Lowering the Latency of Data Processing Pipelines Through FPGA based Hardware Acceleration

Muhsen Owaida Gustavo Alonso
Systems Group
Dept. of Computer Science, ETH Zurich
Zurich, Switzerland
firstname.lastname@inf.ethz.ch

Laura Fogliarini Anthony Hock-Koon
Pierre-Etienne Melet
Amadeus
Sophia-Antipolis, France
firstname.lastname@amadeus.com

ABSTRACT

Web search engines often involve a complex pipeline of processing stages including computing, scoring, and ranking potential answers plus returning the sorted results. The latency of such pipelines can be improved by minimizing data movement, making stages faster, and merging stages. The throughput is determined by the stage with the smallest capacity and it can be improved by allocating enough parallel resources to each stage. In this paper we explore the possibility of employing hardware acceleration (an FPGA) as a way to improve the overall performance when computing answers to search queries. With a real use case as a baseline and motivation, we focus on accelerating the scoring function implemented as a decision tree ensemble, a common approach to scoring and classification in search systems. Our solution uses a novel decision tree ensemble implementation on an FPGA to: 1) increase the number of entries that can be scored per unit of time, and 2) provide a compact implementation that can be combined with previous stages. The resulting system, tested in Amazon F1 instances, significantly improves the quality of the search results and improves performance by two orders of magnitude over the existing CPU based solution.

PVLDB Reference Format:
Muhsen Owaida, Gustavo Alonso, Laura Fogliarini,Anthony Hock Koon, and Pierre-Etienne Melet. Lowering the Latency of Data Processing Pipelines Through FPGA based Hardware Acceleration. *PVLDB*, 13(1): xxxx-yyyy, 2019.
DOI: https://doi.org/10.14778/3357377.3357383

1. INTRODUCTION

Response time is a critical metric in interactive systems such as search engines, payment platforms, or recommender systems (e.g., a flight or hotel reservation system), where the latency in responding to a search query determines the quality of the user experience [8, 10, 28, 1]. Each of these systems has to complete a different task: search engines compute potential matches to the search query, recommender systems identify the user's profile and activity to make suggestions as the user is browsing, while online payment systems detect fraudulent transactions before accepting payments. Nevertheless, the architecture used, whether on-premise installations or in the cloud, is similar in all cases: it involves a variety of databases and a pipeline of data processing stages where each stage runs on a cluster of machines [18].

The most common performance metric for interactive systems is a latency upper-bound given a throughput lower-bound (e.g., processing at least 1000 request per second with a 95-percentile response time of 1 second). To meet throughput constraints (as well as elasticity and fault tolerance), query processing is done in parallel across several machines. To meet latency requirements, the response time of the data processing stages must be predictable, which often implies limiting their scope or the amount of data processed to make sure the resulting response time does not exceed the available budget for the stage [58]. In practice, there is a trade-off latency vs. throughput. Separating stages allows to devote parallel resources to each one of them (improving throughput). On the other hand, merging stages into a single step tends to improve latency by cutting down on network transfers and data movement but reduces the capacity of each stage due to the higher processing cost per query.

In the past, these pipelines involved mostly data intensive operations (e.g., queries over databases). Nowadays, there is a widespread use of machine-learning classifiers and predictors, which are computationally heavier, as they have the ability to improve interactive systems in many ways. Thus, existing recommender systems [6, 55] and search engines [12] such as those used by Amazon, Netflix, LinkedIn, or YouTube, to mention a few, all employ machine-learning within their data processing pipelines. Similarly, the payment systems associated to the same platforms rely as well on machine learning for, e.g., detecting bots or fraudulent transactions [62, 4].

Using a real use case as motivation and baseline, in this paper we explore the interplay of data processing and machine learning in search engines pipelines. We give examples of how the latency constraints restrict the amount of data that can be processed, thereby reducing the quality of the results. We also show the problems associated to improving only query throughput by scaling out the search engine. We then provide a novel solution to flight route scoring through an FPGA-based implementation of gradient-boosted decision tree ensembles [35, 13] that allows processing a larger number of potential results and enables the merging of stages, thereby reducing the overall latency without compromising the system's throughput.

Proceedings of the VLDB Endowment, Vol. 13, No. 1
ISSN 2150-8097.
DOI: https://doi.org/10.14778/3357377.3357383

The contributions of the paper are as follows:

(1) We discuss with a concrete example the latency vs. throughput trade-off common to many data processing pipelines. This is a complex system issue since, as the paper illustrates, improving query throughput is not enough to improve the overall system performance due to the very tight latency requirements and the query volume faced by modern domain search engines.

(2) We identify FPGA-based accelerators as the most suitable processing devices for tackling the trade-off and develop a novel implementation of inference over decision tree ensembles using an FPGA. The design generalizes the state of the art to more complex predicate comparisons on the decision nodes and wider data. The resulting system is two orders of magnitude faster than existing CPU-based systems.

(3) We evaluate the resulting system on a variety of platforms – PCIe connected FPGAs [33, 46], cache coherent FPGAs [37], and Amazon F1 instances [7] – to explore deployments over the typical configurations made available by cloud providers. We also discuss in detail the effects of memory, computing, and I/O bandwidth on the resulting system. This analysis should help to project the results obtained into future systems as the hardware involved is rapidly changing and gaining in both capabilities and capacity.

(4) We break down the cost of the proposed design for the entire use case, including the overheads of starting the computation, data transfers, and memory capacity. Based on this information, we analyze in depth the advantages as well as the potential limitations of the solution we propose.

2. BACKGROUND

2.1 Decision Trees (DT)

Decision trees have been used in the past for, e.g., data mining [21, 22], and are nowadays widely used for a range of machine-learning tasks such as classification (results are distinct labels) and regression (results are values). Implementations range from libraries as part of ML packages (H2O [16], SciKit [44], or XGBoost [13]), large-scale distributed implementations for cloud platforms [43], to database operators such as those available in Oracle Data Mining [38]. While here we focus on data pipelines for search engines, our solution can be applied as-is in the context of databases or data warehouses that use FPGA acceleration [54, 3, 32, 26, 59].

Decisions trees are constructed by recursively splitting the data into groups. The split is usually a greedy algorithm that considers all possible splits and decides which one is best at each step using some cost function determining the accuracy of each possible split. The recursion is stopped by using heuristics such as limiting the depth of the tree or enforcing a minimum number of elements per leaf of the tree. Stopping the recursion correctly is important to avoid overfitting. Typically, a decision tree model is trained on input data annotated with labels (class or prediction) and then tested on a second set of input data for model validation. Inference, the part we explore in this paper, is performed by traversing the tree using the data of the tuple that is to be classified.

Decision trees have several advantages over other machine-learning techniques. They have an intuitive representation that helps with interpretability and they do not require parameter setting as many other ML methods do (thus reducing the effort in data preparation). They can handle categorical and numerical data as well as multiple outputs. Their two most important limitations are variance (the tree changes substantially with small changes in the training set) and locking on local optima as a result of the greedy search.

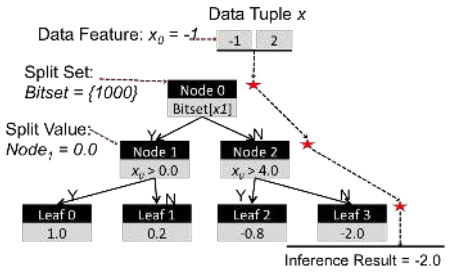


Figure 1: An Example of a Decision Tree. Nodes 0 - 2 are decision nodes and nodes 3 - 6 are leaf nodes.

2.2 Gradient Boosted Decision Trees (GBDT)

The most common form of decision tree models used nowadays are Gradient-Boosting Decision Trees (GBDT), a supervised decision tree ensemble method [35]. The use of ensembles (many trees) addresses the problem of local optima by generating multiple trees with random samples of the input. The use of gradient boosting addressed the problem of variance. Stated simply, GBDT builds a strong learner out of an ensemble of weak learners. A model in GBDT contains K decision trees. To run inference, it first runs inference over all K trees independently, and then combines the inference results. GDBT tends to favor shallow trees without many levels but requires many trees (hundreds to thousands).

In this paper, we use a GBDT model generated with H2O, an open-source platform for big data analytics [16] as it is the system used in production for our use case. We use H2O for training and generating the GBDT model. However, when deploying the model in production, H2O GBDT inference was re-implemented in C++ as the original version of H2O uses Java (the nature of the changes made is detailed later in the paper).

Figure 1 shows an example of a binary decision tree as represented in a H2O Gradient Boosted Machines (GBM), which is an implementation of a GBDT. Each non-leaf node is called a *decision node*. Each decision node defines an operation to choose either the left or right node in the next level. Each leaf node contains the classification or regression result. In GBM trees, two types of operations are used in decision nodes:

- Split value comparison. A decision node performs a comparison operation between a particular data tuple feature and a split value (i.e., floating-point number) to choose either the left or the right branch.
- Split set membership test. A decision node defines a split set, on which it performs membership test for an input data tuple feature. This test is more appropriate for categorical features. However, in GBM trees, a split set might grow to hundreds of bytes, leading to trees with a large memory footprint.

2.3 FPGAs as Accelerators

A field-programmable gate array (FPGA) is an integrated circuit that consists of a matrix of configurable logic, memory, and digital signal processing (DSP) components (Figure 2 [49]). These components are distributed within a grid of configurable routing wires connected to programmable chip I/O blocks. This flexible and programmable fabric can be configured to perform any functionality implemented as a digital circuit (e.g., finite state automate [34], deep learning [15], or data management systems [30], etc.).

FPGAs are programmed using hardware description languages (HDLs) such as VHDL or Verilog. There are also frameworks to use common software programming languages such as OpenCL for Intel FPGAs [24] or C/C++ for Xilinx FPGAs [61]. Recent efforts try to use domain-specific languages for FPGAs [29].

Programs are compiled into FPGA binaries, called bitstreams. The HDL compiler first translates the program statements into a netlist of FPGA primitive components (memory, logic, arithmetic, etc.). It then assigns the netlist to physical components in the FPGA fabric, determining which routing wires should be used to connect them. This latter process is called *Place&Route* and can be very time consuming (often hours for complex designs).

The programming model for FPGAs is quite different from the one for CPUs. Computations are laid out spatially and the programmer has to specify how data and control flows from one logic block to another inside the datapath. Thus, common design challenges when developing algorithms for FPGAs is the amount of space (resources) required and the ability to meet timing (ensuring the data can be moved across the circuit in a correct manner). It must also be noted that the clock rate on an FPGA is 10 to 20 times slower than that of CPUs (150–400MHz vs 2.5–4 GHz). Thus, an FPGA design must be far more efficient than the CPU counterpart to be competitive, something that requires completely different algorithms than those used on conventional processors. One important contribution of this paper is the implementation of a GBDT solution that is significantly faster than those available for CPUs and more general than those available for FPGAs.

2.4 Data Processing with FPGAs

There is a substantial amount of work on using FPGAs for data processing, from databases [56, 59, 26, 32] and cloud systems [14, 11, 2] to machine-learning [15, 23, 42, 25]. The vast body of existing research has focused on the compute capacity of an FPGA device, ignoring key performance overheads such as the data transfer cost between host and FPGA. Another aspect often ignored is the initiation cost of an FPGA accelerator (to start the processing), typically larger than that of a function call on the CPU. Later in the paper, we look in depth at these factors when considering the feasibility of the solution we propose.

3. A FLIGHT SEARCH ENGINE

The starting point is an existing search engine provided by Amadeus. Amadeus is an IT provider covering services such as search, pricing, booking, or ticketing for travel services providers along with automated solutions for applications such as inventory management or departure control. The Flight Search Engine is a service used to recommend flights to sites implementing travel-booking functionality.

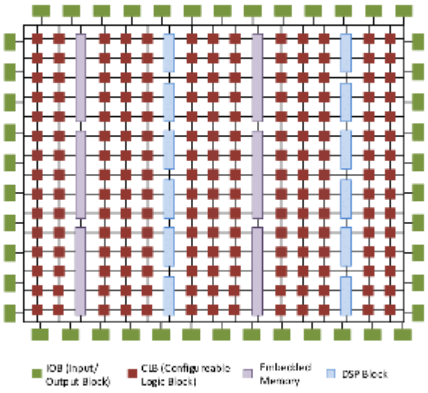


Figure 2: An illustration of a modern Xilinx FPGA architecture showing the different components and their organization in the FPGA fabric.

3.1 Looking for Flights

The Flight Availability Search and Pricing Engine tries to find the cheapest routes between a source and a destination. Given a basic search query (e.g., flight from Zurich (ZRH) in Switzerland to Columbia (CAE) in South Carolina, USA) on a particular date, the engine returns a list of potential routes and their prices. An important quality criterion in such a system is the *look-to-book* ratio, which depends on how good the proposed routes are. Figure 3 shows the different logical components of the engine:

- The Domain Explorer searches the flight domain (available flights from all airlines) and selects up to 750 routes both ways.
- The Route Selection reduces the 750 routes to 150 based on which ones are most likely to be the cheapest.
- The Pricing Engine determines the actual price for the 150 proposed routes.

The bounds on the number of routes considered at each stage arise from the throughput vs. latency trade-off mentioned in the introduction. Details are provided below. The complete search and pricing process must be completed in under 4 seconds.

The flight domain is very large as it consists of all possible combinations of connecting points (airports), carriers (airlines), and flight numbers from source to destination. An example would be a Zurich-Amsterdam-Atlanta-Columbia route, with KLM from Zurich to Amsterdam, then Delta from Amsterdam to Atlanta and to Columbia, plus the time

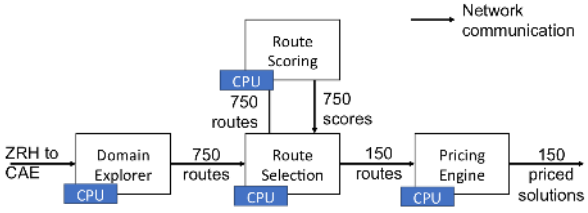


Figure 3: The Flight Availability Search and Pricing engine.

of each particular flight (since there could be several flights from A to B the same day). Depending on the complexity of the request, the flight domain can have millions of combinations. Since selecting and scoring a route is an expensive step, the *Domain Explorer* is limited to providing only 750 routes, making it simpler for the later stages in the pipeline to finish within the allocated time budget.

The *Route Selection* serves two purposes. First, it selects the cheapest routes among those proposed by the *Domain Explorer*. Second, it ensures a proper distribution of the routes among all possible airline and airport combinations (e.g., avoiding biases in the answers favoring one particular carrier or hub). The selection processes of both the *Domain Explorer* and the *Route Selection* are heavily based on domain expert heuristics considering factors such as route length, travel duration, or connection time.

The *Pricing Engine* finally takes the selected routes and calculates the price of the corresponding ticket. This is also an expensive operation as it involves many queries across a number of systems to find out the prices for each leg and the combination of prices offered by each carrier.

3.2 Baseline Route Scoring

The heuristics-based approach to selection in the *Domain Explorer* and the *Route Selection* often induces drastic and somewhat arbitrary cuts to the domain of solutions considered. Such cuts affect the quality of the results. *Route Scoring* (shown in Figure 3) was added to the pipeline to improve the situation.

Route Scoring uses machine-learning to determine which of the proposed routes is most likely to be among the cheapest ones. The baseline implementation for the module uses the H2O framework on production requests to generate a GBDT model. GBDTs were chosen for three main reasons: a predominance of categorical features; the high likelihood of missing features; and a fast training cycle (new models can be generated and tested about every week). The training phase is based on production data.

The *Route Scoring* baseline implementation running on several multicore servers is designed to introduce at most an additional ten milliseconds of latency to Route Selection. The *Route Scoring* module is deployed on its own set of servers, separated from those for *Route Selection*. To ensure the necessary performance, H2O, which is Java-based, was re-implemented in C++. The code was optimized through multiple iterations of profiling and stress testing (see below for more details).

The 56-core CPU servers used to run the *Route Scoring* module can handle around 1.5M routes per second, i.e., 20–30K routes per core. A total of ten servers (including redundancy servers) are dedicated to the *Route Scoring* to handle the entire *Route Selection* traffic. For a single flight availability query, the *Route Selection* component handles a couple of hundreds of routes. This is the same order of magnitude than the baseline *Route Scoring* can reach: hundreds of routes scored in 10 ms in a single process.

To put the importance of route scoring into perspective, the use of the *Route Scoring* module increases the *findability* (the capacity to find the cheapest flights) by approximately 10%. In the flight industry, a higher findability usually translates into more bookings and more traffic from aggregators.

3.3 Problem Statement

Both the initial pipeline and the one augmented with route scoring are limited in the number of routes they can consider because looking at too many routes per query would impact latency. Moreover, since the engine has to serve a large amount of queries per second, a higher cost per query would also be detrimental to the overall throughput. Maintaining a high throughput for heavier queries would imply committing many more servers to the *Route Scoring* module, increasing the overall costs of the Flight Search Engine.

Nevertheless, improving the quality of the results requires to consider more routes per query. To avoid the subsequent impact on latency, the route scoring should be merged with other data processing stages. Doing so would lower the response time as it would eliminate, in some part of the system, the communication over the network. To be amenable to embedding and also increase the number of routes considered per query, the *Route Scoring* has to be able to evaluate hundreds of thousands of routes in a few milliseconds for each flight availability query. In practical terms, the problem to address is to implement the GBDTs system in the same servers as the *Domain Explorer* while providing higher capacity and staying within the given latency bounds.

4. FPGA-GBDT INFERENCE ENGINE

4.1 Overview

The FPGA-GBDT inference engine we propose has been implemented over models generated by H2O. Unlike previous work, we address the problem of implementing complex decision nodes and provide a more compact implementation. The decision nodes in GBM trees use different decision operations compared to, e.g., XGBoost trees and tend to consume much more memory. This makes the trees in GBM larger and more complex than those considered before; as a result, the memory structures used to store the trees had to be modified and tailored to the GBM trees format to efficiently utilize the FPGA's on-chip memory.

The design achieves a high performance by using two techniques: first, it parallelizes the processing of both the trees in the ensemble and the routes data. Second, it eliminates the overhead of the high-rate memory accesses suffered on the CPU by implementing novel, highly distributed, and customized memory structures in the FPGA to store both the trees and the routes data.

4.2 Inference Engine Architecture

Figure 4a shows the inference engine architecture. The design is built out of many *Processing Elements* (PE) organized in *Compute Units* (CU) as illustrated in Figure 4b. A processing element can be programmed at runtime to process one or more decision trees in parallel. A *Compute Unit* consists of 28 PEs, and a tree of single precision floating-point adders that sums up the individual decision tree results produced from the 28 PEs. A *Compute Unit* is designed to process a complete decision tree ensemble. Hence, multiple *Compute Units* are allocated to parallelize the scoring of different routes. The *Collector Unit* then gathers individual route scoring results and writes them back to the FPGA memory.

The *Distributor* is responsible for first replicating the tree ensemble to all CUs, then distributing incoming routes data in a round-robin fashion to CUs for processing. The *I/O*

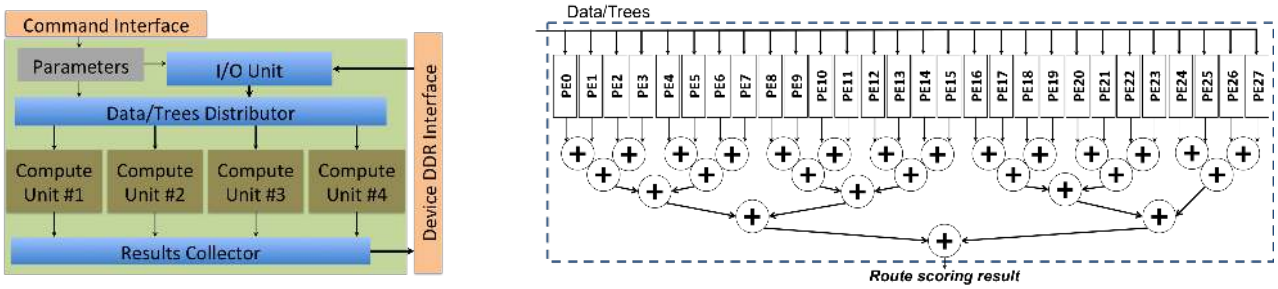


Figure 4: (a) FPGA-GBDT Inference Engine architecture. (b) Compute unit architecture.

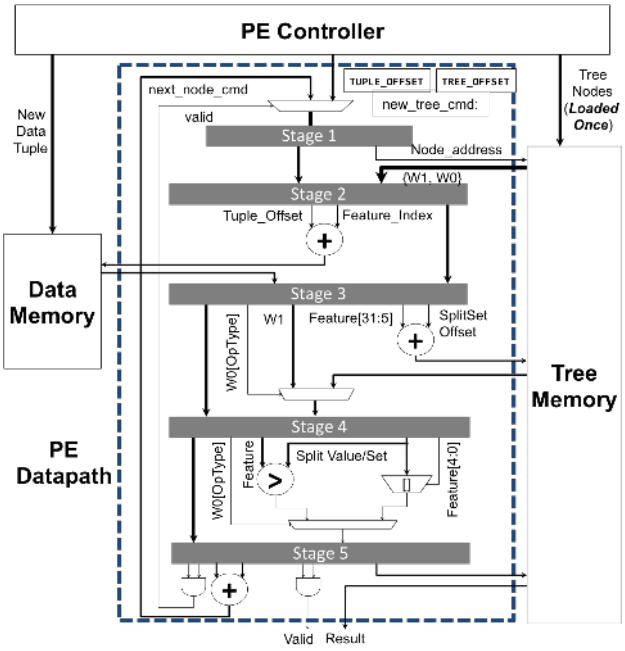


Figure 5: Processing Element (PE) architecture.

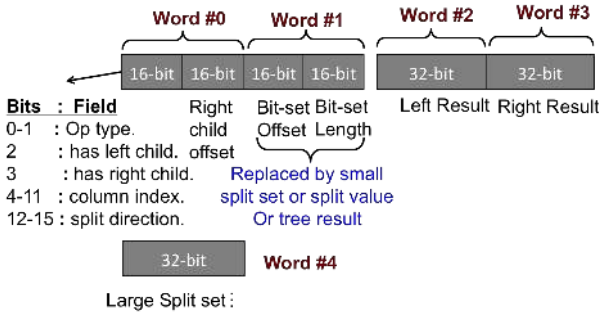


Figure 6: Decision node format and memory layout.

Unit first receives a request with a set of parameters through the command interface to perform a scoring query, then it loads the tree ensemble from the off-chip memory to the FPGA. Once all the trees are stored in the *Compute Units*, the engine becomes ready for data processing. The *I/O Unit* then starts reading the routes data from the FPGA memory and feeding it to the *Compute Units* (through the *Distributor*) for processing.

The engine can process 896 trees in parallel, either for tens of different routes (same trees processed for different routes as in small tree ensembles) or for just a few routes if the tree ensemble consists of hundreds of trees.

Both the *I/O Unit* and the *Collector* have direct access to the FPGA on-board DDR memory. The command interface is connected to the PCIe interface of the FPGA card, such that the software running on the host CPU can write a command directly to start the scoring query.

4.3 Processing Element (PE)

The processing element (Figure 5) consists of two types of components: local memories storing the tree nodes and the input routes data, and a datapath that performs the decision operation in a decision node on input routes.

Both the data and tree memories are 8 KB. Each memory structure is built out of two FPGA dual-port Block RAMs (BRAMs). The data memory BRAMs are organized to offer one 64-bit write port dedicated for storing input routes data and one 32-bit read port used for reading route features during the inference operation. The tree memory BRAMs are organized to offer four 32-bit read/write ports. During the tree ensemble loading step, two ports are used to store the trees, then during the inference step all the four ports are used for read operations. This memory organization offers enough parallel random memory accesses to sustain the processing power of the datapath.

To maximize the utilization of the ports of the tree memory and fit as many trees as possible, we customized the tree node format and memory layout offered by H2O. The original decision node format allocates more bits for the different fields. For example, the fields of Word#0 consumes 5–8 bytes in the original format. Also, the right child offset, and left and right values come after the large split set field in the original format. We mainly reduced the bit-width of the fields in Word#0 and put the large split set as the last field in the tree node to reduce the number of memory accesses to the tree memory. Figure 6 shows the format and memory layout of a tree node. The node consists of at least two or more 32-bit words. The first 32-bit word defines the decision node type including its decision operation, if it has a left or right branch and so on. The second word defines the split value or the split set depending on the decision node type. The decision node memory layout is 32-bit aligned to match the memory port sizes of the tree memory. A left child of a decision node (if it exists) is stored directly after

its parent node, while the right child is stored at the given offset in the first word. Size of split sets ranges from one to tens of 32-bit words. A decision node branches into either a decision node or a leaf node at its left and right branches. If a branch ends with a leaf node, then its value is stored in the decision node occupying the third or fourth word in the tree node as Figure 6 illustrates. If the node has a large split set, it comes after all other fields in the tree node starting from Word#4 (if the node has left and right values). The large split occupies one or more 32-bit words.

The datapath of the processing element (Figure 5) executes a loop of L iterations, where L is the tree depth. Each iteration evaluates one decision node and it takes 8 clock cycles. To benefit from the datapath resources, its logic circuit is fully pipelined, hence 8 trees can be processed in parallel. The datapath pipeline consists of five stages. Stage#1 uses 2x32-bit ports of the tree memory to read the first two words of a tree node (W0, W1). These two words contain all the necessary information to process the tree node. Stage#2 uses the column (feature) index in W0, to calculate the feature address to read from the data memory. Stage#3 reads the appropriate large split set word (if it exists) given the feature value. If the node does not have a large split set then it passes the feature value for the next stage. Stage#4 compares the feature with the split value, performs a membership test on the small split set, and chooses which operation outcome to consider based on the node type. Stage#5 either computes the next node pointer or reads the scoring result from the tree memory if the current node is a leaf node.

4.4 Design Configuration and Constraints

The engine architecture just presented is an architectural template with a set of configurable parameters adjustable at compile time (i.e., during FPGA bitstream generation) to support a certain range of tree ensembles. The template parameters include:

- Number of *Compute Units* ($\#CUs$). The number of CUs controls the degree of data parallelism exploited in the engine. More CUs can be added to the architecture depending on the available amount of FPGA resources.
- Number of *Processing Elements* ($\#PEs$). The number of PEs determines the number of trees that can be processed in parallel. But also the maximum tree ensemble size that can be stored in a CU and processed entirely on the FPGA. It is recommended to increase the number of PEs as the number of trees increases.
- Size of the *Tree Memory* (S_{MAX}). The *Tree Memory* size determines the maximum size of a single decision tree that can be stored in the tree memory and processed entirely on the FPGA.

A given configuration of the engine template supports all tree ensembles that can fit in a single CU, and not just one particular tree ensemble size. For example, the engine architecture shown in the previous Section, which consists of four CUs, 28 PEs and 8KB tree memory, can support any ensemble that satisfies the following conditions:

- The total size of the tree ensemble is less than 224 KB.
- A single tree is fully stored in a single PE tree memory.

The last constraint is important as, even if the first one is satisfied, a tree ensemble may not fit in the CU. For example, consider an ensemble of 29 trees with an average tree size equals 6 KB. The first constraint is satisfied, but after distributing 28 trees on all the PEs, the 29^{th} tree does not fit completely in any PE tree memory. In such a case, we can either add one more PE unit (hence the CU contains 29 PEs) or reduce the number of PEs (e.g., 15 PE) and increase the tree memory size to fit two trees (e.g., 12 KB). While either way allows the engine to fit the tree ensemble, the two differ in performance and resources consumption. Adding one more PE consumes more resources, but offers more performance as it increases parallelism. However, reducing the number of PEs requires less resources but cuts the engine performance by half as it reduces parallelism.

Using more CUs increases the compute performance linearly to match the memory line rate, hence turning the inference from being compute-bound to being memory-bound. Using more PEs also improves the compute performance but it does not scale as well as the CUs. The recommendation is to balance the tree memory size and the number of PEs in a compute unit such that, for the target ensemble, the highest possible utilization of the tree memory and as many PEs is reached.

5. ROUTE SCORING SYSTEM

5.1 FPGA-Software Integration

The inference engine design is platform-agnostic; it can be compiled on both Xilinx and Intel FPGAs. In the experiments we use Xilinx FPGAs to test standalone and cloud systems and Intel FPGAs to test integrated, cache-coherent co-processors.

On the Intel Xeon+FPGA platform, we used the Intel's AAL framework for the FPGA applications development. Atop of AAL, we used Centaur [41], a framework for integrating FPGA accelerators with software applications. Centaur offers concurrent access to multiple accelerators sharing the same FPGA fabric. Further, it offers a thread-like interface to applications with a functional interface for FPGA accelerators. An FPGA accelerator on Intel's platforms has direct access to the CPU memory, which means no explicit data transfers occur from CPU memory to FPGA memory. Instead, the FPGA accelerator, upon starting operation, fetches data directly from the CPU memory and writes results back to CPU memory, overlapping memory accesses with data processing.

On Xilinx platforms, we used the SDAccel application development environment. SDAccel uses the OpenCL computing model, where there is a host (CPU machine) and a compute device (FPGA). The FPGA device can be configured with one or more compute kernels. In our case a compute kernel is an inference engine instance. The data involved in processing is first transferred from the host memory to the FPGA DDR memory. Then, the compute kernels on the device are invoked to start processing. The kernel reads the data from the device memory, processes it, and writes the results back to the device memory. Then, the results are transferred from the device to the host memory back to the software application.

Figure 7 shows the SDAccel application architecture. On the software side, the SDAccel architecture includes device

drivers, runtime environment, and the OpenCL API. SDAccel allows multiple instances of the same FPGA kernel on a single FPGA device.

5.2 Integration in the Flight Search Engine Pipeline

The flight search system deploys the *Route Scoring* as part of the *Route Selection* stage and is deployed on its own servers. There are two potential architectures for integrating the FPGA *Route Scoring* module into the *Route Selection* stage: Attaching FPGA devices through PCIe to the *Route Selection* servers, or deploying the FPGA *Route Scoring* module on its own servers.

The first deployment approach is more favorable from the FPGA perspective. The software running on the CPU producing the routes can stream them at a much higher rate directly to the FPGA through PCIe compared to the streaming bandwidth over the network if the *Route Scoring* module is deployed on its own servers. This tight integration minimizes communication overhead between the *Route Selection* and *Route Scoring* stages, improving the overall response time. This might change in the future with network-attached FPGAs.

The FPGA *Route Scoring* module developed in SDAccel, exposes the scoring operation as a function call.

```
ScoreFPGA(Routes_ptr, Model_ptr, Scores_ptr)
```

The caller passes the pointers to the routes, tree model, and output scores through the *ScoreFPGA* function, which then handles all the steps to transfer data to the FPGA, triggers the computation, and later returns scoring results back to the caller application. This functional interface is convenient for existing software applications and requires minimal code modifications. It offers a drop-in replacement for existing CPU implementations regardless of the final deployment of the FPGA accelerator.

6. EXPERIMENTAL EVALUATION

6.1 Route Scoring Performance on Production Data

6.1.1 Setup

Tested GBDT Model. In this experiment we trained a GBDT model using H2O on historical flight data. A flight route record consists of 25 categorical and numerical 32-bit features. We extended the data set by creating multiple variations of the same request to influence the heuristics in order to have a bigger pool of results than a classic reply.

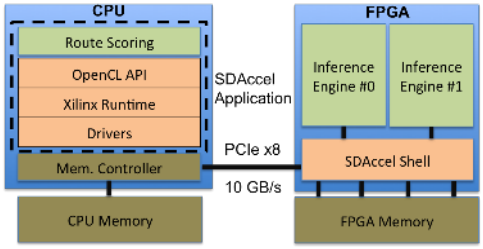


Figure 7: Route Scoring application on a CPU-FPGA server using Xilinx SDAccel environment.

The result was an ensemble of 109 trees, each tree being five levels deep. The trees differ in size, a tree size is 1–2.5 KB, and the total size of the ensemble is nearly 200 KB. The model was tested using one million routes of synthetic data.

Deployment platforms. The experiments were performed on three different machines in the AWS cloud and three stand-alone servers. Table 1 summarizes the characteristics of the different platforms used. For the CPU baseline, we reimplemented the H2O Java version in C++. Profile-based optimizations to the memory control and code streamlining were used to improve performance. The C++ implementation can score 60% more routes than the original H2O Java version. We did not optimize the data format and structures to maintain compatibility with the full specification of H2O.

The FPGA design is implemented in SystemVerilog. The FPGA is configured with two inference engines. Each inference engine has four CUs and each CU contains 28 PEs. The *Tree Memory* in the PE is 8 KB in size, which can accommodate up to four trees of the model used for testing. This is more than enough to fit the trained GBDT model in a single compute unit.

Performance metrics. We use scored routes per second as the performance metric. The throughput is calculated by dividing the number of routes processed (i.e., one million routes) over the query response time, excluding the time overhead for transferring the data and results over PCIe for FPGA-based platforms (discussed later in Section 6.1.2).

6.1.2 Results

On-premises deployment. Figure 8 plots the performance results for on-premises platforms as described in Table 1 for both compute-only and overall performance including the data transfer overhead. Both FPGAs in the Intel's HARP v2 and VCU1525 have the same inference engine configuration, but use different clock frequencies. The HARP v2 is clocked at 200 MHz and the VCU1525 is clocked at 300 MHz.

The plot shows that HARP's performance is not affected by the data movement costs since the inference engine accesses the data in the host memory directly and reading the data can be overlapped with the computation. The VCU1525 deployment with SDAccel requires an explicit step to move data from the host memory to the FPGA DDR before processing can begin, causing an important overhead. As a result, even if the VCU1525 compute-only throughput is higher than HARP due to the 30% higher clock frequency, the overall performance is slightly lower.

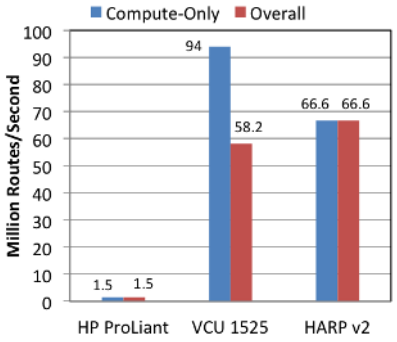


Figure 8: Route Scoring performance when deployed in on-premises platforms.

Table 1: Platforms used in the experimental evaluation.

AWS Instance	CPU	FPGA	PCIe Bandwidth	($/Hr)
CPU C5 2xlarge	8 vCPUs, 16 GiB	-	-	0.34
FPGA F1 2xlarge	8 vCPUs, 122 GiB	1 UltraScale+ VU9P, 64 GiB	10 GB/s	1.65
FPGA F1 4xlarge	16 vCPUs, 244 GiB	2 UltraScale+ VU9P, 128 GiB	10 GB/s	3.30
On Premises	**CPU**	**FPGA**	**CPU-FPGA Bandwidth**	**Cost**
HP ProLiant	56 CPU cores	-	-	11K $
Intel's HARP v2	14 cores, 64 GiB	1 Arria 10	20 GB/s	7.5K $
Xilinx VCU1525	-	1 UltraScale+ VU9P, 64 GiB	10 GB/s	7.5K $
FPGA Device	**Logic Elements**	**On Chip Memory**	**DDR Bandwidth**	
Ultrascale+	147,618 CLBs	2,160 RAMB36E (75.9 Mb)	4 DDR channels, 16 GB/s per channel	
Arria 10	427,200 ALMs	2,713 M20K (55.5 Mb)	-	

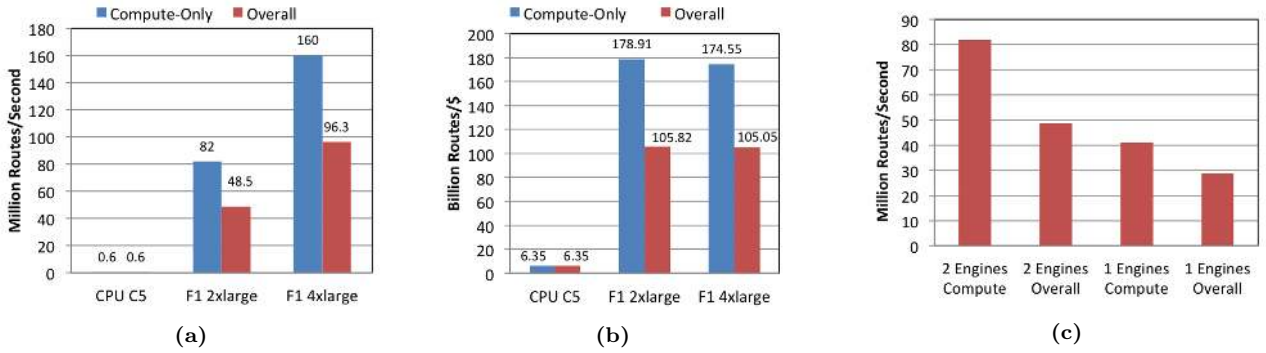


Figure 9: (a) Compute-only and Overall performance on FPGA compared to full CPU performance when deployed in AWS cloud. (b) Route Scoring throughput per Dollar running on AWS cloud. (c) Compute-only and overall performance when mapping a scoring request on one or two inference engines.

AWS cloud deployment. Figure 9 shows the performance results for the AWS instances described in Table 1. The plotted results for FPGA are the full device compute-only as well as the overall performance considering the cost of data transfers over PCIe (the FPGA is configured with two inference engines). The FPGA delivers two orders of magnitude performance improvement over the CPU solution. The huge parallelism of the FPGA (9 billion trees per second) and the tremendous throughput of random memory accesses (168 billion 64-bit read operations per second) enable the FPGA to score nearly 82 million routes per second. Although the FPGA is more expensive than the CPU instance, it is 25 times cheaper to use the FPGA to deliver the same performance as a CPU (Figure 9b).

Data transfer overhead. It is important to see how the performance will change if the data transfer from the host device to the FPGA is considered. Figure 9c compares the compute-only (just inference inside the FPGA) and the overall (including data transfers over PCIe) throughputs, for one and two inference engines instantiated on the FPGA in AWS F1 2xlarge instance. The throughput drops by 30–40% when data transfers are taken into account. Using two inference engines offers double the compute-only performance of a single engine. This is not the case when including the data transfer overhead. Both engines share the same physical PCIe link and the same memory bank on the FPGA side, limiting the benefits of the additional engine.

The limitation of the data transfer overhead we observe is a symptom of the current F1 servers' architecture and the OpenCL compute model used in Xilinx SDAccel. As a comparison, on the Intel's HARP v2 platform, the inference engine on the FPGA can access the CPU main memory directly without the need to first transfer the data to the FPGA memory. This means the HARP v2 performs better than the AWS F1 and VCU1525 FPGA card when the cost of data transfers is considered even if it is nominally a smaller and slower FPGA. Another way to speed up data transfers is on FPGA with a network connection with sufficient bandwidth. The data can then be streamed directly to the scoring engines, eliminating the data transfer overhead and utilizing the full compute throughput.

The existing solution at Amadeus implements the Route Scoring system using a total of 10 HP ProLiant servers, achieving 15 million routes per second in aggregate throughput. If deployed in the AWS Cloud, 25 CPU c5.2xlarge instances are required to maintain the same throughput. Even when considering the cost of data transfers, a single server equipped with a PCIe attached Virtex Ultrascale+ FPGA card deployed on premises, or on an AWS F1 instance, is able to replace all the servers and offer 3 times more throughput being 5 times cheaper than the 25 c5.2xlarge instances.

In terms of comparison with a GPU, such as the AWS p3.2xlarge instance (equipped with a Tesla V100 NVidia GPU) costs 3.06 $/hour, which is nearly double the cost

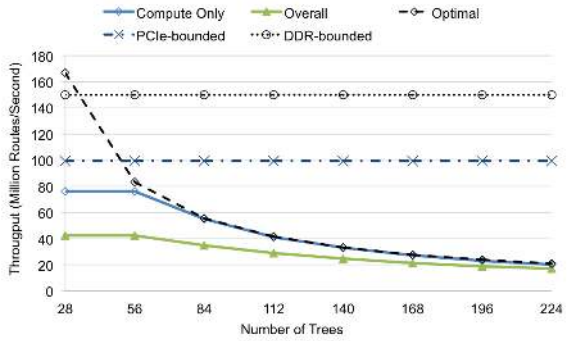


Figure 10: Inference engine performance scaling as trees ensemble size changes.

of the FPGA instance with similar characteristics. For the GPU to reach similar performance per dollar to that of the FPGA, it must double the FPGA throughput (152 Million routes/second). For on-premises deployment, the FPGA and GPU cards used in the AWS instances have a similar price (around 10K$). However, the GPU consumes 2–3x more energy than the FPGA. In a recent report from NVidia [48], a GPU implementation (on V100) of the XGBoost trees (which are much lighter than the ones we consider in this work) delivers, at best, 50 million tuples per second, which is one third of what is required to make the GPU cost competitive to the FPGA in a cloud deployment. In addition to these considerations, we are not aware of any significant results on inference over decision tree ensembles on GPUs (see related work) due to the irregular nature of the computation.

Latency analysis. The 0.6 M routes/second throughput of the AWS C5 CPU instance is the aggregate throughput of 8 cores each processing 75 K routes/second. The current CPU implementation is single threaded. A single core takes 10 ms to process a scoring query involving 750 routes. The FPGA needs 0.25 ms to score a query of 750 routes (from the data in Figure 11a). In addition, the ability to attach the FPGA through PCIe to the Route Selection servers instead of running it on a separate server eliminates the network overhead in the overall latency (see discussion Section).

6.2 Performance Scalability

In the previous Section, we showed the performance of the FPGA solution for a single point in the problem space (i.e., a specific decision tree ensemble and a single scoring request size). To assess the viability of the FPGA solution, we explore in this Section the performance scalability using a range of tree ensemble sizes and different scoring requests sizes.

6.2.1 Scaling performance with tree ensemble size

The purpose of this experiment is to see how the performance of a single configuration of the inference engine changes for different tree ensembles fitting in its allocated tree memories. As established in Section 4, the amount of time spent processing an ensemble depends on the number of trees and on the tree depth and not on the tree size or on the type of tree nodes. Hence, we randomly generated eight ensembles with a different number of trees (28–224). The trees are five levels deep, and all the trees are of the same size, 768 bytes. We run the experiment using the same inference engine configuration as in Section 6.1.

Figure 10 shows the experiment results for a scoring request of one million routes. The plot shows the compute throughput of a single inference engine without including data transfer over PCIe (compute-only), as well as the overall throughput including both data transfer and compute. As the ensemble size grows, the compute throughput drops and dominates the performance, diminishing the overhead of data transfers.

More trees consume more compute cycles of the PE datapath, hence reducing the overall throughput. In Figure 10, the "Optimal" line is computed using Equation 1, which gives the maximum compute capacity of a single engine considering all the PEs allocated in all CUs. The numerator represents the total number of compute cycles of the engine (8 is the PE datapath pipeline depth), and the denominator expresses how it is consumed by the tree ensemble. Ensembles larger than 56 trees are satisfied with the current configuration. The only limiting factor is the number of available compute cycles of the engine. However, it is important to consider other factors in the system that might impose an upper limit on performance such as PCIe bandwidth, and off-chip DDR bandwidth.

$$Optimal = \frac{\#PEs_{Total} * 8 * freq}{((8 * TreeDepth + 8) * N_{trees})} routes/s \quad (1)$$

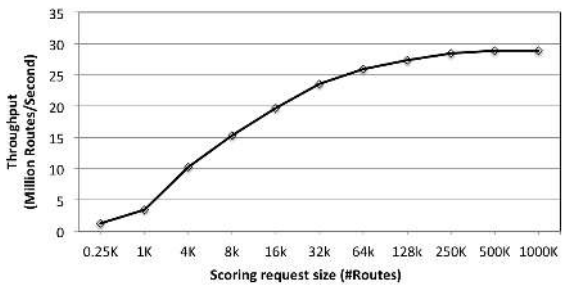
For tree ensembles with 28 and 56 trees, the throughput of the inference engine is bounded by the internal data bus width bringing data to the PEs, which operates at a maximum bandwidth of 2 GB/s (i.e. 20 million routes/second for 100-byte routes). Since each CU has its own data bus, the maximum throughput of the full engine cannot exceed 80 million routes/second (in the current configuration of four CUs). To solve this internal bandwidth limitation, the PEs-CUs distribution in the engine can be changed while maintaining the same number of PEs. Reducing the number of PEs per compute unit to 7 PEs and allocating 16 CUs offers abundant internal bandwidth to consume input data traffic, making the engine memory bound.

It is important to note that these throughput numbers are for a route size of 100 bytes. If the considered route data is much larger (e.g., 1 KB), then the PCIe and DDR upper bounds become much lower and dominate system performance. However, the compute throughput does not change as it does not depend on the data size.

The results of this experiment point to the fact that, while a single configuration of the inference engine can support a range of tree ensembles, its performance is negatively affected by the tree ensemble size. To alleviate this performance degradation, we recommend modifying the inference engine parameters discussed in Section 4.4 to produce a configuration that achieves the maximum possible throughput.

6.2.2 Scaling with request size

An important aspect of the Route Scoring system to examine is the overhead of initiating a scoring request on the overall performance. Initiating a scoring request includes a kernel invocation as well as scheduling and initiating the necessary data transfers. In the previous Sections we used a scoring request of 1 million routes, which is large enough to hide the invocation overhead. In this Section we evaluate the performance for a range of request sizes (i.e., number of



(a)

(b)

Figure 11: (a) Inference engine performance scaling as scoring request size changes. (b) Scoring request response time distribution for different requests sizes.

routes to score). We used the same inference engine configuration and tree ensemble as in Section 6.1, and considered a wide range of request sizes from a couple of hundreds of routes to hundreds of thousands. In these experiments, the FPGA device on the AWS F1 is programmed with one inference engine.

Figure 11a plots the results of the experiment for different request sizes. The throughput of a single inference engine is very low for small requests barely improving over a single CPU server. The overhead of initiating a scoring operation is on the order of hundreds of microseconds, which is much more than the latency of processing a couple hundred routes (approximately $6\mu s$). This observation implies that although the FPGA achieves an order of magnitude higher performance over CPU servers, it will fail to deliver this performance with the current setup where all scoring requests are few hundred routes.

Figure 11b illustrates the response time distribution across different request sizes. The response time components are the computation time, data transfer time (host to FPGA), results transfer time (FPGA to host), loading trees to on-chip memories, and kernel invocation latency. For small request sizes, the kernel invocation overhead dominates the overall request response time. As requests grow in size, the compute part dominates the request response time, followed by the time to transfer data from host memory to FPGA DDR.

A solution to alleviate the kernel invocation overhead is to batch thousands of these small requests to minimize the initiation overhead. It also indicates that the route scoring is suitable to be used inside the *Domain Explorer*, which can generate hundreds of thousands to millions of routes before early stage filtering.

6.3 Resource Usage

Table 2 details the amount of consumed resources by the different components of the FPGA. The critical FPGA resource for the inference engine is the FPGA BRAMs used to store the trees as well as the routes while being processed. A single inference engine consumes 24% of the FPGA memory resources. When allocating two inference engines, 60% of the FPGA BRAMs are used. While there is still a sufficient amount of resources to allocate a third engine, the routing complexity inside the engine architecture and the layout of the Xilinx Ultrascale+ FPGAs make it very hard to allocate three engines in the FPGA. However, it is easier to use the available BRAMs to increase the size of tree memories to fit larger or more trees. The floating-point adders tree consumes nearly 32% of the CLBs occupied by a CU.

The rest of the engine components (*I/O Unit, Collector, State-Machine*, etc.) consume nearly 9% of the overall engine resources (both logic and memory). However, the percentage would be higher if the engine only contained one CU. Then the *I/O Unit* and *State-Machine* would consume nearly 29% of the engine resources. The takeaway point here is that it is preferable to parallelize data/trees processing by allocating as many CUs as possible in the inference engine instead of allocating many small inference engines. The final decision behind the allocation of resources depends on the application characteristics (e.g., many scoring requests using different GBDT models), and the success of generating an FPGA bitstream, which is a daunting process includes synthesis, placement and routing steps of many inference engines on the FPGA.

7. DISCUSSION

The performance improvements achieved on the FPGA opens up several options for integrating the Route Scoring system in the search pipeline even if we are bound by existing architectural configurations. If the Route Scoring is embedded inside the Route Selection stage, the ten dedicated servers currently used for the Route Scoring system in a data center can be replaced by a single FPGA-based server. If the Route Scoring is deployed on a single FPGA server, the server must have sufficient network bandwidth to feed enough data to the FPGA to utilize its full compute capacity. To maintain the current throughput of all ten servers (15 million route per second), a minimum aggregate network bandwidth of 12 Gbps is required, within easy reach of modern data center networks.

An alternative design that removes the need for an additional server for Route Scoring is to equip every Route Selec-

Table 2: Consumed FPGA resources by different modules.

Module	CLBs		BRAMs	
Processing Element	112	0.08%	4	0.19%
Compute Unit	4557	3.1%	121	5.6%
Inference Engine	20076	13.6%	520	24.1%
SDAccel Shell	31936	21.6%	215	10%

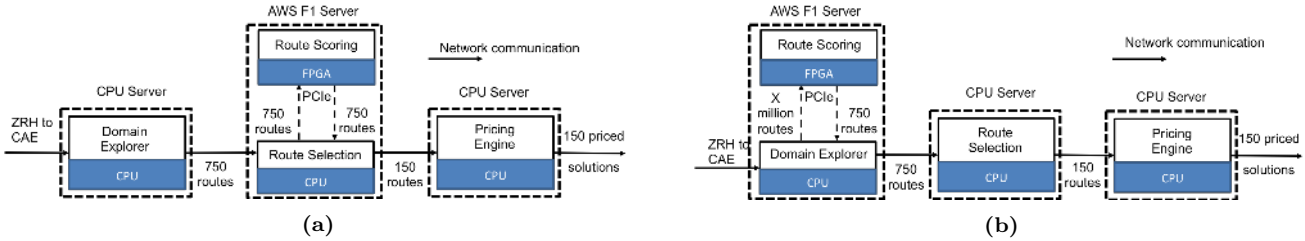


Figure 12: (a)Inserting a small FPGA card in each Route Selection server attached through PCIe.(b) Deploying the Route Scoring as part of the Domain Explorer by attaching an FPGA card to each Domain Explorer server.

tion server with an FPGA implementing the route scoring engines, removing the network overhead (Figure 12). The 48 millions of routes evaluated per second on the FPGA also enable a deployment of the Route Scoring inside the Domain Explorer as it can deal with the much higher number of routes generated there. An FPGA card in every Domain Explorer node would allow the scoring of hundreds of thousands of routes for each flight availability request and 10 ms of latency requirement for route scoring (Figure 12b). The same type of deployment could be applied in the cloud. The Domain Explorer can be deployed on an AWS F1 instance with the Route Scoring inside the FPGA.

The increasing availability of FPGAs and the way they are evolving is in favor of our design. For instance, designs such as those on Microsoft Azure where the FPGA has direct network access [11] or the new Xilinx Alveo cards with network ports on the PCIe attached FPGA would enable a configuration in which the FPGA can directly communicate with other servers, thereby significantly reducing latency in the data movement and increasing the available bandwidth.

The current FPGA implementation assumes that the decision tree ensemble fits in the FPGA memory. However, this might not be the case for very large ensembles. Storing the ensemble in off-chip memory will significantly diminish the FPGA performance as it does not offer the huge random memory bandwidth and low latency of on-chip memory. In prior work, we have developed two solutions to the problem: partitioning the ensemble into smaller partitions that fit in the FPGA and processing them sequentially on the same FPGA [42], or using a cluster of FPGAs to process very large ensembles [40].

8. RELATED WORK

8.1 Low-Latency Data Processing Pipelines

There have been many efforts both in academia and industry to use hardware accelerators such as FPGAs to improve the response time of latency-sensitive data processing pipelines. Microsoft initially developed the Catapult platform [46, 11] to accelerate the search queries of the Bing search engine, in particular the document ranking stage of the Bing data processing pipeline. The document ranking stage employs a series of feature computation, features synthesis, and machine-learning based document scoring. The FPGA acceleration of the ranking service reduced tail latency by 29% while maintaining equivalent throughput.

Alibaba Cloud recently deployed an accelerated database system called X-DB to boost the performance of their services [2]. In addition to accelerating SQL search queries, the FPGA performs the compaction operation, which merges multiple versions of the SSTables in the persistent storage and keeps the latest version. Alibaba was able to achieve up to 60% improvement on throughput and reducing significantly the performance jitters caused by the compaction operation competing with normal transaction processing on the CPU.

Similarly, Baidu has deployed Software Defined Accelerators (SDA) based on Xilinx FPGAs in their datacenters to accelerate data analysis systems (e.g., Spark) [39]. The SDA offers a flexible hardware architecture implementing core SQL operators (join, select, filter, sort, group by, aggregate). A SQL plan is translated into a program to configure the SDA to perform a specific query. For certain benchmarks, the SDA achieves an improvement of one order of magnitude over CPU in terms of query response times.

Sidler et al. [53] have proposed an FPGA solution for accelerating database pattern matching queries, the proposed solution reduces query response time by 70%. Similarly, Kara et al. [27], demonstrated how offloading the partitioning operation of the SQL join operator to the FPGA can significantly improve performance and offer a robust solution.

High-speed trading is another field where FPGAs have been used to significantly improve response time. Tang et al. [57] developed a market-data processing library for FPGAs that achieves an improvement of two orders of magnitude over CPU latency.

All the prior efforts draw parallels to our effort in this work to improve the overall response of a given data processing pipeline. In all the mentioned accelerator solutions, as in ours, the developers strove to meet two conditions: lower latency and accelerator flexibility to adapt to changing workload characteristics and user requirements. The flexibility is achieved through highly parameterized FPGA architectures (Bing search, X-DB) or a custom, software programmable Instruction Set Architecture (Baidu SDA).

8.2 Machine Learning on FPGAs

The stagnating performance of CPUs running machine-learning workloads opened the door for hardware accelerators such as FPGAs and GPUs which are more suited to these type of workloads [15, 7, 60].

There is already quite a lot of work on decision trees for FPGAs [17, 5, 20, 20, 47, 36]. In most cases, it is highly customized to particular applications (e.g., [36]), supports only small decision trees [50] or has been designed for different architectures [42]. Oberg et al. [36] presented an FPGA classification pipeline for the Microsoft Kinect depth image pixels. The problem they addressed was how to accelerate very

deep random forest trees that do not fit inside the FPGA memories. Their ensembles consisted of just a few trees but each tree contains hundreds of thousands of nodes. Their solution used the FPGA off-chip DDR memories to store the trees and was based on reordering of the computation to minimize off-chip DDR memory accesses.

Essen et al. [20] approached the acceleration of decision trees inference by unfolding the tree levels and allocating separate compute and memory resources for each tree level. While this works for ensembles with a small number of decision trees, it does not scale well for large tree ensembles.

The authors report a throughput of 31.25 M tuples/second using four Virtex-6 FPGAs (manufactured with 40 nm technology) for a random forest of 32 trees, each six levels deep. This throughput is compute-only and obtained from simulation. Their design mainly uses LUTs, Flip Flops, and DSPs and consumes half of the FPGA resources and runs on 70 MHz. The Virtex Ultrascale+ FPGA we use offers nearly eight times the amount of resources of their FPGA, and is manufactured with 16 nm process-node. If we reimplement their design on our FPGA (we do not have access to their source code), adapt the design to our GBM trees and ensemble size which is nearly 16 times more complex in operations and memory requirements, we estimate that we will need two Ultrascale+ FPGAs to implement the trees. Running at 200 MHz their solution would deliver about 80 M routes/second. Compared to our solution, using one VCU1525 we achieve 94 M routes/second (compute-only).

Integrating machine-learning in relational databases has been explored to benefit from the optimized data structures and operators inside the RDBMS. Sattler et al. [51] proposed a set of SQL primitives as building blocks for implementing a decision tree classifier in relational databases. To alleviate the overhead of the slow decision tree primitives in a database, Bentayeb et al.[9] proposed an approach to represent an ID3 decision tree using SQL views.

Recently, a large body of research work has explored a range of hardware accelerator solutions for Deep Neural Networks (DNNs), another machine-learning method. Microsoft Brainwave project [15] deploys an FPGA-based DNN processor in the Bing search engine. The FPGA solution meets the strict latency requirements of the Bing data processing pipeline offering an order of magnitude lower response time compared to a CPU-only solution. The Brainwave DNN processor harnesses the vast amount of FPGA resources to create wide parallel datapaths with a custom instruction set. As in our case, the approach must support the frequent updates to the DNN models. Other work has explored the acceleration of machine-learning methods such as KMeans [23], Association Rules [45], Generalized Linear Models [25], etc.

All these systems are based on the same architectural premise we follow in this work: latency-sensitive systems employing machine-learning in their data processing pipelines need to use hardware accelerators to meet their stringent timing constraints while still benefiting from the better results provided by machine-learning methods.

8.3 Machine Learning on GPUs

GPUs offer tremendous parallel processing capacity by allocating thousands of scalar processing elements equipped with double-precision arithmetic units. GPUs use a Single Instruction Multiple Data (SIMD) execution model, where parallel processing units concurrently crunch large amounts of sequential data. The GPU execution model and architecture is suitable for methods such as neural networks but it works less well for non-sequential control flow, low-density computations, and if a significant amount of random memory accesses is involved. These are, unfortunately, precisely the characteristics of decision tree inference [48].

Existing work on using GPUs for decision trees focuses almost exclusively on training and learning of the trees, a task where GPUs excel. Only recently, NVidia described a library for inference over gradient-boosted trees (XGBoost models) that requires users to make the input dataset and the tree ensemble dense. Also, the library allows for only fully grown trees with leaves only at the deepest layer [48]. The input is assumed to be already in the memory of the GPU, in batches of up to a million queries, and stored so as to minimize random accesses. The trees need to be stored in a weaved manner, with processing occurring first for node 0 of all trees, then for node 1, for node 2, etc. which limits the ability to deal with sparse inputs and sparse trees. The reported performance results indicate that our solution is 2x to 4x more cost-effective when considering deployments in AWS cloud as we discussed in Section 6.1.2. Even for very high-end GPUs (NVidia P100 and V100, typically used in supercomputing) and without having to impose any limitations on data allocation, batching, format, or tree representation, our solution achieves twice their performance when comparing their published numbers and ours (the library was not publicly available at the time of writing).

Earlier work [52] has shown that a GPU can deliver speedup of an order of magnitude over a CPU for relatively small ensembles (just a few trees). Liao et al. [31] have proposed a GPU-based solution for both training trees and inference, offering an improvement of an order of magnitude over a CPU baseline implementation. Recent work from NVidia [19] demonstrated the ability of GPUs to deliver an improvement of two orders of magnitude on CPU performance when training GBDTs but there is no mentioning of using the system for inference. The algorithm is now available as part of H2O GPU Edition.

9. CONCLUSIONS

In this paper we have addressed the problem of combining machine learning methods with latency-sensitive search pipelines. Using a real use case, we have shown how the use of specialized hardware (an FPGA-based solution) and a novel design of inference over tree ensembles can lead to a significant performance improvement as well as a potential boost to the quality of the results obtained by enabling the scoring of many more potential answers. Future work will involve the automated generation of optimal designs using a parameterized template.

10. ACKNOWLEDGMENTS

We would like to thank Intel for the generous donation of the HARP platform and Xilinx for the generous donation of the Ultrascale+ board. Part of the work of Muhsen Owaida was funded by a grant from Amadeus.

11. REFERENCES

[1] J. A. Konstan and J. Riedl. Recommender Systems: From Algorithms to User Experience. *User Modeling and User-Adapted Interaction*, 22(1):101–123, 2012.

[2] Alibaba Cloud. When Databases Meet FPGA Achieving 1 Million TPS with X-DB Heterogeneous Computing. `https://www.alibabacloud.com/blog/`.

[3] G. Alonso, Z. Istvan, K. Kara, M. Owaida, and D. Sidler. DoppioDB 1.0: Machine Learning inside a Relational Engine. *IEEE Data Engineering Bulletin*, 42(2):19–31, 2019.

[4] Altexsoft. Fraud Detection: How Machine Learning Systems Help Reveal Scams in Fintech, Healthcare, and eCommerce. Technical report, 2017. `https://www.altexsoft.com/whitepapers/`.

[5] F. Amato, M. Barbareschi, V. Casola, and A. Mazzeo. An FPGA-Based Smart Classifier for Decision Support Systems. In *Proceedings of the International Symposium on Intelligent Distributed Computing (IDC)*, pages 289–299, 2014.

[6] X. Amatriain and J. Basilico. *Recommender Systems in Industry: A Netflix Case Study; Recommender Systems Handbook*. Springer, 2015.

[7] Amazon AWS. Amazon EC2 F1 Instances. `https://aws.amazon.com/ec2/instance-types/f1/`.

[8] I. Arapakis, X. Bai, and B. B. Cambazoglu. Impact of Response Latency on User Behavior in Web Search. In *Proceedings of the International ACM SIGIR Conference on Research & Development in Information Retrieval*, pages 103–112, 2014.

[9] F. Bentayeb and J. Darmont. Decision Tree Modeling with Relational Views. In *Proceedings of the International Symposium on Methodologies for Intelligent Systems (ISMIS)*, pages 423–431, 2007.

[10] J. D. Brutlag, H. Hutchinson, and M. Stone. User Preference and Search Engine Latency. In *JSM Proceedings, Qualify and Productivity Research Section.*, pages 1–13, 2008.

[11] A. Caulfield, E. S. Chung, A. Putnam, H. Angepat, J. Fowers, and et al. A Cloud-Scale Acceleration Architecture. In *Proceedings of the IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 1–13, 2016.

[12] M. Chau and H. Chen. A Machine Learning Approach to Web Page Filtering Using Content and Structure Analysis. *Decision Support Systems*, 44(2):482–494, 2008.

[13] T. Chen and C. Guestrin. XGBoost: A Scalable Tree Boosting System. In *Proceedings of the International Conference on Knowledge Discovery and Data Mining (KDD)*, pages 785–794, 2016.

[14] Y.-T. Chen, J. Cong, Z. Fang, J. Lei, and P. Wei. When Spark Meets FPGAs: A Case Study for Next-Generation DNA Sequencing Acceleration. In *Proceedings of the USENIX Conference on Hot Topics in Cloud Computing (HotCloud)*, pages 64–70, 2016.

[15] E. Chung, J. Fowers, K. Ovtcharov, M. Papamichael, A. Caulfield, and et al. Serving DNNs in Real Time at Datacenter Scale with Project Brainwave. In *Proceedings of the IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 8–20, 2018.

[16] D. Cook. *Practical Machine Learning with H2O: Powerful, Scalable Techniques for Deep Learning and AI*. O'Reilly Media, 2016.

[17] T. G. Dietterich. Ensemble Methods in Machine Learning. In *Proceedings International Workshop on Multiple Classifier Systems (MSC)*, pages 1–15, 2000.

[18] M. Endrei, J. Ang, A. Arsanjani, S. Chua, P. Comte, P. Krogdahl, M. Luo, and T. Newling. *Patterns: Service-Oriented Architecture and Web Services*, volume 1. IBM Redbooks, 2004.

[19] V. Ershov. CatBoost Enables Fast Gradient Boosting on Decision Trees Using GPUs, 2018. `https://devblogs.nvidia.com/category/artificial-intelligence/`.

[20] B. V. Essen, C. Macaraeg, M. Gokhale, and R. Prenger. Accelerating a Random Forest Classifier: Multi-Core, GP-GPU, or FPGA? In *Proceedings of the IEEE International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 232–239, 2012.

[21] J. Gehrke, V. Ganti, R. Ramakrishnan, and W.-Y. Loh. BOAT—Optimistic Decision Tree Construction. In *Proceedings of the International Conference on Management of Data (SIGMOD)*, pages 169–180, 1999.

[22] J. Gehrke, R. Ramakrishnan, and V. Ganti. RainForest - A Framework for Fast Decision Tree Construction of Large Datasets. *Data Mining and Knowledge Discovery*, 4(2):127–162, 2000.

[23] Z. He, D. Sidler, Z. István, and G. Alonso. A Flexible K-Means Operator for Hybrid Databases. In *Proceedings of the International Conference on Field Programmable Logic and Applications (FPL)*, pages 368–3683, 2018.

[24] M. Kainth, D. Pritsker, and H. S. Neoh. FPGA Inline Acceleration for Streaming Analytics. Technical report, 2018.

[25] K. Kara, D. Alistarh, G. Alonso, O. Mutlu, and C. Zhang. FPGA-Accelerated Dense Linear Machine Learning: A Precision-Convergence Trade-Off. In *Proceedings of the IEEE International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 160–167, 2017.

[26] K. Kara, K. Eguro, C. Zhang, and G. Alonso. ColumnML: Column-store Machine Learning with On-the-fly Data Transformation. *PVLDB*, 12(4):348–361, 2018.

[27] K. Kara, J. Giceva, and G. Alonso. FPGA-Based Data Partitioning. In *Proceedings of the International Conference on Management of Data (SIGMOD)*, pages 433–445, 2017.

[28] B. P. Knijnenburg, M. C. Willemsen, Z. Gantner, H. Soncu, and C. Newell. Explaining the User Experience of Recommender Systems. *User Modeling and User-Adapted Interaction*, 22(4):441–504, 2012.

[29] D. Koeplinger, M. Feldman, R. Prabhakar, Y. Zhang, S. Hadjis, R. Fiszel, T. Zhao, L. Nardi, A. Pedram, C. Kozyrakis, and K. Olukotun. Spatial: A Language and Compiler for Application Accelerators. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 296–311, 2018.

[30] B. Li, Z. Ruan, W. Xiao, Y. Lu, Y. Xiong, A. Putnam, E. Chen, and L. Zhang. KV-Direct: High-Performance In-Memory Key-Value Store with Programmable NIC. In *Proceedings of the Symposium on Operating Systems Principles (SOSP)*, pages 137–152, 2017.

[31] Y. Liao, A. Rubinsteyn, R. Power, and J. Li. Learning Random Forests on the GPU. In *Proceedings of Big learning: Advances in Algorithms and Data Management*, pages 1–6, 2013.

[32] D. Mahajan, J. K. Kim, J. Sacks, A. Ardalan, A. Kumar, and H. Esmaeilzadeh. In-RDBMS Hardware Acceleration of Advanced Analytics. *PVLDB*, 11(11):1317–1331, 2018.

[33] N. Mehta. UltraScale Architecture: Highest Device Utilization, Performance and Scalability. Technical report, 2015. `https://www.xilinx.com/support/documentation/white_papers/wp455-utilization.pdf`.

[34] A. Mitra, W. Najjar, and L. Bhuyan. Compiling PCRE to FPGA for Accelerating SNORT IDS. In *Proceedings of the ACM/IEEE Symposium on Architecture for networking and communications systems (ANCS)*, pages 127–136, 2007.

[35] A. Natekin and A. Knoll. Gradient Boosting Machines, a Tutorial. *Frontiers in Neurorobotics*, 7(Dec), 2013.

[36] J. Oberg, K. Eguro, and R. Bittner. Random Decision Tree Body Part Recognition Using FPGAs. In *Proceedings of the International Conference on Field Programmable Logic and Applications (FPL)*, pages 330–337, 2012.

[37] N. Oliver, R. Sharma, S. Chang, et al. A Reconfigurable Computing System Based on a Cache-Coherent Fabric. In *Proceedings of the International Conference on Reconfigurable Computing and FPGAs (ReConFig)*, pages 80–85, 2011.

[38] Oracle Inc. Oracle Data Mining Concepts. `https://docs.oracle.com/database/121/DMCON/toc.htm`.

[39] J. Ouyang, W. Qi, Y. Wang, Y. Tu, J. Wang, and B. Jia. SDA: Software-Defined Accelerator For General-Purpose Big Data Analysis System. In *Proceedings of the IEEE Hot Chips Symposium*, pages 1–23, 2016.

[40] M. Owaida and G. Alonso. Application partitioning on fpga clusters: Inference over decision tree ensembles. In *Proceedings of the International Conference on Field Programmable Logic and Applications (FPL)*, pages 295–300, 2018.

[41] M. Owaida, D. Sidler, K. Kara, and G. Alonso. Centaur: A framework for hybrid cpu-fpga databases. In *Proceedings of the IEEE International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 211–218, 2017.

[42] M. Owaida, H. Zhang, C. Zhang, and G. Alonso. Scalable inference of decision tree ensembles: Flexible design for CPU-FPGA platforms. In *Proceedings of the International Conference on Field Programmable Logic and Applications (FPL)*, pages 1–8, 2017.

[43] B. Panda, J. S. Herbach, S. Basu, and R. J. Bayardo. Planet: Massively parallel learning of tree ensembles with mapreduce. *PVLDB*, 2(2):1426–1437, 2009.

[44] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, et al. Scikit-learn: Machine Learning in Python. *Journal of Machine Learning Research*, 12(nov):2825–2830, 2011.

[45] A. Prost-Boucle, F. Pétrot, V. Leroy, and H. Alemdar. Efficient and Versatile FPGA Acceleration of Support Counting for Stream Mining of Sequences and Frequent Itemsets. *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, 10(3):1–21, 2017.

[46] A. Putnam, A. M. Caulfield, E. S. Chung, D. Chiou, and et. al. A Reconfigurable Fabric for Accelerating Large-Scale Datacenter Services. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, pages 13–24, 2014.

[47] Y. R. Qu and V. K. Prasanna. Scalable and Dynamically Updatable Lookup Engine for Decision-Trees on FPGA. In *Proceedings of the IEEE High Performance Extreme Computing Conference (HPEC)*, pages 1–6, 2014.

[48] S. Rao, T. Nanditale, and V. Deshpande. GBM Inferencing on GPU. *NVIDIA GPU Technology Conference*, 2018. http://on-demand-gtc.gputechconf.com/gtc-quicklink/ghywWyq.

[49] B. Ronak and S. A. Fahmy. Mapping for Maximum Performance on FPGA DSP Blocks. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 35(4):573–585, 2016.

[50] F. Saqib, A. Dutta, and J. Plusquellic. Pipelined Decision Tree Classification Accelerator Implementation in FPGA (DT-CAIF). *IEEE Transactions on Computers*, 64(1):280–285, 2015.

[51] K.-U. Sattler and O. Dunemann. SQL Database Primitives for Decision Tree Classifiers. In *Proceedings of the International Conference on Information and Knowledge Management (CIKM)*, pages 379–386, 2001.

[52] T. Sharp. Implementing Decision Trees and Forests on a GPU. In *Proceedings of the European Conference on Computer Vision (ECCV)*, pages 595–608, 2008.

[53] D. Sidler, Z. István, M. Owaida, and G. Alonso. Accelerating Pattern Matching Queries in Hybrid CPU-FPGA Architectures. In *Proceedings of the International Conference on Management of Data (SIGMOD)*, pages 403–415, 2017.

[54] D. Sidler, M. Owaida, Z. István, K. Kara, and G. Alonso. doppioDB: A Hardware Accelerated Database. In *Proceedings of the International Conference on Management of Data (SIGMOD)*, pages 1659–1662, 2017.

[55] A. Singhal, P. Sinha, and R. Pant. Use of Deep Learning in Modern Recommendation System: A Summary of Recent Works. *International Journal of Computer Applications*, 180(7):17–22, 2017.

[56] B. Sukhwani, H. Min, M. Thoennes, P. Dube, B. Brezzo, S. Asaad, and D. E. Dillenberger. Database Analytics: A Reconfigurable-Computing Approach. In *Proceedings of the IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 19–29, 2014.

[57] Q. Tang, M. Su, L. Jiang, J. Yang, and X. Bai. A Scalable Architecture for Low-Latency Market-Data Processing on FPGA. In *Proceedings of the IEEE Symposium on Computers and Communication*

(ISCC), pages 597–603, 2016.

[58] P. Unterbrunner, G. Giannikis, G. Alonso, D. Fauser, and D. Kossmann. Predictable performance for unpredictable workloads. *PVLDB*, 2(1):706–717, 2009.

[59] Z. Wang, K. Kara, H. Zhang, G. Alonso, O. Mutlu, and C. Zhang. Accelerating Generalized Linear Models with MLWeaving: A One-Size-Fits-All System for Any-Precision Learning. *PVLDB*, 12(7):807–821, 2019.

[60] Xilinx. Accelerating DNNs with Xilinx Alveo Accelerator Cards . Technical report, 2018. `https://www.xilinx.com/support/documentation/white_papers/wp504-accel-dnns.pdf`.

[61] Xilinx. Introduction to FPGA Design with Vivado High-Level Synthesis. Technical report, 2019. `https://www.xilinx.com/support/documentation/sw_manuals/ug998-vivado-intro-fpga-design-hls.pdf`.

[62] M. Zareapoor and P. Shamsolmoali. Application of Credit Card Fraud Detection: Based on Bagging Ensemble Classifier. *Procedia Computer Science*, 48:679–685, 2015.