Decision Tree and Random Forest Implementations for Fast Filtering of Sensor Data

Sebastian Buschjäger and Katharina Morik

***Abstract*— With increasing capabilities of energy efficient systems, computational technology can be deployed, virtually everywhere. Machine learning has proven a valuable tool for extracting meaningful information from measured data and forms one of the basic building blocks of ubiquitous computing. In high-throughput applications, measurements are rapidly taken to monitor physical processes. This brings modern communication technologies to its limits. Therefore, only a subset of measurements, the interesting ones, should be further processed and possibly communicated to other devices. In this paper, we investigate architectural characteristics of embedded systems for filtering high-volume sensor data before further processing. In particular, we investigate implementations of decision trees and random forests for the classical von-Neumann computing architecture and custom circuits by the means of field programmable gate arrays.**

***Index Terms*— Field programmable gate arrays (FPGA), Internet of Things (IoT), machine learning (ML), decision trees, random forest.**

I. Introduction

INFORMATION technology is more and more integrated into every part of life, with applications ranging from factory monitoring, scientific experiments to serving consumer needs. Based on networking protocols and small, energy efficient, embedded systems it is now possible to measure and process data at virtually every place, at any time. In addition, combining multiple embedded systems creates one large ubiquitous computing system [1].

Once measurements are taken, one can either process information locally at the sensing sensor or transmit measurements to a central server. The first approach requires some processing power on the local sensor nodes, whereas the second approach requires a large network bandwidth.

In many applications, the interesting events are rare compared to the volume of sensor measurements. One example of such a high-throughput application arises in the context of smart factories, where the current state of machinery is concurrently monitored: Usually, a machine operates within its operating characteristics and produces measurements according to this normal state. These measurements only inform about the machine working as expected. Once the machine reaches a state outside its operating characteristics, measurements suddenly become very valuable to detect what happened to the machinery.

Another example for such applications can be found in modern astro-physics experiments, e.g. the First G-APD Cherenkov Telescope (FACT) for which we develop methods.[1] By analysing the gamma beams emitted by celestial objects, physicists can derive further insight about the characteristics of these objects. One challenge is the gamma-hadron separation, where only 1 in 1000 or even 10000 measurements account for an interesting gamma event, whereas the rest is produced by background noise [2].

Both applications illustrate a skewed distribution. In order to detect interesting events, a high sampling rate is required. However, communication throughput is limited at the sensing node and thus measurements need to be pre-filtered before transmission to a central server, where data can be examined more thoroughly.

The filtering of interesting vs. uninteresting events can be viewed as a binary classification problem and therefore Machine Learning seems to be a good fit for this problem [2]. In the presented context, we are interested in model application and not model learning. Pre-filtering is a crucial step in every application and thus needs to be trusted. Prediction models produced by Machine Learning such as Neural Networks or the SVM can be difficult to interpret and thus cannot be validated by a domain expert such as the machine operator or the physicist in the above examples [3].

Decisions trees, on the other hand, are easy to interpret and form a simple model, which can be reviewed by domain experts [4]. Combining multiple decision trees in a random forest [5] maintains this interpretability, but offers state-of-the-art prediction accuracies. Therefore, we focus on random forests for pre-filtering.

Sensor data are usually normalized as they can suffer from offset errors as well as scaling errors (see e.g. [6]). In contrast, decision trees can be trained on unnormalized data. This allows us to filter out unwanted events based on raw data, without prior feature processing. Only the reduced volume of the filtered data is then normalized and used for further feature processing on the central computation node.

Computation resources including energy can always be viewed as limited resource. The more energy a computational system needs, the more costly is its operation. The more computational resources a given computation needs, the longer

Manuscript received December 13, 2016; revised May 12, 2017; accepted May 29, 2017. This paper was recommended by Associate Editor A. Sangiovanni Vincentelli. *(Corresponding author: Sebastian Buschjäger.)*

The authors are with the Computer Science 8–Artificial Intelligence Unit, TU Dortmund University, 44227 Dortmund, Germany (e-mail: sebastian.buschjaeger@tu-dortmund.de; katharina.morik@tu-dortmund.de).

Color versions of one or more of the figures in this paper are available online at http://ieeexplore.ieee.org.

Digital Object Identifier 10.1109/TCSI.2017.2710627

[1] https://sfb876.de/fact-tools/

the system needs to run and thus requires more energy leading to increased costs.

Today's hardware landscape offers a variety of small, energy efficient devices with different computational architectures. Application specific integrated circuits (ASIC) offer high throughput with small energy consumption, but suffer from static behaviour. In contrast, normalization constants depend on the environment, e.g. the temperature. Consequently, decision trees as well as these constants may change due to environmental changes. We identify ASICs as a non viable option for massive sensor streams, because their behaviour cannot be adapted to new situations.

Programmable hardware components, such as field programmable gate arrays (FGPAs), can be re-programmed during runtime and thus can alter their functionality. Alternatively, central processing units (CPU) in a von-Neumann computation model interpret instructions, which can also be changed during runtime. Therefore, we identify classical CPUs as well as FPGAs as viable options for pre-filtering sensor data in small, embedded systems [7].

Now, how to organize the analysis of massive sensor streams, so that pre-filtering can be executed already on the sensing devices? More specifically, which computing architecture enables a high throughput for filtering of raw sensor data in resource-restricted computation nodes? In this paper we investigate the relationship between Machine Learning and hardware architecture for the analysis of sensor data. Our contributions are the following.

- **Comparison of FPGAs and CPUs:** We compare FPGAs and von-Neumann CPUs including those which offer single instruction multiple data operations. We consider a simple computational model for both architectures that covers the specific characteristics of each architecture.
- **Random Forests on FPGAs and CPUs:** Based on these computational models, we compare different implementations of random forests on both architectures. The runtime of each implementation is derived on a theoretical basis taking the skewed data distribution of wanted vs. unwanted events into account.
- **Architectural dependent code generation:** We present an architectural dependent code generation tool and use this tool to validate the presented theoretical model. We make our implementation available at https://bitbucket.org/sbuschjaeger/arch-forest.
- **Filtering in astro-particle physics:** We recommend a combination of hardware architecture and implementation that is well suited for filtering sensor data directly where they are measured. We illustrate our approach by a real-world application, namely the FACT telescope that recognizes gamma rays.

The paper is organized as follows. In section II we will present related work. Then, we will introduce our notation for skewed distributions in the context of decision trees, as well as random forests.

In section IV, we will explain the von-Neumann architecture, as well as FPGAs. Additionally, we present a simple theoretical model to measure clock cycles and resources used by different implementations.

After that, we will show different implementations of decision trees and random forests for CPUs and FPGAs in sections V and VI. Last, we will compare our theoretical model to real world hardware and give a recommendation for a combination of hardware and implementation in the context of FACT in section VII. The paper concludes with a discussion in section IX.

II. Related Work

Random forests and decision trees have been studied in the context of CPUs and FPGAs, already. Narayanan *et al.* present in [8] a method to compute the Gini score for decision tree induction on FPGAs with a speed-up factor up to 5.5 compared to a software implementation for CPUs. Unfortunately, Narayanan *et al.* do not consider the application of learnt models, but focus on model learning.

Van Essen *et al.* present in [9] a comprehensive study of different architectures for implementing random forests on CPUs, FPGAs and GPUs. Based on the CATE algorithm for forest induction presented in [10], the authors train a random forest with decision trees of fixed height. By utilizing fixed-size trees, the authors show an effective pipelining approach for tree application on CPUs, FPGAs and GPUs. Unfortunately, the authors only consider energy-hungry hardware, which can be found in desktop and server systems, but not in embedded systems. Additionally, the authors do not take modern CPU vectorization units into account.

In [11], Saqib *et al.* present a hardware-software co-design approach in which a software implementation of decision trees is improved by an FPGA decision tree hardware accelerator. The authors show a speed improvement in classification of around 3.5 compared to a software-only solution. Even though the authors propose a theoretical analysis of their FPGA implementation, they lack a thorough comparisons with CPUs. More specifically, the authors compare their implementation against a slow clocked Microblaze CPU without vectorization unit, which does not represent the state-of-the-art in CPU hardware.

Barbareschi *et al.* present in [12] an implementation scheme for random forests on FPGAs focusing on the fusion rule of a random forest. The authors present a fast and energy efficient way of computing a majority vote on FPGAs. Additionally, they introduce an early prediction function, which estimates the number of logic cells needed for a decision tree during tree induction. The author do not compare their findings against a software implementation.

Kim *et al.* present in [13] an implementation for binary search trees using vectorization units on Intel CPUs and compare their implementation against a GPU implementation. The authors provide insight in how to tailor an implementation to Intel CPUs by taking into account register sizes, cache sizes as well as page sizes. A comparison with other CPU manufactures or with FPGAs is missing.

III. Decision Trees and Random Forests

We consider the filtering of sensor data a binary classification problem. In binary classification problems, we want to find a prediction model $\widehat{f} : \mathcal{X} \to \{0, 1\}$, which predicts

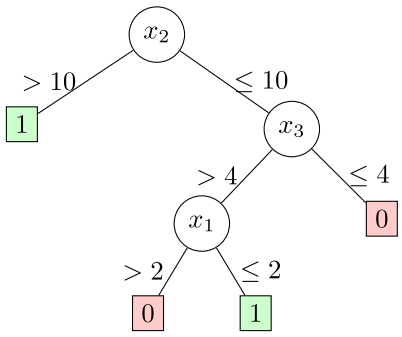

Fig. 1. Simple binary decision tree for 3-dimensional inputs. Classification is performed by starting in the root node and compare the feature x_2 of the current input $\vec{x}$. If $x_2 \leq 10$ holds to be true, classification is continued at the next level where the next comparison is performed. This process is continued until a leaf node is reached.

the class $\widehat{f}(\vec{x})$ for a given observation $\vec{x}$. In the following, we assume that observations are generated using sensors converting an analog measurement to an unnormalized unsigned integer. Therefore, we assume that sensor measurements are represented by a d-dimensional vector $\vec{x}_i \in \mathbb{N}^d$, where each dimension represents a random variable, in other words, a feature. Decision trees can be learnt form labelled training data $(\vec{x}_i, y_i)_{(i=1,\ldots,N)}$ where each tupel $(\vec{x}_i, y_i)$ consists of an observation $\vec{x}_i \in \mathcal{X}$ and the corresponding label $y_i \in \{0, +1\}$ using one of many existing algorithms such as CART, ID3, C4.5 [14]–[16].

In decision trees, the function $\widehat{f}$ is represented by a simple tree structure, where comparisons are performed on each layer and predictions are associated with leaf nodes. Starting from the root node, one starts to compare the feature x_j of the current observation $\vec{x}$ with a split value s_j. Depending on the outcome of this comparison, one either follows the left or right path of the current node. This process is repeated until a leaf node is reached and the prediction associated with the leaf is returned. An example can be found in Figure 1.

To analyse the expected runtime of different decision tree implementations, we want to use the following notation: We assign a unique identifier to each node, so that we decide at node i to either go to the left node j or right node k. Let M denote the number of leafs in a decision tree, then there are M different paths from the root node to a leaf. Every observation takes exactly one path $l(\vec{x})$ from the root node to one leaf. Please note, that all comparisons performed on a path depend on each other: at one node, we decide at which child to look in order to know which comparison we need to evaluate next.

The number of comparisons performed during a path effectively indicates the classification speed: The less comparisons we need to perform, the faster we can classify the given observation $\vec{x}$. By assuming a certain distribution of observations, we can calculate the expected number of comparisons needed for a given tree.

We consider the comparison at node i a Bernoulli experiment in which we will take the path towards the left child j with probability $p_{i\to j}$. The probability to take the path towards the right child k is then given by $p_{i\to k}$. The probabilities $p_{i\to j}$ and $p_{i\to k}$ can be estimated during training by simply counting the number of examples at each node i taking the left and right path. Furthermore, it holds that $p_{i\to j} = 1 - p_{i\to k}$. Hence, following a path consists of a series of Bernoulli experiments $l = (p_{0\to i_1}, p_{i_1\to i_2}, p_{i_2\to i_3}, \ldots, p_{i_{L-1}\to i_L})$. The probability to take path l for a given observation $\vec{x}$ is then given by

$$p(l) = p_{0\to i_1} \cdot \cdots \cdot p_{i_{L-1},L} = \prod_{j=0}^{L} l_j$$

Please note, that we drop the argument $\vec{x}$ of l if we are not interested in one specific observation $\vec{x}$. Since a tree has multiple paths, we again use a unique identifier i to denote a specific path l^i in the tree with their associated probability $p(l^i)$. In this notation, path l^i has length L^i. Therefore, the expected number of comparisons in a tree is given by

$$\mathbb{E}[L] = \sum_{i=1}^{M} p(l^i) \cdot L^i$$

Decision trees tend to overfit, i.e., they learn a way to represent the presented data effectively, without extracting real knowledge. In the extreme case, a decision tree classifies exactly one training data point per layer and thus has a height of N. On the training data, the decision tree will never give a wrong prediction. However, new, yet unseen observations seldomly match the training data and thus often receive a wrong prediction. In order to deal with this problem, decision tree induction algorithms such as CART, ID3 or C4.5 also include some pruning strategy.

Additionally, it can be shown that the combination of multiple, different models into one larger model also prevents the method from overfitting. These ensembles of many, relatively weak classifiers outperform a larger, more complex model. Random Forests combine decision trees with ensembles by training N_{Trees} decision trees, each on a different subset of features and observations. Thus, Random Forests do not suffer from over-fitting and offer state-of-the art classification accuracies on real-world problems [5].

Alternatively, one can use techniques such as bagging or boosting to create a forest of decision trees with similar characteristics also offering state-of-the-art classification accuracy [17], [18].

In order to produce a single prediction from multiple trees, one can use a suitable fusion rule. In this paper, we consider the majority vote, which returns the class that most individual classifiers predict.

IV. Computational Architectures

Today's hardware landscape offers many different processors and embedded computing devices with different characteristics and unique strengths and weaknesses. A thorough analysis of this landscape can only be performed for a specific task taking given requirements into account. Additionally, hardware manufactures are releasing new hardware every year making such an analysis quickly outdated.

In order to keep the following discussion as general as possible, we present a simple model of computation and use

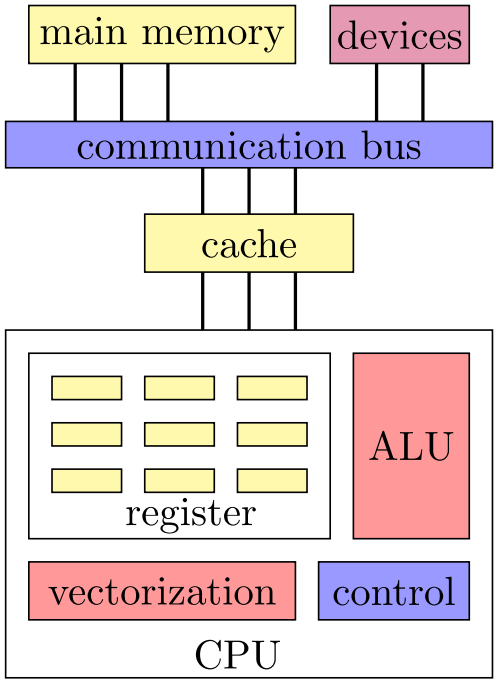

Fig. 2. Sketch of the von-Neumann architecture. The communication bus connects main memory and cache, which again is connected with the CPU. The CPU performs arithmetic logical operations on register values using the arithmetic logical unit (ALU). Vectorization instructions are performed by the vectorization unit on special vector registers. A control logic administers register accesses and instruction execution.

it throughout this paper to assess different implementations of decision trees and random forests.

We first present a theoretical model for von-Neumann CPUs and then for FPGAs.

A. Von-Neumann Architecture

The vast majority of CPUs are implemented using the von-Neumann architecture, in which code and data resides in the same memory. A comprehensive study of von-Neumann CPUs can be found in [7]. A sketch of the von-Neumann architecture is depicted in Figure 2. Code and data are fetched using a common communication bus. The control logic of the CPU decodes instructions and loads data into registers, accordingly. Operations are performed on these registers and results are written back into the main memory using the communication bus if needed. Since the common communication bus is used for data and instruction codes, it forms the *bottleneck of the von-Neumann architecture*. Moreover, an effect known as *memory wall* manifested over the years: With increasing manufacturing capabilities, CPUs have become faster and faster nearly doubling their processing power every 2 years. Memory access speed as well as memory transfer rates, however, could not keep up with this rapid speed-up, making access to main memory a magnitude slower than data processing inside the CPU (see [19]).

In a classical von-Neumann architecture, instructions are clocked, i.e., the execution of each instruction is synchronized to a common clock. von-Neumann CPUs are inherently a single instruction, single data (SISD) system, in which one instruction performs one operation on one data item per clock. In order to cope with data and computation intensive applications several extensions for von-Neumann CPUs have been introduced. In our theoretical model we will consider the following ones.

- **Memory hierarchy:** Memory is arrange in hierarchies, so that instructions and data can be fetched from different hierarchy levels. On the lower hierarchy levels, small but fast memory such as caches can be found, whereas on higher hierarchy levels, larger but slower memory such as main memory is placed.
- **Parallelized memory access:** CPUs perform operations on packs of bits called words. The word-size of a CPU thus denotes how fine-grained a CPU can access individual bits. In order to reduce address lookup, memory access is performed on packs of words in which each memory access loads neighbouring words.
- **Vectorization:** With more and more transistors available, more specialized hardware can be added to the same CPU circuit. Therefore, many CPUs offer additional single instruction multiple data operations (SIMD). These vectorization units use special registers and instructions, which perform a single operation on multiple data items, at the same time.

Given these optimizations in CPUs, we can formulate a more up to date theoretical model for the von-Neumann architecture found in today's hardware. We formalize the instructions available to this theoretical CPU together with the number of clocks needed to execute the instructions. Although we tried to keep this theoretical instruction set architecture close to existing real-world hardware, we do not restrict ourselves to instructions that are available in every CPU. A comparison between our theoretical model and real-world hardware can be found in section VII-A.

The CPU is connected via a common communication bus to main memory as well as peripheral devices. The CPU operates on words of size s_w and has registers of the same size. A cache with size M_c is used, which is organised in cache lines with size s_c which are a multiple of s_w.

The vectorization unit operates on vectorization registers with size $s_v = v \cdot s_w$, where v denotes the degree of vectorization. $load$ and $store$ instructions can be used load and store values from the cache. In case of vectorization units, the $load$ operation can only load continuous memory from the cache into the vector register.

If we want to load values from different, non-continuous memory locations, we first need to store the corresponding memory addresses into one vector register using the $load$ instruction. Once the addresses are present in one register, we can use a $gather$ instruction to load the values placed at the different memory addresses into one vector register. In order to extract scalar values from the vector registers, we can load specific $lanes$ from the register unit.

The results of vector comparisons are saved into the vector register. Since the outcome of a single comparison can be saved using one bit, we only need v bits to save the vector comparison. In order to access these v comparison bits, we use an $extract$ instruction, loading the v comparison bits into

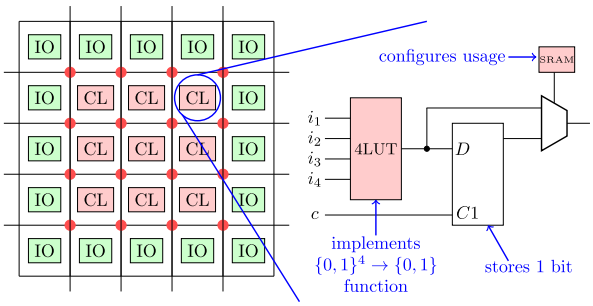

Fig. 3. Configurable logic blocks (CLB) of FPGAs. The logic block shown consists of a look-up table with 4 inputs and one flip-flop. Thus, it can represent any boolean function with 4 inputs or save 1 bit.

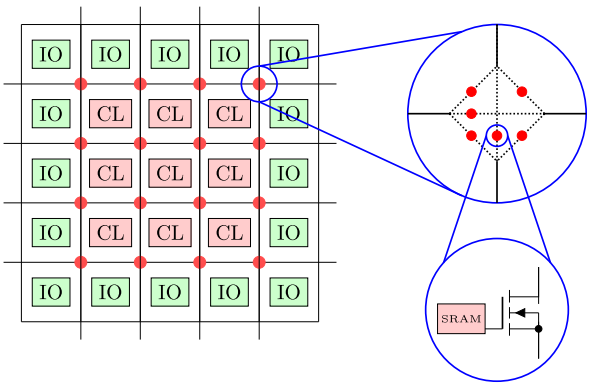

Fig. 4. Signal routing in FPGAs. Each crossing has 6 transistor attached to it, which control each of the lanes. Transistors are programmed using a 1 bit SRAM cell.

a scalar register. For arrays, we assume that they are stored in continuous memory. Memory access on arbitrary indices can also be performed with a single *load* operation, if the specific index is saved in a register.

Our theoretical CPU is also clocked. We denote the clock frequency with C^{CPU}. To further simplify our analysis, we assume that the complete random forest will fit into the cache and we will operate only on elements saved in the registers as well as in the cache. Additionally, we use the following cost model:

- **load:** Accessing a specific word of size s_w already present in the cache takes one clock cycle. Loading continuous memory into a vector register also takes one clock cycle. Loading words smaller than s_w require one load instruction and an additional *lane* access to extract smaller words.
- **gather:** The gather instruction takes one clock cycle.
- **store:** Saving one data item of size s_w or smaller inside the cache takes one clock cycle. Saving a vector register into continuous memory also takes one clock cycle.
- **compare:** Comparisons in von-Neumann architectures consists of two operations. First, the comparison is performed taking one clock cycle. Then, a conditional jump to the next instruction based on the outcome of the comparison is executed, taking another cycle.
- **arithmetic and logic operations:** Boolean operations, as well as accessing specific parts of vectorization registers take one clock cycle.

B. Custom Architectures

FPGAs are reconfigurable hardware, i.e., their functionality is encoded in hardware which can be reprogrammed before and even during execution.

Logic gates are combined with flip-flops into configurable logic cells (cf. Figure 3). Each logic cell contains a truth table of size 2^t saving a boolean function $f\colon \{0,1\}^t \to \{0,1\}$. By programming the logic table, a logic block can image every boolean function of size t. Alternatively, a configurable logic block (CLB) can be configured to act as memory, if necessary.

In order to achieve more complex circuits, the CLBs are connected to each other by signal routes. Signal routing between logic cells is performed by flip-flops and transistors that statically enable or disable signal routes (see Figure 4). The functional logic of the FPGA can be fully specified by programming the look-up tables and flip-flops for signal routing, effectively making the FPGA a reconfigurable circuit. FPGAs are functional complete [20].

FPGAs are essentially free to image every architecture needed for a given problem. This freedom, however, comes along with two major disadvantages. FPGAs only have limited resources and must mimic logic gates with truth tables. Therefore, even though FPGAs are functional complete in theory, they cannot express every function due to resource constraints. Second, FPGAs perform operations at a much lower speed than CPUs do, since they do not implement logic gates directly. In order to cope with these limitations several extensions for FPGAs have been introduced, from which we will cover the following ones in our theoretical model (cf. [21], [22]):

- **Block memory:** Configurable logic blocks are a valuable processing resources, they should not be used as memory, if possible. Therefore, FPGAs usually contain an additional block memory which can be used to save intermediate values. Access speed is similar to caches in CPUs.
- **DSP units:** Often, standard tasks such as addition or multiplication must be performed. Therefore, FPGAs offer dedicated digital signal processing units (DSP) performing these kinds of tasks if desired to save logic blocks.

Since FPGAs can be used to build any hardware architecture, they do not operate on fixed words. Data access and computation can be tailored specifically for the given task at hand. Additionally, it is not required that FPGAs are clocked. Note, however, that block memory as well as DSP units are implemented in fixed hardware and thus use a standardized clocked interface.

For a fair comparison, we take into account that the CLBs are a scarce resource. We model the accesses to block ram also with *load* instructions, which - in case of a CPU - are already given by the instruction set architecture of the CPU. In case of FPGAs, these instructions need to be implemented. Assuming we want to access a specific entry i inside an array $arr[i]$, we need to compute the address of that element first in order to issue the corresponding *load* instruction.

Given the array is stored continuously in the block ram starting at address arr and it contains data items of size s, the address of $arr[i]$ is given by $arr + i \cdot s$. Thus, we need to implement summation and multiplication for address resolution. Integer summation and multiplication is a well-studied problem in literature and many solutions such as carry-look-ahead adder, Ladner-fisher adder or Wallce trees exist (see e.g. [7, Appendix I] or [23, Ch. 12]). Additionally, the FPGAs DSP elements may already offer a corresponding multiply-accumulate operation in fixed hardware. Therefore, we assume that we can implement the address resolution mechanism with a constant delay of one clock cycle using at most r_{addr} CLB resources.

If we want to compare a single bit against a constant value 0 or 1, we can utilize the fact, that FPGAs can address arbitrary bit length. More formally, we need to realise a function $eq\colon \{0,1\}^2 \to \{0,1\}$, which can be implemented using $r_{eq} = 1$ CLBs given that $t \geq 2$.

In order to compute the inequality $a \leq b$ of two unsigned integers $a = (a_1 a_2 \dots a_{s_w})$ and $b = (b_1 b_2 \dots b_{s_w})$ we need to implement a boolean function $le\colon \{0,1\}^{2s_w} \to \{0,1\}$. To do so, we can observe, that the comparisons between a_i and b_i is independent from any other comparison of a_j and b_j in a and b, given $i \neq j$. Therefore, we can build a tree structure, in which packs of $t/2$ bits of a and $t/2$ bits of b are fed into the same CLB. Then, we can merge the output of the t CLBs and feed its output into the next layer. We repeat the process until the remaining inputs fit into one CLB.

More precisely, we build a tree with $n_l = \lceil \frac{2 \cdot s_w}{t} \rceil$ leafs and t children per node. The number of inner nodes n_i can be determined by observing:

$$n_i - n_l = \sum_{k=0}^{log_t(n_l)-1} = t^{log_t(l)} - 1 = n_l - 1$$

Therefore, the number of internal nodes is $n_i = 2 \cdot n_l - 1$. This leads to a total of

$$r_{le} = n_i + n_l = 2 \cdot \left\lceil \frac{2 \cdot s_w}{t} \right\rceil + 1 + \left\lceil \frac{2 \cdot s_w}{t} \right\rceil = \left\lceil \frac{6 \cdot s_w}{t} \right\rceil + 1$$

CLBs required to implement the comparison of two s_w-bit unsigned integers.

In CPUs, we need to perform conditional jumps based on the outcome of the comparisons. In the case of FPGAs, we can use the 1-bit output of the comparison to control the functional units directly. Thus, the comparison combined with branching a only need one clock cycle in FPGAs. Realising a register of size s_w takes s_w CLBs, as each CLB can store up to one bit. Realising a constant of s_w bit can be achieved using $\lceil \frac{s_w}{t} \rceil$ truth tables.

In summary, we assume that the FPGA is running with a clock frequency C^{FPGA} and the following operations are available:

- **load:** Loading a specific word of size s_w from the block memory takes two clock cycle (address resolution and actual memory access) and uses r_{addr} resources.
- **load:** Loading arbitrary words from registers realised with CLBs take one clock.
- **load:** Loading constant words takes zero clocks.
- **store:** Storing a specific word of size s_w in the block memory takes two clock cycle and uses r_{addr} resources.
- **store:** Storing arbitrary words in register realised with CLBs take one clock cycle.
- **store:** Storing a s_w-bit constant value in a register is performed in zero clock cycles and requires s_w CLBs (on CLB saves one bit).
- **compare:** Comparisons are performed using one clock cycle. Comparison against a constant single-bit value needs r_{eq} CLBs, whereas comparison of two s_w-bit integers r_{le} CLBs require.

The reader might wonder why loading a constant word inside the FPGA takes zero clock cycles. In order to execute a specific operation, the corresponding functional unit needs to load the desired operands first. If, however, one operand is a constant value, we can directly hard-wire this constant value into the functional unit. Therefore, loading of constant operands is not required leading to a clock delay of zero clocks.

Because FPGAs do not offer the possibility to perform address resolution combined with loading in one instruction, the access to block memory is slower compared to CPUs. In contrast, FPGAs offer the possibility to perform multiple instructions in parallel. To encourage this property, we will allow up to two concurrent accesses to block memory in one clock.

V. Implementation of Random Forests on Von-Neumann Architectures

It goes without saying, that when implementing random forests effectively on the given hardware, one has to consider the underlying computational model. We will first focus on implementing decision trees and then discuss the implementation of a majority vote.

A. Naive Implementation

Decision trees are closely related to binary search trees and thus implementations can be found in nearly every entry level computer science book (see e.g. [24]). Typically, the nodes of a tree are represented as a single entity containing the split value, a pointer to the children, as well as the prediction. An example of this entity can be found in Figure 5. We explicitly denote the corresponding data types to model the size of the node.

$boolean\_t$ denotes a boolean data type. All other types represent an unsigned integer of necessary size. Since loading words smaller than s_w requires an extra $lane$ access, we align the size of each data type towards s_w. Thus, if values below 2^{s_w} need to be represented, they are stored inside a variable of size s_w. If values above 2^{s_w} need to be represented, we can store them in the next smallest multiple of s_w necessary.

The complete size of a node is given by $s_n = 2 \cdot boolean\_t + feature\_t + split\_t + 2 \cdot node\_t$. For the following, we assume that data types fit into scalar registers of size s_w, thus only one $load$ instruction is necessary to access each field of a node.

```
1: entity Node:
2:     boolean_t prediction        ▷ true or false
3:     boolean_t isLeaf            ▷ true or false
4:     feature_t feature           ▷ feature index
5:     split_t split               ▷ split value
6:     node_t leftChild            ▷ index of left child
7:     node_t rightChild           ▷ index of right child
```

Fig. 5. Implementation of single decision tree node for the naive implementation. Please note, that the fields `leftChild` and `rightChild` point towards the next entry in the tree array.

```
1: function PREDICT(X, tree)
2:     i ← load(0)                                  ▷ 1
3:     r1 ← load(tree[0].isLeaf)                    ▷ 1
4:     while cmp_eq_false(r1) do                    ▷ 2
5:         r1 ← load(tree[i].feature)               ▷ 1
6:         r1 ← load(x[r1])                         ▷ 1
7:         r2 ← load(tree[i].split)                 ▷ 1
8:         if cmp_le(r1, r2) then                   ▷ 2
9:             i ← load(tree[i].leftChild)          ▷ 1
10:        else
11:            i ← load(tree[i].rightChild)         ▷ 1
12:        end if
13:        r1 ← load(tree[i].isLeaf)                ▷ 1
14:    end while
15:    return tree[i].prediction                    ▷ 1
16: end function
```

Fig. 6. Implementation of the prediction for a single decision tree based on a naive approach.

A tree is represented by all its nodes, which can be saved in a simple array structure. The variables $leftChild$ and $rightChild$ point to the next index in that array.

For prediction, one can traverse the tree starting from index zero until a leaf is reached. Let $tree[i]$ denote node i in the array $tree$, then we may access may access a field of node i by writing $tree[i].field$. Please note, that if i is already present inside a register, we allow `load` instructions directly on $tree[i].field$ with one clock cycle.

A naive approach implementation can be found in Figure 6.

First we initialize the intermediate register `i` to zero and load the field `tree[0].isLeaf` into another intermediate register $r1$ (line 2 and 3). Then we compare the contents of register $r1$ against `false` (line 4). If we have not reached a leaf yet, we start to execute the `while`-loop. In essence, we need to compare the current split point $tree[i].split$ with the corresponding feature value $x[tree[i].feature]$. In order to access $x[tree[i].feature]$, we first need to load $tree[i].feature$ into register $r1$ and then load $x[r1]$ into the same register (line $5-6$). Additionally, we load $tree[i].split$ into register $r2$. Once $r1$ and $r2$ contain the desired values, we can compute the comparison and branch accordingly (line $8-11$). Depending on the outcome of the branch, we update i either with $tree[i].leftChild$ or $tree[i].rightChild$. In the end of each loop iteration, we need to update the content of register $r1$ with $tree[i].isLeaf$ (line 13).

```
1: function PREDICT(X, tree)
2:     r1 ← load(x[c_1])                            ▷ 1
3:     r2 ← load(s_1)                               ▷ 1
4:     if cmp_le(r1, r2) then                       ▷ 2
5:         r1 ← load(x[c_2])                        ▷ 1
6:         r2 ← load(s_2)                           ▷ 1
7:         if cmp_le(r1, r2) then                   ▷ 2
8:             ...
9:         else
10:            ...
11:        end if
12:    else
13:        return false                             ▷ 1
14:    end if
15: end function
```

Fig. 7. Implementation of the prediction for a single decision tree based on unfolding the complete tree in its `if-else` structure. Please note, that `c_1,c_2,s_1,s_2` are constants in the source code.

Once a leaf node is reached, the corresponding prediction is returned. The number of clock cycles needed for each instruction is depicted at the end of each line. Notice, that we either need to update register i with the left or right child taking the same time and. Thus one pass through the loop uses a total of 9 clock cycles. Given the expected number of comparisons $\mathbb{E}[L]$ we see that one prediction for a tree takes

$$c_{CPU}^{Naive} = 9 \cdot E[L] + 3$$

clock cycles in total.

B. If-Else-Trees

In the naive implementation, the CPU indirectly accesses the field $tree[i].feature$ to load the corresponding feature value from x. This increases the number of clocks needed, reducing the overall throughput.

Hence, we exploit the observation, that $tree[i].feature$ and $tree[i].threshold$ are already known at compile time, which enables us to unroll the tree in its `if-else` structure and replacing $tree[i].feature$ and $tree[i].threshold$ respectively by their constant values c_i and s_i. Now, the CPU does not need to access $tree[i].feature$, but can directly load the necessary feature $x[c\_i]$. Figure 7 shows a scheme of this approach.

This approach does not require intermediate values to be saved, but only two registers are needed for every comparison (see line $2,3$ or line $5,6$). Since c_i is a constant, there is no need to compute the address for accessing x, but the CPU can directly load the array entry $x[c\_i]$ needed. Once the corresponding split value s_i is also loaded into an intermediate register, the comparison can be performed (line 4 and line 7). After this, both registers are free to be used by the next $if-$branch. Once a leaf node is reached, the prediction is given by a constant `true` or `false` which is loaded into an register for the functions caller to use.

For every $if-$branch taken, there are 4 operations and thus 4 clocks needed. Taking the expected height of the tree

```
1: entity Node:
2:     boolean_t prediction        ▷ true or false
3:     boolean_t isLeaf            ▷ true or false
4:     feature_t features[v]       ▷ v feature indices
5:     split_t splits[v]           ▷ v split values
6:     node_t children[v]          ▷ v indices of children
```

Fig. 8. Implementation of single decision tree node for a SIMD implementation. Each node saves the v child nodes on the most probable prediction path as well as its splits and feature values.

```
1: function PREDICT(X, tree)
2:     i ← load(0)                                  ▷ 1
3:     r1 ← load(tree[0].isLeaf)                    ▷ 1
4:     while cmp_eq_false(r1) do                    ▷ 2
5:         v1 ← load(tree[i].splits)                ▷ 1
6:         v2 ← load(tree[i].features)              ▷ 1
7:         v2 ← gather(x[r2])                       ▷ 1
8:         v2 ← compare(v1, v2)                     ▷ 1
9:         mask ← extract(v2)                       ▷ 1
10:        r1 ← tree[i].nextChildren                ▷ 1
11:        i ← load(r1[mask])                       ▷ 1
12:        r1 ← load(tree[i].isLeaf)                ▷ 1
13:    end while
14:    return tree[i].prediction                    ▷ 1
15: end function
```

Fig. 9. Implementation of the prediction for a single decision tree based on a naive approach.

into account, we see that `if-else` trees are expected to need

$$c_{CPU}^{if-else} = 4 \cdot E[L] + 1$$

clock cycles.

C. SIMD Implementation

`If-trees` reduce memory accesses, but still need to perform $E[L]$ comparisons to traverse the tree. The vectorization unit of the CPU can perform up to v comparisons in one clock cycle and thus offers the possibility to traverse a tree in only $\frac{E[L]}{v}$ cycles in the best case. As already mentioned, Kim *et al.* presented a SIMD implementation in [13], which we will use a basis here.

In order to utilize vector the instructions, we store up to v feature indices, v split values and v children in one node as shown in Figure 8. Since we built these structures during compile-time, we can place them in continuous memory and thus load them into the vectorization register in just one cycle.

Figure 9 shows how the prediction can be performed. First, one loads the v split values as well as the indices for the features into vector register $v1$ and $v2$. Second, we need to load all the corresponding feature values from $\vec{x}$ into another register $v2$. Since we cannot guarantee that we need a continuous part of $\vec{x}$, we need to gather different parts of $\vec{x}$ using the *gather* instruction. Third, we can perform the actual comparison, which is saved into an vector register of size s_v. Since we are only interested into the v bits corresponding to the comparison, we can use the *extract* instruction to extract a bitmask of size v. Last, we can reinterpret this bitmask as index and use it to access the next child node in the tree. Unfortunately, we cannot access $tree[i].nextChildren[mask]$ directly, since we would need to perform two array look-ups in one instruction. Thus, we need to split this indirect access and first load the base address $tree[i].nextChildren$ into register $r1$ and then perform the actual lookup depending on $r1$.

One pass through the $while-$loop takes 10 cycles. Initialization takes again 2 cycles and returning the prediction takes another clock cycles. The question remains, how many loop iterations are expected to be performed for a tree.

1) Depth-First Comparison: Given we have a skewed distribution of positive and negative labels inside the decision tree, there might be a path l from the root node to a leaf node which is taken the majority of the time. In other words, the probability of using a path l may be much higher than any other path, thus $p(l) >> p(l') \ \forall l' \neq l$.

We can utilize this fact, by performing v comparisons on the most probable path l. Then, in the best case, we can skip up to v comparisons if the first v nodes of l match the first v nodes in $l(\vec{x})$. However, in sub-optimal cases where only the first $u < v$ nodes of both paths match, we can only skip the first u nodes. In the worst case, only the first nodes in both paths match and thus we effectively performed only one comparison. Therefore, we call this type of comparison strategy a depth-first comparison, in which we always perform v comparisons on the most probable path of the tree.

The number of loop iterations we expect to perform can be expressed as the number of successful Bernoulli experiments in a row. Let l denote the most probable path in the tree and let $l[0:i]$ denote the sub-path of l which is given by the first i comparisons, then the number of expected loop iterations is given by the sum of every sub-path in l:

$$\mathbb{E}_{SIMD}[l] = \sum_{i=1}^{L} p_{l[0:i]} \cdot i$$

This leads us to an expected number of clock cycles needed:

$$c_{CPU}^{depth-first} = 10 \cdot \frac{\mathbb{E}[L]}{min(v, E_{SIMD})}$$

2) Breadth-First Comparison: In a depth-first comparison, we try to skip up to v comparisons in one clock cycle. This approach works well, if there are only a few paths in the tree which are taken the majority of the time. If the decision tree is more balanced, we are more likely to take different paths with each observation $\vec{x}$ and thus will effectively only perform one comparison per clock.

In order to utilize SIMD instructions in a more controlled way, we can perform multiple comparisons on different paths in the same instruction. We can do this, because each node in the tree has only 2 children. Regardless of the outcome of the comparison at a parent node, we will have to perform one of the two comparisons given by the children.

Thus, if we perform the comparison at the current node, as well as both children with one vectorization instruction, we make sure, that two of tree comparisons are useful. More formally, we are guaranteed to skip at least $\lfloor log_2(v+1) \rfloor$ comparisons with one instruction. In case v is not a exponential

```
1:  function MAJORITYVOTE(X)
2:      predictions[2]
3:      predictions[0] ← load(0)                          ▷ 1
4:      predictions[1] ← load(0)                          ▷ 1
5:      r1 ← predict(x, tree_1)                           ▷ c_1^m
6:      r2 ← load(predictions[r1])                        ▷ 1
7:      r2 ← r2 + 1                                       ▷ 1
8:      store(predictions(r1), r2)                        ▷ 1
9:      ...
10:     r1 ← predict(x, tree_N_Trees)                     ▷ c_{N_Trees}^m
11:     r2 ← load(predictions[r1])                        ▷ 1
12:     r2 ← r2 + 1                                       ▷ 1
13:     store(predictions(r1), r2)                        ▷ 1
14:     r1 ← load(predictions[0])                         ▷ 1
15:     r2 ← load(predictions[1])                         ▷ 1
16:     return cmpIe(r1, r2)                              ▷ 1
17: end function
```

Fig. 10. Implementation of a majority vote.

of 2, we still have some comparison entries left in the vectors. We can use the remaining $m = v - (2^{\lfloor log_2(v+1) \rfloor} - 1)$ slots in the vectorization units and use them to perform the most probable comparisons on the next layer. Given the highest probabilities $p_1, \ldots, p_m$ of that layer, we will match one of the comparisons needed with expectation $\sum_{i=1}^{m} p_i$. This leads us in total to an expected number clock cycles needed for a breadth-first approach of

$$c_{CPU}^{breadth-first} = 10 \cdot \frac{\mathbb{E}[L]}{\lfloor log_2(v+1) \rfloor + \sum_{i=1}^{m} p_i}$$

D. Random Forest Implementation

So far, we discussed the implementation of decision trees. We can use these implementations as building blocks to implement a random forest, by performing every prediction successively and by keeping track of the current vote count.

Figure 10 shows the implementation of a majority vote. First, we create an array with 2 elements for counting every single vote and initialize it with zeros (line $3-4$). Then, we perform the prediction using the first tree and save the result in an intermediate register $r1$ (lines 5 and 6). The prediction of this tree is used to address the array location for the vote count and update the count accordingly (lines $7-8$). This procedure is repeated for every tree in the random forest. In the end, we need to load the two vote counts and compare them with each other (line $14-16$). Please note, that in case we have equal vote counts, we will predict `false`, meaning we do not want to filter the given data item $\vec{x}$, but further process it.

Let c_i^m denote the number of cycles needed for method m and tree i, then we need a total number of cycles for the majority vote of:

$$C_{CPU}^{total} = 2 + \sum_{i=1}^{N_{trees}} c_i^m + 3 = 5 + \sum_{i=1}^{N_{trees}} c_i^m$$

VI. Implementation of Random Forests on FPGAs

Field programmable gate arrays offer reconfigurable hardware and thus do not offer any computing architecture, but are free to mimic every architecture needed. In an naive approach, we can simply reuse the implementations presented to far and find the same theoretical conclusions as discussed.

This however, would not take the very flexible nature of FPGAs into account. First, it has to be noted, that register accesses do not need to be aligned towards given word sizes s_w or vector sizes s_v, but they can exactly be tailored to the problem at hand. Thus, we can use the same node entity depicted in Figure 5, but can also tailor the corresponding data sizes exactly to the specific problem at hand.

Similar to the CPU implementation, we assume that the complete random forest fits into block ram on the FPGA and thus, we only operate on block memory or the logic blocks of the FPGA.

A. Naive Implementation

First, we analyse the naive implementation shown in Figure 6 for FPGAs. The initialization before the $while-$loop can be executed during power-on of the FPGA and thus takes no time (line $2-3$).

Unlike for the CPU, we can issue two *load* instructions at the same time on the FPGA, but each load takes two clock cycles because of address resolution. Thus, we can perform the first *load* operations on $r1$ and $r2$ in parallel (line 5 and 6). However, the remaining *load* instructions depend on each other and thus need to be executed in sequence, leading to a sequence of 4 *load* instructions inside the $while-$loop. Additionally, two comparisons are performed (line 4 and 8), which take 1 clock cycle each, leading to:

$$C_{FPGA}^{Naive} = (0 + 2 \cdot 4 + 2) \cdot \mathbb{E}[L] + 1 = 10 \cdot \mathbb{E}[L] + 1$$

In order to estimate the resources used by this implementation, we observer that two functional units for comparisons and two functional units for address resolution are needed. Additionally, 3 registers are needed giving a total of

$$r^{Naive} = 2 \cdot r_{addr} + 3 \cdot s_w + r_{eq} + r_{le}$$

resources used by this implementation.

B. If-Else Implementation

In the naive implementation we observer, that memory access is a costly operation since we have to perform address resolution first. Similar to the CPU implementation, we can bypass this problem if we unroll the tree in its $if-else-$structure as depicted in Figure 7.

We observe, that we need to access a constant s_i, which is known at compile time. Similar we see, that we need to access $x[c\_i]$, where c_i is also a constant. Thus, the memory address of $x[c\_i]$ is known at compile time and no address resolution unit is required.

So far, we implicitly assumed that the observation $\vec{x}$ has already been copied into the CPUs cache and the FPGAs block memory. It is reasonable to assume, that we could instead copy it directly into the FPGA CLBs cells, because we need

to communicate with the FPGAs in any case.[2] Then, we can hard-wire the entries in $\vec{x}$ and their split values s_i directly into the corresponding comparators. This way, the operands for comparison do not need to be loaded, but only the comparison itself need to be performed taking 1 clock cycle in total:

$$c_{FPGA}^{if-else} = E[L] + 1$$

Since we hard-wire constants into the comparison blocks, we cannot reuse any comparison unit and thus have to implement the comparison of every node of the complete tree. Additionally, we need to store all split values s_i as well as the entire observation $\vec{x}$ inside the CLBs of the FPGA.Given we implement a tree with n nodes, we need

$$r_{FPGA}^{if-else} = n \cdot r_{le} + n \cdot \left\lceil \frac{s_w}{t} \right\rceil + d \cdot s_w$$

resources.

C. SIMD Implementation

FPGAs are well suited for vector operations and thus the SIMD implementation of the CPU can directly be mapped onto the FPGA as presented in Figure 9. In order to access v features at once, we again store the complete tree using the FPGAs CLBs. However, unlike $if-else$ trees, we need to keep track of the current node i and issue $load$ instructions towards the memory generated using CLBs. Thus, memory access still has a delay of two clock cycles.

Again, the number of expected loop iteration depends on the comparison strategy. However, the number of clock cycles per iteration is different. The two load operations before entering the $while-$loop can be performed during power-on and thus require no clock cycles (line 2, 3). Loading values into register $v1$ and $v2$ can be performed in parallel (line $5, 6$) taking 2 cycles. Additionally, we need to gather the entries in x, taking another 2 cycles. The comparisons in line 4 and 8 can be performed and one clock cycle each. Generating the corresponding bitmask from the comparison is for free on the FPGA, since we can hard-wire the comparison bits directly it into the next functional unit. After that, 3 loading operations are performed in sequence, leading to a total of 12 clock cycles.

Looking at the resource consumption of this implementation, we can observer that we need a total of v functional units for address lookup, since the gather instructions performs v lookups in parallel. Also, we need to perform v comparisons using v comparators. Additional, another comparator, which compares $r1$ against a fixed value is required. Last, we need to materialize all nodes, as well as the vector registers and $r1$ and i with the FPGAs logic cells leading to a total of:

$$r_{FPGA}^{SIMD} = v \cdot r_{addr} + v \cdot r_{le} + v \cdot r_{eq} + n \cdot node_t + 2 \cdot v \cdot s_w + 2 \cdot s_w + d \cdot s_w$$

D. DNF Implementation

So far, we mapped CPU implementations on FPGAs, not taking the FPGAs flexible nature into account. To do so, we can formulate an extreme case of the SIMD trees, in which we perform all comparisons in one clock cycle.

[2]Assuming $\vec{x}$ is always copied into the block ram due to hardware constraints, we can load it into the FPGAs CLBs in $\frac{d}{2}$ clock cycles.

```
1: function PREDICT(X, tree)
2:     r1 ← cmpIe(r1, s_1)          ▷ 1
3:     r2 ← cmpIe(r2, s_2)          ▷ 1
4:     ...
5:     return computeDNF(r1, r2, ...)   ▷ 1
6:     return r1                    ▷ 1
7: end function
```

Fig. 11. Implementation of a decision tree by its DNF structure.

This can be done, by observing two things: First, the comparisons in a tree do not depend on each other. Only once we perform the actual prediction, we need to traverse the tree. Thus, we can perform all comparisons given $\vec{x}$ first and then traverse the tree given the pre-computed comparisons. Second, a tree can be represented by a disjunctive normal form (DNF). In a disjunctive normal form, a boolean function is represented as a series of conjunctions connected by disjunctions. We can view the comparison performed at node i as boolean variable, which is either true or false. Then, a particular path from the root node to a leaf is represented as conjunction of all - possibly inverted - comparisons on that path.

Let c_i denote the comparison for node i, the DNF of the tree depicted in Figure 1 is for example $c_{x_2} \vee (\neg c_{x_2} \wedge \neg c_{x_3} \wedge c_{x_1})$.

The DNF for a given tree is independent of the specific observation $\vec{x}$, but only depends on the structure of the tree. Thus, we can pre-compute the DNF formula for a tree and possibly optimize it by the means of boolean function minimization using e.g. the Quine-McCluskey algorithm [25].

Figure 11 shows an implementation of this approach. Similar to the $if-else$ trees, we can store the observation $\vec{x}$ inside the FPGAs CLBs and hard-wire the split values and features directly into the comparators.

Thus, given n nodes in the tree, we have to perform n comparisons in parallel taking only 1 clock cycles. The DNF computation of a tree thus consists of three operations. First, necessary variables need to be inverted, then conjunctions are computed and finally the disjunction can be evaluated. In an naive approach this takes 3 clock cycles, but since the tree structure is known, we can directly encode this into the look-up tables of the FPGA in a similar fashion as explained in sectionVI-B. Thus, computing the DNF takes only one clock cycle. Adding another clock cycle for returning the prediction value, we see, that DNF-Trees only need

$$c_{FPGA}^{DNF} = 3$$

clock cycles.

In order to traverse the complete tree in only one clock cycle, we traded run-time for space. In short, we need n comparators performing comparisons in parallel. Additionally, we need n s_w bits to materialize all split values using the FPGAs CLBs and $d \cdot s_w$ bits to store the observation $\vec{x}$.

Computing the DNF for the tree can be viewed as computing a boolean function with $2^{\mathbb{E}[L]}$ input variables $DNF\colon \{0,1\}^n \to \{0,1\}$. As already explained in section IV-B, we can distribute this across multiple logic blocks. In total, this

implementations needs

$$r_{SIMD}^{DNF} = n \cdot r_{le} + n \cdot s_w + d \cdot s_w + \left\lceil \frac{6 \cdot n}{t} \right\rceil + 1$$

resources.

E. Random Forest Implementation

For FPGAs we will follow the same general procedure as for von-Neumann CPUs shown in Figure 10 to implement the majority vote. However, unlike CPUs, FPGAs do not need to execute trees in sequence, but can execute the trees in fully parallel.

Every tree computes a prediction either corresponding to $true$ or $false$, which can be stored in a single bit, which then can be interpreted as bit vector with N_{Trees} bits. In order to compute the majority vote, we need to count the number of 1-bits inside this bit vector. This is known as the hamming weight of a bit vector and has been studied in literature [26].

Similar to comparing two integers, the hamming weight of a bit vector with N_{Trees} bits can be implemented using $r_{ham} = \lceil log_2(N_{Trees} + 1) \rceil$ CLBs, taking one clock cycle to complete its operation. After that, we can compare the value against $\lfloor \frac{NTrees}{2} \rfloor$, also taking one clock cycle requiring r_{le} resources. Thus, the total clock delay of a random forest given method m is given by:

$$C_{total}^{FPGA} = C_{FPGA}^{m} + 2$$

The amount of resources used is given by:

$$r_{total}^{FPGA} = r^m + r_{ham} + r_{le}$$

VII. Evaluation

In this section, we want to evaluate our findings and present a recommendation for FACT.

First of all, it has to be noted that the theoretical model presented can be used to compare different implementations in terms of resources and clock cycles needed, but does not entirely reflect the run-time on real-world hardware.

von-Neumann CPUs are usually pipelined and thus may perform multiple instructions at the same time. Additionally, we did not incorporate caching effects into the model. In case of FPGAs, our model is closer to real-world hardware. However, it has to be mentioned that it is possible to implement circuits, which theoretical fit onto the FPGA, but cannot be routed by the synthesis tool. Our model does not reflect this. Also, we do not include boolean optimizations performed by the synthesis tool in our model.

A. From Theory to Real-World Hardware

In this section, we want to provide evidence, that our theoretical model matches real-world hardware. To do so, we shortly discuss the specifics of selected real world hardware and compare it against the theoretical model we presented.

1) X86 CPUs: A large portion of CPUs available today implement the X86 instruction set architecture (ISA), which mainly targets server and desktop systems. An overview of the X86 instruction set architecture supported by Intel CPUs can be found in [27].

In today's X86 implementations, we find scalar registers with 64 bit and vectorization register ranging from 128 bit up to 512 bit. Loading values into scalar register is achieved using the MOV instruction, which supports address resolution for memory access in one clock cycle. Similar, vectorization registers are loaded using one of the many vectorial counterparts of the MOV instruction, e.g. $MOVDQA$ in the SSE extension.

Performing scalar comparison is achieved by first performing a comparison using the CMP instruction and then by performing the corresponding jump to the next code segment, e.g. using the je (jump-if-equal) instruction.

Performing vectorial comparisons can be implemented by using the corresponding vector instructions - in case of SSE for example $PCMPGTB$. The resulting bitmask can be extracted using the $MOVMSKPS$ instruction in SSE.

The support for gather instructions is available with the Advanced Vector Extensions 2 (AVX 2), which is for example implemented in Intel Haswell CPUs. Concluding this short summary, we are confident, that our theoretical model captures the characteristics of X86 CPUs.

2) ARM CPUs: The majority of CPUs used in mobile and embedded systems implement the ARM instruction set architecture which is specifically designed for embedded systems. An overview of the ARM ISA can be found in [28].

Until 2011 the ARM ISA supported 32 bit, but switched then towards 64 bit registers. A vectorization extension called NEON with 128 bit registers is also available.

Similar to the X86 architecture, MOV instructions are used to load memory content into registers. The MOV instruction may perform address resolution and thus can also load memory content in one clock cycle.

In order to load content into the vectorization registers, the VLD instruction can be used. Comparisons work the same as in X86 CPUs: First the comparison is performed using the CMP instruction and then a conditional jump is performed, e.g. using the BLT instruction. In case of vector comparison, the NEON extension provides a $VCMP$ instruction. Unfortunately, there is no dedicated instruction to extract the comparisons bitmask. This instruction must be emulated with individual lane accesses on the registers. Also, there is no $gather$ instruction in NEON. In summary, we think that the proposed model captures most of the characteristics of modern ARM CPUs. However, it has to be noted, that vectorization instructions do not match exactly.

3) Xilinx FPGAs: As one of the largest FPGA manufactures, Xilinx offers variety of different FPGA models [22].

Xilinx combines 4 look-up tables with 6 inputs and 1 output into one CLB and offers models with 2000 to 33650 CLBs. Additionally, between 720kb to 13140kb block memory is available. Block memory is implemented in 36kb dual-port memory cells, meaning two separate 18 kb address spaces can be accessed at the same time.

Adding to this, 40 to 740 DSP elements including an pre-adder, a multiplier, an adder as well as an accumulator are available.

Therefore, we are confident, that the presented theoretical model captures the characteristics of modern FPGAs.

VIII. Architectural Dependent Code Generation

In this section, we briefly discuss a code generator which generates code for the presented implementation. As ARM is prominent in embedded systems, we will focus on $native-$trees, $if-else-$ trees and $DNF-$trees.

A. Code Generator

Given a trained decision tree classifier or random forest classifier, we wish to generate an efficient implementation of that classifier for a given architecture, without changing its accuracy. For CPUs, we can either use the ISA of a processor directly by the means of assembler or use a higher level language such as `C/C++`. Since compiler optimization can help improving the overall throughput we generate `C++` code for a given architecture and compile this for the specific processor at hand.

For FPGAs we can either directly generate `VHDL` code or use `C/C++` code as a basis and perform high level synthesis. `VHDL` offers fine control over the generated code and gives us full control over the FPGAs resources. `C/C++` on the other hand lets the high level synthesis tool utilize the block ram and DPS units to the fullest, as it generates board specific code. Additionally, we can structure the generated `C/C++` code so that the high level synthesis tool is able to perform further optimizations (c.f. [29]). Thus, as a first step we will use `C/C++` code in combination with a high level synthesis tool for FPGA code generation. We implemented the code generator in `python` with code templates in `C/C++`. The code generator loads a decision tree or random forest from `sklearn`[3] into an internal structure and automatically detects necessary data ranges for features and splits. Additionally, it is possibly to load trees and forests directly from `JSON` files.

The generated implementation can then be compiled using a standard `C/C++` compiler or the high level synthesis tool.

B. Recommendation for FACT

So far, we presented and discussed all implementations independent from each other. In this section, we want to shortly compare the implementations with each other and provide a hardware and implementation recommendation for filtering of sensor data in the context of FACT.

The FACT telescope performs 300 measurements per second with 1440 sensors, each with a $12-$bit resolution [2] resulting in a data rate of 4.944 Mbps. For filtering-out unwanted events, we trained a random forest classifiers with $N_{Trees} = 50$ decision trees on $N = 6000$ training instances containing unnormalized raw data. To improve the quality of the trees, we prepared the training data to contain 50% of background noise and 50% wanted events. We used the sklearn-learn library for tree induction. Each tree contains an average of 1349 nodes and roughly 675 different paths from the root node to a leaf node. A 10-fold cross-validation shows, that the trained ensemble offers a prediction accuracy close to 80% for gamma-hadron separation.

However, for filtering we are more interested in the number of correctly filtered events. Since we do not want filter-out interesting events, we only discard an element, if the random forest classifiers is certain in its classification. We interpret the number of positive and negative labels in the leafs of a decision tree as the probability of the respective class. For the forest, we can average this probability for all trees, giving a prediction reliability for the class. Thus, we only filter out events, in which the predicted class receives a probability of at least 0.68.

This way, we can filter out roughly 12% of the data before further processing, while performing no misclassification reducing the data rate to 4.35 Mbps.

The question remains, which hardware and which implementation would be the best fit for the trees given. First we will compare the implementations based on the theoretical analysis:

Judging by the number of clock cycles, the FPGA clearly seems to be at an advantage, if we either implement $if-else$ trees or one of the SIMD derivatives. For $if-else$ trees we need $n \cdot r_{le} + n \cdot \left\lceil \frac{s_w}{t} \right\rceil + d \cdot s_w = 1349 \cdot (\left\lceil \frac{6 \cdot 12}{6} \right\rceil + 1) + 1349 \cdot 12 + 1440 \cdot 12 = 51005$ CLBs per tree. Therefore, we identify that $if-else$ simply will not fit into smaller FPGAs.

In case of SIMD derivatives, we see that $dnf-trees$ will also not fit, since they use more CLBs compared to $if-else$ trees. The amount of resources needed by a SIMD tree depends on v. Using $v = 8$, we see that a tree needs $8 \cdot r_{addr} + 8 \cdot \ (\left\lceil \frac{6 \cdot 12}{6} \right\rceil + 1) + 8 \cdot 1 + 1348 \cdot node\_t + 2\ cdot2 \cdot 12 + 2 \cdot 12 + 1440 \cdot 12 = 8 \cdot r_{addr} + 1348 \cdot node_t + 17464$ logic blocks. Since we have 1348 nodes in a tree, we need at least 11 bits to index each node. In order to index 1440 features, we also need 11 bits. Thus we can assume, that $node\_t \approx 47$. Therefore, a SIMD implementation on FPGAs needs at least $1348 \cdot 47 + 17464 = 80820$ CLBs, which is also unlikely to fit on smaller FPGAs.

Thus, we conclude, that for the particular problem at hand, FPGAs cannot be employed. In case of a classical CPU, we find that $if-else$ trees offer a fast and reliable clock delay. For the presented trees, we see that we need $4 \cdot 13 + 1 = 53$ clock cycles on average, leading to a total of 2650 clocks needed to compute the complete forest. Since 300 measurements are needed to be evaluated per second, we need a processor with at least 795000 $Hz = 0.795\ Mhz$. Thus, a small, embedded system with $\approx 1\ Mhz$ clock speed will do the job.

C. Experiments for FACT

In order to critically evaluate the presented theoretical model, we compare three different implementations for FACT using the Zedboard[4] The Zedboard contains an ARM Cortex-A9 with 666 Mhz, 512 Mb DDR RAM and 512 Kb cache. Additionally, this board contains a Xilinx Artix-7 Z-7020 FPGA with 53200 lookup tables, 106400 flip-flops (FF) in total combined with 4.9 Mb block ram and 220 DSP units. The Zedboard contains four Adavanced eXtensibel Interface (AXI) ports, each running at 142 Mhz with 32 bit word sizes leading to a aggregated bandwidth up to 3.8 GB/s.

We used the software `SDSoC` in version `2016.2` to run and compile the experiments. Power consumption was

[3] http://scikit-learn.org/

[4] http://zedboard.org/

TABLE I

THROUGHPUT COMPARISON FOR DIFFERENT IMPLEMENTATIONS OF RANDOM FORESTS AND DECISION TREES. LARGER IS BETTER

	FPGA $\left[\frac{\text{elem}}{\text{ms}}\right]$	ARM $\left[\frac{\text{elem}}{\text{ms}}\right]$
If-Tree	$1480 \pm 2.7 \cdot 10^{-9}$	29000 ± 0.0027
If-Tree-Forest	-	$780 \pm 2.7 \cdot 10^{-9}$
Native-Tree	1170 ± 0.00034	14500 ± 0.00054
Native-Tree-Forest	-	$460 \pm 4.9 \cdot 10^{-9}$
DNF-Tree	$1100 \pm 4.7 \cdot 10^{-10}$	$1900 \pm 4.9 \cdot 10^{-9}$
DNF-Tree-Forest	-	$30 \pm 1.4 \cdot 10^{-13}$

TABLE II

RESOURCE COMPARISON FOR DIFFERENT IMPLEMENTATIONS OF RANDOM FORESTS AND DECISION TREES. SMALLER IS BETTER

	FPGA				ARM [MB]
	LUT	FF	BRAM	DSP	
If-Tree	10244	1317	0	0	1.3
If-Tree-Forest	-	-	-	-	2.4
Native-Tree	31	64	5	0	1.3
Native-Tree-Forest	-	-	-	-	1.8
DNF-Tree	9609	3183	0	0	1.4
DNF-Tree-Forest	-	-	-	-	6.8

estimated using `Vivado` in version `2016.2`. All experiments where performed in the standalone mode of this board, so that no operating system is involved during measurements. We activated the most aggressive optimizations `-O3`. FPGA implementations are clocked with 100 Mhz giving an actual bandwidth of up to 1.6 GB/s.

Table I depicts the average classification throughput in measurments per millisecond for a random forest and for a single decision tree. All tests were repeated 20 times.

One can observe that the $if-else$ trees offer the highest throughput for single trees, as well as for random forests. $Native$ tree implementations also do fairly well on the CPU, whereas $dnf-$ trees offer the smallest throughput.

Looking at the FPGA, we first see that the random forest implementations did not fit onto the FPGA. Thus, the corresponding entries in Table I are missing. The decision tree implementations all fit on the FPGA with a throughput ranging from 1100 to 1480, where $if-else$ trees are the fastest and $dnf-$trees the slowest implementation.

Table II depicts the resource usage of all implementations. For the FPGA we depict the resource usage reported by the synthesis tool. For the CPU, we present the binary size, which is loaded by the first-stage bootloader directly after power-on of the board. Please note, that since the board is used in standalone mode, the binary contains all necessary libraries, e.g. functions for time measurements and output over UART.

One can see, that the $native-$tree uses the least resources fitting up to 40 trees of a random forest onto the FPGA. The dnf-trees as well as $if-else$ trees also fit nicely on the FPGA, but use more resources than $native$-trees. With this implementation, one could roughly fit 5 trees on the FPGA.

Looking at the CPU, one can observe that the binary sizes are around 1.3 MB to 2.4 MB. The dnf-tree forest implementation is an exception in this regard with 6.8 MB. Last, table III displays the power consumption of all models. It is well established that the complete zedboard uses around $4-6$ W in total [30], [31]. This also takes peripheral devices such audio controller or VGA controller into account, which are not needed during deployment for FACT. Therefore, we want to focus on the energy consumption of the ARM processor as well as the FPGA. Direct measuring of these quantities is difficult, because these parts are integrated on the board, so that we will rely on the estimations of the power consumption given by the synthesis tool. We report all estimates for the maximum power consumption during full load. The complete chip uses in total less than 2 W in all configurations, from which the ARM processor uses 1.53 W. The FPGA implementations greatly vary in power consumption ranging from 0.008W to 0.068W, but are all two to three magnitudes lower than what the ARM processor needs.

TABLE III

POWER CONSUMPTION FOR DIFFERENT RANDOM FORESTS AND DECISION IMPLEMENTATIONS. SMALLER IS BETTER

	Power [W]		Power per Element $\left[\frac{\text{nJ}}{\text{elem}}\right]$	
	FPGA	ARM	FPGA	ARM
If-Tree	0.068	1.53	45.95	52.76
If-Tree-Forest	-	1.53	-	1961.54
Native-Tree	0.008	1.53	6.84	105.51
Native-Tree-Forest	-	1.53	-	3326.08
DNF-Tree	0.023	1.53	20.9	805.26
DNF-Tree-Forest	-	1.53	-	51000

Using the throughput measurements from table I, we compute the amount of energy needed to process one measurement. Here one can observe, that on the ARM the $if-else$ implementation offers smallest power consumption because of the large throughput of this implementation. On the FPGA however, the $native$ implementation dominates, as this also uses the least resources. All in all, we see that the FPGA - despite smaller throughput - wins in terms of energy per element for all implementations.

IX. DISCUSSION

In this paper, we investigated when to use FPGAs and when to use CPUs. We evaluated the classical von-Neumann CPU architecture and presented a simple model of computation. In a similar fashion, we examined the characteristics of FPGAs and formulated a theoretical model also for them.

We specifically focused on random forests and decision trees and presented different implementation schemes for each architecture. Based on our theoretical model, it is now possible to estimate the number of clock cycles needed for a given tree taking the distribution of observations into account.

This theoretical model is the first step towards the analysis of different algorithms on an architectural basis without the need of implementing or synthesising the algorithms directly. In future work we plan to extend this model taking caching effects and multi-threading into account.

We applied our theoretical analysis in the context of astro-particle physics, namely for the FACT telescope. Here, we showed, that the learned random forests do not fit onto FPGAs. In contrast, a small embedded system clocked at roughly 1Mhz offers enough computational power to filter

the FACT data, if all trees are unrolled into their $if-else$ structure.

We validated our theoretical model with experiments on real world hardware. The experiments' results largely follow our theoretical model, but also display potential for further work.

First, the presented theoretical model does not exactly match the throughput measured, but it underestimates the CPU. As argued, the evaluation of $if-else$ trees for the random forest need 2650 clock cycles. Looking at the achieved throughput, we see that the implementation only needs around 854 clock cycles, which is roughly 3 times faster than expected. The difference can be explained by the lack of pipelining in our theoretical model. Further work is to enhance the model accordingly. It has to be noted, however, that our model still captures the relative differences between implementations and thus still can be used to determine the fastest implementation.

Second, in terms of the FPGAs resource usage, our model also needs some adjustments with respect to the optimizations of the synthesis tool, which we did not consider in our model. Additionally, one can argue that parts of a tree can be reused in some fashion, e.g. reusing feature indices, which is also not reflected by our model.

Last, we have seen that FPGAs are clearly the best approach in terms of energy consumption per element. The reasons for this are quite clear, as the FPGA is slower clocked and does not need to decode or fetch instructions, but perform classifications directly. From this point of view, FPGAs are a good alternative for implementing decision trees, when it comes to energy consumption per element, if they can keep up with the high volume of data.

Therefore, we conclude this paper with the following recommendations:

- **high-throughput application:** We showed that von-Neumann CPUs combined with an $if-else$ implementation offers the highest throughput with reasonable energy consumption per element. Hence, we recommend this approach for applications with an emphasis on throughput.
- **low power application:** We showed, that FPGAs combined with a $native$ implementation offer the smallest power consumption per element with acceptable throughput. Thus, we recommend this approach for applications with an emphasis on power consumption.

REFERENCES

[1] M. Weiser, "The computer for the 21st century," *Sci. Amer.*, vol. 265, no. 3, pp. 94–104, 1991.

[2] C. Bockermann *et al.*, "Online analysis of high-volume data streams in astroparticle physics," in *Proc. Eur. Conf. Mach. Learn. (ECML)*, 2015, pp. 100–115.

[3] A. Vellido, J. D. Martín-Guerrero, and P. J. Lisboa, "Making machine learning models interpretable," in *Proc. ESANN*, vol. 12. 2012, pp. 163–172.

[4] C. Apté and S. Weiss, "Data mining with decision trees and decision rules," *Future Generat. Comput. Syst.*, vol. 13, no. 2, pp. 197–210, 1997.

[5] L. Breiman, "Random forests," *Mach. Learn.*, vol. 45, no. 1, pp. 5–32, 2001.

[6] R. Isermann and M. Münchhof, *Identification of Dynamic Systems: An Introduction with Applications*. Springer, 2011.

[7] J. Hennessy and D. Patterson, *Computer Architecture: A Quantitative Approach*. Amsterdam, The Netherlands: Elsevier, 2011.

[8] R. Narayanan, D. Honbo, G. Memik, A. Choudhary, and J. Zambreno, "An FPGA implementation of decision tree classification," in *Proc. Design, Autom. Test Europe Conf. Exhibit. (DATE)*, 2007, pp. 1–6.

[9] B. van Essen, C. Macaraeg, M. Gokhale, and R. Prenger, "Accelerating a random forest classifier: Multi-core, GP-GPU, or FPGA?" in *Proc. IEEE 20th Annu. Int. Symp. Field-Program. Custom Comput. Mach. (FCCM)*, May 2012, pp. 232–239.

[10] R. Prenger, B. Chen, T. Marlatt, and D. Merl, "Fast map search for compact additive tree ensembles (CATE)," Lawrence Livermore Nat. Lab. (LLNL), Livermore, CA, USA, Tech. Rep., 2013.

[11] F. Saqib, A. Dutta, J. Plusquellic, P. Ortiz, and M. S. Pattichis, "Pipelined decision tree classification accelerator implementation in FPGA (DT-CAIF)," *IEEE Trans. Comput.*, vol. 64, no. 1, pp. 280–285, Jan. 2015.

[12] M. Barbareschi, S. Del Prete, F. Gargiulo, A. Mazzeo, and C. Sansone, "Decision tree-based multiple classifier systems: An FPGA perspective," in *Proc. Int. Workshop Multiple Classifier Syst.*, 2015, pp. 194–205.

[13] C. Kim *et al.*, "FAST: Fast architecture sensitive tree search on modern CPUs and GPUs," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 2010, pp. 339–350.

[14] L. Breiman, J. H. Friedman, R. A. Olshen, and C. J. Stone, *Cart: Classification and Regression Trees*. Belmont, CA, USA, Wadsworth, 1984.

[15] J. R. Quinlan, "Induction of decision trees," *Mach. Learn.*, vol. 1, no. 1, pp. 81–106, 1986.

[16] J. R. Quinlan, *C4.5 Programs for Machine Learning*. Amsterdam, The Netherlands: Elsevier, 2014.

[17] L. Breiman, "Bagging predictors," *Mach. Learn.*, vol. 24, no. 2, pp. 123–140, 1996.

[18] Y. Freund and R. E. Schapire, "A desicion-theoretic generalization of on-line learning and an application to boosting," in *Proc. Eur. Conf. Comput. Learn. Theory*, 1995, pp. 23–37.

[19] S. Borkar and A. A. Chien, "The future of microprocessors," *Commun. ACM*, vol. 54, no. 5, pp. 67–77, May 2011.

[20] S. Hauck and A. DeHon, *Reconfigurable Computing: The Theory And Practice of FPGA-Based Computation*. San Mateo, CA, USA: Morgan Kaufmann, 2008.

[21] D. F. Bacon, R. Rabbah, and S. Shukla, "FPGA programming for the masses," *Commun. ACM*, vol. 56, no. 4, pp. 56–63, 2013.

[22] *7 Series FPGAS Overview*, accessed on Dec. 2016. [Online]. Available: https://www.xilinx.com/support/documentation/data_sheets/ds180_7Series_Overview.pdf

[23] N. Weste and D. Harris, *CMOS VLSI Design: A Circuits and Systems Perspective*, 4th ed. New York, NY, USA: Addison-Wesley, 2010.

[24] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*, 3rd ed. Cambridge, MA, USA: MIT Press, 2009.

[25] E. J. McCluskey, "Minimization of Boolean functions," *Bell System Tech. J., The*, vol. 35, no. 6, pp. 1417–1444, Nov. 1956.

[26] V. Sklyarov and I. Skliarova, "Digital Hamming weight and distance analyzers for binary vectors and matrices," *Int. J. Innov. Comput., Inf. Control*, vol. 9, no. 12, pp. 4825–4849, 2013.

[27] *Intel Intrinsics*, accessed on Dec. 2016. [Online]. Available: https://software.intel.com/sites/landingpage/IntrinsicsGuide/

[28] *Arm Neon Instructions*, accessed on Dec. 2016. [Online]. Available: http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.dui0489c/CJAJIIGG.html

[29] Xilinx. *Ug902 Vivado High Level Synthesis*, accessed on May 2017. [Online]. Available: https://www.xilinx.com/support/documentation/sw_manuals/xilinx2016_2/ug902-vivado-high-level-synthesis.pdf

[30] J. Monson, M. Wirthlin, and B. L. Hutchings, "Implementing high-performance, low-power FPGA-based optical flow accelerators in C," in *Proc. IEEE 24th Int. Conf. Appl.-Specific Syst., Archit. Process. (ASAP)*, Jun. 2013, pp. 363–369.

[31] P. Moorthy and N. Kapre, "Zedwulf: Power-performance tradeoffs of a 32-node Zynq SoC cluster," in *Proc. IEEE 23rd Annu. Int. Symp. Field-Program. Custom Comput. Mach. (FCCM)*, May 2015, pp. 68–75.

Sebastian Buschjäger, photograph and biography not available at the time of publication.

Katharina Morik, photograph and biography not available at the time of publication.