It achieves a speedup of around 2× compared to RAPIDS on `letters`, a real-world benchmark with 26K trees.

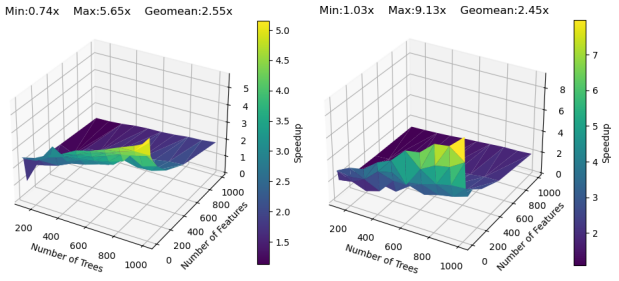
Figure 9 shows that SilvanForge significantly outperforms Hummingbird on all models at batch size 4096 and all except one (64 features, 100 trees), at batch size 512. We also observe that the speedup offered by SilvanForge is



(a) Batch size 512. **(b)** Batch size 4096.

Figure 8. Kernel time speedup of SilvanForge over RAPIDS for random models.

higher when the number of trees in the model is larger and the number of features is smaller (since the data transfer overheads are smaller). Overall, SilvanForge is faster than Hummingbird by about 2.5× at both batch sizes.



(a) Batch size 512. **(b)** Batch size 4096.

Figure 9. Total time speedup of SilvanForge over Hummingbird for random models.

Utility of Schedule Constructs. A detailed analysis of the generated schedules indicate that caching, in-memory representation and reduction optimizations are crucial for performance. We observe that all three representations are necessary, array representation is used 60% of the time, sparse 30% of the time and reorg 10% of the time for the best schedules on the NVIDIA 4060. Next, shared reduction significantly improves performance especially for multi-class models. For example, the `letters` and `covtype` benchmarks achieve speedups of 1.5× with shared reduction (over the best schedule without shared reduction). Shared reduction increases the memory pressure (on shared memory) and not all schedules use it. Finally, while GPUs have a hardware managed cache, explicitly caching the rows in shared memory using the scheduling primitives helps exploit the reuse of rows across trees. We find that caching rows is beneficial for many models with speedups as large as 1.5× over the best schedule without caching.

Comparison on T400. To test the portability of SilvanForge's techniques, we compare SilvanForge on the T400 GPU with RAPIDS and Tahoe. Figure 10 shows that even on the smaller GPU, SilvanForge is able to outperform them consistently (speedup always greater than 1.5×). As we adjust the batch size, we notice trends that are similar to those observed on the RTX 4060.

9.2 Performance on AMD GPUs

While Tahoe, RAPIDS and XGBoost only support NVIDIA GPUs, we demonstrate that SilvanForge is able to generate competitive code for AMD GPUs as well. SilvanForge can target AMD GPUs because it generates a combination of MLIR's gpu dialect and LLVM IR and these can be JIT'ed to AMD GPUs. While our objective here is not to compare directly between AMD and NVIDIA GPUs, we do find that the MI210 (which is more powerful) achieves better inference times on most benchmarks at large batch sizes (≥ 8k). For example, at a batch size of 16k, the MI210 is 2× faster than the RTX 4060 on the `letters` benchmark.

9.3 Evaluation of the Schedule Exploration Heuristic

We evaluated the schedule exploration heuristic described in Section 8 on several fronts.

Efficacy and Speed of Scheduling Heuristic. Figure 11 compares the best schedule found by exhaustive exploration on the RTX 4060 with the schedule found by the exploration heuristic. The plot establishes that the heuristic is able to find schedules that are very close to the best schedule (well within 5%). Importantly, the heuristic method is able to find good schedules in a fraction$(1/80 - 1/100)$ of the time taken by exhaustive exploration. The exploration time ranges between 6 and 167 seconds for the heuristic with a mean of 28.7 seconds. These results show that our heuristic is able to quickly find schedules that are close to the best schedule.

Schedule Sensitivity Across GPUs. We designed additional experiments to evaluate whether the best schedule (as picked by the heuristic) from one GPU can be used on another. Figure 12 reports the geomean performance improvements of using the best schedule on T400 and MI210 compared to using the best schedule found on the RTX 4060 (for a given batch size). As can be seen the T400 specific schedule is 1.05× to 1.2× better with the maximum difference being 2× (`epsilon` at batch size 512). There is a much larger variation in performance on MI120. For example, the geomean speedup over all benchmarks is 1.5× at batch size 16k (maximum 2× for the `letters` benchmark). In conclusion, schedules found on one GPU do not carry over to other GPUs and often result in sub-optimal performance. Schedules need to be tuned on each target to achieve the best performance.

9.4 CPU Improvements

The enhancements made to the compiler enable SilvanForge to explore additional schedules on the CPU compared

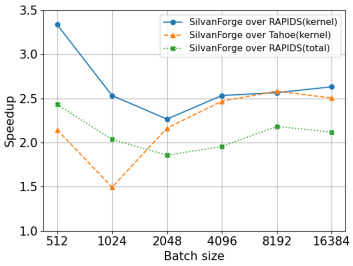


Figure 10. Speedup of SilvanForge over RAPIDS and Tahoe across batch sizes on T400.

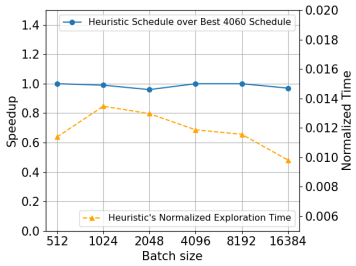


Figure 11. Slowdown of heuristic vs best schedule on RTX 4060 and heuristic schedule exploration time normalized w.r.t. full exploration time.

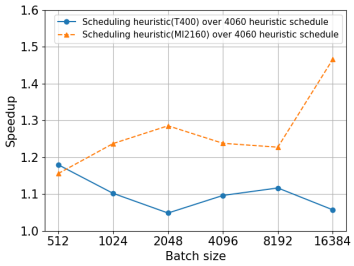


Figure 12. Geomean speedup of a hardware specific schedule on T400 and MI210 vs the best schedule from RTX 4060.

to Treebeard. In particular, we find that the ability to parallelize across trees improves performance significantly at small batch sizes. At batch size 32, we find that the geomean speedup over all 8 real-world benchmark models is 2.2× with a max speedup of 5×. At batch size 64, the average speedup is 1.1× with a max speedup of 2×. At batch size 32, parallelizing across trees is faster for all models and at batch size 64 the Treebeard schedule that parallelizes across rows is faster for only 2 of the 8 models. For small batch sizes, parallelizing across rows does not offer the best performance as there is limited reuse of trees in L1 cache. Also, the amount of work per thread is very small leading to high overheads. Parallelizing across trees addresses both these problems.

Overall, our evaluation shows that SilvanForge is able to efficiently generate high-performance code for processors ranging from NVIDIA and AMD GPUs to Intel CPUs. On all platforms and models that we tested on, SilvanForge significantly outperforms state-of-the-art systems like RAPIDS, Tahoe, Hummingbird, XGBoost and Treebeard.

10 Discussion

This section qualitatively compares SilvanForge's scheduling language with that of prior systems and provides details on the wide range of use-cases SilvanForge can support.

10.1 SilvanForge's Scheduling Language

Unlike existing scheduling languages that target regular computations [19, 45], SilvanForge addresses the challenges associated with decision tree inference. A key contribution of our work is the characterization of a scheduling space for this domain and the demonstration of significant performance variations across different scheduling strategies. The proposed scheduling language builds on a loop-centric foundation, extending it with SilvanForge-specific operators that enable domain-specific optimizations.

This approach is similar in spirit to how TVM [19] extends the Halide [45] scheduling language with domain-specific primitives for deep neural network (DNN) compilation, such as *tensorization* and *cooperative data loading*. While TVM and Halide both target optimizations in similar regular domains, the introduction of these additional primitives allows TVM to express optimizations beyond those supported by Halide.

10.2 Choice of Hardware Platform and Batch Size

Decision tree models are used in a wide variety of applications, ranging from interactive web applications to data-science pipelines that process large volumes of data. The latency and throughput requirements of an application heavily influence whether to target inference computation to the CPU or GPU, as well as the optimal batch size. For interactive applications requiring low latency, CPUs with small batch sizes are often the best fit. Our evaluation shows that for these smaller batch sizes, CPUs perform competitively, with the primary overheads of GPU usage being kernel launch and transfer overheads. While the exact crossover point between CPU and GPU depends on the model, we expect it to be close to 100 rows in most cases. A parameter for this can easily be added to our scheduling heuristic.

Larger batch sizes, while increasing latency, offer significant throughput improvements due to better reuse of model parameters and more efficient parallelization. Applications processing large volumes of data together can capitalize on larger batches, where GPU acceleration provides both performance and cost benefits. The two NVIDIA GPUs we ran experiments on are much cheaper than the CPU we used, yet they deliver an order of magnitude better performance.

In summary, SilvanForge addresses the needs of a wide range of applications by allowing users to explore the trade-offs by seamlessly targeting computations to either CPU or GPU. This enables data scientists to fully leverage the processing capabilities of their hardware.

11 Related Work

This section discusses prior work related to SilvanForge.

Decision Tree Inference Systems: Tahoe [57] is a library-based system that picks between four predefined strategies to implement decision tree inference on GPUs. RAPIDS

FIL [6], NVIDIA's library for decision tree inference, implements some heuristics to pick a good configuration for every model.[12] But these techniques are limited, and the library uses a single strategy (splits trees across 256 threads) and in-memory representation (reorg) for all models. Similarly, XGBoost's [18] GPU library [12] uses the same strategy for all models and batch sizes.

Some compilers for decision tree ensembles have been proposed in the literature [5, 37, 42]. Treebeard and Treelite exclusively target CPUs and all their optimizations are designed purely for performance on CPUs. Treelite [5] is a model compiler that only generates `if-else` code for each tree in the model. lleaves [17], a system similar to Treelite, generates `if-else` code for each tree in the model and can only target CPUs. Treebeard [42] is the work most closely related to SilvanForge. While we build on top of Treebeard, SilvanForge is a significant enhancement over Treebeard (Section 4). Hummingbird [37] compiles decision tree inference to tensor operations, thereby enabling them to be run using tensor-based frameworks like PyTorch [40]. The generated tensor code performs all tree walks in parallel, stores the individual tree predictions in a tensor and finally performs a reduction. This approach often produces multiple kernels as all tensor operators cannot be fused. For example, the reduction operator needs a kernel of its own.

On CPUs, XGBoost [18], LightGBM [30] and scikit-learn [4] are extremely popular. Yggdrasil [25] is a system that integrates with several libraries, and provides easy to use abstractions for decision tree training and inference. A recent paper [53] describes an adaptive mechanism to pick one of a few predefined parallelization and vectorization strategies. Other systems that hide dependency stalls by interleaving tree walks [13], implement optimized algorithms for tree inference [35, 36] and improve cache performance of decision tree ensembles on CPUs [28, 52] have been proposed. Some systems have been proposed to parallelize decision tree training on CPUs and GPUs [27, 38].

As we report in Section 9, SilvanForge's domain-specific approach achieves significantly better performance than several existing systems while also providing better portability.

Other Systems and Techniques: Ren et al. [47] design an intermediate language and a virtual machine to enable vector execution of decision tree inference. However, this virtual machine is implemented by hand on different target processors. Jo et al. [29] describe code transformations and runtime techniques that vectorize tree-based applications but these optimizations are not specific to decision trees. Inspector-executor systems [34, 41] 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.

[12]It picks the number of rows that each thread block should process based on shared memory capacity.

Code Generation Systems for Other Domains: TVM [19], Tiramisu [16], and Tensor Comprehensions [55] are optimizing compilers for DNNs that can target a variety of processors. Similarly, Halide [45] is a DSL and compiler primarily designed for image processing applications. The concept of separating the computation from the schedule was effectively utilized by Halide and has since been adopted by several other systems [16, 19, 58]. Libraries that compose or generate optimized implementations for BLAS [3, 54, 56] and signal processing [21, 44] have also been developed. However, SilvanForge is the first system that provides state-of-the-art performance across targets by implementing a scheduling language for decision tree inference.

Reductions: CUB [9] and Thrust [11] are libraries that implement reductions on GPUs. However, it is not possible to fuse these functions with other computations as required in SilvanForge. Reddy et al. [46] describe language constructs in Pencil [15] to express reductions and optimize them using the polyhedral framework. Their system does not express the hierarchical nature of reductions and also only targets GPUs. Suriana et al. [51] extend Halide to add support for reductions in the Halide scheduling language and to synthesize reduction operators. De Gonzalo et al. [23] 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 Conclusions

Two trends motivate the need for systems that provide portable performance for ML inference – machine learning is becoming more ubiquitous and hardware is getting more diverse. This paper discussed the challenges in targeting decision tree models to run at peak performance on CPUs and GPUs. To address these, we designed SilvanForge, a schedule-guided, retargetable compiler for decision tree inference. We demonstrated that code generated by SilvanForge is significantly faster than existing systems like XGBoost, RAPIDS FIL, Hummingbird and Tahoe. We obtained such improvements because our scheduling language was able express more combinations of optimization strategies, and our schedule exploration technique was able to quickly find high-performance schedules.

Acknowledgments

We would like to thank our shepherd, Eddie Kohler, and the reviewers for their insightful feedback that significantly improved our paper. We would also like to thank the artifact reviewers for evaluating our artifact. We are grateful to NI India for providing financial support to the first author.

References

[1] [n. d.]. Kaggle AI Report 2023. https://www.kaggle.com/ai-report-2023. Accessed: 2024-04-16.

[2] [n. d.]. Kaggle State of Data Science and Machine Learning 2021. https://www.kaggle.com/kaggle-survey-2021. Accessed: 2022-04-16.

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

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

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

[6] 2019. RAPIDS Forest Inference Library: Prediction at 100 million rows per second. https://medium.com/rapids-ai/rapids-forest-inference-library-prediction-at-100-million-rows-per-second-19558890bc35. Accessed: 2024-04-15.

[7] 2020. Intel Machine Learning Benchmarks. https://github.com/IntelPython/scikit-learn_bench.

[8] 2020. The total cost of ownership of Amazon SageMaker. https://pages.awscloud.com/rs/112-TZM-766/images/Amazon_SageMaker_TCO_uf.pdf.

[9] 2024. CUB: API Reference for CUB. https://docs.nvidia.com/cuda/cub/index.html. Accessed: 2024-04-15.

[10] 2024. RAPIDS: GPU Accelerated Data Science. https://rapids.ai/. Accessed: 2024-04-15.

[11] 2024. Thrust. https://developer.nvidia.com/thrust. Accessed: 2024-04-15.

[12] 2024. XGBoost GPU Support. https://xgboost.readthedocs.io/en/stable/gpu/index.html. Accessed: 2024-04-15.

[13] 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

[14] 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

[15] 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

[16] 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.

[17] Simon Boehm. [n. d.]. *lleaves*. https://github.com/siboehm/lleaves

[18] 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

[19] 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

[20] 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

[21] 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

[22] Wilson W.L. Fung, Ivan Sham, George Yuan, and Tor M. Aamodt. 2007. Dynamic Warp Formation and Scheduling for Efficient GPU Control Flow. In *40th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 2007)*. 407–420. https://doi.org/10.1109/MICRO.2007.30

[23] 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

[24] 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]

[25] Mathieu Guillame-Bert, Sebastian Bruch, Richard Stotz, and Jan Pfeifer. 2023. Yggdrasil Decision Forests: A Fast and Extensible Decision Forests Library. In *Proceedings of the 29th ACM SIGKDD Conference on Knowledge Discovery and Data Mining* (Long Beach, CA, USA) *(KDD '23)*. Association for Computing Machinery, New York, NY, USA, 4068–4077. https://doi.org/10.1145/3580305.3599933

[26] 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

[27] 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.

[28] 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

[29] 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.

[30] 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.

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

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

[33] 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

[34] 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

[35] 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

[36] 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

[37] 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

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

[39] 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

[40] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. 2019. PyTorch: An Imperative Style, High-Performance Deep Learning Library. In *Advances in Neural Information Processing Systems 32*. Curran Associates, Inc., 8024–8035. http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf

[41] 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

[42] 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

[43] 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

[44] 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

[45] 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

[46] 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

[47] 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

[48] Charitha Saumya, Kirshanthan Sundararajah, and Milind Kulkarni. 2022. DARM: Control-Flow Melding for SIMT Thread Divergence Reduction. In *2022 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. 1–13. https://doi.org/10.1109/CGO53902.2022.9741285

[49] 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

[50] 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

[51] 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.

[52] 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

[53] Jan Van Lunteren. 2023. Accelerating Decision-Tree-Based Inference Through Adaptive Parallelization. In *2023 32nd International Conference on Parallel Architectures and Compilation Techniques (PACT)*. 176–186. https://doi.org/10.1109/PACT58117.2023.00023

[54] 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

[55] 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

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

[57] 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* (United Kingdom) *(EuroSys '21)*.

ACM, NY, USA, 426–440. https://doi.org/10.1145/3447786.3456251

[58] 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