Efficient traversal of decision tree ensembles with FPGAs

Romina Molina [a,d,e], Fernando Loor [a,b], Veronica Gil-Costa [a,b,*], Franco Maria Nardini [c], Raffaele Perego [c], Salvatore Trani [c]

[a] *Universidad Nacional de San Luis, Argentina*
[b] *National Commission of Sc. and Tech., Argentina*
[c] *ISTI-CNR, Pisa, Italy*
[d] *Università degli Studi di Trieste, Trieste, Italy*
[e] *The Abdus Salam International Centre for Theoretical Physics (ICTP), Trieste, Italy*

A B S T R A C T

System-on-Chip (SoC) based Field Programmable Gate Arrays (FPGAs) provide a hardware acceleration technology that can be rapidly deployed and tuned, thus providing a flexible solution adaptable to specific design requirements and to changing demands. In this paper, we present three SoC architecture designs for speeding-up inference tasks based on machine learned ensembles of decision trees. We focus on QuickScorer, the state-of-the-art algorithm for the efficient traversal of tree ensembles and present the issues and the advantages related to its deployment on two SoC devices with different capacities. The results of the experiments conducted using publicly available datasets show that the solution proposed is very efficient and scalable. More importantly, it provides almost constant inference times, independently of the number of trees in the model and the number of instances to score. This allows the SoC solution deployed to be fine tuned on the basis of the accuracy and latency constraints of the application scenario considered.

Keywords:
System on Chip
FPGA
Machine learning
Decision trees

1. Introduction

System on Chip (SoC) based Field Programmable Gate Arrays (FPGAs) has shown to be an efficient solution for improving the performance of applications due to their inherent parallelism. FPGAs are energy-efficient and provide high computing power due to the possibility of adapting the FPGA-based designs to a particular architecture. The SoC devices integrate a micro-controller, processors, DSPs, memory modules, oscillators, counters, timers, external interfaces, AD/DA, among other components. The SoC architecture can improve the performance in applications requiring both high-performance computations, and a sequential, processor-intensive functionality. Because of the complexity of chips, this technology can be programmed not only with VHDL [12] or Verilog [35], but also with higher level hardware description languages (HDL) such as SystemVerilog, SystemC, C/C++. Despite this possibility, the design process of SoC implementations is demanding and includes requirements specification, software/hardware partitioning (SW/HW co-design), hardware development and testing, software development and testing, system integration and testing.

During the last years, SoC computing power has been improved through a technology that allows the incorporation and seamless integration of heterogeneous resources. SoC-based FPGAs have been used in many research and development areas such as a control [4,24,36], power electronics [1,18,42,48], signal processing [37,44,45], image processing [9,10,14,28], virtualization [47] among others. Recently, SoC-based FPGAs have been used also for boosting the performance of Machine Learning (ML) applications. In many contexts, the widespread adoption of complex machine-learned models asks for novel efficient algorithmic solutions aimed at making fast and scalable both the off-line training of these models and their on-line use. We focus our attention on additive *ensembles of decision trees* and we investigate their efficient deployment on SoC-based FPGA architectures. These ML models, generated by boosting meta-algorithms that iteratively learn decision trees by incrementally optimizing a given loss function, have been shown to be the most general and competitive solutions for several "difficult" inference tasks such as ranking documents, items or posts in Web search engines, e-Commerce platforms, or online social networks, respectively. In these applications incoming rate of requests and quality-of-service expectations are very high thus the inference needs to be fast and must complete within small time budgets. All these requirements are very challenging to fulfill.

[*] Corresponding author at: Universidad Nacional de San Luis, Argentina.
E-mail address: gvcosta@email.unsl.edu.ar (V. Gil-Costa).

In this paper, we focus on exploiting SoC characteristics to efficiently deploy QuickScorer (QS), the state-of-the-art algorithm for the traversal of large tree ensembles [7,22] constituting the most efficient solution for the deployment of complex ML models [11,39–41]. Web-scale search services, designed to support a peak request stream of many thousands of queries per second, are deployed on cluster infrastructures including thousands of servers. Each server holds a portion of the data and the same partition is replicated on several servers to improve data availability, throughput and to support fault-tolerance. The results of each query are computed in parallel on all the data partitions and are then merged and ranked for high precision by means of the ML ranking model. Cost-effectiveness, fault tolerance and energy consumption considerations make larger clusters of commodity or mid-end servers to be preferred to comparable infrastructures built out of a smaller number of high-end servers [3]. Moreover, the CPU utilization of these servers is usually kept below 40% to support sudden peaks of queries and even an energy efficient server consumes about half its full power when doing almost no work [2]. The cost and power/performance competitiveness of SoC makes this technology to be very attractive for this particular application, where the high cost and power consumption of GPUs make their adoption prohibitive [38].

To demonstrate the capabilities of SoC technologies in addressing the challenges listed above, we propose and explore three architecture designs of the QS with different port configurations, replication degrees, communication settings on two embedded SoC devices, the PYNQ-Z1 and the Zynq UltraScale+ MPSoC. The investigation of QS implementations on embedded SoC devices is challenging due to the limited processing and storage resources available in these devices and the large space for alternative design choices offered. Interestingly, we show that with our solution the execution time for the inference task is almost constant until we reach the saturation of the hardware resources available in the device, independently of the number of instances scored and the number of trees in the ML model. This characteristic permits to choose the best suited FPGA device on the basis of the latency or accuracy requirements of the specific deployment thus optimizing the cost performance ratio of the solution.

The remaining of this paper is organized as follows. In Section 2 we review the related work, while in Section 3 we describe the QS algorithm. In Section 4 we present the architecture design to accelerate the QS algorithm on SoC-based FPGAs. In Section 5, we present the experimental setups and the experimental results, while Section 6 concludes the work.

2. Related work

Some previous work in the technical literature show that FPGAs can be successfully used to accelerate different machine learning algorithms. Lin et al. [19] evaluate the trade-off between machine learning context switch time and design performance (area utilization) on FPGAs. The authors present three different hardware designs applied to random forest classifiers.

The work presented by Narayanan et al. [31] implements on a FPGA device a decision tree classification algorithm. The authors report a speed-up of 5.58*x* achieved by reordering the computations and exploiting a bitmapped data structure. Miteran et al. [27] present an automatic hardware implementation of decision rules. The authors validated their proposal on real cases showing that it is possible to find a good trade-off between the hardware implementation cost and the classification error.

Nagarajan et al. [29] present an approach to perform multi-dimensional probability density functions estimation using Gaussian kernels on FPGAs. The results show a speed-up of 20*x*. Tracy et al. [13] deploy the Random Forest machine learning algorithm on an automata processor, which is a re-configurable co-processor accelerator. The implementation is based on a pipelined architecture that exhibits execution time linear with the number of features.

Neshatpour et al. [32] propose a heterogeneous architecture that integrates general-purpose CPUs with a dedicated FPGA to evaluate data mining and machine learning algorithms. The authors report a speed-up of 2.72*x*. Van Essen et al. [43] propose an analysis of FPGAs, GPUs and multi-core CPUs for accelerating compact random forest classifiers. The authors conclude that FPGAs provide the solution with the highest performance but require a multi-chip/multi-board system to execute even forests of modest size.

The work in [20] presents a new and lightweight decision tree learning system based on FPGAs showing a speedup up to 1581*x*. Nakahara et al. [30] compare the performance of random forest models on FPGA, CPU and a GPU implementations. The FPGA-based solution achieves a speed-up of 10.7*x* compared to the GPU-based one, and a speed-up of 14.0*x* compared to the CPU-based implementation, while also reducing the power consumption with respect to those approaches.

Owaida et al. [34] present a CPU-FPGA platform for tree ensemble classifiers. The platform includes a software driver to manage the FPGAs memory resources. The authors showed that FPGAs features provide an advantage over CPU based solutions for applications with frequent random memory accesses. Later, Owaida et al. [33] analyzed three mapping strategies to implement large decision tree ensembles over a cluster of FPGAs with floating-point precision. The results achieved show a linear performance improvement with the number of FPGA nodes being used.

In this paper, we focus on the exploitation of SoC parallelism on FPGA devices for QuickScorer (QS), the state-of-the-art algorithm for performing fast inference with tree ensembles [7,22]. Previous contributions showed the performance advantages resulting from the exploitation of different levels of parallelism in QS. As depicted in Fig. 1, QS exploits a particular representation of the tree ensemble based entirely on linear arrays accessed with high locality. This characteristic permits a very fast traversal of the tree ensemble at inference time by dealing with features and peculiarities of modern CPUs and memory hierarchies. Given the arrays representing the tree ensemble and a set of instances of feature vectors to score, both inter-instance and intra-instance parallelization strategies have been effectively exploited to parallelize QS on multicore/manycore platforms. Inter-instance parallelism is the most immediate, and takes advantage from the fact that several feature vectors can be scored independently and thus in parallel. This strategy is the most effective in a multi-core scenario, where multiple threads, also exploiting SIMD co-processors, run in parallel to score multiple input instances [17,23]. The intra-instance strategy partitions the scoring of a single feature vector into parallel subtasks, and takes advantage from the fact that, as discussed in the next Section, QS allows different features in the feature vectors to be processed in parallel accessing the read-only representation of the ensemble. Indeed, in order to exploit massive and fine-grained parallelism of manycore GPU platforms both parallelism strategies can be combined as in [17]. Moreover, to better exploit the upper fast levels of GPU memory hierarchy, the ensemble of trees can be partitioned in blocks, while we orchestrate the access to lower levels of the memory to force memory coalescing. The resulting solution is able to achieve a speed-up of up to 102.6x over the sequential version of QS on public learning-to-rank datasets when employing a NVIDIA GTX 1080 GPU [17].

In this work, we investigate if the characteristics of QS that made possible the efficient exploitation of multicore/manycore

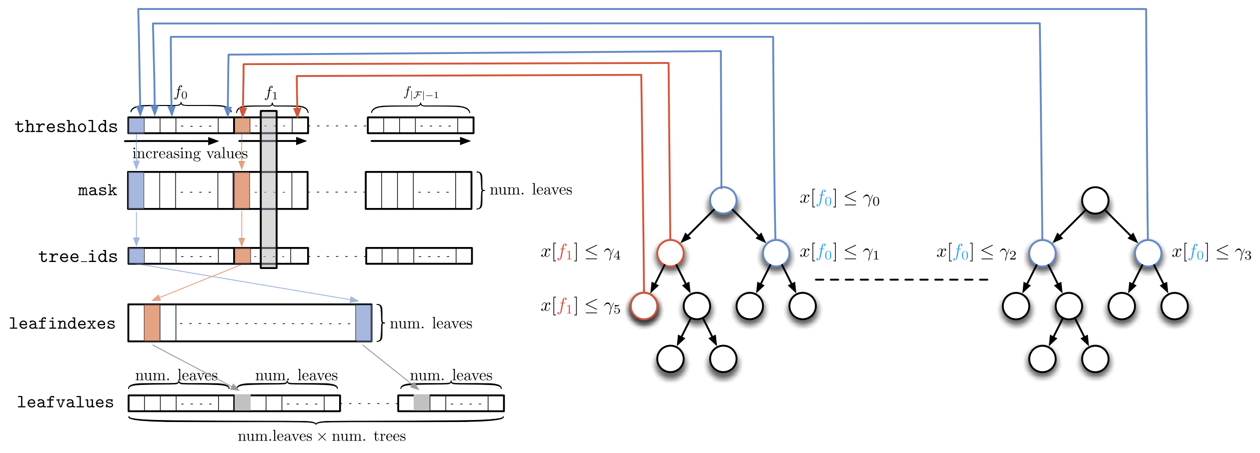


Fig. 1. Data layout of the QS algorithm.

platforms, are relevant also for SoC implementations. We thus advance the state of the art for efficient traversal of ML tree ensembles by proposing different architectures for SoC-based FPGA versions of QS. We discuss the efficiency of these different designs and report on experiments conducted on public datasets showing that our SoC-based FPGA implementations of QuickScorer are both efficient and scalable.

3. Background: the QuickScorer algorithm

A ML tree ensemble encompasses several binary decision trees as illustrated in the rightmost part of Fig. 1. The internal nodes of each tree of the ensemble are associated with a Boolean test over the value of a specific feature characterizing the input instance to be scored/predicted. Each leaf node stores instead a value representing the contribution of the specific tree to the final prediction.

Let us denote the ensemble with $\mathcal{T} = \{T_0, T_1, \dots T_m\}$, and let Λ be the maximum number of leaves of each tree. Moreover, let $\mathbf{x}$ be the vector of feature values representing an input instance. Let $\mathcal{F}$ be the feature set, and let $|\mathcal{F}|$ be the number of dimensions of feature vector $\mathbf{x}$. We use f to refer to the $f-$th feature, with $\mathbf{x[f]}$ storing the value of the feature. Moreover, let $s(\mathbf{x})$ be the numerical score eventually computed for input vector $\mathbf{x}$. Determining $s(\mathbf{x})$ requires the traversal of all the trees in the ensemble to devise all the tree contributions and to compute their *sum*. The goal of QS is making fast the traversal of $\mathcal{T}$ to compute $s(\mathbf{x_i})$ for a large batch of input instances x_i, $i = 0, \dots, n$.

The traversal of a decision tree t performed by QS can be viewed as the process of converting a bitvector `leafindexes[t]` of Λ bits, where all bits are initially set to 1 (Mask Initialization phase), to a final bitvector where the *leftmost 1* identifies the *exit leaf* of the tree [22]. The bitvector is manipulated through a series of bit masking operations that use a set of pre-computed bitvectors `mask`, still of Λ bits, each associated with an internal branching node of t (Mask Computation phase). To pre-compute these `mask`s, we consider that the right branch is taken if the branching internal node is recognized as a *false node*, i.e., if its binary test fails. Whenever a false node is identified, we annotate the set of unreachable leaves in `leafindexes[t]` through a *logical AND* ($\wedge$) with the corresponding `mask` bitvector. Therefore, the purpose of `mask` is to set to 0 all the bits of `leafindexes[t]` corresponding to the unreachable leaves of t, i.e., all the leaves that belongs to the *left subtree* not selected by the failed test of the branching node. The reader is invited to refer to [22] for a detailed explanation of the QS algorithm.

Algorithm 1: QuickScorer.

Input :
- $\mathbf{x}$: input feature vector
- $\mathcal{T}$: ensemble of binary decision trees, with
  - `thresholds`: sorted sublists of thresholds, one sublist per feature
  - `tree_ids`: tree's ids, one per internal split node
  - `mask`: node bitvectors, one per internal split node
  - `offsets`: offsets of the blocks of triples
  - `leafindexes`: result bitvectors of size Λ, one per each tree
  - `leafvalues`: output values, one per each tree leaf

Output:
- Final score of $\mathbf{x}$

```
1  QuickScorer(x, T):
2    foreach $t \in 0,1,\dots,|\mathcal{T}|-1$ do                 // Mask Initialization
3      leafindexes[t] ← 11...11
4    foreach $f \in 0,1,\dots,|\mathcal{F}|-1$ do                 // Mask Computation
5      $i \leftarrow$ offsets[f]
6      $end \leftarrow$ offsets[f+1]
7      while $\mathbf{x}[f] >$ thresholds[i] do
8        $t \leftarrow$ tree_ids[i]
9        leafindexes[t] ← leafindexes[t] ∧ mask[i]
10       $i \leftarrow i+1$
11       if $i \geq end$ then
12         break
13   $score \leftarrow 0$
14   foreach $t \in 0,1,\dots,|\mathcal{T}|-1$ do                 // Score Computation
15     $j \leftarrow$ index of leftmost bit set to 1 of leafindexes[t]
16     $l \leftarrow t \cdot \Lambda + j$
17     $score \leftarrow score +$ leafvalues[l]
18   return score
```

Algorithm 1 illustrates the QS algorithm for the fast traversal of the ensemble. The algorithm restructures the data layout of an ensemble of decision trees to leverage modern memory hierarchies and reduce the branch prediction errors to limit the control hazards. In addition, QS accesses data structures with high locality, since the tree forest traversals, repeated for each input instance, are transformed into a scan of linear arrays (see the code in Algorithm 1 and the leftmost part of Fig. 1). QS supports both general and oblivious [16] binary decision trees. The former are decision trees where the internal split nodes are independent of each other and as a consequence the trees can be unbalanced. The latter are a special kind of decision trees where all nodes at the same level test the same feature with the same threshold. As a consequence, oblivious trees are balanced.

To efficiently identify *all the false nodes* in the ensemble, QS processes the branching nodes of all the trees *feature by feature*, taking advantage of the commutative and associative property

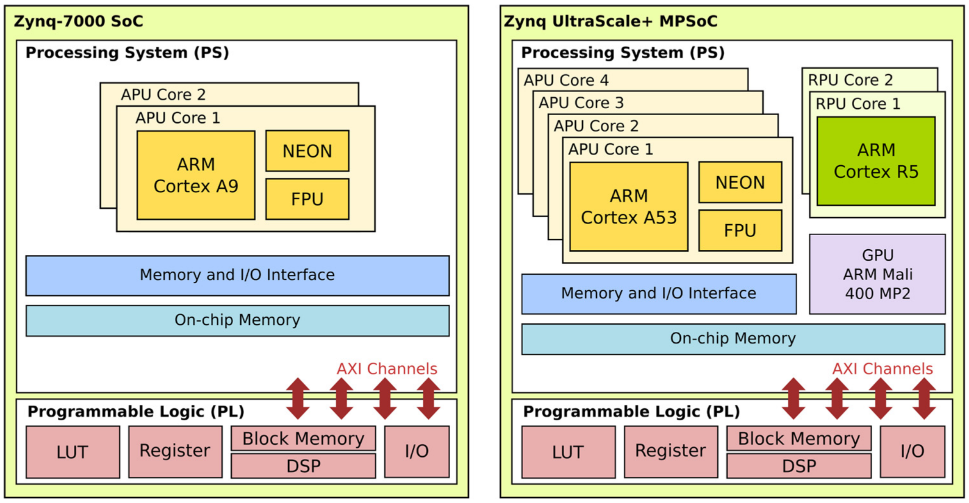


Fig. 2. Diagram of the experimental platforms: PYNQ-Z1 (left) and UltraScale (right).

of the *logical AND* operand that allows to perform the masking operations for traversing each tree of the ensemble in arbitrary order. Specifically, for each feature f, it builds a list $\mathcal{N}_f$ of triples $(\texttt{thresholds}, \texttt{mask}, \texttt{tree\_ids})$, where `thresholds` is the test threshold of a branching node of tree t performing a test over the feature f of the input instance $\mathbf{x}$, `tree_ids` is the id of the tree t that contains the branching node, where the id is used to identify the bitvector `leafindexes` to update and `mask` is the pre-computed mask that identifies the leaves of t that are unreachable when the associated test evaluates to false. The data structure layout is illustrated in Fig. 1. Hereinafter, we refer to the triples $(\texttt{thresholds}, \texttt{mask}, \texttt{tree\_ids})$ and to the `leafvalues` as *the model data structure*. Note that the model data structure is pre-computed off-line and accessed in read-only mode, as opposed to the `leafindexes` which are instance dependent and updated at runtime. $\mathcal{N}_f$ is sorted in ascending order of `thresholds`. Hence, when processing $\mathcal{N}_f$ sequentially, as soon as a test evaluates to true, i.e., $\mathbf{x}[f] \leq \texttt{thresholds}$, the remaining occurrences of $\mathcal{N}_f$ evaluate to true as well, and thus their evaluation can be safely skipped thus reducing the number of operations performed with respect to competitor solutions [22].

4. Accelerator design exploration

4.1. Architecture

We deploy the QuickScorer algorithm on two SoC devices with different capacities to evaluate how the hardware limitations of the SoCs affect the final model. We use a SoC instead a single FPGA because the former allows to faster deploy the algorit-hms than using an FPGA connected to a desktop CPU. We use a mid-level device named Xilinx Zynq$^{\text{TM}}$ SOC-based platform also known as PYNQ-Z1. It is composed of a Dual-core ARM-based CPU plus reconfigurable logic. The other device has higher capacities. It is the Zynq UltraScale+ MPSoC with a quad-core ARM Cortex$^{\text{TM}}$-A53 applications processor, dual-core Cortex-R5 real-time processor and Mali-400 MP2 graphics processing unit.

Fig. 2 (left) shows an illustrative diagram of the PYNQ-Z1 experimental platform, while Fig. 2 (right) refers to the UltraScale device. The platforms consist of a SoC-style integrated Processing System (PS) and a Programmable Logic (PL) block on a single die. The PS communicates with the IP block of the PL through the AXI-4 Interface, which supports a subset of the AMBA AXI4 protocol designed for high-speed data streaming. The PS integrates ARM application processors (dual-core for the PYNQ-Z1 and quad-core for the UltraScale), AMBA interconnect, internal memories, external memory interfaces, and peripherals including USB, Ethernet, SPI, SD/SDIO, I2C, CAN, UART, and GPIO. The PS runs independently of the PL and boots at power-up or reset. The PL has different components like the Look Up Table (LUT), the Flip Flops (FF), the digital signal processor (DSP) and the block memory (BRAM) which are used to implement the Intellectual Property (IP) blocks. The UltraScale SoC has resources with larger capacity than the Pynq SoC (504K vs 13K programmable logic cells, 1728 vs 220 DSP slices, 11 MB vs 630 KB of Block RAM). This comparison gives an idea of the hardware capabilities across the family of devices provided by Xilinx.

After selecting the SoC devices we have to map the architecture design to the chip. In general, the vectors are sent to the PL. Then, an IP block executes the QuickScorer and returns the results to the PS. There are two possibilities to manage the SoC. (1) The first one is to keep an operating system on the PS, to control the functions of the device. (2) Another possibility is to use C/C++ code avoiding the overhead of an operating system, known as a “bare-metal” implementation. Additionally to these possibilities, we need to set how to communicate the PS, the PL and the DDR memory. Due to the large number of feature vectors to be processed, we store those vectors into the DDR memory of the devices. The most practical way to access DDR memory for this particular case is by instantiating a DMA controller within the programmable logic, which will be controlled by the PS. In turn, the DMA block can communicate with the PS through various ports, among which are the GP, HP, and ACP ports.

In a regular DMA operation, the master block initiates the transaction of data, while the slave block responds to the transaction already started. The interconnection between the different blocks of the system is performed by the AXI-4 stream buses. To make this communication possible, the high-performance AXI slave ports of the Zynq are enabled. The AXI-Lite Interface allows the processor to communicate with the AXI DMA block to configure, initialize and monitor the data transfer. In other words, by using the AXI DMA block, data is transferred from one part of the system to another. Different combinations of the DMA, PS and IP blocks settings are described in the next section.

We used Vivado Design Suite 2019.1 to implement the QS algorithm in the IP block. This tool allows the C++ version of QS be directly converted into Register Transfer Level (RTL) code for hardware implementation. Such high level synthesis tools permit to remarkably reduce the time of design and, at the same time, to improve the design space exploration.

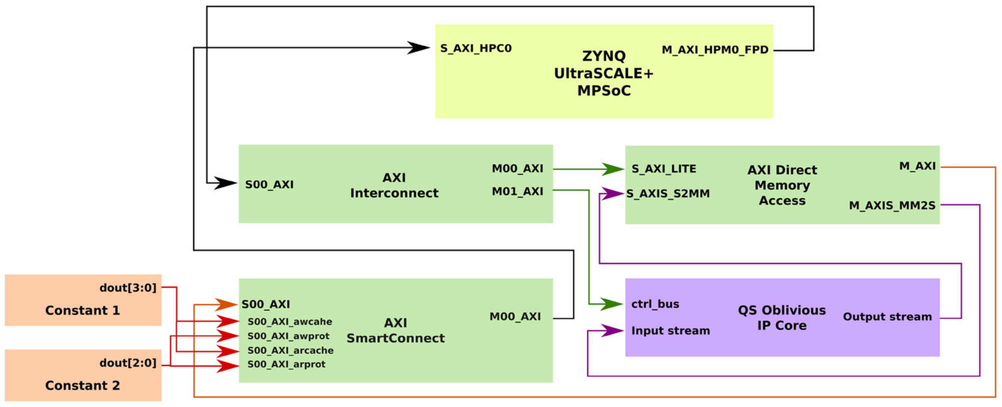


Fig. 3. Architecture design for the UltraScale device with a single DMA.

4.2. Design overview

In this section we discuss the three different architectures we designed to implement the QS algorithm on the PYNQ-Z1 and the UltraScale devices. The first one includes an instance of the QuickScorer IP block communicated via a DMA block with the PS. This architecture design allows to analyze the impact of the DMA block that controls the communication between the QS IP block and the PS. The second architecture design is intended to increase the performance of the algorithm by replicating the QS IP blocks. Finally, the third architecture design has a Linux image with support for Python on the PS and requires 2 DMAs to control a single IP block, since the Python functions for reading and writing by DMA require individual DMA blocks of each operation. This last architecture design aims to show how performance can be drastically affected when using high level development tools.

We recall from Section 3 that the ultimate goal of QS is making fast the traversal of a given tree ensemble $\mathcal{T}$ to compute the scores $s(\mathbf{x_i})$ for a large batch of feature vectors x_i, $i = 0, \ldots, n$.

Fig. 3 shows the first architecture design for the Zynq UltraScale+ MPSoC board composed of a single DMA and a single IP responsible for accelerating the QS algorithm. The DMA is responsible to manage the communication between the IP block and the PS. The PS executes configuration tasks and enables the IP block. In other words, the PS is responsible of the resource management. The architecture also includes the Zynq processing system, reset system, and the interconnection blocks.

The AXI DMA IP block is responsible for the transfer data between the FPGA and the DDR memory. To perform this operation, AXI DMA has two channels: MM2S (memory-mapped to stream) and S2MM (stream to memory-mapped). In most applications, the High Performance (HP) ports are preferable to the Accelerator Coherency Port (ACP) ports to perform the communication due the higher bandwidth, and to avoid the disturbance of contents of L2 cache memories [25,26]. Therefore, the architecture design presented in Fig. 3, enables the AXI High Performance Coherent (HP) port on the Zynq to perform a coherent transfer of the feature vectors $\mathbf{x_i}$ from the FPGA device and the host memory. With hardware-managed I/O coherency it is possible to simplify the software, improve the system performance, and reduce the power by sharing on-chip data from APU caches. To this end, two constants are used to enable the coherence transaction, the AxCACHE and AxPROT which must be set with the right values to enable cache snooping.

Fig. 4 shows the same architecture design for the PYNQ-Z1 board. The scheme is similar to the configuration with a single DMA for the UltraScale showed in Fig. 3, but in this case it is not necessary to enable additional signals for HP ports. Notice that the architecture design presented in Fig. 4 is also valid for ACP ports. The only difference is at the port enablement level.

The second architecture design is presented in Fig. 5 for the PYNQ-Z1. It includes an additional IP to implement a second instance of the QS algorithm in order to perform the computation of scores $s(\mathbf{x_i})$ on disjoint subsets of feature vectors. A second DMA block is also included to perform the communication to/from the PS. This approach aims to evaluate the performance achieved by the QuickScorer when replicating its corresponding IP block. The ports HP0 and HP2 are enabled to transfer data, and for each IP block a DMA block is instantiated. We select the ports HP0 and HP2 because they share different buses to communicate.

In the two previous architecture designs (Fig. 3, 4 and 5) the PS loads libraries to control different components and interfaces of the board, such as the DMA used for communications between the PS and the PL. That scheme is called bare-metal implementation, since there is no operating system running in the PS. In Fig. 6, we present a third architecture which provides a bootable Linux image allocated in the PS, with a running version of Python and other open-source libraries, which make possible to perform the control of the functions in the FPGA boards. This third architecture, deployed on both the PYNQ-Z1 and the UltraScale boards, includes two blocks of DMA supporting data exchanges. One DMA is used to communicate data from the PS to the PL and a second DMA to communicate in the opposite direction.

This scheme is intended for developers willing to work at a high level of abstraction, hiding the low-level configuration details that have to be taken into account for bare-metal code development. The Python layer helps in fact developers to expedite the implementation of SoC solutions on FPGA boards and to easily customize the hardware platform and the interfaces. Unfortunately, this high level of abstraction does not allow the developer to fine tune the implementation and introduces large overheads making the resulting deployment absolutely not competitive in term of execution time with the previously discussed bare-metal solutions. Anyway, we discuss also this architecture as a further possibility to follow in the case the performance requirements are not strict.

4.3. Implementation details

High Level Syntheses (HLS) tools allow to create hardware from a high-level of abstraction, using directives to specify concurrency and pipelining opportunities. In this work we analyze the SoC based implementation of QuickScorer without re-coding techniques, in order to estimate resource consumption and execution times. To this end, several directives are inserted in the C++ code such as the PIPELINE, INLINE, UNROLL and INTERFACE.

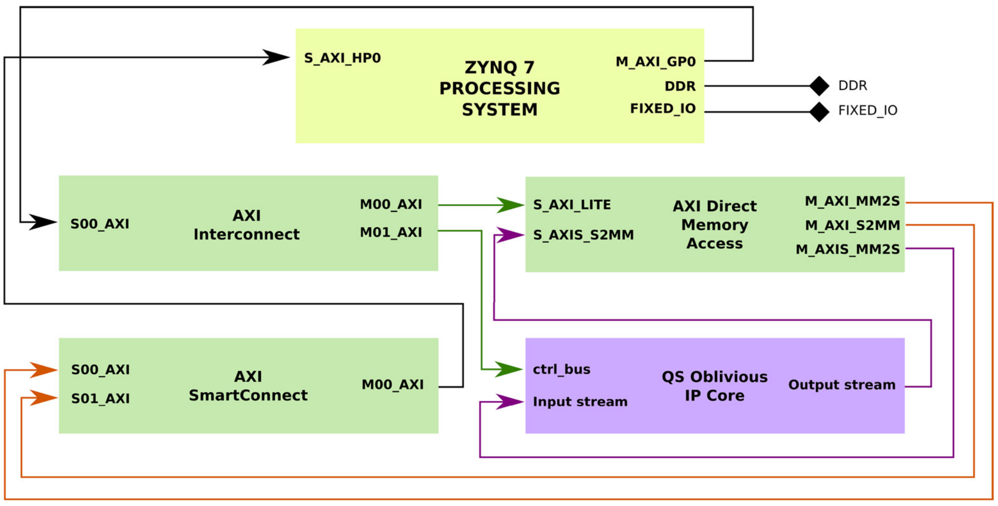


Fig. 4. Architecture design for the PYNQ-Z1 device with a single DMA.

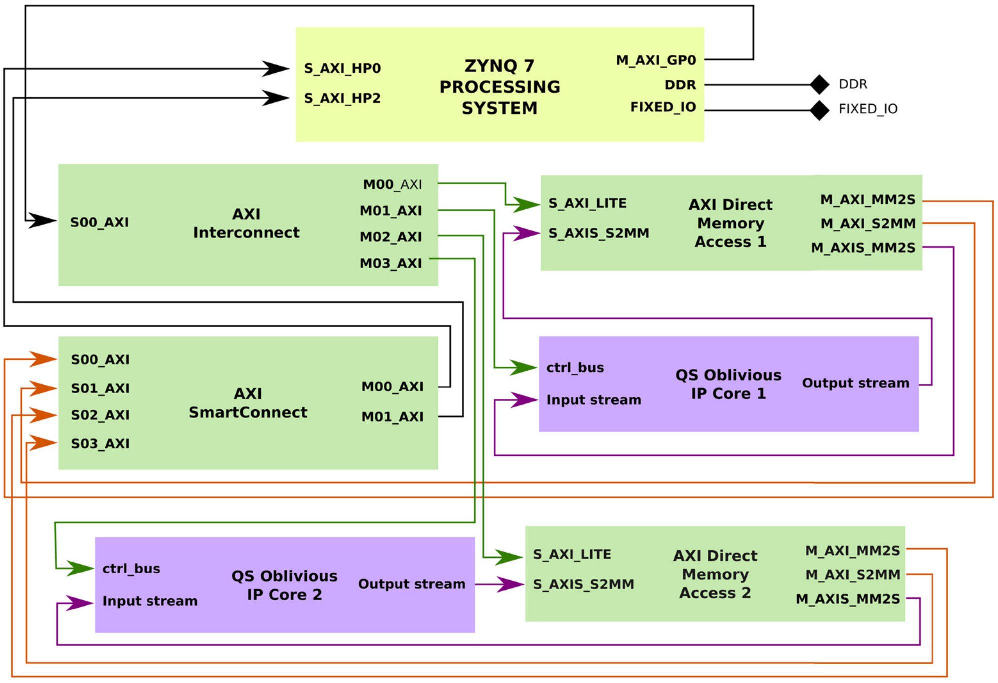


Fig. 5. Architecture design for the PYNQ-Z1 with two IP blocks.

Algorithm 2: PIPELINE Directive.

```
1 scorerVanilla_init:
2   for (int g=0; $g < FACTOR$; $g++$) do
3     #pragma HLS PIPELINE
4     scorerVanilla(&inputVector[g× TOTAL_FEATURES],
        &outputScorer[FACTOR]);
```

We use the PIPELINE directive to optimize the insertion (push) and extraction (pop) of data from the stream. We extract the feature vectors from the input stream removing the associated control logic. After the QuickScorer is executed, the final scores computed for the feature vectors are packaged into an output stream adding the corresponding control signals. The PIPELINE directive is also used to speed-up the execution of the function admitting new inputs vectors. Algorithm 2 shows the use of the PIPELINE directive.

The FACTOR variable defines the number of vectors. TOTAL_FEATURES represents the size of the vector, which is the total number of features. Once the inputs vectors are stored into the on-chip memory, the scorerVanilla() function is executed to compute in a pipeline the final score for each input vector. When we call a function, a certain amount of clock cycle overhead is associated with the call. The INLINE directive can minimize the overhead associated with performing a function call. In this work, it is applied to the scoreVanilla() function that calls the scorer functionality.

When processing more than one input vector, the output values are stored into an array. In this case, the UNROLL directive is used to store the data into the final stream. This directive allows to improve the latency.

The INTERFACE directive is used to manage the inputs and outputs of the IP block through a port with a specific I/O protocol. In this work, we used axis for the input (query vector) and the output (scorer value), which implies that all ports are defined as an AXI4-Stream interface. For control signals, we selected the s_axilite using the AXI4-Lite interface. Algorithm 3 shows how we implemented this directive.

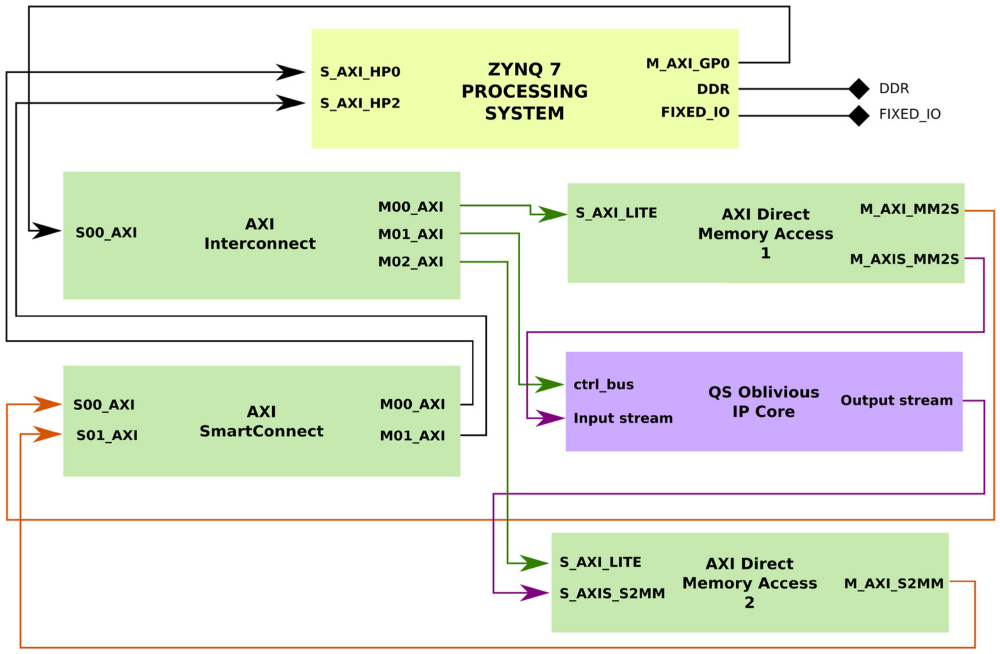


Fig. 6. Architecture design for the PYNQ-Z1 with 2 DMAs and Python.

Algorithm 3: DIRECTIVES.

```
1 #pragma HLS INTERFACE s_axilite register port=return bundle=ctrl_bus
2 #pragma HLS INTERFACE axis off port=outputScorer
3 #pragma HLS INTERFACE axis off port=vectorInput
```

5. Experimental assessment

5.1. Experimental settings

We conduct experiments by using machine-learned ranking models based on ensembles of regression trees. These models are trained on a publicly available learning-to-rank dataset, namely MSLR-WEB10K[1] [21]. The dataset consists of 10,000 queries and 1,200,192 query-document pairs represented as vectors of 136 real-valued features. The query-document pairs are labeled with a relevance judgment ranging from 0 (irrelevant) to 4 (perfectly relevant), assessing the degree at which a given document is relevant for the specific query. The dataset is split in training, validation and testing set according to a 60%-20%-20% scheme. Moreover, it is split into 5-fold, with the *cross-validation* technique (i.e., instances are rotated among the train/vali/test splits). In this work, since the objective is to evaluate the efficient traversal of tree ensembles and not the effectiveness and robustness of the trained model, we use only the first fold, namely MSLR-WEB10K-F1. Indeed, in terms of scoring time, the average inference time of a document in one fold is exactly the same of a document belonging to a different fold, and it is only related to the characteristics of the model (i.e., the number of trees and the shape of each tree).

We use training data from MSLR-WEB10K-F1 to train λ-MART [46] and Oblivious-λ-MART [16] models by optimizing NDCG@10 (a well-known IR metric commonly used to assess the quality of a list of ranked items [15]). Both models generate additive ensembles of regression trees aiming at finding a scoring function that produce an ordering of documents as close as possible to the ideal ranking. The difference is that the former adopts an independent splitting criterion, i.e., each split node is chosen independently from the others, while the latter train balanced trees, where, at each level, all the branching nodes test the same feature-threshold pair. However, it is important to highlight that the results of the paper can be also applied to analogous tree-based models generated by different state-of-the-art learning algorithms, e.g., GBRT [8]. The ranking models trained are the following:

- `100T_10L_NObl`: an ensemble of 100 non-oblivious trees with 10 leaves.
- `100T_8L_Obl`: an ensemble of 100 oblivious trees with 8 leaves.
- `1000T_8L_Obl`: an ensemble of 1,000 oblivious trees with 8 leaves.

To train these models we used QuickRank, an open-source C++11 framework implementing several state-of-the-art learning-to-rank algorithms [5]. The models are trained on the training set of MSLR-WEB10K-F1, with the validation set used for early stopping (i.e., a technique used for avoiding overfitting). We evaluate the performance achieved at inference time by the QS algorithm on the test set, with the three architecture designs presented in Section 4.2 on the PYNQ-Z1 and the UltraScale running at 100 MHz and 150 MHz, with HP and ACP ports and with different number of input feature vectors. The performance measure used for all the tests is the latency in microseconds from the time when the feature vectors are sent from the PS to the PL, until all the score results are received back in the PS.

In the next section, we report on the execution times measured for a single execution of each test since there are no other applications running in the PS. The PL only hosts the logic related to the synthesized QS hardware, which minimizes the possibility of variance in the execution times. To validate this claim, we show in Table 1 the mean and standard deviation (σ) measured for 10 executions with models `100T_10L_NObl` and `100T_8L_Obl` for a number of features vectors ranging from 1 to 128 on the PYNQ- Z1 with bare-metal development when the board is set at 100 MHz. In all cases, we show that the value of σ is very small.

[1] http://research.microsoft.com/en-us/projects/mslr/.

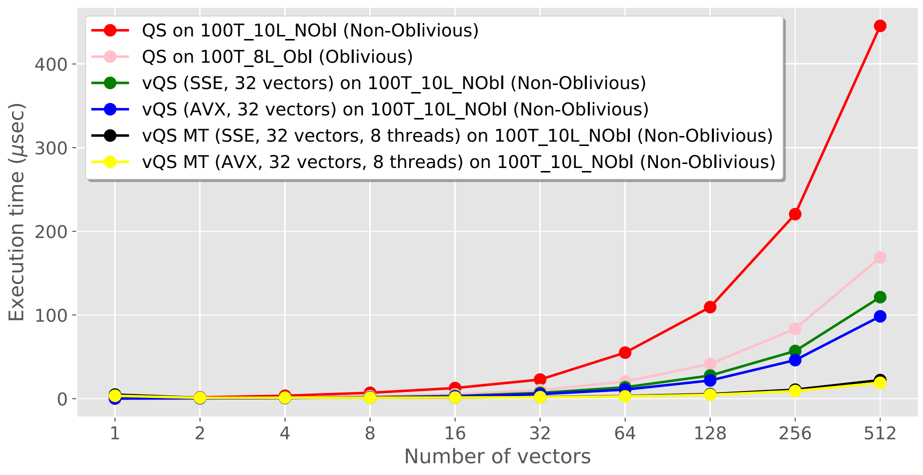


Fig. 7. Total execution time (μs) as a function of the number of feature vectors scored using different CPU implementations of QuickScorer on the `100T_10L_NObl` and `100T_8L_Obl` models.

Table 1
Statistical analysis: Average execution time (in μs) and standard deviation (σ).

	100T_10L_NObl		100T_8L_Obl	
	Mean	σ	Mean	σ
1	3.30	0.02	3.25	0.08
8	3.32	0.02	3.25	0.03
16	3.33	0.04	3.25	0.04
32	3.35	0.01	3.26	0.03
64	3.37	0.20	3.26	0.03
128	3.38	0.03	3.26	0.04

5.2. Results

In this section, we first present the results obtained by running the QuickScorer algorithm on an Intel Xeon CPU E5-2630 v3 (2.40 GHz) with 16 hyper-threaded cores and 192 GB of RAM. We then present the results obtained when employing the PYNQ-Z1 and UltraScale FPGA devices.

Fig. 7 reports the execution time (in μs) of different versions of QS required to process an increasing number of input feature vectors with models `100T_10L_NObl` and `100T_8L_Obl`. In details, we experiment the single-thread CPU version (QS) [7,22], a vectorized version that employs instruction-level parallelism using SSE and AVX instruction sets (vQS-SSE and vQS-AVX) to score 32 feature vectors in parallel [23], and a multi-threaded version that perform thread-level parallelism on top of the instruction level ones (vQS MT) [17]. In our experiments, vQS MT employs 8 threads each one running vQS-SSE and vQS-AVX on blocks of 32 feature vectors. For one input vector, QS on `100T_10L_NObl` (non-oblivious model) reports an execution time of 0.7 μs, while processing 512 input vectors requires 445.5 μs. QS on `100T_8L_Obl` (oblivious model) reports instead faster executions times starting from 0.3 μs for one input vector and 168.9 μs for 512 input vectors. As expected, we measured an execution time which increases linearly with the number of feature vectors scored although perturbations to this linearity are observed when the input vectors to score are a few. Results also show that instruction-level parallelism and thread-level parallelism help in reducing the total execution time. When instruction-level parallelism is employed, vQS-SSE and vQS-AVX score 32 feature vectors in parallel on 128 and 256 registers, respectively. For one feature vector, vQS-SSE requires 0.26 μs while vQS-AVX requires 0.25 μs. When increasing the number of vectors the difference between the two versions increases revealing a better performance for vQS-AVX. For 512 input vectors, vQS-SSE requires 121.3 μs while vQS-AVX requires 98.4 μs showing a reduction in the execution time of vQS-AVX of about 19% with respect to vQS-SSE. The use of thread-level parallelism on top of the instruction-level one further reduces the execution time. For one feature vector, vQS MT with 8 threads requires 4.9 μs and 3.4 μs when using SSE and AVX instruction sets, respectively. This result shows that, for a small number of vectors, the overhead introduced by the multi-threading framework (OpenMP [6]) significantly hurts the performance of the method. However, when increasing the number of vectors, e.g., 512, vQS MT requires 22.2 μs and 19 μs when using SSE and AVX instruction sets, respectively, showing a total speedup on vQS-SSE and vQS-AVX of up to 5.4x, and on QS of up to 23.4x.

Fig. 8 shows instead the results obtained with the SoC implementations on the PYNQ-Z1 and the UltraScale FPGA devices. As in the previous figure the *x*-axis in each plot shows the number of input feature vectors scored while we report in the *y*-axis the execution time in microseconds for the different implementations tested. Specifically, the curves in each plot refer to the execution times achieved with bare-metal development when the board is set at 100 MHz and 150 MHz using the ACP and the HP ports and architectures with a single IP (as illustrated in Fig. 4) or two IP blocks (see Fig. 5). The four plots refer to results obtained: with the non-oblivious model `100T_10L_NObl` on the PYNQ-Z1 (Fig. 8 (a)) and on the UltraScale device (Fig. 8 (c)); with the oblivious model `100T_8L_Obl` on the PYNQ-Z1 (Fig. 8 (b)) and the UltraScale device (Fig. 8 (d)).

With the PYNQ-Z1 device, the best results obtained with the oblivious model range from 2.72 μs for one input vector to 3.02 μs for 240 input vectors. The best results obtained with the non-oblivious model range instead from 2.41 μs for one input vector to 3.57 μs for 240 input vectors. With the more powerful UltraScale device, the best results obtained with the oblivious model range from 0.35 μs for one input vector to 0.4 μs for 1024 input vectors. The best results obtained with the non-oblivious model range from 0.36 μs for one input vector to 0.42 μs for 1024 input vectors.

The curves plotted in Fig. 8 shows that, as expected, with a higher clock rate the execution time is reduced. For the non-oblivious model (`100T_10L_NObl`), execution times obtained with 150 MHz are in average 13% lower than the execution time reported with 100 MHz. For the oblivious model (`100T_8L_Obl`), the improvement achieved with 150 MHz is 17% in average. Regarding the execution times reported with HP and ACP ports, we can observe that the ACP port configuration allows us to obtain slightly lower execution times than the HP port configuration for the oblivious model and the PYNQ-Z1 device. However, the other

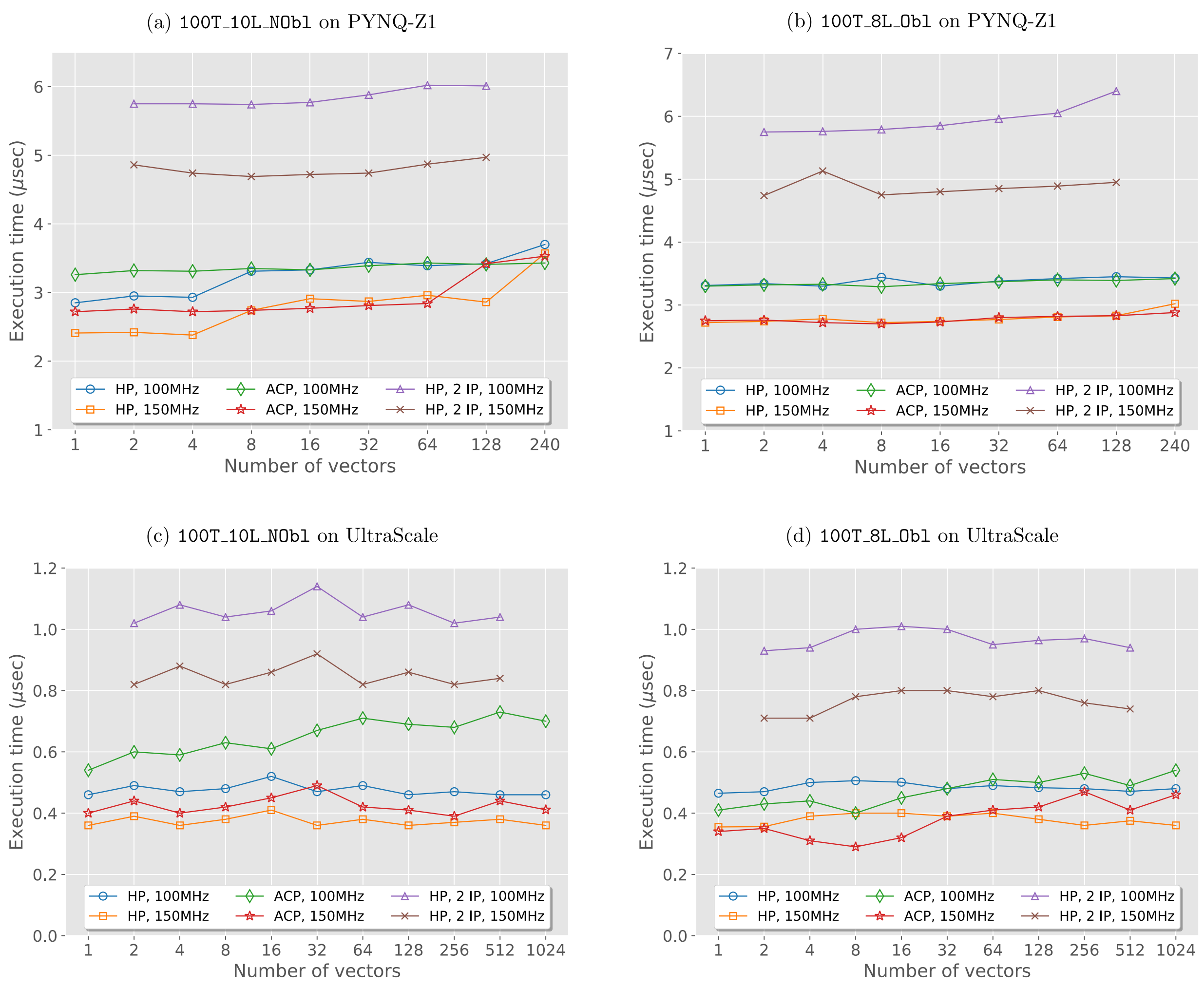


Fig. 8. Execution times for the different FPGA configurations and the `100T_10L_NObl` and `100T_8L_Obl` tree ensembles. All cases running at 100 MHz and 150 MHz with HP (one and two IP blocks) and ACP ports.

experiments show that both port configurations present very similar performance.

Additionally, by looking at the curves labelled `HP, 2 IP` plotted in Fig. 8 we can see that the architecture designed with two IP blocks almost doubles the execution time with respect to the corresponding architecture with a single IP block running at the same clock rate. This is because the architecture designed with two IP blocks requires additional logic for replicating the ML model and the DMA modules performing the communication between the PS and the IP blocks. The overhead due to the management of these additional DMA modules drastically affects the performance of the system making the solution with two IP blocks not competitive with the one using a single IP block. Moreover, the resources used for deploying the second QS IP block limit also the number of input vectors fitting in the BRAM of the device. We see from the plots that the curves reporting the execution times for the two IP deployment are shorter than the ones for the single IP.[2] Specifically, they end in correspondence of 128 or 512 input vectors for the PYNQ-Z1 and Ultrascale devices, respectively. Larger sets of feature vectors do not fit in the board memory and cannot be processed in a single batch. Conversely, the architectures with a single QS IP block use less hardware resources because they do not require additional logic for the DMA modules and for replicating the ML model: the plots reported in Fig. 8 show that the UltraScale and PYNQ-Z1 boards can fit in this case up to 1024 or 240 input vectors before saturating the BRAM memory.

Anyway, the most important characteristic of the presented SoC-based FPGA architectures is evident when we compare the plot in Fig. 7 with any of the plots reported in Fig. 8. While on a traditional CPU the (sequential) execution time increases linearly with the number of vectors scored, the same does not happen for the FPGA deployment: we can see in fact from Fig. 8 that on SoC-based FPGA hardware the number of feature vectors scored does not impact significantly the execution time because the computation required to process the input instances is performed in parallel inside the QS IP block(s). In other words, in order to process more input instances we only need to increase -in the FPGA- the number of accumulators and the logical components executing Algorithm 1 in parallel on all the vectors by accessing read-only a single shared ML model.

Of course, the number of input vectors scored impacts instead the use of resources. In Table 2 we report the percentage of utilization of various resources of the PYNQ-Z1 device for processing the largest sets of input vectors tested. Specifically, from the table we see that the QS algorithm in a single IP block saturates the BRAM with 240 input vectors (99.64%), while we almost saturate the BRAM (82.50%) of the same device with only 128 input vectors when two QS IP blocks are used. Thus, scoring on this device with the most efficient one-IP solution more that 240 feature vectors

[2] Of course these curves start at $x = 2$, i.e., with a single vector assigned to each of the two IP blocks.

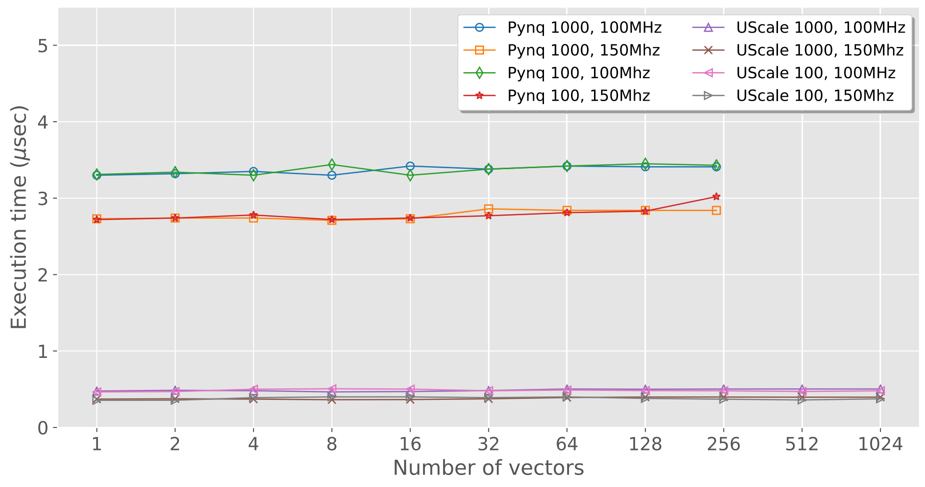


Fig. 9. Execution times obtained on the PYNQ-Z1 and UltraScale devices with the models `100T_8L_Obl` and `1000T_8L_Obl`, i.e., the ensembles with 100 and 1,000 oblivious trees.

Table 2
Resource utilization for the `100T_10L_NObl` model with one (240 feature vectors) and two IP blocks (64 feature vectors).

Resource	Utilization (%)	
	1 IP - 240 vectors	2 IP - 128 vectors
LUT	12.10	33.79
LUTRAM	13.82	17.48
FF	7.37	20.31
BRAM	**99.64**	**82.50**
DSP	1.36	2.73

would require to partition the instances in batches of at most 240 vectors and process these batches sequentially, one at the time.

Fig. 9 shows the execution times obtained on both the PYNQ-Z1 and the UltraScale devices by the architecture with one single IP block with the models `100T_8L_Obl` and `1000T_8L_Obl`, i.e., the ensembles with 100 and 1,000 oblivious trees. The plot shows that the two curves reporting the execution times for models with 100 and 1,000 trees almost overlap. Thus, besides the number of input vectors processed, also the number of decision trees in the ensemble does not impact significantly the execution time of the QS algorithm running on the FPGA device. For what memory and logic resources usage is concerned, the ML model is in fact much less demanding than the input vectors. The model with 100 trees occupies only 16 kB while the model with 1,000 trees requires about 164 kB. Conversely, to manage 1024 feature vectors we use $32\ bits \times 136\ features \times 1024 = 557$ kB plus the memory for the ML model which is accessed by all the logic components and accumulators used to compute the scores in parallel. Therefore, as far as the memory of the FPGA is not saturated, we can increase the model size or the number of input vectors without significantly affecting the execution times.

Finally, in Fig. 10 we report the execution time in microseconds obtained with the PYNQ-Z1 and the UltraScale devices when executing the Python-based QS version with the `100T_10L_NObl` and `100T_8L_Obl` models. Python introduces a huge overhead due to the additional IP blocks used to implement the Python-associated code into the FPGA (see Fig. 6). As discussed above, we report the results achieved with this architecture only to show that it can constitute a possible alternative when performance requirements are not strict and the time available for FPGA coding is very short. However, from the curves in Fig. 10, we see that the execution times for the Python version are about 4 order of magnitude higher that those obtained with the bare-metal implementations. Moreover, differently from the bare-metal cases the execution time with Python development increases as we increase the number of input vectors scored. It is thus apparent that at the cost of a more complex and time-consuming coding, the bare-metal development allows the execution time of the QS algorithm to be drastically reduced.

How SoC-based QS advances the state of the art. The previous experiments show a very important characteristic of SoC-based FPGA bare-metal implementations of QS: the execution times measured for this solution are almost constant and independent of the number of feature vectors processed and the number of trees of the tree ensemble. This of course holds only if we do not saturate the resources of the specific FPGA device used. Such characteristic, that does not hold for the CPU versions of QS, is very interesting for capacity and hardware sizing planning since it gives engineers the possibility of choosing the most efficient and cost-effective FPGA device to use on the basis of the requirements of accuracy (depending on the number of trees) and throughput (depending on the number of input instances processed in parallel) of the specific application at hand. This is of paramount importance for any large-scale deployment of ML ensemble models subject to real-time or near real-time constraints.

6. Conclusions and future work

In this paper we presented and evaluated three SoC-based FPGA architecture designs to accelerate inference with ML models based on ensembles of decision trees. In particular, we focused on the QuickScorer state-of-the-art algorithm for performing fast and accurate inference tasks by traversing large tree ensembles. The architecture designs were deployed on two embedded system-on-chip, the PYNQ-Z1 and the Zynq UltraScale+ MPSoC. The first architectural design used a single IP block to deploy the QS algorithm and a single DMA to communicate between the PS and the PL. The second architecture uses two IP blocks to deploy two complete instances of the algorithm. As a last design, we investigated also the use of Python for the FPGA algorithm development. We evaluated different configurations exploiting the ACP and HP ports and analysed the impact of the clock frequency on the execution time. The experimental results showed that the port configuration does not affect remarkably the performance of QS, while, as expected, a higher clock frequency reduces the execution times. All the tests performed clearly highlighted that the bare-metal implementation using a single QS IP block and a single DMA to communicate between the PS and the PL largely outperformed the other

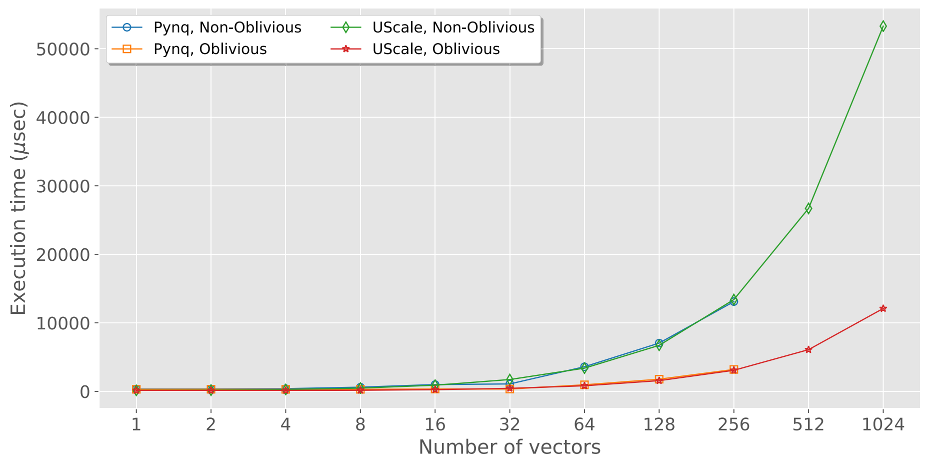


Fig. 10. Execution times obtained with the QS Python development on the PYNQ-Z1 and the UltraScale devices.

architectural designs while the use of Python introduce enormous and unacceptable overheads.

Interestingly, we showed that our bare-metal implementations achieve nearly constant execution times as we increase the number of feature vectors processed until the limits of the hardware are reached and the BRAM is saturated. Similarly, also the number of trees in the model impacts only slightly the inference time. We recall that scaling on both these dimensions is very important since in many applications the inference has to be performed on very large batches of items and larger the number of trees in the ensemble more accurate is in general the ML model [5]. A solution like the one investigated in this paper, which provides almost constant inference time if the saturation of the FPGA resources is not reached, can have a high impact on many application scenarios (e.g., Web or product search, social media ranking or recommendation, on-line advertisement, etc.) where engineers have very strict and contrasting requirements on latency, accuracy and hardware cost to satisfy.

As future work, we plan to optimize the representation used for the feature vectors in order to reduce their memory occupation and consequently increase the number of instances scored in parallel on low-cost FPGA devices. Moreover, we plan to investigate the relations between SoC capacity and QS execution time to derive a general cost and performance model for fine-tuning FPGA-accelerated inference tasks.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgments

This work was partially supported by the project *HAMLET: Hardware Acceleration of Machine LEarning Tasks* funded by CONICET (Argentina) and CNR (Italy) 2017–2018 collaboration program, by the TEACHING project, funded by the EU Horizon 2020 Research and Innovation program (Grant agreement ID: 871385), and by the OK-INSAID project, funded by the Italian Ministry of Education and Research (GA no. ARS01_00917).

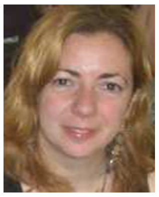
References

[1] H. Bai, H. Luo, C. Liu, D. Paire, F. Gao, A device-level transient modeling approach for the fpga-based real-time simulation of power converters, IEEE Trans. Power Electron. 35 (2) (2019) 1282–1292.
[2] L.A. Barroso, U. Hölzle, The case for energy-proportional computing, Computer 40 (12) (2007) 33–37.
[3] L.A. Barroso, J. Dean, U. Holzle, Web search for a planet: the Google cluster architecture, IEEE MICRO 23 (2) (2003) 22–28, https://doi.org/10.1109/MM.2003.1196112.
[4] B. Behnam, M. Mansouryar, Modeling and simulation of a dc motor control system with digital pid controller and encoder in fpga using Xilinx system generator, in: Instrumentation Control and Automation, 2011, pp. 104–108.
[5] G. Capannini, C. Lucchese, F.M. Nardini, S. Orlando, R. Perego, N. Tonellotto, Quality versus efficiency in document scoring with learning-to-rank models, Inf. Process. Manag. 52 (6) (2016) 1161–1177.
[6] L. Dagum, R. Menon, Openmp: an industry-standard api for shared-memory programming, IEEE Comput. Sci. Eng. 5 (1) (1998) 46–55, https://doi.org/10.1109/99.660313.
[7] D. Dato, C. Lucchese, F.M. Nardini, S. Orlando, R. Perego, N. Tonellotto, R. Venturini, Fast ranking with additive ensembles of oblivious and non-oblivious regression trees, ACM Trans. Inf. Syst. 35 (2) (2016) 15:1–15:31.
[8] J.H. Friedman, Greedy function approximation: a gradient boosting machine, Ann. Stat. (2001) 1189–1232.
[9] G. Georgis, G. Lentaris, D. Reisis, Acceleration techniques and evaluation on multi-core cpu, gpu and fpga for image processing and super-resolution, J. Real-Time Image Process. 16 (4) (2019) 1207–1234.
[10] V. Gil-Costa, R.S. Molina, R. Petrino, C.F.S. Paez, A.M. Printista, J.D.D. Gazzano, Field-programmable gate array (FPGA) technologies for high performance instrumentation, in: IGI GLobal, 2004, pp. 138–170, Ch. Hardware Acceleration of CBIR System with FPGA-Based Platform.
[11] X. He, J. Pan, O. Jin, T. Xu, B. Liu, T. Xu, Y. Shi, A. Atallah, R. Herbrich, S. Bowers, J.Q. Candela, Practical lessons from predicting clicks on ads at Facebook, in: Proc. 8th International Workshop on Data Mining for Online Advertising, 2014, pp. 5:1–5:9.
[12] U. Heinkel, W. Glauert, M. Wahl, The VHDL Reference: A Practical Guide to Computer-Aided Integrated Circuit Design (Including VHDL-AMS) with Other, John Wiley & Sons, Inc., New York, NY, USA, 2000.
[13] T. Tracy II, Y. Fu, I. Roy, E. Jonas, P. Glendenning, Towards machine learning on the automata processor, in: High Performance Computing, 2016, pp. 1–19.
[14] A.A. Ingle, V.G. Raut, Hardware software co-simulation of edge detection for image processing system using delay block, in xsg, Res. Eng. Technol. 3 (4) (2014) 549–553.
[15] K. Järvelin, J. Kekäläinen, Cumulated gain-based evaluation of ir techniques, ACM Trans. Inf. Syst. 20 (4) (2002) 422–446.
[16] P. Langley, S. Sage, Oblivious decision trees and abstract cases, in: Working Notes of the AAAI-94 Workshop on Case-Based Reasoning, Seattle, WA, 1994, pp. 113–117.
[17] F. Lettich, C. Lucchese, F.M. Nardini, S. Orlando, R. Perego, N. Tonellotto, R. Venturini, Parallel traversal of large ensembles of decision trees, IEEE Trans. Parallel Distrib. Syst. 30 (9) (2019) 2075–2089.
[18] L. Li, C. Sau, T. Fanni, J. Li, T. Viitanen, F. Christophe, F. Palumbo, L. Raffo, H. Huttunen, J. Takala, S.S. Bhattacharyya, An integrated hardware/software design methodology for signal processing systems, J. Syst. Archit. 93 (2019) 1–19.
[19] X. Lin, R.S. Blanton, D.E. Thomas, Random forest architectures on fpga for multiple applications, in: Proceedings of the on Great Lakes Symposium on VLSI, 2017, pp. 415–418.
[20] Z. Lin, S. Sinha, W. Zhang, Towards efficient and scalable acceleration of online decision tree learning on fpga, in: IEEE 27th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM), 2019, pp. 172–180.
[21] T.-Y. Liu, Learning to rank for information retrieval, Found. Trends Inf. Retr. 3 (3) (2009) 225–331.

[22] C. Lucchese, F.M. Nardini, S. Orlando, R. Perego, N. Tonellotto, R. Venturini, Quickscorer: a fast algorithm to rank documents with additive ensembles of regression trees, in: Proc. ACM SIGIR, 2015, pp. 73–82.
[23] C. Lucchese, F.M. Nardini, S. Orlando, R. Perego, N. Tonellotto, R. Venturini, Exploiting CPU SIMD extensions to speed-up document scoring with tree ensembles, in: Proc. ACM SIGIR, 2016, pp. 833–836.
[24] S. Majumder, J.F. Dalsgaard Nielsen, T. Bak, A. la Cour-Harbo, Reliable flight control system architecture for agile airborne platforms: an asymmetric multiprocessing approach, Aeronaut. J. 123 (1264) (2019) 840–862.
[25] S. Min, S. Huang, M. El-Hadedy, J. Xiong, D. Chen, W. Hwu, Analysis and optimization of I/O cache coherency strategies for soc-fpga device, in: 29th International Conference on Field Programmable Logic and Applications, FPL 2019, Barcelona, Spain, September 8–12, 2019, 2019, pp. 301–306.
[26] S. Min, S. Huang, M. El-Hadedy, J. Xiong, D. Chen, W. Hwu, Analysis and optimization of I/O cache coherency strategies for soc-fpga device, CoRR, arXiv:1908.01261 [abs].
[27] J. Miteran, J. Matas, J. Dubois, E. Bourennane, Automatic fpga based implementation of classification tree, in: Symposium on Signals, Circuits and Systems (SCS), 2004, pp. 189–192.
[28] A. Mohammed, E. Rachid, H. Laamari, High level fpga modeling for image processing algorithms using Xilinx system generator, Comput. Sci. Telecommun. 5 (6) (2014) 1–8.
[29] K. Nagarajan, B. Holland, A.D. George, K.C. Slatton, H. Lam, Accelerating machine-learning algorithms on fpgas using pattern-based decomposition, Signal Process. Syst. 62 (1) (2011) 43–63.
[30] H. Nakahara, A. Jinguji, T. Fujii, S. Sato, An acceleration of a random forest classification using altera SDK for OpenCL, in: Proceedings of the International Conference on Field-Programmable Technology, 2016, pp. 289–292.
[31] R. Narayanan, D. Honbo, G. Memik, A. Choudhary, J. Zambreno, Interactive presentation: an fpga implementation of decision tree classification, in: Proceedings of the Conference on Design, Automation and Test in Europe, 2007, pp. 189–194.
[32] K. Neshatpour, M. Malik, M.A. Ghodrat, A. Sasan, H. Homayoun, Energy-efficient acceleration of big data analytics applications using fpgas, in: Proceedings of the IEEE International Conference on Big Data (Big Data), 2015, pp. 115–123.
[33] M. Owaida, G. Alonso, Application partitioning on FPGA clusters: inference over decision tree ensembles, in: International Conference on Field-Programmable Logic and Applications, 2018, pp. 295–300.
[34] M. Owaida, H. Zhang, C. Zhang, G. Alonso, Scalable inference of decision tree ensembles: flexible design for CPU-FPGA platforms, in: International Conference on Field Programmable Logic and Applications, 2017.
[35] S. Palnitkar, Verilog®Hdl: A Guide to Digital Design and Synthesis, second edition, Prentice Hall Press, Upper Saddle River, NJ, USA, 2003.
[36] J. Pérez Fernández, M. Alcázar Vargas, J.M. Velasco García, J.A. Cabrera Carrillo, J.J. Castillo Aguilar, Low-cost fpga-based electronic control unit for vehicle control systems, Sensors 19 (8) (2019) 1834.
[37] B. Popa, M. Roman, R.L. Constantinescu, Fast Fourier processing and real-time transformation system for a dynamic vibration signal, in: 20th International Carpathian Control Conference (ICCC), 2019, pp. 1–6.
[38] M. Qasaimeh, K. Denolf, J. Lo, K. Vissers, J. Zambreno, P.H. Jones, Comparing energy efficiency of cpu, gpu and fpga implementations for vision kernels, in: 2019 IEEE International Conference on Embedded Software and Systems (ICESS), 2019, pp. 1–8.
[39] I. Segalovich, Machine learning in search quality at Yandex, in: Presentation at the Industry Track of the 33rd Annual ACM SIGIR Conference, 2010, https://goo.gl/xUAq3r.
[40] A. Shchekalev, Using GPUs to accelerate learning to rank, https://goo.gl/seikPf, 2014.
[41] D. Sorokina, E. Cantu-Paz, Amazon search: the joy of ranking products, in: Proc. ACM SIGIR, 2016, pp. 459–460.
[42] A. Thangavelu, M. Varghese, M. Vaidyan, Novel fpga based controller design platform for dc-dc buck converter using hdl co-simulator and Xilinx system generator, in: Instrumentation Control and Automation, 2012, pp. 270–274.
[43] B. Van Essen, C. Macaraeg, M. Gokhale, R. Prenger, Accelerating a random forest classifier: multi-core, gp-gpu, or fpga?, in: IEEE 20th International Symposium on Field-Programmable Custom Computing Machines, 2012, pp. 232–239.
[44] M. Vidal, R. Cruces, G. Zurita, Digital fir filter design for diagnosing problems in gears and bearings using Xilinx's system generator, in: 2014 IEEE ANDESCON, 2014, pp. 1–1.
[45] E.C. Vivas González, D.M. Rivera Pinzón, E.J. Gomez, Implementation and simulation of iir digital filters in fpga using Matlab system generator, in: 2014 IEEE 5th Colombian Workshop on Circuits and Systems (CWCAS), 2014, pp. 1–5.
[46] Q. Wu, C.J. Burges, K.M. Svore, J. Gao, Adapting boosting for information retrieval measures, Inf. Retr. 13 (3) (2010) 254–270.
[47] T. Xia, Y. Tian, J.-C. Prevotet, F. Nouvel, Ker-one: a new hypervisor managing fpga reconfigurable accelerators, J. Syst. Archit. 98 (2019) 453–467.
[48] J. Yuan, X. Guo, C. Wang, X. You, Fpga resource optimization method for hardware in the loop real-time simulation of power converters, in: 2019 IEEE Applied Power Electronics Conference and Exposition (APEC), 2019, pp. 2849–2854.

Romina S. Molina received her master's degree "Master in Computer Science" from Universidad Nacional de San Luis (UNSL) Argentina, in 2017, and her bachelor's degree "Electronic engineering with an orientation in digital systems" from Universidad Nacional de San Luis (UNSL) Argentina, in 2010. Her main research interests are digital signal processing, digital control, high performance computing, system retrieval, FPGA and SOC. She is a member of the investigation group "Visión artificial y control digital" at UNSL, a member of the Multidisciplinary Laboratory, ICTP, Trieste a member of the Laboratorio di Elaborazione Segnali e Immagini - IPL, Università degli studi di Trieste. Currently she is realizing her PhD in Industrial and Information Engineering, at Università degli studi di Trieste.

Fernando Loor received his degree in Electronic Engineering (2016) at Universidad Nacional de San Luis (UNSL), Argentina. He is a PhD. student in Computer Science at UNSL. He holds a scholarship from CONICET. He is also a professor assistant in the courses of "Signals and Systems" and "Digital Signal Processing" at the UNSL. His email address is floor@unsl.edu.ar.


Veronica Gil-Costa received her MSc (2006) and PhD (2009) in Computer Science, both from Universidad Nacional de San Luis (UNSL), Argentina. She is a former researcher at Yahoo! Labs Santiago hosted by the University of Chile. She is currently an associate professor at the University of San Luis and researcher at the National Research Council (CONICET) of Argentina. Her email address is gvcosta@unsl.edu.ar.

Franco Maria Nardini (http://hpc.isti.cnr.it/~nardini) is a researcher with the National Research Council of Italy. His research interests focus on web information retrieval, machine learning and data mining. He authored more than 50 papers in peer reviewed international journals and conferences. In 2015, he received the ACM SIGIR 2015 Best Paper Award.

Raffaele Perego (http://hpc.isti.cnr.it/~raffaele) is a research director at ISTI-CNR, where he leads the High Performance Computing Lab (http://hpc.isti.cnr.it/). His main research interests include large-scale information systems, information retrieval, web mining and artificial intelligence. He co-authored more than 170 papers on these topics published in journals and proceedings of international conferences. He chaired the ACM SIGIR conference in 2016 and the ECIR conference in 2020.

Salvatore Trani is a researcher with the Italian National Research Council. He received the Ph.D. in Computer Science from the University of Pisa in 2017. His research interests focus on Information Retrieval (IR), Machine Learning (ML) and Semantic Enrichment. He served as a program committee member of several top-level conferences of IR and ML. He authored more than 15 papers in peer-reviewed international journals, conferences and other venues.