A Fast and Scalable Manycore Implementation for an On-Demand Learning to Rank Method

Mateus F. e Freitas[1], Daniel X. de Sousa[2], Wellington S. Martins[1], Thierson C. Rosa[1], Rodrigo de M. Silva[2], Marcos A. Gonçalves[2]

[1]Instituto de Informática – Universidade Federal de Goiás (UFG)
Caixa Postal 131 – 74.001-970 – Goiânia – GO – Brazil

[2]Departamento de Ciência da Computação (UFMG) Belo Horizonte, Brazil

{mateusfreitas,thierson,wellington}@inf.ufg.br
{danielxs,rmsilva,mgoncalv@}dcc.ufmg.br

Abstract. *Learning to rank (L2R) works by constructing a ranking model from training data so that, given unseen data (query), a somewhat similar ranking is produced. Almost all work in L2R focuses on ranking accuracy leaving performance and scalability overlooked. In this work we present a fast and scalable manycore (GPU) implementation for an on-demand L2R technique that builds ranking models on the fly. Our experiments show that we are able to process a query (build a model and rank) in only a few milliseconds, achieving a speedup of 508x over a serial baseline and 4x over a parallel baseline for the best case. We extend the implementation to work with multiple GPUs, further increasing the speedup over the parallel baseline to approximately x16 when using 4 GPUs.*

1. Introduction

Learning to rank (L2R) is the application of machine learning in the construction of ranking models for information retrieval systems [Liu 2009]. The learned ranking model is then used with unseen items (queries) to produce a permutation that is somewhat similar to rankings in the training data. A well known application of L2R is in commercial web search engines to compute relevance of web pages for a given query. Other applications include translation systems [Duh and Kirchhoff 2008], computational biology [Henneges et al. 2011], and recommender systems [Lv et al. 2011].

Many machine learning algorithms have been used in L2R in order to create ranking functions of great quality, such as Random Forest [Hastie et al. 2009] and Support Vector Machines [Yue et al. 2007]. Most of these works focus on ranking accuracy, without performance (processing time) and scalability concerns. However, in scenarios where the ranking function should be often updated to fit new queries, the time spent in the re-training can be a drawback. In [Chapelle et al. 2011] this issue is mentioned as the flexibility to adapt to future queries that might differ from the training set. Although overlooked in the literature, performance and scalability are important criteria in L2R task and the main goal of this work.

Most current L2R algorithms operate on batch mode and use complex machine learning techniques to produce a single ranking model that is then used for subsequent queries. However, if future queries differ considerably from those used in the training process, the learned model may potentially hurt the retrieval effectiveness. There have

been efforts to consider query differences in the L2R process [Geng et al. 2008]. Some proposals pre-trains a finite number of models, for example using query categorization, clustering, or nearest neighbors, and selects the most suitable when ranking a query. However, there may be a large number of possible models given the extremely large space of queries. An elegant solution to this problem was proposed by Veloso et al. Instead of pre-determined ranking models, their LRAR (Learning to Rank with Association Rules) method generates a query customized model on the fly. The method uncovers patterns in the training data by generating association rules on a demand-driven basis at query-time. Then, the generated rules are used to estimate the relevance of documents in the test set. The method is extremely effective but demands high computational costs, specially when rule sizes increase.

In this paper we propose to use parallel processing to overcome some of the performance problems of the LRAR method. Our proposal, called PROFL (Parallel Rule-based On the Fly Learning to Rank), makes use of different techniques. First, it represents the training dataset using bitmaps as inverted lists. This allows us to compose itemsets by making fast bitwise AND operations like a set intersection. The support of an itemset can also be calculated by counting the numbers of bits set as 1 in the bitmap. Second, we implement a cache scheme to be able to re-use itemsets and their corresponding association rules. A fingerprint function is used to create a unique identifier for an itemset. Thus an itemset is easily mapped to the cache, facilitating its access and preventing its recomputation. Third, we evenly distributed the work (itemsets/rules generation) to be processed in parallel blocks by using the concept of combinadic, that calculates one element of a lexicographical combination. Since the itemsets/rule are derived from combinations of items (itemsets), the range of combinations can be easily divided among the processors. Last, we extend the parallel solution to a multi-GPU environment replicating the inverted lists (bitmaps) and equally dividing the documents among the GPUs.

We conduct our experiments on well-known benchmark datasets: MSLR-WEB10K (from Microsoft Research[1]), and LETOR3 [2]. The experimental results show that our parallel proposal outperforms all baselines evaluated. When processing the LETOR3 dataset, we have achieved a speedup of up to $508x$ over a sequential baseline, and up to $3.8x$ over a parallel baseline for the best case. We have also experimented with a multi-GPU plataform, which further increased the performance, producing a speedup of $15.77x$ for WEB10K over the parallel baseline using 4 GPUs. As we describe in Section 5, the speedup behaves almost linearly to the number of GPUs when the parallel workload is big enough.

The remainder of the paper is organized as follows. In the following section we present the related work. Then, in section 3 we describe the LRAR (Learning to Rank with Association Rules) method. The proposed solution, PROFL (Parallel Rule-based On the Fly Learning to Rank), is introduced in section 4. We then discuss our experimental results in section 5. Finally, we close the paper with conclusions and future works.

2. Related Work

Some research has been done in the area of parallel machine learning [Upadhyaya 2013],

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

[2] http://research.microsoft.com/en-us/um/people/letor/

aiming at speeding up the computation or increasing the size of the datasets being processed. However, almost none of these studies targeted the L2R sub-field of information retrieval. With the growing importance of the subject, some researchers have proposed efficient L2R through the use of parallel processing [Shukla et al. 2012, Wang et al. 2015, Jin et al. 2015, De Sousa et al. 2012]. However, the use of parallelism in L2R has focused on accelerating the training phase of standard solutions, i.e., those based on a batch strategy. The work in [De Sousa et al. 2012] is the only one, as far as we know, that supports on demand learning to rank, similarly to our proposal.

The authors in [De Sousa et al. 2012] propose a parallel version of the LRAR algorithm, denominated as PLRAR, that runs on a GPU. Although producing very competitive ranking effectiveness and good speedups (x127 in relation to LRAR), their approach of one document per thread does not scale well requiring a very large dataset to fully occupy the GPU. On the other hand, our approach of one document per block of threads can fully occupy a GPU for different sizes of datasets. And even when processing a single query, we are able to generate thousands of threads, fully utilizing the GPU. This allowed us to easily extend the solution to a multi-GPU environment, making the system much more scalable. In addition, our proposal implements a cache system that permits the reuse of rules, avoiding unnecessary computation.

3. Learning to Rank with Association Rules

Learning to Rank (L2R) works with a training set $\mathcal{D}$ composed of records in the form $\langle q, d, r\rangle$, where q is a query, d is a document and $r \in \{r_0, ..., r_k\}$ is a relevance value of d to q. A document d is represented as a list of m features $f_1, f_2, ..., f_m$. The set $\mathcal{D}$ is used to learn a ranking function ϕ, which maps the features of a document to a relevance value r. A test set $\mathcal{T}$ is composed of records in the form $\langle q, d, ?\rangle$, where the relevance r is unknown and it will be estimated by the function ϕ.

We give an example below, showing the records that compose the training set $\mathcal{D}$ and its structure. The set is composed of $|\mathcal{D}|$ records, that belongs to j queries, and each query has a disjoint subset of $\mathcal{D}$. In this example, the query of id 1 has a records in total, and the last query begins at the bth record and ends at the last record $|\mathcal{D}|$.

$$
\begin{aligned}
q:1,\ d_1 \ &: [PageRank(d_1),\ c(q_t, d_1),\ idf(q_t),\ ...,\ f_m],\ r = 1 \\
&\ \ \vdots \\
q:1,\ d_a \ &: [PageRank(d_a),\ c(q_t, d_a),\ idf(q_t),\ ...,\ f_m],\ r = 0 \\
&\ \ \vdots \\
q:j,\ d_b \ &: [PageRank(d_b),\ c(q_t, d_b),\ idf(q_t),\ ...,\ f_m],\ r = 0 \\
&\ \ \vdots \\
q:j,\ d_{|\mathcal{D}|} \ &: [PageRank(d_{|\mathcal{D}|}),\ c(q_t, d_{|\mathcal{D}|}),\ idf(q_t),\ ...,\ f_m],\ r = 0
\end{aligned}
$$

For each record, there is a document represented by its features and a relevance value associated with it. The features from the document vector range from simple formulas that use frequency of terms, to proprietary ranking functions like PageRank [Page et al. 1999]. The PageRank function gives a ranking number to a web page based on the number of links that point to that page. For the LETOR dataset, some features use

$c(q_t, d)$ and $idf(q_t)$ [Qin et al. 2010]. $c(q_t, d)$ denotes the number of occurrences of query term q_t in document d, and $idf(q_t)$ is the inverse document frequency (IDF) of that term. The IDF gives a greater weight to terms that appear rarely, and smaller weight to frequent terms. The complete definition of the features can be found at LETOR's specification.

In [Veloso et al. 2008] the combination of *Learning to Rank with Association Rules*, LRAR for short, creates a method capable of providing a flexible ranking for unseen samples at query time[3]. We mean flexible, because the rules are created from each new document, therefore, enabling an on-demand solution that produces only the needed association rules to create a specific ranking function.

The function ϕ in LRAR is created using a set of association rules of the form $\mathcal{X} \xrightarrow{\theta} r_i$, where $\mathcal{X}$ (also denominated of itemset) is a combination of several features from the training set. For example, $\mathcal{X}_j^z = PageRank(d_j) \wedge c(q_t, d_j) \wedge idf(q_t)$ refers to rule z, document j, and features composed by the functions $PageRank$, c and idf regarding d_j[4]. In addition, r_i is the relevance value and θ is the *confidence*, the conditional probability of the consequent (relevance value) to occur given the antecedent (itemset). In other words, the confidence means the chance of $\mathcal{X}$ (created from an unseen document) appear in documents in the training set which have the relevance value r_i. Each rule has also a support, which is the frequency of both $\mathcal{X}$ and r_i in $\mathcal{D}$ [Agrawal et al. 1993].

In LRAR, the association rules are used to estimate the document relevance. The support is referred as $\sigma(\mathcal{X} \longrightarrow r_i)$, and the confidence is referred as $\theta(\mathcal{X} \longrightarrow r_i)$. The rule is valid when the thresholds σ_{min} and θ_{min} are satisfied.

Eq.1, from [Veloso et al. 2008], is used to obtain the score of the relationship between the document and a relevance value. The equation sums the confidence of all rules of the *projected rule set*, $\mathcal{R}_d$. A *projected rule set* has the rules obtained from the unseen document matched against the training documents restrict to a relevance value r_i. Since a lot of rules can be generated when combining three or more features in the antecedent, the log function is used to smooth the relationship.

$$s(d, r_i) = \frac{\sum_{(\mathcal{X} \to r_i) \in \mathcal{R}_d} \theta(\mathcal{X} \to r_i)}{\log |\mathcal{R}_d|} \tag{1}$$

The ϕ function of LRAR, is defined by the Eq. 2, which is a normalized linear combination of the scores with each relevance value. After each document d_j has its relevance estimated, the final ranking is in the decreasing order of $\phi(d_j)$.

$$\phi(d) = \sum_{i=0}^{k} (r_i \times \frac{s(d, r_i)}{\sum_{j=0}^{k} s(d, r_j)}) \tag{2}$$

The LRAR algorithm has shown relevant ranking quality against other L2R algorithms, as described in experimental results of [Veloso et al. 2008]. The good results were also confirmed in [Silva et al. 2011], which applied an active learning method to reduce the original training set and improve the processing time, without damaging the

[3]Hereafter, we refer to test samples as unseen documents, as they are obtained only in scoring phase.

[4]For simplicity of notation, we omit the number of the document and the rule of $\mathcal{X}$ when necessary.

ranking quality. The dataset reduction was over 90%. Given the importance of this dataset reduction it has been used in [De Sousa et al. 2012] and in this work.

4. Parallel Rule-based On the Fly Learning to Rank (PROFL)

Our main contribution in this work is a fast and scalable implementation of LRAR, which is denominated PROFL, a Parallel Rule-based On the Fly Learning to Rank. Next, we describe the main concepts and techniques used in our proposal, as well as a detailed description of the proposed parallel implementation.

4.1. Thread Block Approach

The thread block approach exploits the parallelism by block, that is, many threads contribute to performing the job of a block, and many blocks work in parallel. Following this strategy, each unseen document is assigned to a block, and thus, many threads process a document in a parallel fashion. In order to provide a balanced workload between the threads, each thread receives a similar number of rules to be created. The number of rules depends on the number of items (features) from the unseen document (defined by m), the number of threads in a block, and also the k value, which is a user parameter defining the maximum rule size. First, all rules are computed, each one receiving an index z, with $0 \leq z < \binom{m}{k}$. Then the rules are assigned to the threads, with each thread getting $\left\lceil \frac{\binom{m}{k}}{max\_threads} \right\rceil$ rules.

We have chosen the thread block approach because of its scalability in the GPU, enforcing the full occupancy of the hardware. Since usually there are hundreds or thousands of documents associated with a query, we can launch plenty of blocks, each one containing many threads. This way, the memory latency, which is a serious issue in GPUs, can be more efficiently hidden. An alternative approach, used in PLRAR, is to assign a document to a thread and not to a block of threads. This has the disadvantage of poor GPU occupancy since a relatively small number of threads will be active.

4.2. Combinadic

As we described before, the combinadic [McCaffrey 2004] provides an element from a set of distinct combinations of integer values, when given an index. The combinadic is the key point of our thread block proposal and its scalability. It allows each thread to obtain the proper subset of rules to be created in a fast manner.

The Algorithm 1 [Buckles and Lybanon 1977] describes the combinadic, which receives an index value z, the number of elements n, and the combination size k, where $0 \leq z < \binom{n}{k}$. It provides an element of a lexicographical combination of integer values, the itemset[5]. The algorithm maps the value z to one subset of size k of $\mathcal{X} = \{0, 1, 2, ..., (n-1)\}$, by solving $z = \binom{c_1}{k} + \binom{c_2}{k-1} + ... + \binom{c_k}{1}$, restrict to $n > c_1 > c_2 > ... > c_k$. For this, the algorithm searches for the y where the current binomial $\binom{y}{k}$ is less or equal than the current z, in a descending order of y. Each y that satisfies that inequality is added to a list, which is returned by the end of the algorithm.

With the combinadic algorithm each thread is assigned only an index value in order to obtain the rules. As a result, it is not necessary to compute the previous combina-

[5]Recalling that a rule is formed by the itemset (the antecedent) and the relevance value (the consequent).

Algorithm 1 Combinadic – Generation of a Subset of Integer values.

Require: Index z, Number of elements n, combination size k
Ensure: A lexicographical combination
Let $comb$ be a list
for $y = (n-1)$ **to** 1 **do**
  if $\binom{y}{k} \leq z$ **then**
    $comb$.add(y)
    $z = z - \binom{y}{k}$
    $k = k - 1$
Return $comb$

tions until it reaches the current index. That way, each thread can independently compute its own rules as soon as the index is assigned to it.

4.3. Caching Itemsets

In PROFL, the rules are created from each unseen document. Considering the similarity between these documents, there is a great demand for same rules. Thus, it seems almost obvious to take advantage of a cache strategy, commonly used in data processing. However, memory accesses occurs frequently in a cache, and the GPU's global memory has high latency, which is its drawback. To handle this issue, we have implemented a caching strategy that lowers the number of accesses in a hash table and allows each thread to work in an independent parallel processing.

Before a new rule is created in PROFL, a thread searches for the itemset in the hash table. To perform this, we have implemented a fingerprint function developed by [Atreas and Karanikas 2007], which maps a set of integer values, an itemset, to a floating point number. The Eq. 3 describe the fingerprint function.

$$fingerprint(\mathcal{X}) = \sum_{i=1}^{n} \frac{f_i}{p_i} \tag{3}$$

where $\mathcal{X} = \{f_1, f_2, ..., f_n\}$ and p_i is the ith prime number greater than the max index value for a f_i.

In order to map the fingerprint output function to the hash table, the floating point number is transformed by taking its 64 bits as a 64 bits integer L. Then this long integer can be used with a simple hash using a modulo operation with the table size, $L \ \% \ HASH\_SIZE$. The hash table stores the itemset bitmap, and also its support for each relevance value. When the itemset is found, only the support values are returned, which is necessary to compute the confidence value.

One important advantage of this fingerprint function is its fast processing to obtain the index value from an itemset. Thus, if an itemset is not found, PROFL searches for the subset with the $k-1$ prefix and calculates its fingerprint to access the hash table. Hence, a new itemset is created taking into account intersections that were already done.

We use an array to store the hash table, and the linear probing with open addressing as the strategy for resolving the hash collisions. By applying our hashing strategy, each thread was given an independent and parallel access pattern. Though this eventually lead to decreasing the amount of coalesced memory access of some threads, the time saved when creating a new itemset have improved the overall processing time, as we describe in

section 5. The fingerprint function applied in PROFL has not been studied before in this context, showing a promising cache implementation in other data processing applications.

4.4. Data Representation

In order to improve the parallelism in GPU, the data representation is a critical issue, because it defines the memory access pattern. Thus, we have represented the training set using bitmaps as inverted lists. A bitmap is created for each item (feature), and the ith bit indicates whether the item occurs or not at the ith document. This is a compact method to implement an inverted list, allowing to execute fast intersections between bitmaps with bitwise AND operations. Also, the itemset support can be calculated by counting the number of bits set as 1.

The PROFL improves the bitmap operation by casting the bitmap's elements into a vectorized integer type (`int4`) , forcing the compiler to produce 128 bytes load and store instructions. Thus, increasing the memory throughput of the GPU [Luitjens 2013].

In addition, we also have used the GPU shared memory to store the bitmap of each relevance value and the global memory to store the cache's hash table. Since the bitmap of a relevance value is often used to create a rule, and the latency in shared memory is lower than the global memory, this setting has a good impact in performance.

4.5. The PROFL Algorithm

In this section, we describe the PROFL algorithm, our parallel proposal to LRAR method, taking into account the explanation of thread block approach, combinadic algorithm, parallel cache, and the bitmap representation described in the previous subsections.

The PROFL begins in CPU, receiving as input the training documents from the reduced training set (as explained in Section 3), the unseen documents, the max rule size, and the confidence and support thresholds. Each document is represented as a list of m features, which are transformed in itemsets of size 1 (only one feature). The bitmap representation of the training set is processed in CPU, afterwards, the data is allocated in GPU memory, and each unseen document d_i is processed in Algorithm 2.

Algorithm 2 describes the GPU (kernel) processing of PROFL. In line 1 the size of an itemset is defined. In lines 2-3, the disjoint index for rules is created, and the combinadic algorithm is called in line 5 to obtain the itemset of the rule. In line 6 the algorithm applies the fingerprint function (described in Section 4.3) to check if the itemset was already created. If found in the hash table, the support of the itemset is requested in line 8. If not, the algorithm searches for a smaller itemset in the hash table, the prefix of size s of the original itemset, and inserts the smaller itemset in L, in lines 10-15. PROFL searches for a smaller itemset in order to use some bitmaps that were already created. Line 16 creates a new itemset by using the prefix L. Lines 18-19 check if support and confidence values of the rules are greater than the thresholds. If it is true, then the confidence values are summed from several rules. For this, we have applied a parallel reduction in a shared memory array, as described in [Harris et al. 2007]. Finally, in line 21, the ranking can be computed by using the retrieved scores of each document.

Besides our proposal of PROFL using one GPU, we have also evaluated a multi-GPU version. The Algorithm 3 shows the multi-GPU implementation of PROFL, where

Algorithm 2 Kernel Execution of PROFL – Parallel Rule-based On the Fly Learning to Rank.

Require: Training Set Bitmap, Unseen document with n items, max_rule_size, σ_{min} and θ_{min} thresholds
Ensure: The ranking score
{The following code is executed by threads identified by t.}
for $k = 1$ **to** max_rule_size **do**
  #rules = $\lceil \binom{m}{k} / max\_threads \rceil$
  $ruleIndexes =$ **from** $t * \#rules$ **to** $min((t+1) \cdot \#rules, \binom{m}{k})$
  for all ruleIndexes z **do**
    $itemset$ = Combinadic(z, n, k)
    h = fingerprint($itemset$)
    if Cache(h) is true **then**
      p = Cache(h)
    else
      $L = \emptyset$
      for $s = (k-1)$ **to** 1 **do**
        h = fingerprint(prefix(itemset, s))
        if Cache(h) is true **then**
          include Cache(h) in L
          break
      p = create_itemset(itemset, L)
      insert itemset into cache
    for all relevance value rel **do**
      check $\sigma(p \longrightarrow rel) \geq \sigma_{min}$ and $\theta(p \longrightarrow rel) \geq \theta_{min}$
  Sum the confidence value from several rules
compute the ranking for $document_i$ from all generated rules

Algorithm 3 CPU Execution of Multi-GPU PROFL.

Require: Training Set Bitmap, Unseen Documents, #GPUs, max_rule_size, σ_{min} and θ_{min} threshold
Ensure: The ranking score
for $g = 0$ **to** #GPUs **parallel do**
  set_gpu_device(g)
  Copy training set bitmap to GPU g
  $\#docs = \lceil \frac{N}{\#GPUs} \rceil$
  $unseenDocumentSet$ = from $g \cdot \#docs$ to $min((g+1) \cdot \#docs, N)$
  Call Algorithm PROFL(Training Set Bitmap, unseenDocumentSet, max_rule_size, σ_{min}, θ_{min})

each GPU is controlled by a CPU thread. The algorithm begins by associating the CPU's threads with the GPUs, and copying the training set bitmap to all GPUs memories. Afterwards, the unseen documents are split into the available GPUs, and each GPU calls Algorithm 2. The for iteration performs in parallel by using the API OpenMP[6] in order to manage the parallel execution of multiple GPUs.

5. Experiment Evaluation

In this section, we describe a set of experiments performed to evaluate our parallel proposal. We first describe the hardware where the experiments took place, as well as the datasets used to benchmark our implementation. We finalize this section discussing our results against the LRAR and PLRAR, a serial and a parallel baseline, respectively.[7]

Our experimental results were conducted with the following CPU settings: CentOs 7.2.1511 64-bits operating system, an Intel Xeon E5-2620 2GHz and 16GB of ECC RAM. We used a GPU setting composed of four GeForce Zotac Nvidia GTX Titan Black, with 6GB of RAM and 2,880 CUDA cores each, totaling 11,520 cores.

[6]Available in `http://openmp.org/wp/openmp-compilers/`
[7]These baselines were described in sections 2 and 3.

Table 1. Average number of documents in each fold of the datasets.

	Collections						
	TD2003	HP2003	NP2003	TD2004	HP2004	NP2004	WEB10K
Queries	10	30	30	15	15	15	2000
Test Set	9,812	29,521	29,731	14,834	14,882	14,767	240,039
Training Set	29,435	88,564	89,195	44,488	44,646	44,301	720,116
Red. Tr. Set	671	1091	995	593	859	658	7337

As for the software, the CPU code was compiled with GCC 4.8.5, while the GPU code was compiled with CUDA Toolkit 7.5[8]. All the codes targeted the native architecture and had the O3 optimization flag set. All executions were performed without concurrent process. The reported results are average of 10 independent runs. The standard deviation and confidence values were negligible being below 1%. Each GPU block was executed with 256 threads, because of the shared memory usage in our implementation. With 256 threads, there can be three concurrent blocks, instead of one with 512 threads. The size used for the hash table was 4.5GB, so that 27 million free positions could be used for the smaller datasets (LETOR3) and 4.6 million for the biggest (WEB10k). The remaining memory was necessary for other auxiliary structures and the system. We have used the values 1 and 0.001 for the support and confidence threshold, respectively.[9]

The Table 1 describes the datasets used in our work. Datasets from LETOR3 (TD2003, TD2004, HP2003, HP2004, NP2003 and NP2004) and WEB10K, which are freely available. We have applied a 5-fold cross-validation procedure [Hastie et al. 2009], following the Machine Learning area to ensure confident results. Cross-validation splits the entire collection into 5 parts, and assigns three parts as a training set, one as an unseen set and the remaining one as a validation set. The latter was used by the baselines to find the support and confidence parameters. Each fold receives a different assignment.

As already mentioned in section 3, we have used the reduced version of the dataset for all executions, applying the active learning method defined in [Silva et al. 2011]. This reduction was applied to each fold. It is worth noting that this is the first time that the WEB10K dataset was used in an association rules based ranking task.

5.1. Results

We start by confirming in Tables 2 and 3, that our parallel proposal outperforms all other baselines. Even using only one GPU, we are able to improve the processing time over 508x and 4x, against the serial and parallel implementation, respectively. Tables 2 and 3 show the performance for our proposal PROFL, as well as LRAR and PLRAR. Both tables demonstrate the results for rules of size 3 and 4[10]. The time to create the training set bitmap is not included in these tables, which are over 0.15 and 2.0 seconds for LETOR and WEB10K, respectively.

Table 2 describes the results using only one GPU. For instance, the time to PROFL process a query is 0.2 second, while LRAR spends up to 100 seconds in a single query, which means a speedup of $500x$ for the best case. This result shows how appropriate our

[8]Available in `https://developer.nvidia.com/cuda-toolkit-archive`

[9]These values were also used in LRAR and PLRAR baselines.

[10]The maximum rule size used in our experiments is 4, since rules of size 5 do not increase the effectivity [De Sousa et al. 2012].

Table 2. Execution time (seconds). Bold represent best results.

	Execution time for rules of size 3				
	Time for 1 query			Speedup	
	PROFL	PLRAR	LRAR	PLRAR	LRAR
TD2003	**0.09**	0.13	2.43	1.39	26.11
HP2003	**0.08**	0.10	2.38	1.32	31.52
NP2003	**0.08**	0.10	2.36	1.31	31.26
TD2004	**0.09**	0.12	2.83	1.31	30.28
HP2004	**0.10**	0.13	2.81	1.32	29.38
NP2004	**0.09**	0.12	2.70	1.31	28.83
WEB10K	**0.02**	0.15	2.60	7.72	134.52
	Execution time for rules of size 4				
	PROFL	PLRAR	LRAR	PLRAR	LRAR
TD2003	**0.18**	0.61	81.66	3.35	445.28
HP2003	**0.16**	0.61	81.48	3.81	507.96
NP2003	**0.16**	0.60	80.36	3.71	498.09
TD2004	**0.21**	0.60	102.48	2.93	496.92
HP2004	**0.21**	0.67	106.27	3.20	506.31
NP2004	**0.21**	0.64	101.19	3.09	491.42
WEB10K	**1.95**	7.94	*	4.07	*

Table 3. Speedup for PROFL with multiple GPUs.

	Execution time for rules of size 3					
	Time for 1 query		Speedup-2		Speedup-4	
	PROFL-2	PROFL-4	PLRAR	LRAR	PLRAR	LRAR
TD2003	0.09	0.09	1.42	26.66	1.44	26.97
HP2003	0.07	0.07	1.35	32.21	1.38	32.87
NP2003	0.07	0.07	1.34	32.00	1.37	32.58
TD2004	0.09	0.09	1.34	30.98	1.37	31.56
HP2004	0.09	0.09	1.35	30.00	1.37	30.56
NP2004	0.09	0.09	1.35	29.59	1.37	30.05
WEB10K	0.02	0.02	7.97	138.92	8.13	138.92
	Execution time for rules of size 4					
	PROFL-2	PROFL-4	PLRAR	LRAR	PLRAR	LRAR
TD2003	0.14	0.12	4.32	574.31	5.20	691.68
HP2003	0.12	0.10	5.20	692.27	6.37	849.18
NP2003	0.12	0.10	5.07	680.89	6.20	832.40
TD2004	0.15	0.12	4.02	681.76	4.97	842.28
HP2004	0.15	0.12	4.41	696.54	5.44	860.19
NP2004	0.15	0.12	4.25	674.76	5.23	831.63
WEB10K	0.99	0.50	8.04	*	15.77	*

proposal is to an on-demand processing. Even though PROFL obtained the best performance against the baselines, there is only a little difference between it and PLRAR when using rules of size 3 for LETOR3 datasets. This is because there are few combinations to be processed. Otherwise, considering the huge WEB10K dataset and rules of size 4 for all datasets, one can see a strong performance of PROFL, showing the advantage of our improvements with the thread block approach, use of parallel cache, and the vectorized bitmap representation.

In the WEB10K dataset there is a greater number of features and documents.

Thus, as shown in Table 2, LRAR suffers more with the increase of features, so it takes longer to process a single document, making PROFL up to $134x$ faster. When processing rules of size 4, the speedup over PLRAR is $4x$ for the best case. Since PLRAR is a GPU implementation, our proposal shows strong evidence of a better performance. When executing LRAR with rules of size 4, only 10% of the first fold was processed after 20 hours, making it unable to process WEB10K.

Table 3 shows the results using our multi-GPU PROFL, where PROFL–2 and PROFL–4 describes the number of GPUs. For multi-GPUs, the improvement is obtained with rules of size 4, and the time decreases with more GPUs. The speedup of PROFL in LETOR datasets against PLRAR is up to $5.2x$ and $6.37x$ for 2 and 4 GPUs respectively. Likewise, the speedup against LRAR increases in a few more hundreds, reaching up to $696x$ and $860x$, for 2 and 4 GPUs. The thread block approach also helps the scalability of multiple GPUs, when there are fewer documents for each query. Taking into account the WEB10K, where the generation of rules demands more processing, the PROFL obtained a linear speedup, achieving $8x$ and $15.77x$, for 2 and 4 GPUs, when using rules of size 4. Considering rules of size 3, the multi-GPUs experiments do not show any improvements This occurs because of the overhead of the PCIe to access many GPUs, the small number of intersections and the input time that dominates the processing time.

6. Conclusion

In this paper we have described a new parallel implementation of LRAR, denominated as PROFL. Our experimental evaluation shows that PROFL outperforms the baselines. For instance, using a single GPU the speedup of PROFL reached over $500x$ and $4x$ faster than serial and GPU baselines, respectively, for the best case.

The faster processing of PROFL are due to i) the proposal of a parallel cache with fingerprint function, providing an independent and parallel hash accessing, ii) the thread block approach that was made possible by using the combinadic method, that allows to distribute the work evenly among the threads, and also iii) our vectorized bitmap representation, which increases the memory throughput of the GPU and allows fast intersections that are done with bitwise operations. In addition, we have implemented a multi-GPU PROFL, which shows a linear scalability when more GPUs are used.

Concerning the on-demand aspect, which needs to compute new models on the fly, we have shown that PROFL is a good candidate to improve the processing time. This result opens up several research possibilities as future works. For instance, we plan to apply our itemset computation to find out people's behavior on social-media, especially in social events where statistical properties can change over time, without a predefined way. In this situation, the prediction for new information are useful only if obtained in a short time.

References

[Agrawal et al. 1993] Agrawal, R., Imieliński, T., and Swami, A. (1993). Mining association rules between sets of items in large databases. *ACM SIGMOD Record*.

[Atreas and Karanikas 2007] Atreas, N. and Karanikas, C. (2007). A fast pattern matching algorithm based on prime numbers and hashing approximation. *NATO Security through Science Series -D: Information and Communication Security*.

[Buckles and Lybanon 1977] Buckles, B. P. and Lybanon, M. (1977). Algorithm 515: Generation of a vector from the lexicographical index [g6]. *TOMS*.

[Chapelle et al. 2011] Chapelle, O., Chang, Y., and Liu, T.-Y. (2011). Future directions in learning to rank. In *Yahoo! Learning to Rank Challenge*.

[De Sousa et al. 2012] De Sousa, D. X., Rosa, T. C., Martins, W. S., Silva, R., and Gonçalves, M. A. (2012). Improving on-demand learning to rank through parallelism. In *WISE*.

[Duh and Kirchhoff 2008] Duh, K. and Kirchhoff, K. (2008). Learning to rank with partially-labeled data. In *SIGIR*. ACM.

[Geng et al. 2008] Geng, X., Liu, T.-Y., Qin, T., Arnold, A., Li, H., and Shum, H.-Y. (2008). Query dependent ranking using k-nearest neighbor. In *SIGIR*. ACM.

[Harris et al. 2007] Harris, M. et al. (2007). Optimizing parallel reduction in cuda. *NVIDIA Developer Technology*.

[Hastie et al. 2009] Hastie, T., Tibshirani, R., and Friedman, R. (2009). *The Elements of Statistical Learning*.

[Henneges et al. 2011] Henneges, C., Hinselmann, G., Jung, S., Madlung, J., Schütz, W., Nordheim, A., and Zell, A. (2011). Ranking methods for the prediction of frequent top scoring peptides from proteomics data. *Journal of Proteomics & Bioinformatics*.

[Jin et al. 2015] Jin, J., Cai, X., Lai, G., and Lin, X. (2015). Gpu-accelerated parallel algorithms for linear ranksvm. *The Journal of Supercomputing*.

[Liu 2009] Liu, T.-Y. (2009). Learning to rank for information retrieval. *Foundations and Trends in Information Retrieval*.

[Luitjens 2013] Luitjens, J. (2013). Cuda pro tip: Increase performance with vectorized memory access. `https://devblogs.nvidia.com/parallelforall/cuda-pro-tip-increase-performance-with-vectorized-memory-access/`, acessed in june 2016.

[Lv et al. 2011] Lv, Y., Moon, T., Kolari, P., Zheng, Z., Wang, X., and Chang, Y. (2011). Learning to model relatedness for news recommendation. In *WWW*. ACM.

[McCaffrey 2004] McCaffrey, J. (2004). Generating the mth Lexicographical Element of a Mathematical Combination. `http://msdn.microsoft.com/en-us/library/aa289166(v=vs.71).aspx`, acessed in june 2016.

[Page et al. 1999] Page, L., Brin, S., Motwani, R., and Winograd, T. (1999). The pagerank citation ranking: bringing order to the web.

[Qin et al. 2010] Qin, T., Liu, T.-Y., Xu, J., and Li, H. (2010). Letor: A benchmark collection for research on learning to rank for information retrieval. *Information Retrieval*.

[Shukla et al. 2012] Shukla, S., Lease, M., and Tewari, A. (2012). Parallelizing listnet training using spark. In *SIGIR*. ACM.

[Silva et al. 2011] Silva, R., Gonçalves, M. A., and Veloso, A. (2011). Rule-based active sampling for learning to rank. In *ECML PKDD*. Springer.

[Upadhyaya 2013] Upadhyaya, S. R. (2013). Parallel approaches to machine learning—a comprehensive survey. *JPDC*.

[Veloso et al. 2008] Veloso, A. A., Almeida, H. M., Gonçalves, M. A., and Meira Jr, W. (2008). Learning to rank at query-time using association rules. In *SIGIR*. ACM.

[Wang et al. 2015] Wang, S., Wu, Y., Gao, B. J., Wang, K., Lauw, H. W., and Ma, J. (2015). A cooperative coevolution framework for parallel learning to rank. *TKDE*.

[Yue et al. 2007] Yue, Y., Finley, T., Radlinski, F., and Joachims, T. (2007). A support vector method for optimizing average precision. In *SIGIR*. ACM.