Noname manuscript No.
(will be inserted by the editor)

Parallelizing the Traversal of Large Ensembles of Decision Trees

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

Received: date / Accepted: date

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. These models are computationally demanding: indeed, 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 traversing large ensembles of regression trees, that tip the balance towards machine-learnt models that are, at the same time, effective, fast, and scalable.

F. Lettich
Univ. Federal do Ceará, Brazil and Univ. of Venice, Italy
E-mail: francesco.lettich@gmail.com

C. Lucchese
Ca' Foscari University of Venice, Italy
E-mail: claudio.lucchese@unive.it

F. M. Nardini
ISTI–CNR, Italy – E-mail: francomaria.nardini@isti.cnr.it

S. Orlando
Ca' Foscari University of Venice, Italy
E-mail: orlando@unive.it

R. Perego
ISTI–CNR, Italy – E-mail: raffaele.perego@isti.cnr.it

N. Tonellotto
ISTI–CNR, Italy – E-mail: nicola.tonellotto@isti.cnr.it

R. Venturini
University of Pisa, Italy
E-mail: rossano.venturini@unipi.it

1 Introduction

Recent advances in Machine Learning (ML) allow to model phenomena and solve problems that were previously too complex for computers to handle. This radical change has opened new horizons for improving quality, speed, and accuracy of actionable solutions for complex problems in many diverse application domains. However, the complexity of machine-learnt models and their widespread adoption require novel algorithmic solutions aimed at rendering fast and scalable both their training and use. Unlike the main stream of research on efficiency, which aims at making faster the off-line training process, this paper focuses on speeding-up the online application of a particular kind of learnt ML models, i.e., those based on additive *ensembles of decision trees.* These models, 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! Learning to Rank Challange [1], which fostered the development of several Learning-to-Rank (LtR) algorithms aimed at addressing the fundamental problem of ranking items according to their relevance to queries [10,11]. The most robust and effective ranking models turned out to be those based on ensembles of regression trees learnt with the GBRT [12] and λ-MART [13] algorithms. Indeed, within this challenge, all top competitors leveraged decision trees and ensemble methods, and the winner deployed a total of 24,000 regression trees in an ensemble model. 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 [2,?]. 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 [4]. Finally, ensembles of regression trees are part of production-level pipelines for ads clickthrough rate predictions [3], and are the most common choice in solutions for ML competitions, such as Kaggle[1].

In large-scale production systems, complex ensembles with thousands of trees have to be deployed to achieve high quality results meeting user satisfaction. In addition, due to the incoming rate of requests and quality-of-service expectations, the traversal of such tree ensembles has to be fast, and must complete within small time budgets. All these requirements are very challenging to fulfill, and this paper exactly focuses on techniques for speeding-up the traversal of large tree ensemble by exploiting parallelism.

Without loss of generality, in the following we concentrate on the WSE scenario, and adopt the LtR terminology to discuss the state-of-the-art and investigate parallel algorithms to traverse large tree-based ensemble models. 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 trees, given an item represented as a feature vector. Large scale WSEs commonly exploit LtR solutions within a multi-stage ranking architecture [5,6,7,8, 9] to realize their top-k retrieval systems. 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. Consider now a stage of this architecture that ranks items by applying an additive ensemble of regression trees. Given the input instance represented by a feature vector $\mathbf{x}$, the ranking model predicts a relevance score $s(\mathbf{x})$ (observed value) that is eventually used to rank the set of candidate documents. The internal (or branching) nodes in all the trees of the ensemble are associated with a Boolean test over the value of a specific feature. Each leaf node stores instead the tree prediction, representing the potential contribution of the tree to the final document score. The scoring of $\mathbf{x}$ requires the traversal of all the trees in the ensemble to devise all the tree prediction and it is computed as their *weighted sum*. Typical figures [1] regarding the complexity of a tree-based ranker deployed on the last stage of a WSE ranking pipeline are reported in Table 1.

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

Table 1: Typical complexity of WSE document ranking with LtR tree ensembles.

Dimension	Number
Trees	$1,000 - 20,000$
Leaves per tree	$4 - 64$
Documents scored per query	$3,000 - 10,000$
Features per query-document pair	$100 - 1,000$

In this paper we investigate multi/many-core parallelization strategies making the traversal of large ensemble of trees fast and scalable. In particular, we investigate diverse strategies to parallelize QuickScorer (QS), the state-of-the-art algorithm for the traversal of ensembles of binary decision trees [14,17]. The goal is to allow the deployments of large and complex ML models, able to produce very accurate and precise ranking of a set of documents within a small time budget. Orthogonally, when the desired level of accuracy is already granted by a given model, we can rely on a parallel scorer to reduce latency and increase throughput. We designed several parallel versions of QS, aiming to assess the various opportunities offered by modern architectures. Specifically, we deal with: (i) processor instructions set extensions to vectorize code, (ii) multi-core architecture for shared memory multi-threading, and (iii) many-core architecture of modern graphic cards (GPUs) exploiting massive data parallelism.

We report on extensive experiments conducted on three publicly available LtR datasets, namely the MSN, the Yahoo LETOR challenge, and the Istella datasets. In the experiments we investigate strengths and limitations of the proposed parallel algorithms. Although the tested datasets that are 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, social media filtering and ranking, online advertisement, classification or regression tasks on big data. Moreover, even if the learnt models used in the experiments are additive ensembles of regression trees, the same techniques discussed in the paper can be exploited for other tree-based ensembles, like random forests used for classification, regression, or other tasks.

The rest of this paper is structured as follows. Section 2 presents the proposals discussed in the literature for speeding-up the scoring time of LtR rankers based on ensembles of regression trees. We also give some background on the features of modern CPUs relevant for discussing the state of the art. The last part of the related work section presents in a concise way our algorithm QS, whose parallelization is discussed in the following. Section 3 defines the possible strategies to parallelize QS. Then, in Section 4 we present a comprehensive analysis regarding the use of single-instruction multiple-data (SIMD) extended instruction sets to vectorize QS. Section 5 deals with multi-core shared memory NUMA architecture, and thus discusses how to combine SIMD and multi-threading to parallelize QS. In Section 6, we also explore the possible strategies to exploit many-core GPUs to speed-up QS by massive parallelism. Section 7 reports on the results of our comprehensive experimental evaluation. Finally, we conclude our investigation in Section 8.

2 Related work

The IR community has recently investigated possible strategies to reduce the scoring time of the most effective LtR rankers based on ensembles of regression trees [6,7,16,14,17]. These strategies can be roughly divided into two orthogonal groups: tree removal and algorithm optimizations.

Tree removal strategies focus on boosting the scoring time by limiting the number of trees processed, trading off effectiveness for efficiency. Cambazoglu *et al.* [7] propose to early terminate the trees traversal, on a single query-document basis, as soon as it becomes clear that the score contributions of the remaining trees will be low, while Lucchese *et al.* [18] propose to statically remove a subset of low-contributing trees in the ensemble and re-train the weights of the remaining ones according to a given effectiveness measure.

Algorithmic optimizations, although do not change the time complexity, aim to better exploit the underlying CPU architecture, in particular instruction-level parallelism (ILP), data-level parallelism (DLP) and memory hierarchies [19]. Modern architectures mainly realize ILP through *pipelines*, where a pipeline is a chain of functional stages able to execute in parallel distinct parts of different instructions. Indeed, each computational core (processor) of a modern CPU includes multiple pipelines and adopts a *superscalar* design, i.e., an architecture where multiple instructions can be simultaneously dispatched to the various processor's pipelines. Note that although the instructions of a sequential program should be issued *in-order*, thus following the sequential flow of control, a superscalar processor speculatively inspect a sequential program to detect independent instructions to execute in parallel, to fully exploit the ILP capability of the processor. This speculative technique dynamically issues and executes instructions *out-of-order*, thus modifying the original control flow of a sequential program. The key factors to fully exploit such processors is thus to have sequential programs that include many *independent instructions* that can executed in any order and thus in parallel. A *data flow dependency* between a pair of consecutive instructions prevents the parallel dispatching of these instructions. In addition, another source of inefficiency for superscalar processors is the presence of *control dependencies*, caused by branch instructions that determine the next instruction to execute. Modern processors exploit effective branch predictors, which guess the branch directions (taken/not-taken) to prefetch and execute the correct next instruction. For example, branches used to implement loops are easily predictable, as a loop body is likely re-executed many times except for exit condition. Finally, the current technological trend depicts a scenario in which we observe a continuous widening of the processor-memory gap, due a faster pace of increase in processor speed than in RAM memory latency. Modern CPUs try to fill this gap by adopting very complex cache memory hierarchies. To take advantage of the multiple levels of caches present in such complex memory hierarchies, the layout of data structures and the associated access pattern have to be designed so as to exploit spatial and temporal locality, by also reducing cache misses.

Coming back to the issues of exploiting modern CPUs to accelerate scoring with forests of binary regression trees, the algorithm VPRED [6] 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 low predictable, 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 maintain busy the multiple pipelines of a processor. Thus, VPRED scores multiple query-document pairs on the same tree, and this allows a processor to identify and issue in parallel independent instructions 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. Finally, Tang *et al.* [16] propose a cache-conscious layout scheme for trees data, with up to 50% improvement over VPRED.

The QuickScorer (QS) algorithm [14,17] restructures the data layout and the processing of regression trees to leverage modern memory hierarchies and reduce the branch prediction errors to limit the control hazards. Experimental results show that QS is up to 6.6× faster than VPRED. 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. Finally, data structures and data access are re-organized accordingly to reduce cache misses.

As our proposals exactly parallelize QuickScorer, we will use the next subsection to describe this algorithm in details.

Algorithm 1: QuickScorer

```
QuickScorer(x,$\mathcal{T}$):
    foreach $T_h \in \mathcal{T}$ do
        leafindexes[$h$] ← 11...11
    foreach $f_\phi \in \mathcal{F}$ do                    // Mask Computation
        foreach $(\gamma, \texttt{mask}, h) \in \mathcal{N}_\phi$ in asc. order of $\gamma$ do
            if $\mathbf{x}[\phi] > \gamma$ then
                leafindexes[$h$] ← leafindexes[$h$] ∧ mask
            else
                break
    $score \leftarrow 0$
    foreach $T_h \in \mathcal{T}$ do                    // Score Computation
        $j$ ← index of leftmost bit set to 1 of leafindexes[$h$]
        $l \leftarrow h \cdot \Lambda + j$
        $score \leftarrow score$ + leafvalues[$l$]
    return $score$
```

2.1 QuickScorer

Let us denote with $\mathcal{T} = \{T_0, T_1, \ldots\}$ an ensemble of binary decision trees. Each internal (or branching) node of T_h is associated with a Boolean test over a specific feature $f_\phi \in \mathcal{F}$, and a constant threshold $\gamma \in \mathbb{R}$, where tests are of the form $\mathbf{x}[\phi] \leq \gamma$. Algorithm 1 illustrates the QS [17] algorithm for the fast traversal of the ensemble. One of the most important feature of QS is that it computes $s(\mathbf{x})$ by only identifying the branching nodes whose tests evaluate to false, called *false nodes*. For each false node detected in $T_h \in \mathcal{T}$, QS updates a bitvector associated with T_h, thus storing information that is finally exploited to identify the *exit leaf* of T_h that contributes to the final score $s(\mathbf{x})$. To this end, QS maintains for each tree $T_h \in \mathcal{T}$ a bitvector `leafindexes[`h`]`, made of Λ bits, one per leaf. Initially, every bit in `leafindexes[`h`]` is set to 1. Moreover, each branching node is associated with a pre-computed bitvector `mask`, still of Λ bits, identifying the set of unreachable leaves of T_h in case the corresponding test evaluates to false. To precompute these bitvectors, we consider that the left branch is taken if the binary test performed by a branching node succeeds or, equivalently, the right branch is taken if a branching node is recognized as false. Whenever a false node is identified, the set of unreachable leaves `leafindexes[`h`]` is updated through a *logical AND* ($\wedge$) with `mask`. Eventually, the leftmost bit set in `leafindexes[`h`]` identifies the leaf corresponding to the score contribution of T_h, stored in the lookup table `leafvalues`.

To efficiently identify all the *false nodes* in the ensemble, QS processes the branching nodes of all the trees *feature by feature*. Specifically, for each feature f_ϕ, QS builds a list $\mathcal{N}_\phi$ of tuples $(\gamma, \texttt{mask}, h)$, where γ is the predicate 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 precomputed 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 surely evaluate to true as well, and their evaluation is thus safely skipped.

We call *mask computation* the first step of the algorithm 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 the *Struct of Arrays* (SoA) data layout rather than a classic *Array of Structs* (AoS) [20]. 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. Moreover, the SoA layout simplifies the data parallel implementations, in particular the GPU-based parallel code, as discussed in Sec. 6.

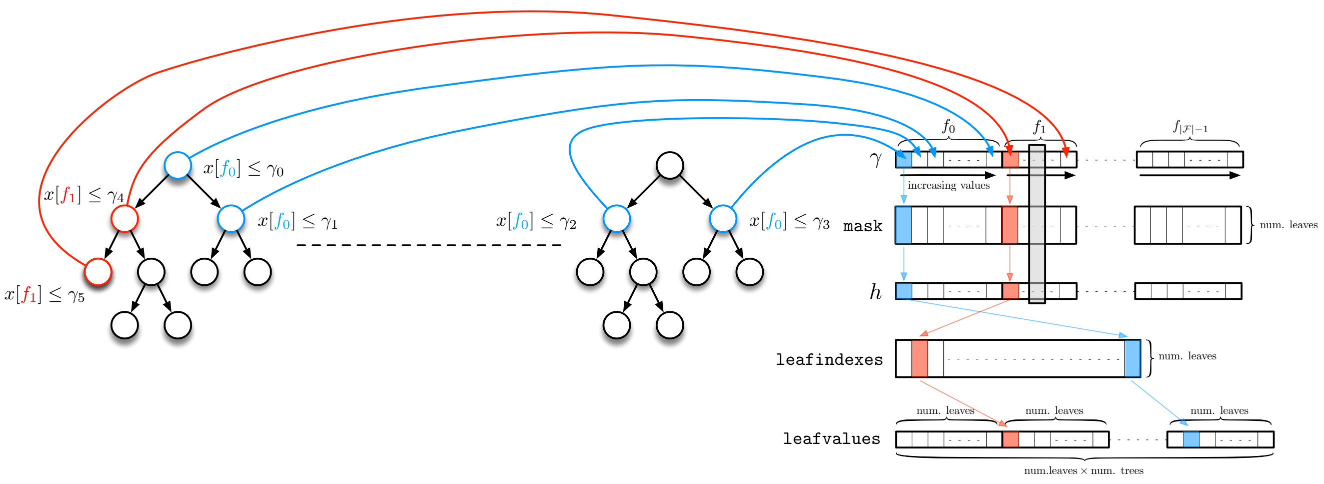


Fig. 1: Data layout example of the QS algorithm.

3 Parallelization strategies for QuickScorer

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.
- Combining inter-document and intra-document parallelism.

In the following paragraphs we introduce in detail each strategy.

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, we observe that 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 available 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.

In QS subtasks can be naturally obtained 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 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.

Combining inter-document and intra-document parallelism. This strategy exploits the combined use of massive and fine-grained parallelism. Indeed, 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.

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: indeed, 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 fit. Consequently, ensembles must be *partitioned* in blocks of τ trees, such that the data structure of each block fits into the cache memory. 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 tradeoff between the size of tree blocks and the number of documents evaluated in parallel.

4 Vectorized QuickScorer

In this section we discuss V-QuickScorer (vQS)[2], an enhanced single-threaded QS that exploits CPU vector extensions to boost the efficient traversing of additive ensembles of regression trees. The results in this section were introduced preliminarly in [21].

Modern CPUs have extended instruction sets and wide registers to realize Data-Level Parallelism (DLP). These instructions permit the parallel execution of the same operation on different data (a.k.a., single instruction multiple data (SIMD) paradigm).

Streaming SIMD Extensions (SSE) and Advanced Vector Extensions (AVX) are sets of instructions exploiting wide registers of 128 and 256 bits. A single SIMD instruction performs parallel operations on simple data types, e.g., a 128 bit register can manage four single precision or two double precision floats simultaneously. Note that Intel's Xeon Phi processor provides 512 bits register, and such wide registers are planned to be included in next-generation mainstream processors.

Given the need of scoring multiple documents at the same time for a given query, *inter-document parallelism* is the most natural source of parallelism that can profit from the exploitation of SIMD instructions, although the number of documents scored in parallel is bounded by the SIMD capabilities of the specific processor, which is equal to 8 in our case.

Both the *mask computation* and *score computation* steps of QS can be engineered to take advantage of SIMD instructions and registers. During the first step, multiple documents can be tested against a given node predicate, and their `leafindexes[`h`]` updated in parallel. Similarly, the scores of multiple documents can be computed simultaneously during the second step. The data structure `leafindexes` used to encode the exit leaves must be replicated to allow multiple documents to be scored simultaneously.

In the following we adopt a notation similar to Intel Intrinsics[3] for specifying SIMD instructions. For each instruction, we have (i) a prefix _mm or _mm256 stating if it operates on either 128 or 256 bits registers; (ii) the name of the performed operation; (iii) a suffix indicating types and number of base operands packed in registers, where for example _ps and _pd stand for 32 and 64 bit floats, respectively. For instance,

$$\vec{c} = \text{\_mm\_cmpgt\_ps}(\vec{a}, \vec{b})$$

corresponds to a SIMD instruction that works on two registers $\vec{a}$ and $\vec{b}$ of 128 bits, each storing a sequence of four single precision floats, and performing four *greater than* comparisons in parallel. The result is stored to another 128-bit register $\vec{c}$, which contains four 32-bits sequences of 1s or 0s, depending on the test outcome of the four comparisons. In the following, we use the notation $\vec{c} \equiv \langle c_3, c_2, c_1, c_0 \rangle$ to refer to the four elements of a SIMD register. We also use $\overrightarrow{c_{3:2}}$ and $\overrightarrow{c_{1:0}}$ to denote respectively the most and least significant pairs of elements in $\vec{c}$.

The specific optimizations used by vQS depend on both SIMD register width and maximum number of leaves Λ in the ensemble. We first discuss how vQS exploits 128 bit registers when $\Lambda = 32$ (Sec. 4.1), then we introduce 256 bit registers (Sec. 4.2). Finally, we highlight the main differences for the case $\Lambda = 64$ when either 128 or 256 bit registers are exploited (Sec. 4.3).

4.1 vQS with 128 bits registers and 32 leaves

We first discuss the *mask computation* step of vQS, see Alg. 2. As QS, vQS identifies *false nodes* by processing

[2] Source code: https://github.com/hpclab/vectorized-quickscorer

[3] https://software.intel.com/sites/landingpage/IntrinsicsGuide

Algorithm 2: vQS (128 bits registers, $\Lambda = 32$)

```
1  V-QuickScorer( $\{\mathbf{x}_i\}_{i=0,1,2,3}$, $\mathcal{T}$, scores$_{3:0}$):
2      foreach $T_h \in \mathcal{T}$ do
3          leafindexes[$h$] $\leftarrow$ 11...11
4      foreach $f_\phi \in \mathcal{F}$ do                      // Mask Computation Step
5          foreach $(\gamma, \mathtt{mask}, h) \in \mathcal{N}_\phi$ in asc. order of $\gamma$ do
6              $\overrightarrow{\gamma} \leftarrow$ _mm_set1_ps($\gamma$)
7              $\overrightarrow{x} \leftarrow$ _mm_set_ps($\mathbf{x}_3[\phi], \mathbf{x}_2[\phi], \mathbf{x}_1[\phi], \mathbf{x}_0[\phi]$)
8              $\overrightarrow{c} \leftarrow$ _mm_cmpgt_ps($\overrightarrow{x}, \overrightarrow{\gamma}$)
9              if $\neg$(_mm_test_all_zeros($\overrightarrow{c}, \overrightarrow{c}$)) then
10                 $\overrightarrow{b} \leftarrow$ _mm_load_ps(leafindexes$_{3:0}[h]$)
11                 $\overrightarrow{m} \leftarrow$ _mm_set1_ps(mask$[n]$)
12                 $\overrightarrow{y} \leftarrow$ _mm_andnot_ps($\overrightarrow{m}, \overrightarrow{c}$)
13                 $\overrightarrow{y} \leftarrow$ _mm_andnot_ps($\overrightarrow{y}, \overrightarrow{b}$)
14                 _mm_store_ps(leafindex$_{3:0}[h]$, $\overrightarrow{y}$)
15             else
16                 break
17     $\overrightarrow{s_{1:0}} \leftarrow$ _mm_set1_pd(0)                  // Score Computation Step
18     $\overrightarrow{s_{3:2}} \leftarrow$ _mm_set1_pd(0)
19     foreach $T_h \in \mathcal{T}$ do
20         $\forall i = 3{:}0 :\ j_i \leftarrow$ index leftmost 1 bit of leafindex$_i$[$h$]
21         $\forall i = 3{:}0 :\ l_i \leftarrow h \cdot \Lambda + j_i$
22         $\overrightarrow{v_{1:0}} \leftarrow$ _mm_set_pd(leafvalues$[l_1]$, leafvalues$[l_0]$)
23         $\overrightarrow{v_{3:2}} \leftarrow$ _mm_set_pd(leafvalues$[l_3]$, leafvalues$[l_2]$)
24         $\overrightarrow{s_{1:0}} \leftarrow$ _mm_add_pd($\overrightarrow{s_{1:0}}, \overrightarrow{v_{1:0}}$)
25         $\overrightarrow{s_{3:2}} \leftarrow$ _mm_add_pd($\overrightarrow{s_{3:2}}, \overrightarrow{v_{3:2}}$)
26     _mm_store_pd($\overrightarrow{s_{1:0}}$, scores$_{1:0}$)
27     _mm_store_pd($\overrightarrow{s_{3:2}}$, scores$_{3:2}$)
```

feature thresholds in ascending order. vQS exploits 128 bits register to compare multiple documents simultaneously against each feature threshold. To this end, the input of vQS is a set of 4 feature vectors $\{\mathbf{x}_i\}_{i=0,1,2,3}$.

Since document features are stored as single precision floats, we have a first register $\overrightarrow{\gamma}$ storing 4 copies of the same test threshold γ, and a second register $\overrightarrow{x}$ storing the features $\{\mathbf{x}_i[\phi]\}_{i=0,1,2,3}$ of the 4 input instances (lines 6-7). A single SIMD instruction is used to test the feature values of these four documents against the threshold (line 8). If at least one test evaluates to false, i.e., vQS finds at least one *false node*, `leafindexes` is updated and the next threshold of the same feature is processed. Otherwise, vQS processes the next feature.

Unlike QS, we need to verify the true condition for all the 4 documents, thus introducing some overhead when vQS performs useless tests on some of the 4 documents. The update of `leafindexes` should occur only for the feature vector $\mathbf{x}_i$ for which $\mathbf{x}_i[\phi] > \gamma$. Since SSE-4.2 does not support masked/predicated SIMD instructions, to avoid conditional branches vQS implements the update with two bitwise operations. Let $\mathtt{leafindexes}_i[h]$ be a 32 bits vector ($\Lambda = 32$), relative to tree T_h and associated with document $\mathbf{x}_i$. Let variable c_i store a sequence of 1-only or 0-only 32 bits, depending on the outcome of the test $\mathbf{x}_i[\phi] > \gamma$. We can rewrite the update as:

$$\mathtt{leafindexes}_i[h] \leftarrow (\mathtt{mask}[n] \vee \neg c_i) \wedge \mathtt{leafindexes}_i[h]$$

where the *bitwise logical or* has the effect of leaving $\mathtt{mask}[n]$ unaltered when $\mathbf{x}_i[\phi] > \gamma$. We can re-write the above expression by applying the De Morgan law to implement the update rule with two executions of the SIMD function $\mathsf{andnot}(x, y) = \neg x \wedge y$.

The layout of `leafindexes` is tree-wise, i.e., given a tree T_h the bitvectors $\mathtt{leafindexes}_i[h]$ of the four $\mathbf{x}_i$ are stored contiguously in memory. As shown in Alg. 2 (lines 10–14), this allows us to load the four bitvectors with a single 128-bit load instruction, and to apply them the two SIMD `andnot` instructions. Indeed, first the four $\mathtt{leafindexes}_{3:0}[h]$ are loaded to register $\overrightarrow{b}$ and $\mathtt{mask}[n]$ is replicated into $\overrightarrow{m}$. After composing $\overrightarrow{m}$, $\overrightarrow{c}$ and $\overrightarrow{b}$, the updated $\mathtt{leafindex}_{3:0}[h]$ are finally copied back to memory.

The *score computation* step is also parallelized (see Alg. 2, from line 17). To provide the required precision, tree predictions are stored as double precision float values (64 bits), but this implies that only 2 document scores can be processed simultaneously using 128 bits registers. Thus, vQS uses two registers, namely $\overrightarrow{s_{1:0}}$ and $\overrightarrow{s_{3:2}}$, to maintain the scores of the 4 documents. For each tree h, the predicted partial scores relative to the 4 input instances $\{\mathbf{x}_i\}_{i=0,1,2,3}$ are similarly stored to $\overrightarrow{v_{1:0}}$ and $\overrightarrow{v_{3:2}}$, and added up to update the final document scores. Finally, the computed scores for the 4 documents are copied to $\mathtt{scores}_{3:0}$.

4.2 vQS with 256 bits registers and 32 leaves

With registers of 256 bits, we can increase the parallelism degree of vQS. Trivially, 8 document features tests can be performed simultaneously instead of 4, and 4 document scores updated in parallel instead of 2. We do not report the pseudocode for document feature testing and document scores calculation, as they simply require to adopt the 256 bits version of the instructions illustrated above.

More interestingly, AVX-2 also provides additional instructions, such as _mm256_maskstore_ps: it stores a 256 bits register to memory on the basis of a mask that enables/disables the copies of sub-groups of 32 bits. This makes it possible to *conditionally* update each of the 8 elements of $\mathtt{leafindexes}_{7:0}$ (or to leave it unchanged), depending on the outcomes of the 8 document tests which are stored in $\overrightarrow{c}$. Lines 12–14 of Alg. 2 are replaced as follows, where the vector variables involved are now 256 bits registers:

$$\overrightarrow{y} \leftarrow \mathsf{\_mm256\_and\_ps}(\overrightarrow{m}, \overrightarrow{b})$$
$$\mathsf{\_mm256\_maskstore\_ps}(\mathtt{leafindexs}_{7:0}[h], \overrightarrow{c}, \overrightarrow{y})$$

4.3 vQS with 64 leaves

Increasing the maximum number of tree leaves Λ impacts on the size of the arrays `leafindexes` and `mask`, as each element they store is Λ bits wide. As a consequence, the number of elements that can be processed simultaneously in a register decreases. Note that $\vec{c}$ stores each of the threshold test results as a string of 32 bits. Conversely, now the elements of $\vec{m}$ and $\vec{b}$ are 64 bits wide. Depending on the register size, this mismatch is handled differently.

For 128 bits registers, let $\vec{c} \equiv \langle c_3, c_2, c_1, c_0 \rangle$ be the outcomes of the four comparisons against a threshold γ. For the subsequent update of the 64 bits masks, vQS requires to process $\vec{c}$ in order to obtain two variables, storing only two comparison outcomes of 64 bits each. To this end, we use the following two *unpacking* instructions, working on the low and high half of $\vec{c}$, respectively.

$\langle c_1, c_1, c_0, c_0 \rangle \equiv$ `_mm_unpacklo_ps`$(\vec{c}, \vec{c})$
$\langle c_3, c_3, c_2, c_2 \rangle \equiv$ `_mm_unpacklhi_ps`$(\vec{c}, \vec{c})$

Once prepared these two result variables, they are used in the subsequent andnot operations, as in Alg. 2. The only difference is that the code from line 12 to 14 must be repeated twice, one for updating the two copies of leafindexes associated with the former pair of documents, and the other for the latter pair.

When using 256 bits registers, vQS performs 8 tests in parallel, and updates the 8 copies of `leafindexes` by exploiting two blocks of SIMD instruction, each performing 4 operations in parallel on the 64 bits data structures. Again, given $\vec{c} \equiv \langle c_7, c_6, \ldots, c_0 \rangle$, vQS needs two vectors with the following layout:

$\langle c_3, c_3, c_2, c_2, c_1, c_1, c_0, c_0 \rangle$ and
$\langle c_7, c_7, c_6, c_6, c_5, c_5, c_4, c_4 \rangle$.

Unfortunately, the 256 bits versions of the unpacking instructions work by considering each 256 bits register as two 128 bits lanes, and thus pick the least/most significant 64 bits from each lane. Hence, vQS adopts a different layout for $\vec{c}$ to apply the above unpacking instructions. vQS loads the 8 features of $\vec{x}$, to be compared with the threshold γ, in the following interleaved order:

$$\vec{x} \leftarrow \text{\_mm256\_set\_ps}(\ \mathbf{x}_7[\phi], \mathbf{x}_6[\phi], \mathbf{x}_3[\phi], \mathbf{x}_2[\phi], \mathbf{x}_5[\phi], \mathbf{x}_4[\phi], \mathbf{x}_1[\phi], \mathbf{x}_0[\phi]\ )$$

This new 256 bits instruction replaces line 7 of Alg. 2.

5 Multi-threading 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 providing several cores, it is interesting to investigate a multi-threading parallelization of QuickScorer. Indeed, 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) this hybrid parallel implementation of QuickScorer.

Even in the case of multi-threading parallelism, we have different possible parallelization strategies, either inter-, intra-document, or a mixed one. From preliminary tests, we found that *inter-document* is always the best performing strategy. 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). So we adopt the same inter-document strategy among the various threads and within each single thread. For example, a multicore CPU with 8 cores, with vector registers of 128 bits, may run 8 threads, where each thread scores 4 documents in parallel, for a total of $8 \times 4 = 32$ documents scored in parallel.

In our study, we also consider the complexity of the shared-memory architecture of modern multiprocessor. 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 indeed distributed to each node, namely a multicore CPU, thus introducing two different speeds for accessing the shared memory: a fast access to local one and a slower access to remote one. This means that migrating a thread from a multicore CPU to another may hinder performance, and thus a good practice is to restrict threads to continue to run on the same multicore CPU where they were created.

We implemented the multi-threaded version of QS by using OpenMP [22], an API that supports multi-platform and multi-language shared memory multiprocessing programming. To realize inter-document parallelism with OpenMP, it is enough to denote as a `parallel for` the loop that iterates over the documents to score. In more details, a single-thread program calls Algorithm 2 from within a `for`, thus scoring 4 documents at a time (8 documents, when 256 bit SIMD instructions are exploited). 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 are 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 cache, 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, and thus we can conclude that multi-threading does not impact too much on cache performance of vQS MT.

6 GPU-based QuickScorer

In this section we discuss the GPU-based parallelization of QuickScorer. Before detailing the strategies adopted, we first introduce a background on GPUs architecture and some related works concerning the exploitation of GPUs to score/classify with large forests of decision trees. The pseudo-code, used to describe the GPU-based parallelization of QS, uses a high-level notation that resembles the constructs of common parallel programming abstractions used in GPU programming environments like CUDA [23].

6.1 GPU architectural background

During the latest years the possibility of using GPUs featuring *thousands of cores* to parallelize general purpose computations has attracted the interest of many researchers and developers. GPUs showed potential for substantial performance gains when compared to traditional multicore architectures [19]. However, effectively exploiting the computational power of GPUs is usually far from trivial. In the following we sketch the internal organization of GPUs, and discuss issues and best practices to effectively exploit their computational power.

Although GPUs contains thousands of cores, indeed simpler functional units orchestrated by centralized SIMD controls, each core is slower than those of typical CPUs and has limitations concerning its access to the device memory, thus resulting in potential contentions unless specific conditions are satisfied [24]. Moreover, GPU cores must carefully coordinate their actions; this is usually a complex issue, considered their internal organization. Proper algorithms, designed with the architectures of the GPUs in mind, are needed to maximize the performance and obtain significant gains over CPU-based algorithms – a goal which is not always possible to pursue, depending on the characteristics of the targeted problem [25].

GPU cores are grouped in *Streaming Multiprocessors* (SMs): if m is the number of SMs in a GPU and each SM features n cores, the total number of cores is equal to $m \cdot n$. Each SM is able to run *data-parallel tasks*, organized as *blocks* of *threads* (*thread-blocks*). Each thread constitutes an abstract entity that represents the execution of a *kernel*, which in turn represents a small function/program. The same kernel code is shared across all the threads of all the thread-blocks.

The execution model associated with thread-blocks requires to exclusively assign each thread-block to a given SM, where the threads of the block are executed concurrently. Thread-blocks have typically many more threads than the cores available in a single SM, but only a subset of these threads can run in parallel at a time. Specifically, each SM is able to schedule and execute in parallel one or more groups of threads, called *warps*. A warp is a sort of 32-way SIMD thread, as it consists of 32 *synchronous, data-parallel threads*, executed by an SM in lockstep according to a SIMD paradigm [24,26]. Due to this model of execution, it is important to avoid *branch divergence* within a warp; more precisely, threads inside a warp may diverge due to a data-dependent conditional branch, but this eventually forces the warp to execute serially each branch , while threads not belonging to that path are kept inactive. Overall, this may cause an under-utilization of the GPU's computational resources.

An important feature of the various families of GPUs is the number of warps whose execution can be supported by a given SM. This depends on the scheduler features, the number of cores available, and other characteristics of the architecture. It is worth remarking that multi-threading, which in GPUs becomes multi-warping, is a well known technique for hiding latency, due for example to memory access. Latency hiding is effective if the probability of having warps that are eligible for executing on a given SM is high. To this end, a good strategy is to increase the *occupancy* of the SM, i.e., the amount of active warps per SM. We can obtain this result by increasing the number of threads assigned to each thread-block, but also tuning the code to allow the GPU to run concurrently more that one thread-block per SM. In this way, warps from different blocks can be activated and possibly executed simultaneously.

GPUs also own different types of memories. Each SM is equipped with *private* (thread) registers, a small read-only *constant memory*, an L1 cache, and a fast *shared memory* unit. The last kind of memory, characterized by low latency and high bandwidth, is actually shared among the cores of the SM, hence threads within a warp can access shared memory locations in parallel. Shared memory is allocated per thread-block; in particular, since it is important to favour the concurrent execution of several thread-blocks by a given SM, only a portion of the SM's shared memory can be allocated and accessed by each thread-block. In addition, GPUs feature a *global memory* – the adjective *global* here refers to the scope of the memory, as it can be accessed and modified both from the host (CPU) and the GPU cores – which has a much lower throughput than the shared memory, but is orders of magnitudes greater in terms of size. Accesses to global memory generally pass through a L2 cache that is shared among all the SMs. In some new models, included the one used for our experiments, each SM is also equipped with a dedicated L1 cache, which can optionally cache all the loads to global memory.

To achieve optimal performance the programmer must be aware of this complex memory hierarchy and properly orchestrate memory accesses and transfers. To this end, we need to consider the execution model of threads, in particular the hierarchy *blocks–warps*; more precisely, the grouping of threads into warps is not only relevant to computation but also when accessing the global and shared memories.

For what concerns global memory accesses, even mediated through a faster L2 cache shared by all GPU's SMs, GPU devices try to *coalesce* loads/stores issued by the threads of a warp into as few memory transactions as possible. Consequently, if such threads, identified by consecutive IDs, access consecutive words in global memory, their accesses can be merged (*coalesced*) in 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 – the size of a single block is 32 bytes on current NVIDIA GPUs. For example, the 32 threads of a warp can access the global memory in parallel with four memory transactions if their aggregated requests result in four blocks of 32 bytes, each aligned at 32 bytes. Conversely, if the requests are strided the resulting bandwidth is lower, since a greater number of transactions are needed to fulfill the requests.

Shared memory represents the other main actor in the GPUs' hierarchy of memories: this kind of memories is very fast – typically one order of magnitude faster than global memory – and each SM is equipped with its own unit; consequently, all the threads within a thread-block have access to the same shared memory unit. Each unit is typically small in size – as a term of reference, NVIDIA Pascal GPUs feature 64 – 96 KB (depending on the model) of shared memory per SM, while individual thread-blocks have access to at most 48 KB – and structured in *interleaved* memory banks, where the rationale behind the interleaved layout is to achieve high memory bandwidth. This occurs because successive word addresses are assigned to successive memory banks, while memory banks can work together to serve concurrent requests from the various threads of a warp. In general terms, to maximize the effective bandwidth of a shared memory unit one has to minimize bank conflicts, i.e., to orchestrate the accesses among the threads of a warp such that they access words belonging to distinct banks. Thanks to the interleaved organization and many other minor optimizations, access to shared memory is overall much faster than global memory.

All in all, if an algorithm needs to randomly access a data structure without a statically predictable pattern, it is desirable to move such data to shared memory, since random memory accesses directed to global memory cannot be coalesced. However, given the limited size of SM's shared memory we can exploit from within a thread-block, we have to allocate on such memories suitable structures, i.e., small data structures that can fit in shared memory, or that can be partitioned by limiting the accesses from parallel threads to a *sufficiently small* partition at a time.

Also, since a shared memory unit is shared across all the threads of a thread-block, it is then possible to exploit the unit for thread cooperation. Finally, we report that the latest generations of GPUs support very efficient shared memory atomic operations at hardware level, thus greatly mitigating the negative effects deriving from race conditions between the threads of a thread-block/warp – this occurs when the threads access concurrently the same shared memory locations.

6.2 Related work

Literature about GPU-based algorithms to score/classify with forests of trees is limited to classifiers based on small ensembles of random-trees. Schulz *et al.* [27] and Van Essen *et al.* [28] point out that devising this kind of approaches presents relevant challenges, in the sense that random trees are characterized by structural irregularity; this, in turn, represents a serious obstacle when considering the execution model and the hierarchy of memories characterizing modern GPUs. In a few selected cases it may be possible to overcome the aforementioned issues by carefully designing GPU algorithms that exploit the properties of a given problem, such as in the case of the image labeler [27]. In general, however, these solutions have a very limited scope.

Van Essen *et al.* [28] propose a GPU-based approach that use Compact Random Trees (CRFs), i.e., random trees having fixed depth, to control the structure and the size of trees, thus fitting well the architectural peculiarites of the GPUs. Overall, we note that the proposed solution resembles a less refined version of the approach proposed by Asadi *et al.* [6], thus implying benefits and limitations similar to VPRED. In the experimental evaluation the authors show how their solution outperforms the CPU-based counterpart in terms of performance, economic costs, and power consumption. We also report, however, that the authors did use ensembles having a fixed, limited size – 32 CRFs per ensemble – while each CRF had a fixed, limited depth (six levels); finally, the authors did not consider the effects that variations in the size of the ensembles, depth of the CRFs, and number of features have on the performance.

6.3 GPU-QuickScorer

There are two main challenges in designing algorithms for GPUs – these apply to the GPU-based parallelization of QuickScorer as well. The first challenge is to make available a sufficiently large degree of parallelism to profit from the thousands of cores available. The second challenge is to take advantage of the GPUs' 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 sufficient parallelism degree in parallelizing QS, we combine inter- and intra-document parallelism, as discussed in Section 3. This hybrid technique is indeed useful to exploit both coarse and fine-grained parallelism, as we want to potentially process in parallel p_1 documents (*inter-document* parallelism) by 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 score in parallel to a single block of threads. Therefore we have $p_1 = M$, where M is the number of blocks of threads 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. Indeed, we can benefit from an increased task granularity (see Section 3), by assigning multiple documents to each thread-block.

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 $(\gamma, \texttt{mask}, h)$ associated with a given feature. In addition, since we have usually less warps than features ($\frac{N}{32} \ll |\mathcal{F}|$), we also end up increasing the task granularity by assigning more features per warp.

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

- the *model data structure*, composed of the tuples $(\gamma, \texttt{mask}, h)$ encoding the branch nodes of the forest $\mathcal{T}$, and `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 access it feature-by-feature, with perfect spatial locality. Conversely, QS accesses in *read-write* mode the *output* vectors without any regularity, since it is not possible to exploit any predictable pattern to promote locality.

The model data structure is too large to be stored in shared memory, given the limited size of the latter (typically few tens of KBs). For example, let us consider a model $\mathcal{T}$ made up of a forest of 10,000 trees: for only storing the `leafvalues` of all the trees, where each leaf value is represented as a `double` of 8 bytes, we need a space of about 5 MB. The size of global memory, however, is typically in the order of 4 − 8 GB, thus representing the best candidate to store $\mathcal{T}$, even if high-throughput access to such memory is only possible if we can exploit *coalescing*.

Indeed, 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 by QS *feature-by-feature* by means of *linear scans* to 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 $(\gamma, \texttt{mask}, h)$ associated with a feature $f_\phi \in \mathcal{F}$: in turn, this leads to coalesced accesses. We finally mention that by partitioning $\mathcal{T}$ we increase the chance that individual tree-blocks fit into the shared L2 cache, thus further increasing the bandwidth that can be achieved.

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, each having 64 leaves, require

Algorithm 3: GPU-QuickScorer

```
GPU-QuickScorer($\mathcal{D}$,$\mathcal{T}$):
    copyHostToGPU&Transpose($\mathcal{T}$)
    foreach document batch $D$, $D \subseteq \mathcal{D}$ do
        copyHostToGPU($D$)
        $\mathbf{S}_D \leftarrow \{s_0 = 0, \cdots, s_{|D|-1} = 0\}$
        foreach tree-block $T$, $T \subseteq \mathcal{T}$ do
            pos_pivot ← findFalseNodesGPU ($\mathcal{D}, T$)
            updateScoresGPU ($T$, pos_pivot, $\mathbf{S}_D$)
        return $\mathbf{S}_D$
```

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 translate into dealing with random memory accesses (no coalescing), thus incurring in negative impacts on performance. Hence, rather than storing `leafindexes` to global memory, it is far more efficient to partition the model, as already discussed in Section 3, to make the output vector suitable for shared memory. More precisely, $\mathcal{T}$ gets 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` at least fit the amount of shared memory available per thread-block. Performance-wise, the shared memory is at least one order of magnitude faster than global memory, it does not require coalescing, it can serve up to 32 parallel requests, and it can effectively manage atomic operations at hardware level. As mentioned above, another motivation to adopt a small size τ of model partitions is to increase the chance that each partition fit into the L2 cache, thus minimizing cache miss rates and thus average global memory latency.

A final remark concerns where the input data, i.e., the features vectors associated with the documents to score for a given query, gets stored: more precisely, the feature vectors are moved from the CPU to the GPU global memory in large batches $D \subseteq \mathcal{D}$; since the global memory is large in size, it can easily host large batches of input documents $D \subseteq \mathcal{D}$: for example, 10,000 documents, with 1,000 features each, take only 40 MB.

The GPU algorithm. GPU-QuickScorer (QS$_{\text{GPU}}$), the GPU-based version of QuickScorer, is roughly structured in two phases, sketched in Algorithm 3.

High-level overview. The algorithm starts by transferring the entire model from the host (CPU) memory to the GPU global memory (line 2). We note that the uploaded model comes already partitioned into disjoint tree-blocks $T \subseteq \mathcal{T}$, where each T is stored separately in contiguous regions of the global memory. The documents to be scored are also transferred from the host memory to the GPU global memory in batches (line 4), while their initial scores are set to zero (line 5). At the same time, each batch of documents gets also transposed in parallel by the GPU (line 2); more precisely, the original layout of D is an array of vectors of features (we denote each vector by $\mathbf{x} \in D$), while the transposed layout features an array of vectors where each vector contains the values of the same feature $f_\phi \in \mathcal{F}$ across all the documents in D. We note that this layout allows to parallelize the retrieval of false nodes for all the tree-blocks $T \subseteq \mathcal{T}$ – we discuss this point later.

Subsequently, the algorithm iterates across the various tree-blocks $T \subseteq \mathcal{T}$ (line 6). Unlike the sequential QS, false nodes are first identified (findFalseNodesGPU, line 7) before proceeding to update `leafindexes` and the scores of the documents (updateScoresGPU, line 8).

First phase – finding the stop 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 following inequality holds: $\mathbf{x}[\phi] \leq \gamma$. In other words, a pivot separates the false nodes from the true nodes among the branching nodes that perform their tests over f_ϕ. It is worth remarking that using the transposed layout for a given batch of documents D allows to use an efficient GPU-based vectorized binary search [4] to search in parallel the pivots and store them in **pos_pivot**; also, the transposed layout arranges the values of a given f_ϕ for all the documents in D contiguously, thus allowing to coalesce read accesses performed by the threads of a warp.

Finally, we note that the operation carried on by findFalseNodesGPU relies on an inter-document parallelization strategy, thus distinguishing itself from the computation that follows. This separation implies a better access to the SoA data structures holding the tree-based model: indeed, findFalseNodesGPU needs, for all $\mathbf{x} \in D$, to access just the field γ in the related SoA data structure holding the tuples, thus favoring a better exploitation of the L1/L2 global memory caching – this implies also that accesses to the fields `mask` and h are delegated to the subsequent phase in updateScoresGPU.

Second phase – updating the scores. Once the positions of the pivots are available in **pos_pivot** (line 7),

[4] To this end, we exploit the Thrust library, v1.7.0, provided by the CUDA framework.

the algorithm must proceed 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 only need to access **pos_pivot**, since the original documents $\mathbf{x} \in D$ and the field γ in the tuples $(\gamma, \mathtt{mask}, h)$ are no longer needed. This second phase is realized by the function UPDATESCORESGPU (line 8, Algorithm 3), whose pseudocode is detailed in Algorithm 4.

As stated above, to maximize GPU computation we need to feed the thousands of GPU cores by combining the *inter-document* and *intra-document* parallelization strategies. For what concerns inter-document parallelization, each document is assigned to a single block of threads – indeed, in Algorithm 4, line 2, we use the notation $\mathbf{parallel_{block}}$ to indicate that each iteration of the loop is assigned to a different block of threads. As discussed before, the model $\mathcal{T}$ comes partitioned to make sure that a block of threads has sufficient resources to manage `leafindexes` in shared memory (line 3). For what concerns intra-document parallelization, this is achieved within each block of threads 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 $\mathbf{parallel_{thread}}$ 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 proceeds to the mask computation step, where we take advantage of the grouping of threads into warps. Indeed, 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 $\mathbf{parallel_{warp}}$ (line 7). Going further on, the construct $\mathbf{parallel_{thread}}$ 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 the tuples, accesses performed by the threads of a warp are distributed sequentially in global memory, thus allowing to exploit *coalesced* accesses. Subsequently, the retrieved masks are used to update in parallel the `leafindexes` of the corresponding trees: accesses to `leafindexes` are random and potentially conflicting, thus atomic updates are employed to guarantee consistency (line 9). Finally, the loop ends when all the false nodes have been processed for the currently considered document $\mathbf{x}$ and feature f_ϕ, i.e., until the position $\mathbf{pos\_pivot}[\mathbf{x}][f_\phi]$ in the tuple array is reached by some of the threads of the warp. Note that when this position is reached from within a warp, some of the 32 threads of the warp may result inactive, since `leafindexes` must not be modified for nodes after the pivot. This may hinder GPU efficiency, and thus the overall performance of the parallel algorithm.

After that all the features have been processed, the algorithm has to update the document score by adding the contributions of the currently considered tree-block T. First the vector `leafindexes` gets 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 lies in 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, it possible to maximize the chance that `leafvalues` fits into the L2 cache by picking up a proper τ.

Finally, the threads of the block performs 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)[5].

7 Experiments

In this section we discuss the results of extensive experiments conducted to assess the performance of the different parallelization strategies applied to QS. Specifically, our goal is to measure the scoring efficiency of the different proposed parallel versions of QS using an off-the-shelf shared-memory multiprocessor, and compare their performance with the GPU-based versions. For all the solutions, we also evaluate and explain the performance results obtained by using tools for low-level profiling, e.g. to measure cache-misses or GPU branch divergence.

7.1 Datasets and experimental settings

We conduct experiments on three publicly available datasets, namely the MSN, Yahoo LETOR, and ISTELLA ones, which are datasets commonly used in the scientific community to evaluate LtR solutions. The characteristics of the three datasets are listed in Table 2. All the datasets provide query-document pairs labeled with relevance judgments ranging from 0 (irrelevant) to 4 (perfectly relevant).

[5] To this end, we exploit the CUB library, v1.7.0 (https://nvlabs.github.io/cub/).

Algorithm 4: The UPDATESCORESGPU kernel

```
UPDATESCORESGPU($T$, pos_pivot, $\mathbf{S}_{\mathcal{D}}$):
    parallel_block foreach $\mathbf{x} \in \mathcal{D}$ do
        shared leafindexes[$\tau$], where $\tau = |T|$
        parallel_thread foreach $t_h \in T$ do
            leafindexes[$h$] ← 11...11
        synchthreads
        parallel_warp foreach $f_\phi \in \mathcal{F}$ do                          // ① Mask Computation Step
            parallel_thread foreach $(\gamma, \mathtt{mask}, h) \in \mathcal{N}_\phi^T$ in ascending order, up to the pos_pivot[x][$f_\phi$] -th element do
                leafindexes[$h$] ← (leafindexes[$h$] $\wedge_{\mathrm{atomic}}$ mask)
        synchthreads
        local $accScores$ ← 0                                                    // ② Score Accumulation Step
        parallel_thread foreach $t_h \in T$ do
            local $j$ ← index of the leftmost bit set in leafindexes[$h$]
            local $l \leftarrow h \cdot \Lambda + j$
            $accScores$ ← $accScores$ + leafvalues[$l$]
        synchthreads
        $\mathbf{S}_{\mathcal{D}}[\mathbf{x}] \leftarrow \mathbf{S}_{\mathcal{D}}[\mathbf{x}] + BlockSumReduction(accScores)$   // ③ Score Reduction Step
    return $\mathbf{S}_{\mathcal{D}}$
```

– The MSN dataset is available at http://research.microsoft.com/en-us/projects/mslr/. The dataset is divided into five folds. In this work, we use only the first fold, namely MSN-1.
– The Yahoo dataset is available at http://learningtorankchallenge.yahoo.com. It consists of two distinct datasets (Y!S1 and Y!S2). In this paper we use the Y!S1 dataset.
– The Istella (Full) dataset is available at http://blog.istella.it/istella-learning-to-rank-dataset/. To the best of our knowledge, this dataset is the largest publicly available LtR dataset, particularly useful for large-scale experiments on the efficiency and scalability of LtR solutions [17]. Moreover, it is the first public dataset being representative of a real-world ranking pipeline, with long lists of results including large numbers of irrelevant documents for each query, as also discussed in [29].

The experimental methodology adopted is the following. We use training data from MSN-1, Y!S1, and Istella datasets to train λ-MART [13] models by optimizing NDCG@10. The different generated models are ensembles of trees, whose number of leaves is equal to either 32 or 64. To train these models we use QuickRank[6], an open-source LtR C++11 framework that provides efficient implementations of LtR algorithms [5]. It is worth noting that the results reported in the paper, concerning the efficiency at testing time of tree-based scorers, are *independent* of the specific LtR algorithm used to train individual ensembles of trees.

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 192 GB RAM. Each Xeon CPU has 8 general-purpose cores; each core has a dedicated L1 cache of 32 KB, a dedicated L2 cache of 256 KB, and a shared L3 cache of 20 MB. The machine runs Ubuntu Linux 14.04.5 LTS (kernel GNU/Linux 3.13.0-121-generic (x86_64)).

The system also includes an NVIDIA GTX 1080 GPU. The GPU is equipped with a 2 MB L2 cache, 8 GB GDDR5X RAM, and 20 streaming multiprocessors, each having 128 cores, a 96 KB shared memory unit (48 KB accessible by individual thread-blocks), and a 48 KB L1 cache.

All the version 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` afre used for the CUDA compiler. More specifically, the former flag enables global memory caching via L1 cache, while the latter generates optimized code that targets the GPU used in the experiments.

To measure the efficiency of the above methods we run 10 times the scoring code on the test sets of the MSN-1, Y!S1, and Istella datasets. We then compute the average per-document scoring cost. Moreover, to profile the behavior of each CPU-based QS version, we employ `perf`[7], a performance analysis tool available under Ubuntu Linux distributions. Analoguously, to profile the GPU-based version of QS we employ `nvperf`, a GPU performance analysis tool provided by the NVIDIA CUDA framework.

[6] http://quickrank.isti.cnr.it

[7] https://perf.wiki.kernel.org

Table 2: Dataset features: i) number of features, ii) number of queries in train/validation/test sets, iii) total number of documents in train/test sets, and iv) average number of document per query in the test set.

Property	Dataset		
	MSN-1	Y!S1	Istella
# features	136	700	220
# queries in training	6,000	19,944	23,319
# queries in validation	2,000	2,994	–
# queries in test	2,000	6,983	9,799
Total # documents in train	723,412	473,134	7,325,625
Total # documents in test	241,521	165,660	3,129,004
Average # documents per query in test	120.7	23.72	319.31

7.2 Vectorized QuickScorer

We report the performance in terms of per-document scoring time (μs) of Vectorized QuickScorer (vQS) against the sequential QS version in Table 3. To ease the reading, in the following we refer to the vQS employing AVX-2 as vQS-256 – we limit ourselves to this version as it always outperforms the one using 128 bits registers.

From the table, we see that vQS outperform QS with significant speedups; more precisely, when models feature trees having $\Lambda = 32$ leaves each, we observe that the best performance is always obtained by vQS-256, which yields speedups ranging from 1.9x to 3.2x over QS. The observed trend in performance remains the same when $\Lambda = 64$; in this context, we also observe that vQS-256 remains the fastest method even if its speedup over the sequential implementation of QS slightly decreases and ranges from 1.2x to 1.8x.

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. In these tests we compare QS against vQS-256, using for testing the largest and most challenging ISTELLA dataset. Experiments on the other datasets are not reported here, as they exhibit a similar behavior.

Table 5 reports the results achieved with all measurements normalized per-document and per-tree. It is worth specifying that L3 cache references accounts for those references which are not found (misses) in any of the previous levels of cache, while L3 cache misses account for the percentage of L3 cache references that miss in L3 as well.

Interestingly, the analysis reveals that the use of the AVX-2 256 bit instruction set causes a significant decrease in the average number of instructions needed to score a single document. This reduction justifies the speedup achieved by the SIMD implementation. In terms of branch figures, vQS-256 shows lower misprediction 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 utilisation. As in the preceding case, L3 cache misses and references are always lower than the ones of QS, thus revealing a more effective use of the cache.

7.3 Multi-threading QuickScorer

In this section we discuss the performance results obtained by vQS MT against vQS and QS. We employ OpenMP to distribute the threads of vQS MT among the processing cores available within our multiprocessor, indeed a NUMA system with 2 nodes, where each node is an 8-way multicore CPU equipped with a local memory. Each thread uses the SIMD instructions to score bunches of documents at a time (till 8 documents at a time), and thus its implementation is based on vQS. The results, reported in terms of per-document scoring time (μs), are thus obtained by running the vQS MT on the 16 physical cores of our NUMA multiprocessor, without using hyper-threading – namely, an INTEL technology that allows 2 threads to run simultaneously on the same core by sharing its computational resources. In fact, our threads are compute-bound, and we experimentally verified that the adoption of hyper-threading actually reduces the overal performance. Moreover, we use the `numactl` tool to force a thread allocation on the NUMA architecture. This means that a thread will use the local memory of the node where it runs for all its life-cycle. This avoid slower accesses to memory of different nodes of the NUMA architecture. Table 4 finally reports the results.

From the table, we see that vQS MT's speedups range from 8.5x to 14x in the case of 32 leaves, while they range from 6.3x to 12.5x in the case of 64 leaves.

Table 3: Per-document scoring time in μs of QS and vQS on MSN-1, Y!S1, and Istella datasets. Speedups over QS are reported between parentheses.

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-256		**2.8**	(2.5×)	**3.9**	(3.2×)	**3.1**	(2.9×)	**17.4**	(1.9×)	**20.8**	(2.1×)	**14.3**	(2.4×)
QS	64	12.4	(–)	19.2	(–)	13.5	(–)	70.7	(–)	83.3	(–)	69.8	(–)
vQS-256		**8.3**	(1.5×)	**10.4**	(1.8×)	**7.9**	(1.7×)	**60.8**	(1.2×)	**64.6**	(1.3×)	**46.3**	(1.5×)

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-256		**39.6**	(1.9×)	**44.2**	(2.0×)	**31.1**	(2.3×)	**87.8**	(2.1×)	**88.5**	(2.1×)	**64.8**	(2.4×)
QS	64	194.8	(–)	186.9	(–)	167.4	(–)	470.5	(–)	377.2	(–)	326.1	(–)
vQS-256		**146.9**	(1.3×)	**136.8**	(1.4×)	**105.9**	(1.6×)	**321.7**	(1.5×)	**274.1**	(1.4×)	**236.6**	(1.4×)

Instruction level analysis. The low-level statistics performed on Istella with 64-leaves λ-MART models (Table 5) show that vQS MT inherits the same figures from vQS-256. The use of OpenMP to parallelize vQS does not incur in computational overhead, as the instruction count is the same as vQS-256. The same considerations can be done for the total number of branches and branch mis-predictions. In terms of L3 cache misses, vQS MT shows an increased number of misses when dealing with the increasing size of models. This is an expected behaviour, due to the architecture of the Intel Xeon processor presenting a L3 cache shared among all the cores. Finally, we report that the high number of threads working concurrently during the scoring process affects negatively the temporal and spatial locality, hence leading to a higher number of cache misses than its competitors.

7.4 GPU-based QuickScorer

This section discusses the performance of GPU-QuickScorer (QS_{GPU}). GPUs provide large computing power at the cost of a few constraints that impact on the design and tuning of algorithms. Such constraints are similar in nature, yet they quantitatively differ across GPUs. Hereinafter, without loss of generality, we focus on the GPU used in this experimental evaluation, an NVIDIA GTX 1080 GPU, and report its constraints in Table 6.

First, we must consider that a single thread-block can access only up to 48 KB of shared memory. Within the *updateScoresGPU* kernel, which represents the time-dominant component of QS_{GPU}, QS_{GPU} exploits the shared memory of each SM to store `leafindexes`. If $\Sigma_\tau = \tau \cdot \Lambda/8$, with $\Sigma_\tau \leq 48$ KB, represents the size (in bytes) of the shared memory footprint of a model of τ trees, we have that each thread-block can process at most τ trees.

Example. Given a large model composed of $20,000$ trees with $\Lambda = 32$, the `leafindexes` data structure for all the trees of the model would require $\Sigma_\tau = 20,000 \cdot 32/8 = 80,000$ bytes of storage, beyond the 48 KB limit. Therefore, the maximum number of trees that can be included in a single partition of the model is $\tau = 12,288$ for $\Lambda = 32$, and $\tau = 6,144$ for $\Lambda = 64$.

As discussed previously, this does not represent a major issue for QS_{GPU}, as any model can be evaluated in chunks of any custom size τ. However, splitting a model into pieces 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 the *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*.

Table 4: Per-document scoring time in μs of vQS and vQS MultiThread on MSN-1, Y!S1, and Istella datasets. Speedups over vQS-256 are reported in parentheses.

Method	Λ	Number of trees/dataset											
		1,000						5,000					
		MSN-1		Y!S1		Istella		MSN-1		Y!S1		Istella	
vQS-256	32	2.8	(–)	3.9	(–)	3.1	(–)	17.4	(–)	20.8	(–)	14.3	(–)
vQS MT	32	**0.2**	(14.0×)	**0.4**	(9.8×)	**0.3**	(10.3×)	**1.4**	(12.4×)	**1.9**	(10.9×)	**1.2**	(11.9×)
vQS-256	64	8.3	(–)	10.4	(–)	7.9	(–)	60.8	(–)	64.6	(–)	46.3	(–)
vQS MT	64	**0.7**	(11.9×)	**1.0**	(10.4×)	**0.7**	(11.3×)	**4.9**	(12.4×)	**10.2**	(6.3×)	**3.7**	(12.5×)

Method	Λ	Number of trees/dataset											
		10,000						20,000					
		MSN-1		Y!S1		Istella		MSN-1		Y!S1		Istella	
vQS-256	32	39.6	(–)	44.2	(–)	31.1	(–)	87.8	(–)	88.5	(–)	64.8	(–)
vQS MT	32	**3.2**	(12.4×)	**4.1**	(10.8×)	**2.5**	(12.4×)	**7.3**	(12.0×)	**8.2**	(10.8×)	**7.6**	(8.5×)
vQS-256	64	146.9	(–)	136.8	(–)	105.9	(–)	321.7	(–)	274.1	(–)	236.6	(–)
vQS MT	64	**12.5**	(11.8×)	**14.6**	(9.4×)	**8.8**	(12.0×)	**46.2**	(7.0×)	**35.1**	(7.8×)	**26.1**	(9.1×)

Method	Number of Trees				
	1,000	5,000	10,000	15,000	20,000
Instruction Count					
QS	67	70	79	81	73
vQS-256	**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-256	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-256	**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: Per-tree per-document low-level statistics on Istella with 64-leaves λ-MART models.

Table 6: 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

Secondly, τ determines the maximum number of thread-blocks that can run concurrently on the same SM; indeed, each SM is equipped with only 96 KB of shared memory, which limits the number of thread-blocks that can be concurrently executed to a maximum of $\beta_\tau = \lfloor 96\text{ KB}/\Sigma_\tau \rfloor$.

While satisfying the last constraint we also aim 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. We note that if we maximize occupancy by simply increasing the number of threads per thread-block, we may end up reducing the number of thread-blocks concurrently executed per SM, due to the constraint on the total number of active threads per SM (max 2,048).

Consequently, besides *occupancy* it is also important to maximize the *number of thread-blocks* per SM, which in turn depends (i) on the number of threads assigned to each thread-block and (ii) on the total number of thread-blocks deployed. Indeed, maximizing the number of thread-blocks per SM maximizes also the chance of keeping SMs busy, as warps stalled by block-level barriers can be masked by other *eligible* warps.

All in all, given a specific τ the policy we use to maximize occupancy, while maximizing also the number of thread-blocks concurrently executed per SM, 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 concurrent thread-blocks per SM (i.e, 2,048/*n_threads*), provided that this number is not greater than β_τ. Note also that the number of thread-blocks per SM 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 6). We validated the 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 $\tau = 4{,}000$ and $\Lambda = 32$. 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, related to *n_threads*, is *n_blocks*, i.e., the overall number of thread-blocks to allocate when executing the kernel *updateScoresGPU*. This parameter is not critical, since it is sufficient to allocate a large number of thread-blocks, i.e., $n\_blocks \gg m * \beta_\tau$, where $m * \beta_\tau$ represents the maximum number of blocks actually running in parallel on the $m = 20$ SMs of our GPU. Indeed, since we have to score huge amounts of query-document pairs and the pairs are distributed over the thread-blocks, we have that the greater the number of thread-blocks, the finer the granularity of aggregated tasks assigned to each thread-block, which in turn guarantees a better load balancing of the workload distributed over the SMs.

The last key factor is the L2 cache memory size (2 MiB), which has a strong influence on the tuning of τ. Indeed, the cache size impacts on the access time to the remaining data structures stored in the GPU global memory. More precisely, for each internal node QS$_{\text{GPU}}$ 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, we observe that the cache size imposes a stricter upper bound than the shared memory constraint.

Example. QS$_{\text{GPU}}$ requires $(\Lambda/8 + 2) \cdot (\Lambda - 1)$ plus $8 \cdot \Lambda$ bytes for each tree. Given a model with 20,000 trees, the 2 MiB constraint gets already violated when $\tau = 5,000$ with $\Lambda = 32$, and $\tau = 2,000$ with $\Lambda = 64$.

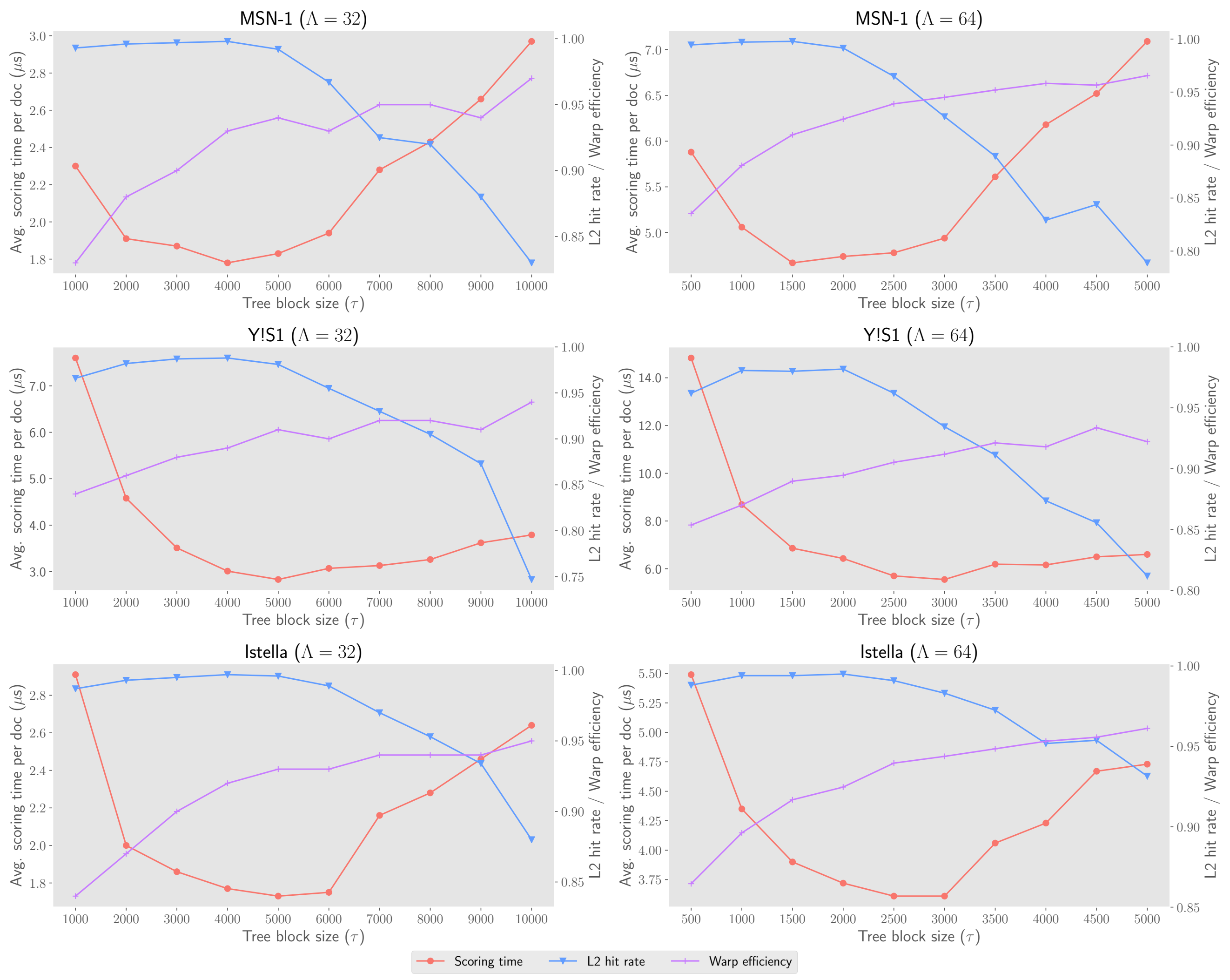
All in all, to exploit the computational power of a GPU we need to pursue two contrasting goals: *maximizing* the *warp efficiency* by using a large τ, and *maximizing* the *hit rate* of the L2 cache by using a sufficiently small τ. We argue that these goals can be achieved by using a value of τ that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

In the batch of experiments that follows, we validate our analytic performance model and the choice of τ for optimal performance. Indeed, 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_blocks*, is set to the highest possible value, $64K - 1$, as this ensures the best possible load balancing of the workload. The dataset features are $|\mathcal{T}| = 20,000$ 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.

From the Figure, 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 tree-blocks implies less pivots, which in turn reduces the number of times 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; indeed, when the cache performance starts to degrade ($\tau \geq 6,000$ with $\Lambda = 32$, and

Fig. 2: Performance analysis of GPU-QuickScorer 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 (32K). The primary Y axis is used to report the average scoring time per document (in μsec.), while the secondary Y axis is used to report the L2 cache hit rate and the Warp efficiency.



$\tau \geq 3,000$ with $\Lambda = 64$), the scoring time starts to increase noticeably. In the next section we compare the best scoring times achieved by QS$_{\text{GPU}}$ with the ones obtained by the other parallel solutions.

7.5 Overall comparison

From the experimental results discussed so far, we already saw the interesting speedups obtained by exploiting different types of parallelization. In this section we summarize and compare the best scoring times obtained by QS$_{\text{GPU}}$, vQS MT, and the sequential QS over different datasets and learned tree models. Recall that vQS MT exploits both SIMD and multi-threading parallelism to fully exploit our NUMA multiprocessor – we remind that our CPU is composed of 2×8 cores, and each core allows 8-way SIMD parallelism (AVX-2). Table 7 shows the per-document scoring time in μs, and the speedups obtained, for different sizes $|\mathcal{T}|$ of the model, and different numbers of leaves Λ. When $|\mathcal{T}|$ gets larger, QS$_{\text{GPU}}$ 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$_{\text{GPU}}$'s performance are shown in the *Run-time configuration* columns. Note that the number of threads for vQS MT is always set to 16, i.e., the number of cores available in our NUMA multiprocessor.

Table 7: Per-document scoring time in μs of QuickScorer, vQS MultiThread, and GPU-QuickScorer on MSN-1, Y!S1, and Istella datasets. Speedups are reported between parentheses – (black) for vQS MultiThread vs. QuickScorer, **(bold black)** for GPU-QuickScorer vs. QuickScorer, and **[bold red]** for GPU-QuickScorer vs. vQS MultiThread.

$\mathcal{D}$	$\|\mathcal{T}\|$	Λ	Method	Run-time configuration		Scoring time (speedup)		
				# threads	τ			
MSN-1	1,000	32	QS	1	–	7.0	(–)	
			vQS MT	16	–	0.2	(35x)	
			QS_{GPU}	128	1,000	**0.19**	**(36.8x)**	**[1.05x]**
		64	QS	1	–	12.4	(–)	
			vQS MT	16	–	0.7	(17.7x)	
			QS_{GPU}	256	1,000	**0.25**	**(49.6x)**	**[3.00x]**
	5,000	32	QS	1	–	33.7	(–)	
			vQS MT	16	–	1.4	(24.1x)	
			QS_{GPU}	512	5,000	**0.44**	**(76.6x)**	**[3.2x]**
		64	QS	1	–	70.7	(–)	
			vQS MT	16	–	4.9	(14.4x)	
			QS_{GPU}	256	1,500	**1.08**	**(65.5x)**	**[4.5x]**
	10,000	32	QS	1	–	74.6	(–)	
			vQS MT	16	–	3.2	(23.3x)	
			QS_{GPU}	512	5,000	**0.86**	**(86.7x)**	**[3.7x]**
		64	QS	1	–	194.8	(–)	
			vQS MT	16	–	12.5	(15.6x)	
			QS_{GPU}	256	1,500	**2.29**	**(85.1x)**	**[5.5x]**
	20,000	32	QS	1	–	183.7	(–)	
			vQS MT	16	–	7.3	(25.2x)	
			QS_{GPU}	512	4,000	**1.79**	**(102.6x)**	**[4.1x]**
		64	QS	1	–	470.5	(–)	
			vQS MT	16	–	46.2	(10.2x)	
			QS_{GPU}	256	1,500	**4.67**	**(100.8x)**	**[9.9x]**
Y!S1	1,000	32	QS	1	–	12.4	(–)	
			vQS MT	16	–	**0.4**	**(31x)**	
			QS_{GPU}	128	1,000	0.85	(14.6x)	**[0.47x]**
		64	QS	1	–	19.2	(–)	
			vQS MT	16	–	1.0	(19.2x)	
			QS_{GPU}	256	1,000	**0.75**	**(25.6x)**	**[1.3x]**
	5,000	32	QS	1	–	43.8	(–)	
			vQS MT	16	–	1.9	(23.1x)	
			QS_{GPU}	512	5,000	**0.94**	**(46.6x)**	**[2x]**
		64	QS	1	–	83.3	(–)	
			vQS MT	16	–	10.2	(8.2x)	
			QS_{GPU}	512	3,000	**1.78**	**(46.8x)**	**[5.7x]**
	10,000	32	QS	1	–	88.7	(–)	
			vQS MT	16	–	4.1	(21.6x)	
			QS_{GPU}	512	5,000	**1.56**	**(56.9x)**	**[2.6x]**
		64	QS	1	–	186.9	(–)	
			vQS MT	16	–	14.6	(12.8x)	
			QS_{GPU}	512	2,000	**3.45**	**(54.2x)**	**[4.2x]**
	20,000	32	QS	1	–	185.1	(–)	
			vQS MT	16	–	8.2	(22.6x)	
			QS_{GPU}	512	5,000	**2.82**	**(65.6x)**	**[2.9x]**
		64	QS	1	–	377.2	(–)	
			vQS MT	16	–	35.1	(10.8x)	
			QS_{GPU}	512	3,000	**5.55**	**(68x)**	**[6.3x]**
Istella	1,000	32	QS	1	–	8.9	(-)	
			vQS MT	16	–	0.3	(29.7x)	
			QS_{GPU}	128	1,000	**0.28**	**(31.2x)**	**[1.07x]**
		64	QS	1	–	13.5	(-)	
			vQS MT	16	–	2.5	(5.4x)	
			QS_{GPU}	256	1,000	**0.37**	**(36.5x)**	**[6.8x]**
	5,000	32	QS	1	–	34.5	(-)	
			vQS MT	16	–	1.2	(28.8x)	
			QS_{GPU}	512	5,000	**0.50**	**(69x)**	**[2.4x]**
		64	QS	1	–	69.8	(-)	
			vQS MT	16	–	3.7	(18.9x)	
			QS_{GPU}	512	3,000	**1.03**	**(67.8x)**	**[3.6x]**
	10,000	32	QS	1	–	71.4	(-)	
			vQS MT	16	–	2.5	(28.6x)	
			QS_{GPU}	512	5,000	**0.96**	**(74.4x)**	**[2.6x]**
		64	QS	1	–	167.4	(-)	
			vQS MT	16	–	8.8	(19x)	
			QS_{GPU}	512	3,000	**2.07**	**(80.8x)**	**[4.3x]**
	20,000	32	QS	1	–	157.2	(-)	
			vQS MT	16	–	7.6	(20.7x)	
			QS_{GPU}	512	5,000	**1.73**	**(90.9x)**	**[4.4x]**
		64	QS	1	–	326.1	(-)	
			vQS MT	16	–	26.1	(12.5x)	
			QS_{GPU}	512	3,000	**3.63**	**(89.8x)**	**[7.2x]**

First, we note that $\mathrm{QS}_{\mathrm{GPU}}$ achieves consistent speedups over the sequential QS – up to 102.6x, 65.6x, and 90.9x on MSN-1, Y!S1, and Istella, respectively. Analogously, $\mathrm{QS}_{\mathrm{GPU}}$ achieves consistent speedups over vQS MT – up to 9.9x, 6.3x, and 7.2x for MSN-1, Y!S1, and Istella, respectively. In general, we observe that $\mathrm{QS}_{\mathrm{GPU}}$ achieves the best results over its competitors when the size of the ensemble gets large in terms of number of trees and leaves of each tree – indeed, 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. For smaller models, e.g., $|\mathcal{T}| = 1,000$ and $\Lambda = 64$, the advantage of GPU over multi-threading+SIMD is very limited, as expected.

8 Conclusions

In this paper we presented and evaluated several strategies to parallelize the traversal of large ensembles of decision trees. We motivated this research with 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. Although we discussed the various parallelization strategies within the LtR framework, they are "general", as the problem of traversing large ensembles of decision trees is agnostic of the specific scenario of use: Web or product search, social media ranking or recommendation, on-line advertisement, classification or regression tasks on big data, etc.

The main advantage of our parallelization strategies is to provide increased throughput, which in turn allows to better satisfy quality-of-service constraints. While the use of larger ensembles favours better accuracy and precision, the reduced scoring times allow to stay within smaller time budgets. The proposed strategies take advantage of the algorithmic framework introduced by QuickScorer, the state-of-the-art in the literature, to leverage different types of parallelism available in modern CPUs and GPUs. We compared the proposed parallel solutions with the sequential version of QuickScorer. The CPU-based parallelization strategies, namely vQS-256 (SIMD) and vQS MT (multi-threading + SIMD), achieved important speedups over QuickScorer: more precisely, vQS-256 obtained speedups up to 3.2× (32 leaves per tree) and 1.8× (64 leaves per tree), while vQS MT achieved speedups up to 14× (32 leaves per tree) and 12.5× (64 leaves per tree). 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 misprediction, and L3 cache-miss rates.

The best results were obtained by the GPU-based parallelization: GPU-QuickScorer achieved relevant performance gains with speedups of 102.6× (32 leaves per tree) and 100.8× (64 leaves per tree) over QuickScorer. 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.

References

1. O. Chapelle and Y. Chang, "Yahoo! learning to rank challenge overview.," *Journal of Machine Learning Research-Proceedings Track*, vol. 14, pp. 1–24, 2011.
2. I. Segalovich, "Machine learning in search quality at Yandex." Presentation at the industry track of the 33rd Annual ACM SIGIR Conference. http://download.yandex.ru/company/presentation/yandex-sigir.ppt, 2010.
3. X. He, J. Pan, O. Jin, T. Xu, B. Liu, T. Xu, Y. Shi, A. Atallah, R. Herbrich, S. Bowers, and J. Q. n. Candela, "Practical lessons from predicting clicks on ads at facebook," in *Proceedings of the Eighth International Workshop on Data Mining for Online Advertising*, (New York, NY, USA), pp. 5:1–5:9, ACM, 2014.
4. D. Sorokina and E. Cantu-Paz, "Amazon search: The joy of ranking products," in *Proceedings of the 39th Annual ACM SIGIR Conference*, pp. 459–460, ACM, 2016.
5. G. Capannini, C. Lucchese, F. M. Nardini, S. Orlando, R. Perego, and N. Tonellotto, "Quality versus efficiency in document scoring with learning-to-rank models," *Information Processing & Management*, 2016. In press.
6. N. Asadi, J. Lin, and A. P. de Vries, "Runtime optimizations for tree-based machine learning models.," *IEEE Transactions on Knowledge and Data Engineering*, vol. 26, no. 9, pp. 2281–2292, 2014.
7. B. B. Cambazoglu, H. Zaragoza, O. Chapelle, J. Chen, C. Liao, Z. Zheng, and J. Degenhardt, "Early exit optimizations for additive machine learned ranking systems," in *Proceedings of the Third International Conference on Web Search and Web Data Mining (WSDM)*, pp. 411–420, ACM, 2010.
8. R.-C. Chen, L. Gallagher, R. Blanco, and J. S. Culpepper, "Efficient cost-aware cascade ranking in multi-stage retrieval," in *Proceedings of the 40th International ACM SIGIR Conference on Research and Development in Information Retrieval*, SIGIR '17, (New York, NY, USA), pp. 445–454, ACM, 2017.
9. N. Tonellotto, C. Macdonald, and I. Ounis, "Efficient and effective retrieval using selective pruning," in *Proceedings of the Sixth ACM International Conference on Web Search and Data Mining*, WSDM '13, (New York, NY, USA), pp. 63–72, ACM, 2013.
10. T.-Y. Liu, "Learning to rank for information retrieval," *Foundations and Trends in Information Retrieval*, vol. 3, no. 3, pp. 225–331, 2009.
11. C. J. Burges, "From ranknet to lambdarank to lambdamart: An overview," Tech. Rep. MSR-TR-2010-82, June 2010.

12. J. H. Friedman, "Greedy function approximation: a gradient boosting machine," *Annals of Statistics*, pp. 1189–1232, 2001.
13. Q. Wu, C. J. Burges, K. M. Svore, and J. Gao, "Adapting boosting for information retrieval measures," *Information Retrieval*, 2010.
14. C. Lucchese, F. M. Nardini, S. Orlando, R. Perego, N. Tonellotto, and R. Venturini, "Quickscorer: A fast algorithm to rank documents with additive ensembles of regression trees," in *Proceedings of the 38th International ACM SIGIR Conference on Research and Development in Information Retrieval (SIGIR)*, pp. 73–82, ACM, 2015.
15. D. Dato, C. Lucchese, F. M. Nardini, S. Orlando, R. Perego, N. Tonellotto, and R. Venturini, "Fast ranking with additive ensembles of oblivious and non-oblivious regression trees," *ACM Trans. Inf. Syst.*, vol. 35, pp. 15:1–15:31, Dec. 2016.
16. X. Tang, X. Jin, and T. Yang, "Cache-conscious runtime optimization for ranking ensembles.," in *Proceedings of the 37th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval (SIGIR)*, pp. 1123–1126, 2014.
17. D. Dato, C. Lucchese, F. M. Nardini, S. Orlando, R. Perego, N. Tonellotto, and R. Venturini, "Fast ranking with additive ensembles of oblivious and non-oblivious regression trees," *ACM Trans. Inf. Syst.*, vol. 35, pp. 15:1–15:31, Dec. 2016.
18. C. Lucchese, F. M. Nardini, S. Orlando, R. Perego, F. Silvestri, and S. Trani, "Post-learning optimization of tree ensembles for efficient ranking," in *Proceedings of the 39th International ACM SIGIR Conference on Research and Development in Information Retrieval*, SIGIR '16, (New York, NY, USA), pp. 949–952, ACM, 2016.
19. J. L. Hennessy and D. A. Patterson, *Computer Architecture, Fifth Edition: A Quantitative Approach*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 5th ed., 2011.
20. W.-m. W. Hwu, *GPU Computing Gems Jade Edition*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1st ed., 2011.
21. C. Lucchese, F. M. Nardini, S. Orlando, R. Perego, N. Tonellotto, and R. Venturini, "Exploiting cpu simd extensions to speed-up document scoring with tree ensembles," in *Proceedings of the 39th International ACM SIGIR Conference on Research and Development in Information Retrieval*, SIGIR '16, (New York, NY, USA), pp. 833–836, ACM, 2016.
22. "OpenMP Application Programming Interface." http://www.nvidia.com.
23. "CUDA C Programmin Guide (ver. 8.0)," 2017. http://openmp.org.
24. H. S. and K. H., "An analytical model for a GPU architecture with memory-level and thread-level parallelism awareness," *ACM SIGARCH Computer Architecture News*, vol. 37, no. 3, pp. 152–163, 2009.
25. V. W. Lee, C. Kim, J. Chhugani, M. Deisher, D. Kim, A. D. Nguyen, N. Satish, M. Smelyanskiy, S. Chennupaty, P. Hammarlund, R. Singhal, and P. Dubey, "Debunking the 100x GPU vs. CPU myth: An evaluation of throughput computing on CPU and GPU," in *Proceedings of the 37th Annual International Symposium on Computer Architecture*, ISCA '10, (New York, NY, USA), pp. 451–460, ACM, 2010.
26. H. J.L. and P. D.A., *Computer Architecture - A Quantitative Approach (5. ed.)*. Morgan Kaufmann, 2012.
27. H. Schulz, B. Waldvogel, R. Sheikh, and S. Behnke, "Curfil: Random forests for image labeling on gpu.," in *VISAPP (2)*, pp. 156–164, Citeseer, 2015.
28. B. Van Essen, C. Macaraeg, M. Gokhale, and R. Prenger, "Accelerating a random forest classifier: Multi-core, gpgpu, or fpga?," in *Proceedings of the 20th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pp. 232–239, IEEE, 2012.
29. D. Yin, Y. Hu, J. Tang, T. D. Jr., M. Zhou, H. Ouyang, J. Chen, C. Kang, H. Deng, C. Nobata, J.-M. Langlois, and Y. Chang, "Ranking relevance in yahoo search," in *Proceedings of the 22nd ACM SIGKDD Conference on Knowledge Discovery & Data Mining (KDD)*, ACM, 2016. In press.
30. G. Capannini, D. Dato, C. Lucchese, M. Mori, F. M. Nardini, S. Orlando, R. Perego, and N. Tonellotto, "Quality versus efficiency in document scoring with learning-to-rank models," *Information Processing and Management*, 2016.