FPGA-BASED ACCELERATION OF NEURAL NETWORK FOR RANKING IN WEB SEARCH ENGINE WITH A STREAMING ARCHITECTURE

Jing YAN[1,2], *Ning-Yi XU*[*,1], *Xiong-Fei CAI*[1], *Rui GAO*[1], *Yu WANG*[+,1,2], *Rong LUO*[2], *Feng-Hsiung Hsu*[1]

[1]Hardware Computing Group, Microsoft Research Asia, Beijing, China, 100190
[2]Department of Electronic Engineering, Tsinghua university,Beijing, China, 100084
email: {v-jiy, ningyixu, xfcai, ruigao, fhh}@microsoft.com, {yu-wang,luorong}@mail.tsinghua.edu.cn

ABSTRACT

Web search engine companies are intensively running learning to rank algorithms to improve the search relevance. Neural network (NN)-based approaches, such as LambdaRank, can significantly increase the ranking quality. While, their training is very slow on a single computer and inherent coarse-grained parallelism could be hardly utilized by computer clusters. Thus an efficient implementation is necessary to timely generate acceptable NN models on frequently updated training datasets. This paper presents our work in accelerating LambdaRank with an FPGA-based hardware accelerator. A SIMD streaming architecture is proposed to i) efficiently map the query-level NN computation and data structure to FPGA, ii) fully exploit the inherent fine-grained parallelism, and iii) provide scalability to large scale datasets. The accelerator shows up to 17.9X speedup over the software implementation on datasets from a commercial search engine.

1. INTRODUCTION

Web search based Internet advertising services have become a huge business in recent years with billions of annual earnings. To attract more users and obtain larger market share, search engine companies are intensively using ranking techniques to increase their search relevance, which is determined by ranking functions that rank resultant documents (URLs) according to their similarities to the input query. Many factors affect the ranking function for search relevance, such as page content, title, 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, "learning to rank" algorithms have been actively designed and applied to automatically learn complex ranking functions from large-scale training sets in recent years [1].

*Corresponding author

+Yu WANG's work was partially supported by National Natural Science Foundation of China No.60870001 and TNList Cross-discipline Foundation.

Among the proposed algorithms, NN-based approaches, such as RankNet [2] and LambdaRank [3][4] have shown superior quality over others. RankNet, which uses backward propagate (BP) NN to optimize a smooth cost function, was the first work that was evaluated on a commercial search engine. LambdaRank, implemented with RankNet models, has shown significant relevance improvements on commercial search engine datasets. While, it is still very slow. On a typical dataset, it costs half a day for acceptable results.

This paper proposes the design of a FPGA-based accelerator for the LambdaRank algorithm. We devise a SIMD streaming architecture to accelerate the training process for the web relevance ranking. With this architecture, the LambdaRank accelerator presents up to 17.9X speedup compared with the software on datasets from a commercial Web search engine, and more speedup is expected through further performance optimizations. Experiments also show that this architecture scales well to the large scale training data.

This paper is organized as follows: Section 2 describes the LambdaRank algorithm for relevance ranking, and analyzes the software implementation. Section 3 presents the design of the accelerator architecture. The experimental results and performance model are discussed in section 4. Section 5 concludes the paper and discusses the future work.

2. APPLICATION: LAMBDARANK FOR RANKING

2.1. LambdaRank algorithm

LambdaRank is proposed by Christopher J.C. Burges in 2005 [3] to minimize any multivariate, non-differentiable cost function. The algorithm is provided by a two-layer NN formulation (noted as LambdaRank in the following), although the idea can be applied to any differentiable function class. We will briefly describe how LambdaRank is applied to train the ranking functions for Web search. Ranking functions are used to assign scores to web pages that match the queries from users, and these scores determine the order of returned pages.

The input data, or so called training dataset, is struc-

978-1-4244-3892-1/09/$25.00 ©2009 IEEE

tured with queries that are issued by search engine users. A dataset has N_Q queries, and the q^{th} query has N_{Dq} documents. Each document is expressed by N_F document features. Each document feature is a 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 document length and PageRank [5]). Thus, the same document may be represented by different feature vectors for different queries due to the existence of query-dependent features. We use $feature_{qdf}$ to represent the f^{th} feature value of the d^{th} document in the q^{th} query. A ranking model is actually a function of document features. The proposed 2-layer NN model has N_F input nodes, N_H hidden nodes and one output node. The model parameters include: i) the hidden node weight w_{hf} assigned to the edge from the f^{th} input node to the h^{th} hidden node; ii) the output node weight w_h assigned to the edge from the h^{th} hidden node to the output node; and iii) hidden node threshold θ_h assigned to the h_{th} hidden node.

For training, a label $l_d \in$ {"bad match","good match", etc.} is assigned to each document d by people, according to its relevance to the corresponding query. In the same query, two documents are defined as a pair when they have different labels. In web relevance ranking, normalized discounted cumulative gain (NDCG) [6] is usually used as the measurement to evaluate the quality of the ranking results. A higher NDCG value means better relevance.

Instead of defining the cost function C, LambdaRank defines the gradient of C with respect to the score s_j of the document j, for the q^{th} query. The λ functions are defined to reflect human intuitions of particular quality measure.

Furthermore, LambdaRank is proposed as a batch learning per query (weights are updated for each query). For each query in the training data, firstly, scores $s_j, j = 1, ..., n$ of each document are calculated by

$$s_j = \sum_{h=0}^{N_H} w_h \tanh(\sum_{f=0}^{N_F} w_{hf} feature_{qjf} + \theta_h) \quad (1)$$

This step is called forward propagation (FP).

Let P be the set of document pairs indices and $w_k \in \mathbb{R}$ be the model parameters. The total cost is $C_T \equiv \sum_{\{i,j\} \in P} C(s_i, s_j)$. Let P_i be the set of indices j for which $\{i, j\}$ is a valid pair, and let D be the set of document indices. The derivative of C_T with respect to w_k is

$$\frac{\partial C_T}{\partial w_k} = \sum_{i \in D} \frac{\partial s_i}{\partial w_k} \sum_{j \in P_i} \frac{\partial C(s_i, s_j)}{\partial s_i} \quad (2)$$

Then the second step calculates $\lambda_i \equiv \sum_{j \in P_i} \frac{\partial C(s_i, s_j)}{\partial s_i}$ for each $i = 1, ..., n$. The term $\frac{\partial C(s_i, s_j)}{\partial s_i}$ is designed to reflect the intent of NDCG, which is beyond the scope of this paper. We use

$$\frac{\partial C(s_i, s_j)}{\partial s_i} = \frac{(2^{l_i} - 2^{l_j})}{1 + e^{s_i - s_j}} log(\frac{1 + rank(j)}{1 + rank(i)}) \quad (3)$$

Algorithm 1: Pseudo code of LambdaRank

Input: training round number T, N_H, document features and pair indices, η
Output: w_{hf}, w_h, θ_h

Initialization: $w_{hf} = random(), w_h = 0, \theta_h = 0$
for $t = 1$ **to** T **do**
  for $q = 1$ **to** N_Q **do**
    /* ***FPCalc*** */
    for $d = 1$ **to** N_{Dq} **do**
      for $h = 1$ **to** N_H **do**
        for $f = 1$ **to** N_F **do**
          $s_{dh} += w_{hf} * feature_{qdf}$
        end
        $s_{dh} = \tanh(s_{dh} + \theta_h)$
      end
      $s_d += w_h * s_{dh}$
    end
    /* ***LambdaCalc*** */
    $Sort(s_i, i = 1, ..., N_{Dq})$ to get $rank(i), i = 1, ..., N_{Dq}$
    for $i = 1$ **to** N_{Dq} **do**
      for $j \in P_i$ **do**
        /* ref to equation(3) */
        $\lambda(i) += Partial(s_i, s_j, rank(i), rank(j), l_i, l_j)$
      end
    end
    /* ***BPCalc*** */
    for $h = 1$ **to** N_H **do**
      for $d = 1$ **to** N_{Dq} **do**
        for $f = 1$ **to** N_F **do**
          $\frac{\partial s_d}{\partial w_{hf}} = w_h(1 - s_{dh}^2) f_{dqf}$
        end
        $\frac{\partial s_d}{\partial w_h} = s_{dh}$
        $\frac{\partial s_d}{\partial \theta_h} = w_h(1 - s_{dh}^2)$
      end
      Update θ_h, w_{hf}, w_h with their δ and η using equation (4)
    end
  end
end

to calculate λs, where $rank(j)$ is the ranking position of s_j in this query.

The third step is back propagation with

$$w_k' = w_k + \eta \frac{\partial C_T}{\partial w_k} = w_k + \eta \sum_{i \in D} \frac{\partial s_i}{\partial w_k} \lambda_i \quad (4)$$

to update the NN model parameters. Here, η is the learning rate. The partial derivatives with respect to scores $\frac{\partial s_i}{\partial w_k}$ could be computed in the same way with a traditional BPNN algorithm. The pseudo code of the algorithm is given in **Algorithm 1**.

2.2. Software profiling

The software is implemented with C++, compiled with Microsoft Visual Studio 2008, and optimized for speed with SSE3 option. A typical run with 300 rounds in software costs about 7 hours on the benchmark dataset whose size is 1.22 GB and query number is 15,986. Much more training rounds may be required for an acceptable result. The profiling results show that *FPCalc* and *BPCalc* take 6.11 hours of all. Intensive floating point operations and cache misses are the major reasons to the long computation time.

3. FPGA-BASED ACCELERATOR DESIGN WITH A STREAMING ARCHITECTURE

3.1. AJAW: FPGA-based accelerator platform

AJAW board uses an Altera Stratix-II FPGA (EP2S180C5) as the main computation engine for its good performance/cost ratio at the time we built it. A Xilinx Virtex-5 LXT FPGA (LX50T) is also

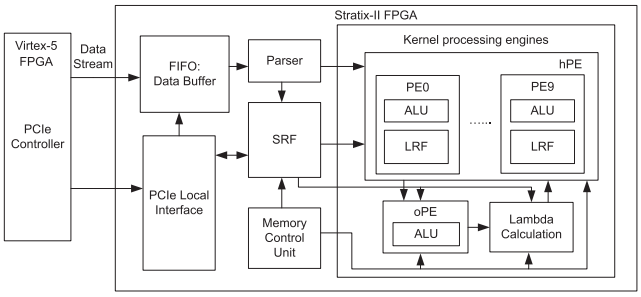

Fig. 1. The block diagram of the FPGA-based accelerator.

used to provide a PCI Express (PCIe) interface with the host computer. The embedded PCIe endpoint hard core supports up to 8 lanes (x8) and 4 GBps bidirectional throughput in theory. AJAW board also supports 2 DDR2 modules which can provide up to 16 GB capacity.

3.2. Streams and kernels

We organize the LambdaRank algorithm implementation into *streams* and *kernels* to expose the inherent locality and concurrency for acceleration [7]. In our design, *streams* contain a set of training data which are separated by queries, and data exchanged among kernels; *kernels* are *FPCalc*, *LambdaCalc*, and *BPCalc*, which are corresponding to the three major routines in the algorithm.

Because of the similarities of *FPCalc* and *BPCalc* in the computation, we map both of them to the so-called hidden-layer processing engine (hPE), which implements the hidden nodes of the NN model. Similarly, the output node is implemented by the output layer processing engine (oPE). $LambdaCalc$ is mapped to a separate processing engine. The structure of the accelerator is presented in figure 1.

3.3. Bandwidth hierarchy

To support the stream processing, we provide three levels of storage that form a data *bandwidth hierarchy* [7]. The first level is composed of the local register files (LRFs), which store data used within *kernels*. The second level is the stream register file (SRF), which is used to exchange data among *kernels*. We map all the inputs and outputs of *FPCalc*, *LambdaCalc* and *BPCalc* to SRF. The third level is the host computer memory, which is used to store the training data. In the accelerator, bandwidths of three levels are: 343.8 Gbps via LRFs, 56 Gbps via SRF, and 6.4 Gbps via the host computer memory. The ratio 54:8.75:1 means 98.4% of all data accesses are captured locally in the FPGA and only 1.6% of all data accesses occur outside the FPGA.

3.4. HW/SW partitioning and data preparation

The software part of the accelerator will i) initialize the parameters of the NN model (line 1 in the pseudo code), ii) organize the training data into a regular format in the order of the hardware component consuming it, and iii) attach some pre-computing results. Then the software sends the preprocessed data to the accelerator hardware with DMA write operations. In each round, the accelerator consumes the training data from the software, then sends NN models back to the software on the host computer. At last, the software stores the training results (parameters of the NN model in each round) to files.

3.5. Parallelism utilization

In the implementation, data dependencies make three major tasks cannot run in a fully overlapped manner. According to the pseudo code, the first procedure in *LambdaCalc* is to sort all the document scores from *FPCalc*. Thus, *LambdaCalc* cannot start until *FPCalc* outputs the first score. Similarly, line 27 in *BPCalc* cannot start until *LambdaCalc* outputs the first λ. That is, line 27 cannot start until *FPCalc* finishes, because each λ needs to calculate over all the scores from *FPCalc*, according to the equation 3. Another dependency is that *FPCalc* cannot start until all parameters of the NN model are updated in *BPCalc*. Restricted by these data dependencies, we design the task-level overlapping scheme to fully utilize the inherent parallelism.

To utilize the data-level parallelism, the hidden node loops (line 5 and line 19 in the pseudo code) are unrolled completely, and each hidden node is implemented with a hPE as described in section 3.2. We implement the hidden node layer with 10 hidden nodes which provides the best learning results. Moreover, we partially unroll the document loops (line 4 and line 20 in the pseudo code) with two sets of computation modules in arithmetic logic units (ALUs) of hPE. Thus, hPE can consume two document features in each cycle.

3.6. Data representation and arithmetic units design

In software, data are represented as double precision floating point numbers. In hardware, we measured the ranking quality of models generated by hardware with a behavior model in C++ for the accelerator. The experiments indicate that the single precision floating point representation is acceptable for the application.

In the hardware implementation, we don't use the floating point arithmetic IP cores provided by the FPGA vendor because of their high latencies. We build an arithmetic unit library for the single precision floating point number, which contains multipliers, accumulators (can be used as adders/subtracters), fixed to float converters, etc. The library only supports parts of the IEEE 754 standard. In detail, we don't implement the logic for handling denormal numbers and some rounding modes except round to even mode, which is the most widely-used in the real applications. Table 1 shows comparisons between our implementations and IP cores generated by MegaWizard in Quartus II 7.1 from Altera. They are all synthesized with EP2S180C5 FPGA.

4. RESULTS

We implement the LambdaRank algorithm with AJAW board. The RTL code is compiled by Quartus II 7.1. The synthesis result shows 80% logic and 30% memory bits usage in EP2S180C5 FPGA. The implementation runs at 100MHz, in which frequency the peak performance is 11.7 GFLOPS.

Table 1. Comparison of single precision floating point arithmetic units on EP2S180C5.

Unit	ALUTs	Latency (cycles)	Freq (MHz)
Adder_Custom	670	3	104.5
Adder_Altera	868	7	106.5
Multiplier_Custom	80	3	133.2
Multiplier_Altera	123	5	137.4

4.1. Speedup analysis and performance model

We run 300 rounds on the benchmark dataset. The pure software takes about 7.1 hours on Intel Xeon 2.40 GHz processor with 2 GB RAM. The accelerator takes only about 32.1 minutes. It shows 13.26X speedup over the pure software. On other datasets, the speedup is observed to be within the range from 13.0X to 17.9X.

We further conduct experiments on datasets with increasing sizes. The experimental results show that the speedup ratio keeps around 13 $\sim$ 15 times with different data sizes from 604 MB to 1.65 GB. The ratio will drop when the dataset is too large to fit into the host memory. Then the performance bottleneck will be the bandwidth of the secondary storage, which may be a hard disk, a RAID or a high performance solid-state disk (SSD).

The performance of the LambdaRank accelerator is determined by the smaller one of the communication bandwidth and the computation bandwidth. The peak PCIe x4 DMA bandwidth of AJAW board is measured to be 800 MBps when transferring 1GB data from host memory to FPGA. The peak computation bandwidth is designed to be also 800 MBps with two 32-bit data streams in parallel (refer to section 3.2). However, when the computation engines are running *LambdaCalc* and *BPCalc*, training data from Virtex-5 are not consumed. Thus, the computation bandwidth could not reach its peak value when averaged over the whole training process. The logic simulation shows that the computation bandwidth can achieve an average of 443 MBps. In-system experiment shows a total bandwidth of about 203 MBps. This degradation comes from pipeline stalls brought by the discontinuous data stream between the host computer and the FPGA. Increasing the data buffer size in the PCIe FPGA can tolerate more variance in the data stream consumption. To remove this bandwidth bottleneck, on-board DDR2 modules can be used as a much larger buffer.

4.2. Quality of hardware generated ranking models

We execute 5 runs for hardware and software respectively with the same settings on the benchmark dataset, and take the best 10 NN models from each run to calculate NDCG on a validation dataset. Figure 2 shows the average NDCG values together with error bars. Values of error are doubled for a better view. The absolute difference of average values is 0.00034, which is much less than the variance 0.00324 among different runs with software.

5. CONCLUSION AND FUTURE WORK

This paper describes our design of the FPGA-based accelerator for LambdaRank. We propose a SIMD streaming architecture to exploit the inherent parallelism and locality in the application, and provide a scalability to the large scale dataset. The implementation can provide an acceleration rate of up to 17.9X on commercial web search datasets. This accelerator has been used by domain experts within Microsoft to reduce the training time.

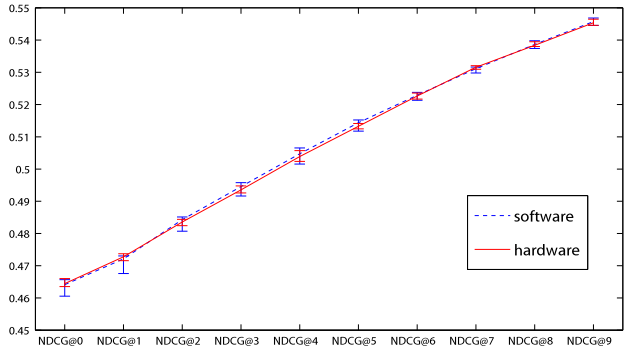

Fig. 2. NDCG of models trained with hardware accelerator and software. 300 rounds on the benchemark dataset.

Several improvements can be applied to further increase the performance of the current implementation. The current PCIe DMA design is going to be upgraded to support PCIe x8 which is expected to double the bandwidth between the host computer memory and the FPGA. With on-board DDR2 modules, a larger data buffer to tolerate the variance of data streaming should also be implemented.

6. ACKNOWLEDGEMENTS

The authors would like to thank Liang-Wei GE, Chong-Qi ZHAO, Yiling HSIEH, Jian OUYANG for their support in this work, and Lei ZHANG for his useful feedback.

7. REFERENCES

[1] *NIPS'05 workshop on learning to rank*, 2005.

[2] C. Burges, T. Shaked, E. Renshaw, A. Lazier, M. Deeds, N. Hamilton, and G. Hullender, "Learning to rank using gradient descent," in *ICML '05*, 2005, pp. 89–96.

[3] C. J. C. Burges, "Ranking as learning structured outputs," in *Proceedings of the NIPS 2005 workshop on Machine Learning*, Dec. 2005, pp. 7–11.

[4] R. R. Christopher J. C. Burges and Q. V. Le, "Learning to rank with nonsmooth cost functions," in *Proceedings of NIPS 2006*, Dec. 2006, pp. 193–200.

[5] S. Brin and L. Page, "The anatomy of a large-scale hypertextual web search engine," *Computer Networks and ISDN Systems*, vol. 30, no. 1-7, pp. 107–117, 1998.

[6] K. Jarvelin and J. Kekalainen, "Ir evaluation methods for retrieving highly relevant documents," in *SIGIR '00*, 2000, pp. 41–48.

[7] U. Kapasi, S. Rixner, W. Dally, B. Khailany, J. H. Ahn, P. Mattson, and J. Owens, "Programmable stream processors," *Computer*, vol. 36, no. 8, pp. 54–62, Aug. 2003.