Parallel Traversal of Large Ensembles of Decision Trees

Francesco Lettich, Claudio Lucchese, Franco Maria Nardini, Salvatore Orlando, Raffaele Perego, Nicola Tonellotto, Rossano Venturini

Abstract—Machine-learnt models based on additive ensembles of regression trees are currently deemed the best solution to address complex classification, regression, and ranking tasks. The deployment of such models is computationally demanding: to compute the final prediction, the whole ensemble must be traversed by accumulating the contributions of all its trees. In particular, traversal cost impacts applications where the number of candidate items is large, the time budget available to apply the learnt model to them is limited, and the users' expectations in terms of quality-of-service is high. Document ranking in web search, where sub-optimal ranking models are deployed to find a proper trade-off between efficiency and effectiveness of query answering, is probably the most typical example of this challenging issue. This paper investigates multi/many-core parallelization strategies for speeding up the traversal of large ensembles of regression trees thus obtaining machine-learnt models that are, at the same time, effective, fast, and scalable. Our best results are obtained by the GPU-based parallelization of the state-of-the-art algorithm, with speedups of up to 102.6x.

© 2018 IEEE – Final version available at https://doi.org/10.1109/tpds.2018.2860982

Index Terms—Efficient Machine Learning, Learning-to-Rank, Decision Tree Ensembles, Parallel Algorithms, SIMD, NUMA multiprocessors, GPUs.

1 Introduction

Recent advances in Machine Learning (ML) have opened new horizons for modeling phenomena and devising effective and actionable solutions for complex problems in diverse application domains. In many application contexts the widespread adoption of complex machine-learnt models asks for novel efficient algorithmic solutions. Instead of focusing on improving the efficiency of the off-line training phase of complex machine learning models, this paper deals with the deploying phase of such learnt models, by efficiently exploiting their application to a stream of data instances. In particular, the learnt models on which we focus are additive *ensembles of decision trees*. These models, which are generated by boosting meta-algorithms that iteratively learn simple decision trees by incrementally optimizing some given loss function, have been shown to be the most general and competitive solutions for several "difficult" tasks. For example, consider the Yahoo! challenge [?], which fostered the development of novel Learning-to-Rank (LtR) methods, i.e., supervised ML techniques aimed at addressing the fundamental problem of *ranking* items according to their relevance to queries [?], [?]. All these ML-based rankers assign a numerical score to each item, in turn used to reorder the input list, and thus are commonly known as *scorers*. The most robust and effective ranking models resulted the ones based on ensembles of regression trees, learnt with the GBRT [?] and λ-MART [?] algorithms. All top competitors in this challenge leveraged decision trees and ensemble methods, and the winner deployed an ensemble model encompassing a total of 24,000 regression trees. Another notable example regards Yandex, the main Russian search engine, which repeatedly announced the exploitation of very large tree-based ranking models within their systems, and solutions based on multi/many-core parallelism to speed-up both their training and testing [?], [?]. Also Amazon uses more than 100 tree-based models, one model per category per site, for ranking the products returned as answer to user queries [?]. Finally, ensembles of regression trees are used in production for ads clickthrough rate prediction [?], and are the most common choice in solutions for ML competitions, such as Kaggle[1].

Without loss of generality, in the following we concentrate on the Web Search Engine (WSE) scenario, and adopt the LtR terminology to discuss the state-of-the-art and investigate parallel algorithms to traverse large tree-based ensembles. However, all algorithms and processing strategies discussed can be applied "*as is*" in other scenarios, different from document ranking in WSE, since they regard the general problem of traversing large forests of binary decision trees, given an item represented as a *feature vector*. Specifically, they can be thus used to deploy any ensemble model trained on any kind of continue, binary, and ordinal feature. Categorical features are instead not efficiently handled by our solution since it does not support efficient set-inclusion operations.

- *F. Lettich is with Federal University of Ceará, Brazil. E-mail: francesco.lettich@gmail.com*
- *C. Lucchese and S. Orlando are with Ca' Foscari University of Venice, Italy. E-mail: claudio.lucchese@unive.it, orlando@unive.it*
- *F. M. Nardini, R. Perego and N. Tonellotto are with ISTI–CNR, Italy. E-mail: f.nardini@isti.cnr.it, r.perego@isti.cnr.it, n.tonellotto@isti.cnr.it*
- *R. Venturini is with University of Pisa, Italy. E-mail: rossano.venturini@unipi.it*

1. https://www.kaggle.com/competitions

Large scale WSEs commonly exploit LtR solutions within a multi-stage ranking architecture [?], [?], [?], [?], [?] optimized for top-k retrieval. This architecture design aims to find a trade-off between effectiveness and efficiency, by applying increasingly accurate and computationally expensive models at each stage, where each stage re-ranks candidate items coming from the previous stage by also pruning some of them. Ensembles with thousands of regression trees are commonly deployed in the last ranking stage, to achieve high quality results that meet user satisfaction. Due to the incoming rate of queries and QoS expectations, the efficiency requirements of the ranking stage- are very strict, and thus the traversal of complex tree ensemble model to score candidate items has to complete within a very small *time budget*. Table 1 illustrates the difficulty in meeting such requisites, by reporting typical figures of tree-based rankers used by large scale WSEs [?]. Note that, for each query, the ranking stage has to reorder hundreds/thoudsands documents, while each query-document pair is in turn represented by a multidimensional numerical vector of hundreds of features. Each ensemble model is composed of thousand of trees, each featuring tens of leaves, where leaves store the partial numerical contribution of a tree to the final score of a document.

TABLE 1: Typical figures for WSEs' tree ensembles.

Number of trees in the ensemble	$1,000 - 20,000$
Leaves per tree	$4 - 64$
Documents per query	$3,000 - 10,000$
Features per query-document pair	$100 - 1,000$

In this paper we investigate multi/many-core parallelization strategies for making the traversal of ensembles fast and scalable. In particular, we investigate diverse strategies to parallelize QUICKSCORER (QS), the state-of-the-art algorithm for traversing tree ensembles [?], [?] (we present QUICKSCORER in Section 2). Making faster QS allows the deployment of very large, complex, and effective ML models, by still producing the final ranking of a set of documents within a fixed small time budget. Alternatively, when the desired level of accuracy is already granted by a given model, we can rely on a parallel implementation of QS to reduce latency and increase query processing throughput. We devise in Section 3 three possible general strategies for parallelizing QS by exploiting the various opportunities offered by modern CPU architectures and studying them in three separate sections. Specifically, we study: (i) SIMD extensions of CPUs to vectorize the code (Section 4), (ii) multi-core architecture for shared memory multithreading (Section 5), and (iii), many-core graphic cards (GPUs) exploiting massive data parallelism (Section 6). We report in Section 7 on extensive experiments conducted on three publicly available LtR datasets. In the experiments we fine-tune the parallel algorithms to investigate strengths and limitations of the proposed solutions. Although the experimental setting is the one commonly used by the scientific community to evaluate LtR solutions, the results achieved and the lessons learnt are completely general and can be exported without modifications to other use cases characterized by similar efficiency and effectiveness requirements – for instance, product search and recommendation,

Algorithm 1: QUICKSCORER

1. QUICKSCORER ($\mathbf{x}$,$\mathcal{T}$):
2.   **foreach** $T_h \in \mathcal{T}$ **do**
3.   `leafindexes[`h`]` $\leftarrow$ `11...11`
4.   **foreach** $f_\phi \in \mathcal{F}$ **do**   `// Mask Computation`
5.   **foreach** $(\gamma$, `mask`, $h) \in \mathcal{N}_\phi$ *in asc. order of* γ **do**
6.   **if** $\mathbf{x}[\phi] > \gamma$ **then**
7.   `leafindexes[`h`]` $\leftarrow$ `leafindexes[`h`]` $\wedge$ `mask`
8.   **else**
9.   **break**
10. $score \leftarrow 0$
11.   **foreach** $T_h \in \mathcal{T}$ **do**   `// Score Computation`
12. $j \leftarrow$ index of leftmost bit set to 1 of `leafindexes[`h`]`
13. $l \leftarrow h \cdot \Lambda + j$
14. $score \leftarrow score +$ `leafvalues[`l`]`
15.   **return** $score$

social media filtering and ranking, on-line advertisement, classification or regression tasks on big data. Finally, Section 8 discusses related works and concluding remarks are presented in Section 9.

2 QUICKSCORER

Let us denote with $\mathcal{T} = \{T_0, T_1, \ldots\}$ an ensemble of binary decision trees, and let Λ be the maximum number of leaves of each tree. Moreover, let $\mathbf{x}$ be the feature vector representing an input instance (e.g., a query-document pair in the LtR WSE scenario). Let $\mathcal{F}$ be the feature set, and let $|\mathcal{F}|$ be the number of dimensions of vector $\mathbf{x}$. We use ϕ to refer to the $\phi-$th feature, with $\mathbf{x}[\phi]$ storing the value of feature $f_\phi \in \mathcal{F}$. Moreover, let $s(\mathbf{x})$ be the numerical score eventually computed for $\mathbf{x}$ by traversing $\mathcal{T}$.

Branching decisions in internal nodes of a tree take the form of a boolean test $\mathbf{x}[\phi] \leq \gamma$, where γ is a real-valued threshold for feature f_ϕ. The output of each decision tree $T_h \in \mathcal{T}$, i.e, its contribution to the score $s(\mathbf{x})$, corresponds to the so-called *exit leaf* of T_h, identified by traversing the tree with the input instance $\mathbf{x}$. QS identifies this leaf by a bitvector `leafindexes[`h`]`, made of Λ bits, one per leaf.[2] Specifically, the bits of this bitvector set to 0 denote the leaves that cannot be reached during the tree traversal for a given $\mathbf{x}$.

The traversal of a decision tree performed by QS can be viewed as the process of converting a bitvector `leafindexes[`h`]`, where all bits are initially set to 1, to a final bitvector where the *leftmost 1* identifies the *exit leaf* of the tree [?]. 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 nodes of T_h. 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[`h`]` 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[`h`]` corresponding to the unreachable leaves

2. Hereinafter we will focus on ensembles whose trees have 32 or 64 leaves ($\Lambda = 32, 64$), since they provide the best trade-off between effectiveness and efficiency in the LtR scenario [?].

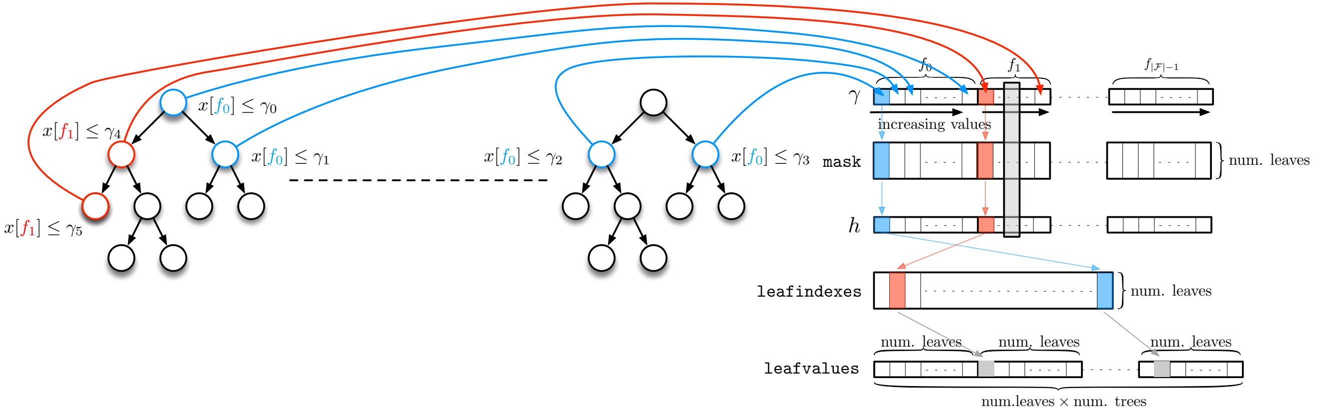


Fig. 1: Data layout example of the QS algorithm.

of T_h, 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 **[?]** for the formal proof of the correctness of this process.

Alg. 1 illustrates the QS **[?]**, **[?]** algorithm for the fast traversal of the ensemble. The algorithm restructures the data layout of an ensemble of regression 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 query-document pair, is transformed into a scan of linear arrays.

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 of the *logical AND* operand, according to which the masking operations for traversing each tree of the ensemble can be made in arbitrary order. Specifically, for each feature f_ϕ, it builds a list $\mathcal{N}_\phi$ of tuples $(\gamma, \texttt{mask}, h)$, where γ is the test threshold of a branching node of tree T_h performing a test over the feature f_ϕ of the input instance $\mathbf{x}$, and `mask` is the pre-computed mask that identifies the leaves of T_h that are un-reacheable when the associated test evaluates to false. The data structure layout is illustrated in Fig. 1. Hereinafter, we refer to the tuples $(\gamma, \texttt{mask}, h)$ 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 document dependent and updated at runtime. $\mathcal{N}_\phi$ is sorted in ascending order of γ. Hence, when processing $\mathcal{N}_\phi$ sequentially, as soon as a test evaluates to true, i.e., $\mathbf{x}[\phi] \leq \gamma$, the remaining occurrences of $\mathcal{N}_\phi$ evaluate to true as well, and thus their evaluation can be safely skipped. We call *mask computation* the first step, during which all the bitvectors `leafindexes[`h`]` are updated, and *score computation* the second step, where such bitvectors are used to retrieve tree predictions.

To make efficient and cache-friendly the access to the QS data structure we adopt a *Struct of Arrays* (SoA) data layout rather than a classic *Array of Structs* (AoS) **[?]**. According to the SoA layout, the tuples $(\gamma, \texttt{mask}, h)$ are stored in three independent arrays, hence solving possible alignment issues due to different sizes of the fields of each tuple and simplifying the exploitation of data parallelism.

Time and Space Complexity. The running time of QS depends on the number of *false nodes* evaluated during the mask computation step and on the number of partial scores accumulated during the score computation step. In the worst case, the number of false nodes evaluated is $\Lambda|\mathcal{T}|$ and the number of partial scores accumulated is $|\mathcal{T}|$. Therefore, the complexity of QS in the worst case is linear time, i.e., $\Theta(\Lambda|\mathcal{T}|)$ time. However, experiments on real datasets **[?]** show that the number of false nodes visited per tree is only a small fraction of Λ, and, admittedly, QS outperforms other algorithms with better worst-case time complexity.

Concerning the space needed to score a feature vector $\mathbf{x}$ of $|\mathcal{F}|$ elements, the read-only data structures used to store the tree ensemble $\mathcal{T}$ depends on the number of trees ($|\mathcal{T}|$), the number of internal branching nodes ($\Lambda - 1$) and leaves (Λ) of each tree. Specifically, for each internal branching node QS uses $\Lambda/8{+}2{+}4 = \Theta(\Lambda)$ bytes to store, respectively, the node's bitvector `mask` of Λ bits, the corresponding tree ID h (stored as a 16 bit int), and the threshold γ used in the test (stored as a single precision float). In total, $|\mathcal{T}|(\Lambda - 1)(\Lambda/8 + 6) = \Theta(|\mathcal{T}|\Lambda^2)$ bytes to store the internal nodes of the whole ensemble. In addition, QS stores in `leafvalues` the score contribution of each leaf (stored as a double), for a total of $8\Lambda|\mathcal{T}|$ bytes, and manages a per-tree bitvector `leafindex` for a total of $\Lambda\mathcal{T}/8$ bytes. Thus the space complexity of QS is $\Theta(|\mathcal{T}|\Lambda^2)$ bytes.

3 QuickScorer Parallelization

In the following we will focus our analysis on the parallelization of the mask computation step as the parallelization of the score computation step can be achieved with a a straightforward parallel reduction.

Given a set of query-document pairs, we want to investigate the following scoring parallelization strategies:

- *Inter-document parallelism:* multiple documents are evaluated in parallel;
- *Intra-document parallelism*: multiple features, trees, or nodes are evaluated in parallel;
- *Hybrid parallelism*: combining the two strategies above.

Inter-document parallelism. The rationale behind this strategy stems from the observation that each document can be scored independently. Inter-document parallelism

takes advantage of this property, and employs multiple threads to score simultaneously multiple documents. Although the latency associated with each document scoring does not improve over the sequential case, inter-document parallelism ensures better throughput in terms of number of documents scored per time unit. To realize this strategy, the data structure associated with the model must be shared among all the threads, while each document must be associated with its own copy of `leafindexes`.

Intra-document parallelism. The key idea behind this strategy is to partition the scoring of a single document into subtasks that can be executed in parallel. Consequently, intra-document parallelization aims at reducing the scoring latency of each document, which in turn has the effect of increasing the throughput.

Subtasks can be naturally identified in QS by decomposing the work performed over features; more precisely, each subtask consists in *processing* the list of tuples $(\gamma, \texttt{mask}, h) \in \mathcal{N}_\phi$ associated with a single feature f_ϕ, and *updating* the corresponding `leafindexes`. Note that different tuples in $\mathcal{N}_\phi$, related to different features f_ϕ, may be associated with the same tree T_h. As a consequence, updating `leafindexes` may generate race conditions that have to be properly managed. Depending on the targeted architecture, race conditions can be generally managed in two different ways. On the one hand, one can eliminate race conditions by creating one copy of `leafindexes` per subtask, provided that increasing the memory footprint does not represent an issue; this strategy, however, incurs in the additional cost of having to perform a final merge of the various `leafindexes` – this can be achieved by logical AND operations. On the other hand, if memory occupancy represents a major concern (such as in the case of GPUs) `leafindexes` must be shared across the subtasks, thus requiring the use of atomic updates to manage race conditions.

Hybrid parallelism. This strategy exploits the combined use of massive and fine-grained parallelism. The idea is to process p_1 documents independently in parallel (inter-document parallelism) by using p_2 threads to score each document (intra-document parallelism), for a total of $p = p_1 \cdot p_2$ threads.

Tuning performance. All the above strategies have room for several performance improvements that attempt to leverage *task granularity* and *model partitioning*. Given a workload split in independent tasks among a pool of concurrent workers, task granularity impacts *load balancing*: in general, the smaller the granularity and the larger the number of tasks, the better the resulting balancing. For each of the parallelization strategies introduced above, task granularity can be opportunely tuned from the finest to the coarsest level; in the inter-document case, the finest granularity can be achieved by associating each task with a single document, while in the intra-document case the finest granularity is achieved by associating each task with a single feature. In both cases, granularity can be simply increased by assigning multiple documents (features) to individual task.

As modern CPU and GPU architectures feature complex memory hierarchies, devising algorithms with a reduced memory footprint may provide remarkable benefits: smaller data structures, accessed with high spatial and temporal locality, easily fit into the smaller – but faster – cache memories. Large tree ensembles, however, may be too large to even fit the lowest level of cache. Consequently, an ensemble model may need to be *partitioned* in blocks of trees, such that the data structure of each block fits into a given cache size.

The final score of a document then becomes the sum of the scores produced by the various blocks. In this context we note that inter-document parallelism improves the temporal locality of memory accesses, as smaller blocks of the model are used to score the documents; however, the very same parallelism requires multiple copies of `leafindexes`, thus increasing the memory footprint of the algorithm. Overall, making the best use of cache memories requires to find a proper trade-off between the size of tree blocks and the number of documents evaluated in parallel.

Correctness and complexity. We discuss here the correctness of the parallelization strategies illustrated above, starting from the most expensive mask computation step.

The first strategy, named *inter-document*, is straightforwardly correct. Each parallel thread scores one of n documents, and for this purpose is provided with a private copy of the `leafindexes` bitvectors so as to make the scoring of each document independent. The second parallelization strategy, named *intra-document*, performs a parallel visit of the branching nodes of the given ensemble. Note that the parallel visiting order is different from the sequential one. This does not affect the correctness of QS, thanks to the commutativity and associativity of the *logical AND* operations applied to the `leafindexes` bitvectors. However, this parallelism introduces race conditions on `leafindexes` which can be solved either by using per-thread replicas of `leafindexes` that are eventually merged together, or by enforcing atomic updates.

Regarding the score computation step, a trivial parallel add reduction is performed, after accessing in parallel the exit leaves of all trees in $\mathcal{T}$, to retrieve their additive contributions to the final score of a given query-document pair.

The worst-case parallel complexity of QS on p processors for scoring n documents is $\Theta\big(\frac{n\Lambda|\mathcal{T}|}{p}\big)$ time. Limited slowdowns with respect to a linear speedup derive from synchronization overheads (atomic updates in *intra-document* parallelism) or from an increase in the memory footprint (replication in *inter-document* parallelism), which may negatively impact the performance of the various levels of dedicated/shared caches.

4 VECTORIZED QUICKSCORER

In this section we discuss V-QUICKSCORER (vQS)[3], the enhanced single-threaded version of QS that exploits *CPU vector extensions* to leverage Data-Level Parallelism (DLP) (preliminary results concerning this section were already presented in [**?**]). The extended DLP instruction sets present in modern CPUs permit the parallel execution of the same operation on different data items, i.e., they allow to realize

3. Source code: https://github.com/hpclab/vectorized-quickscorer

the *single instruction multiple data* (SIMD) paradigm. Specifically, Streaming SIMD Extensions (SSE) and Advanced Vector Extensions (AVX) are sets of Intel's *vector instructions* that exploit wide registers of 128 and 256 bits, where the registers pack multiple elements of some simple data type. For instance, a 256 bit register can store eight single precision or four double precision floats. We note that recent high-end processors already provide support for even larger, i.e., 512 bits, registers. For the sake of clarity, we hereinafter refer to the 256 bits register architecture (AVX), the most common SIMD extension present in current Intel CPUs; we observe that VQS can be modified to support wider registers in a straightforward manner.

Inter-document parallelism represents the most natural source of parallelism that can be exploited in VQS, although the number of documents scored in parallel is bounded by the number of parallel ways of the AVX-256 extension. Both the *mask computation* and *score computation* steps of QS can be parallelized. During the first step, multiple documents can be tested against a given node condition, and their `leafindexes` updated in parallel. Similarly, the scores of multiple documents can be computed simultaneously during the second step. Inter-document parallelism requires to replicate the data structure `leafindexes` used to encode the exit leaves. VQS interleaves bitvectors of 8 different documents (256 bits) in consecutive memory locations, i.e., `leafindexes[8h+i]`, $\forall i = 7{:}0$. This allows to update 8 `leafindexes` simultaneously with a single SIMD operation.

In the following we adopt a generic vectorial notation where variables stored in wide registers are accented by a right arrow and SIMD operations have a subscript denoting the width of the parallel operand. For example, the subscript s used in the following parallel operand:

$$\vec{c} \leftarrow \quad \vec{a} >_s \vec{b}$$

corresponds to a SIMD operation where an *element-wise greater-than* comparison is executed between $\vec{a}$ and $\vec{b}$ on their operands with *single precision* equivalent size (the subscript d is used for double precision) and the result is stored to $\vec{c}$. If 256-bit registers are used, this parallel operation $>_s$ would perform 8 simultaneous comparison on 32-bit operands, while $>_d$ would perform 4 simultaneous operations on 64-bit operands. Finally, we use the notation $\mathtt{m}[end : start]$ to address consecutive elements of the array `m` from *start* to *end* inclusive.

Specific optimizations used by VQS depend on the maximum number of leaves Λ in the trees of the ensemble. Alg. 2 shows the pseudocode of VQS when $\Lambda = 32$. As regards the *mask computation* step, VQS identifies *false nodes* like QS, by processing feature thresholds in ascending order; however, VQS exploits 256 bit registers to compare multiple documents simultaneously against each feature threshold. Since document features are stored as single precision floats, we have a first register, $\vec{\gamma}$, that stores 8 copies of the same test threshold γ, and a second register, $\vec{x}$, that stores the features $\{\mathbf{x}_i[\phi]\}_{i=0,1,\dots,7}$ of the 8 input instances (lines 6-7). A single SIMD instruction is then used to test the feature values of the documents against the threshold (line 8): if all tests evaluate to false (line 9), i.e., VQS does not find any *false node*, the inner loop terminates and the next feature is processed. Unlike QS, VQS needs to pass the test over 8 documents at a time to break the loop (line 10), which introduces some overhead.

Algorithm 2: VQS (with $\Lambda = 32$)

```
1  V-QuickScorer( {x_i}_{i=0,1,...,7}, T, scores[7:0]):
2    foreach T_h ∈ T do
3      ∀i = 7:0 : leafindexes[8h + i] ← 11...11
     ① Mask Computation Step
4    foreach f_φ ∈ F do
5      foreach (γ, mask, h) ∈ N_φ in asc. order of γ do
6        →γ ← (γ, γ, γ, γ, γ, γ, γ, γ)
7        →x ←
           (x_7[φ], x_6[φ], x_5[φ], x_4[φ], x_3[φ], x_2[φ], x_1[φ], x_0[φ])
8        →c ← →x >_s →γ
9        if →c = 0 then
10         break
11       →m ← (mask, mask, ..., mask)
12       →b ← (leafindexes[8h + 7], ..., leafindexes[8h + 0])
13       →y ← →m ∧_s →b
14       leafindexes[8h + 7 : 8h + 0] ←(→c)— →y
     ② Score Computation Step
15   →s1 ← (0, 0, 0, 0)
16   →s0 ← (0, 0, 0, 0)
17   foreach T_h ∈ T do
18     ∀i = 7:0 : j_i ← index leftmost 1 bit of leafindex_i[h]
19     ∀i = 7:0 : l_i ← h · Λ + j_i
20     →v1 ← (leafvalues[l_7], ..., leafvalues[l_4])
21     →v0 ← (leafvalues[l_3], ..., leafvalues[l_0])
22     →s1 ← →s1 +_d →v1
23     →s0 ← →s0 +_d →v0
24   scores[7:4] ← →s1
25   scores[3:0] ← →s0
```

The bitvector update is implemented as shown in Alg. 2 (lines 12–14). First, `mask` is replicated eight times into $\vec{m}$, then the eight $\mathtt{leafindexes}[8h + 7 : 8h + 0]$ are loaded to the register $\vec{b}$. Subsequently, a *bit-wise logical and* between $\vec{b}$ and $\vec{m}$ produces the updated bitvectors, regardless of the test outcome $\vec{c}$. Finally, a *masked store* operation is used to copy back to memory the updated bitvectors: the copy is performed only for the bits of $\vec{y}$ where the corresponding bits $\vec{c}$ are set.

The *score computation* step is also parallelized (Alg. 2, from line 15). To provide the required precision, tree predictions are stored as double precision float values (64 bits) – this implies that only 4 document scores can be processed simultaneously using 256 bits registers. Thus, VQS uses two registers, namely $\vec{s_1}$ and $\vec{s_0}$, to maintain the scores of 8 documents. For each tree h, the predicted partial scores relative to the 8 input instances $\{\mathbf{x}_i\}_{i=0,1,\dots,7}$ are similarly stored to $\vec{v_1}$ and $\vec{v_0}$, and added up to update the final document scores. Finally, the final document scores are copied to $\mathtt{scores}[7{:}0]$.

Processing ensemble models with trees featuring many leaves, e.g. $\Lambda = 64$, impacts on the size of the arrays of bitvectors `leafindexes` and `mask`. As a consequence, the number of elements that can be processed simultaneously in a register decreases. Note that $\vec{c}$ stores each result of a threshold comparison test as a string of 32 bits; conversely, the bitvectors `mask` and `leafindexes`, stored respectively in $\vec{m}$ and $\vec{b}$, are 64 bits wide. VQS handles this mismatch as follows: instead of processing 8 input feature vectors $\mathbf{x}_i[\phi]$ at once, VQS processes only 4 vectors in parallel by packing them with replication:

$\vec{x} \leftarrow (\mathbf{x}_3[\phi], \mathbf{x}_3[\phi], \mathbf{x}_2[\phi], \mathbf{x}_2[\phi], \mathbf{x}_1[\phi], \mathbf{x}_1[\phi], \mathbf{x}_0[\phi], \mathbf{x}_0[\phi])$

Indeed, by replicating twice each $\mathbf{x}_i[\phi]$ VQS produces 64 bits masks that represent the output of the comparison tests, thus solving the mismatch; in turn, this allows to process the register $\vec{x}$ without any other modification of Alg. 2.

5 MULTI-THREADED QUICKSCORER

The SIMD-based parallelization strategies presented in Section 4 are restricted within a single thread running on a single CPU core. Since modern CPUs are multiprocessors that provide several cores, it is interesting to investigate a multi-threaded parallelization of QUICKSCORER. We can mix SIMD and MIMD parallelism by running multiple threads, where each thread exploits inter-document fine-grained SIMD parallelism as discussed in Section 4. We call VQS-MT (Vectorized QUICKSCORER Multi-Threading[4]) such hybrid parallel implementation of QUICKSCORER.

Even in the case of multi-threading parallelism we have different possible parallelization strategies, either inter-, intra-document, or hybrid. From preliminary tests we found that *inter-document* is always the best choice: this is due to the large number of documents to score per query (from hundreds to thousands in real settings) and the limited number of cores of multicores/multiprocessors (from tens to hundred in common configurations). We adopt the same inter-document strategy among the various threads and within each single thread; for example, a multicore CPU with 8 cores, each with 256 bits wide AVX vector registers, may run 8 threads with each thread scoring 8 documents (when $\Lambda = 32$) in parallel, for a total of $8 \times 8 = 64$ documents scored in parallel.

In our study we also consider the complexity of the shared-memory architecture of modern multiprocessors. In particular, such system may include several multicore CPUs, also called sockets or nodes, all accessing the same shared memory according to a NUMA (Non-Uniform Memory Access) scheme. To increase memory bandwith, the shared memory in a NUMA scheme is distributed to each node, namely a multicore CPU, thus introducing two different speeds for accessing the shared memory – fast access to the local one and slower access to remote ones; migrating a thread from a multicore CPU to another may hinder performance, hence it is good practice to restrict the execution of threads to the same multicore CPU where they were created.

We implemented the multi-threaded version of QS by using OpenMP [?], an API that supports multi-platform and multi-language shared memory multiprocessing programming. To realize inter-document parallelism with OpenMP, we denote as a `parallel for` the loop that iterates over the documents to score. More precisely, a single-thread program calls Algorithm 2 from within a `for`, thus scoring either 8 (for $\Lambda = 32$) or 4 documents at a time (for $\Lambda = 64$). Using the directives of OpenMP, the output and temporary data structures used to score each group of 4 (8) documents are declared *private*, and thus allocated on a per-thread basis. Specifically, such private data structures are the `leafindexes`[3:0] (`leafindexes`[7:0]) bitmask arrays, and the final `scores`[3:0] (`scores`[7:0]) accumulators.

A final remark concerns the lower levels of cache equipping each multicore CPU, and the possible issues deriving from their shared use by multiple threads. In general, running multiple threads, each operating on a different working set, may increase the pressure on the shared levels, since each thread needs a different cache residency for their data. In our case, however, the largest dataset is the tree-based model that is accessed *read-only* by all the threads. The per-document read-write data, namely the *private* data structures mentioned above, are small. From our tests, we will show that multi-threading does not impact too much on shared cache performance of VQS-MT, except for very large ensemble models.

6 GPU-BASED QUICKSCORER

Before detailing the strategies adopted to parallelize QUICKSCORER on GPUs, we first introduce some GPU background [?], [?], [?]. Although the GPU terminology used in the following refers to the NVIDIA CUDA framework, different programming frameworks and architectures adopt similar solutions with slightly different names.

6.1 GPU architectural background

A GPU includes m multithreaded *streaming multiprocessors* (SMs), each with n cores. Each SM is able to run blocks of threads, i.e., *thread-blocks*, executing the same kernel code. The unit of scheduling is the *warp*, composed of 32 *synchronous data-parallel threads* belonging to a given thread-block. Each SM is equipped with *private registers* used to manage the stack of threads, and a fast but relatively small *shared memory*. Since such shared memory is statically partitioned and assigned to the thread-blocks running on the SM, the amount of shared memory required by each thread-block influences the number of blocks the SM can run concurrently. Moreover, GPUs feature a *global memory* that can be accessed and modified by both the host CPU's cores and the GPU SMs. Accesses to the global memory benefit of an L2 cache, which is shared among all the SMs. In some models, including the one used for our experiments, each SM is also equipped with a dedicated L1 cache.

Performance optimization for GPUs. In the context of GPUs, multi-threading becomes multi-warping as the units of scheduling are groups/warps of 32 identical threads. As in the case of multi-threading, executing warps concurrently is useful to hide latencies, for instance those caused by global memory accesses. To this end, a good strategy is to optimize the utilization of each SM by increasing the number of active warps per SM, thus realizing a sort of *excess parallelism*. This can be attained by increasing the number of thread-blocks concurrently running on each SM, along with the number of threads per block. The maximum number of warps per SM depends, however, on the specific GPU scheduler, on the number of cores per SM, and on other characteristics of the GPU architecture family considered. In addition, due to the SIMD-style execution of warps, it is important to avoid *branch divergence* within warps, since these may cause under-utilization of GPU's computational resources, thus hindering *warp efficiency*.

To achieve optimal performance, the complex memory hierarchy of GPUs must be properly exploited. As regards

4. Source code: https://github.com/hpclab/multithread-quickscorer

global memory, GPU devices try to *coalesce* loads/stores issued by the threads of a warp into as few global memory transactions as possible. Consequently, if threads identified by consecutive IDs access consecutive words in global memory, their accesses can be merged (coalesced) into fewer memory transactions, thus fully exploiting the bandwidth of global memory. Consequently, the cost of accessing the global memory is measured in terms of number of memory transactions needed to load/store memory blocks (of up to 128 bytes). A fast L2 cache, shared by all SMs, may significantly reduce global memory latencies.

The *shared memory* of each SM is structured in interleaved memory banks; memory banks can thus work in parallel to serve concurrent requests from the threads of a warp. However, performance degrades in presence of bank conflicts due to the serialization of conflicting accesses. As a general rule, if an algorithm needs to randomly access a data structure and the access pattern is not predictable, it is desirable to move such data to shared memory. In fact, random memory accesses directed to global memory cannot be coalesced, while the same requests directed to the shared memory may have the chance to be fulfilled with high throughput if memory banks conflicts do not occur. However, given the limited size of shared memory available per thread-block, suitable data structures have to be allocated.

6.2 GPU-QuickScorer

From the above discussion we highlight two main challenges in designing efficient GPU algorithms: providing a sufficiently large degree of parallelism to profit from the thousands of cores available and deal with the GPU complex hierarchy of memories, by properly defining the layout of data structures and orchestrating over such data structures the accesses of the parallel threads.

Inter- and intra-document parallelism and task granularity. To obtain a sufficient parallelism degree in parallelizing QS, we combine inter- and intra-document parallelism, as discussed in Section 3. This hybrid technique allows both coarse and fine-grained parallelism to be exploited, by processing in parallel p_1 documents (*inter-document* parallelism) using p_2 parallel threads for scoring each document (*intra-document* parallelism) – thus yielding $p = p_1 \cdot p_2$ threads running in parallel. The way to realize this parallelism scheme on a GPU is to assign each of the p_1 documents to a different thread-block. Therefore we have $p_1 = M$, where M is the number of thread-blocks allocated, in turn scheduled over the m GPU SMs. Moreover, if N is the number of parallel threads in each thread-block, we have that $p_2 = N$ threads run concurrently to score a given document. We increase the granularity of the inter-document task (see Section 3) by assigning multiple documents per block, since in general we have to score large batches of input documents $D \subseteq \mathcal{D}$, where $|D| \gg M$.

Within each thread-block, rather than assigning each document feature to a single thread of the block, we assign *each feature* to a *warp*. The main reason behind this strategy is to favor coalesced accesses to the tuples $(\gamma, \mathtt{mask}, h)$ associated with a given feature. In addition, since we have usually fewer warps than features ($\frac{N}{32} \ll |\mathcal{F}|$), we also end up increasing the intra-document task granularity by assigning more features per warp.

Model partition and allocation. We recall that QS adopts two main data structures besides the *input* vector $\mathcal{D}$:

- the *model data structure*, composed of the tuples $(\gamma, \mathtt{mask}, h)$ encoding the branch nodes of the forest $\mathcal{T}$, and of `leafvalues`, the vector that stores the scores associated with the leaves of the trees in $\mathcal{T}$;
- the *output* vectors `leafindexes` – one for each tree of the forest $\mathcal{T}$; these are updated during the computation to eventually identify the exit leaves of the trees.

The *model data structure* is *read-only*, and QS accesses it feature-by-feature, with perfect spatial locality. Conversely, QS needs to access randomly the *output* vectors in *read-write* mode, thus not allowing to exploit any predictable access pattern that promote locality. The model data structure is too large to be entirely stored in the GPU shared memory (typically few tens of KBs). For example, let us consider a model $\mathcal{T}$ made up of a forest of $10,000$ trees, with $\Lambda = 64$ leaves each: just for storing the `leafvalues` of all the trees, where each leaf value is represented as a `double` of 8 bytes, we need about 5 MB. The size of global memory, is typically in the order of 4 – 8 GB, thus representing the best candidate to store $\mathcal{T}$. However, high-throughput access to such memory is possible only when exploiting *coalescing*.

QS perfectly fits the above requirement: the model data structure is stored as a *Structure of Arrays* in global memory, and it is accessed sequentially *feature-by-feature* by means of *linear scans* that promote spatial locality. More precisely, we assign each feature to a warp, thus assigning to the threads of the warp consecutive memory locations that store the values of the tuples $(\gamma, \mathtt{mask}, h)$ associated with a feature $f_\phi \in \mathcal{F}$: in turn, this leads to coalesced accesses. We finally mention that partitioning $\mathcal{T}$ increases the chance that individual tree-blocks fit into the L2 cache, thus further increasing the achieved memory bandwidth.

Regarding the output vector `leafindexes`, its size depends on the number of trees $|\mathcal{T}|$ and on the maximum number of leaves Λ. Considering the example above: $10,000$ trees with 64 leaves per tree require 80 KB of memory, which is greater than the amount of shared memory typically available for individual thread-blocks. Since the threads of a warp perform unpredictable read-write accesses to `leafindexes`, using the global memory to manage `leafindexes` would unfortunately result into random memory accesses that cannot be coalesced; this, in turn, would impact negatively the performance. Hence, rather than storing `leafindexes` into the global memory it is far more efficient to partition the model, as already discussed in Section 3. More precisely, $\mathcal{T}$ is partitioned into multiple tree-blocks $T \subseteq \mathcal{T}$ of size $\tau = |T|$, where τ is chosen to be small enough to make the corresponding `leafindexes` fitting the amount of shared memory available per thread-block. Since the shared memory is at least one order of magnitude faster than the global memory and it does not require coalescing, it can serve up to 32 parallel requests by effectively managing potential conflicts with efficient atomic operations. As mentioned above, another motivation to adopt a small size for τ is to increase the chance that each

Algorithm 3: GPU-QuickScorer

```
1  GPU-QuickScorer (𝒟,𝒯):
2      copyHostToGPU(𝒯)
3      foreach document batch D, D ⊆ 𝒟 do
4          copyHostToGPU&Transpose(D)
5          S_D ← {s_0 = 0, ··· , s_{|D|−1} = 0}
6          foreach tree-block T, T ⊆ 𝒯 do
7              pos_pivot ← findFalseNodesGPU (𝒟, T)
8              updateScoresGPU (T, pos_pivot, S_D)
9          return S_D
```

partition fits into the L2 cache, thus minimizing the average memory access latency.

A final remark concerns where the input data, i.e., the feature vectors associated with the documents to score, are stored. These vectors are moved from the CPU to the GPU global memory in large batches of $D \subseteq \mathcal{D}$ documents. As the global memory is large, it can easily host large batches of input documents: for instance, a batch of $10,000$ documents, each having $1,000$ features, takes only about 40 MB.

The GPU algorithm. GPU-QuickScorer (QS_{GPU}), the GPU version of QuickScorer[5], is sketched in Algorithm 3. The algorithm starts by transferring the entire model from the host (CPU) memory to the GPU global memory (line 2). We note that the model is stored in a partitioned layout of disjoint tree-blocks $T \subseteq \mathcal{T}$, and it is entirely loaded into the global memory (line 2). The documents to be scored are transferred from the host memory to the GPU global memory in batches (line 4), while their initial scores are set to zero (line 5). Each batch of documents gets also transposed in parallel by the GPU; more precisely, the original layout of D is a matrix where each row stores a document vector $\mathbf{x} \in D$, while the transposed layout stores document vectors column-wise. This arranges the same feature of different documents in contiguous memory locations, thus allowing to coalesce memory accesses.

After completing the transfer, the algorithm iterates across the various tree-blocks $T \subseteq \mathcal{T}$ (line 6). Unlike the sequential QS, the identification of false nodes (findFalseNodesGPU, line 7) is executed entirely before computing `leafindexes` and the document scores (updateScoresGPU, line 8).

First phase – finding pivot positions. For each feature f_ϕ in each document $\mathbf{x} \in D$ the GPU threads in findFalseNodesGPU work in parallel to identify the positions of the so-called *pivots* in the sorted list of tuples $(\gamma, \texttt{mask}, h)$. Given a feature f_ϕ and a document $\mathbf{x} \in D$, a pivot represents the *greatest position* in the sorted list of tuples such that for all the subsequent positions the inequality $\mathbf{x}[\phi] \leq \gamma$ holds. A pivot thus separates the false nodes from the true nodes in the branching nodes that perform their tests over f_ϕ. Note that using the transposed layout for a given batch of documents D allows us to use an efficient GPU binary search algorithm[6] to search in parallel the pivots and store them in **pos_pivot**.

Finally, we highlight that the separation between the findFalseNodesGPU and the updateScoresGPU steps implies a better access to the data structures holding the tree-based model. findFalseNodesGPU needs, for all $\mathbf{x} \in D$, to access just the thresholds γ, while updateScoresGPU needs to access just `mask` and h. This limits the memory footprint of QS_{GPU}, thus favoring a better exploitation of the global memory caching.

Second phase – updating document scores. Once the positions of the pivots are available in **pos_pivot** (line 7), the algorithm proceeds to update the partial scores of the currently considered batch of documents by adding the contributions of the tree-block T, $T \subseteq \mathcal{T}$. We note that at this point we do not need to access the model thresholds γ, but just the masks `mask` and tree ids h. This second phase is realized by function updateScoresGPU (line 8, Algorithm 3) detailed in Algorithm 4.

Algorithm 4: The updateScoresGPU kernel

```
1  updateScoresGPU (T, pos_pivot, S_𝒟):
2      parallel_block foreach x ∈ 𝒟 do
3          shared leafindexes[τ], where τ = |T|
4          parallel_thread foreach t_h ∈ T do
5              leafindexes[h] ← 11...11
6          synchthreads
           ① Mask Computation Step
7          parallel_warp foreach f_φ ∈ ℱ do
8              parallel_thread foreach (γ, mask, h) ∈ 𝒩_φ^T
                 in ascending order, up to the pos_pivot[x][f_φ] -th element
               do
9                  leafindexes[h] ← (leafindexes[h] ∧_atomic
                     mask)
10         synchthreads
           ② Score Accumulation Step
11         local accScores ← 0
12         parallel_thread foreach t_h ∈ T do
13             local j ← index of the leftmost bit set in leafindexes[h]
14             local l ← h · Λ + j
15             accScores ← accScores + leafvalues[l]
16         synchthreads
           ③ Score Reduction Step
17         S_𝒟[x] ← S_𝒟[x] + BlockSumReduction(accScores)
18     return S_𝒟
```

As discussed above, we need to combine *inter-document* and *intra-document* parallelization strategies to optimize the utilization of the GPU cores. For what concerns inter-document parallelization, each document is assigned to a single block of threads – in Algorithm 4, line 2, we use the notation $\textbf{parallel}_{\textbf{block}}$ to indicate that each iteration of the loop is assigned to a different block of threads. The model $\mathcal{T}$ is partitioned to make sure that each block of threads has sufficient resources to store `leafindexes` in the shared memory (line 3). For what concerns intra-document parallelization, this is achieved within each thread-block by properly orchestrating the operations conducted within the *mask computation* and *score computation* steps.

First, the elements of `leafindexes` are initialized (line 4) – we use the keyword $\textbf{parallel}_{\textbf{thread}}$ to indicate that iterations of the loop are partitioned among the threads of the thread-block and executed in parallel. A barrier (line 6), denoted by **synchthreads**, makes sure that the initialization is completed by all threads before proceeding further.

The algorithm then performs the mask computation step, where we take advantage of the grouping of threads into warps. We explicitly assign a different feature f_ϕ to each warp of the thread-block – to this end, we note the use of the notation $\textbf{parallel}_{\textbf{warp}}$ (line 7). Going further on,

5. Source code: https://github.com/hpclab/gpu-quickscorer
6. Available in the Thrust library, v1.7.0, provided by the CUDA framework.

the construct $\textbf{parallel}_{\textbf{thread}}$ at line 8 indicates the nested parallelism within each warp, where the threads process in parallel the set of tuples, $\mathcal{N}_\phi^T$, associated with ϕ in the tree-block T. Due to the memory layout used with tuples, the accesses performed by the threads of a warp are distributed sequentially in global memory, thus yielding *coalesced* accesses. Subsequently, the retrieved masks are used to update in parallel the `leafindexes` of the corresponding trees. Since the accesses to `leafindexes` are random and potentially conflicting, atomic updates are employed to guarantee consistency (line 9). Finally, the loop ends when all the false nodes for the current document $\mathbf{x}$ and feature f_ϕ have been processed, i.e., until the position $\textbf{pos_pivot}[\mathbf{x}][f_\phi]$ in the tuple array is reached by some of the threads of the warp. Note that when this position is reached within a warp, some of its threads may result inactive, since `leafindexes` must not be modified by the nodes following the pivot: the resulting branch divergence may have a limited impact on the algorithm performance.

When all the features have been processed, the algorithm proceeds to update the document score by adding the contributions of the currently considered tree-block T. First, the vector `leafindexes` is partitioned among the threads of the thread-block (line 12) such that each thread accumulates in a private, local register the contributions of a subset of trees by identifying their exit leaves in `leafindexes`. We note that accesses to `leafvalues` cannot be coalesced and that this data structure is stored in the global memory. However, thanks to the fact that the model is partitioned into several sub-forests $T \in \mathcal{T}$, and that `leafvalues` is typically small in size, by picking up a proper τ it is possible to maximize the chance that `leafvalues` fits into the L2 cache. Finally, the threads of the block perform a block-wise *sum-reduction* over the accumulated scores, thus yielding a partial score that is used to update the overall score of the document in global memory (line 17)[7].

7 EXPERIMENTS

We conduct experiments on three public datasets that are commonly used in the scientific community to evaluate LtR solutions. The training/test datasets are composed of query-document pairs, in turn represented by multidimensional feature vectors, each labeled with relevance judgments ranging from 0 (irrelevant) to 4 (perfectly relevant). Specifically, the datasets used are: the first fold of the MSN (hereinafter MSN-1)[8], the Yahoo datasets (hereinafter Y!S1)[9], and the full Istella dataset (hereinafter Istella)[10]. Table 2 reports their main characteristics.

We use these datasets to train additive ensembles of boosting regression trees by exploiting the λ-MART [?] algorithm, which aims to optimize NDCG@10, i.e., a common measure used to evaluate ranking quality [?]. The trees of the ensembles learnt have a maximum number of leaves Λ equal to either 32 or 64. In more detail, we use the efficient implementation of λ-MART provided by QuickRank[11], an open-source LtR C++11 framework [?]. However, in the paper we focus on the efficiency at testing time of tree ensembles, and thus our results are *independent* of the specific LtR algorithm used to train them.

TABLE 2: Main characteristics of the three datasets.

	MSN-1	Y!S1	Istella
length of feature vectors	136	700	220
queries in train/validation	8,000	22,938	23,319
docs in train/validation	958,671	544,217	7,325,625
queries in test	2,000	6,983	9,799
docs in test	241,521	165,660	3,129,004
avg docs/query in test	120.7	23.72	319.31

Experimental setting for efficiency tests. For the tests we use a shared-memory NUMA multiprocessor equipped with two Intel Xeon CPU E5-2630-v3 clocked at 2.40 GHz (3.20 GHz in turbo mode), and with 192 GB RAM. Each Xeon CPU has 8 general-purpose cores, where each core has a dedicated L1/L2 cache of 32/256 KB, and a shared L3 cache of 20 MB. The system also includes an NVIDIA GTX 1080 GPU, with a global memory of 8 GB GDDR5X, a 2 MB L2 cache, 20 SMs each featuring 128 cores, a 96 KB shared memory unit (48 KB accessible by individual thread-blocks), and a dedicated 48 KB L1 cache.

All the versions of QUICKSCORER are written in C++11, and are compiled with GCC 6.3.0, plus the latest version of CUDA 8 for the GPU version. The `-O3` flag is used for the GCC compiler, while the flags `-Xptxas=-dlcm=ca` and `-gencode arch=compute_61,code=sm_61` are used for the CUDA compiler. In detail, the former flag enables global memory caching via L1 cache, while the latter generates optimized code for the GPU used in the experiments.

To measure the efficiency of the above methods, we run for 10 times the scorer on the test sets of the MSN-1, Y!S1, and Istella datasets, and compute the average per-document scoring cost. Moreover, to profile the behavior of each CPU-based QS version, we employ `perf`[12], a performance analysis tool available under Ubuntu Linux distributions. Analoguously, to profile GPU-QUICKSCORER we employ `nvperf`, a GPU performance analysis tool provided by the NVIDIA CUDA framework.

7.1 Vectorized and Multi-threaded QUICKSCORER

Table 3 reports the per-document scoring times (μs) of vQS and vQS-MT, and compares them with those achieved by the sequential QS. While vQS is a single-threaded algorithm that uses the AVX-256 SIMD extension, vQS-MT is the multi-threaded version of vQS that exploits all the 16 cores of our NUMA system. From the table we see that vQS outperforms QS with significant speedups. When dealing with trees having $\Lambda = 32$ leaves each, we observe that vQS achieves speedups over QS that range between 1.9x and 3.2x. The superiority of vQS persists also when $\Lambda = 64$. In this case we observe that the speedups achieved by vQS over QS – speedups that range between 1.2x and 1.8x – slightly decrease due to the halved number of documents that can be scored in parallel with respect to $\Lambda = 32$ (4 rather than 8). As explained in Section 4, this is a consequence

7. CUB lib v1.7.0, https://nvlabs.github.io/cub/
8. http://research.microsoft.com/en-us/projects/mslr/
9. http://learningtorankchallenge.yahoo.com
10. http://blog.istella.it/istella-learning-to-rank-dataset/
11. http://quickrank.isti.cnr.it
12. https://perf.wiki.kernel.org

TABLE 3: Per-document scoring time (μs) of QS, VQS, and VQS-MT on MSN-1, Y!S1, and Istella. Speedups of both VQS and VQS-MT over QS are between round brackets $(\cdot)$, and speedups of VQS-MT over VQS between squared brackets $[\cdot]$.

Method	Λ	Number of trees/Dataset											
		1,000						5,000					
		MSN-1		Y!S1		Istella		MSN-1		Y!S1		Istella	
QS	32	7.0	(–)	12.4	(–)	8.9	(–)	33.7	(–)	43.8	(–)	34.5	(–)
VQS	32	2.8	(2.5x)	3.9	(3.2x)	3.1	(2.9x)	17.4	(1.9x)	20.8	(2.1x)	14.3	(2.4x)
VQS-MT	32	0.2	(35.0) [14.0x]	0.4	(31.0) [9.8x]	0.3	(29.6x) [10.3x]	1.4	(24.1x) [12.4x]	1.9	(23.1x) [10.9x]	1.2	(28.8x) [11.9x]
QS	64	12.4	(–)	19.2	(–)	13.5	(–)	70.7	(–)	83.3	(–)	69.8	(–)
VQS	64	8.3	(1.5x)	10.4	(1.8x)	7.9	(1.7x)	60.8	(1.2x)	64.6	(1.3x)	46.3	(1.5x)
VQS-MT	64	0.7	(17.7x) [11.9x]	1.0	(19.2x) [10.4x]	0.7	(19.3x) [11.3x]	4.9	(14.4x) [12.4x]	10.2	(8.2x) [6.3x]	3.7	(18.9x) [12.5x]
Method	Λ	Number of trees/Dataset											
		10,000						20,000					
		MSN-1		Y!S1		Istella		MSN-1		Y!S1		Istella	
QS	32	74.6	(–)	88.7	(–)	71.4	(–)	183.7	(–)	185.1	(–)	157.2	(–)
VQS	32	39.6	(1.9x)	44.2	(2.0x)	31.1	(2.3x)	87.8	(2.1x)	88.5	(2.1x)	64.8	(2.4x)
VQS-MT	32	3.2	(23.3x) [11.5x]	4.1	(21.6x) [10.8x]	2.5	(28.6x) [12.4x]	7.3	(25.2x) [12.0x]	8.2	(22.6x) [10.8x]	7.6	(20.7x) [8.5x]
QS	64	194.8	(–)	186.9	(–)	167.4	(–)	470.5	(–)	377.2	(–)	326.1	(–)
VQS	64	146.9	(1.3x)	136.8	(1.4x)	105.9	(1.6x)	321.7	(1.5x)	274.1	(1.4x)	236.6	(1.4x)
VQS-MT	64	12.5	(15.6x) [11.8x]	14.6	(12.8x) [9.4x]	8.8	(19.0x) [12.0x]	46.2	(10.2x) [7.0x]	35.1	(10.7x) [7.8x]	26.1	(12.5x) [9.1x]

of the increase in the space required to store bitvectors `mask` and `leafindexes` (64 bits), thus limiting to only 4 the number of elements that can be stored in a single vector register of 256 bits.

We also compare VQS-MT with VQS and QS. We employ OpenMP to distribute VQS among the processing cores available within our multiprocessor. Each thread thus runs VQS and uses the AVX SIMD extensions to score bunches of documents at a time (either 8 or 4 documents, depending on Λ). The final results, reported in terms of per-document scoring time (μs), are obtained by running VQS-MT on the 16 physical cores of our NUMA multiprocessor, without the use of hyper-threading – our threads are compute-bounded, and we experimentally verified that the adoption of INTEL's hyper-threading actually reduces the overall performance. Moreover, we use the `numactl` tool to force thread allocation on the NUMA architecture. This means that a thread will use the local memory of the node where it is executed during all its life-cycle; this avoids to slow down the accesses to the memory of different nodes of the NUMA architecture. We also experiment several OpenMP loop scheduling policies, i.e., `static`, `dynamic`, `guided`, and `auto`, and report the best results obtained by using the `auto` strategy.

Table 3 shows that when $\Lambda = 32$ VQS-MT achieves speedups ranging between 20.7x and 35x over QS, while it achieves speedups ranging between 6.3x and 14x over VQS. Note that the best possible speedup of VQS-MT over VQS is 16x (linear speedup), since our multiprocessor has 16 cores. We are able to approach the best possible speedup only when considering the smallest MSN dataset (1,000 trees), as this allows to better fit the various levels of dedicated/shared cache. When considering $\Lambda = 64$, VQS-MT achieves speedups over QS that range between 8.2x and 19.3x: again, we notice a performance degradation with respect to the $\Lambda = 32$ case caused by the reduced SIMD parallelism within each thread. Finally, when $\Lambda = 64$ the speedups achieved by VQS-MT over VQS are between 6.3x and 12.5x: this still represents a good result, but it is worse than the result achieved for the $\Lambda = 32$ case. The performance degradation is mainly due to the increased size of the tree-based models (the size of all bitwise masks are doubled), which in turn increases the competition among threads over the L3 cache of each multi-core CPU: in fact, the worst speedups are obtained when considering very large ensembles (20,000 trees).

Instruction level analysis. We use the `perf` tool to measure the total number of instructions, number of branches, number of branch mis-predictions, L3 cache references, and L3 cache misses for the different scorers, running on a single core of the Intel Xeon CPU by exploiting its SIMD extension. In these tests we compare QS against VQS over ISTELLA, the largest and most challenging dataset. The experiments conducted on the other datasets are not reported, as they exhibit similar results. Table 4 reports the results achieved with all measurements normalized per-document and per-tree. It is worth specifying that the figures for L3 cache references reported in the table account for memory accesses that cause a miss in any of the previous levels of cache, while L3 cache misses account also for the percentage of L3 cache references that miss in L3.

Interestingly, the analysis reveals that the use of the AVX-256 instruction set causes a significant decrease in the average number of instructions needed to score a single document. This reduction justifies the speedups achieved by VQS. In terms of branch figures, VQS shows lower mis-prediction rates than QS. The total number of per-tree per-document branches is also lower, demonstrating that the chosen parallelism represents a good strategy to increase the throughput of QS. The same results are achieved for the cache utilization. As in the preceding case, L3 cache misses and references are always lower than the ones achieved by QS, thus revealing a more effective use of the cache.

The instruction level analysis of the multi-threaded implementation of QUICKSCORER (VQS-MT) shows that it inherits the same figures from VQS. The use of the OpenMP library to parallelize VQS does not incur in computational overhead, as the instruction count is the same as VQS – the same considerations hold for the total number of branches and branch mis-predictions. In terms of L3 cache misses, the number of misses increases for VQS-MT when the model size increases – this is again caused by an increased competition in the usage of the L3 cache. Finally, we argue that the high number of threads working concurrently affects negatively the temporal and spatial locality, thus leading to a higher number of cache misses.

7.2 GPU-based QUICKSCORER

GPUs have constraints that impact on the design and tuning of algorithms, with different GPU models possessing quantitatively different constraints. Since the constraints are similar in nature across different GPUs, without loss of generality we focus on the GPU used in our experiments, a NVIDIA GTX 1080. Table 5 reports its constraints. In the following we discuss the possible impact of these constraints on the design of QS_{GPU} – specifically, *warp efficiency*, *number of resident thread-blocks and occupancy*, and *L2 cache size impact*.

Warp efficiency. Within the UPDATESCORESGPU kernel, which represents the time-dominant component of the algorithm, QS_{GPU} exploits the shared memory of each SM to store `leafindexes`, where each bitvector is of Λ bits. Therefore, let $\Sigma_\tau = \tau \cdot \Lambda/8$ be the size (in bytes) of the shared memory footprint of a model of τ trees for each thread-block. Since a thread-block is constrained to access up to 48 KB of shared memory, then $\Sigma_\tau \leq 48$ KB.

Example. Given a large model composed of $20,000$ trees with $\Lambda = 32$, the size of the `leafindexes` data structure would be $\Sigma_\tau = 20,000 \cdot 32/8 = 80,000$ bytes, thus much larger than the limit of 48 KB. Therefore, the maximum number of trees of a single partition of the ensemble is $\tau = 12,288$ for $\Lambda = 32$, and $\tau = 6,144$ for $\Lambda = 64$.

TABLE 4: Per-tree per-document low-level statistics on Istella with 64-leaves λ-MART models.

Method	Number of Trees				
	1,000	5,000	10,000	15,000	20,000
Instruction Count					
QS	67	70	79	81	73
VQS	**57**	61	**66**	**65**	**57**
VQS-MT	**57**	**60**	**66**	**65**	**57**
Num. branch mis-predictions (above) Num. branches (below)					
QS	0.139	0.036	0.022	0.013	0.010
	7.86	7.44	8.34	8.62	7.64
VQS	0.03	**0.004**	**0.002**	**0.002**	**0.001**
	4.47	4.81	**5.22**	**5.17**	4.56
VQS-MT	0.02	**0.004**	0.003	**0.002**	**0.001**
	4.45	4.80	**5.22**	**5.17**	**4.55**
L3 cache misses (above) L3 cache references (below)					
QS	0.005	**0.001**	**0.001**	**0.002**	**0.004**
	2.0	1.47	1.57	1.75	1.94
VQS	**0.004**	0.003	0.025	0.004	0.026
	0.51	**1.04**	**1.31**	1.86	**1.38**
VQS-MT	0.005	0.004	0.190	0.085	0.151
	0.47	1.14	1.59	**1.62**	1.64

TABLE 5: NVIDIA GTX 1080 constraints.

NVIDIA GTX 1080 Feature	Limit
Threads (warps) per SM	2,048 (64)
Threads (warps) per thread-block	1,024 (32)
Thread-blocks per SM	32
Shared memory per SM	96 KB
Shared memory per thread-block	48 KB
Registers per thread-block	65 K
Warp schedulers per SM	4
L2 cache size	2 MB

As discussed previously, this does not represent a major issue for QS_{GPU}, as any model can be evaluated in partitions of any custom size τ. However, splitting a model into partitions introduces overhead, as part of the threads constituting a warp become inactive when reaching a pivot that separates *true* nodes from *false* ones in a given partition of the model. This inefficiency is measured by *warp efficiency*, i.e., the average fraction of active threads per executed warp. We observe that each partition made of τ trees has its own set of pivots: therefore, increasing the partitions of the model increases proportionally the number of pivots, thus harming warp efficiency. Even if this phenomenon is more evident with large feature sets, or when the number of false nodes is very small, in general we have that the larger the number of partitions (or, equivalently, the smaller τ), the smaller the *warp efficiency*.

Number of resident thread-blocks and occupancy. τ also determines the maximum *number of resident thread-blocks* that can run concurrently on the same SM. Since each SM is equipped with only 96 KB of shared memory, this limits the number of resident thread-blocks to a maximum of $\beta_\tau = \lfloor 96 \text{ KB}/\Sigma_\tau \rfloor$.

While satisfying the above constraint, it is also important to maximize *occupancy*, i.e., the average ratio between the number of *active warps* per cycle per SM and the *maximum number of active warps* that are supported per SM (64 in our GPU, for a total of 2,048 threads). Generally, by maximizing occupancy we increase the chance for SM schedulers to hide/tolerate warp stalls caused by global memory accesses.

Note that we could maximize occupancy by simply increasing the number of threads per thread-block. However, this strategy may end up reducing the number of per-SM resident thread-blocks, due to the constraint on the total number of threads per SM (max 2,048). Instead, along with occupancy it is also important to maximize the number of resident thread-blocks, as this allows to mask stalled warps of a block with *eligible* warps (i.e., warps that are ready to issue their next instruction) of other thread-blocks. In fact, all the warps of a thread-block can be stalled due to a block-level barrier.

Given a specific τ, the policy we adopt to maximize occupancy, along with the number of per-SM resident thread-blocks, is to choose the *minimum* number of threads per thread-block (*n_threads*) that allows to run *enough* concurrent thread-blocks per SM with *full occupancy*. More

formally:

$$\begin{aligned} \textbf{min} \quad & n_threads = 2^n \\ \textbf{subject to} \quad & 5 \leq n \leq 10; \\ & 2 \leq \tfrac{2{,}048}{n_threads} \leq \beta_\tau. \end{aligned}$$

Note that the first constraint forces $n_threads$ to be a multiple of 32 (warp size) and a divisor of 1,024 (max threads per thread-block). Since we minimize $n_threads$, this choice actually maximizes the number of per-SM resident thread-blocks (i.e, 2,048/$n_threads$), provided that this number is not greater than β_τ.

Note also that the number of resident thread-blocks must be at least 2: indeed, the maximum shared memory allocated to each thread-block is 48 KB – exactly half of the total shared memory available – while the maximum $n_threads$ per thread-block is 1,024, exactly half of the maximum number of threads per SM that guarantees full occupancy (see Table 5).

We validated the above policy by conducting a grid search over τ and $n_threads$, with ensembles featuring 20,000 trees and $\Lambda = \{32, 64\}$ leaves per tree (results are omitted for brevity).

> *Example*. Let τ = 4,000 and Λ = 32. A single thread-block requires $\Sigma_\tau \approx$ 16 KB, which allows to have at maximum $\beta_\tau = 96/16 = 6$ concurrent thread-blocks per SM. If we use 6 thread-blocks per SM, we can have at most $n_threads = 320$ threads per thread-block, due to the limit of 2,048 threads per SM, thus yielding a total of 1,920 threads: this implies a sub-optimal occupancy, i.e., $1,920/2,048 = 0.9375 < 1$. We rather use $n_threads = 2^9 = 512$, which yields full occupancy and 2,048/512 = $4 \leq \beta_\tau$ thread-blocks per SM.

Another parameter is n_total_blocks, i.e., the overall number of allocated thread-blocks to be dinamically scheduled on the various SMs. In general we have that the greater n_total_blocks, the finer the granularity of aggregated task assigned to each thread-block, hence guaranteeing a better load balancing of the workload distributed over the SMs. Specifically, we need that $n_total_blocks \gg m \times \beta_\tau$, where m is the number of GPU's SMs (m = 20 in our GPU), and $m \times \beta_\tau$ is the maximum number of resident blocks that can run concurrently on all the SMs of a GPU. As QS_{GPU} typically needs to score huge amounts of query-document pairs, our problem setting allows us to easily set n_total_blocks to a sufficiently high value – value much larger than $m \times \beta_\tau$ – to achieve optimal performance, while ensuring at the same time that each thread-block has an adequate workload.

L2 cache size impact. The L2 cache memory size (2 MB) has a strong influence on the tuning of τ. The cache size also impacts on the access time to the remaining data structures stored in the GPU global memory.

More precisely, for each internal node QS_{GPU} uses $\Lambda/8+2$ bytes to store, respectively, the node's bitvector `mask` and tree ID h, while it uses 8 bytes (a double) for each leaf score in `leafvalues`. In general, note that the cache size imposes a stricter upper bound than the shared memory constraint.

> *Example*. QS_{GPU} requires $(\Lambda/8 + 2) \cdot (\Lambda - 1)$ plus $8 \cdot \Lambda$ bytes for each tree. Given a model with $20,000$ trees, the 2 MB constraint is already violated when $\tau = 5,000$ with $\Lambda = 32$, or $\tau = 2,000$ with $\Lambda = 64$.

To exploit the computational power of a GPU we need to pursue two contrasting goals: *maximizing* the *warp efficiency* by using a large τ, and *maximizing* the *hit rate* of the L2 cache by using a sufficiently small τ. We argue that these goals can be achieved by using a value of τ that is sufficiently close to the size of the L2 cache, while the number of threads per thread-block, $n_threads$, and the number of per-SM resident thread-blocks can be statically determined as shown above.

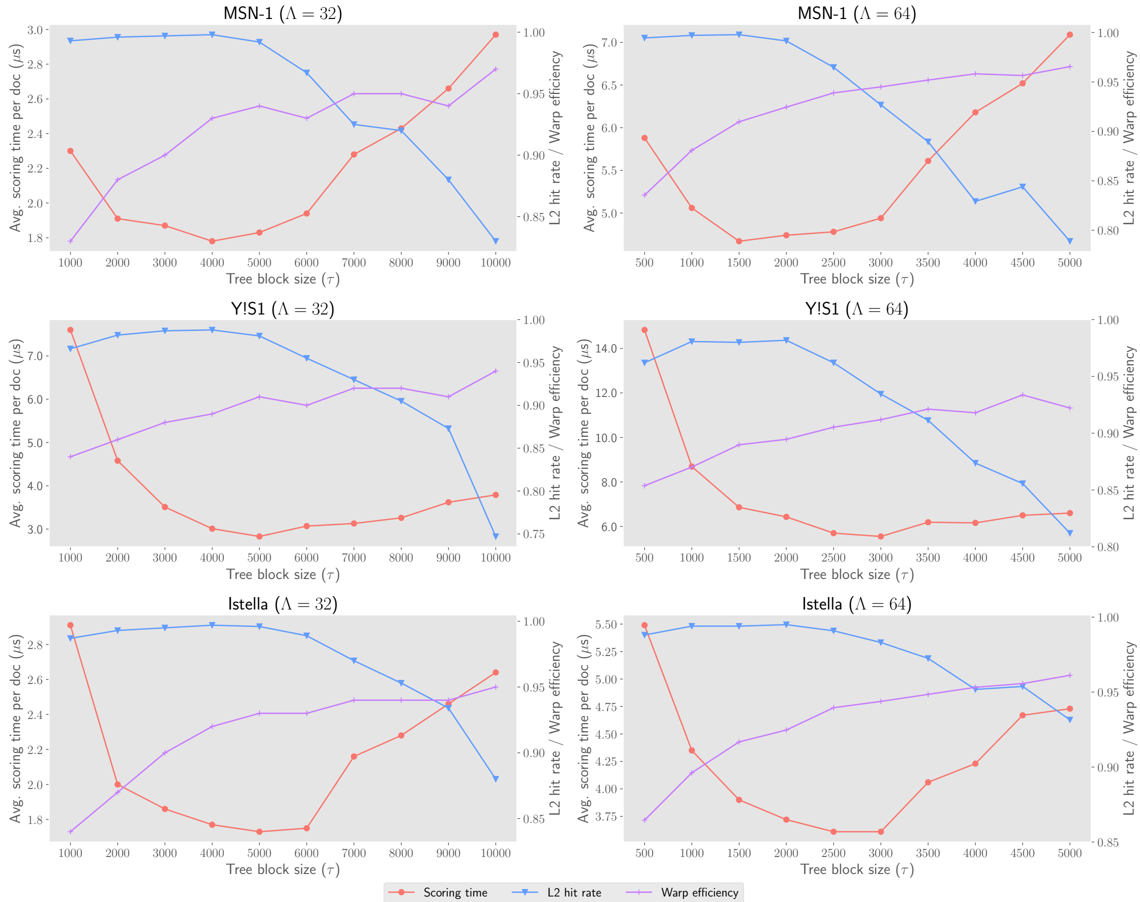
In the batch of experiments that follows we validate our analytic performance model and the choice of τ for optimal performance. We vary τ in the $[1,000 - 10,000]$ range, and for each value of τ we set the number of threads per thread-block, $n_threads$, by means of the previously illustrated policy. Also, the number of thread-blocks, n_total_blocks, is set to the highest possible value, $64K - 1$, as this ensures an optimal balancing of the workload. The datasets feature $|\mathcal{T}| = 20,000$ and $\Lambda = \{32, 64\}$. We also use `nvprof` to collect two profiling metrics, i.e., the L2 *cache hit ratio* and the *warp efficiency*. Finally, we report that the trends observed in the plots can be reproduced with different values of $|\mathcal{T}|$ and Λ (we omit the results for brevity). Figure 2 presents the results of the analysis.

We first notice that the L2 hit rate remains close to 1 until $\tau \leq 5,000$ ($\Lambda = 32$) and $\tau \leq 2,000$ ($\Lambda = 64$); this is expected, considering the amount of L2 cache available (2 MB) and the space required by each partition of the model. Secondly, we observe that warp efficiency increases as τ increases (this reduces the number of tree-blocks): this is again expected, as having less partitions implies less pivots, which in turn reduces the chances that part of the threads making up a warp become inactive when reaching some pivot of some feature.

Overall, increasing τ improves the scoring time (this is mainly due to the improved warp efficiency) until the L2 hit rate remains close to 1, thus indicating the existence of a tradeoff. When the cache performance starts to degrade ($\tau \geq 6,000$ with $\Lambda = 32$, and $\tau \geq 3,000$ with $\Lambda = 64$), the scoring time starts to increase noticeably.

Table 6 shows the per-document scoring time in μs and the speedup obtained, for different sizes $|\mathcal{T}|$ of the model and different numbers of leaves Λ. When $|\mathcal{T}|$ becomes large, QS_{GPU} partitions the model in different blocks of size τ, and adopts a suitable number of threads per thread-block to ensure the best occupancy of each SM in the GPU. Details about the two parameters that are crucial for QS_{GPU}'s performance are reported in the *Threads per block* and τ columns. QS_{GPU} achieves consistent speedups over the sequential QS – up to 102.6x, 65.6x, and 90.9x on MSN-1, Y!S1, and Istella, respectively. In general, we observe that QS_{GPU} dominates the implementations previously discussed in all the settings considered, and achieves the best results when the size of the ensembles becomes large in terms of number of trees and leaves per tree – in fact, the best results are always achieved when $|\mathcal{T}| = 20,000$ and $\Lambda = 64$, as the larger computational workload to score each query-document pair favours the massive parallelism of GPUs.

Fig. 2: Performance analysis of QS_{GPU} by varying the size of tree-blocks τ. $\Lambda = 32$ (left) and 64 (right), $|T| = 20,000$, variable number of threads per thread-block, and fixed number of thread-blocks ($64K - 1$).



8 Related Work

The Information Retrieval community has recently investigated possible strategies to reduce the scoring time of the most effective LtR rankers based on ensembles of regression trees. These strategies can be roughly divided into two groups: tree removal strategies and algorithm optimization.

Tree removal strategies focus on boosting the scoring time by limiting the number of trees processed, trading off effectiveness for efficiency. Cambazoglu *et al.* [?] proposed to early terminate the trees traversal, on a single query-document basis, as the score contributions of the remaining trees become low. Lucchese *et al.* [?] proposed to statically remove low-contributing trees and re-train the weights according to a given effectiveness measure.

Algorithmic optimization, although do not change the time complexity, aims to better exploit the underlying CPU architecture, in particular instruction-level parallelism, data-level parallelism, and memory hierarchies [?]. A first proposal toward efficient traversal of binary regression trees was the VPRED algorithm [?]. VPRED stores such trees as binary heaps implemented as linear arrays, and substitutes the branches, needed to select the traversal path of a tree in a traditional code, with a sequence of instructions that use the results of each Boolean test to identify the index of the next heap cell to visit. Since the directions of branches employed by a traditional tree traversal code are quite unpredictable, this optimization tries to remove the problem at the root, by completely removing conditional statements. However, this technique, aiming to transform control dependencies into data dependencies, is not enough to fully exploit the multiple pipelines of a processor. To achieve a better exploitation of the pipelines of the CPU VPRED scores multiple query-document pairs on the same tree thus allowing the processor to identify and issue independent instructions in parallel working on distinct pairs. The memory footprint of VPRED is not so large, since it accesses a tree of the ensemble at a time to score groups of documents. Another investigation toward efficient traversal of forests of regression trees relies in the definition of a blocking strategy allowing a better temporal and spatial cache location of the scoring process. Tang *et al.* [?] propose a cache-conscious layout for ensemble models able to achieve a up to 50% improvement over VPRED.

The QuickScorer (QS) algorithm [?], [?], which restructures the data layout and the processing of an ensemble of regression trees to leverage modern memory hierarchies and reduce branch prediction errors to limit control hazards, resulted up to 6.6x faster than VPRED.

Literature about GPU-based algorithms that score/classify with forests of trees is limited to classifiers based on small ensembles of random trees and have a very narrow scope. Schulz *et al.* [?] exploited the characteristics of a given problem, i.e., image labeling, to overcome issues related to the structural irregularity of random trees. Van Essen *et al.* [?] proposed a GPU-based approach that use Compact

TABLE 6: Per-document scoring time in μs of QS and QS_{GPU} on MSN-1, Y!S1, and Istella datasets. Speedups of QS_{GPU} vs. QS are reported between parentheses.

$\lvert\mathcal{T}\rvert$	Λ	Method	Threads per block	τ	Time	
			MSN-1			
1,000	32	QS	–	–	7.0	(–)
		QS_{GPU}	128	1,000	0.19	(36.8x)
	64	QS	–	–	12.4	(–)
		QS_{GPU}	256	1,000	0.25	(49.6x)
5,000	32	QS	–	–	33.7	(–)
		QS_{GPU}	512	5,000	0.44	(76.6x)
	64	QS	–	–	70.7	(–)
		QS_{GPU}	256	1,500	1.08	(65.5x)
10,000	32	QS	–	–	74.6	(–)
		QS_{GPU}	512	5,000	0.86	(86.7x)
	64	QS	–	–	194.8	(–)
		QS_{GPU}	256	1,500	2.29	(85.1x)
20,000	32	QS	–	–	183.7	(–)
		QS_{GPU}	512	4,000	1.79	(102.6x)
	64	QS	–	–	470.5	(–)
		QS_{GPU}	256	1,500	4.67	(100.8x)
			Y!S1			
1,000	32	QS	–	–	12.4	(–)
		QS_{GPU}	128	1,000	0.85	(14.6x)
	64	QS	–	–	19.2	(–)
		QS_{GPU}	256	1,000	0.75	(25.6x)
5,000	32	QS	–	–	43.8	(–)
		QS_{GPU}	512	5,000	0.94	(46.6x)
	64	QS	–	–	83.3	(–)
		QS_{GPU}	512	3,000	1.78	(46.8x)
10,000	32	QS	–	–	88.7	(–)
		QS_{GPU}	512	5,000	1.56	(56.9x)
	64	QS	–	–	186.9	(–)
		QS_{GPU}	512	2,000	3.45	(54.2x)
20,000	32	QS	–	–	185.1	(–)
		QS_{GPU}	512	5,000	2.82	(65.6x)
	64	QS	–	–	377.2	(–)
		QS_{GPU}	512	3,000	5.55	(68x)
			Istella			
1,000	32	QS	–	–	8.9	(-)
		QS_{GPU}	128	1,000	0.28	(31.2x)
	64	QS	–	–	13.5	(-)
		QS_{GPU}	256	1,000	0.37	(36.5x)
5,000	32	QS	–	–	34.5	(-)
		QS_{GPU}	512	5,000	0.50	(69x)
	64	QS	–	–	69.8	(-)
		QS_{GPU}	512	3,000	1.03	(67.8x)
10,000	32	QS	–	–	71.4	(-)
		QS_{GPU}	512	5,000	0.96	(74.4x)
	64	QS	–	–	167.4	(-)
		QS_{GPU}	512	3,000	2.07	(80.8x)
20,000	32	QS	–	–	157.2	(-)
		QS_{GPU}	512	5,000	1.73	(90.9x)
	64	QS	–	–	326.1	(-)
		QS_{GPU}	512	3,000	3.63	(89.8x)

Random Forests (CRFs), i.e. forests of binary decision trees having fixed depth, to control the structure and the size of trees, thus fitting well the architectural characteristics of the GPUs. The approach adopts design choices that make it similar to VPRED [**?**]: it stores decision trees as binary heaps and visits each tree by employing a less refined traversal strategy than the one used in VPRED – more precisely, loops and conditional statements are not completely eliminated. Given a CRF and a set of documents to score, the GPU-based approach first partitions the CRF into sub-forests that can fit into the texture cache of a streaming multiprocessor, and assign each sub-forest to a multiprocessor. Each document is then assigned to a single GPU thread, which accumulates the scores of the sub-forest assigned to the multiprocessor executing the thread. Partial scores are finally reduced by the host (CPU). Due to the data structures and the traversal strategy used, the approach proposed in [**?**] suffers of the same limitations that characterize VPRED when compared to more recent approaches.

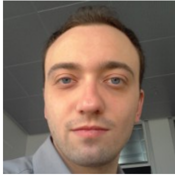
9 Conclusion

In this paper we presented and evaluated several strategies to parallelize the traversal of large ensembles of decision trees. The motivation of this this research is the need of deploying large tree forests in real large-scale settings, and using such complex ML models to process each incoming item within a small time budget. Large ensembles of decision trees are adopted in different ML scenarios such as Web or product search, social media ranking or recommendation, on-line advertisement, classification/regression tasks, etc. We focused the proposed parallelization strategies within the LtR framework, where a relevance ordering of documents w.r.t. a user query is induced by the scores assigned to the documents. The proposed solutions are seamlessy applicable to any of the discussed ML scenarios and boost the efficiency of decision trees processing.

Our proposed strategies take advantage of the algorithmic framework introduced by QuickScorer, the state-of-the-art algorithm in the literature, to leverage different types of parallelism available in modern CPUs and GPUs. We compared the proposed parallel solutions with the original sequential version of QuickScorer. The CPU-based parallelization strategies, namely vQS (SIMD) and vQS-MT (multi-threading + SIMD), achieved large speedups over QuickScorer: more precisely, vQS obtained speedups up to 3.2x (32 leaves per tree) and 1.8x (64 leaves per tree), while vQS-MT achieved speedups up to 35.0x (32 leaves per tree) and 19.3x (64 leaves per tree) on a 16 cores machine. The performance gains originated from the exploitation of different types of parallelism coupled with an efficient use of CPU resources, as observed from the low-level monitoring of instruction counts, branch mispredictions, and L3 cache-miss rates. The main advantage of our parallelization strategies is to provide increased throughput, which in turn allows to better satisfy quality-of-service constraints. In fact, while the use of larger ensembles improves the accuracy of machine-learned models, the reduced scoring times allow to process such models within smaller time budgets.

Our GPU-based parallelization strategy, namely GPU-QuickScorer, provides the lowest document scoring times in all the tested settings and should be the solution of choice when absolute performance maximization is more important than the increase of the hardware cost for GPUs equipment. Specifically, GPU-QuickScorer achieved speedups up to 102.6x (32 leaves per tree) and 100.8x (64 leaves per tree) over QuickScorer on a NVIDIA GTX 1080 GPU. These impressive performance gains are the result of a careful design of the data layout and of the orchestration of the accesses over the GPU-QuickScorer data structures of the massive number of parallel threads run by modern GPUs. As future work we plan to investigate if these characteristics of our solution can be exploited by a Field-Programmable Gate Arrays (FPGAs) implementation.

Acknowledgements. This paper is partially supported by the BIGDATAGRAPES (grant agreement Nº780751) project that received funding from the European Union's Horizon 2020 research and innovation programme under the Information and Communication Technologies programme.



Francesco Lettich (https://www.linkedin.com/in/francesco-lettich-43741a7) is a researcher at the Federal University of Ceará (Brazil). He received his Ph.D. from the Universitá Ca' Foscari di Venezia in 2015. His main research interests focus on general purpose computing on GPU and analysis of mobility data.

Claudio Lucchese (http://hpc.isti.cnr.it/~claudio) is an associate professor at Ca' Foscari University of Venice. He received his MS.c. and Ph.D. from Ca' Foscari University in 2003 and 2008, respectively. His main research activities are in the areas of data mining techniques for information retrieval and large-scale data processing. He published more than 100 papers on these topics in international journals and conferences.

Franco Maria Nardini (http://hpc.isti.cnr.it/~nardini) is a researcher at National Research Council of Italy. His research interests focus on Web Information Retrieval, Data Mining, and Machine Learning. He authored more than 50 papers in peer reviewed international journal and conferences and received the Best Paper Awards at ACM SIGIR 2015.

Salvatore Orlando (http://www.dais.unive.it/~orlando) – MSc (1985) and PhD (1991) in Computer Science, University of Pisa – is a full professor at Ca' Foscari University of Venice. His research interests include data and web mining, information retrieval, parallel/distributed systems. He published over 140 papers in journals and conference proceedings on these topics.

Raffaele Perego (http://hpc.isti.cnr.it/~raffaele) is a senior researcher at ISTI-CNR, where he leads the HPC Lab (http://hpc.isti.cnr.it/). His main research interests include high performance computing, Web information retrieval, and data mining. He coauthored more than 140 papers on these topics published in journals and proceedings of international conferences.

Nicola Tonellotto (http://hpc.isti.cnr.it/~khast) is a researcher at National Research Council of Italy. He received his Ph.D. from the Information Engineering Department of the University of Pisa in 2008. His main research interests include Cloud and Web information retrieval. He published over 50 papers in journals and proceedings of international conferences. He received the Best Paper Award at ACM SIGIR in 2015.

Rossano Venturini (http://pages.di.unipi.it/rossano) is a researcher at Computer Science Department, University of Pisa. He received his Ph.D. from University of Pisa in 2010. His research interests are mainly focused on the design and the analysis of algorithms and data structures with focus on indexing and searching large textual collections. He received two Best Paper Awards at ACM SIGIR in 2014 and 2015.