FPGA Acceleration of RankBoost in Web Search Engines

NING-YI XU, XIONG-FEI CAI, RUI GAO, LEI ZHANG, and FENG-HSIUNG HSU
Microsoft Research Asia

Search relevance is a key measurement for the usefulness of search engines. Shift of search relevance among search engines can easily change a search company's market cap by tens of billions of dollars. With the ever-increasing scale of the Web, machine learning technologies have become important tools to improve search relevance ranking. RankBoost is a promising algorithm in this area, but it is not widely used due to its long training time. To reduce the computation time for RankBoost, we designed a FPGA-based accelerator system and its upgraded version. The accelerator, plugged into a commodity PC, increased the training speed on MSN search engine data up to 1800x compared to the original software implementation on a server. The proposed accelerator has been successfully used by researchers in the search relevance ranking.

Categories and Subject Descriptors: B.7.1 [**Integrated Circuits**]: Types and Design Styles—*Algorithms implemented in hardware*

General Terms: Algorithms, Performance, Design

Additional Key Words and Phrases: FPGA, hardware acceleration

ACM Reference Format:
Xu, N.-Y., Cai, X.-F., Gao, R., Zhang, L., and Hsu F.-H. 2009. FPGA acceleration of RankBoost in Web search engines. ACM Trans. Reconfig. Techn. Syst. 1, 4, Article 19 (January 2009), 19 pages. DOI = 10.1145/1462586.1462588. http://doi.acm.org/10.1145/1462586.1462588.

1. INTRODUCTION

Web search–based ad services have become a huge business in the last few years with billions of annual earnings and more than a hundred billion dollars of total generated market cap for search engine companies. For a search engine, the key to attract more users and obtain larger market share is its search relevance [Fan et al. 2004], which is determined by the ranking function that ranks resultant documents according to their similarities to the input query.

Author's address: N.-Y. Xu, Platforms and Devices Center, Microsoft Research Asia, Sigma Center, No. 49 Zhichun Road, Haidian, Beijing, 100190, China.

Information retrieval (IR) researchers have studied search relevance problem for many decades. Representative methods include Boolean model, vector space model, probabilistic model, and the language model [Baeza-Yates and Ribeiro-Neto 1999]. Earlier search engines were mainly based on such IR algorithms. When the internet bubble crashed in 2000, most observers believed that the search business was a minor business and search relevance was not an important problem. But just a year before, in 1999, Google proposed the PageRank algorithm to measure the importance of Web pages [Brin and Page 1998] and created a quantum leap in search relevance for search engines. This leap created a rapid decline of user populations for the then-existing search engines and proved beyond any doubt the importance of search relevance. Many factors affect the ranking function for search relevance, such as page content, title, anchor, URL, spam, and page freshness. It is extremely difficult to manually tune ranking function parameters to combine these factors in an ever-growing Web-scale system. Alternatively, to solve this problem, machine learning algorithms have been applied to automatically learn complex ranking functions from large-scale training sets.

Earlier algorithms for ranking function learning include Polynomial-based regression [Fuhr 1989], Genetic Programming [Fan et al. 2004], RankSVM [Joachims 2002] and classification-based SVM [Nallapati 2004]. However, all these algorithms have only been evaluated on small-scale datasets due to their high computational cost. RankNet [Burges et al. 2005] was the first work that was evaluated on a commercial search engine. Motivated by RankNet, we revisited more machine learning algorithms. RankBoost [Freund et al. 2003] was identified to be a promising algorithm, because it has comparable relevance performance with RankNet and a faster ranking function, even though it has higher computation cost in the learning process. To learn a ranking function for a real search engine, large-scale training sets are usually used for better generalization, and numerous runs are required to exploit best results on frequently updated datasets. Thus the computational performance, measured by the time to obtain an acceptable model over a large dataset, is critical to the ranking function learning algorithm. With an initial software implementation, a typical run to obtain a model on MSN search engine data for RankBoost still takes two days, which is too slow for training and tuning of the ranking function. To further reduce the computation time of RankBoost, we designed an FPGA-based accelerator system.

As an alternative to multicore CPUs and GPUs, field-programmable gate-arrays (FPGAs) can be used as customized computing engines for accelerating a wide range of applications. High-end FPGAs can fully utilize bit-level parallelism and provide a very high memory access bandwidth. For example, an Altera Stratix-II FPGA, which is utilized in our accelerator, contains millions of gates, and provides Tbps accessing bandwidth to several megabytes of on-chip memory.[1] Due to its great design flexibility and its high performance

[1]http://www.altera.com

which is close to Application Specific Integrated Circuits (ASICs), FPGA has been recognized to be highly competitive with general purpose CPUs for high performance computing on a large number of applications [Underwood and Hemmert 2004; El-Ghazawi et al. 2006].

In this article, we propose FAR: FPGA-based Accelerator for RankBoost. We first develop an approximation to the RankBoost algorithm, which reduces the scale of computation and storage from $O(N^2)$ to $O(N)$ and speeds up the software by 3.4 times. We then map the algorithm and related data structures to our customized FPGA-based accelerator platform. An SIMD (Single Instruction Multiple Data) architecture with multiple processing engines (PEs) is implemented in the FPGA. In a typical experimental run on the MSN search engine data, training time is reduced from 46 hours to 16 minutes on a commodity PC equipped with the first implementation of FAR (called FAR1). This is an achievement of 170.6x improvement in performance [Xu et al. 2007]. Several FAR1 systems are then successfully used in research work in Microsoft Research Asia. Per request from the users, one year later, we implement the second version of FAR (called FAR2) on a much powerful FPGA accelerator board, and achieve an acceleration rate of up to 1800x.

The main contribution of this work includes (i) extension of the RankBoost algorithm to improve Web search relevance ranking (ii) efficient computation method for RankBoost, (iii) the mapping of the algorithm to two generations of efficient hardware accelerator architecture, and (iv) demonstration of its superior performance and its usage.

The remainder of this article is organized as follows. Section 2 describes the RankBoost algorithm for search relevance and discusses the bottleneck in the software implementation. Section 3 presents the design of FAR. Section 4 describes the performance model and discusses experimental results. Section 5 concludes the article and discusses future work.

2. APPLICATION: RANKBOOST FOR WEB SEARCH RELEVANCE

To better understand the application, in this section we give a brief introduction to RankBoost and describe how RankBoost is used to learn ranking functions in a Web search engine. In addition, the computational complexity is also discussed.

RankBoost is a variant of AdaBoost, which was proposed by Freund and Schapire 1995. For a typical classification problem, the AdaBoost training process tries to build a strong classifier by combining a set of weak classifiers, each of which is only moderately accurate. AdaBoost has been successfully applied to many practical problems (including image classification [Liu et al. 2004], face detection [Viola and Jones 2001], and object detection [Laptev 2006]) and demonstrated effective and efficient performance. The proposed FPGA-based RankBoost accelerator could be easily extended to support such applications to achieve similar performance improvement.

For detailed background and theoretical analysis of boosting algorithms, please refer to Schapire [1999, 2001], Freund et al. [2003], and Iyer et al. [2000].

Algorithm: RankBoost
Initialize: initial distribution D_1 over $\chi \times \chi$
Do for $t = 1, ..., T$:
(1) Train **WeakLearn** using distribution D_t.
(2) **WeakLearn** returns a weak hypothesis h_t.
(3) Choose $\alpha_t \in R$
(4) Update weights: for $\forall(x_0, x_1)$,
$D_{t+1}(x_0, x_1) = \frac{D_t(x_0,x_1)exp(-\alpha_t(h_t(x_0)-h_t(x_1)))}{Z_t}$
where Z_t is the normalization factor:
$Z_t = \sum_{x_0,x_1} D_t(x_0, x_1)exp(-\alpha_t(h_t(x_0) - h_t(x_1)))$.
Output: the final hypothesis:
$H(x) = \sum_{t=1}^{T} \alpha_t h_t$.

Fig. 1. The original RankBoost algorithm.

2.1 RankBoost for Standard Ranking Problem

In a standard ranking problem, the goal is to find a ranking function to order the given set of objects. Such an object is denoted as an instance x in a domain (or instance space) χ. As a form of feedback, information about which instance should be ranked above (or below) one another is provided for every pair of instances. This feedback is denoted as a function $\Phi : \chi \times \chi \rightarrow R$, where $(x_0, x_1) > 0$ means x_1 should be ranked above x_0 and vice versa. The learner then attempts to find a ranking function $H : \chi \rightarrow R$ which is as consistent as possible to the given , by asserting that x_1 is preferred than x_0 if $H(x_1) > H(x_0)$.

The RankBoost algorithm is proposed to learn the ranking function H by combining a given collection of ranking functions. The pseudo code is given in Figure 1. RankBoost operates in an iterative manner. In each round, the procedure WeakLearn is called to select the best weak ranker from a large set of candidate weak rankers. The weak ranker has the form $h_t : \chi \rightarrow R$, and $h_t(x_1) > h_t(x_0)$ means that instance x_1 is ranked higher than x_0 in round t. A distribution D_t over $\chi \times \chi$ (i.e., document pairs) is maintained during the training process to reflect the importance of ranking the pair correctly. The weight $D_t(x_0, x_1)$ will be decreased if h_t ranks x_0 and x_1 correctly ($h_t(x_1) > h_t(x_0)$), and increased otherwise. Thus, D_t will tend to concentrate on the pairs that are hard to rank. The final strong ranker H is a weighted sum of the selected weak rankers in each round.

2.2 Extending RankBoost to Web Relevance Ranking

To extend RankBoost to Web relevance ranking, two issues need to be addressed. The first issue is how to generate training pairs. The instance space for search engines does not consist of a single set of instances to be ranked amongst each other, but is instead partitioned by queries issued by users. For each query q, some of the returned documents are manually rated using a relevance score, from 1 (means poor match) to 5 (means excellent match). Unlabeled documents are given a relevance score 0. Based on the rating scores (ground truth), the training pairs for RankBoost are generated from the returned documents for each query. The second issue is how to define weak rankers. In this work, each weak ranker is defined as a transformation of a

Algorithm: Weak ranker selection
Input: $\pi(d)$ and feature vector $(f_0(d), ...f_{N_f}(d))$ of all documents
  for$(k = 0; k < N_f; k++)$
    for$(d = 0; d < N_{doc}; d++)$
      $Hist_k(f_k(d)) += \pi(d)$
    endfor;
    for$(i = N_{bin} - 1; i > -1; i--)$
      $Integral_k(i) = Hist_k(i) + Hist_k(i+1)$
    endfor;
  endfor;
Output: Maximum$(Integral_k(i))$

Fig. 2. The pseudo-code for weak ranker selection in M3.int ((2) and (3) of Appendix B).

document feature, which is a one-dimensional real-valued number. Document features can be classified as query-dependent features (such as query term frequencies in a document and term proximity) or query-independent features (such as PageRank [Brin and Page 1998]). Thus, the same document may be represented by different feature vectors for different queries due to the existence of query-dependent features. The extended RankBoost algorithm for Web relevance ranking is shown in Appendix A.

2.3 WeakLearn Design for Performance and Computation Complexity Analysis

In both the original and extended RankBoost (Figure 2 and Appendix A), WeakLearn is the main routine in each round of the algorithm. Clearly, its design is critical to the quality and speed of RankBoost. This subsection will discuss the design of WeakLearn, and its hardware implementation are addressed in the next section.

WeakLearn chooses from among N_f document features (as weak rankers), the one that yields minimum pair-wise disagreement relative to distribution D over N_{pair} document pairs. Here, we first need to determine the form of weak rankers. We tried out several forms of weak ranker proposed in Freund et al. [2003] to decide on the best form. For low complexity and good ranking quality, we define the weak ranker in the form:

$$h(d) = \begin{cases} 1, \text{ if } f_i(d) > \theta \\ 0, \text{ if } f_i(d) \le \theta or f_i(d) \text{ is undefined} , \end{cases} \quad (1)$$

where $f_i(d)$ denotes the value of feature f_i for document d, and θ is a threshold value. There are totally N_θ threshold values for a feature. To find the best $h(d)$, WeakLearn needs to check all possible combinations of feature f_i and threshold θ_s (refer to equation (3) for the definition of θ_s). Freund et al. [2003] present three methods for this task. The first method (noted as M1) is a numerical search, which is not recommended due to its complexity [Iyer et al. 2000]. The second method (M2) needs to accumulate the distribution $D(d_0, d_1)$ for each document pair to check each (f_i, θ) combination, and thus has a complexity of $O(N_{pair} N_f N_{theta})$ per round. Our initial software (M2.2Dint) reduces

Table I. Computational Complexity of Main Routines in Each Training Round of RankBoost

Main routines in each round		Pair-wise algorithm		Document-wise algorithm	
		Naive M2	M2.2Dint	Naive M3	M3.int
Weak learner	π calculation	-	-	$O(N_{pair})$	$O(N_{pair})$
	Best ranker search	$O(N_{pair}N_fN_\theta)$	$O(N_{pair}N_f)$	$O(N_{doc}N_fN_\theta)$	$O(N_{doc}N_f)$
Weight update		$O(N_{pair})$			

the computational complexity to $O(N_{pair}N_f)$ using 2-D integral histograms,[2] and a complete run still needs about 150 hours.

For speedup, we implement the third method (M3) (as shown in Appendix B), which tries to find the optimal $r(f, \theta)$ by generating a temporary variable $\pi(d)$ for each document. WeakLearn only needs to access π in a document-wise manner for each feature and each threshold, that is, $O(N_{doc}N_fN_\theta)$ in a straightforward implementation (noted as Naive M3). We further reduce the computational complexity to $O(N_{doc}N_f)$ using integral histograms, and the new algorithm is noted as M3.int. The computational complexities of these routines are listed in Table I for comparison. Substituting N_{pair}, N_f and N_θ with the benchmark values given in Section 4.1, it is clear that M3.int has the smallest computational complexity among all algorithms. Therefore, we focus on M3.int in this paper. Both software and hardware implementations of M3.int are presented in the following section.

3. RANKBOOST ACCELERATOR DESIGN

In this section, we describe M3.int algorithm and its accelerator design. Devising an efficient algorithm and mapping the algorithm to efficient hardware architecture are the key challenges addressed in this work.

3.1 M3.int: Document-Wise WeakLearn Based on Integral Histograms

Weak learner M3 (as shown in Appendix B) tries to find the weak ranker with optimal $r(f, \theta)$, by generating a temporary variable $\pi(d)$ for each document. r is defined as:

$$r(f_k, \theta_s^k) = \sum_{d: f_k(d) > \theta_s^k} \pi(d) \, . \tag{2}$$

A straightforward implementation calculates r in $O(N_{doc}N_fN_\theta)$ time per round, and it is apparent that the calculation of r is the bottleneck for the whole RankBoost algorithm. The first reason is that the number of r values is huge. Assuming we take 400 rounds and 256 levels of threshold on the dataset described in Section 4.1, there will be tens of millions $|r(f_k, \theta_s^k)|$ values. The second reason is that during r calculation, WeakLearn needs to iteratively access a large amount of data, including all of the training feature values that may occupy several gigabytes of memory.

[2]This algorithm accumulates $D(d_0, d_1)$ to a 2-D integral histogram with $(f_i(d_0), f_i(d_1))$ as coordinates in $O(N_{pair})$, then selects the best threshold of f_i in $O(N_\theta)$.

M3.int efficiently calculates r with integral histograms in $O(N_{doc}N_f)$ time. This gives close to 2 orders of magnitude reduction in computational complexity compared to Naive M3 in normal cases. The algorithm is described as follows.

For each feature $f_k, k : 1...N_f$, the feature values $\{f_k(d)|d : 1...N_{doc}\}$ for all documents can be classified into N_{bin} bins. The boundaries of these bins are

$$\theta_s^k = \frac{f_{max}^k - f_{min}^k}{N_{bin}} s + f_{min}^k, s = 0, 1, ..., N_{bin}, \tag{3}$$

where f_{max}^k (resp. f_{min}^k) is the maximum (resp. minimum) value over all f_k in the training dataset. Then each document d can be mapped to one of the bins according to the value of $f_k(d)$:

$$Bin_k(d) = floor(\frac{f_k(d) - f_{min}^k}{f_{max}^k - f_{min}^k} - 1) . \tag{4}$$

The histogram of $\pi(d)$ over feature f_k is then built using:

$$Hist_k(i) = \sum_{d:Bin_k(d)=i} \pi(d), i = 0, ..., (N_{bin} - 1) . \tag{5}$$

Then, we can build an integral histogram by summing elements in the histogram from the right ($i = N_{bin} - 1$) to the left ($i = 0$). That is,

$$Integral_k(i) = \sum_{a>i} Hist_k(a), i = 0, ..., (N_{bin} - 1) . \tag{6}$$

Based on Equations (2) to (5), it can be deduced that the value $Integral_k(i)$ equals to $r(f_k, \theta_i^k)$. The pseudo code of r calculation algorithm is summarized in Figure 2. With this algorithm, we obtain more than 3 times acceleration over M2.2Dint. However, runtime profiling (Section 4.3) reveals that r calculation still occupies almost 99% of the CPU time. Thus, our attention is next focused on this task. We observe that integral histograms for different features can be built independently of each other. This feature-level parallelism makes it possible to accelerate the algorithm using various techniques, such as distributed computing and FPGA-based accelerator, as described in the next subsection.

3.2 Mapping M3.int to FPGA-Based Accelerator

To further improve the performance of RankBoost, we investigate an implementation of M3.int using FPGA. This section introduces the accelerator system, describes the SIMD architecture for building the integral histograms, and discusses the implementation issues.

3.2.1 *STAR-III: FPGA acceleration board with PCI interface and large on-board memory.* The accelerator is a PCI card with FPGA and memories (SDRAM, SRAM), as shown in Figure 3. On the board, an Altera Stratix-II FPGA (EP2S60) is used as the core computation component. The on-board memories include DDR SDRAM (1 GB, the maximum possible is 2 GB) and 32MB SRAM (not used in this work). The board is designed by the authors

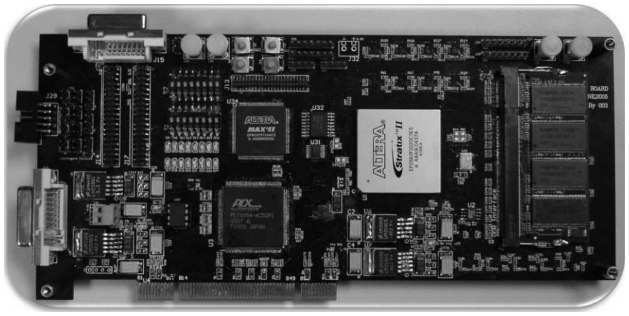

Fig. 3. STAR-III accelerator: FPGA-based PCI card with on-board memories.

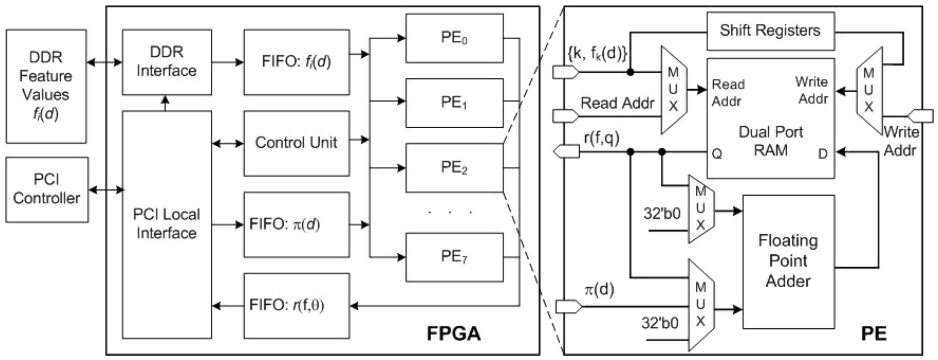

Fig. 4. The SIMD architecture and PE micro-architecture for RankBoost accelerator.

of this paper, and several Chinese universities have got STAR-III boards for various research projects.

3.2.2 *Building Integral Histograms with an SIMD Architecture.* The most time-consuming part of the RankBoost algorithm is building the integral histograms. We propose an SIMD architecture to separately build several integral histograms with multiple PEs at the same time, as shown in Figure 4.

At the initialization step, the software on host computer sends the quantized feature values to DDR memory through the PCI bus, PCI controller and FPGA. Note that the data are organized in such a way as to enable streaming memory access, making full use of the DDR memory bandwidth. In each training round, the software calls WeakLearn to compute $\pi(d)$ for every document, and sends $\pi(d)$ to the FIFO in FPGA. The control unit (CU) in FPGA then instructs the PE arrays to build histograms and integral histograms, and sends the results $r(f, \theta)$ to the output FIFO. CU is implemented as a finite-state machine (FSM), which halts or resumes the pipeline in PEs depending on the status of each FIFO. When CU indicates that the calculation of r has completed, the software reads back these r values and selects the maximum. The software then updates the distribution $D(d_0, d_1)$ over all pairs and begins the next round.

To build the histogram for a single feature as described in Equation (4), it needs to accumulate π values that has the same feature value. That is, normalized feature value $Bin_k(d)$ of the document d will select a "bin" to accumulate the $\pi(d)$. In a naive implementation of PE, direct computation of Equation (2) processes $\pi(d)$ values for a single feature sequentially. When the adjacent feature values $Bin_k(d_i)$ and $Bin_k(d_i+1)$ are the same, $\pi(d_i)$ and $\pi(d_i+1)$ will be added to the same position in the histogram. The floating point adder will then have to add bubbles to its pipeline to avoid read-after-write hazard. The design shown in Figure 4 avoids such data hazard by building multiple (we select 16) histograms in the pipeline and the dual port RAM in an interleaved manner. That is, the feature values $fk(dj)$ are organized in the sequence of $\{f_0(d_0), f_1(d_0), f_2(d_0), \ldots, f_{15}(d_0), f_0(d_1), f_1(d_1), f_2(d_1), \ldots, f_{15}(d_1), \ldots, f_0(d_{N_doc}), f_1(d_{N_doc}), \ldots, f_{15}(d_{N_doc})\}$. PE uses k, $f_k(d_j)$ as address to read out value stored in bin $f_k(d_j)$ of histogram k, and adds it to current $\pi(d_j)$. The result of floating point adder will be written back to the RAM with delayed read address from shift registers. Histograms are built after all the feature values are processed. Then CU (control unit) will send read/write address and control signals to PE to build integral histograms in a similar interleaved manner. This microarchitecture of PE (Figure 4) could support full-pipelined operation, which is important for the hardware performance.

3.2.3 *Efficient Mapping to STAR-III Board.* The next challenging task is to select the parameters for PE and PE array, such as input data precision, and the number of pipeline stages, and the number of PEs. These parameters should be selected to achieve the performance goal of the accelerator while meeting multiple conflicting constraints such as area, frequency, memory bandwidth limit, precision, scalability, etc. For example, increasing the number of PEs may not increase the system performance when the DDR memory cannot send enough data to PE arrays due to limited bandwidth. After several trial implementations and experiments, we made the following conclusions:

(1) Increasing the threshold levels to more than 256 will not optimize the generalization ability of the final strong ranker. Thus, each feature can be quantized to 8 bit or less.
(2) The DDR memory on STAR-III can provide feature data in a bandwidth of about 1GBps in streaming access mode.
(3) When features are quantized to 8 bit, PE arrays will get 1 G feature values per second, that is, $1G/N_f$ documents per second. Thus the bandwidth for accessing $\pi(d)$ values (single precision) is about $4Bytes \times 1G/N_f$, which is much lower than the bandwidth of PCI bus.
(4) After optimization with aggressive pipelining (16 stages) and high effort place-and-route in Quartus-II software, the maximum frequency of PEs is 180MHz.
(5) Double precision floating point arithmetic doesn't show better quality and will slow down the training speed.

Following the above arguments, a PE array with 8 PEs will consume feature values in the bandwidth of $1.44GBps$, which is beyond the upper bound of the

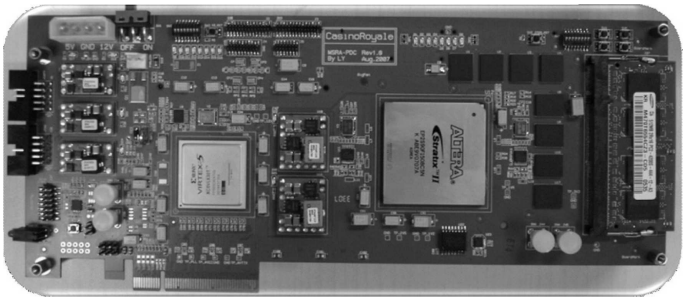

Fig. 5. AJAW FPGA acceleration board.

DDR memory bandwidth on STAR-III. Thus, increasing the number of PEs to more than 8 will not shorten the computation time further. So the final FAR1 configuration is selected as 8 PEs with a 16-stage pipeline, and its implementation in EP2S60 costs 11,357 (23%) ALUTs and 2,102K (83%) memory bits.

3.3 FAR2: Upgrade of FAR1 with New Accelerator Board

After FAR1 [Xu et al. 2007] is used by researchers in Microsoft Research Asia for almost one year, users request us to upgrade the accelerator to support even larger data set. The reason is that training data size has been doubled in the past year, and it will continue growing in the future. Because the training data are loaded into the on-board memory, we will have to increase the size of on-board memory in the new implementation. This subsection describes how we upgrade FAR1 with our new accelerator board, which is called **AJAW**.

3.3.1 *AJAW: FPGA Acceleration Board with PCI Express Interface and Large On-Board Memory*. AJAW board is the second generation of FPGA acceleration boards designed in Platforms and Devices Center, Microsoft Research ASIA. Several Chinese universities are developing applications on AJAW boards. It is designed to utilize the state-of-art technologies to provide a much more powerful acceleration platform than STAR-III (Section 3.2.1). AJAW board has two DDR2 modules, which support up to 16GB capacity and 6.25GBps bandwidth. A Xilinx Virtex-5 LXT FPGA[3] is used to provide PCI Express interface with host computer. The FPGA has an embedded PCI Express endpoint hard core, which could support up to 8 lanes access. This host interface solution can support up to 4GB/s bidirectional throughput in theory. AJAW board continues to use Altera Stratix-II family FPGAs as the main computation engine. The AJAW board is shown in Figure 5.

3.3.2 *FAR2: Extending the SIMD Architecture to AJAW*. AJAW board has similar architecture with STAR-III board, so it is possible to reuse the FPGA logic of FAR1 by extending the 8-way SIMD architecture proposed in section 3.2.2. It is straightforward to increase the number of PEs in FPGA to increase the throughput of processing the training data from SRAMs. The

[3]http://www.xilinx.com

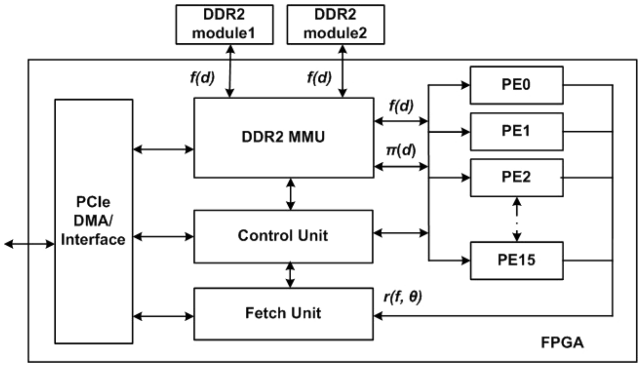

Fig. 6. FAR2 FPGA architecture.

number of PEs is then doubled to 16, which simplifies the data path redesign and achieves efficient resource usage. So, we accept 16-way SIMD architecture for the RankBoost FPGA logic on AJAW board (called FAR2 logic).

One challenge of FAR2 logic is how to fully utilize the throughput provided by the two DDR2 modules. That is, we need to efficiently combine the two asynchronous data streams from two individual DDR2 modules into one synchronous data stream. To achieve this, we design a MMU that connects PCI Express interface, DDR2 modules and PEs, as shown in Figure 6. The MMU handles storing data from PCIe interface to DDR2 modules, fetching data out and feeding to PE arrays.

To write data to DDR2 memory, the major problem is how to dispatch synchronous data from PCIe to two asynchronous DDR2 modules. PCIe and two DDR2 modules are in three different clock domains, so we insert two asynchronous FIFOs to separate them, and data from PCIe are alternately dispatched to the two FIFOs. To read data from DDR2 memory, synchronizing different datasteams is the major problem. We insert two read queues between DDR2 memory data port and PE arrays to smooth the synchronization variants. The queues are written individually and read simultaneously.

4. RESULTS

This section presents the RankBoost accelerator performance models and the performance experiments results.

4.1 Experiment Setup and Training Time Results

To compare the performance of different implementations, we use a benchmark dataset whose parameters are shown in Table II.

The performance of each implementation is shown in Table III. We run the RankBoost algorithm for 300 rounds, which is a typical length of training to obtain a model with good ranking accuracy. The initial Naive M3 algorithm (run on a Quad-Core Intel Xeon 2.33 GHz processor with 7.99 GB RAM) took about 46 hours to complete the training rounds. The M3.int version took only 4.4 hours. The distributed version (refer to 4.4) with 4 computation threads

Table II. Benchmark Dataset Parameters

Number of queries (N_q)	25,724
Number of documents (N_{doc})	1,196,711
Number of pairs (N_{pair})	15,146,236
Number of features (N_f)	646
Original data size (MB)	2,992
Compressed data size (MB)	748

Table III. Time for a Complete Run of 300 Rounds

Implementation	Time(hour)	Speedup
SW(Naive M3)	46.06	1.00
SW(M3.int)	4.43	10.40
4-thread distributed SW(M3.int)	0.78	59.05
HW accelerator (FAR)	0.27	170.59
HW accelerator (FAR2)	0.17	270.94

further reduced the time to slightly below one hour. All these software implementations are compiled using Microsoft Visual Studio 2005, optimized for speed with SSE2 option. The first version hardware accelerator (FAR1) plugged into a commodity PC (Dell GX620 3.00 GHz Intel Pentium 4 system with 2GB RAM) took only 16 minutes, giving a speedup of around 171 times over the original software. With FAR2, the acceleration rate is further improved to 271 times.

4.2 Performance Model

In this section, the performance models are deduced for software Naive M3, software M3.int, hardware M3.int and the distributed M3.int. According to the running process of RankBoost (Figure 2), all these implementations share the same high level performance model:

$$t_{total} = t_{ini} + N_{round} * t_{round} \ , \tag{7}$$

where initialization time t_{ini} and per training round time t_{round} are dependent on the implementation. As the program initialization time t_{ini} for both single-threaded software and hardware implementations are relatively small, we model only the executing time of each training round (t_{round}), which includes 3 main routines according to the computational complexities listed in Table I. It can be concluded that implementations are different in the routine of best ranker search.

Thus, for software Naive M3 and M3.int, we can use linear models to estimate $t_{round(NaiveM3)}$ and $t_{round(M3.int)}$ as:

$$t_{round(NaiveM3)} = \alpha N_{pair} + \beta_0 N_f N_{doc} N_\theta \tag{8}$$

$$t_{round(M3.int)} = \alpha N_{pair} + \beta_1 N_f N_{doc} \ . \tag{9}$$

The first term αN_{pair} represents the time for π calculation and weight update, and these two routines are the same for all 4 implementations. The second term reflects the best ranker search routines in software.

For $t_{round(HW)}$, the time for best ranker search can be divided into communication time $t_{comm.HW}$ and computation time $t_{comp.HW}$. Here, communication

refers to the transfer of integral histograms from FPGA to host computer through PCI (or PCI Express) in DMA mode. These histograms occupy $4N_fN\theta$ bytes RAM space. Thus, the time to transfer this amount of data can be noted as $t_{comm.HW} = aN_fN_\theta + b$, where b is the system overhead of a single DMA transfer. The computation time involves going through all data in the SDRAMs in a streaming manner, and it is constraint be the SDRAM memory bandwidth. So hardware computation time can be represented as $t_{comp.HW} = \beta_{HW}N_fN_{doc}$, where β_{HW} is the reciprocal of SDRAM bandwidth, and N_fN_{doc} is the size of compressed features in it. In summary, we have:

$$t_{round(HW)} = \alpha N_{pair} + \beta_{HW}N_fN_{doc} + aN_fN_\theta + b \ . \tag{10}$$

The ratio $t_{round(NaiveM3)}/t_{round(HW)}$ and $t_{round(M3.int)}/t_{round(HW)}$ is therefore the speedup obtained through acceleration.

The time model for the distributed implementation has a slightly different format, as it involves a significant initialization time component for the host machine to distribute feature data (N_fN_{doc}) to client computers. The per-round best ranker search is handled by the clients, whereas the π calculation and weight update remains on the host. The time performance model for the complete run can be represented as:

$$t_{total(Dist)} = \gamma N_fN_{doc} + N_{round}(\alpha N_{pair} + t_{comp(Dist)} + t_{comm(Dist)}) \ . \tag{11}$$

Here, t_{comp} refers to the best ranker search time, and t_{comm} consists of (1) the time for the host to send updated pair weights to all clients, and (2) the time for the host to collect the result of best ranker search from all clients. Each client is responsible for a portion of feature data, thus the distributed best ranker search time can be modeled as:

$$t_{comp(Dist)} = \max_{m\in[1\ldots N_{thread}]}(\lceil \frac{N_f}{N_{thread}} \rceil N_{doc}\beta_m) \tag{12}$$

and the communication time is

$$t_{comm(Dist)} = \sum_{m=1}^{N_{thread}} (c_mN_{doc} + d_m) \ . \tag{13}$$

The speedup obtained through hardware acceleration over the distributed implementation is given by the ratio $t_{total(Dist)}/(N_{round} * t_{round(HW)})$.

4.3 Parameter Estimation and FAR1/FAR2 Performance

In order to estimate the parameters in our performance model, we measured the processing time of the main routines on the benchmark dataset, averaged over 300 rounds, as shown in Table IV. The "best ranker search" routine is obviously the bottleneck task for M3.int software implementation, because it occupies 98.7% of the per-round computation time. FAR1 accelerates this task by 67.01 times, giving per-round speedup of 17.28 times.

We measured the runtime on datasets with varying N_q, N_{pair}, N_f, N_{doc} and N_θ settings. Then we apply linear regression to infer parameters in the

Table IV. Time (Seconds) of Main Routines in Each Round. Machine Specifications are as Listed in Subsection 4.1. Optimized SW is Described in Subsection 4.6

Routine	SW(M3.int)	Optimized SW	FAR1		FAR2	
π calculation	0.110	0.110	0.176		0.137	
Best ranker search	52.068	26.157	t_{comp}	0.719	t_{comp}	0.362
			t_{comp}	0.058	t_{comp}	0.049
Weight update	0.958	0.959	2.122		1.375	
Total time per-round	53.136	27.226	3.075		1.923	

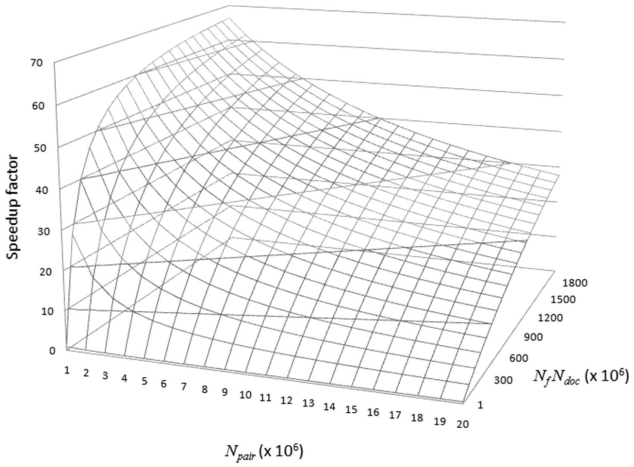

Fig. 7. Speedup of FAR1 over the software M3.int for each round. Execution times for varying N_{pair}, N_{doc} and N_f values are estimated using the derived performance model.

performance model. With the parameters and the model, the accelerator performance on a certain dataset could be estimated. Here we take the speedup ratio of FAR1 over software M3.int as an example. The surface in Figure 7 represents the estimated speedup ($t_{round(M3.int)}/t_{round(FAR1)}$) on datasets with fixed $N = 256$ and varying N_{pair}, N_{doc} and N_f. The values on $N_{doc} * N_f$ axis also represent the size of compressed feature data to be loaded to the DDR memory of the accelerator. The value of $N_{doc} * N_f$ ranges within $[0, 2 \times 10^9]$, because the STAR-III board supports up to 2GB memory. As the feature data size increases, more speedup is achieved. On the other hand, as N_{pair} increases, the time portion of π calculation and weight update will increase, and the whole speedup will decrease.

FAR2 is upgraded from FAR1 to handel dataset that is larger than 1GB, which is the maximum capacity of the DDR module we installed on FAR1 board (STAR-III). To illustrate the performance of FAR2, we measured the acceleration rate of FAR2 over the software implementation (Naive M3) on several real world datasets, as shown in Figure 8. It can be concluded that FAR1/FAR2 accelerators increased the training speed by $2 \sim 3$ orders of magnitude on MSN search engine data, compared to the original software implementation on a server.

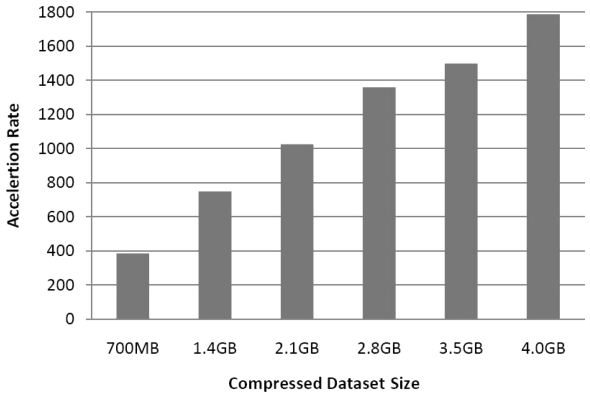

Fig. 8. Speedup of FAR2 over the software Naive M3 for each round. Datasets are selected from real application.

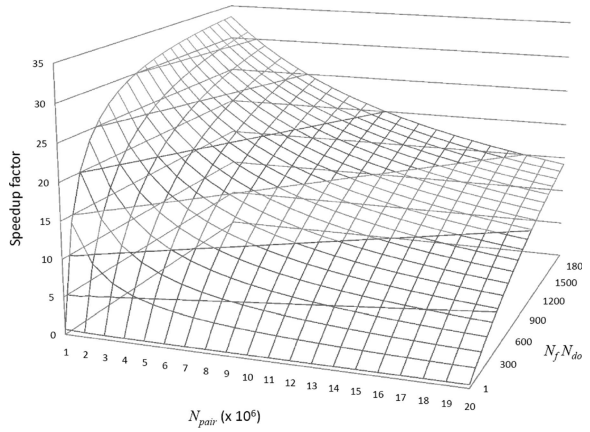

Fig. 9. Speedup of FAR1 over the optimized software M3.int for each round.

4.4 Software Optimization and Distributed Software Performance

The dataset size for the software version is only limited by the capacity of the hard disk, because it loads feature data from hard disk to memory only when the feature is used in each round. However, disk access slows down the software, and it is possible to achieve faster speed by loading all feature data beforehand, provided that we have enough user memory space to store the whole data. The performance of this version is listed in Table III, and the new derived speedup estimation is shown in Figure 9. We note that FAR is still superior to the optimized software version in terms of speed. Certainly, a distributed version of the optimized software could have to support a larger dataset with high performance that is comparable to FAR1/FAR2. To investigate this, we compute the speedup of the distributed implementation with increasing number of threads over single threaded

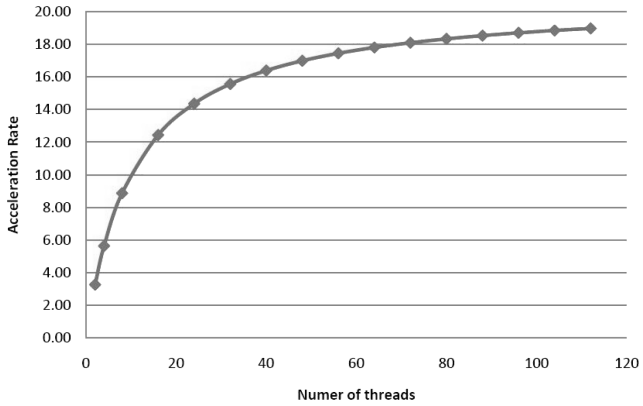

Fig. 10. Distributed SW speedup over software Naive M3.

Table V. Power and Energy Consumption for RankBoost on Different Implementations

Implementations	Power	Time (hours)	Energy (kWh)
1 computer (1×CPU)	200W	1535.33	307.07
1 computer (1×FAR1)	220W	9	1.98
1 computer (1×FAR2)	240W	5.67	1.36
2 computers (Dual-core, 4 threads)	400W	26	10.40
26 computer (Dual-core, 52 threads)	5200W	9	46.80

M3.int. Figure 10 shows the trend of the speedup, as estimated using the time performance model described in Section 4.2. We see that around 52 threads are needed to achieve the same speedup (17.28x) as FAR1. This is a relatively high resource/power demand, which strengthens the case for our hardware accelerator. Furthermore, because of the high communication cost and long initial time, acceleration rate of distributed software is saturated to 20.82x as thread number increases, which will never catch up with the FAR2 acceleration rate (27.63x) on the same dataset.

4.5 Power Consumption Comparison

Power recently becomes an essential concern for implementations of large scale computing in datacenters. Here we compare the energy (power × time) consumption for running RankBoost (10,000 rounds) on different implementations, as listed in Table V. It can be concluded that FAR2 has the best energy efficiency. To have the same time performance with FAR1, we have to use distributed software on 26 servers, which requires 23.64 times more energy than FAR1 in a computer.

4.6 Algorithm Quality Experiments and New Opportunities Enabled by FAR

To compare the relevance quality of generated ranking models, we run RankBoost with the same settings as RankNet. Ranking accuracy was measured by Normalized Discounted Cumulative Gain (NDCG) [Jarvelin and Kekalainen

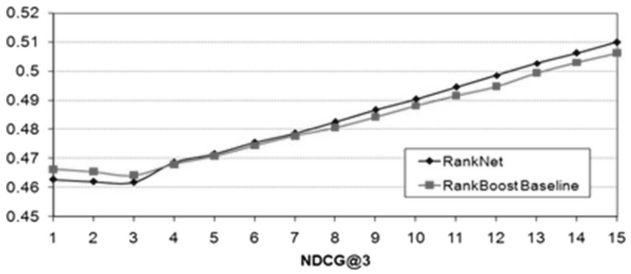

Fig. 11. NDCG comparison between RankNet and RankBoost.

2002]. As shown in Figure 11, RankBoost yields comparable results to RankNet. This baseline performance, as computed using the algorithm in Appendix A, has since improved significantly as more research effort are put into it [Tsai et al. 2007; Qin et al. 2006]. Several FAR systems have been successfully used by the Web Search and Mining Group in Microsoft Research Asia. The speedup enabled by FAR has rendered RankBoost more attractive than other candidate algorithms (such as RankNet and RankSVM). An application for a commercial search engine requires about 200,000 rounds to obtain a meaningful model, which takes FAR1 less than 2.5 days to complete. While the distributed software on four servers will need a week to get the same result. It is fair to say that our accelerator opens up the opportunity to accomplish such compute-heavy tasks in much less time and expense.

5. CONCLUSIONS

This article describes our efforts in the design of FPGA-based accelerators for the RankBoost algorithm in Web search engines. To build efficient accelerators, we first optimized the RankBoost algorithm to reduce the computation complexity and to make it suitable for parallel implementation in hardware logic. Then we designed an efficient SIMD architecture to fully utilize the hardware resources on our two generations of customized accelerator board. This SIMD architecture can be easily scaled up to fit future accelerator board with bigger memory and higher host bandwidth. We are also combining several FPGA boards with our distributed software to handle more data with better scalability.

Empirical results show that FAR accelerators can accelerate the original software by $2 \sim 3$ orders of magnitude in commercial Web search datasets. Compared to the software implementations, the FPGA-based accelerators show great advantage in performance, cost and power consumption. It has been successfully used by researchers in search relevance ranking and in some commercial applications. Several Chinese universities are also developing their own applications on our accelerator cards. Future work will also focus on extending the experience of designing this accelerator to other similar machine learning algorithms.

APPENDIX

A. RANKBOOST FOR RELEVANCE RANKING

Given:

N_q queries $\{q_i | i = 1, ..., N_q\}$.

N_q^i documents $\{d_j^i | j = 1, ..., N_q^i\}$ for each query q_i, where $\sum_{i=1}^{N_q^i} N_i = N_{doc}$.

N_f features $\{f_k(d_j^i) | k = 1, ..., N_f\}$ for each document d_j^i.

N_{pair} pairs (d_{j1}^i, d_{j2}^i) generated by ground truth rating $R(q_i, d_j^i)$ or R_j^i.

Initialize: initial distribution $D_1(d_{j1}^i, d_{j2}^i)$ over $\chi \times \chi$

Do for $t = 1, ..., T$:

(1) Train **WeakLearn** using distribution D_t.

(2) **WeakLearn** returns a weak hypothesis h_t, weight α_t.

(3) Update weights: for $\forall(d_0, d_1)$,

$$D_{t+1}(d_0, d_1) = \frac{D_t(d_0, d_1) exp(-\alpha_t(h_t(d_0) - h_t(d_1)))}{Z_t}$$

where Z_t is the normalization factor:

$$Z_t = \sum_{d_0, d_1} D_t(d_0, d_1) exp(-\alpha_t(h_t(d_0) - h_t(d_1))).$$

Output: the final hypothesis: $H(x) = \sum_{t=1}^{T} \alpha_t h_t$.

B. WEAKLEARN FOR RANKBOOST IN RELEVANCE RANKING (M3)

Given: Distribution $D(d_0, d_1)$ over all pairs

Compute:

(1) For each document d, compute $\pi(d) = \sum_{d'}(D(d', d) - D(d, d'))$

(2) For every feature f_k and every threshold θ_s^k, compute $r(f_k, \theta_s^k) = \sum_{d: f_k(d) > \theta_s^k} \pi(d)$

(3) Find the maximum $|r^*(f_{k^*}, \theta_{s^*}^{k^*})|$

(4) Compute $\alpha = \frac{1}{2} ln(\frac{1+r^*}{1-r^*})$

Output: weak ranking $f_{k^*}, \theta_{s^*}^{k^*}$ and α

ACKNOWLEDGMENTS

The authors would like to thank Qing Yu, Chao Zhang, Tao Qin, Vivy Suhendra and Jian Ouyang for their support in this work, and Wei Lai, Jun-Yan Chen and Tie-Yan Liu for their useful feedback. The authors would also thank our editor Dwight Daniels for his wonderful work.

REFERENCES

BAEZA-YATES, R. AND RIBEIRO-NETO, B. 1999. *Modern Information Retrieval*. Addison Wesley.

BRIN, S. AND PAGE, L. 1998. The anatomy of a large-scale hypertextual web search engine. *Comput. Netwo. ISDN Syst. 30*, 1-7, 107–117.

BURGES, C., SHAKED, T., RENSHAW, E., LAZIER, A., DEEDS, M., HAMILTON, N., AND HULLENDER, G. 2005. Learning to rank using gradient descent. In *Proceedings of the 22nd International Conference on Machine Learning*. ACM, New York, 88–96.

EL-GHAZAWI, T., BENNETT, D., POZNANOVIC, D., CANTLE, A., UNDERWOOD, K., PENNINGTON, R., BUELL, D., GEORGE, A., AND KINDRATENKO, V. 2006. Is high-performance reconfigurable computing the next supercomputing paradigm? In *Proceedings of the ACM/IEEE Conference on Supercomputing*. ACM, New York, 219–228.

FAN, W., GORDON, M. D., PATHAK, P., XI, W., AND FOX, E. A. 2004. Ranking function optimization for effective Web search by genetic programming: An empirical study. In *Proceedings of the 37th Hawaii International Conference on System Sciences*. 8–16.

FREUND, Y., IYER, R., SCHAPIRE, R., AND SINGER, Y. 2003. An efficient boosting algorithm for combining preferences. *Mach. Learn. 4*, 933–969.

FREUND, Y. AND SCHAPIRE, R. E. 1995. A decision-theoretic generalization of on-line learning and an application to boosting. In *Computational Learning*, Lecture Notes in Computer Science, vol. 904. Springer, Berlin, 23–37.

FUHR, N. 1989. Optimum polynomial retrieval functions based on the probability ranking principle. *ACM Trans. Inform. Syst. 7*, 3, 183–204.

IYER, R. D., LEWIS, D. D., SCHAPIRE, R. E., SINGER, Y., AND SINGHAL, A. 2000. Boosting for document routing. In *Proceedings of the 9th International Conference on Information and Knowledge Management*. ACM, New York, 70–77.

JARVELIN, K. AND KEKALAINEN, J. 2002. Cumulated gain-based evaluation of ir techniques. *ACM Trans. Inform. Syst. 20*, 4, 422–446.

JOACHIMS, T. 2002. Optimizing search engines using click through data. In *Proceedings of the ACM SIGKDD Conference on Knowledge Discovery and Data Mining (KDD)*. 133–142.

LAPTEV, I. 2006. Improvements of object detection using boosted histograms. In *Proceedings of British Machine Vision Conference*.

LIU, X., ZHANG, L., LI, M., ZHANG, H., AND WANG, D. 2004. Boosting image classification with lda-based feature combination for digital photograph management. *Patt. Recogn.* Special Issue on Image Understanding for Digital Photos.

NALLAPATI, R. 2004. Discriminative models for information retrieval. In *Proceedings of the 27th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*. ACM, New York, 1401–1406.

QIN, T., LIU, T.-Y., TSAI, M.-F., ZHANG, X.-D., AND LI, H. 2006. Learning to search Web pages with query-level loss functions. In Microsoft Research Tech. rep.

SCHAPIRE, R. E. 1999. A brief introduction to boosting. In *Proceedings of International Joint Conference on Artificial Intelligence*. 1401–1406.

SCHAPIRE, R. E. 2001. The boosting approach to machine learning: An overview. In *Proceedings of the MSRI Workshop on Nonlinear Estimation and Classification*.

TSAI, M.-F., LIU, T.-Y., QIN, T., CHEN, H.-H., AND MA, W.-Y. 2007. Frank: a ranking method with fidelity loss. In *Proceedings of the 30th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*. ACM, New York, 383–390.

UNDERWOOD, K. AND HEMMERT, K. 2004. Closing the gap: Cpu and fpga trends in sustainable floating-point blas performance. In *Proceedings of 12th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM'04)*. 219–228.

VIOLA, P. AND JONES, M. 2001. Robust real-time object detection. Tech. rep. in Compaq Cambridge Research Lab.

XU, N.-Y., CAI, X.-F., GAO, R., ZHANG, L., AND HSU, F.-H. 2007. Fpga-based accelerator design for rankboost in Web search engines. In *Proceedings of the IEEE International Conference on Field-Programmable Technology (FPT'07)*. 33–40.

Received June 2008; revised August 2008; accepted October 2008