FPMR: MapReduce Framework on FPGA
A Case Study of RankBoost Acceleration

Yi Shan[1,2] Bo Wang[1,2] Jing Yan[1,2] Yu Wang[1] Ningyi Xu[2] Huazhong Yang[1]
[1]Tsinghua National Laboratory for Information Science and Technology
Department of Electronic Engineering
Tsinghua University, Beijing 100084, China
{shany08, wangb06, j-yan03}@mails.tsinghua.edu.cn
{yu-wang, yanghz}@tsinghua.edu.cn
[2]Hardware Computing Group
Microsoft Research Asia
Beijing, China
{v-yishan, v-jiy, v-wdavid, xu.ningyi}@microsoft.com

ABSTRACT
Machine learning and data mining are gaining increasing attentions of the computing society. FPGA provides a highly parallel, low power, and flexible hardware platform for this domain, while the difficulty of programming FPGA greatly limits its prevalence. MapReduce is a parallel programming framework that could easily utilize inherent parallelism in algorithms. In this paper, we describe FPMR, a MapReduce framework on FPGA, which provides programming abstraction, hardware architecture, and basic building blocks to developers.

An on-chip processor scheduler is implemented to maximize the utilization of computation resources and achieve better load balancing. An efficient data access scheme is carefully designed to maximize data reuse and throughput. Meanwhile, the FPMR framework hides the task control, synchronization, and communication away from designers so that more attention can be paid to the application itself. A case study of RankBoost acceleration based on FPMR demonstrates that FPMR efficiently helps with the development productivity; and the speedup is 31.8x versus CPU-based implementation. This performance is comparable to a fully manually designed version, which achieves 33.5x speedup. Two other applications: SVM, PageRank are also discussed to show the generalization of the framework.

Categories and Subject Descriptors
B.5.1 [**Register-Transfer-Level Implementation**]: Design – *Control design, Data-path design, Styles*

C.3 [**Special-Purpose and Application-Based Systems**]: Microprocessor/microcomputer applications;

General Terms
Performance, Design

Keywords
MapReduce, FPGA framework, RankBoost

FPGA'10, February 21-23, 2010, Monterey, California, USA.

1. INTRODUCTION
Efficient computing of machine learning and data mining has gained much more attention of the computing society in recent years, while it becomes more and more challenging with the ever growing data size and much higher performance requirements. As the physical constraints are preventing frequency scaling of CPUs and the power consumption is becoming a critical problem, parallel computing becomes the dominant paradigm for large scale computing applications. FPGA has been widely explored in various high performance computing applications in recent years [1]. Compared with other parallel computing platforms, such as multi-cores, clusters and GPGPUs, the main advantages of FPGA are i) FPGA is reconfigurable and easy to change functionalities without changing the platform; ii) logic elements in FPGA work in a naturally fine-grained parallel way with high flexibility; and iii) FPGA is one of the best hardware devices that can follow the Moore's Law persistently [2].

However, the popularity of FPGA-based computing is limited by the low programming productivity compared with other platforms, such as GPGPU and multi-core. Practically, the most time-consuming and essential part is usually the hardware architecture exploration and the register transfer level implementation. Although some synthesis tools (e.g. AutoPilot[3], CatapultC[4] and ImpulseC[5]) can generate optimized RTL code from descriptions in high-level programming languages (such as C, C++, or SystemC) and user constraints, developers still need to design sophisticated hardware structures to efficiently map random programs to circuits to achieve an acceptable performance. A. DeHon et al. concluded some design patterns for FPGA-based computing [6], while the abstraction level of proposed guidelines are not utilizing the characteristics of specific application domains.

MapReduce is a parallel programming model proposed by Google [7] for the ease of massive data processing and has been successfully applied to many applications [7, 8, 9, 10]. This model provides two primitives, *map* and *reduce*. As shown in Figure 1, the input data to a computing task is split into many <key,value> pairs and a *map* function processes these pairs to generate a set of *intermediate* <key,value> pairs. The intermediate pairs with the same intermediate key are grouped together and passed to *reduce* function. The communication model within MapReduce is transparent to users so as to alleviate the development efforts. Users only need to design the *map* and *reduce* function. Then the MapReduce runtime framework takes care of the parallel execution by issuing multiple *map* and *reduce* tasks to

computation nodes. MapReduce greatly reduces the complexity of designing parallel computing programs, and provides efficiency for data-intensive applications [7]. In [9], many standard machine learning algorithms has been adapted to the MapReduce framework on multicore machines, illustrating its benefits to the machine learning community.

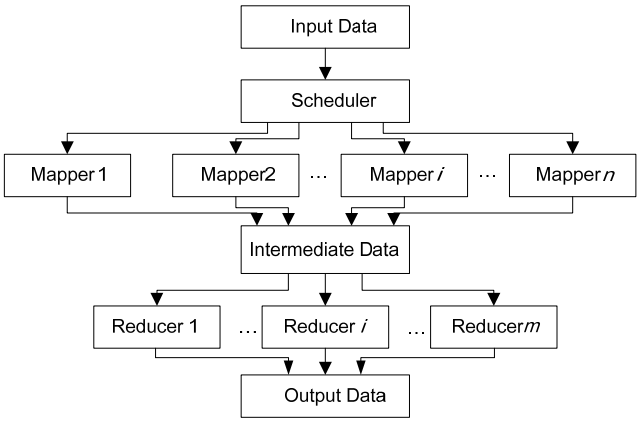

Figure 1. MapReduce Data Flow

MapReduce model has been explored on most parallel computing platforms in the past few years. Google implemented the first and largest MapReduce system on its clusters [7]. A multi-core version, Phoenix [11, 12], was later developed to explore the parallelism on shared memory systems. Phoenix automatically manages thread creation and dynamic task scheduling. Its problem is, the memory and I/O usage of one task may detrimentally affect others [12] and this problem becomes more crucial when the thread number increases. Meanwhile, the high power consumption of multi-core chips will be a wall for their massive usage. On general purpose GPU platforms, MapReduce framework was also explored [13]. However, GPU prefers coalesced memory access pattern, which makes it fumble while dealing with complex data structure and the SIMT architecture restricts its computation performance to handle irregular applications. In [14], a MapReduce framework on Cell clusters was implemented. Yeung, et.al [15] adopted both GPU and FPGA to implement a MapReduce framework. This framework leaves scheduling work to the host CPU and uses GPU and FPGA as co-processors.

Machine learning and data mining algorithms usually operate iteratively on a large corpus of regular data, and there are coarse-grained parallelisms exist in these data. Thus, it is easy to utilize the data locality and parallelism with streaming processing and parallel computing. FPMR provides mappers and reducers to utilize the parallelism, and the data access scheme provides efficient streaming access to the training data.

In this paper, we focus on a general and scalable MapReduce framework on FPGA to shorten the development cycles of the FPGA-based computing for machine learning and data mining. In this framework, multi-level parallelism can be utilized, ranging from bit-level to task-level. To demonstrate the feasibility of the proposed framework, we implement RankBoost algorithm, an efficient learning algorithm which is extensively used in real applications. The results show that our proposed design simplifies the hardware programming significantly with an appreciable speedup. The accelerator achieves 31.8x speedup (by 146 *map* instances and 1 *reduce* instances) compared with the results of a software implementation. We further expound its performance bottleneck, resource utilization, and the achievable data bandwidth, to discuss the implementation and optimization of the framework for such data-intensive applications. Specially, this paper makes following contributions.

- The reconfigurable ability of FPMR framework allows designers to place various *mappers* and *reducers* on chip to achieve the best performance according to the characteristics of the device and the application.
- An *on-chip* dynamic scheduling policy is adopted so as to maximize the utilization of computation resources and achieve better load balancing. Meanwhile, task control and communication are hidden away from the designers so that designers can focus on the application itself.
- An efficient data access scheme is implemented to maximize the data reuse and alleviate the bandwidth bottleneck. Dynamic data synchronization can be also achieved by this data control scheme of the framework.

To the best of our knowledge, this is the first on-chip scheduled MapReduce framework on FPGA. With this framework, the development cycles can be greatly reduced.

The remainder of this paper is organized as follows. Section 2 introduces our FPGA-based design of the MapReduce framework. Section 3 invokes an application: RankBoost on FPGA to serve as a case study of FPMR. Section 4 shows the experimental results and discussions of the case study. Section 5 discusses the mapping of Support Vector Machine，PageRank onto FPMR. Section 6 concludes the paper.

2. FPMR FRAMEWORK

In this section, we will introduce the FPMR framework. Dedicated processors are designed for different applications under FPMR framework. Dynamic on-chip scheduling and efficient data control are also included in FPMR to hide the task control, communication, and data synchronization away from designers.

2.1 Framework Overview

The MapReduce data flow can be simplified as follows.

map : < key , value > → *intermediate* < key , value >
reduce : *intermediate* < key , value > → result

The initial <key,value> pairs are prepared by CPU and then transferred to the FPGA through PCI-E bus or CPU bus, e.g. HyperTransport or FSB. The configuration parameters shown in Table 1 are written down to the registers in FPGA.

Table 1. Configuration Parameters

Name	Description
#map_task	number of tasks for mappers
#reduce_task	number of tasks for reducers
#data	number of <key,value> pairs

Then the *map* and *reduce* operations are done by *mappers* and *reducers* on FPGA. What is more, task scheduling and data dispatching are also done on chip. The FPMR framework shown in Figure 2 is partitioned into four parts: *processors* (mapper/reducer/merger), *processor scheduler*, *data controller* and *storage*.

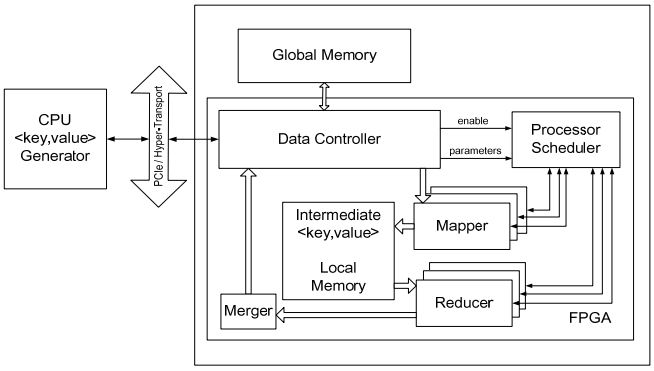

Figure 2. FPMR Framework

The *mappers* process the initial input <key,value> pairs and generate the *intermediate* <key,value> pairs. The *reducers* then merge the intermediate pairs to obtain the final results. In some applications, the outputs of reducers need to be further processed to get a single result, in which case a *merger* will be implemented. The *processor scheduler* generates control signals to schedule mappers and reducers. The *data controller* takes charge of communicating with the host CPU, dispatching data to the mappers, and receiving data from the reducers.

The basic work flow and scheduling policy are shown in Figure 3.

1. Generate < key, value > pairs on the host.
2. Write the configuration parameters to registers on FPGA.
3. Initialize DMA data transferring, copy the <key, value> pairs from the CPU to FPGA board.
4. The processor scheduler assigns the tasks to each mapper.
5. Mappers process the assigned < key, value > and store the generated *intermediate* <key, value> in the local memory under the control of data controller.
6. When a mapper finishes its job and there are jobs left, the processor scheduler will assign another job to it.
7. When some intermediate pairs are generated and one or more reducers are idle, the scheduler will assign the intermediate pairs to idle reducers.
8. When all the tasks are finished, the results are returned to the host main memory by the data controller.

Figure 3. The basic work flow and scheduling policy of FPMR

From Figure 3, it can be seen that our on-chip dynamic scheduling policy helps to achieve higher computation resources utilization, especially for applications whose parallel tasks take unequal time. When a *mapper* or *reducer* finishes earlier than others, it will *take* some more work instead of staying idle.

2.2 Processor

There are two types of processors on chip, *mappers* and *reducers.* Mappers and reducers are specifically designed according to the target application. They are both triggered and their tasks are assigned by the processor scheduler. Mappers request <key,value> pairs from data controller, generate the *intermediate* <key,value> pairs, and store the *intermediate* <key,value> pairs in the local memory. Then reducers deal with a set of intermediate pairs to obtain the final results. The ratio of mappers to reducers is determined by the workloads of these two parts. For those applications with complex computation, pipelined strategies will be adopted to achieve higher data throughput.

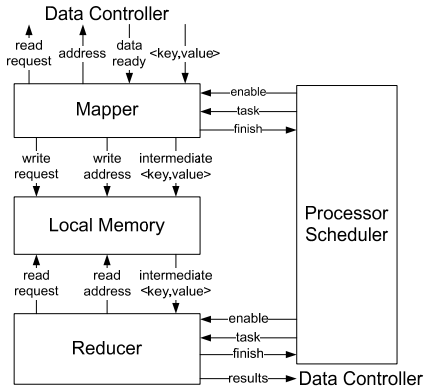

Figure 4. The data exchange between mappers and reducers

It is worth noting that the working time of mappers may be different from one to another in some data-dependant algorithms, so that a processor scheduler is essential for the collaboration between mappers and reducers. The data exchange between mappers and reducers is shown in Figure 4.

The interface of the mapper is designed as follows.

```
module mapper(...);
   input  enable, task_id
   input  [m:0] key;
   input  [n:0] value;
   output finish, read_request, write_request;
   output [j:0] int_key;
   output [k:0] int_value;
   output [i:0] read_addr, write_addr;

   // user defined codes below
   ...

Endmodule
```

The interface of the reducer is designed as follows.

```
module reducer(...);
   input  enable, task_id
   input  [m:0] int_key;
   input  [n:0] int_value;
   output finish, read_request, write_request;
   output [k:0] result;
   output [i:0] read_addr, write_addr;

   // user defined codes below
   ...

Endmodule
```

The designers only need to pay attention to the internal structure of mappers and reducers by using the interfaces within these two modules.

2.3 Processor Scheduler

Processor scheduler is designed to dynamically utilize the hardware resources by monitoring the status of each mapper and reducer. There are two sets of queues in the processor scheduler. One queue set is for mappers and the other queue set is for reducers. Each queue set consists of two queues, one queue for idle processors and the other for pending tasks. The idle processor queue records the id of the idle mappers or reducers. The configuration parameters, *#map_task* and *#reduce_task* define the task number and are used to initialize task queues. The numbers of mappers and reducers are decided by the designers based on the available FPGA resources.

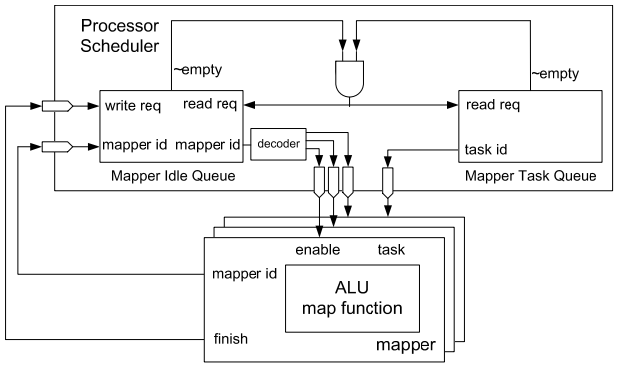

Figure 5. The internal structure of mapper scheduler

To better illustrate the scheduling policy, here we take the idle processor queue and task queue for mappers as an example. The mechanism and scheduler structure are the same for reducers. Figure 5 is the internal structure of a mapper scheduler. If both the mapper idle queue and mapper task queue are not empty, the processor scheduler will extract the first task in the task queue and assign it to the first mapper in the idle queue. Then this mapper's id is also extracted from the idle mapper queue. When a mapper finishes its task, the processor scheduler will add its id into the idle queue again to wait for the next task. The intermediate pairs generated by a mapper also have an id which will be added into the reducer task queue. In such a scheme, mappers and reducers cooperate with each other to keep all the processors as busy as possible.

2.4 Storage Hierarchy and Data Controller

There are three levels of storage in this framework. The first level is the global memory, which stores the initial <key,value> pairs. The second is the local memory, which stores the intermediate <key,value> pairs and serves as the shared memory for mappers and reducers. The third level is the register file in each processor, which is for temporary variables, configuration parameters, and results.

Global memory For machine learning and data mining applications, the <key,value> pairs usually occupy large amount of memory, so large capacity memory will be used, such as DDRx SDRAMs. Not only the large capability and high bandwidth can be provided, but also the scalability can be easily achieved by implementing *multiple* DDRx SDRAMs.

Local memory The local memory can be implemented as on-chip RAMs. The intermediate results obtained from a mapper are stored in the local memory and the reducer will fetch the intermediate data from the local memory. On-chip RAMs can provide this shared memory functionality with low access latency. Multiple RAMs can be implemented, and they can be accessed by mappers and reducers simultaneously.

Register file The register file stores the temporary variables, parameters of the framework, and results during the processor operation. This level of memory can be accessed extremely fast, therefore the performance will be increased by well utilization of the register files.

Data Controller

The data controller is responsible for the following three functions: 1) to communicate with CPU and transfer data between the host and the on board memory; 2) to dispatch requested data to mappers; 3) to store the output data from reducers.

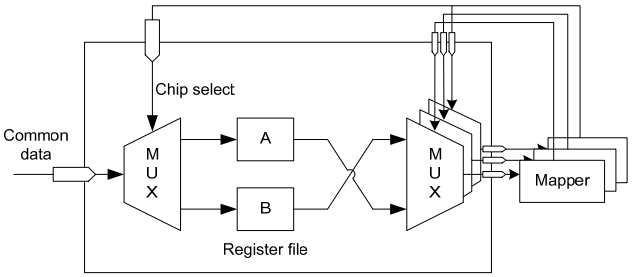

Figure 6. The internal structure of common data path

To transfer data between the host and FPGA board, four data transmission types are used: write/read register and write/read memory. Large scale data write/read can adopt the DMA way.

Several mappers may request data at the same time by sending requests to the request queue in the data controller. These requests will be satisfied one after another. Similarly, when reducers send requests for returning the output results to the global memory, the requests will also be inserted into the data returning queue and these requests will be also satisfied sequentially one after another. It is worth noting that only when the result is stored back to the on-chip memory, the reducer will be set to be idle again.

In machine learning and data mining applications, some parts of the data are the same for all mappers and needed to be transferred to all the processors when a new iteration begins. In our FPMR framework, a ***common data path*** **(*CDP*)** is built in the data controller to avoid the redundant data transfer. Two sets of registers inside the data controller are connected to the common data path. A ping-pong strategy is adopted to control these two register sets. Figure 6 is an illustration of CDP. When the mapper is reading register set A, the chip select of the set B is marked high and the common data from the global memory is transferred to set B at the same time. This strategy can reduce the occupation of the memory bandwidth; while overlapping the common data transfer time by computation time. The common data path is used in the RankBoost acceleration and SVM analysis.

3. A CASE STUDY: RANKBOOST

In this section, we first introduce the primitives of RankBoost [16], a recently proposed ranking algorithm. Then we show the detailed FPGA implementation based on our FPMR framework.

3.1 RankBoost Introduction

RankBoost [16] is a Boosting algorithm targeting for rankings. Giving an exact and complete ranking for large scale objects is difficult. RankBoost is a promising algorithm for this problem by combining many "weak" hypothesises which are partly or nearly right. The result ranking function will be highly accurate by many rounds of training on large scale dataset.

The training data set is composed of documents. Each document d is expressed by a feature vector $\{ f_i(d) \mid, i = 1, 2, .. N_f \}$ indicating the relevance with the query feature. A distribution $D(d_0, d_1)$ is defined as the importance of document. $D(d_0, d_1)$ is positive if d_0 is more relevant than d_1. This distribution covers all the document pairs and is updated in each training round. The flow of RankBoost is described in Algorithm1.

Algorithm 1 : RankBoost Algorithm

Input : $D_0(d_0, d_1)$, $\pi(d)$ and $f(d)$ of all documents

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

for $t \leftarrow 1$ **to** T

Train **WeakLearn** using distribution D_t

WeakLearn returns a weak hypothesis h_t and weight α_t

Update distribution weights: for all (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 :

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

endfor

The most time consuming procedure of RankBoost is WeakLearn, which consumes more than 95% execution time [17]. WeakLearn gives a weak ranking hypothesis h based on the features ofdocuments and the current distribution. $h(d)$ is a binary threshold function, i.e. for any document d

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

where $f_i(d)$ denotes the value of feature f_i for document d, and θ is a threshold value.To find the best $h(d)$ in each round, WeakLearn needs to check all the possible combinations of feature f_i and threshold θ to ensure the accuracy.

In WeakLearn procedure, the feature f_i and threshold θ are found so that h has the maximum ranking correctness r, defined as:

$$r_{i,\theta} = \sum_{d_0, d_1} D(d_0, d_1)(h_{i,\theta}(d_0) - h_{i,\theta}(d_1))$$

To reduce the computation complexity, we define the π value as follows, which is updated in each round.

$$\pi(d) = \sum_{d'}(D(d', d) - D(d, d'))$$

Then r can be obtained as follow.

$$r_{i,\theta} = \sum_{d} h_{i,\theta}(d)\pi(d) = \sum_{f_i(d) \ge \theta} \pi(d)$$

In [17], to map the algorithm to hardware more efficiently, the WeakLearn is transformed from continuous style to discrete style by discretizing the continuous $f_k(d)$ to several separate *bins*. The threshold value θ_s for each *bin* are calculated as follows.

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

where $f_{\max}^k$ and $f_{\min}^k$ are maximum and minimum value of k-th feature $f_k(d)$ with respect to all documents. To accommodate with the hardware structure, each feature is divided into 256 *bins*.

Then the *bin* value for $f_k(d)$ is mapped as follows.

$$bin_k(d) = floor(\frac{f_k(d) - f_{\min}^k}{f_{\max}^k - f_{\min}^k} - 1)$$

After transformation, the correctness $r_{i,\theta}$ is obtained through finding the max $integral_k(i)$. To calculate $integral_k(i)$, firstly a histogram of $\pi(d)$ over feature f_k should be built.

$$hist_k(i) = \sum_{d:bin_k(d)=i} \pi(d), \quad i = 0,...,(N_{bin} - 1)$$

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:

$$integral_k(i) = \sum_{a>i} hist_k(a), i = 0,...,(N_{bin} - 1)$$

The discrete WeakLearn procedure is shown in Algorithm 2. Firstly, an integral histogram is built over all documents. After finding the value $integral_{max}$ as well as the corresponding feature index f_{max} and bin index bin_{max}, the hypothesis h and weight α_t are calculated in the following form.

$$h(d) = \begin{cases} 1, & if\ bin_{f\max}(d) > bin_{\max} \\ 0, & if\ bin_{f\max}(d) \le bin_{\max} \ \text{ or } bin_{f\max}(d) \text{ is undefined} \end{cases}$$

$$\alpha = \frac{1}{2}\ln\left(\frac{1 + integral_{fmax}}{1 - integral_{fmax}}\right)$$

Algorithm 2 : WeakLearn Procedure in RankBoost Algorithm

Input : $\pi(d)$ for t-th round and $bin(d)$ of all documents

Output : a weak hypothesis h_t and weight α_t

(1) **for** $k \leftarrow 0$ **to** N_f

(2) **for** $d \leftarrow 0$ **to** N_d

(3) $hist_k(bin_f(d)) \leftarrow hist_k(bin_f(d)) + \pi(d)$

(4) **Endfor**

(5) **for** $i \leftarrow N_{bin} - 1$ **to** 0

(6) $integral_k(i) \leftarrow hist_k(i) + integral_k(i+1)$

(7) **Endfor**

(8) **Endfor**

(9) **Find** the max { $integral_{fmax}(bin_{max})$ }

(10) **for** $d \leftarrow 0$ **to** N_d

(11) **if** $bin_{fmax}(d) > bin_{max}$

(12) $h_t(d) \leftarrow 1$

(13) **Else**

(14) $h_t(d) \leftarrow 0$

(15) **Endfor**

(16) $\alpha_t = \frac{1}{2}\ln\left(\frac{1 + integral_{fmax}}{1 - integral_{fmax}}\right)$

3.2 RankBoost on FPMR Framework

In this subsection, the mapping strategy and hardware implementation of RankBoost are described in detail as a case study of FPMR framework.

3.2.1 Mapping RankBoost to FPMR

The most time-consuming, WeakLearn procedure, will be done on FPGA. Data pair initialization and π values update are assigned to the software.

To map WeakLearn procedure onto MapReduce framework, the procedure is decomposed into two parts, *histogram building* and *integral histogram calculation*. Each mapper is responsible to build a histogram for a feature (line 2-4 in Algorithm 2) and a reducer is responsible to calculate the integral on these histograms (line 5-7 in Algorithm 2). In accordance with the mapping scheme, the initial pairs and the intermediate pairs are defined as follows.

initial < key , value >	: < *fi*, ($bin_{fi}(d)$, $\pi(d)$)>
intermediate < key , value >	: <*fi*, $hist_{fi}$>

The denotations above are described below.
fi is the *feature index*;
$bin_{fi}(d)$ is the transformed *fi*-th feature values of all documents;
$\pi(d)$ is the π value of all documents;
$hist_{fi}$ is the mapper-generated histogram of the *fi*-th feature.

The *map* function for RankBoost can be described as follows.

```
map (int key, pair value):
  // key : feature index fi
  // value : document bin_fi, document π
  for each document d in value :
     hist(bin_fi(d)) = hist(bin_fi(d)) + π(d)
  EmitIntermediate (fi, hist_fi);
```

Here, only one histogram building task is assigned to one mapper. Otherwise the intermediate <key, value> pairs generated by the mappers will be too large to store in the on-chip memory.

The *reduce* function for RankBoost can be described as follows.

```
reduce (int key, array value) :
  // key : feature index fi
  // value : histograms hist_fi , fi = 1…N_f
  for each histogram hist_fi
     for i = N_bin – 1 to 0
        integral_fi(i) = hist_fi(i) + integral_fi(i+1)
  EmitIntermediate (fi, integral_fi)
```

The ratio of mappers to reducers is determined by their relative throughput. The computation complexity of map function is $O(N_f \times N_{doc})$ which is several magnitudes higher than that of reduce, which is only $O(N_f \times N_{bin})$. As a result, only one reducer is implemented while up to 146 mappers are realized. The number of mappers is limited by the on-chip resources, which will be further discussed in Section 4.

When integral histograms are built over all the features, the *merger* finds the maximum integral value as the output result. The update of weak hypothesis $h_t(d)$ and weight α_t are done on the host.

In this way, tasks are assigned to different mappers and reducers dynamically. The data requests of these processors are processed by data controller automatically. The on-chip processors work concurrently. So, we only need to map the applications onto *map* and *reduce* functions, and design the specific *mapper* and *reducer*, the parallelism can be achieved naturally.

3.2.2 Hardware Implementation based on FPMR

The RankBoost on FPMR framework is shown in Figure 7.

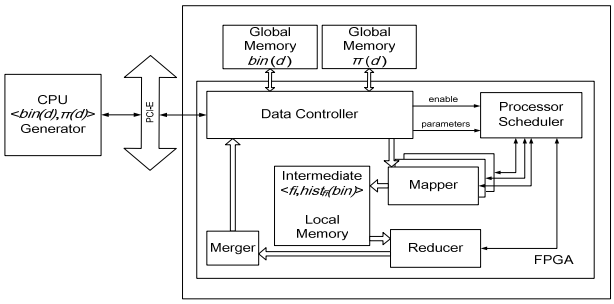

Figure 7. RankBoost on FPMR Framework

In this design, two conventional DDR2 SDRAMs are used as the global memory, separately for *bin* and π values. They are stored by features for access convenience. The *bin* values stay the same for all training rounds, so only the π values need to be transferred each round. At the end of WeakLearn procedure, the maximum integral histogram value, corresponding *bin* and *feature* will be returned to the host to update the π values. The π value calculation and weight updating are the major software computation tasks.

Mappers and reducers are in charge of building histogram and integral histogram respectively. Processor scheduler controls the working status of these processors by dynamically assigning tasks. The three level storage and data controller make the memory hierarchy efficient for the system.

Mapper

In a mapper, a histogram is built for every feature. The generated histogram for a feature, $hist_f$, is stored in a dual port RAM. For every document, $bin_f(d)$ serves as the read address and the target $hist[bin_f(d)]$ value will be added by the corresponding π. After several cycles' delay of floating point adder, results will be stored in the RAM, and the same $bin_f(d)$ also serves as the write address. Two adjacent documents may have the same *bin*, the second add operation must wait until the previous results are updated in the *hist* RAM. After all the add operations, the results in the hist RAM are transferred to the corresponding Local Memory.

It will be explained in section 5 that although a pipeline is not used here, we increase the number of mappers in order to avoid the processors' computation capability to become the performance bottleneck. When all documents of a feature are processed, the generated histogram is written back to the local memory for reduction.

The implementation of *mapper* is shown in Figure 8.

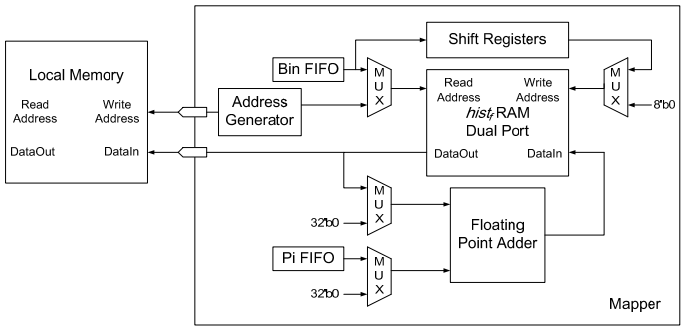

Figure 8. The internal structure of mapper

Reducer

In a reducer, an integral histogram will be built based on the histograms for all the features that are built by mappers. The reducer requests data from the Local Memory that is addressed by the task. There is also a floating point adder to build the integral histogram by accumulating the $hist_f$ of a feature from the last to the first. The add operation also has to be done sequentially, and waits for several cycles for the results to perform the next add. An integral histogram is built over each feature and the maximum integral value is picked out by the floating point comparator. After all the features are handled, the maximum value, the corresponding feature index f_i, and the *bin* value will be sent back to the host.

Reducer's implementation is shown in Figure 9.

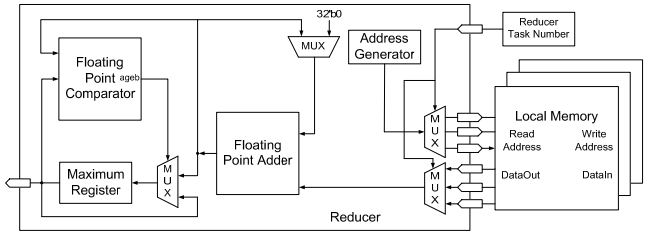

Figure 9. The internal structure of reducer

Processor scheduler

When we finish transferring the <key,value> pairs from host CPU to FPGA board, the processor scheduler starts working. It activates or halts *mappers* and *reducers* according to the status of the *mapper*/*reducer* queue and task queue.

When all the pairs are processed by *mappers* and *reducers*, scheduler sets the finish register to high so as to tell the software to read the output data format and fetch the result via DMA read. Then, the updating process starts to generate the $\pi(d)$ for the next round.

Storage hierarchy and data controller

Storage hierarchy is designed for storing the data on different levels and takes advantage of the locality. Two DDR2 SDRAMs are used as global memory for *bin* and π values. Intermediate data, *hist*, are stored in multiple on-chip RAMs. Mappers and reducers can share these local memories for data exchange. Register files are used to store temporary variables, parameters of framework, and results of WeakLearn.

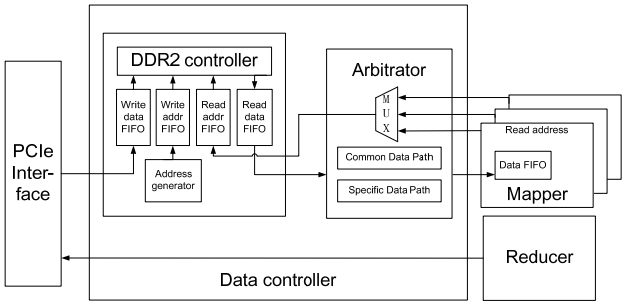

Figure 10. The internal structure of data controller

Data controller is responsible for transferring π and *bin* values from the host to on board memory, dispatching these data from global memory to mappers and returning the maximum integral histogram to the host. Figure 10 is an illustration of data controller. In this system, it takes several clock cycles for DDR2 memory to satisfy a mapper's request. To access memory more efficiently, N pairs are returned to a mapper with respect to a request. Then in the following $N \times t_{compute}$, this mapper will keep busy and it is the turn for other mappers to fetch data. In accordance to the fact that the π values are the same for all mappers for each data fetch, a common data path (CDP) is used to avoid redundant data transfer. To cooperate with this common data path, two sets of registers are defined inside the data controller for the ping-pong strategy. In this way, the time for π values fetching will be overlapped by computation. *Reducers* only need to access the local memory for the intermediate data, so their interconnection to data controller is only the result output.

4. EXPERIMENTAL RESULTS

This section introduces the experimental setup and presents the results. The framework with CDP and without CDP are both tested and theoretically analyzed. The scalability of this framework is also discussed in this section.

4.1 Experimental Setup

To test the performance of the RankBoost acceleration on FPMR, a real world dataset for a commercial search engine is used. The dataset information is illustrated as Table 2. The feature value of each document is firstly compressed into an 8-bits *bin* value (0~255) which occupies only 1 byte. Four *bin* values are merged into a 32-bits integer for storage. The π value is stored in the single-precision float type which occupies 4 bytes memory.

Table 2. Benchmark Dataset

#documents	1,196,711
#features	2,576
#pairs	15,146,236
data size	2.89 GB

A computer with an Intel Pentium 4 3.2GHz processor, 4GB DDR400 memory is used as the platform for software implementation. The FPGA is Altera Stratix II EP2S180F1508. Quartus II 8.1 and ModelSim 6.1 are used for hardware simulation. Based on the critical path delay and the practical bandwidth of PCI-E, the frequency is set to 125MHz. Two Micron 667MHz DDR2 SDRAM models are used in the simulation. The theoretical bandwidth is as follows, while the actual bandwidth is about 2GB/s.

Theoretical Bandwidth = 667MHz×64bits/8=5.3GB/s

Each mapper is responsible for millions of documents with respect to a feature while the reducer only processes the 256-bin histograms generated by each mapper. The workload of mappers is much heavier than that of the reducer, so the bottleneck may lie on one of the following two aspects: 1) the computation ability of mappers and 2) the data bandwidth between global memory and mappers. In the framework without CDP, the bandwidth is the bottleneck due to the massive redundant memory access. After using CDP, the logic resources on FPGA become the performance limitation.

4.2 Experimental results without CDP

Table 3 shows the results of a pure software implementation and FPGA acceleration with up to 64 mappers and 1 reducer.

Table 3. Execution time on CPU and FPMR (without CDP)

#mapper	#reducer	WL/s	Total/s	Speedup	
				WL	Total
1	1	320.89	321.96	0.325	0.327
2	1	160.45	161.52	0.650	0.652
4	1	80.224	81.293	1.300	1.296
8	1	40.112	41.181	2.600	2.559
16	1	20.056	21.125	5.200	4.988
32	1	10.090	11.159	10.33	9.443
52	1	6.228	7.297	16.74	14.44
64	1	6.228	7.297	16.74	14.44
Optimized software		104.30	105.37	1	1

(*) WL stands for the WeakLearn procedure.

The WeakLearn procedure takes up to 97% computation time. This time-consuming procedure achieves up to 16.74x speedup in our FPMR framework. Due to the time spent on the weight updating and π value calculation, the total speedup is 14.44x.

Because two DDR2 memories are used for *bin* and π respectively, the bit-width for both *bin* and π are 128 bits. A maximum of 16 *bin* values and 4 π values can be fetched at one clock cycle. Then the π value throughput will become the bottleneck, because one *bin* value corresponds to one π value. For one data pair $<bin(d),\pi(d)>$, it takes 13 cycles for a mapper to process. So 52 clock cycles are required for a mapper to finish a 4 pair suit. As a result, at most 52 mappers can be implemented to achieve the best performance. If more mappers are added, no more performance gain can be obtained since the bottleneck now is the π value memory bandwidth rather than the computation power. Figure 11 is the sequence chart of mappers without using CDP.

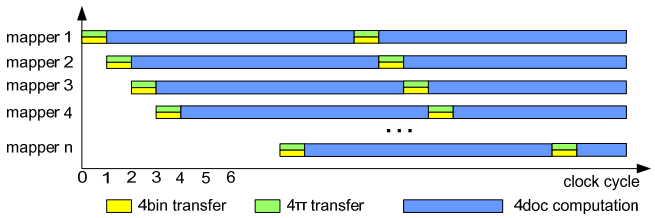

Figure 11. The sequence chart of mappers without CDP

4.3 Experimental results with CDP

In the above design, the system bottleneck is the π value memory bandwidth. However, the π values belong to the common data and the redundant transfer can be avoided using common data path. The sequence chart of mappers with common data path is shown in Figure 12.

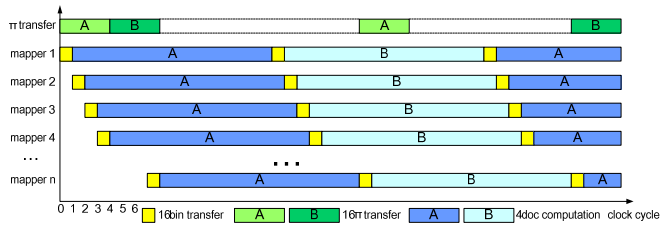

Figure 12. The sequence chart of mappers with CDP

As we can see, the time for π value transfer is overlapped by computation. To fully utilize the *bin* memory bandwidth, 16 *bin* values are fetched at a time. The ping-pong memory to store π values containts 64 bytes so that up to 16 π values can be prefetched. In this way, the π values of the same documents need to be read only once from DDR2 memory and will no longer be the bottleneck. To fully utilize the bandwidth of π value memory, 16 *bin* values should be fetched at a time. The π value throughput is as follows.

$$16\times 8\text{bits}\times 125\text{MHz} = 2\text{ GB/s}$$

Each data fetch requires $13\times 16 = 208$ clock cycles to process, as a result, a maximum of 208 mappers can be placed on chip to achieve the maximum throughput. However, due to the FPGA resource limitation, only 146 mappers can be placed on chip. The experimental results for mappers with CDP are shown in Table 4.

Table 4. Execution time on CPU and FPMR (with CDP)

#mapper	#reducer	WL/s	Total/s	Speedup	
				WL	Total
1	1	320.9	321.96	0.33	0.33
2	1	160.5	161.52	0.65	0.65
4	1	80.22	81.293	1.30	1.30
8	1	40.11	41.181	2.60	2.56
16	1	20.06	21.125	5.20	4.99
32	1	10.09	11.159	10.33	9.44
52	1	6.228	7.297	16.74	14.44
64	1	5.107	6.176	20.42	17.06
128	1	2.616	3.685	39.87	28.59
146	1	2.242	3.311	46.52	31.82
Optimized software		104.3	105.37	1	1

(*) WL stands for the WeakLearn procedure.

In Table 4, the speed up of WeakLearn procedure is expected to be linear until the mapper number reaches 146 while achieving 46.52x speedup. With the common data path, the performance of WeakLearn procedure can be 2.7 times of that without CDP. The total speedup, 31.8x, is comparable with a fully manually designed version [17] which achieved 33.5x speedup.

4.4 Scalability

Figure 13 shows the theoretical speedup for different mapper/reducer ratio. The WeakLearn speedup is linear before the maximal mapper number is reached. The total speedup is not linear due to π calculation and weight updating. The CDP method can greatly relieve the bandwidth pressure and extend the maximal mapper number to 208, along with approximate 4x speedup than the system without CDP.

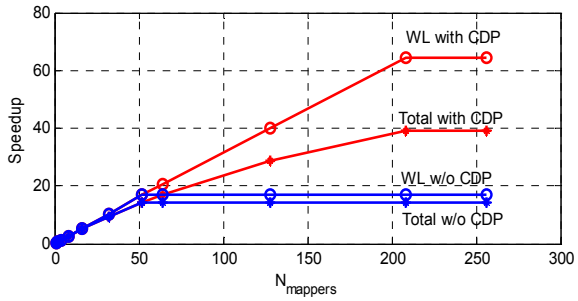

Figure 13. The speedup of different mapper numbers

Practically, due to the resource limitation of Stratix EP2S180 FPGA, only a maximum of 146 mappers can be placed on chip before the new data bandwidth limit is reached. The resource occupation is listed in Table 5.

Table 5. FPGA resource occupation

Mapper	1	2	4	8	16
ALUT	1%	2%	3%	5%	10%
Register	1%	2%	4%	6%	11%
Mapper	32	52	64	128	146
ALUT	19%	31%	38%	75%	86%
Register	17%	32%	39%	81%	89%

A higher performance can be achieved when using FPGAs with more ALUTs and registers. Our framework is scalable and can utilize the maximal resources of the underlying devices.

5. DISCUSSION

A large variety of applications can be accelerated in MapReduce framework. In [18], ten machine learning applications are chosen to be accelerated with MapReduce. All of them can be fit into our framework. Here, two examples on machine learning and data mining are selected to illustrate the mapping methods as well as to demonstrate FPMR's ability of dealing with computation-intensive and load-unbalancing problems.

5.1 Support Vector Machine

Support Vector Machine (SVM) [19], is a solution of the classification and nonlinear function estimation problems based on a convex quadratic programming (QP). An efficient approach for SVM training is the Sequential Minimal Optimization (SMO) [20]. To fit the SMO approach into our framework, the algorithm is decomposed into map function and reduce function, which is similar to [21]. A brief description is shown in Algorithm 3.

Algorithm 3 : SVM Training

Input : training data x_i, label y_i, $\forall i \in \{1...n\}$

Output : weights α_i

(1) Initialize : $\alpha_i, f_i, b_{high}, b_{low}, i_{high}, i_{low}$

(2) **repeat**

(3) **for** $i \leftarrow 1$ **to** n

(4) $$f_i' \leftarrow f_i + (\alpha_{i_{high}}' - \alpha_{i_{high}}) y_{i_{high}} \Phi(\vec{x}_{i_{high}}, \vec{x}_i) + (\alpha_{i_{low}}' - \alpha_{i_{low}}) y_{i_{low}} \Phi(\vec{x}_{i_{low}}, \vec{x}_i)$$

(5) **Endfor**

(6) Compute $b_{high}, b_{low}, i_{high}, i_{low}$

(7) $b_{high} = \min \{f_i : i \in I_{high}\}$, $b_{low} = \max \{f_i : i \in I_{low}\}$

(8) Update α_{ihigh} and α_{ilow}

(9) **until** $b_{low} < b_{high} + 2\tau$

The training dataset and initialized variables are firstly transported to the global memory on FPGA. Then according to the two selected weights, α_{ihigh} and α_{ilow}, the scheduler assigns tasks to **mapper**s to update the Karush-Kuhn-Tucker optimality conditions, i.e. update f_i, for the remaining set of weights (line 4 in Algorithm 3). It is obvious that this operation can be naturally parallel since no data-dependent lies between different f_i. The corresponding data are sent to each mapper from global memory by data controller. When a mapper finishes its work of calculating the new weight, the new weight is stored into the local memory and another weight update job is assigned to it. The f_i update is the most time-consuming part due to the large number of documents and the complicated calculation of kernel $\Phi(x, x_i)$. For illustration, the most popular kernel, the Gaussian kernel, is calculated as below.

$$\Phi(\vec{x}_i, \vec{x}_j, \gamma) = \exp\left\{-\gamma \left\| \vec{x}_i - \vec{x}_j \right\|^2\right\}$$

Meanwhile, **reducer**s are responsible for finding the two maximally violating weights and updating the index set (line 6-8 in Algorithm 3). When the two new candidate weights are found, scheduler updates them and issues the next round. This loop will end when all the points meet the optimality condition.

The data need to be transferred are three feature vectors, x_{ihigh}, x_{ilow} and x_i, which may contain more than thousands of elements. It is worth noting that, in each training round, the feature vectors x_{ihigh} and x_{ilow} are the same for all mappers, because all mapper share the same index i_{high} and i_{low}. Therefore, these two vectors should be transferred to all mappers using the *common data path* before the first mapper using these data. This strategy can save up to 2/3 bandwidth and further allow more parallel mappers.

Compared with RankBoost which is a data-intensive application, SVM is both data-intensive and computation-intensive. Then logic resources of the underlying FPGA will become the major bottleneck.

5.2 PageRank

PageRank [22] is a method for computing the relative rank of web pages based on the Web link structure. Algorithm 4 is the power method for PageRank computation.

Algorithm 4 : Power method of PageRank

Input: web matrix A, escape vector E, initial ranking vector R_0

Output: final ranking vector R

(1) Initialize R randomly to be R_0, then let $k \leftarrow 0$

(2) **repeat**

(3) Compute $R_{k+1} \leftarrow AR_k$

(4) $d \leftarrow \|R_k\|_1 - \|R_{k+1}\|_1$

(5) $R_{k+1} \leftarrow R_{k+1} + dE$

(6) $k \leftarrow k + 1$

(7) **until** $\| R_{k+1} - R_k \| < \varepsilon$

The most time-consuming part of the PageRank computation is Step (2), which takes more than 95% of the total execution time. Due to the huge number of web pages, the web-matrix is stored in a sparse format. So Step 3 is to perform a sparse matrix vector multiplication (SpMV). The parallelism in this step can be explored in a MapReduce way. Algorithm 5 is the computation of SpMV with CSR(Compressed Sparse Row) format sparse matrix. The CSR format matrix consists of three arrays, A_{val} for the value of non-zeros, A_{col} for the column index of corresponding non-zeros and A_{row} for the serial number of the first non-zeros in a row.

Algorithm 5 : Sparse Matrix-Vector Multiplication ($Y = A \cdot X$)

Input : square matrix A of size N_{row}, vector X

Output : vector Y

(1) **for** $i \leftarrow 0$ **to** $N_{row} - 1$

(2) $r_{begin} \leftarrow A_{row}[i]$

(3) $r_{end} \leftarrow A_{row}[i+1]$

(4) $acc \leftarrow 0$

(5) **for** c $\leftarrow r_{begin}$ to r_{end}

(6) $acc \leftarrow acc + A_{val}[c] \cdot X[A_{col}[c]]$

(7) **Endfor**

(8) $Y[i] \leftarrow acc$

(9) **Endfor**

The matrix A and vector R is firstly transferred to DDR2 memory on FPGA and then assigned the vector R and a row of matrix A to a mapper. The vector-vector multiplication (line 2-7 in Algorithm 5) is conducted within a **mapper** and then result is returned and collected(line 8 in Algorithm 5) by the **reducer**. The remaining parts of the computation can be either executed on FPGA or on CPU since they take much less time than the SpMV.

In this application, the non-zeros between different rows vary drastically, and the execution time for each mapper may differ from each other. However, the dynamic scheduling policy will ensure the load balancing and keep mappers as busy as possible.

6. CONCLUSION AND FUTURE WORKS

This paper introduces FPMR, a MapReduce framework on FPGA, which provides programming abstraction, hardware architecture

and basic building blocks to developers. High parallelism can be easily achieved on FPMR, while the programming efforts are alleviated. Using this framework, designers only need to map the applications onto the mapper modules and the reducer modules. Task scheduling, communication, and data synchronization are done by the framework automatically.

In the case study of RankBoost, 31.8x speedup is achieved with 146 mappers and 1 reducer, comparable with a fully manually designed version where the speedup is 33.5x. The tradeoffs among resources, performance, and memory bandwidth are also discussed. As the technology advances, the resource of FPGA will increase and more and more processors can be placed on chip for higher performance. Finally, the bandwidth of memory will be the limiting factor during the application acceleration based on FPMR.

In our future work, we would like to investigate into the combination of automated HLS tools such as AutoPilot[3], which has already shown encouraging results in both performance and productivity of hand-coded applications[23]. The mapper and reducer modules can be directly written in high-level languages and automatically translated to hardware languages and then integrated into the framework. Also, we would like to support dynamic memory management with hardware paging for complex applications. Then we plan to test the efficiency and productivity of the framework on applications of different levels, ranging from machine learning to basic parallel primitives. We are also working on the open source release of the framework.

7. ACKNOWLEGEMENT

This work was supported by National Key Technological Program of China No. 2008ZX01035-001, NSFC No. 60870001, MSRA UR project, AMD China university program and Tsinghua National Laboratory for Information Science and Technology Cross-discipline Foundation. The authors would like to thank the anonymous reviewers for their useful and detailed suggestions and comments on this paper.

8. REFERENCES

[1] Martin C. Herbordt, Tom VanCourt, Yongfeng Gu, Bharat Sukhwani, Al Conti, Josh Model, Doug DiSabello, Achieving High Performance with FPGA-Based Computing , Computer, Volume 40 , Issue 3 (March 2007), Pages 50-57

[2] FPGAs and Moore's Law , http://www.ciol.com/Semicon/Design-Trends/News-Reports/FPGAs-and-Moores-Law/111108112450/0/

[3] AutoPilot, AutoESL, www.autoesl.com

[4] CatapultC, Mentor Graphics, www.mentor.com

[5] Impulse C, IA Technologies, www.impulseaccelerated.com

[6] A.DeHon et al., "Design Patterns for Reconfigurable Computing," Proc. 12th Ann. IEEE Symp. Field-Programmable Custom Computing Machines, IEEE CS Press, 2004, pp. 13-23.

[7] J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. In Sixth Symposium on Operating System Design and Implementation (OSDI), San Francisco, CA, 2004.

[8] Apache Hadoop, http://hadoop.apache.org/

[9] C. Chu, S. Kim, Y. Lin, Y. Yu, G. Bradski, A. Y. Ng, and K. Olukotun. Map-reduce for machine learning on multicore. In NIPS'07: Proceedings of Twenty-First Annual Conference on Neural Information Processing Systems. Neural Information Processing Systems Foundation, 2007.

[10] Adam Pisoni. Skynet, Apr. 2008. http://skynet.rubyforge.org.

[11] Colby Ranger, Ramanan Raghuraman, Arun Penmetsa, Gary Bradski, and Christos Kozyrakis. Evaluating MapReduce for Multi-core and Multiprocessor Systems. Proceedings of the 13th Intl. Symposium on High-Performance Computer Architecture (HPCA), Phoenix, AZ, February 2007.

[12] Richard M. Yoo, Anthony Romano, and Christos Kozyrakis. Phoenix Rebirth: Scalable MapReduce on a Large-Scale Shared-Memory System. Proceedings of the 2009 IEEE International Symposium on Workload Characterization (IISWC), Austin, TX, October 2009.

[13] Bingsheng He, Wenbin Fang, Qiong Luo, Naga K. Govindaraju, and Tuyong Wang. Mars: A MapReduce Framework on Graphics Processors. PACT 2008.

[14] M. Mustafa Rafique, Benjamin Rose, Ali R. Butt, D.S.Nikolopoulos. CellMR: A Framework for Supporting MapReduce on Asymmetric Cell-Based Clusters. IPDPS'09

[15] J. H. Yeung, C. Tsang, K. Tsoi, B. S. Kwan, C. C. Cheung, A. P. Chan, and P. H. Leong. Map-reduce as a programming model for custom computing machines. IEEE Symposium on Field-Programmable Custom Computing Machines, 2008.

[16] Yoav Freund, Raj Iyer, Robert E. Schapire and Yoram Singer, An efficient boosting algorithm for combining preferences, The Journal of Machine Learning Research, Volume 4 , (December 2003), Pages: 933 – 969

[17] NY Xu, XF Cai, R Gao, L Zhang, FH Hsu, FPGA Acceleration of RankBoost in Web Search Engines, ACM Transactions on Reconfigurable Technology and Systems (TRETS), Volume 1 , Issue 4 (January 2009), Article No. 19

[18] C. Chu, S. Kim, Y. Lin, Y. Yu, G. Bradski, A. Y. Ng, and K. Olukotun. Map-reduce for machine learning on multicore. In NIPS '07: Proceedings of Twenty-First Annual Conference on Neural Information Processing Systems. Neural Information Processing Systems Foundation, 2007.

[19] Boser, B., Guyon, I., Vapnik, V. A training algorithm for optimal margin classifiers. Proc. of the Fifth Annual Workshop on Computational Learning Theory, Pittsburgh, ACM, pp, 144-152

[20] J. Platt. Sequential minimal optimization: A fast algorithm for training support vector machines. Technical Report MSR-TR-98-14, Microsoft Research

[21] Bryan Catanzaro, Narayanan Sundaram and Kurt Keutzer. Fast support vector machine training and classification on graphics processors, Proceedings of the 25th International Conference on Machine Learning, Helsinki, Finland, 2008.

[22] L. Page, S. Brin, R. Motwani, and T. Winograd. The pagerank citation ranking: Bringing order to the web. Technical report, Stanford Digital Library Technologies Project, 1998.

[23] Ningyi Xu, Even Microsoft uses AutoESL's C synthesis to speed up its SW, http://deepchip.com/items/0482-06.html