Taming Model Serving Complexity, Performance and Cost: A Compilation to Tensor Computations Approach

Supun Nakandala[m,c*], Karla Saur[m], Gyeong-In Yu[m,s,*], Konstantinos Karanasos[m],
Carlo Curino[m], Markus Weimer[m], Matteo Interlandi[m]
[m]Microsoft, [c]UCSD, [s]SNU
{<name>.<surname>}@microsoft.com,
snakanda@eng.ucsd.edu, gyeongin@snu.ac.kr

ABSTRACT

Machine Learning (ML) adoption in the enterprise requires simpler and more efficient software infrastructure—the bespoke solutions typical in large web companies are simply untenable. Model scoring, the process of obtaining prediction from a trained model over new data, is a primary contributor to infrastructure complexity and cost, as models are trained once but used many times.

In this paper we propose HUMMINGBIRD, a novel approach to model scoring, which compiles featurization operators and traditional ML models (e.g., decision trees) into a small set of tensor operations. This approach inherently reduces infrastructure complexity and directly leverages existing investments in Neural Networks' compilers and runtimes to generate efficient computations for both CPU and hardware accelerators. Our performance results are surprising: despite replacing sparse computations (e.g., tree traversals) with dense ones (tensor operations), HUMMINGBIRD is competitive and even outperforms (by up to 3×) hand-crafted kernels on micro-benchmarks, while enabling seamless end-to-end acceleration (with a speedup of up to 1200×) of ML pipelines.

PVLDB Reference Format:
Supun Nakandala, Karla Saur, Gyeong-In Yu, Konstantinos Karanasos, Carlo Curino, Markus Weimer, and Matteo Interlandi. Taming Model Serving Complexity, Performance and Cost: A Compilation to Tensor Computations Approach. *PVLDB*, 00(0): xxx-yyy, 2020.
DOI: https://doi.org/TBD

1. INTRODUCTION

Enterprises increasingly look to Machine Learning (ML) to help solve business challenges that escape imperative programming and analytical querying—examples include predictive maintenance, customer churn prediction and supply-chain optimizations [36]. To do so, they typically turn to technologies now broadly termed *"traditional ML"*, a term coined to contrast them with Deep Neural Networks (DNNs). A recent analysis by Amazon Web Services found that 50 to 95% of all machine learning applications in an organization are based on traditional ML [29]. Our own analysis [57] of $6M$ notebooks in public GitHub repositories, as well as Microsoft's internal use of ML.NET [27], paints a similar picture: NumPy [62], Matplotlib [10], Pandas [51] and scikit-learn [55] are the four most used libraries. All four provide functions for traditional ML. As a point of comparison with DNN frameworks, scikit-learn is roughly $5\times$ more prominent than PyTorch [54] and TensorFlow [12] combined, and is growing faster than both.

When it comes to owning and operating ML solutions, enterprises differ from early adopters in their focus on long-term costs of ownership and amortized return on investments [60]. As such, enterprises are highly sensitive to: (1) complexity, (2) performance, and (3) overall operational efficiency of their software infrastructure [13]. In each of those regards, *model scoring* (the process of presenting a trained model with new data to get a prediction) is a key driving factor. Regarding complexity of software infrastructure, models are scored in a variety of environments, and thus scoring dominates the complexity due to portability, maintainability, deployment, and monitoring concerns. With respect to performance, model scoring is often in the critical path of interactive or analytical enterprise applications. Hence, latency and throughput for scoring models are important concerns for enterprises. Finally, looking at total cost of ownership of a data science solution, *model scoring is responsible for 45-65% of all costs*. This is true even when considering simple cloud-native solutions, including support personnel costs, and model building/exploration costs [29]. In short, the reason to focus on model scoring is simple: most models are trained infrequently, in an offline fashion in resource-rich/uniform cloud environments, but they are scored many times and deployed in performance-critical, diverse environments (e.g., scale-out batch or interactive serving, personal computers, mobile and IoT devices).

The Underlying Challenge. Now that we established model scoring as a primary source of complexity and cost for ML deployments, we zoom in to search for core underlying challenges. We begin by observing that the output of the iterative process of designing and training traditional ML models is not *just* a trained model but a *predictive pipeline:* A DAG of operators that typically include: (1) *featurizers*, which could be either stateless imperative code (e.g., string tokenization) or data transformations fit to the data (e.g., min/max normalization); and (2) *models*, commonly decision tree ensembles or (generalized) linear models, fit to the data. Typical pipelines contain up to tens of operators out of a set of multiple hundreds [57]. It is important to notice that a *prediction can only be rendered using the whole pipeline* as it was fit to the training data. Moreover, today's featurizer and model implementations are not expressed in a shared logical abstraction, but rather in an ad-hoc fashion using programming languages such as R, Python, Java, C++ or C#. This hints to the core problem with today's approaches to model scoring: *the combinatorial explosion of supporting many operators (and frameworks) across multiple target environments*.

*The work was done while the author was at Microsoft.

Proceedings of the VLDB Endowment, Vol. 00, No. 0
ISSN 2150-8097.
DOI: https://doi.org/TBD

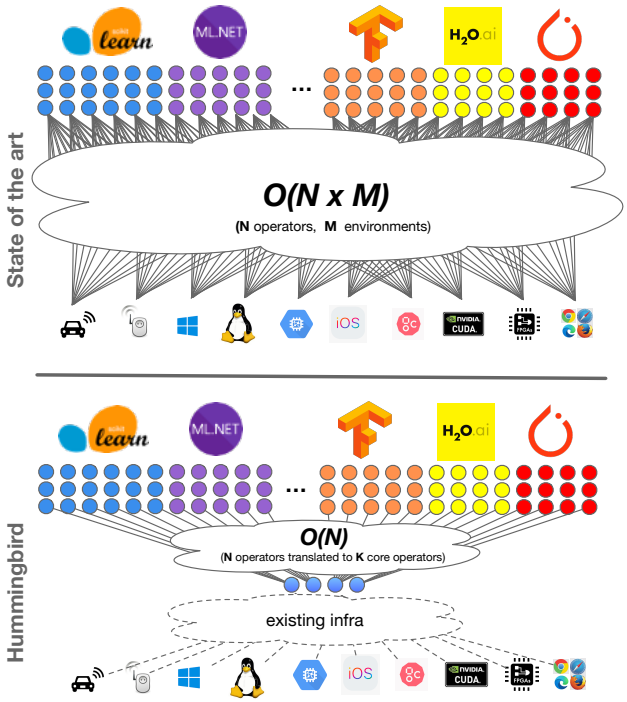


Figure 1: Model scoring software complexity: state-of-the-art (top) VS HUMMINGBIRD (bottom).

Figure 1 highlights this visually by showing how existing solutions lead to an $O(N \times M)$ explosion to support N operators from various ML frameworks against M deployment environments (e.g., how to run a scikit-learn model on an embedded device?). Furthermore, in [57] we show that the number of libraries used in data science (a metric correlated to N) is still increasing—a roughly $4\times$ growth in the last 2 years. Our expectation is that M is also destined to grow as ML is applied more and more widely across a broad range of enterprise applications and hardware (e.g., [14, 1, 38, 40, 6]). From the vantage point of implementing runtimes for model scoring, this is a daunting proposition. In fact, we argue that any brute-force approach tackling all combinations directly would dilute engineering focus leading to costly and less optimized solutions. It is paramount to bypass the $N \times M$ explosion somehow.

Our solution. HUMMINGBIRD leverages compiler/optimizer techniques to translate a broad set of traditional ML operators into a small set of K core operators, reducing the cost to $O(N) + O(K \times M)$ as shown in Figure 1. This key intuition is also behind the design of the ONNX model format [22], and its various runtimes [5]. With HUMMINGBIRD we take one further bold step, and demonstrate that this set of core operators can be reduced to tensor computations and therefore be executed over DNN frameworks. This allows us to piggyback on existing (massive) investments in DNN compilers/runtimes/specialized-hardware to cover the "running K operators across M environments" part of our challenge, reducing the infrastructure complexity to support traditional ML to just $O(N)$ operator translations. Additionally, this cost can be absorbed by each of the input frameworks, as no central coordination or standardization is necessary.[1] This translates to reduced infrastructure complexity, improved resource efficiency, and better portability.

Contributions. While the journey to a hardened product is ongoing, in this paper we answer three fundamental research questions:

1. Can traditional ML operators (both linear algebra-based such as linear models, and algorithmic ones such as decision trees) be translated to tensor computations?
2. Can the resulting (dense) computations in tensor space be competitive with the (sparse) imperative alternatives we get as input (e.g., traversing a tree)?
3. Can HUMMINGBIRD help in reducing software complexity and improving model portability?

The results are surprising as we answer all the above questions positively. To do so, we: (1) port thousands of benchmark predictive pipelines to two DNN backends; (2) show that we can seamlessly leverage hardware accelerators and deliver speed-ups of up to $3\times$ against hand-crafted GPU kernels, and up to $1200\times$ against CPU state-of-the-art—on average this could translate to 25 to $50\times$ lowered scoring infrastructure costs for batch scenarios; and (3) qualitatively confirm improvements in software complexity and portability by showing that we can run scikit-learn predictive pipelines across CPUs, GPUs (and IPUs [14]) with less than $5K$ lines of code.

Organization. The remainder of this paper is organized as follows: Section 2 contains ML and DNN background, while Section 3 contains an overview of the HUMMINGBIRD system. Section 4 describes some operator implementations and optimizations; the experimental evaluation is in Section 5. The paper ends with related work and conclusions, respectively in Sections 6 and 7.

2. BACKGROUND AND CHALLENGES

We first provide necessary background on traditional ML and DNNs. We then explain the challenges of compiling traditional ML operators and predictive pipelines into tensor computations.

2.1 Traditional ML and DNNs

Traditional predictive pipelines. The result of the data science workflow over traditional ML are predictive pipelines, i.e., Directed Acyclic Graphs (DAGs) of operators such as trained models, preprocessors, featurizers, missing-value imputers. The process of presenting a trained predictive pipeline with new data to obtain a prediction is referred to in literature interchangeably as: model scoring/inference/serving, pipeline evaluation, or prediction serving. We favor model scoring in our writing, but at times use some of the above alternatives.

Packaging a trained pipeline into a single artifact is common practice [27]. These artifacts are then embedded inside host applications, or containerized and deployed in the cloud to perform model scoring [56, 33]. ML.NET (.NET-based), scikit-learn (Python-based), and H_2O (Java-based) are popular toolkits to train and generate pipelines. However, it is important to note that they are primarily optimized for training, not for scoring. Scoring predictive pipelines is challenging, as their operators are implemented in imperative code, and do not follow a shared logical or physical abstraction. Supporting every operator in all target environments requires huge effort: this is why these frameworks have typically limited portability.

DNNs. Deep Neural Networks (DNNs) are a family of ML models that are based on artificial neurons [37]. They take raw features as input and perform a series of transformation operations. Unlike traditional ML where the ML transformations are complex and diverse, transformations in DNNs are drawn from a small set of simple tensor transformations (e.g., generic matrix multiplication, element-wise ops). Hence, an entire DNN can be represented using a DAG of tensor operators. In recent years, DNNs have been extremely successful in vision and natural language processing tasks [45, 35]. Common frameworks used to author and train DNNs are TensorFlow [12], PyTorch [54], CNTK [9], and MXNet [11]. While these

[1] As we work on open-sourcing HUMMINGBIRD we are in fact engaging with ONNX-Runtime, TVM, ML.NET and scikit-learn OSS communities.

frameworks can also be used to perform model scoring, next we discuss systems specifically designed for that.

Runtimes for DNN Model Scoring. To cater to the demand for DNN model inference, a new class of systems has emerged. ONNX Runtime (ORT) [4], TorchScript [7], and TVM [31] are popular examples of such systems. These capitalize on the relative computational simplicity of neural networks: they accept a DAG of tensor operations as input, which they execute by implementing a small set of highly optimized operator kernels on multiple hardwares. Focusing on just the prediction serving scenario also enables these systems to perform additional inference-specific optimizations, which are not applicable for training. HUMMINGBIRD is currently compatible with all such systems, but we focus our experiments on PyTorch/TorchScript and TVM as runtimes backends.

2.2 Challenges

HUMMINGBIRD combines the strength of traditional ML pipelines on structured data [48] with the computational and operational simplicity of DNN runtimes for model scoring. To do so, it relies on a simple yet key observation: once a model is trained, it can be represented as a *prediction function* transforming input features into a prediction score (e.g., 0 or 1 for binary classification), regardless of the training algorithm used. The same observation naturally applies to featurizers fit to the data. Therefore, HUMMINGBIRD only needs to compile the prediction functions (not the training logic) for each operator in a pipeline into tensor computations and stitch them appropriately. Towards this goal, we identify two key challenges.

Challenge 1: *How can we map traditional predictive pipelines into tensor computations?* Pipelines are generally composed of operators (with predictive functions) of two classes: *algebraic* (e.g., scalers or linear models), and *algorithmic* (e.g., one-hot encoder and tree-based models). While translating algebraic operators into tensor computations is straightforward, the key challenge for HUMMINGBIRD is the translation of algorithmic operators. Algorithmic operators perform arbitrary *data accesses and control flow decisions*. For example, in a decision tree ensemble potentially every tree is different from each other, not only with respect to the structure but also the decision variables and the threshold values. Conversely, tensor operators (such as matrix multiplication, element-wise operations) perform *bulk operations* over the entire set of input elements.

Challenge 2: *How can we achieve efficient execution for tensor-compiled traditional ML operators?* The ability to compile predictive pipelines into DAGs of tensor operations does not imply adequate performance of the resulting DAGs. In fact, common wisdom would suggest the opposite: even though tensor runtimes naturally support execution on hardware accelerators (e.g., GPUs, IPUs, TPUs [40], ASICs), tree-based methods and commonly used data transformations are well known to be difficult to accelerate [32] even using custom-developed implementations.

3. SYSTEM OVERVIEW

In this section we explain our high-level approach to overcome the challenges outlined in Section 2.2, and present HUMMINGBIRD's architecture and some implementation details. We conclude this section by explaining the assumptions and limitations of our work.

3.1 High-level Approach

In HUMMINGBIRD, we cast algorithmic operators into tensor computations by *introducing a degree of redundancy*, which includes both *computational redundancy* and *storage redundancy*. With computational redundancy, we perform computations for more than what is needed for optimal execution. With storage redundancy we create data structures that store more than what we actually need. These redundancies enable us to transform the arbitrary data accesses and control flow of the original algorithmic operators into bulk operations compilable into tensor computations.

Based on the level of redundancy introduced, different compilation strategies are optimal. Therefore, different tensor implementations exist for a given traditional ML operator. We will discuss the compilation strategies for representative operators in Section 4. The tensor implementation to be used is informed by model characteristics (e.g., tree-structure for tree-based models, or sparsity for linear models) and runtime statistics (e.g., batch size of the inputs). *Heuristics at the operator level*, *runtime-independent optimizations at the pipeline level*, and *runtime-specific optimizations at the execution level* enable HUMMINGBIRD to further improve predictive pipelines performance end-to-end. The dichotomy between runtime-independent and -specific optimizations allow us to both (1) apply optimizations unique to traditional ML, and not captured by the DNN runtimes; and (2) exploit DNN runtime optimizations once the traditional ML is lowered into tensor computations. Finally, by compiling traditional predictive pipelines into tensor computations, HUMMINGBIRD is able to run end-to-end pipelines on all the hardware platforms supported by the target tensor runtimes.

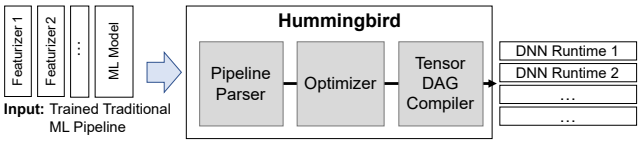


Figure 2: High-level architecture of HUMMINGBIRD.

3.2 System Architecture and Implementation

The high-level architecture of HUMMINGBIRD is shown in Figure 2. HUMMINGBIRD has four main components: (1) *Pipeline Parser*, (2) *Optimizer*, and (3) *Tensor DAG Compiler*. Given a predictive pipeline and a set of input parameters (i.e., batch size, input type, target DNN runtime, target hardware device), the Pipeline Parser generates an in-memory Intermediate Representation (IR) object encoding each operator in the pipeline and related input/output dependencies. The Optimizer then runs optimization passes over the IR to produce a potentially modified IR. Furthermore, if there is more than one potential compilation strategy for an operator, the Optimizer annotates the IR with the compilation strategy to be used for that specific operator given the input parameters. Afterwards, the Tensor DAG Compiler picks the optimized IR object and compiles it into tensor operations following the target DNN runtime format. The runtime-specific optimizations are triggered at this level. Finally, the model is exported in the native format of the target runtime for model prediction.

The current version of HUMMINGBIRD is implemented as an extension of ONNXMLTools [16], and supports compiling scikit-learn pipelines into PyTorch/TorchScript and TVM output formats. HUMMINGBIRD's IR and Pipeline Parser are based on an extension of the IR and parser used in the skl2onnx converter [25]. HUMMINGBIRD currently supports over 40 scikit-learn operators (listed in Table 1), which are compiled into a small set of tensor operators (listed in Table 2 for the PyTorch runtime). Adding support for more ML operators and other ML tools (e.g., ML.NET [27], H_2O [8]) is currently an ongoing work.

3.3 Assumptions and Limitations

In this paper, we make a few simplifying assumptions. First, we assume that predictive pipelines are "pure" and do not contain arbitrary user-defined operators. There has been recent work [58] on compiling imperative UDFs into relation algebra, and we plan

Table 1: Scikit-learn operators currently supported in HUMMINGBIRD.

Supported ML Models
LogisticRegression, SVC, NuSVC, LinearSVC, SGDClassifier, LogisticRegressionCV, DecisionTreeClassifier/Regression, RandomForestClassifier/Regression, ExtraTreesClassifier, GradientBoostingClassifier/Regression, XGBClassifier/Regression, LGBMClassifier/Regression, HistGradientBoostingClassifier, MLPClassifier, BernoulliNB, GaussianNB, MultinomialNB
Supported Featurizers
SelectKBest, VarianceThreshold, SelectPercentile, PCA, KernelPCA, TruncatedSVD, FastICA, SimpleImputer, Imputer, MissingIndicator, ColumnTransformer, RobustScaler, MaxAbsScaler, MinMaxScaler, StandardScaler, Binarizer, KBinsDiscretizer, Normalizer, PolynomialFeatures, OneHotEncoder, LabelEncoder, FeatureHasher

Table 2: PyTorch tensor operators used by the Tensor DAG Compiler.

`matmul, add, mul, div, lt, le, eq, gt, ge, &, |, ≪, ≫, bitwise_xor, gather, index_select, cat, reshape, cast, abs, pow, exp, arxmax, max, sum, relu, tanh, sigmoid, logsumexp, isnan, where`

to make use of such techniques in HUMMINGBIRD in the future. Second, we do not support sparse data well. We found that current support for sparse computations on DNN runtimes is primitive and not well optimized. We expect advances in DNN frameworks to improve on this aspect—TACO [43] is a notable such example. Finally, although we support string operators, we currently do not support text feature extraction (e.g., `TfidfVectorizer`). The problem in this case is twofold: (1) compiling regex-based tokenizers into tensor computations is not trivial, and (2) representing arbitrarily long text documents in tensors remains an open challenge in general.

4. COMPILATION

HUMMINGBIRD currently supports compiling many representative algorithmic operators into tensor computations. Given their popularity [57], in Sections 4.1 and 4.2 we explain our approach for tree-based models. Section 4.3 gives a summary of other techniques that we use for both algorithmic and arithmetic operators.

Table 3: Notation used in Section 4.1

Symbol	Description
N, I, L, F, C	Ordered lists with all nodes, internal nodes, leaf nodes, features, and classes, respectively.
$X \in \mathsf{R}^{n \times \lvert F \rvert}$	Input records (n is the number of records).
$A \in \mathsf{R}^{\lvert F \rvert \times \lvert I \rvert}$	$A_{i;j} = \begin{cases} 1, & I_j \text{ evaluates } F_i \\ 0, & \text{Otherwise} \end{cases}$
$B \in \mathsf{R}^{\lvert I \rvert}$	$B_i = \texttt{ThresholdValue}(I_i)$
$C \in \mathsf{R}^{\lvert I \rvert \times \lvert L \rvert}$	$C_{i;j} = \begin{cases} -1, & L_j \in \texttt{RightSubTree}(I_i) \\ 1, & L_j \in \texttt{LeftSubTree}(I_i) \\ 0, & \text{Otherwise} \end{cases}$
$D \in \mathsf{R}^{\lvert L \rvert}$	$D_k = \sum_{k \in L \xrightarrow{path} \texttt{Root}} \mathbf{1}(k == \texttt{LeftChild}(\texttt{Parent}(k)))$
$E \in \mathsf{R}^{\lvert L \rvert \times \lvert C \rvert}$	$E_{i;j} = \begin{cases} 1, & L_i \xrightarrow{map\ to} C_j \\ 0, & \text{Otherwise} \end{cases}$

Strategy	Memory	Runtime
GEMM	$O(\lvert F \rvert \lvert N \rvert + \lvert N \rvert^2 + \lvert C \rvert \lvert N \rvert)$	$O(\lvert F \rvert \lvert N \rvert + \lvert N \rvert^2 + \lvert C \rvert \lvert N \rvert)$
TreeTraversal	$O(\lvert N \rvert)$	$O(\lvert N \rvert)$
PerfectTreeTraversal	$O(2^{\lvert N \rvert})$	$O(N)$

Table 4: Worst-case memory and runtime analysis of different tree compilation strategies, assuming the number of input records and number of trees are fixed. The notation is explained in Table 3.

4.1 Compiling Tree-based Models

HUMMINGBIRD has three different strategies for compiling tree-based models for classification tasks. Strategies differ based on the degree of redundancy introduced. In Section 4.2, we explain how HUMMINGBIRD picks the best strategy. Table 3 explains the notation used in this section. We summarize the worst-case runtime and memory footprints of each strategy in Table 4. HUMMINGBIRD currently supports only trees built over numerical values: support for missing and categorical values is currently under development. For the sake of presentation, we assume all decision nodes perform $<$ comparisons.

Strategy 1: `GEMM`. We cast the evaluation of a tree as a series of three GEneric Matrix Multiplication (GEMM) operations interleaved by two element-wise logical operations. Given a tree, we create five tensors which collectively capture the tree structure: A, B, C, D, and E. A captures the relationship between input features and internal nodes. B is set to the threshold value of each internal node. For any leaf node and internal node pair, C captures whether the internal node is a parent of that internal node, and if so, whether it is in the left or right sub-tree. D captures the count of the internal nodes in the path from a leaf node to the tree root, for which the internal node is the left child of its parent. Finally, E captures the mapping between leaf nodes and the class labels. Given these tensors, Algorithm 1 presents how we perform tree scoring for a batch of input records X. A graphical representation of an execution of the GEMM strategy is depicted in Figure 3.

Algorithm 1 GEMM Strategy (Notation explained in Table 3)

Input : $X \in \mathsf{R}^{n \times \lvert F \rvert}$, Input records
Output : $R \in \{0,1\}^{n \times \lvert C \rvert}$, Predicted class labels

```
/* Evaluate all internal nodes                          */
T ← GEMM(X, A)                          // T ∈ R^{n×|I|}
T ← T < B                               // T ∈ R^{n×|I|}

/* Find the leaf node which gets selected               */
T ← GEMM(T, C)                          // T ∈ R^{n×|L|}
T ← T == D                              // T ∈ R^{n×|L|}

/* Map selected leaf node to class label                */
R ← GEMM(T, E)                          // R ∈ R^{n×|C|}
```

The first GEMM is used to match each input features with the internal node(s) using it. The following $<$ operations is used to evaluate all the internal decision nodes and produces a tensor of 0s and 1s based on the false/true outcome of the conditions. The second GEMM operation generates an encoding for the path composed by the true internal nodes, while the successive $==$ operation returns the leaf node selected by the encoded path. Note that logical operators will *broadcast* [19] B and D tensors to match the dimensions of the other operand for performing element-wise operations. Finally, the third GEMM operation maps the selected leaf node to the class label.

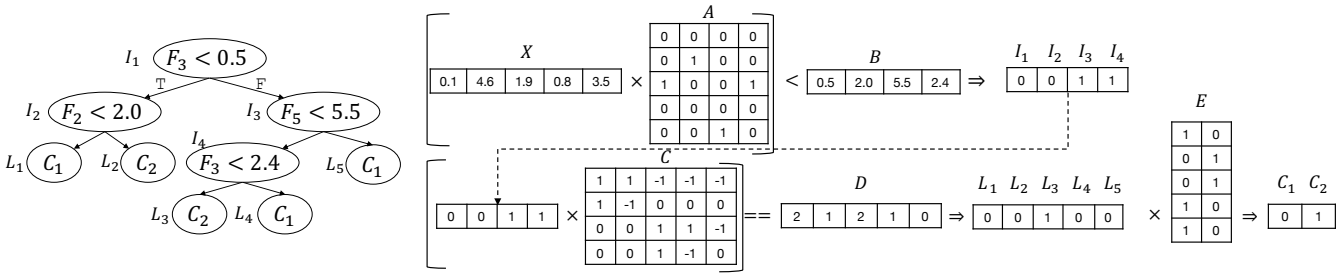


Figure 3: Compiling an example decision tree using the GEMM strategy (algorithm 1).

While we explained this strategy in the context of a single tree and a classification task, it is easily extended to support tree ensembles and regression tasks too. For tree ensembles, we create the above 2-dimensional tensors for each tree and batch them together. As the number of leaf nodes and internal nodes can vary among trees, we pick the maximum number of leaf nodes and internal nodes for any tree as the tensor dimensions and pad the smaller tensor slices with zeros. During scoring, we invoke the batched variants of GEMM and logical operations and perform a final ReduceMean operation over the batched dimension to generate the ensemble output. For regression tasks, we initialize E with label values.

Table 5: Additional notation used in Strategy 2: TreeTraversal

Symbol	Description
$N_L \in \mathsf{Z}^{\lvert N\rvert}$	$N_{L_i} = \begin{cases} \texttt{LeftChild}(N_i), N_i \in I \\ i, \text{Otherwise} \end{cases}$
$N_R \in \mathsf{Z}^{\lvert N\rvert}$	$N_{R_i} = \begin{cases} \texttt{RightChild}(N_i), N_i \in I \\ i, \text{Otherwise} \end{cases}$
$N_F \in \mathsf{Z}^{\lvert N\rvert}$	$N_{F_i} = \begin{cases} k, (N_i \in I) \wedge (N_i \text{ evaluates } F_k) \\ 1, \text{Otherwise} \end{cases}$
$N_T \in \mathsf{R}^{\lvert N\rvert}$	$N_{T_i} = \begin{cases} \texttt{ThresholdValue}(N_i), N_i \in I \\ 0, \text{Otherwise} \end{cases}$
$N_C \in \mathsf{Z}^{\lvert N\rvert \times \lvert C\rvert}$	$N_{C_{i,k}} = \begin{cases} 1, (N_i \in L) \wedge (N_i \xrightarrow{map\ to} C_k) \\ 0, \text{Otherwise} \end{cases}$

Strategy 2: TreeTraversal. In the GEMM strategy, we incorporated a high-degree of computational redundancy by evaluating all internal nodes and leaf nodes when only a few of them actually need to be evaluated. Here, we try to reduce the computational redundancy by mimicking the typical tree traversal—but implemented using tensor operations. In this strategy, the tree structure is captured by five tensors: N_L, N_R, N_F, N_T, and N_C. We formally define these tensors in Table 5. The same column index (last dimension) across all tensors corresponds to the same tree node. N_L and N_R capture the indices of the left and right nodes for a given node. If the node is a leaf node, we set these to the index of the given node. Similarly, N_F and N_T capture the feature index and threshold value for each node, respectively. For leaf nodes, we set N_F to 1 and N_T to 0. Finally, N_C captures the class label of each leaf node. For internal nodes this can be any value; we set it to 0.

Given these tensors, Algorithm 2 presents how we perform scoring for a batch of input records X. We use Gather and Where operations which can be used to perform index-based slicing and conditional value selection. These operators are available in most modern tensor runtimes. We first initialize an index tensor T_I corresponding to all records in X, which points to the root node. Using T_I, we Gather the corresponding feature indices and use them

Algorithm 2 TreeTraversal Strategy (Notation in Tables 5)

Input : $X \in \mathsf{R}^{n \times \lvert F\rvert}$, Input records
Output : $R \in \{0,1\}^{n \times \lvert C\rvert}$, Predicted class labels

```
/* Initialize all records to point to k,
   with k the index of Root node.          */
T_I ← {k}^n                                // T_I ∈ Z^n
for i ← 1 to TREE_DEPTH do
    /* Find the index of the feature
       evaluated by the current node. Then
       find its value.                      */
    T_F ←Gather(N_F, T_I)                  // T_F ∈ Z^n
    T_V ←Gather(X, T_I)                    // T_V ∈ R^n
    /* Find the threshold, left child and
       right child                          */
    T_T ←Gather(N_T, T_I)                  // T_T ∈ R^n
    T_L ←Gather(N_L, T_I)                  // T_L ∈ Z^n
    T_R ←Gather(N_R, T_I)                  // T_R ∈ Z^n
    /* Perform logical evaluation. If true
       pick from T_L; else from T_R.        */
    T_I ←Where(T_V < T_T, T_L, T_R)        // I ∈ Z^n
end
/* Find label for each leaf node           */
R ←Gather(N_C, T_I)                        // R ∈ Z^n
```

to Gather the corresponding feature values from X. Similarly, we also Gather left node indices, right node indices, and node thresholds. Using these gathered tensors, we then invoke a Where operation which checks for the tree node decisions. Based on the evaluation, for each record the Where operator either returns the left child index or right child index. To perform full tree scoring, the above steps have to be repeated until we reach a leaf node for all records in X. We exploit the fact that (1) TREE_DEPTH is a known property of the input model at compilation time, and (2) all leaf nodes are at a depth $\leq$ TREE_DEPTH, to iterate for that fixed number of iterations to ensure that all records have found their corresponding leaf node. Tensors are created in such a way that if one of the indices reaches a leaf node before running for TREE_DEPTH iterations, the same class label will keep getting selected. At compile time, we unroll all iterations and remove the for loop to improve efficiency. For ensembles, we create tensors for each tree and batch them together. However, between trees the number of nodes and dimensions may differ, so we use the maximum node count for any tree as the dimension and pad the remaining elements with zeros.

Strategy 3: PerfectTreeTraversal. Similar to the previous strategy, this strategy also mimics the tree traversal. However, here we assume the tree is a *perfect binary tree*. In a perfect binary tree, all internal nodes have exactly two children and all leaf nodes are at the same depth level. Assume we are given a non-perfect binary tree with a TREE_DEPTH of D, and L_k is a leaf node which is at a depth of $D_k < D$. To push L_k to a depth D, we replace L_k with a perfect sub-tree of depth $D - D_k$ and map all the leaf nodes of the sub-tree

to C_k: the label of the original leaf node. The decision nodes in the introduced sub-tree are free to perform arbitrary comparisons as the outcome is the same along any path. By pushing all leaf nodes at depth $< D$ to a depth of D, we transform the original tree to a perfect tree with the same functionality.

Table 6: Additional notation used in Strategy 3: PerfectTreeTraversal

Symbol	Description
$I' \in \mathsf{Z}^{2^D - 1}, L' \in \mathsf{Z}^{2^D}$	Internal and leaf nodes of the transformed perfect tree ordered by level.
$N'_F \in \mathsf{Z}^{\lvert I' \rvert}$	$N'_{F_i} = k \iff I'_i$ evaluates F_k
$N'_T \in \mathsf{R}^{\lvert I' \rvert}$	$N'_{T_i} = \texttt{ThresholdValue}(I'_i)$
$N'_C \in \mathsf{Z}^{\lvert L' \rvert \times \lvert C \rvert}$	$N'_{C_{i,k}} = \begin{cases} 1, N_i \xrightarrow{map\ to} C_k \\ 0, \text{Otherwise} \end{cases}$

Working on perfect trees enables us to get rid of N_L and N_R tensors as we can now calculate them analytically, which also reduces memory lookup overheads during scoring. Thus we create only three tensors to capture the tree structure: N'_F, N'_T, and N'_C. We explain these tensors formally in Table 6. They capture the same information as N_F, N_T, N_C but have different dimensions and have a strict condition on the node order. Both N'_F and N'_T have 2^{D-1} elements and the values correspond to internal nodes generated by level order tree traversal. N'_C has 2^D elements with each corresponding to an actual leaf node from left to right order.

Algorithm 3 PerfectTreeTraversal Strategy (Notation in Tables 6)

Input : $X \in \mathsf{R}^{n \times \lvert F \rvert}$, Input records
Output : $R \in \{0,1\}^{n \times \lvert C \rvert}$, Predicted class labels

```
/* Initialize all records to point to the
   root node.                                   */
T_I ← {1}^n                                     // T_I ∈ Z^n
for i ← 1 to TREE_DEPTH do
    /* Find the index of the feature
       evaluated by the current node. Then
       find its value.                          */
    T_F ←Gather(N_F, T_I)                       // T_F ∈ Z^n
    T_V ←Gather(X, T_I)                         // T_V ∈ R^n
    /* Find the threshold                       */
    T_T ←Gather(N_T, T_I)                       // T_T ∈ R^n
    /* Perform logical evaluation. If true
       pick left child; else right child.       */
    T_I ← 2 × T_I + Where(T_V < T_T, 0, 1)      // I ∈ Z^n
end
/* Find label for each leaf node               */
R ←Gather(N'_C, T_I)                            // R ∈ Z^n
```

Given these tensors, in Algorithm 3 we present how PerfectTreeTraversal works. From a high-level point of view, it is very similar to the TreeTraversal strategy with only a few changes. First, the index tensor T_I is initialized to all ones as the root node is always the first node. Second, we get rid of finding the left index and right index of a node and using them in the `Where` operation. Instead, the `Where` operation returns 0 for true case and 1 for the false case. By adding this to $2 \times T_I$ we get the index of the child for the next iteration. For ensembles, we use the maximum `TREE_DEPTH` of any tree as D for transforming trees to perfect trees. We create tensors separate for each tree and batch them together for N'_C. But for N'_F and N'_T instead of batching, we interleave them together in some order such that values corresponding to level i for all trees appear before values corresponding to level $i+1$ of any tree. This enables better memory coalescing and improves performance.

4.2 Heuristics-based Strategy Selection

For a given classical ML operator, there can be more than one compilation strategy available. In the previous section we just explained three such strategies for tree-based models. In practice, no strategy consistently dominates the others, but each is preferable in different situations based on the input and model structure. For instance, the `GEMM` strategy gets significantly inefficient as the size of the decision trees gets bigger because of the large number of redundant computations. This strategy performs $O(2^D)$ (D is the height of the tree) computations whereas the original algorithmic operator needs to perform only $O(D)$ comparisons. Nevertheless, with small batch sizes or a large number of smaller trees, this strategy can be actually performance-wise optimal on modern hardware, where `GEMM` operations can run highly efficiently. With large batch sizes and taller trees, TreeTraversal techniques typically outperform the `GEMM` strategy and PerfectTreeTraversal is slightly faster than vanilla TreeTraversal due to the reduced number of index lookups and better coalesced memory accesses. But if the trees are too deep, we cannot implement PerfectTreeTraversal because the $O(2^D)$ memory footprint of the associated data structures will be prohibitive. In such cases, we resort to TreeTraversal.

The exact crossover point where `GEMM` strategy outperforms TreeTraversal strategies is determined by the characteristics of the tree model (e.g., number of trees, maximum depth of the trees), runtime statistics (e.g., batch size), and the underlying hardware (e.g., CPUs, GPUs). For instance, from our experiments (see Figure 7) we found that on CPUs the `GEMM` strategy performs better for shallow trees (≤ 3 on CPU, ≤ 10 on GPU) or for scoring with smaller batch sizes. For tall trees, using PerfectTreeTraversal when $D \leq 10$ gave a reasonable trade-off between memory footprint and runtime, which leaves vanilla TreeTraversal the only option for very tall trees ($D > 10$). These heuristics are currently hard-coded in HUMMINGBIRD, but can be overridden by the user if necessary.

Recall that, in addition to heuristics, our approach also leverages runtime-independent optimizations at the Optimizer level and runtime-specific optimizations at the Tensor DAG Compiler level. Due to space constraints, we discuss runtime-independent optimizations in the technical report [61]. We refer the interested readers to [7, 31] for more details on runtime-dependent optimizations.

4.3 Summary of Other Techniques

Next, we discuss a few other techniques that we use across many ML operators to efficiently compile them into tensor computations.

Exploiting Automatic Broadcasting. Broadcasting [19] is the process of making two tensors shape compatible for element-wise operations. Two tensors are said to be shape compatible if each dimension pair is the same or one of them is 1. At execution time, tensor operations implicitly repeat the size 1 dimensions to match the size of the other tensor, without allocating memory for these expansions. In HUMMINGBIRD, we heavily use this feature to execute some computation over multiple inputs. For example, consider performing an one-hot encoding operation over column $X_i \in \mathsf{R}^n$ with a vocabulary $V \in \mathsf{Z}^m$. In order to implement this using tensor computations, we `Reshape` X_i to $[n, 1]$ and V to $[1, m]$ and calculate $R = \texttt{Equal}(X, V)$, $R \in \{0,1\}^{n \times m}$. The `Reshape` operations are for free because they only modify the metadata of the original tensor. However, this approach performs redundant comparisons as

it essentially checks the feature values from all records against all vocabulary values, which is different from an imperative approach.

Minimize Operator Invocations. Given two approaches to implement an ML operator we found that often times picking the one which invokes fewer operators outperforms the other—even if it performs extra computations. Consider a featurizer that generates feature interactions. Given an input $X \in \mathsf{R}^{n \times d}$, with $d = |F|$, it generates a transformed output $R \in \mathsf{R}^{n \times \frac{d\ (d+1)}{2}}$ with $R_i = [X_{i;1}^2, ..., X_{i;d}^2, X_{i;1}X_{i;2}, ...X_{i;d-1}X_{i;d}]$. One way to implement this operator is to compute each new feature separately by first `gather`ing the corresponding input feature columns, perform an element-wise `mul`tiplication, and con`cat`enate all new features. However, this approach requires performing $d^2 + d + 1$ operations and hence is highly inefficient due to high operator scheduling overheads. Alternatively, one could implement the same operator as follows. First, `reshape` X into $X' \in \mathsf{R}^{n \times d \times 1}$ and $X'' \in \mathsf{R}^{n \times 1 \times d}$. Then perform a batched `GEMM` using these inputs, which will create $R' \in \mathsf{R}^{n \times d \times d}$. Finally, `Reshape` R' to $R'' \in \mathsf{R}^{n \times d^2}$. Notice that each row in R'' has all the values of the corresponding row in R, but in a different order. It also has some redundant values due to commutativity of multiplication (i.e., $x_i x_j = x_j x_i$). Hence, we perform a final `Gather` to extract the features in the required order, and generate R. This approach in fact performs roughly twice the computations than the previous approach and also increases the peak memory footprint roughly by a factor of two. However, it enables us to implement the feature interaction operator in just two tensor operations which runs highly efficiently on tensor runtimes.

Avoid Generating Large Intermediate Results. While exploiting automatic broadcasting becomes useful in many cases, in certain cases it can become extremely inefficient due to the materialization of large intermediate tensors. Consider the Euclidean distance matrix calculation, which is a popular sub-operation in many ML operators (e.g., SVMs, KNearestNeighbor). Given two tensors $X \in \mathsf{R}^{n \times d}$ and $Y \in \mathsf{R}^{m \times d}$, the objective is to calculate tensor $D \in \mathsf{R}^{n \times m}$, where $D_{i;j} = ||X_i - Y_j||_2^2$. Implementing this using broadcasting requires first reshaping X to $X' \in \mathsf{R}^{n \times 1 \times d}$, Y to $Y' \in \mathsf{R}^{1 \times m \times d}$, calculate $(X' - Y') \in \mathsf{R}^{n \times m \times d}$, and perform a final `sum` reduction over the last dimension. This approach causes a size blowup by a factor of d in intermediate tensors. Alternatively, a popular trick [28] is to use the quadratic expansion of $D_{i;j} = ||X_i||_2^2 + ||Y_j||_2^2 - 2 \cdot X_i Y_j^T$ and calculate the individual terms separately. This avoids generating a large intermediate tensor.

Fixed Length Restriction on String Features. Arbitrary lengths of string features pose a challenge for HUMMINGBIRD. Strings are commonly used for categorical features in traditional ML datasets, and operators like one-hot encoding and feature hashing in traditional ML tools natively support string features. To support string features, HUMMINGBIRD imposes a fixed length restriction with the length being determined by the max size of any string in the vocabulary. Vocabularies are generated during training and can be accessed at compile time by HUMMINGBIRD. Fixed length strings can then be encoded into an int8 data type and processed by tensor runtimes.

5. EXPERIMENTAL EVALUATION

In our experimental evaluation we answer the following questions: (1) how does HUMMINGBIRD perform for models from popular tree ensemble algorithms such as XGBoost [30]? (2) Can we exploit hardware accelerators to increase the performance of traditional ML models and featurizers? (3) What are the performance trade-offs introduced by the different tree implementations? (4) What is the speedup introduced by HUMMINGBIRD for real world end-to-end pipelines? To address the above questions, we report three micro-benchmark experiments showing how HUMMINGBIRD performs compared to current state-of-the-art for inference over (1) tree ensembles (Section 5.1.1); (2) other featurization operators and ML models (Section 5.1.2); and (3) the need for heuristics for picking the best tree-model implementation (Section 5.1.3). Finally, we carry one end-to-end evaluation using complete pipelines (Section 5.2). We evaluate both CPUs and hardware accelerators (GPUs).

Hardware and Software Setup. For all the experiments (except explicitly stated otherwise) we used an Azure NC6 v2 machine equipped with 112 GB of RAM, an Intel Xeon CPU E5-2690 v4 @ 2.6GHz (6 virtual cores), and an Nvidia P100 GPU. It runs Ubuntu 18.04 with PyTorch 1.3.1, TVM 0.6, scikit-learn 0.21.3, XGBoost 0.9, ONNX runtime 1.0, RAPIDS 0.9, and CUDA 10. We run TVM with `opt_level` 3 when optimizations do not fail; 0 otherwise.

Experimental Setup. We run all the experiment 5 times and report the truncated mean by averaging the middle values. In the following, we use ONNX-ML to indicate running an ONNX-ML (i.e., traditional ML part of the standard) on the ONNX runtime. Additionally, we use **bold numbers** to highlight the best performance for the specific setup (CPU or GPU, record-at-a-time or batch). Note that both scikit-learn and ONNX-ML do not support hardware acceleration.

5.1 Micro-benchmarks

5.1.1 Tree Ensembles

Setup. This experiment is run over a set of popular datasets used for benchmarking gradient boosting frameworks [20]. We first do a 80%/20% train/test split over each dataset. Successively, we train (over the 80% split) a scikit-learn's *random forest*, *XGBoost* [30], and *LightGBM* [42] model using the default parameters of the benchmark. Specifically, we set the number of trees to 500, and maximum depth to 8. For XGBoost and LightGBM we use the scikit-learn API. Note that each algorithm generates trees with different structures, and this experiment helps with understanding how HUMMINGBIRD behaves with various tree types and dataset scales. For example, XGBoost generates balanced trees, LightGBM mostly generates skinny tall trees, while random forest is a mix between the two. Finally, we run inference on the trained models over the test dataset using different batch sizes. We compare the results against HUMMINGBIRD with different runtime backends, and a ONNX-ML version of the model generated using ONNXMLTools [16]. When evaluating over GPU, we also compared against NVIDIA RAPIDS Forest Inference Library (FIL) [24]. We don't compare against GPU implementations for XGBoost or LightGBM because we consider FIL as state-of-the-art [17]. For the batch experiments, we use all six cores in the machine, while for record-at-a-time experiments we use one core. We set a timeout of 1 hour for each experiment.

Table 7: Datasets used from the tree ensembles micro-benchmark

Datasets	#Rows	#Columns	Task
Fraud	$285K$	28	Binary
Epsilon	$500K$	2000	Binary
Year	$515K$	90	Regression
CovType	$581K$	54	Multiclass
Higgs	$11M$	28	Binary
Airline	$115M$	13	Binary

Datasets. We used 6 of the 7 datasets from NVIDIA's gbm-bench [20] (Bosh contains missing values with HUMMINGBIRD currently does not support). Each datasets is described in Table 7. The datasets cover a wide spectrum of use-cases: from regression to multiclass classification, from few hundred thousand rows to 100 million, and from few tens of columns to 2000.

Table 8: Batch Experiments (10K records at-a-time) for both CPU (6 cores) and GPU. Reported numbers are in seconds.

Algorithm	Dataset	Baselines (CPU)		Hummingbird CPU			Baselines (GPU)	Hummingbird GPU	
		Sklearn	ONNX-ML	PyTorch	TorchScript	TVM	RAPIDS FIL	TorchScript	TVM
Rand. Forest	Fraud	**2.5**	7.1	8.0	7.8	3.0	not supported	0.044	**0.015**
	Epsilon	9.8	18.7	14.7	13.9	**6.6**	not supported	**0.13**	**0.13**
	Year	1.9	6.6	7.8	7.7	**1.4**	not supported	0.045	**0.026**
	Covtype	**5.9**	18.1	17.22	16.5	6.8	not supported	0.11	**0.047**
	Higgs	**102.4**	257.6	314.4	314.5	118.0	not supported	1.84	**0.55**
	Airline	1320.1	timeout	timeout	timeout	**1216.7**	not supported	18.83	**5.23**
LightGBM	Fraud	3.4	5.9	7.9	7.6	**1.7**	**0.014**	0.044	**0.014**
	Epsilon	10.5	18.9	14.9	14.5	**4.0**	0.15	0.13	**0.12**
	Year	5.0	7.4	7.7	7.6	**1.6**	**0.023**	0.045	0.025
	Covtype	51.06	126.6	79.5	79.5	**27.2**	not supported	0.62	0.25
	Higgs	198.2	271.2	304.0	292.2	**69.3**	0.59	1.72	**0.52**
	Airline	1696.0	timeout	timeout	timeout	**702.4**	5.55	17.65	**4.83**
XGBoost	Fraud	1.9	5.5	7.7	7.6	**1.6**	**0.013**	0.44	0.015
	Epsilon	7.6	18.9	14.8	14.8	**4.2**	0.15	0.13	**0.12**
	Year	3.1	8.6	7.6	7.6	**1.6**	**0.022**	0.045	0.026
	Covtype	42.3	121.7	79.2	79.0	**26.4**	not supported	0.62	**0.25**
	Higgs	126.4	309.7	301.0	301.7	**66.0**	0.59	1.73	**0.53**
	Airline	1316.0	timeout	timeout	timeout	**663.3**	5.43	17.16	**4.83**

List of Experiments. We run the following set of experiments: (1) batch inference, both on CPU and GPU; (2) request/response scenario where one single record is scored at a time; (3) scaling experiments by varying batch sizes, both over CPU and GPU; (4) evaluation on how HUMMINGBIRD behaves on different GPU generations; (5) dollar cost per prediction; (6) memory consumption; (7) validation of the produced output wrt scikit-learn; and finally (8) time spent on compiling scikit-learn models.

Batch Inference. Table 7 reports the inference time (in seconds) for random forest, XGBoost and LightGBM models run over the 6 datasets. The batch size is set to $10K$ records. How performance varies with the batch sizes will be described later in this section. Looking at the CPU numbers from the table, we can see that:

1. Among the baselines, scikit-learn models outperform ONNX-ML implementations by 2× to 3×. This is because ONNX-ML is not currently optimized for batch inference.
2. Looking at the HUMMINGBIRD's backends, there is not a large difference between PyTorch and TorchScript, and in general these backends perform comparable to ONNX-ML.
3. The TVM backend provides the best performance on 15 experiments out of 18. In the worst case TVM is 20% slower (than scikit-learn); in the best cases it is up to 2× faster compared to the baseline solutions.

Let us look now at the GPU numbers of Table 7:

1. Baseline RAPIDS does not support random forest nor multiclass classification tasks. For the remaining experiments, GPU acceleration is able to provide speedups of up to 300× compared to CPU baselines.[2]
2. Looking at HUMMINGBIRD backends, TorchScript is about 2 to 3× slower compared to RAPIDS. TVM is instead the faster solution on 14 experiments out of 18, with a 10% to 20% improvement wrt RAPIDS.

[2]The original FIL blog post [17] claims GPU acceleration to be in the order of 28× for XGBoost, versus close to 300× in our case (Airline). We think that the difference is in the hardware: in fact, they use 5 E5-2698 CPUs for a total of 100 physical cores, while we use a E5-2690 CPU with 6 (virtual) physical cores. Additionally, they use a V100 GPU versus a P100 in our case.

The results are somehow surprising: HUMMINGBIRD targets the high-level tensor APIs provided by PyTorch and TVM, and still it is able to outperform custom C++ and CUDA implementations. These results highlight the intuition that DNN frameworks can be used as generic compilers for workloads beyond deep learning.

Request/Response. In this scenario, one single record is scored at a time. For this experiment we run inference over the entire test datasets, but with batch size equal to one. The results are depicted in Table 9. [3] As we can see:

1. Differently than the batch scenario, ONNX-ML is much faster compared to scikit-learn, in some cases even more than 100×. That reason is that ONNX-ML is currently optimized for single record, single core inference, whereas scikit-learn design is more towards batch inference.
2. PyTorch and TorchScript, again, behave very similarly. For random forest they are faster than scikit-learn but up to 5× slower compared to ONNX-ML. For LightGBM and XGBoost they are sometimes on par with scikit-learn, sometime slower. In the following experiments, we omit PyTorch as its performance is similar to TrochScript.
3. TVM again provides the best performance in 11 cases out of 15, with a best case of 3× compared to the baselines.

These results are again surprising, considering that tensor operations should be more optimized for bulk workloads rather than request/response scenarios. In the next section we will see how performance evolves as we change the batch size.

Scaling the Batch Size. We study how the performance of baselines and HUMMINGBIRD's backends change with the batch size. Figures 4a and 4b depicts the performance variation over CPU and GPU, respectively. We report only a few combinations of dataset / algorithm, but all the other combinations behave similarly. Starting with the CPU experiment, we can see that ONNX-ML has the best runtime for batch size of 1, but then its performance remains flat as we increase the batch size. TorchScript and scikit-learn did not

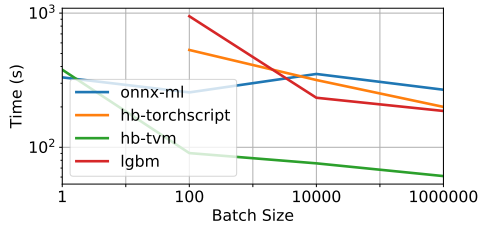
[3]Note that we removed the Airline dataset since no system was able to complete within the 1 hour timeout.

Table 9: Request/Response experiments (1 record at-a-time on 1 CPU core). Numbers are in seconds.

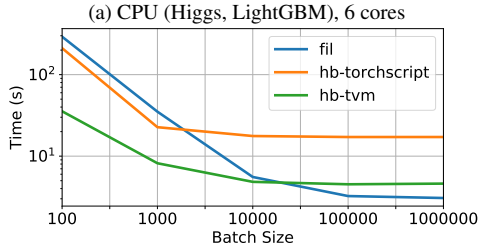
Algorithm	Dataset	Baselines		Hummingbird		
		Sklearn	ONNX-ML	PyTorch	TorchScript	TVM
Rand. Forest	Fraud	1688.22	**9.96**	84.95	75.5	11.63
	Epsilon	2945.42	32.58	153.32	134.17	**20.4**
	Year	1152.56	18.99	84.82	74.21	**9.13**
	Covtype	3388.50	35.49	179.4	157.8	**34.1**
	Higgs	timeout	**335.23**	timeout	timeout	450.65
LightGBM	Fraud	354.27	12.05	96.5	84.56	**10.19**
	Epsilon	40.7	29.28	167.43	148.87	**17.3**
	Year	770.11	16.51	84.55	74.05	**9.27**
	Covtype	135.39	209.16	854.07	822.93	**42.86**
	Higgs	timeout	**374.64**	timeout	timeout	391.7
XGBoost	Fraud	79.99	**7.78**	96.84	84.61	10.21
	Epsilon	121.21	27.51	169.03	148.76	**17.4**
	Year	98.67	17.14	85.23	74.62	**9.25**
	Covtype	135.3	197.09	883.64	818.39	**43.65**
	Higgs	timeout	585.89	timeout	timeout	**425.12**

Table 10: Peak memory consumption for the Fraud dataset. Batch size of 1000 records.

Algorithm	Framework	Memory
Random Forest	Sklearn	180MB
	ONNX-ML	265MB
	TorchScript	375MB
	TVM	568MB
LightGBM	Sklearn	182MB
	ONNX-ML	258MB
	TorchScript	370MB
	TVM	620MB
XGBoost	Sklearn	392MB
	ONNX-ML	432MB
	TorchScript	568MB
	TVM	811MB



(a) CPU (Higgs, LightGBM), 6 cores



(b) GPU (Airline, LightGBM)

Figure 4: Performance wrt scaling the batch size.

complete within the timeout for batch equal to 1, but, past 100, they both scale linearly as we increase the batch size. TVM is comparable to ONNX-ML for batch of 1; for batches of 100 records it gets about 5× faster, while it scales like TorchScript for batches greater than 100. This is likely due to the fact that TVM applies a set of optimizations (such as operator fusion) that introduce a constant-factor speedup compared to TorchScript.

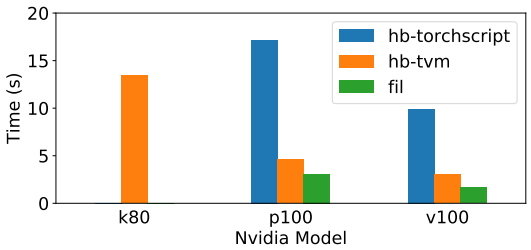
Looking at the GPU numbers (Figure 4b), TorchScript and TVM again follow a similar trend, with TVM being around 3× faster than TorchScript. Both TVM and TorchScript plateau at about a batch size of $10K$. RAPIDS FIL is slower than TorchScript for small batch sizes, but it scales better than HUMMINGBIRD. This is because of its custom CUDA implementation that is able to better use hardware under higher utilization. Interestingly, FIL as well plateaus at around $100K$ records. The custom CUDA implementation introduces a 50% gain over HUMMINGBIRD with TVM runtime.

Scaling Hardware. We tested how RAPIDS FIL and HUMMINGBIRD (TorchScript and TVM) scale as we change the GPU model. For this experiment we tried both with a large batch size ($1M$ records, Figure 11a) to maximize hardware utilization, and a smaller batch size ($1K$, Figure 11b). We ran this on all datasets across random forest, LightGBM, XGBoost with similar results, and present the Airline dataset (the largest) with LightGBM as a representative sample. We tested on three NVIDIA devices: K80 is the oldest (2014), P100 is the medium (2016), while V100 is the newer (2017). From the figures, in general we can see that: (1) RAPIDS FIL does not run on the K80 because it is an old generation; (2) with batch of 1K we get slower total inference time because we don't utilize the full hardware; (3) TorchScript and TVM runtimes for HUMMINGBIRD scale similarly on different hardware, although TVM is consistently 4 to 7× faster; (4) FIL scales similarly to HUMMINGBIRD, although it is 50% faster on large batches, 3× slower for smaller batches; (5) TorchScript is not optimal in memory management because for batches of $1M$ it fails on the K80 with an OOM exception. Finally, we also were able to run HUMMINGBIRD on the new Graphcore IPU [14]. At the moment, we are only able to run inference on a single decision tree (with numbers on par with or faster than our decision tree experiments). We look forward to expanding the experiment with tree ensembles.

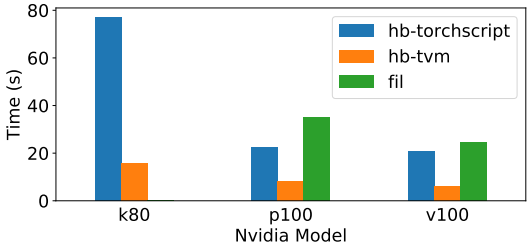
Cost. Figure 5 shows the cost comparison between the Azure VM instance equipped with GPU, and a comparable one without GPU (E8 v3). The plot shows the cost of executing 100k samples with batch size 1000 for random forest. The cost is calculated based on the hourly rate of each VM divided by the amortized cost of a single prediction. We executed scikit-learn on the CPU, and TorchScript and TVM on the GPU for comparison. We found that the CPU cost was significantly higher (between 10x-120x) across all experiments. [4] An interesting result was that the oldest GPU was the most cost effective, with the K80 and TVM having the lowest cost for 13 out of the 18 experiments (including LightGBM and XGBoost, not pictured). This result is explained by the fact that the K80 is readily available at significantly lower cost.

Memory Consumption. We measured the peak memory consumption over the Fraud dataset and for each algorithm. We used the `memory_usage` function in the `memory_profiler` library [2]. The numbers are reported in Figure 10, and are related to the execution over 1 core with a batch size of 1000 records. As we can see, scikit-learn is always the most memory efficient. ONNX-ML consumes from 10% to 50% more memory than scikit-learn, while HUMMINGBIRD with TorchScript runtime consumes from 50% to about 2× more memory than scikit-learn. Conversely, TVM con-

[4]Note: airline times out for random forest for CPU with 1k batch.



(a) Batch size $1M$



(b) Batch size 1K

Figure 6: Performance across GPUs for Airline, LightGBM

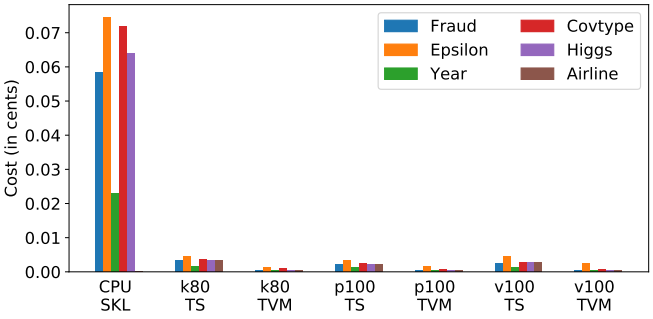


Figure 5: Cost for random forest 100k samples, batch size of 1k.

sumes from 2× to 3× more memory wrt scikit-learn. Note that the batch size influences the total memory consumption.

Output Validation. Since we run tree ensemble models as tensor operations, we could introduce rounding errors over floating point operations. Therefore, we need to validate that indeed the outputs produced by the different systems match. To evaluate this, we used the `numpy testing.assert_allclose` function, and we set a relative error of 10^{-5}, and an absolute error of 10^{-5}. We validate both the final scores and the probabilities (when available) for all combinations of datasets and algorithms. Out of the 18 experiments listed in Table 8, 9 of them returned no mismatches for HUMMINGBIRD, 12 in the ONNX-ML case. Among the mismatches, the worst case for HUMMINGBIRD is random forest with Covtype where we have 0.8% of records differing from the original scikit-learn output by more than the relative and absolute errors. For the Epsilon dataset, HUMMINGBIRD with random forest returns a mismatch on 0.1% of records. All the remaining mismatches effect less than 0.1% of records. Note that the differences are small. The biggest mismatch is of 0.086 (absolute difference) for Higgs using LightGBM. For the same experiment ONNX-ML has an absolute difference of 0.115.

Conversion Time. Table 12 shows the time it takes to convert a trained model into a target framework. The numbers are related to the generation of model running on a single core. As we can see, converting a model to ONNX-ML can take up to few tens of seconds; HUMMINGBIRD with PyTorch backend is constantly about 2× to 3× faster wrt ONNX-ML in converting random forests models, while it varies for LightGBM and XGBModels. TorchScript

Table 12: Conversion times (sec) for models with parallelism of 1 core.

Algorithm	Dataset	ONNX-ML	Hummingbird		
			PyTorch	TorchScript	TVM
Rand. Forest	Fraud	1.28	0.55	0.58	102.37
	Epsilon	7.53	2.63	2.67	108.64
	Year	7.11	2.77	2.86	69.99
	Covtype	9.87	2.16	2.2	106.8
	Higgs	8.25	2.41	2.44	103.77
	Airline	6.82	2.42	2.53	391.07
LightGBM	Fraud	1.34	0.98	1.06	3.42
	Epsilon	11.71	7.55	7.60	9.95
	Year	9.49	6.11	6.15	8.35
	Covtype	32.46	22.57	23.12	26.36
	Higgs	6.73	25.04	26.3	109
	Airline	11.52	6.38	6.47	8.19
XGBoost	Fraud	0.55	0.65	0.7	86.59
	Epsilon	6.86	25.89	25.94	113.4
	Year	5.66	23.4	23.54	110.24
	Covtype	9.87	2.16	2.20	106.8
	Higgs	6.73	25.04	26.3	109

models are generated starting from PyTorch models, and in general this further compilation step does not introduce any major overhead. Finally, conversion to TVM is much slower compared to the other approaches, and it might take more than 3 minutes. This is due to code generation and optimizations introduced in TVM.

As a final note: parallel (i.e., more than 1 core) and GPU execution introduced further conversion time overheads, especially on TVM. For instance, TVM can take up to 40 minutes to convert a random forest model for execution on GPU.

5.1.2 *Operators*

Setup. This micro-benchmark is a replication of the suite comparing scikit-learn and ONNX-ML operators [3]. We tested all scikit-learn operators of the suite that are supported by both ONNX-ML and HUMMINGBIRD (minus tree ensembles models, already discussed in the previous section). The total number of operators we tested in this micro-benchmark is 13, and are a mix of ML models (Logistic Regression, Support Vector Machines, etc.) and featurizers (e.g., Binarizer, Polynomial, etc.). For this micro-benchmark we either score 1 single record (request/response) or 1 million records.

Datasets. For this micro-benchmark we used the Iris datasets [21] with 20 features. We generated the number of rows based on the experiments: $1M$ for batch and just 1 for request / response.

List of Experiments. We carried the following set of experiments: (1) request/response scenario with one single record (on CPU); (2) batch inference over the $1M$ records, both on CPU and GPU; (3) memory consumption and conversion time. We don't report output validation metrics as outputs of all transformed operators are correct.

Request / Response. Table 13 (left hand-side) contains the times to score 1 record. The results are similar to the request/response scenario for the tree ensemble micro-benchmark. Namely, ONNX-ML outperform both scikit-learn and HUMMINGBIRD in 9 out of 13 cases. Note, however, that all frameworks are within a factor of 2. The only outlier is polynomial featurizer which is about 10× faster on HUMMINGBIRD with TVM backend. This is because all operators, except polynomial featurizer, are relatively compute light.

Batch Inference. The batch numbers are reported on the right hand-side of Table 13. On CPU, scikit-learn is faster than ONNX-ML, up to 6× for polynomial featurizer, although in most of the cases the two systems are within a factor of 2. HUMMINGBIRD with

Table 13: Experiments for operators on both CPU (single core) and GPU. Reported numbers are in milliseconds.

Operator	Request/Response (1 Record)				Batch (1 Million Records)					
	Baselines (CPU)		Hummingbird CPU		Baselines (CPU)		Hummingbird CPU		Hummingbird GPU	
	Sklearn	ONNX-ML	TorchScript	TVM	Sklearn	ONNX-ML	TorchScript	TVM	TorchScript	TVM
LogisticRegression	0.087	**0.076**	0.1	0.1	970	1540	260	**47**	**13**	15
SGDClassifier	**0.098**	0.1	0.12	0.1	180	1540	270	**49**	**11**	15
LinearSVC	0.077	**0.05**	0.11	0.1	110	69	260	**51**	**12**	18
NuSVC	0.086	**0.072**	4.1	0.14	3240	4410	**2800**	3000	140	**72**
SVC	0.086	**0.074**	2.3	0.12	1690	2670	**1520**	1560	120	**41**
BernoulliNB	0.26	**0.1**	0.07	0.11	280	1670	290	**65**	**12**	0.014
MLPClassifier	0.15	0.11	**0.1**	0.12	930	1860	**910**	1430	**17**	31
DecisionTreeClassifier	0.087	**0.074**	0.44	0.12	59	1610	560	**35**	**13**	16
Binarizer	0.064	**0.053**	0.063	0.1	98	75	**39**	59	**38**	**38**
MinMaxScaler	0.066	0.060	**0.058**	0.1	92	200	78	**57**	**38**	**38**
Normalizer	0.11	**0.063**	0.072	0.1	94	140	**83**	97	**39**	40
PolynomialFeatures	1.2	1	0.5	**0.1**	4030	29160	6380	**3130**	**340**	error
StandardScaler	0.069	**0.048**	0.059	0.1	150	200	77	**58**	**38**	**38**

TorchScript backend is competitive with scikit-learn, whereas with TVM backend HUMMINGBIRD is faster on 8 out of 13 operators, with in general a speedup of about 2× compared to scikit-learn.

If now we focus to the GPU numbers, we see that HUMMINGBIRD with TorchScript backend compares favorably against TVM on 11 operators out of 13. This is in contrast with the tree ensemble micro-benchmark where the TVM backend was faster than the TorchScript one. We suspect that this is because TVM optimizations are less effective on these "simpler" operators. In this micro-benchmark, GPU acceleration does not provide the speedup we instead saw for the tree ensemble models. In general, we see around 2× performance improvement over the CPU runtime: only polynomial featurizer run much faster, with almost a 10× improvement. Again, this is because the listed operators (with the exclusion of polynomial featurizer) are compute light. Unfortunately, TVM returns a runtime error when generating the polynomial featurizer model for GPU execution.

Memory Consumption and Conversion Time. We measured the peak memory consumed and conversion time for each operator on each framework. We used batch inference over 1000 records. For memory consumption, the results are in line with what we already saw in Section 5.1.1. ONNX-ML implementation introduces a 10% to 20% memory overhead. HUMMINGBIRD with TorchSchript and TVM backends are about to 2× to 3× more memory hungry than scikit-learn, respectively. Regarding the conversion time, for ONNX-ML and HUMMINGBIRD with TorchScript, the conversion time is in the order of few milliseconds (30ms for TorchScript, 100ms for ONNX-ML). The TVM backend is slightly slower but still in the order of few tens of milliseconds (exception for NuSVC and SVC which take up to 3.2 seconds). In comparison with the numbers reported for the tree ensembles models (Table 12), we confirm that these operators are simpler, even from a compilation perspective.

5.1.3 Tree Models Implementation

Next, we test the different tree-based models implementation to make the case for the heuristics.

Datasets. For this micro-benchmark we employ a synthetic dataset randomly generated with 5000 rows and 200 features.

Experiments Setup. We study the behavior of the tree implementations as we change the training algorithm, the batch size, and the tree depth. For each experiment we set the number of trees to 100. We will use the TVM runtime backend (similar results hold for the other runtimes as well). Each experiment is run on 1 CPU core.

Results. Figure 7 shows the comparison between the different tree implementations, and two baselines: the original scikit-learn model and the related to-ONNX-ML-translated version. In the top part of the figure we run all experiments using a batch size of 1; on the bottom part of the figure we instead use a batch size of 1000. In the column on the left-hand size, we generate trees with a max depth of 3; 7 for the middle column, and 12 for column on the right-hand side of the figure. In general, two things are apparent: (1) our implementation is always as fast as or better than the baselines (as we already saw in Section 5.1.1); and (2) no tree implementation is always better than the others. Interestingly, the GEMM implementation outperforms the other two for small batch sizes, whereas TreeTraversal and PerfectTreeTraversal are better over larger batch sizes. Between TreeTraversal and PerfectTreeTraversal, the latter is usually the best performant (although not by a large margin). PerfectTreeTraversal however creates balanced trees, and fails for very deep trees. The heuristics used in HUMMINGBIRD to pick at compilation time the tree implementation are motivated by this experiment. Similar conclusions hold also for GPU execution.

5.2 End-to-end Pipelines

Setup. In this experiment we test HUMMINGBIRD over end-to-end real world pipelines. We downloaded the 72 tasks (and related datasets) composing the OpenML-CC18 suite [23]. Among all the tasks, we discarded all the "not pure scikit-learn" ML pipelines, i.e., we discarded all the pipelines not exclusively composed by scikit-learn operators (e.g., containing also arbitrary Python code). We successively discarded all the pipelines returning a failure during training. 88% of the remaining pipelines are exclusively composed by operators supported by HUMMINGBIRD, for a total of 2328 ML pipelines. Among this, 11 failed during inference due to runtime errors in HUMMINGBIRD; we report the summary of executing 2317 pipelines. These pipelines contain an average of 3.3 operators, which is in line with what we observed elsewhere [57].

Datasets. For this experiment we have 72 datasets in total [23]. The datasets are a curated mix specifically designed for ML benchmarking. We did the typical 80%/20% split between training and inference. The smaller dataset has just 100 records, the bigger 19264, while the median value is 462. The minimum number of columns for a dataset is 4, the maximum 3072, with a median of 30.

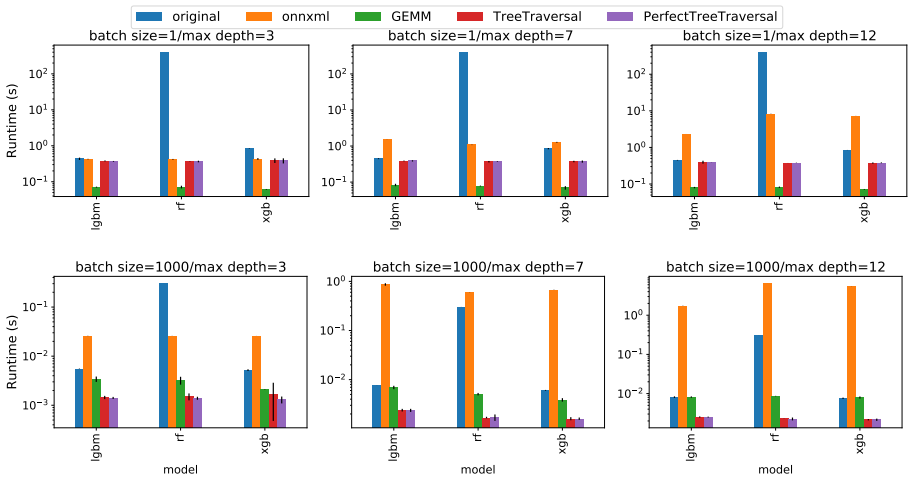


Figure 7: Comparison between the different tree implementations as we vary the batch size and tree depth.

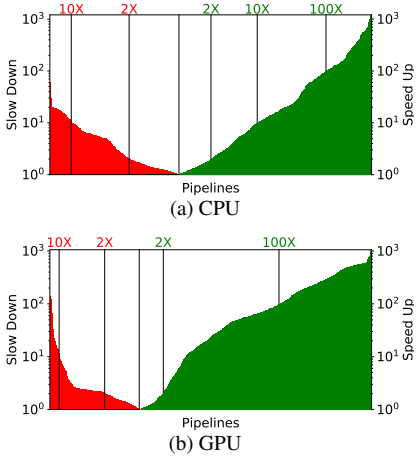


Figure 8: Speedup/slowdown of pipelines when using HUMMINGBIRD wrt baseline Sklearn.

List of Experiments. We run all pipelines and report the speedup / slowdown introduced by HUMMINGBIRD with TorchScript backend wrt baseline scikit-learn on both CPU and GPU. We do not report any conversion time / memory consumption / output validation results since these are the aggregation of the numbers reported previously.

Results. Figure 8 summarizes the speedup / slowdown introduced by HUMMINGBIRD when scoring all 2317 pipelines. As we can see, HUMMINGBIRD is able to accelerate about 60% of the pipelines on CPU (8a). In general, the slowest pipeline gets about 60× slower wrt scikit-learn, the fastest instead gets a 1200× speed up. The slowdowns are due to a couple of factors: (a) the datasets used for these experiments are quite small; (b) some pipelines contain largely sparse operations (i.e., SVM on sparse inputs); (c) several pipelines are small and do not require much computation (e.g., a simple inputer followed by a small decision tree). These three factors are highlighted also by the fact that even if we move computation to the GPU (8b), still 27% of the pipelines have some slowdown. Note however that (1) both sparse and small pipelines can be detected at compile time, and therefore we can return a warning or an error; (2) DNN frameworks are continuously adding new sparse tensor operations (e.g., [26]); and (3) an option could be to add a specific runtime backend for sparse tensor operations (e.g., we have a prototype integration with TACO [43]). In general, DNN frameworks are relatively young, and HUMMINGBIRD will exploit any future improvement with no additional costs.

With GPU acceleration (Figure 8b), 73% of the pipelines show some speedup. The slowest pipeline gets about 130× slower wrt scikit-learn, the fastest instead gets a speedup of 3 orders of magnitude. Some of the pipelines get worse from CPU to GPU execution. This is due to (1) sparsity; (2) small compute; and (3) data movements between CPU and GPU memory. Indeed we run all pipelines on GPU, even the ones for which in practice would not make much sense (e.g., a decision tree with 3 nodes). We leave as future work an extension to our heuristics for picking the right hardware backend.

6. RELATED WORK

PyTorch [54], TensorFlow [12], MXNet [11], CNTK [9] are DNN frameworks that provide easy-to-use (tensor-based) APIs for authoring DNN models, and heterogeneous hardware support for both training and inference. Since the performance of these systems is comparable [63], we picked PyTorch as the default runtime backend for HUMMINGBIRD. Beyond these popular frameworks, inference runtimes such as ONNX [4], nGraph [15], TVM [31], and TensorRT [18] provide optimizations and efficient execution targets, specifically for inference. To prove the versatility of our approach, we have tested HUMMINGBIRD with both PyTorch and TVM. HUMMINGBIRD uses a two-level, logical-physical optimization approach similar to relational databases. First, we apply logical optimizations based on the operators composing the pipeline. Afterwards, physical operator implementations are selected based on model statistics, and physical rewrites, which are externally implemented by the DNN runtime, are executed (e.g., algebraic rewrites, operator fusion). Willump [44] uses a similar two-level optimization strategy, although it targets Weld [53] as its low level runtime and therefore it cannot natively support inference on hardware accelerators. Conversely, HUMMINGBIRD casts ML pipelines into tensor computations. Hence, it takes advantage of DNN serving systems and eases the deployment on many target environments. Other optimizers for inference pipelines, such as Pretzel [47], only target higher level (logical) optimizations. While HUMMINGBIRD is currently built on top of skl2onnx, another option could be MLIR [46].

Several works deal with executing tree (ensemble) [24, 52, 59] and graphical models [49, 50] on hardware accelerators. These systems however provide custom implementations specific to a target hardware (e.g., NVIDIA GPUs for RAPIDS FIL [24], FPGAs for [52]). HUMMINGBIRD takes advantage of existing NNs solutions to improve efficiency of traditional ML prediction on various hardware and platforms, hence minimizing engineering efforts. Finally, while this work only covers inference, similar techniques can be used for training as well [64]. HUMMINGBIRD can be used alone, but also integrated with other optimizers, e.g., Raven [41].

7. CONCLUSIONS

In this paper, we explore the idea of using frameworks such as PyTorch and TensorFlow, not as plain DNN systems but as generic compilers and optimizers for heterogeneous hardware. Our use-case is "traditional" ML inference. We ported 40+ data featurizers and traditional ML models into tensor operations, and tested their performance over two DNN frameworks (PyTorch and TVM) and over different hardware (CPUs and GPUs). The results are surprising: even though HUMMINGBIRD targets high-level tensor operations, it is able to outperform custom C++ and CUDA implementations. To our knowledge, HUMMINGBIRD is the first system able to run classical ML inference on heterogeneous hardware, while proving that DNN frameworks are mature enough to be used as generic compilers for heterogeneous hardware.

We think HUMMINGBIRD opens many possibilities that we didn't explore in this work. For instance, approximate evaluation of pipelines using quantization, a feature now available in many DNN frameworks. Another open question is whether this same approach can also be used to run relational operators over accelerators.

8. REFERENCES

[1] Cerebras Chip. https://www.wired.com/story/power-ai-startup-built-really-big-chip/.

[2] Memory profiler for Python.

[3] ONNX-ML vs Sklearn Benchmark.

[4] ONNX Runtime. https://github.com/microsoft/onnxruntime.

[5] ONNX Supported Frameworks and Backends. https://onnx.ai/supported-tools.html.

[6] Sambanova: Massive Models for Everyone.

[7] TorchScript Documentation. https://pytorch.org/docs/stable/jit.html.

[8] H2O Algorithms Roadmap. https://github.com/h2oai/h2o-3/blob/master/h2o-docs/src/product/flow/images/H2O-Algorithms-Road-Map.pdf, 2015.

[9] CNTK. https://docs.microsoft.com/en-us/cognitive-toolkit/, 2018.

[10] Matplotlib. https://matplotlib.org/, 2018.

[11] MXNet. https://mxnet.apache.org/, 2018.

[12] TensorFlow. https://www.tensorflow.org, 2018.

[13] Esg technical validation: Dell emc ready solutions for ai: Deep learning with intel. https://www.esg-global.com/validation/esg-technical-validation-dell-emc-ready-solutions-for-ai-deep-learning-with-intel, 2019.

[14] Graphcore IPU. https://www.graphcore.ai/, 2019.

[15] nGraph. https://www.ngraph.ai/, 2019.

[16] ONNXMLTools. https://github.com/onnx/onnxmltools, 2019.

[17] RAPIDS Forest Inference Library. https://medium.com/rapids-ai/rapids-forest-inference-library-prediction-at-100-million-rows-per-second-19558890bc35, 2019.

[18] Tensor-RT. https://developer.nvidia.com/tensorrt, 2019.

[19] Broadcasting Semantic. https://www.tensorflow.org/xla/broadcasting, 2020.

[20] Gradient Boosting Algorithm Benchmark. https://github.com/NVIDIA/gbm-bench, 2020.

[21] Iris dataset. https://scikit-learn.org/stable/auto_examples/datasets/plot_iris_dataset.html, 2020.

[22] ONNX. https://github.com/onnx/onnx/blob/master/docs/Operators.md, 2020.

[23] OpenML-CC18 Benchmark. https://www.openml.org/s/99, 2020.

[24] RAPIDS cuML. https://github.com/rapidsai/cuml, 2020.

[25] skl2onnx Converter. https://github.com/onnx/sklearn-onnx/, 2020.

[26] The Status of Sparse Operations in Pytorch. https://github.com/pytorch/pytorch/issues/9674, 2020.

[27] Z. Ahmed, S. Amizadeh, M. Bilenko, R. Carr, W.-S. Chin, Y. Dekel, X. Dupre, V. Eksarevskiy, S. Filipi, T. Finley, et al. Machine learning at Microsoft with ML.NET. In *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery Data Mining*, KDD '19, page 2448–2458, New York, NY, USA, 2019. Association for Computing Machinery.

[28] S. Albanie. Euclidean distance matrix trick. 2019.

[29] Amazon. The total cost of ownership (tco) of amazon sagemaker. https://pages.awscloud.com/rs/112-TZM-766/images/Amazon_SageMaker_TCO_uf.pdf, 2020.

[30] T. Chen and C. Guestrin. Xgboost: A scalable tree boosting system. In *Proceedings of the 22Nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '16, pages 785–794, New York, NY, USA, 2016. ACM.

[31] T. Chen, T. Moreau, Z. Jiang, L. Zheng, E. Yan, M. Cowan, H. Shen, L. Wang, Y. Hu, L. Ceze, C. Guestrin, and A. Krishnamurthy. Tvm: An automated end-to-end optimizing compiler for deep learning. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, OSDI'18, pages 579–594, Berkeley, CA, USA, 2018. USENIX Association.

[32] D. Crankshaw, G. Sela, C. Zumar, X. Mo, J. E. Gonzalez, I. Stoica, and A. Tumanov. Inferline: ML inference pipeline composition framework. *CoRR*, abs/1812.01776, 2018.

[33] D. Crankshaw, X. Wang, G. Zhou, M. J. Franklin, J. E. Gonzalez, and I. Stoica. Clipper: A low-latency online prediction serving system. In *NSDI*, 2017.

[34] M. Dash and H. Liu. Feature selection for classification. *Intelligent data analysis*, 1(3):131–156, 1997.

[35] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. *arXiv e-prints*, page arXiv:1810.04805, Oct 2018.

[36] FirmAI. Machine Learning and Data Science Applications in Industry. https://github.com/firmai/industry-machine-learning.

[37] I. Goodfellow, Y. Bengio, and A. Courville. *Deep learning*. MIT press, 2016.

[38] Intel. Machine learning fpga. https://www.intel.com/content/www/us/en/products/docs/storage/programmable/applications/machine-learning.html, 2020.

[39] Z. Jia, J. Thomas, T. Warszawski, M. Gao, M. Zaharia, and A. Aiken. Optimizing dnn computation with relaxed graph substitutions. In *Proceedings of the 2nd Conference on Systems and Machine Learning (SysML'19)*, 2019.

[40] N. P. Jouppi, C. Young, N. Patil, D. A. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers, R. Boyle, P. Cantin, C. Chao, C. Clark, J. Coriell, M. Daley, M. Dau, J. Dean, B. Gelb, T. V. Ghaemmaghami, R. Gottipati, W. Gulland, R. Hagmann, R. C. Ho, D. Hogberg, J. Hu, R. Hundt, D. Hurt, J. Ibarz, A. Jaffey, A. Jaworski, A. Kaplan, H. Khaitan, A. Koch, N. Kumar, S. Lacy, J. Laudon, J. Law, D. Le, C. Leary, Z. Liu, K. Lucke, A. Lundin, G. MacKean, A. Maggiore, M. Mahony, K. Miller, R. Nagarajan, R. Narayanaswami, R. Ni, K. Nix, T. Norrie, M. Omernick, N. Penukonda, A. Phelps, J. Ross, A. Salek, E. Samadiani, C. Severn, G. Sizikov, M. Snelham, J. Souter, D. Steinberg, A. Swing, M. Tan, G. Thorson, B. Tian, H. Toma, E. Tuttle, V. Vasudevan, R. Walter, W. Wang, E. Wilcox, and D. H. Yoon. In-datacenter performance analysis of a tensor processing unit. *CoRR*, abs/1704.04760, 2017.

[41] K. Karanasos, M. Interlandi, F. Psallidas, R. Sen, K. Park, I. Popivanov, D. Xin, S. Nakandala, S. Krishnan, M. Weimer, Y. Yu, R. Ramakrishnan, and C. Curino. Extending relational query processing with ML inference. In *CIDR 2020, 10th Conference on Innovative Data Systems Research, Amsterdam, The Netherlands, January 12-15, 2020, Online Proceedings*. www.cidrdb.org, 2020.

[42] G. Ke, Q. Meng, T. Finley, T. Wang, W. Chen, W. Ma, Q. Ye, and T.-Y. Liu. Lightgbm: A highly efficient gradient boosting decision tree. In I. Guyon, U. V. Luxburg, S. Bengio,

H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, editors, *Advances in Neural Information Processing Systems 30*, pages 3146–3154. Curran Associates, Inc., 2017.

[43] F. Kjolstad, S. Kamil, S. Chou, D. Lugato, and S. Amarasinghe. The tensor algebra compiler. *Proc. ACM Program. Lang.*, 1(OOPSLA):77:1–77:29, Oct. 2017.

[44] P. Kraft, D. Kang, D. Narayanan, S. Palkar, P. Bailis, and M. Zaharia. Willump: A Statistically-Aware End-to-end Optimizer for Machine Learning Inference. *arXiv e-prints*, page arXiv:1906.01974, Jun 2019.

[45] A. Krizhevsky, I. Sutskever, and G. E. Hinton. Imagenet classification with deep convolutional neural networks. *Commun. ACM*, 60(6):84–90, May 2017.

[46] C. Lattner, J. Pienaar, M. Amini, U. Bondhugula, R. Riddle, A. Cohen, T. Shpeisman, A. Davis, N. Vasilache, and O. Zinenko. MLIR: A Compiler Infrastructure for the End of Moore's Law. *arXiv e-prints*, page arXiv:2002.11054, Feb. 2020.

[47] Y. Lee, A. Scolari, B.-G. Chun, M. D. Santambrogio, M. Weimer, and M. Interlandi. PRETZEL: Opening the black box of machine learning prediction serving systems. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 611–626, Carlsbad, CA, Oct. 2018. USENIX Association.

[48] P. Li. Robust logitboost and adaptive base class (abc) logitboost. In *n Proceedings of the Twenty-Sixth Conference Annual Conference on Uncertainty in Artificial Intelligence (UAI'10)*.

[49] N. Ma, X. Xia, and V. Prasanna. Exact inference on manycore processors using pointer jumping. In *IASTED International Conferences on Informatics, Actapress*, 2010.

[50] N. Ma, Y. Xia, and V. K. Prasanna. Parallel exact inference on multicore using mapreduce. In *2012 IEEE 24th International Symposium on Computer Architecture and High Performance Computing*, pages 187–194. IEEE, 2012.

[51] W. Mckinney. pandas: a foundational python library for data analysis and statistics. 01 2011.

[52] M. Owaida, H. Zhang, C. Zhang, and G. Alonso. Scalable inference of decision tree ensembles: Flexible design for cpu-fpga platforms. In *2017 27th International Conference on Field Programmable Logic and Applications (FPL)*, pages 1–8, Sep. 2017.

[53] S. Palkar, J. Thomas, D. Narayanan, P. Thaker, R. Palamuttam, P. Negi, A. Shanbhag, M. Schwarzkopf, H. Pirk, S. Amarasinghe, et al. Evaluating end-to-end optimization for data analytics applications in weld. *Proceedings of the VLDB Endowment*, 11(9):1002–1015, 2018.

[54] A. Paszke, S. Gross, S. Chintala, G. Chanan, E. Yang, Z. DeVito, Z. Lin, A. Desmaison, L. Antiga, and A. Lerer. Automatic differentiation in pytorch. In *NIPS-W*, 2017.

[55] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in python. *J. Mach. Learn. Res.*, 12:2825–2830, Nov. 2011.

[56] N. Polyzotis, S. Roy, S. E. Whang, and M. Zinkevich. Data lifecycle challenges in production machine learning: A survey. *SIGMOD Record*, 47(2):17–28, 2018.

[57] F. Psallidas, Y. Zhu, B. Karlas, M. Interlandi, A. Floratou, K. Karanasos, W. Wu, C. Zhang, S. Krishnan, C. Curino, and M. Weimer. Data science through the looking glass and what we found there, 2019.

[58] K. Ramachandra, K. Park, K. V. Emani, A. Halverson, C. Galindo-Legaria, and C. Cunningham. Froid: Optimization of imperative programs in a relational database. *Proc. VLDB Endow.*, 11(4):432–444, Dec. 2017.

[59] T. Sharp. Implementing decision trees and forests on a gpu. In D. Forsyth, P. Torr, and A. Zisserman, editors, *Computer Vision – ECCV 2008*, pages 595–608, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

[60] C. Suisse. The apps revolution manifesto—volume 1: The technologies. https://aka.ms/enterprise-application-lifespan, 2012.

[61] Supun Nakandala, Karla Saur, GyeongIn Yu, Konstantinos Karanasos, Carlo Curino, Markus Weimer, and Matteo Interland. Taming Model Serving Complexity, Performance and Cost: A Compilation to Tensor Computations Approach. https://scnakandala.github.io/papers/TR_2020_Hummingbird.pdf, 2020. Technical Report.

[62] S. van der Walt, S. C. Colbert, and G. Varoquaux. The numpy array: A structure for efficient numerical computation. *Computing in Science Engineering*, 13(2):22–30, March 2011.

[63] A. A. Vincent Deuschle and V. Markl. End-to-end benchmarking of deep learning platforms. In *TPCTC Workshop*, 2019.

[64] G. Yu, S. Amizadeh, A. Pagnoni, B. Chun, M. Weimer, and M. Interlandi. Making classical machine learning pipelines differentiable: A neural translation approach. *CoRR*, abs/1906.03822, 2019.

APPENDIX

A. RUNTIME INDEPENDENT OPTIMIZATIONS

We discuss four novel optimizations, which are unique to traditional ML scoring. To the best of our knowledge, this is the first time that they are being applied in ML scoring and can be even implemented in traditional ML tools. However, our approach of separating out the prediction pipeline from training pipeline and compiling it into tensor computations makes it easy to implement these optimizations in HUMMINGBIRD.

A.1 Optimizations

Feature Selection Push-Down. Feature selection is a popular operation that is often used as the *final featurization step* as it reduces over-fitting and improves the accuracy of the ML model [34]. However, during scoring, it can be pushed down in the pipeline to avoid redundant computations such as scaling and one-hot encoding for discarded features or even reading the feature at all. This idea is similar to the concept of projection push-down in relation query processing but through user-defined table functions, which in our case are the ML operators. For operators such as feature scaling, which performs 1-to-1 feature transformations, selection push-down can be easily done. However, for operators such as one-hot encoding and polynomial featurization, which perform 1-to-m or m-to-1 feature transformations, the operator will have to absorb the feature selection and stop generating those features. For example, say one-hot encoding is applied on a categorical feature column which has a vocabulary size of 10, but 4 of those features are discarded by the feature selector. In such cases, we can remove such features from the vocabulary. After such absorbing, it is possible that some of the input features can still be discarded as they are not used at all, which

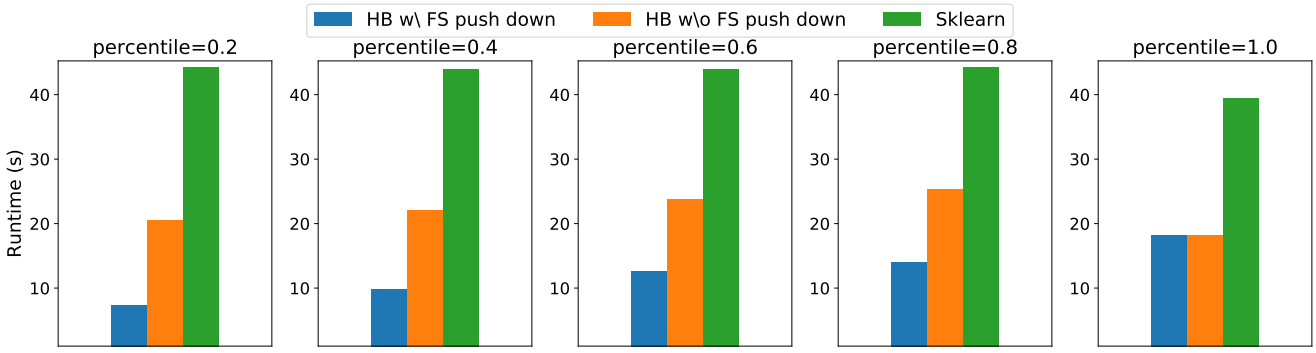


Figure 9: Feature selection push down + algebraic rewrite

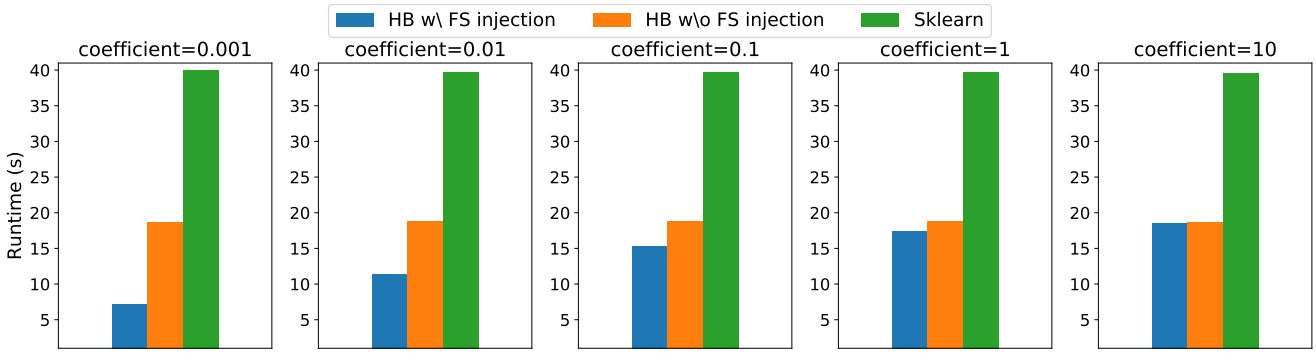


Figure 10: Feature selection injection + selection push down + algebraic rewrite.

allows us to push the feature selection even further. Note that for some "blocking" operators [47], such as feature normalizers, it is not possible to push-down the feature selection.

Feature Selection Injection. Even if the original pipeline doesn't have a feature selection operator, it is possible to inject one and then push it down to avoid redundant computations. Linear models with L1 regularization (Lasso) is a typical example where feature selection is implicitly performed. The same idea can be extended to tree-based models to prune the features that are not used as decision variables. In both of these examples, the ML model also has to be updated to take into account the pruned features. For linear models we prune the zero weights; for tree models, we update the indices of the decision variables.

Algebraic Rewrites. We found opportunities to rewrite several operators that perform linear algebra operations into a single `GEMM` operation. Consider a pipeline that trains a logistic regression model and has feature scaling and matrix decomposition (e.g., PCA) as the featurization steps. It is algebraically represented in Eq. 1-LHS.

$$\texttt{sigmoid}\left(\left(\left(\frac{X-\alpha}{\beta}\right)\cdot W_{PCA}\right)\cdot W_{LR}+B_{LR}\right) = \texttt{sigmoid}(X\cdot W+B) \quad (1)$$

Notice that parentheses in Eq. 1-LHS capture the order in which the operators were trained and it requires performing 5 tensor operations: 2 element-wise ops for scaling; two `GEMM` ops for matrix decomposition and logistic regression; and a final `sigmoid` op for logistic regression. It is possible to use linear algebra properties and represent the same pipeline using two ops as shown in RHS, where tensor W and B can be pre-computed and used during scoring. These kinds of patterns are in fact very common in classical ML; any subset of scaling, matrix decomposition, and linear models constitutes such patterns. However, they do not appear in DNNs due to the use of non-linear transformations and hence most tensor runtime optimizers are oblivious of these opportunities. HUMMINGBIRD's optimizer has a roster of such patterns and checks for potential rewrites during optimization.

Batching Stacked Models. Recall that in Section 4.1, we batched the tensors from all trees into single tensors and performed batched tensor operations. Alternatively, we could have stored them separately and invoke tensor operations on each tree. Though this would yield the same result it will be highly inefficient due to two reasons: (1) high operator invocation overhead and (2) high memory access overhead (due to multiple reading of the input tensor X). It is possible to apply the same batching optimization across multiple ML operators. Consider a stacked ML model that is composed of logistic regression, linear SVM, and Bernoulli Naive Bayes models. While these models are conceptually different during scoring all three of them are essentially performing a `GEMM` operation. Thus, it is possible to batch them together into one `GEMM` operation to reduce overheads. Efficiently finding this type of graph substitutions in an arbitrary tensor computation DAG is still an active area of research [39]; hence not supported by many DNN runtimes. On the other hand, HUMMINGBIRD implements few patterns as the one above, directly over traditional ML operators and triggers batching rewrites.

A.2 Experimental Evaluation

We ran an experiment to measure the benefits of the feature selection push down. In Figure 9 we compare HUMMINGBIRD (on TorchScript runtime) with and without feature selection push-down, and the baseline implementation of the pipelines in scikit-learn. For

this we use the OpenML Nomao dataset and use a pipeline which trains a logistic regression model with L2 loss. The featurization part contains one-hot encoding for categorical features, missing value imputation for numerical values, followed by feature scaling, and a final feature selection operator (scikit-learn SelectKBest). We vary the percentile of features that are picked by the feature selection operator. HUMMINGBIRD pushes down the feature selection and also rewrites feature scaling and logistic regression into one GEMM operation. Note that this rewrite is only possible as a result of feature selection push down. In general, we can see that HUMMINGBIRD without the above optimization is about 2× faster than scikit-learn in evaluating the pipelines. For small percentiles, the optimizations deliver a further 3×. As we increase the percentile of features that are selected (excluding percentile= 1.0, which is no feature selection) the runtime of HUMMINGBIRD both w/ and w/o optimizations increase, although with the optimization HUMMINGBIRD is still 2× faster than without.

We also ran an experiment to evaluate whether we can improve the performance of pipelines with sparse models by injecting (and then pushing down) feature selection operators. The pipeline is same as in the previous case but without the feature selection operator. Instead we train the logistic regression model with L1 regularization. In Figure 10 we vary the L1 regularization coefficient and study how much performance we can gain. We see with very sparse models we can see up to 3× improvement wrt HUMMINGBIRD w/o optimization. Performance gains dissipate as we decrease the sparsity of the model.