Hardware Acceleration for DBMS Machine Learning Scoring: Is It Worth the Overheads?

Zahra Azad*, Rathijit Sen†, Kwanghyun Park†, Ajay Joshi*
*Boston University, †Microsoft Gray Systems Lab
{zazad, joshi}@bu.edu, {rathijit.sen, kwanghyun.park}@microsoft.com

***Abstract*—Query processing for data analytics with machine learning scoring involves executing heterogeneous operations in a pipelined fashion. Hardware acceleration is one approach to improve the pipeline performance and free up processor resources by offloading computations to the accelerators. However, the performance benefits of accelerators can be limited by the compute and data offloading overheads. Although prior works have studied acceleration opportunities, including with accelerators for machine learning operations, an end-to-end application performance analysis has not been well studied, particularly for data analytics and model scoring pipelines.**

In this paper, we study speedups and overheads of using PCIe-based hardware accelerators in such pipelines. In particular, we analyze the effectiveness of using GPUs and FPGAs to accelerate scoring for random forest, a popular machine learning model, on tabular input data obtained from Microsoft SQL Server. We observe that the offloading decision as well as the choice of the optimal hardware backend should depend at least on the model complexity (e.g., number of features and tree depth), the scoring data size, and the overheads associated with data movement and invocation of the pipeline stages. We also highlight potential future research explorations based on our findings.

I. Introduction

Over the past decade, the demands of data processing workloads have increased rapidly and the amount of data generated has increased exponentially [1]. Big Data analytics organizes and extracts the valued information from this rapidly growing datasets and has a significant impact on so many different applications such as scientific explorations, healthcare, governance, finance and business analytics, and web analysis [1]. The algorithms for Big Data analytics are getting more complex with higher computational demands, driven by the success of machine learning (ML) models and applications.

An important platform to extract and analyze valuable information from Big Data is a database management system (DBMS) optimized for analytical data processing [2]–[6]. Enterprise DBMSs already support ML services [7]–[9], and we expect the trend of integration and co-evolution of traditional query processing and ML workloads to continue in future [4], [5]. This enables ML applications to leverage the well-established capabilities of DBMS to efficiently manage and control access to large amounts of critical data that are already stored in the DBMS, and it removes the need for users to maintain separate coherent copies of large datasets for traditional query processing and ML applications.

In this paper, we focus on analytic query processing with ML model scoring on Microsoft SQL Server by using its capability of executing user Python code [10]. In this application environment, users submit Transact-SQL (T-SQL) queries that include scoring of ML models where both models and data are stored in the database. Query processing involves executing a pipeline that includes invocation of a Python process, copying data, scoring a model, and returning the results.

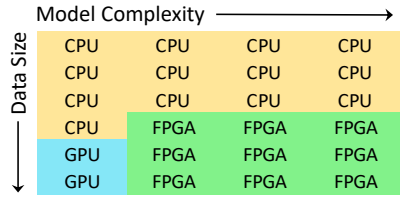


Fig. 1: The best-performing hardware for scoring a random forest model depends on the model complexity and data size.

DBMS ML operations are significantly more compute-intensive than the traditional relational operations, especially with ML over large datasets [11], [12]. Specialized hardwares such as Graph Processing Units (GPUs) and Field-Programmable Gate Arrays (FPGAs) can help accelerate the analytics and ML scoring pipelines by handling the computational demands of ML algorithms and also free up processor cores for other work. The architectural flexibility of FPGAs allows the design of very deep pipelines to handle compute intensive workloads [13]–[20], [21], [22], [2], [3], [6].

However, it turns out that sometimes using an accelerator is not efficient and the best accelerator to use at other times depends on the data size and model characteristics. Figure 1 shows the best-performing hardware backend — CPU (no accelerator offload), GPU, and FPGA — for the ML scoring operations of a random forest model, for different combinations of the size of the scoring dataset, dataset features, and model complexity. Since both data and models depend on the particular user query presented at run time, a scheduler that aims for the best performance would need to make the accelerator offloading decisions dynamically.

Furthermore, as we will show later (Section IV), the situation changes dramatically when the end-to-end pipeline and application performance is taken into account. While accelerators are still useful, additional overheads associated with data copy and invocation of pipeline stages cause the region of interest for using accelerators to shift towards larger

data sizes and ML models with more complex computations.

Thus, to fully understand the effectiveness of using accelerators to speed up DBMS ML scoring pipelines, we need to consider the big picture and include the latencies of all the tasks involved such as pre-processing of input data, extracting the ML model information, transferring input data and models to accelerators, accelerator setup/configuration, signaling task completion, returning results, and post-processing the results. This end-to-end aspect has been largely overlooked in prior works [13]–[20], [23] that have focused mainly on the FPGA compute capacity and the speedup of the offloaded task.

The main contributions of our work are the following.

1) An end-to-end performance analysis of the DBMS ML scoring pipeline to find sources of performance bottlenecks and inform future research on mitigating those challenges. Based on our observations, even for compute intensive models and a large number of records, the end-to-end performance is limited by the non-scoring stages in DBMS ML scoring pipelines; The ML scoring time is in order of milliseconds but the time of non-scoring stages are in order of microseconds (Section IV).
2) An in-depth study of the accelerator offloading overheads. This analysis helps find the achieved overall performance speedup for ML scoring and decide on when offloading a task to an accelerator is beneficial. Figure 1 shows that in the case of small number of records and less compute intensive models, the optimal decision is keeping the ML scoring on the CPU to avoid the high accelerator offloading cost (a wrong decision to offload to an accelerator can increase the latency by 10× (Section IV)) . However, as the model complexity or the size of the scoring dataset increases, offloading ML scoring to an accelerator becomes the optimal choice (a wrong decision to not offload to an accelerator can result in 70× lower throughput (Section IV)).
3) Offloading the ML scoring stage of the DBMS ML scoring pipeline to different platforms and comparing the overall performance numbers to find the best-performing hardware accelerator (FPGA and GPU) based on the number of records and model complexity. We found that as the number of records, dataset features, and model sizes increase, it is a better option to offload the scoring to the FPGA. For example, for a random forest model with 128 trees each 10 level deep, with 1 million records for a dataset with large number of features, while the GPU has 16.5× higher throughput than that of the CPU, the FPGA throughput is up to 4.2× more than that of the GPU (and 69.7× that of the CPU). However, for the same number of records, but with a smaller model (1 tree with 10 level) and a dataset with less number of features, the GPU outperforms, with a throughput of 2.3× more than that of the FPGA (Sections IV-C2 and IV-C3).

The rest of the paper is structured as follows. We briefly describe our analytics and model scoring pipeline in Section II. In Section III we discuss the GPU and FPGA accelerators that we considered for our study, along with an overview of the offloading overheads. We present and analyze our experimental results in Section IV and discuss related work in Section V.

II. ML Scoring in SQL Server with Python

In this work, we focus on ML model scoring in SQL Server using its capability of executing users' Python scripts that in turn can use other libraries or utilities that implement the scoring functionality.

Figure 2 shows the high-level architecture of the analytics and scoring pipeline. Users submit analytic queries that invoke custom Python scripts that score ML models, an example of which is included in Figure 3, to SQL Server. SQL Server launches an external Python process to execute the script that involves pre-processing, scoring, post-processing, and returning of results. The scoring can happen either on the CPU (❶), or on a PCIe-attached hardware accelerator (❷).

Figure 3 shows an example query that invokes a stored procedure containing a user-supplied Python script that scores an ONNX model to classify iris flowers. The invoking query specifies the model and dataset to use, both of which are obtained from database tables. The models are stored in serialized binary form, in either an off-the-shelf [24], [25] or custom format. Pre-processing for ML scoring involves deserializing the model, extracting features and getting the data ready for the scoring engine. Other steps include setting up the environment for scoring. The prediction results are saved as a Pandas DataFrame and returned to DBMS.

Users can score any ML model with SQL Server by invoking custom Python scripts. For this study, we focus on tree ensemble models and in particular, on the random forest model, which is one of the top models being used in a wide range of classification and regression applications [26].

Figure 4a illustrates an example decision tree model that consists of decision nodes and leaf nodes. Each decision node in a decision tree of a random forest represents a decision rule, consisting of a comparison attribute and a comparison value (threshold), to choose either the left or right node in the next level. Each leaf node represents a scoring outcome. The parameters of all decision trees in the forest are learned from the training phase. These parameters include the comparison attributes, comparison values of each decision node, and the class labels of each leaf node. During scoring, an example input traverses all trees from the root to a leaf node according to the criteria of decision nodes. In the case of random forest regressor, the final outcome is an average of each tree's prediction, and for a random forest classifier, each tree's classification is combined into a final classification through a majority vote mechanism.

III. Hardware Acceleration

We now present an overview of the PCIe-based GPU and FPGA accelerators that we considered for our study and discuss the overheads of offloading ML model scoring to them.

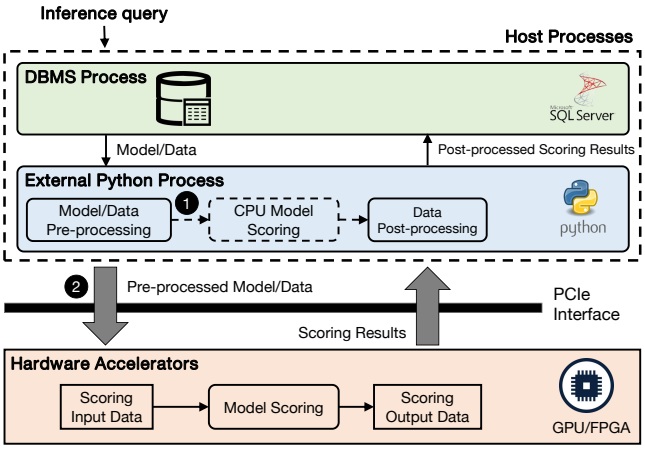
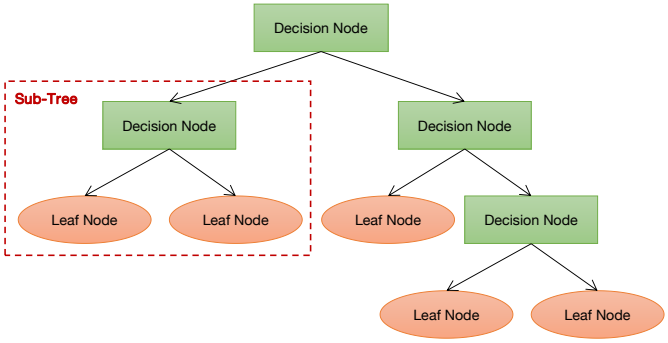


Fig. 2: High-level architecture of ML scoring, using CPUs or hardware accelerators, in SQL Server with Python.

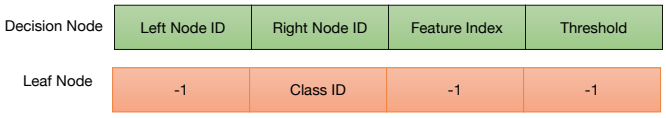
```
CREATE OR ALTER PROCEDURE predict_iris (@model_id int, @input_query nvarchar(max))
AS
BEGIN
DECLARE @model_binary varbinary(max) = (select Data from models where id = @model_id);
EXECUTE sp_execute_external_script
@language = N'Python'
, @script = N'
import onnxruntime as rt
import pandas as pd
sess_opts = rt.SessionOptions()
sess = rt.InferenceSession(model)

features = ["SepalLengthCm" ,"SepalWidthCm", "PetalLengthCm", "PetalWidthCm"]
inputs = data[features].values[:,:].astype("float32")
inputs = {sess.get_inputs()[0].name: inputs[:,0:4]}

nn_output=sess.run(None,inputs)

OutputDataSet=pd.DataFrame(nn_output[0])
'
  @input_data_1=@input_query,
  @input_data_1_name = N'data',
  @params = N'@model varbinary(max)',
  @model = @model_binary
WITH RESULT SETS ((“classid" bigint))
END;
GO

EXEC predict_iris 1,'select * from iris_data'
```

Retrieve model
Import libraries
Setup inference session
Extract features, prepare input data
Score model
Return results
Bind parameters
Specify model and input data set
Query that calls stored procedure

Fig. 3: Example T-SQL query that invokes a stored procedure to score an ONNX model to classify Iris flowers.



(a) An Example of Decision Tree



(b) Decision Tree Nodes Memory Layout

Fig. 4: Decision Tree

A. GPU Accelerator

GPU is a highly parallel multi-core processor that can efficiently perform compute intensive tasks (e.g., ML training and scoring) on large blocks of data. We use two high performance GPU ML libraries, RAPIDS and Hummingbird, to accelerate random forest model scoring on the GPUs.

RAPIDS is a suite of open source libraries that is designed based on different data science libraries and workflows to accelerate ML workloads [27]. Its main components are: (1) cuDF: used to perform data processing tasks (Pandas like API [28]) (2) cuML: used to create GPU-accelerated ML models (Scikit-learn like API) (3) cuGraph: used to perform graphic tasks (Graph-Theory API). cuDF provides a Pandas-like API for dataframe manipulation, so when using RAPIDS for ML scoring, we need to convert the input dataframe to a cuDF dataframe. The cuDF dataframe is fed into a cuML model for scoring. Scoring of random forest models in RAPIDS involves recursively traversing decision tree nodes based on the condition evaluated at each node. Tree layouts in a forest are optimized to improve memory performance, but the strategy is less effective at higher tree depths due to control divergence across trees [29].

Hummingbird [30] uses a different approach — it converts traditional ML models (e.g., decision tree, random forest, and gradient boost models) into tensor computations that can be offloaded to GPUs for acceleration using popular neural network frameworks. It parallelly evaluates multiple nodes and paths in the tree, e.g., using matrix multiplications, instead of doing a traditional sequential traversal, but may do redundant computations in the process.

B. FPGA Accelerator

FPGAs consist of a fabric of programmable logic blocks, specialized resources such as digital signal processors (DSPs), and a collection of small on-chip SRAM arrays (referred to as block RAMs) that can be connected via programmable multiplexers to implement arbitrary logic, computation, and memory. FPGA-based accelerator implementations can use very deep and custom data processing pipelines, so they can perform ML operations with higher efficiency compared to fixed architectures such as CPUs or GPUs [13]–[21].

There is a large degree of parallelism in random forest scoring algorithm that can be exploited in the hardware implementations of a random forest scoring accelerator, to improve the ML scoring performance. The existing parallelism is between input samples, different trees in the forest, and levels of each tree. This causes a high demand for concurrent memory accesses to read tree parameters in a forest. This can be well addressed by the abundant of available memory resources on an FPGA.

To offload random forest scoring to an FPGA-based accelerator, we first need to extract the model parameters and send them to the accelerator to be stored in the FPGA's memory.

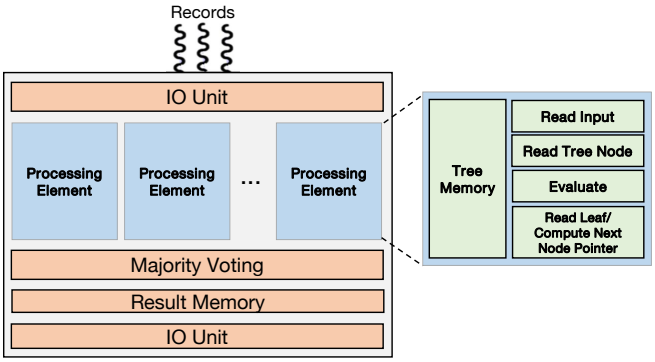


Fig. 5: Inference Engine Architecture

We consider a specific memory layout for each decision and leaf node in the forest as shown in Figure 4b. Each node is represented with four features. First feature of each node defines the node type, a negative value indicates a leaf node, otherwise it is a decision node. If the node is a decision node, then the four features are left node, right node, comparison attribute, and comparison value. Although the only feature required for a leaf node is the output class id, we keep the same memory format for the leaf node to ease the FPGA memory indexing. As the model gets more complex, the number of nodes in the forest increases, and the FPGA memory resources becomes the limiting factor in an FPGA implementation of a random forest inference engine.

Figure 5 shows our proposed random forest inference engine architecture. The inference engine includes 128 processing elements, each processing one of the trees in the forest up to depth 10. The number of processing elements and tree depth that can be processed on the FPGA are limited by the available amount of BRAM and the number of tree nodes that can be stored on the FPGA. Our memory layout assumes a full binary tree with no missing nodes. As a result, each tree consumes a memory footprint equaling 2^{10} words. If the number of trees is greater than 128, we need to call the inference engine multiple times to process all the trees, and may need to store part of the model in FPGA memory, resulting in lower performance. However, note that the default parameter for the number of trees in popular ML frameworks such as Scikit-learn [31] and MADlib [32] is 100. Also, prior works [33] have shown that for different low-density and high-density datasets, model accuracy is hard to improve significantly (below 1%) by increasing the numbers of trees beyond 64.

Our current implementation does not support processing trees with more than 10 levels, they need to be processed by the CPU. An extension to our current design can send the results of processing 10 levels of trees back to the CPU's memory so that the rest of the operation, evaluating levels from depth 10 onward, be done on the CPU.

Before starting the ML scoring, all the model information (tree nodes) are transferred into the tree memory of each processing element. Once the outcomes of all trees are ready, they are passed to the majority voting unit to decide the final outcome. The output class for each input record is stored into the result memory. Once all the input records are processed, the contents of the result memory are transferred back to the CPU through the I/O interface (i.e., PCIe interface). To process multiple inputs, we do not need a separate input transfer phase; our system architecture supports multi-threaded ML scoring contexts with custom PCIe interface and queue managements [34]. We can spawn as many threads as required to process all the input records. Threads are one cycle apart, each thread (input sample) calls the ML scoring engine one cycle after the previous thread. This way we can handle millions of scorings in parallel.

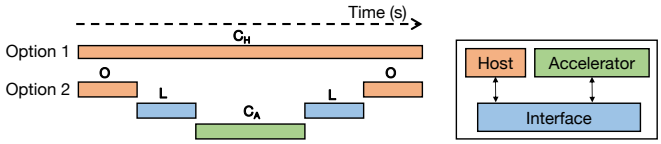


Fig. 6: Accelerator Offload Overhead

C. Accelerator Offload Overheads

Figure 6 shows two different example hardware backends for running an ML model scoring operation. In Option 1, the entire application is run on the host CPU and C_H is the time to complete the scoring on the CPU. In Option 2, the scoring is offloaded to a hardware accelerator. The difference between these two options is that in the case of Option 1, all the data required for scoring is already available in the host (CPU) memory, while in the case of Option 2, the host CPU needs to explicitly transfer the input data and the model to the hardware accelerator's memory and copy back the results. O indicates the host offload overhead (the time spent on configuring the accelerator and setting up the communication link), L is the data transfer overhead, and C_A is the time taken by the accelerator. Although typically $C_A < C_H$, we need to also include the offloading overheads to determine the overall model scoring time (also see Section IV-B), which itself is a part of the end-to-end query time (details in Section IV-D).

IV. Evaluation

In this section we describe our experimental setup, followed by a discussion of the overall model scoring time of FPGAs, and a comparison of the overall scoring time for different backends—FPGA, CPU, and GPU. We then discuss the break-down of the overall end-to-end latency for a T-SQL query for different backends.

A. Experimental Setup

We used two datasets with different number of data features: 1) **IRIS [35]**, which is a multi-class classification dataset with 4 features, 3 classes, and 150 data samples. The IRIS dataset by itself is small, and so we generated 1M data samples by replicating the original samples; 2) **HIGGS [36]**, which is a binary classification dataset with 28 features and 11M data samples. We used a subset of HIGGS dataset for both training

and test. For CPU experiments, we used up to 52 threads on a dual-socket Intel Xeon Platinum 8171M processor with 26 cores (52 threads) per socket running at 2.6 GHz. For FPGA experiments we used Intel Stratix 10 GX 2800 on this machine. For GPU experiments, we used NVIDIA Tesla P100 available in an Azure NC6s_v2-series Virtual Machine.

We used both Scikit-learn [31] and ONNX [24] models for CPU experiments. All the ONNX models are created by converting models from Scikit-learn into ONNX using sklearn-onnx toolkit [37]. For GPU experiments, we used NVIDIA RAPIDS [27] and Microsoft Hummingbird [38] libraries. For our FPGA experiments we implemented random forest inference engine and mapped it to the FPGA implementation, we extract the ONNX model information and transfer it to the tree memories on the FPGA through PCIe 3.0 x16. The FPGA design is clocked at 250 MHz and is programmed only once for all the experiments with different tree ensemble structures.

B. Overall Model Scoring Time on FPGA

The overall model scoring time on FPGA is the round-trip latency corresponding to Python code $\rightarrow$ FPGA scoring $\rightarrow$ Python code, and includes the main time components described below. We focus on the FPGA-based accelerator as an example; use of other accelerator substrates would also include similar components.

1) **Input transfer:** the time spent on transferring the required data for scoring to the FPGA. This data includes model information and the records. In our implementation, there is an overlap between record transfer and scoring operation. Hence, input transfer time is only the time spent on transferring the model information to the processing elements' tree memory. (L component in Figure 6b)
2) **FPGA setup:** the time CPU spends to set up the FPGA connection and call the inference engine (O component in Figure 6b).
3) **Scoring:** the scoring time on the FPGA, i.e., the computation time (C_A component in Figure 6b).
4) **Completion signal:** the time overhead of signaling the task completion from the FPGA back to the CPU through an interrupt (O component in Figure 6b).
5) **Result transfer:** the time spent on transferring the results back to the CPU from the FPGA (L component in Figure 6b).
6) **Software overhead:** the time CPU spends on calling different FPGA functions in the software code (O component in Figure 6b).

We measured the overall FPGA scoring time for different combinations of data examples (i.e., from 1 to 1 million records), model complexities (i.e., 1 tree and 128 trees), and datasets (IRIS and HIGGS). Figure 7a shows the detailed breakdown of the model scoring time using FPGA for 1 record, the different datasets and the different number of trees in the model. As we can see, by increasing the model complexity, i.e, number of trees from 1 to 128 or the number of dataset features (from 4 in IRIS to 28 in HIGGS), the input transfer time increases because we need to transfer larger models to the FPGA. The scoring time also increases for more complex models and higher record counts. Result transfer time is only dependent on the number of records, as we get more records to score, we need to transfer more results back to the CPU from FPGA. However, FPGA setup, completion signal, and software overhead remain the same as they are independent of the model complexity. FPGA setup overhead is less than completion signal overhead because the former one is done by setting Control/Status Registers (CSRs) and latter is done through interrupt. Also we can see that, regardless of model complexity, for the small number of records, input transfer time and the software overhead are the dominant components in the overall model scoring time. Although the scoring itself is in the order of nanoseconds (ns), the overall time is in milliseconds (ms) due to the accelerator offloading overheads.

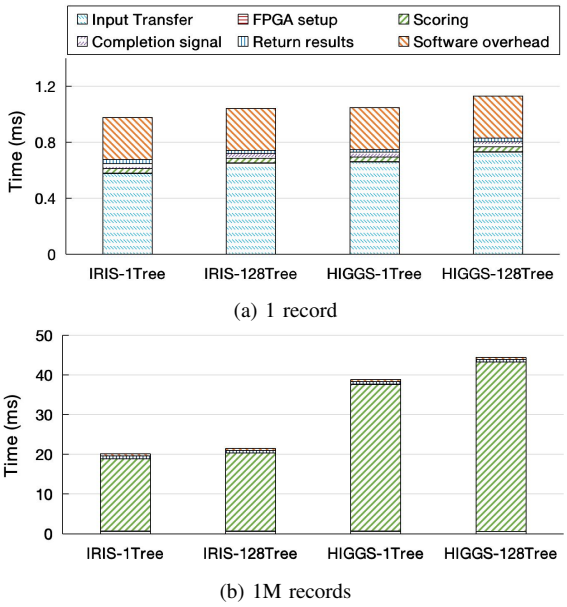
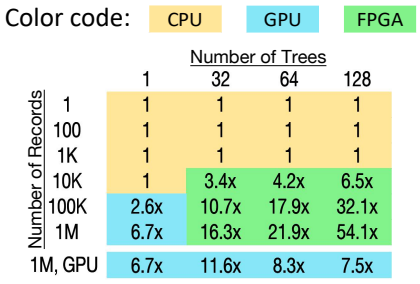


Fig. 7: Overall FPGA model scoring time breakdown as the number of records changes.

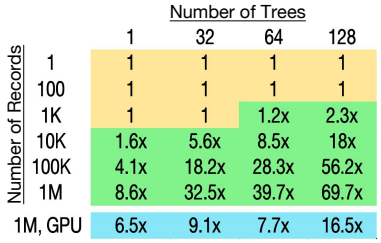
Figure 7b shows the results for 1 million records, the different datasets, and the different number of trees in the model. As we increase the number of records from 1 (in Figure 7a) to 1 million (in Figure 7b), the scoring time (in the order of tens of milliseconds) dominates the overall FPGA model scoring time compared to the offloading components. In fact, FPGA setup, completion signal, and software overheads stay the same. However, the result transfer time increases as we need to transfer more number of scoring results back to the CPU (its increase is trivial compared to scoring time). We can also see that as the model size grows (1 tree to 128 trees), the scoring time increases and offloading cost becomes even more negligible compared to the scoring time and quickly amortizes.

C. Scoring Performance on Different Hardware Backends

In this section, using the latency and throughput of scored records as the performance metrics we compare three different



(a) IRIS



(b) HIGGS

Fig. 8: Speedups of the best-performing hardware over CPU for model scoring (trees are 10 level deep) for IRIS and HIGGS dataset. The bottom row (1M, GPU) shows GPU speedups over CPU for 1M records, for reference.

backends—CPU, GPU, and FPGA for scoring performance. We compute the throughput metric by dividing the total number of records over the overall model scoring time on each hardware backend.

1) Optimal Hardware Backend: In Figure 8, we show the 'shmoo' plot for which backend among CPU, GPU, and FPGA gives us the best performance for a given combination of the number of trees in the random forest model (X-axis) and the number of records (Y-axis) for both IRIS and HIGGS dataset. For each combination we also plot the best achievable speedup as compared to the CPU. For example, for 1M records and 128 trees, the green cell in the plot for the IRIS dataset indicates that the FPGA is the best-performing backend and it runs $>54\times$ faster than the CPU. The bottom-most row for IRIS and HIGGS 'shmoo' plots shows the best GPU performance among GPU-HB and GPU-RAPIDS speedups for 1 million records and different number of trees as compared to CPU.

Figure 8 shows that offloading ML scoring to an accelerator is not always the best choice. For a simple model (small number of trees) or small number of records, i.e., the first three rows in the tables, we don't get any performance benefit from offloading the scoring to an accelerator because the offloading overhead is the dominant time component in the overall model scoring time. However, as the model complexity and/or the number of records increases, by offloading to an accelerator, we can get $>69\times$ speedup in the model scoring time. Another interesting point to be noted in Figure 8 is that given that HIGGS has more data features than IRIS dataset and it generates larger models (more compute-intensive scoring), even with smaller number of records (1K in HIGGS compared to 10K in IRIS), offloading to an accelerator is beneficial.

In terms of which accelerator performed better, i.e. GPU vs. FPGA, we can see that for a random forest with a small model (single tree), for larger record counts, the GPU can perform better than the FPGA for IRIS but not for HIGGS. However, for larger model sizes (tree counts and dataset's features) and record counts, the FPGA becomes faster than the GPU for both IRIS and HIGGS.

In RAPIDS, each thread block on the GPU processes one data sample, and all threads in a block cooperate in computing the prediction for that data sample. For each data sample, the threads load the data sample and the trees are cyclically distributed among the threads. Each thread computes the predictions of the trees assigned to it, and returns the predicted value. At each node in each tree, the result of the condition evaluation determines the next node to consider in that tree. Thus, different threads may follow divergent evaluation paths down the tree, and this may get exacerbated with increasing model complexity.

Hummingbird, on the other hand, converts decision node evaluations into tensor computations that are computed in parallel. An analysis of GPU performance counters with nvprof profiling tool [39] indicates that the average warp execution and SM (Streaming Multiprocessor) efficiencies of most kernels are 100%, or close to that, and much higher than for some kernels with many invocations in RAPIDS. However, there were more instructions executed and more L2/DRAM traffic for Hummingbird. The main contributors to issue stalls for both were memory dependency (data request), execution dependency, and other stalls, with memory dependency stalls usually being the dominant one.

In case of the FPGA, all the required data for scoring (the model and records) are stored in the FPGA on-chip RAM (BRAM) upfront, thus there is no additional memory access overhead. Our FPGA has $\sim$28.6MB BRAM, whereas our GPU has 4MB L2 cache on our GPU. For a random forest with 128 trees and dataset with 1M records, the FPGA runs about $7\times$ (= 54.1/7.5, see Figure 8) and $4.2\times$ (= 69.7/16.5, see Figure 8) faster than the GPU for IRIS and HIGGS datasets respectively.

2) Scoring Latency: **IRIS**: Figure 9 shows the overall scoring latency (in ms) on CPU, GPU, and FPGA for different numbers of records and model complexities for the IRIS and the HIGGS datasets. As we mentioned earlier (Section IV-A), for CPU experiments, we used both ONNX and Scikit-learn models with 1 and 52 CPU threads. Figures 9a and 9b show the scoring time for a random forest model with 1 tree, and 6 and 10 level tree depths, respectively (due to on-chip memory limitation, the maximum tree depth supported on our FPGA implementation is 10). Similarly, Figures 9c and 9d show the scoring time for a random forest with 128 trees, and 6 and 10 level tree depths, respectively. In the case of CPU, for 1 tree and the number of records less than about 5K (using piecewise linear interpolation), ONNX has a lower scoring latency than Scikit-learn, but for a larger number of records, Scikit-learn has a better performance. The reason is that unlike Scikit-learn, ONNX is not currently optimized for batch scoring [30].

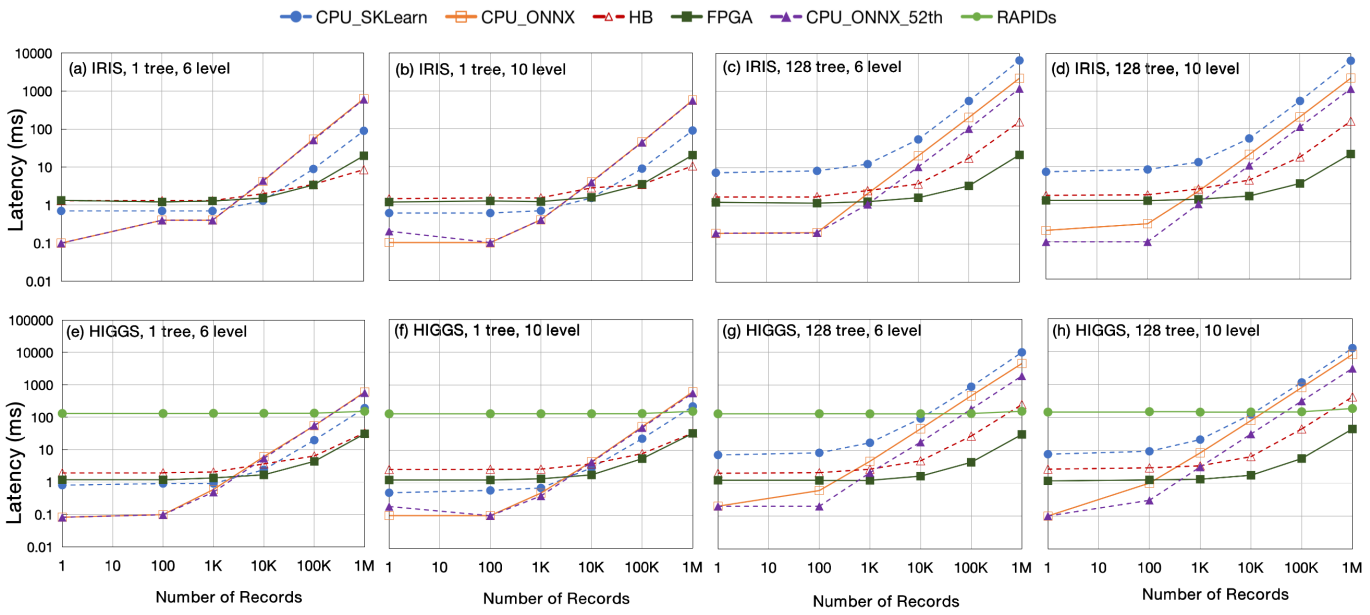


Fig. 9: Scoring latency for models with different tree counts and tree levels for the IRIS and HIGGS datasets running on CPU, GPU, and FPGA. Here CPU_SKLearn = Scikit-learn model running on the CPU with 52 threads, CPU_ONNX = ONNX model running on the CPU with 1 thread, HB/RAPIDS = Hummingbird/RAPIDS model running on the GPU, FPGA = ONNX model running on the FPGA, CPU_ONNX_52th = ONNX model running on the CPU with 52 threads.

Thus, in the rest of our analysis, for each number of records, we select the model with the best performance for the CPU to compare with the GPU and FPGA numbers.

Sensitivity to the number of records: Figure 9 shows that for different models (different number of trees and tree levels), as expected, by increasing the number of records, the scoring latency increases, and it affects the choice of the back-end hardware we should use for scoring. As we can see in Figures 9a and 9b, with a simple model (only 1 tree with different number of levels), if we have less than 10K records, CPU has the lowest latency and is the most suitable hardware for scoring. However, as the number of records increases over 10K, scoring on FPGA or GPU-HB is faster compared to CPU. The reason is that, for large number of records, the scoring time is the dominant element in the overall scoring time (see Figure 7b), which can be greatly accelerated by FPGA and GPU-HB due to their massively parallel computing capabilities and deep computation pipelines. For instance, for 1M records, the FPGA and GPU-HB are up to 2.9× and 6.7× (also see Figure 8, IRIS, 1 tree, 10 level, '1M, GPU') faster than the CPU-SKLearn, respectively. In this case, even if the offloading time is high, the large acceleration in FPGA or GPU-HB results in a reduction in the scoring latency

Sensitivity to the model complexity: In Figures 9c and 9d, the model complexity is increased by increasing the number of trees in the forest (from 1 tree to 128 trees). As we can see, even for a complex model, for less than 1K records, CPU is still the best hardware back-end to run scoring for the small number of records (the same trend as in Figure 9a and Figure 9b with 1 tree). Similarly, as before, for a larger number of records, the FPGA and GPU-HB performance is better than that of the CPU. For example, for 1M records, FPGA and GPU-HB can perform scoring 54× and 7.5× faster than CPU-ONNX-52th (also see Figure 8, IRIS, 128 trees, 10 level, '1M GPU'), respectively. Note that by increasing the number of trees to 128, the crossover point (where the FPGA/GPU-HB performance becomes better than the CPU performance) in Figures 9c and 9d shifts left to 1K records compared to 10K records in Figures 9a and 9b. This is because for a more complex model, the scoring task is compute intensive, and it can be significantly accelerated by FPGA/GPU-HB. Another important takeaway from comparing Figures 9a – 9d is that as the model complexity increases, the best-performing accelerator (FPGA) speedup over the CPU also increases, e.g., for 1M records, it increases from 2.9× (1 tree, 6 level) to 54× (128 trees, 10 level, also see Figure 8, IRIS). This is because CPU is not as good as FPGA in intensive parallel computing due to the its limited capacity to extract data-level parallelism.

Broadly, from Figure 9, as the model gets more complex (more trees and levels) or as the number of records increases, FPGA scoring is faster than GPU and CPU. As we discussed earlier, this could be due to the GPU's high cache misses and memory traffic [40], [41] and CPU's limited parallel computation capacity.

HIGGS: In Figures 9e – 9h, we show the scoring latency for HIGGS dataset, for random forest models with different combinations of number of trees (1 and 128 trees) and tree depths (6 and 10 levels). HIGGS dataset has 28 features and generates a more complex ML model compared to the IRIS dataset. There are only two output classes for this dataset, thus

the model is a binary classifier and is also supported by GPU RAPIDS Library. Hence, for HIGGS, we used both RAPIDS and Hummingbird (HB) libraries for the GPU experiments.

Sensitivity to the dataset features: As we can see in Figures 9e – 9h, with different number of trees, for smaller record counts, both FPGA and GPU are slower than the CPU in terms of the scoring times. However, for larger number of records, the FPGA and GPU are faster than the CPU, e.g., for a combination of 1M records and a forest with 128 trees, GPU-RAPIDS and FPGA scoring times are about 16.5× and 69.7× (also see Figure 8, HIGGS) faster than CPU-ONNX-52th, respectively. Also, as in IRIS, by increasing the number of trees (model complexity) from 1 in Figures 9e and 9f compared to 128 in Figures 9g and 9h, respectively, the GPU-RAPIDS/HB and FPGA speedup increases from (6.5×, 8.6×) to (16.5×, 69.7×). These observations are due to the same reasons explained for the IRIS dataset.

Although we have seen the same trend in the latency graphs for both IRIS and HIGGS datasets, in the case of the IRIS dataset the crossover point is higher for both models with 1 and 128 trees (10K, 1K) compared to HIGGS (5K, 500). The reason is that a dataset with low number of features such as IRIS generates simpler/smaller models than a dataset with more features such as HIGGS. Hence, for a simple model (such as for IRIS), the speedup we get from accelerators over the CPU is smaller and the offloading overhead is not amortized for the small number of records. However, even with smaller number of records, for a complex model (such as for HIGGS) the overhead can be amortized because the accelerator speedup is more significant. Another key observation is that by increasing the number of dataset features, the amount of GPU/FPGA speedup grows. For example, for 128 trees and 1M records, FPGA and GPU-RAPIDS/HB speedups are about 69.7× and 16.5× for HIGGS dataset compared to 54× and 7.5× in IRIS dataset (also see Figure 8). This is because the generated model for HIGGS dataset is more complex than the one generated for IRIS. So we can accelerate scoring more significantly on the accelerators compared to the CPU with its more limited computation capability.

Figures 9e – 9h also show the scoring time when using RAPIDS with GPU. The latency of the RAPIDS model running on the GPU (GPU-RAPIDS) is very high for small number of records. The reason is that GPU-RAPIDS has a separate data pre-processing step to convert the Numpy array to a cuDF data frame, which takes about 120 ms for our input size. Hence, for small number of records and a simple model (1 tree), where the scoring time is less than the pre-processing time, GPU-RAPIDS fails to deliver any speedup. However, for a complex model (128 trees) and more than 700K records, it performs faster than GPU-HB and gets closer to the FPGA as the time required for the pre-processing step gets amortized against the complexity of the model and record size.

3) Scoring Throughput: For small number of records, latency is a good metric for measuring performance, but as the number of records increases, throughput (number of scorings per second) becomes an important performance metric for batch scoring. Figures 10a – 10d show the throughput results in the units of million scoring per second for IRIS dataset and different hardware backends. As we can see in Figure 10, the FPGA/GPU-HB throughput is very low for small number of records, but this throughput increases with an increase in the number of records. The main reason for this is that by increasing the input size, we get higher speedup due to the high parallel computation capacity of the FPGA/GPU-HB, and the accelerator offloading cost gets amortized.

Figure 10a and Figure 10b show that for a large number of records (>10K), the GPU-HB and FPGA have a higher throughput than the CPU-SKLearn (e.g., 6.7× and 2.9×, respectively, for 1M records) due to their massive parallel computing capabilities and speedup over the CPU scoring. As we increase the number of trees to 128 (see Figure 10c and Figure 10d), GPU and FPGA throughput grows significantly over the CPU. As we mentioned earlier, this is because by increasing model complexity, the scoring task becomes more compute intensive, which can be well accelerated by FPGA/GPU over the CPU.

As we increase the number of trees, the FPGA throughput becomes more than all other hardware backends. As the model size or records count increases, the amount of required memory to store the model and records increases, which can result in increased cache misses and memory traffic for the GPU [40], [41]. For our FPGA implementation, we only used the on-chip BRAM and thus avoided the high cost of cache misses and memory accesses.

Figures 10e – 10h show the scoring throughput in million scorings per second for the HIGGS dataset. As we can see in Figures 10e and 10f, both FPGA and GPU-HB throughputs are very close for a simple model with 1 tree. For the record count less than 10K, the CPU throughput is also close to FPGA and GPU-HB throughputs. On the other hand, for a larger number of records (e.g., 1M), due to the massive parallel computing capabilities of the accelerators, FPGA and GPU-HB throughputs are 8.6× and 6.5× more than the CPU-SKLearn's throughput. Also, as the number of trees in the model increases (see Figure 10g and Figure 10h), FPGA throughput becomes significantly higher compared to the other hardware backends. FPGA throughput is about 4.2× (= 69.7/16.5, also see Figure 8, HIGGS, 128 trees) better than GPU-RAPIDS throughput which is due to its deep computation pipelines and low latency BRAM access. With larger models and input sizes, the increase in data footprints can negatively impact cache performance on the GPU [40], [41]. GPUs with larger caches can improve the slopes of the GPU performance curves and shift the crossover points in Figures 9 and 10. For records more than 700K, GPU-RAPIDS throughput starts getting better than GPU-HB. This observation implies that for larger numbers of records the fixed cost of data pre-processing step in GPU-RAPIDS gets amortized better than that in GPU-HB.

D. End-to-End Query Time

Figure 11 shows the expected end-to-end time breakdown for the T-SQL query that includes ML model scoring assuming

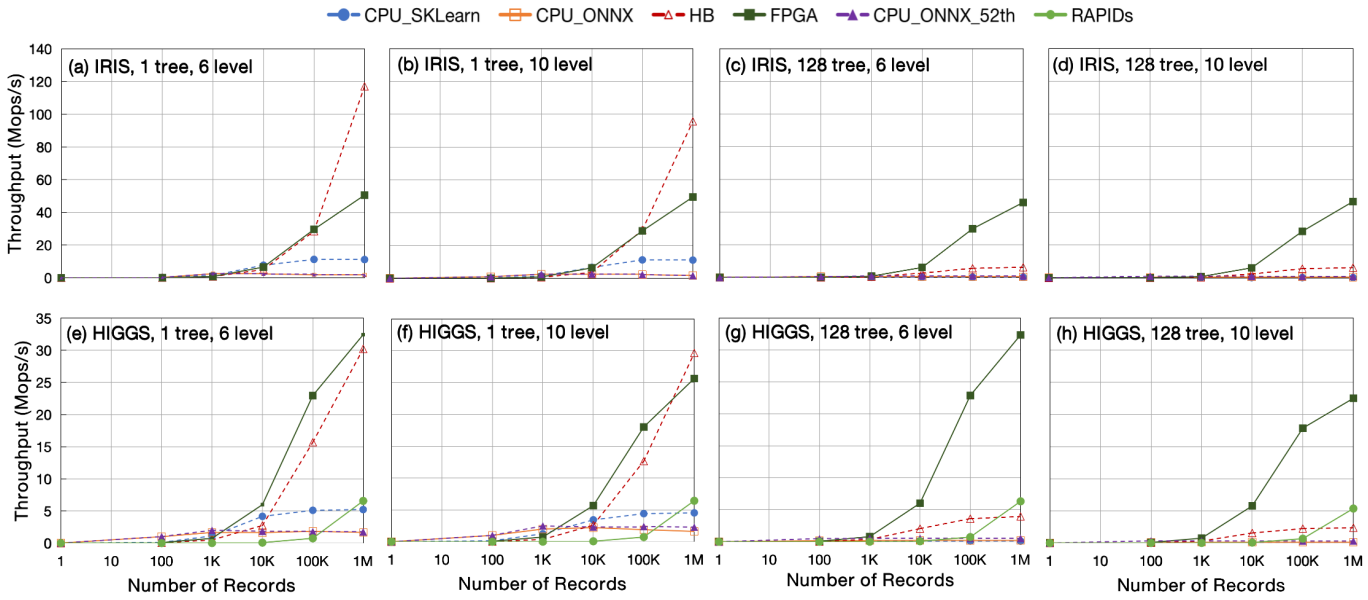


Fig. 10: Scoring throughput for models with different tree counts and tree levels for the IRIS and HIGGS datasets running on CPU, GPU, and FPGA. Here CPU_SKLearn = Scikit-learn model running on the CPU with 52 threads, CPU_ONNX = ONNX model running on the CPU with 1 thread, HB/RAPIDS = Hummingbird/RAPIDS model running on the GPU, FPGA = ONNX model running on the FPGA, CPU_ONNX_52th = ONNX model running on the CPU with 52 threads.

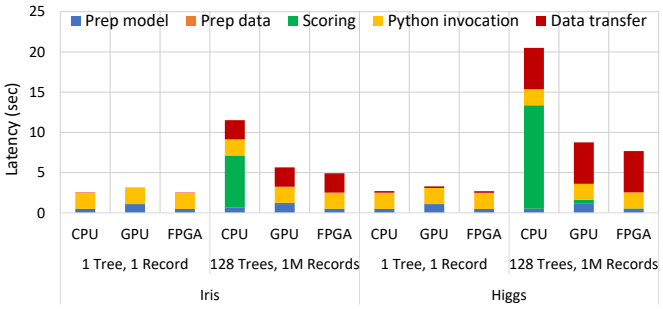


Fig. 11: End-to-end T-SQL query latency breakdown, assuming single-threaded CPU execution.

single-threaded CPU execution. The components of the end-to-end query time are as follows (details in Section II):

- **Model pre-processing:** this includes time taken for deserializing the ML model.
- **Data pre-processing:** this includes time taken for extracting features and preparing the input data for scoring.
- **Model scoring:** this is the overall scoring time on the FPGA (Section IV-B).
- **Python invocation time:** this includes time taken for launching the external Python process.
- **Data transfer time:** this includes time taken for the (transparent) copying of data and results from SQL Server to the external Python process and vice versa.

For all three backends, we observe that for a small model, i.e., small tree count and small number of records, e.g., random forest with one tree and one record, the scoring time is very small and the dominant elements are the Python invocation and model pre-processing times. However, as the number of records and model complexities increase, the scoring time becomes the dominant component in the case of CPU execution. Offloading the scoring step to an GPU/FPGA can significantly decrease the scoring time, and in turn make data transfer time the dominant time component in the overall query time. For example, with 1M records of HIGGS dataset, we expect to see a query speedup of about 2.6×, which is beneficial, but less than for just the ML scoring part due to data transfer and other components of the query processing time.

E. Future research

We point out that there are two sets of overheads that affect the overall T-SQL query time—the hardware backend overheads (setup and data movement), and the application/analytics pipeline overheads. While the former has been considered in prior works [17], [21], [42], [43], the latter has not been well studied so far in the context of accelerating end-to-end analytics and model scoring pipelines. To the best of our knowledge, this paper is the first to study these overheads (see Figure 11). There is another difference between the two types of overheads: the former includes more intrinsic hardware limits (e.g., PCIe bandwidth limits), whereas the latter includes more software-level and customizable overheads in the way that the pipelines have been set up. A tighter integration of the ML scoring functionality within the DBMS would reduce a lot of the application overheads [4], [5], [7], but an external Python invocation could allow for more customizations including in the choice of the ML scoring engine and accelerator. Similarly, the different accelerator integration options—tightly-coupled, loosely coherent/non-coherent coupled, and decoupled, need

to be carefully studied to determine the integration that is best suited for ML scoring within the DBMS. In addition, the traditional techniques—pipelining, parallelism, etc., should be explored when designing the accelerator micro-architecture. Broadly, any future research on performance models, accelerator development, and scheduling should consider both types of overheads for optimal system design and operation.

V. Related Work

ML training/scoring with FPGA acceleration. There is an ever growing popularity in using ML algorithms to extract and process the information from raw data. ML applications require intensive computations and large memory bandwidth, but CPUs fail to achieve desired performance due to their limited resources. As a result, various FPGA-based acceleration solutions have been proposed for ML algorithms. [13]–[20], [44], [45]. FPGAs provide the advantages of high performance, high energy efficiency and reconfigurability.

Unfortunately, most of these FPGA-based ML accelerator designs only focus on the computations on the FPGA and the speedup of the offloaded inference task, and don't consider the communication/offloading overheads between the FPGA and CPU in their performance analysis. In this paper, we look at the complete picture and analyze the performance considering all the time components in end-to-end time of the application.

FPGA acceleration of data processing pipeline. Prior works have explored FPGA utilization to accelerate the compute-intensive stages of pipelined applications and its effects on the end-to-end application time. Owaida et al., [21] integrate FPGA-based inference in the search engine pipeline to accelerate route scoring stage. They report the end-to-end route scoring time on CPU and FPGA with time breakdowns (invocation overhead, data transfer, tree loading, compute, result transfer). They also discuss the scalability of FPGA-based ML inference with regard to model complexity (tree ensemble size) and input request size. The target application in their paper is different from the one investigated in our paper. Their measured FPGA time breakdown is also limited to a single model and dataset dimension, and there is no comparison with GPU performance.

Catapult [44] exploits FPGA acceleration for Bing web search ranking application, which is a large datacenter workload. In this application, query-specific features are generated from documents, processed, and then passed to a machine learned model to determine how relevant the document is to the query. In that work, the FPGA was used to accelerate feature generation stage of the pipeline and not scoring. Brainwave [46] leverages Catapult to accelerate deep neural networks for real-time scoring. The target application investigated in our paper is different and we additionally compare FPGA and CPU performance results with GPU results.

Other prior works [2], [3], [6] extend database engines with FPGAs to accelerate database operators for analytic queries. The target application in these papers is the same as the one studied in our paper. However, they do not measure the end-to-end application time and also FPGA time breakdown for different model complexities and dataset dimensions. Also, there is no comparison with GPU performance. Eryilmaz et al. [43] study FPGA acceleration of aggregation operators, but do not consider machine learning applications in their work.

AI tax. Richins et al. [23] emphasize on the need for the end-to-end performance analysis of AI workloads and discuss the need to carefully account for the AI tax. They find that storage and network bandwidth become major bottlenecks with increasing AI acceleration. Hence, evaluating standalone AI accelerator without considering the full application environment is misleading. They report the breakdown of end-to-end inference time including pre-processing, inference, and post-processing times. However, they do not report the scoring time breakdown for different hardware accelerators such as FPGAs and GPUs. They only emulate an AI accelerator inference time, by calls to sleep function and dividing sleep times by the speedup factor, to find the impact of faster AI on the data center and on the workload as a whole.

VI. Conclusion

In this paper, we studied the speedups and overheads for CPU, GPU and FPGA acceleration of random forest models, as part of analytic query processing in Microsoft SQL Server with its external Python process execution capability. The benefits of offloading ML scoring to accelerators depends on the hardware backend, model complexity, and data size as well as on how closely the ML scoring pipeline is integrated with the DBMS. Broadly, for models with lower complexity and for smaller data sizes, it is preferable to use the CPU and not a dedicated accelerator for scoring, while for models with higher complexity and large data sizes it is better to use FPGA-based accelerator for ML scoring. Future research on performance models should account for application and pipeline overheads, in addition to accelerator overheads, for forecasting cost-benefit tradeoffs with acceleration.

VII. Acknowledgments

We thank Blake Pelton and Rajas Karandikar for FPGA setup and tooling advice, and Matteo Interlandi and Karla Saur for Hummingbird usage advice. Zahra did this research work during a summer internship at the Microsoft Gray Systems Lab (GSL). We thank Carlo Curino, GSL team members, and anonymous reviewers for their valuable feedback on this work.

References

[1] R. Kune, P. K. Konugurthi, A. Agarwal, R. R. Chillarige, and R. Buyya, "The anatomy of big data computing," *Software: Practice and Experience*, vol. 46, no. 1, pp. 79–105, 2016.

[2] D. Mahajan, J. K. Kim, J. Sacks, A. Ardalan, A. Kumar, and H. Esmaeilzadeh, "In-RDBMS hardware acceleration of advanced analytics," *arXiv preprint arXiv:1801.06027*, 2018.

[3] D. Sidler, Z. István, M. Owaida, K. Kara, and G. Alonso, "doppiodb: A hardware accelerated database," in *Proceedings of the 2017 ACM International Conference on Management of Data*, 2017, pp. 1659–1662.

[4] A. Agrawal, R. Chatterjee, C. Curino, A. Floratou, N. Godwal, M. Interlandi, A. Jindal, K. Karanasos, S. Krishnan, B. Kroth, J. Leeka, K. Park, H. Patel, O. Poppe, F. Psallidas, R. Ramakrishnan, A. Roy, K. Saur, R. Sen, M. Weimer, T. Wright, and Y. Zhu, "Cloudy with high chance of DBMS: a 10-year prediction for enterprise-grade ML," in *Proceedings of Conference on Innovative Data Systems Research (CIDR)*, 2020.

[5] 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 *Proceedings of Conference on Innovative Data Systems Research (CIDR)*, 2020.

[6] G. Alonso, Z. Istvan, K. Kara, M. Owaida, and D. Sidler, "doppiodb 1.0: Machine learning inside a relational engine." *IEEE Data Engineering Bulletin*, vol. 42, no. 2, pp. 19–31, 2019.

[7] "Native scoring with PREDICT statement in SQL Server," http://docs.microsoft.com/en-us/sql/t-sql/queries/predict-transact-sql?view=sql-server-2017, 2019.

[8] Redshift ML. [Online]. Available: https://aws.amazon.com/blogs/big-data/create-train-and-deploy-machine-learning-models-in-amazon-redshift-using-sql-with-amazon-redshift-ml

[9] Big Query ML. [Online]. Available: https://cloud.google.com/bigquery-ml/docs

[10] "SQL machine learning documentation," https://docs.microsoft.com/en-us/sql/machine-learning/?view=sql-server-ver15.

[11] "Gartner Report on Analytics," gartner.com/it/page.jsp?id=1971516.

[12] "SAS Report on Analytics," sas.com/reg/wp/corp/23876.

[13] 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)*, 2017, pp. 1–8.

[14] M. Owaida, A. Kulkarni, and G. Alonso, "Distributed inference over decision tree ensembles on clusters of FPGAs," *ACM Transactions on Reconfigurable Technology and Systems*, vol. 12, no. 4, Sep. 2019.

[15] F. Amato, M. Barbareschi, V. Casola, and A. Mazzeo, "An FPGA-based smart classifier for decision support systems," in *Intelligent Distributed Computing VII*, F. Zavoral, J. J. Jung, and C. Badica, Eds. Cham: Springer International Publishing, 2014, pp. 289–299.

[16] B. Van Essen, C. Macaraeg, M. Gokhale, and R. Prenger, "Accelerating a random forest classifier: Multi-core, GP-GPU, or FPGA?" in *2012 IEEE 20th International Symposium on Field-Programmable Custom Computing Machines*, 2012, pp. 232–239.

[17] 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)*, 2017, pp. 1–8.

[18] A. Elkanishy, D. T. Rivera, P. M. Furth, A. A. Badawy, Y. Aly, and C. P. Michael, "FPGA-accelerated decision tree classifier for real-time supervision of Bluetooth SoC," in *2019 International Conference on ReConFigurable Computing and FPGAs (ReConFig)*, 2019, pp. 1–5.

[19] J. Oberg, K. Eguro, R. Bittner, and A. Forin, "Random decision tree body part recognition using FPGAs," in *22nd International Conference on Field Programmable Logic and Applications (FPL)*, 2012, pp. 330–337.

[20] Y. R. Qu and V. K. Prasanna, "Scalable and dynamically updatable lookup engine for decision-trees on FPGA," in *2014 IEEE High Performance Extreme Computing Conference (HPEC)*, 2014, pp. 1–6.

[21] M. Owaida, G. Alonso, L. Fogliarini, A. Hock-Koon, and P.-E. Melet, "Lowering the latency of data processing pipelines through FPGA based hardware acceleration," *Proceedings of the Very Large Data Base Endowment*, vol. 13, no. 1, p. 71–85, Sep. 2019.

[22] N. P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers *et al.*, "In-datacenter performance analysis of a tensor processing unit," in *Proceedings of the 44th Annual International Symposium on Computer Architecture*, 2017, pp. 1–12.

[23] D. Richins, D. Doshi, M. Blackmore, A. T. Nair, N. Pathapati, A. Patel, B. Daguman, D. Dobrijalowski, R. Illikkal, K. Long, D. Zimmerman, and V. J. Reddi, "AI tax: The hidden cost of AI data center applications," *arXiv preprint arXiv:2007.10571*.

[24] J. Bai, F. Lu, K. Zhang *et al.*, "ONNX: Open Neural Network Exchange," https://github.com/onnx/onnx, 2019.

[25] "pickle — Python object serialization." [Online]. Available: https://docs.python.org/3/library/pickle.html

[26] 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," *CoRR*, 2019. [Online]. Available: http://arxiv.org/abs/1912.09536

[27] S. Raschka, J. Patterson, and C. Nolet, "Machine learning in Python: Main developments and technology trends in data science, machine learning, and artificial intelligence," *arXiv preprint arXiv:2002.04803*, 2020.

[28] W. McKinney, "Data structures for statistical computing in Python," in *Proceedings of the 9th Python in Science Conference*, S. van der Walt and J. Millman, Eds., 2010, pp. 51 – 56.

[29] J. Zedlewski, "RAPIDS forest inference library: Prediction at 100 million rows per second," 2019. [Online]. Available: https://medium.com/rapids-ai/rapids-forest-inference-library-prediction-at-100-million-rows-per-second-19558890bc35

[30] S. Nakandala, K. Saur, G.-I. Yu, K. Karanasos, C. Curino, M. Weimer, and M. Interlandi, "A tensor compiler for unified machine learning prediction serving," in *14th USENIX Symposium on Operating Systems Design and Implementation(OSDI)*, 2020.

[31] 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," *Journal of Machine Learning Research*, vol. 12, pp. 2825–2830, 2011.

[32] J. M. Hellerstein, C. Ré, F. Schoppmann, D. Z. Wang, E. Fratkin, A. Gorajek, K. S. Ng, C. Welton, X. Feng, K. Li, and A. Kumar, "The MADlib analytics library: Or MAD skills, the SQL," *Proceedings of the Very Large Data Base Endowment*, vol. 5, no. 12, p. 1700–1711, Aug. 2012.

[33] T. M. Oshiro, P. S. Perez, and J. A. Baranauskas, "How many trees in a random forest?" in *Proceedings of the 8th International Conference on Machine Learning and Data Mining in Pattern Recognition*, ser. MLDM'12. Berlin, Heidelberg: Springer-Verlag, 2012, p. 154–168.

[34] M. S. Riazi, K. Laine, B. Pelton, and W. Dai, "Heax: An architecture for computing on encrypted data," in *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2020, pp. 1295–1309.

[35] D. Dua and C. Graff, "UCI machine learning repository," 2017. [Online]. Available: http://archive.ics.uci.edu/ml

[36] P. Baldi, P. Sadowski, and D. Whiteson, "Searching for exotic particles in high-energy physics with deep learning," *Nature communications*, vol. 5, no. 1, pp. 1–9, 2014.

[37] sklearn-onnx: Convert your scikit-learn model into onnx. [Online]. Available: http://onnx.ai/sklearn-onnx/

[38] S. Nakandala, K. Saur, G. Yu, K. Karanasos, C. Curino, M. Weimer, and M. Interland, "Taming model serving complexity, performance and cost: A compilation to tensor computations approach."

[39] Nvidia profiling tools. [Online]. Available: https://docs.nvidia.com/cuda/profiler-users-guide/index.html#nvprof-overview

[40] J. Browne, D. Mhembere, T. M. Tomita, J. T. Vogelstein, and R. Burns, "Forest packing: Fast parallel, decision forests," in *Proceedings of the 2019 SIAM International Conference on Data Mining*, 2019, pp. 46–54.

[41] N. Asadi, J. Lin, and A. P. De Vries, "Runtime optimizations for tree-based machine learning models," *IEEE transactions on Knowledge and Data Engineering*, vol. 26, no. 9, pp. 2281–2292, 2013.

[42] M. S. B. Altaf and D. A. Wood, "LogCA: A high-level performance model for hardware accelerators," in *Proceedings of the 44th Annual International Symposium on Computer Architecture*, ser. ISCA '17. Association for Computing Machinery, 2017, p. 375–388.

[43] Z. F. Eryilmaz, A. Kakaraparthy, J. M. Patel, R. Sen, and K. Park, "FPGA for aggregate processing: The good, the bad, and the ugly," in *International Conference on Data Engineering (ICDE)*, 2021.

[44] D. Chiou, "The Microsoft Catapult project," in *2017 IEEE International Symposium on Workload Characterization (IISWC)*, 2017, pp. 124–124.

[45] A. M. Caulfield, E. S. Chung, A. Putnam, H. Angepat, J. Fowers, M. Haselman, S. Heil, M. Humphrey, P. Kaur, J. Kim, D. Lo, T. Massengill, K. Ovtcharov, M. Papamichael, L. Woods, S. Lanka, D. Chiou, and D. Burger, "A cloud-scale acceleration architecture," in *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2016, pp. 1–13.

[46] J. Fowers, K. Ovtcharov, M. Papamichael, T. Massengill, M. Liu, D. Lo, S. Alkalay, M. Haselman, L. Adams, M. Ghandi, S. Heil, P. Patel, A. Sapek, G. Weisz, L. Woods, S. Lanka, S. K. Reinhardt, A. M. Caulfield, E. S. Chung, and D. Burger, "A configurable cloud-scale DNN processor for real-time AI," in *Proceedings of the 45th Annual International Symposium on Computer Architecture*, ser. ISCA '18. IEEE Press, 2018, p. 1–14.