RankBoost Acceleration on both NVIDIA CUDA and ATI Stream platforms

Bo WANG*, Tianji WU*, Feng YAN‡, Ruirui LI†, Ningyi XU† and Yu WANG*
*Department of Electronic Engineering
Tsinghua University
Beijing, China
Email: {wangb06, wutj06}@mails.tsinghua.edu.cn, yu-wang@mail.tsinghua.edu.cn
†Hardware Computing Group
Microsoft Research Asia
Beijing, China
Email: {v-ruirli, xu.ningyi}@microsoft.com
‡Department of Computer Science
Purdue University
West Lafayette, Indiana
Email: yan12@purdue.edu

***Abstract*—NVIDIA CUDA and ATI Stream are the two major general-purpose GPU (GPGPU) computing technologies. We implemented RankBoost, a web relevance ranking algorithm, on both NVIDIA CUDA and ATI Stream platforms to accelerate the algorithm and illustrate the differences between these two technologies. It shows that the performances of GPU programs are highly dependent on the utilization of GPU's hardware memory architectural features. In this work, we accelerated RankBoost algorithm on both platforms, and we achieved 22.9X speedup on CUDA and 9.2X speedup on ATI Stream respectively. Then we made a comparison on the differences of memory architecture between NVIDIA CUDA and ATI Stream.**

***Keywords*-GPGPU; CUDA; ATI Stream; RankBoost acceleration**

I. Introduction

Nowadays web search based services are becoming increasingly important. One of the most vital features for a search engine is the relevance of ranking, which gives an order of returning pages in accordance with their similarities to user's input query. A lot of factors affect the ranking function, such as URLs, authors, page freshness, and number of occurrence. In an ever-growing web-scale system, it is impossible to manually determine the weight of each factor and combine them into a single function. Thus, machine learning algorithms are adopted to produce ranking functions from these large-scale training sets.

The machine learning algorithms learn ranking functions from pairs of instances that have different relevance levels. Many works have been done in this field. Earlier machine learning algorithms for web ranking include Polynomial-based regression [4], Genetic Programming [2], RankSVM [5] and classification-based SVM [7]. However, these algorithms are relatively time-consuming and only tested on small datasets. Compared with the above ranking algorithms, RankBoost has a faster ranking speed, and achieves good relevance performance on large-scale datasets. Similar to AdaBoost, RankBoost applies a boosting algorithm to the ranking problem by combining the output of weak hypotheses to learn a powerful ranking function. The efficiency and satisfactory performance make this algorithm very promising.

However, the training process of RankBoost still takes a long time. For example, the original RankBoost takes several days to finish a training and obtain a model on the MSN search engine [9], which is too slow for the regular ranking model refreshing. Several methods have been proposed to reduce the computation complexity of RankBoost in [3], but none of them explore the parallel implementation of the algorithm. Recently, several works explored the parallelism of RankBoost. The FPGA based accelerator [9] implemented a modified RankBoost algorithm (M3.int) to explore the feature-level parallelism. Later, this parallelism is further explored on a combination of hardware accelerators (FPGA) and computer cluster with MPI [6].

The general purpose graphics processing units (GPGPU) have emerged as a powerful platform for massively parallel computation. There are two main computing platforms for GPGPU, namely Computing Unified Device Architecture (CUDA) proposed by NVIDIA and ATI Stream proposed by AMD. In accordance with the specific architectures, we implemented RankBoost algorithm on both platforms and achieve 22.9X and 9.2X speedup on CUDA and ATI Stream platforms respectively. Additionally we briefly compared the two platforms and gave some recommendations for GPU memory utilization.

The remainder of the paper is organized as follows. In Section I, we introduce the RankBoost algorithm and review related works. In Section III, both GeForce 9800 GTX+ architecture and ATI Radeon HD 4870 architecture are briefly introduced to provide with a basic understanding for the implementation. In Section IV and Section V, implementations of RankBoost on CUDA and ATI Stream are described respectively. Then we conduct a comparison of these two implementations in Section VI. The experimental results are presented and discussed in Section VII. Section VIII concludes the paper.

II. Preliminaries and Related Works

A. The RankBoost Algorithm

RankBoost is a boosting algorithm for web search engines, targeting to give an order for a set of documents based on the relevance. Each document d is expressed by a feature vector $\{f_i(d)|i = 1, 2, ..., N_f\}$. RankBoost tries to combine the weak hypotheses, which are functions of the features, into a final strong hypothesis, with the help of pre-assigned scores of the documents. These scores are usually given by human experiences according to their relevance to the input query. If document d_0 scores higher than d_1, then they form a pair. In the RankBoost algorithm, we maintain a distribution over all pairs, and each pair has a positive distribution value $D(d_0, d_1)$. A higher distribution means the more importance of ranking a certain scored pair correctly. The distributions are updated in each round, and those pairs that are not ranked correctly by the weak hypotheses will gain more importance with an increasing distribution.

The core procedure of RankBoost is called WeakLearn, which, in each round, gives a weak ranking hypothesis h based on the features of documents and the current distributions. In the M3.int WeakLearn Algorithm [9], h is binary, i.e. for any document d

$$h_{i,\theta}(d) = \begin{cases} 0, \text{if } f_i(d) < \theta \text{ or } f_i(d) \text{ is undefined} \\ 1, \text{if } f_i(d) \geq \theta \end{cases}$$

In WeakLearn procedure, the feature f_i and threshold θ are selected for h to maximize ranking correctness r, which is defined as follow.

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

The π value is defined to reduce the computation complexity.

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

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

To maximize r, WeakLearn has to explore all possible thresholds at a certain interval and all possible features.

In M3.Int WeakLearn [9], for each feature f_i, the values $f_i(d)$ are classified into N_θ bins so that for each feature, r is obtained by an integral histogram. That is, first calculate the histogram of each feature:

$$\text{Hist}_i(b) = \sum_{f_i(d) \text{ in bin } b} \pi(d)$$

Then r is obtained by calculating the prefix sum of the histogram of each feature as follow:

$$r_{i,b} = \sum_{\beta=b}^{N_\theta} \text{Hist}_i(\beta)$$

The most time consuming procedure of RankBoost is WeakLearn, which typically took more than 90% of all computation time on CPU implementations. However, WeakLearn can be parallelized since the computation of both histogram and prefix sum are parallelizable.

B. Parallelization Platforms

FAR (FPGA-based Accelerator for RankBoost) [9] is a hardware acceleration engine for RankBoost, which significantly improves the computation speed compared with CPU M3.Int implementation. FAR utilizes an SIMD architecture to implement RankBoost, with multiple processing engines (PEs) concurrently building multiple integral histograms. It utilizes the feature level parallelism.

FAR is based on an FPGA board with PCI (or PCI-e) interface to the host system. Bin data, i.e. classified feature data are first stored on to the FPGA board via PCI. In each round, the updated π is transferred on to the board. And after calculation, the maximum r and the corresponding feature and bin are returned to the host, which then generate the weak ranking hypothesis h.

Compared with the FPGA implementation FAR, GPU has a similar SIMD architecture, which makes us believe that RankBoost can also be accelerated on GPU platform. Moreover, GPU usually has higher computation power to price ratio than FPGA. On the other hand, programming on GPU platform is generally easier since GPU programs are instruction based software. Thus the GPU development cycle is generally shorter than FPGA's.

III. GPU Architecture

In this section, we briefly introduce the architectures of GeForce 9800 GTX+ and ATI Radeon HD 4870, which are used in our experiments. The introduction here is to help readers better understand the strategies and specific details we adopt to implement the algorithm.

A. GeForce 9800 GTX+ Architecture

GeForce 9800 GTX+ is compatible with the NVIDIA Compute Capability Specification 1.1 [8]. The GPU chip on GeForce 9800 GTX+ has 16 multiprocessors, and each multiprocessor consists of eight scalar processors. Scalar processors of the same multiprocessor execute the same instruction at any clock cycle, but may operate on different data. Processors on the same multiprocessor can communicate with each other by using a 16KB shared memory. [8]

1) Computation: From the software viewpoint, threads are the smallest unit of parallel computation and they are assigned to multiprocessors in unit of blocks and each block can hold a maximum of 512 threads in GeForce 9800 GTX+. A thread block can only be assigned to one multiprocessor. However, a multiprocessor can have several thread blocks. In the case of Compute Capability 1.1, threads of a block are scheduled in a warp size of 32. One thing should be noted is that threads of the same warp execute exactly the same instruction at one instruction issue. This feature is crucial for the implementation of **float atomic add operation**, which is described in Section IV.

2) Memory Hierarchy: There are six kinds of memory on a CUDA compatible device and three most frequently used kinds of memory are discussed below.

Register - A set of local 32-bit registers are available for a multiprocessor. For a Compute Capability 1.1 device, the total number of registers per multiprocessor is 8192. Generally, accessing a register will cost only one clock cycle so that it should be firstly used to achieve the best performance.

Shared memory - A shared memory is an on-chip memory which is shared by all scalar processors in a multiprocessor. Each multiprocessor has a 16KB shared memory in Compute Capability 1.1. The shared memory is divided into 16 banks. Different banks can be accessed simultaneously. However, when two or more threads are trying to access the same bank, a bank conflict is generated and the accesses are serialized. In the implementation of RankBoost, shared memory is used to store histograms due to its capability of scalar processor communication and fast access speed.

Global memory - Global memory is implemented as an off-chip DRAM and the read/write latency is very high. A single floating point value read operation may take 400 to 600 clock cycles. Additionally, the read and write operations are not cached. To efficiently access the global memory, the operations should be performed in a coalesced pattern, so that the maximum memory bandwidth can be achieved.

B. ATI Radeon HD 4870 Architecture

AMD released a general purpose programming platform for ATI GPUs, named ATI Stream [1]. In this section, we briefly discuss the hardware functionality and character of ATI RV770 GPGPU.

1) Computation: The stream cores or ALUs are organized as 5-way VLIW processors, called thread processors. Each thread processor contains four normal cores that can perform 32-bit integer or floating point arithmetic, and one transcendental core that can perform transcendental functions such as trigonometric or exponential functions. These five cores can perform different scalar operations concurrently if there are no data dependencies between them.

In RV770, 16 thread processors are grouped into a SIMD engine and there are 10 SIMD engines altogether. All thread processors in a SIMD engine performs the same instruction at any time, but on different datasets; different SIMD engines can perform different instructions.

There are two types of kernels, pixel shader (PS) and compute shader (CS).

2) Memory Hierarchy: In RV770, several memory resources can be used, each with different accessing constraints and speed. Utilizing the memory resources effectively is usually the key to GPGPU programming, and this is also true for ATI Stream platform.

general purpose registers (GPRs) are the fastest memories. Each thread has access to up to 127 GPRs in float4 type, which is a short vector with 4 single precision floating point elements, named x, y, z and w. At least 4 clause temporary registers are included in GPRs. They can not preserve data beyond a low-level ALU instruction clause.

local data sharing memory (LDS) - Each SIMD engine has a 16KB LDS which enables low latency data sharing between threads in the same SIMD. Every thread in a thread group owns an equal sized part of the LDS memory that it can write to; and every thread can read the whole LDS memory. This access model prevents bank conflicts.

off-chip graphic memory is the largest and slowest memory resource. These memory resources are also called streams. It supports several access models. Scratch buffer - a read/write buffer similar to private GPRs, but supports a much larger size since it is located in the off-chip memory. *Texture sampling* - reads from read-only input streams into GPRs, supports random accesses. *Pixel output* - writes to write-only output streams to a hardware generated location corresponding to the coordinates of the current thread. Each thread can write to at most 8 streams, known as 8-way Multiple Render Targets (MRT), which is only applicable in pixel shader. *Global buffer* - a read/write buffer that all threads have access to. This is the only output method for compute shader.

Stream reading and writing is cached by hardware controlled on-chip L1 and L2 caches. However, the global buffer is uncached.

IV. Acceleration of RankBoost with CUDA

As we have already discussed, the WeakLearn procedure is the most time-consuming part of the RankBoost algorithm, since it consists of histogram computation on a per-feature basis, and the number of features is usually very large. However, the histogram obtained from one feature does not depend on those of other features. Thus, the histograms on different features can be computed in parallel.

It should be noted that the histogram computation in WeakLearn is different from the traditional histogram computation, e.g. in computer vision. The tradition histogram is to show the occurrence frequency of data elements. Algorithm 1 is used to build a traditional histogram.

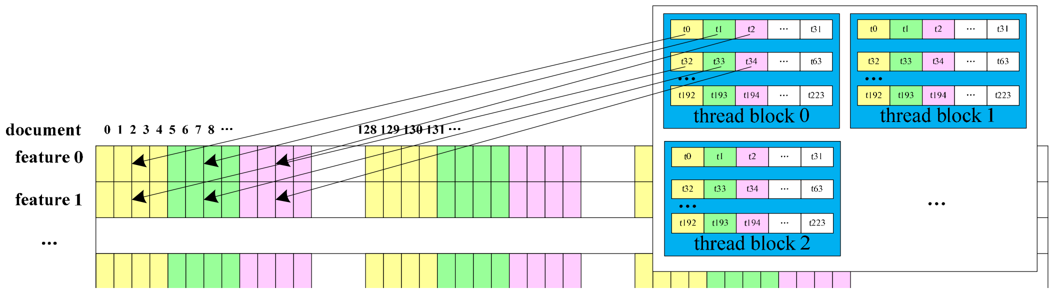

Figure 1. CUDA threads mapping scheme

Algorithm 1: Traditional Histogram

```
for $i$ = 0 to $DATA\_LENTH - 1$ do
    hist[data[i]] = hist[data[i]] + 1
end for
```

In WeakLearn, the algorithm computes a variant histogram, which increases the corresponding bin of a histogram with a float number, not always an integer 1 as in the traditional one. Additionally, an integral operation is conducted on the histogram to obtain the final result. We name it **float integral histogram** in this paper. The detailed operations are listed as Algorithm 2.

Algorithm 2: Float Integral Histogram

```
for $i$ = 0 to $DATA\_LENTH - 1$ do
    hist[data[i]] = hist[data[i]] + pi[i]
end for
for $i$ = $HISTO\_SIZE - 1$ to 1 stride -1 do
    hist[i-1] = hist[i-1] + hist[i]
end for
```

When mapping this algorithm to the GPU, the architecture constraints should be taken into account. The index array and the π array are very large and can only be stored in the global memory. In the histogram algorithm, access to these two arrays is sequential, or say coalesced. However, due to the indetermination of the data in the index array, the accesses to the array $hist$ are random. Thus, it is optimal to store the array $hist$ in the shared memory, rather than the global memory. The size of shared memory leads to several further limitations.

To fully utilize GPU computation resources, we proposed a mapping scheme that 32 threads, i.e. a warp of threads, share a histogram in the shared memory. Fig. 1 is an illustration of the mapping scheme. Threads read index elements and corresponding π value from global memory and perform the addition of π to a correspondent histogram bin. Threads in a warp operate on every four data elements consecutively, so the coalesced access requirement is met. To maximize the computation resource utilization, we assign 7 warps, i.e. 224 threads, in a thread block. One warp deals with a feature and maintains its own histogram. This constraint is resulted from the 2KB memory requirement for a 256-bin histogram. It is worth noting that we cannot assign 8 warps because 16kb shared memory are not all available for users, several bytes memory are reserved for kernel launch and parameters passing.

Each index data element, which is an integer data, is quantified to 8 bits. Thus, it takes only one 32bit read operation to read four index data elements. To cooperate with this compression, `float4` type is used to store π values so that the read of four π values is also done in one operation.

A critical problem we have to deal with is that two or more threads may perform the addition at the same histogram bin, which may lead to an error that only one addition is successfully performed, because Compute Capability 1.1 device does not support atomic write in the shared memory.

To cope with this problem, we take advantage of the SIMD mechanism that threads in the same warp always execute same instructions at a clock cycle. Additionally, it is guaranteed that when two or more threads collide on writing to the same location, only one of them will succeed. With these two properties, we define our **float atomic addition** as Algorithm 3.

Algorithm 3: Float Atomic Addition

```
repeat
    s_hist[index].$tag$ = $thread\_tag$
    s_hist[index].$value$ = s_hist[index].$value$ + $\pi$
until s_hist[index].$tag$ == $thread\_tag$
```

Here, the `tag` is used as a flag to distinguish whether a thread has successfully write in the π value. One thing should be addressed is that the hardware scheduling mechanism guaranteed that the two operations on `tag` and on `value` are done by the same thread in the same collision condition. From the model above, we know that when n threads are trying to write to the same location, exactly n iterations will this loop be performed. Thus we can see that the conflicts will decrease the performance and the algorithm efficiency is data-dependent.

V. Acceleration of RankBoost with ATI Stream

The fundamentals of implementing RankBoost on ATI Stream platform are similar to that of the CUDA platform, but there are some special constraints to be concerned.

Firstly, each stream on ATI architecture has a size constraint that each dimension of the stream cannot exceed 8192 elements [1] . In other words, the maximum size for a 1-D array is 8192, while the maximum size for a 2-D matrix is 8192×8192. Since each element is 128 bits, the maximum size of a stream is $8192 \times 8192 \times 128b = 1GB$. In ATI Brook+ language, a certain technique called address virtualization can be used to virtually manipulate 1-D array larger than 8192 elements. This is achieved basically by cutting the 1-D array into smaller arrays to form a 2-D matrix, with both height and width no more than 8192.

In our case, the size constraint means that the data of a single feature cannot fit into a single row of a GPU stream. For a single row, the maximum size is $8192 elements \times 16B/elements = 128kB$. Since each document's value has been mapped to a bin index ranging from 0 to 255, the maximum document number for a single row is also 128k. Nevertheless the documents number N_{doc} can easily exceed this number. Thus, the documents' value (bin index) for a single feature needs multiple rows to be stored, similar to address virtualization, but in a 2-D manner, since we have multiple features. Let the width of each row be w_{doc}, which is also the width of the whole data stream, i.e. $w_{doc} = w_{stream}$. The height of a feature is

$$h_{feature} = \lceil N_{doc}/w_{doc} \rceil$$

If there are M features in the stream, the total height of the stream is

$$h_{stream} = M\lceil N_{doc}/w_{doc} \rceil$$

Secondly, we need to use scatter output instead of stream output to export the result of the histogram, since each pixel shader can write to at most 8 destinations (totally 32 floating-point numbers), which is not enough for our histogram output. Scatter output uses the uncached global buffer, which is a linear space capable for both reading and writing. Generally, accessing global buffer is slower than stream output, so the latter is preferred if possible. However, in our case, scatter output is not the bottleneck of the histogram algorithm, as discussed later.

Considering all these characteristics, we implement the Weak Learn procedure with 4 GPU kernels that run one by one sequentially. Fig. 2 shows the block diagram of the procedure. The GPU kernels are discussed below.

Hist - partial histogram kernel. Each thread of this kernel processes a single row of the input data stream to generate a partial histogram. The output is a 2-D matrix with width of 64 (in terms of float4 elements, i.e. 256 floating points). Each row of it represents the partial histogram of the corresponding row in the input data stream. Therefore the height of the output matrix is identical to the input, i.e. h_{stream}. Hist is the most time-consuming kernel among all the 4 GPU kernels.

On the ATI Stream platform, each thread can up to 127 general purpose registers (GPRs), including at least 4 clause temp registers. All GPRs are 128-bit wide, and support relative addressing using a special register AR. So the 256-histogram of each thread can be stored in 64 GPRs, which is the fastest onboard memory resource. After obtaining the partial histogram in GPRs, the kernel then exports the values to the global buffer.

Reduce - merge the partial histogram to the final histogram. This kernel takes the output of Hist as input, adds up the corresponding $h_{feature}$ elements to generate the result. Compared with the input stream, the output stream's size is reduced to $64 \times M(W \times H)$.

Expand - expand each float4 elements of the histogram to 4 separate (float4) elements, each with x element as the histogram data, and y, z elements as the feature and bin coordinate. w elements are not used. The result is output using stream write, with size expanded to $256 \times M(W \times H)$.

Scan - calculate the prefix sum of each row of the expanded histogram. The output stream has the same size as the input stream, i.e. $256 \times M(W \times H)$. After Scan, the results are transferred back to the CPU to find the maximal value and weight updating. The maximum $|r(f_k, \theta_s^k)|$ is obtained after comparison, along with the index k, s and α.

Each round of the above procedures processes M features. To finish all dataset, we need $\lceil N_{feature}/M \rceil$ flows. M should be chosen as large as possible, so that 1) the data size for each flow is large enough to be mapped to enough threads for GPU and 2) the overhead for changing kernels can be minimized. However, a valid M value should ensure that 1) h_{stream} do not exceed the limit of 8192 and 2) the Graphics Card has enough onboard memory space.

VI. Implementation Comparisons

In this section, the differences between the CUDA implementation and the ATI Stream implementation are discussed. From these differences we put forward some comments for these two GPGPU platforms.

A. Mapping Scheme Differences

As we have discussed in Section V, the ATI Stream architecture does not support 2-D matrix with either width or height larger than 8192. CUDA, on the other hand, does not have such constraints. That is why it is necessary to fold the data of each feature into multiple lines on the ATI architecture. After folding, the fact that each feature is represented as a 2-D matrix naturally lead to the threads

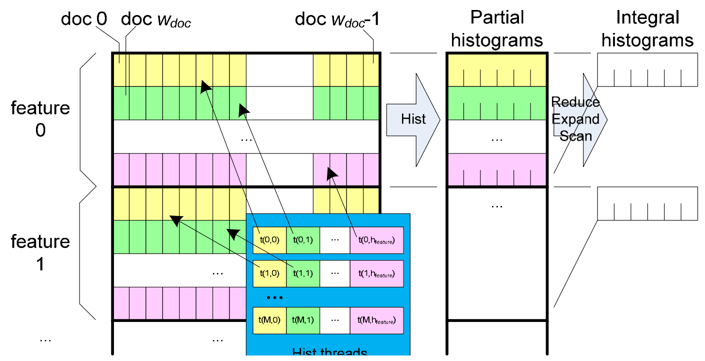

Figure 2. ATI Stream WeakLearn scheme

mapping scheme based on rows (or columns) of features, i.e. each thread process one row of a (folded) feature matrix. This mapping scheme not only eases the complicity of addressing corresponding data of a certain thread given the thread ID (or coordinates), but also, on the ATI architecture, shows better performance than our mapping scheme for CUDA.

B. Shared Memory Size Constraint

On a CUDA Compute Capability 1.1 device, there are only 16kB shared memories for a thread block, which denies the naive idea that a block contains more than 15 threads and each thread has its own sub-histogram. To maintain high parallelism and maximize the resource utilization, the float atomic addition operation is adopted. However, this further leads to the write conflicts.

On the ATI Stream side, things are much better. Registers can be used to store private histogram for each thread. No write conflicts or data dependency exist here. However, ATI Compute Abstract Layer (CAL) with Intermediate Language (IL) has to be used to utilize the registers as arrays.

For future development of CUDA devices, the increase of shared memory size will relieve this constraint and allow higher parallelism. Nonetheless, support for the float atomic operations on shared memory is also of great help for higher efficiency.

C. Shared Memory Access Model

In our CUDA implementation, the per-multiprocessor shared memory is used to store and accumulate the histograms. However, this is inapplicable for the ATI Stream architecture, since currently, ATI Stream does not support the shared memory access model like CUDA.

Although there are 16kB of per-SIMD local shared memory (LDS) available on RV770 GPGPU, it only supports a write-private read-anywhere model. In this model, each thread owns an equal piece of the LDS. Each thread can only write to its own piece of the memory while being able to read from any spaces owned by other threads in a same thread group. The maximum size of memory that one thread can own is 256B, which is not enough to store the result of 256-histogram (which needs 256B$\times$4=1KB of memory). On the other hand, the LDS currently supports only hardcoded writing address, so that it cannot be used for accumulating histograms.

Due to the differences in shared memory access models between CUDA and ATI Stream, the histogram accumulators differ, as well as the merge of several partial histograms of a feature into one.

D. Comments on two platforms

Based on the above comparisons, the random-read random-write access model enables GeForce 9800 GTX+ to build histograms on the shared memory directly, while the write-private read-anywhere access model prevents Radeon HD 4870 from doing so. However, each thread of Radeon HD 4870 can access up to 123 float4 type private registers and up to 128 threads can work in one thread block. This number is much bigger than the 8192 registers for a thread block in GeForce 9800 GTX+. Additionally, registers in Radeon HD 4870 support indexed addressing, which is capable for applications such as building histograms.

The data type used in Radeon HD 4870 is organized in four-component vectors, e.g. float4 type. This organization is more suitable for those operations dealing with vectors. While dealing with scalar type, although the compiler itself can optimize the utilization of vector operations, it may be more difficult for the programmer to fully exert the computation power compared with GeForce 9800 GTX+.

VII. Experimental Results and Analysis

This section presents the results of performance experiments on a real dataset from a commercial search engine. Then the differences of the two platforms are discussed. Finally, two performance models for both implementations are proposed.

A. Experimental Results

To compare the acceleration of GPU platforms, we implement an optimized CPU-based RankBoost i.e. M3.int [9]. Table I shows the parameters of the dataset.

Table I
PARAMETERS OF BENCHMARK DATASET

	dataset
number of features	2,576
number of documents	173,354
number of pairs	2,927,004
data size	428MB

The algorithm has been run for 100 rounds, and the result in Table II shows the average execution time for both CPU and GPU implementations. The CPU results are done on a computer with an Intel Pentium 4 3.2GHz processor, 1.5GB DDR400 memory.

On the NVIDIA side, the GPU testing system has a NVIDIA GeForce 9800 GTX+. On the ATI side, we use an ATI Radeon HD 4870 (RV770 chip) graphic card. The results we get are listed as follow.

Table II
COMPUTATION TIME FOR DATASET

Architecture	Time per round ms		Speedup	
	WL	Total	WL	Total
CPU	11844.4	12281.5	1	1
CUDA	99.1	536.5	120	22.9
ATI Stream	801.0	1328.0	14.8	9.2

In Table II, WL stands for WeakLearn procedure in RankBoost. In the CPU version, WeakLearn procedure takes about 96% computation time. With the acceleration of CUDA and ATI Stream, this procedure is computed 120X and 14.8X faster respectively and the major bottleneck turns to be the other procedures of the algorithm, such as π calculation and weight updating. Owing to the execution time of the sequential procedures, the total speedup we achieved is 22.9X and 9.2X respectively.

B. CUDA Performance Model

Compared with the thousands of iterations of training process, the program initialization time is relatively small and can be omitted. Here we only model the training process. The performance model of CPU version has been defined and analyzed in [9]. Let t_{CPU} be the mean time of one round execution. The performance model is

$$t_{CPU} = \alpha N_{pair} + \beta_{CPU} N_f N_{doc}$$

The first item αN_{pair} represents the time for π calculation and weight updating. The second item $\beta_{CPU} N_f N_{doc}$ represents the time for finding the best hypothesis.

The π calculation and weight updating in GPU are the same as the CPU one. But the searching for the best weak hypothesis step is different. The π data is downloaded to the GPU before the computation. The max r on each feature and the corresponding bin are transferred back after the computation. Data transfer between GPU and CPU is through the PCI-E bus. Generally, the download speed is faster than the upload speed in GPU. Thus we define

$$t_{download} = a N_{doc}$$

$$t_{upload} = b N_f$$

where a is the coefficient of GPU download speed and b is the coefficient of GPU upload speed.

As mentioned before, the CUDA-accelerated RankBoost is data-dependent owing to the shared memory write conflict. Therefore the computing time should take the characteristics of dataset inside.

$$\begin{aligned} t_{CUDA} = {} & \alpha N_{pair} + \beta_{CUDA} N_f N_{doc} \\ & + \gamma P_{conflict} N_f N_{doc} + a N_{doc} + b N_f \end{aligned}$$

The first item is for π calculation and weight updating, same as the CPU version. The second item is for fetching data from the global memory, where β_{CUDA} is reciprocal to global memory bandwidth. The time of addition is directional proportional to max write conflicts in a warp of threads. Thus, we define the percentage of the conflict threshold values as

$$P_{conflict} = \frac{\sum_i (\text{max conflicts in warp } i)}{N_f N_{doc}}$$

Fig. 3 shows the estimated speed up based on our system.

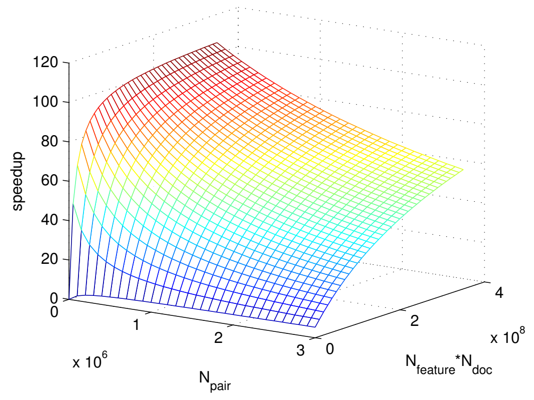

Figure 3. Estimated Speed-up of CUDA

C. ATI Stream Performance Model

Now we model the execution time of each round in the training process of the ATI GPGPU based implementation. The execution time consists of three parts, GPU execution time, CPU execution time and data transfer time.

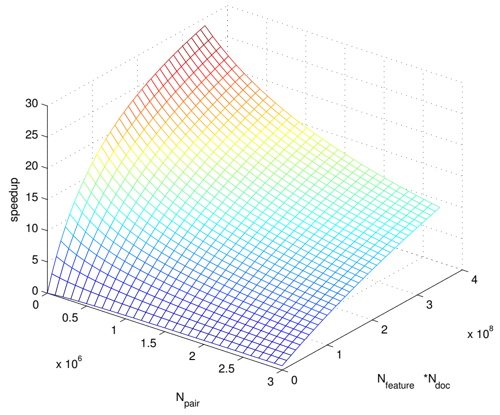

Figure 4. Estimated Speed-up of ATI Stream

The GPU execution time consists of the time spent for executing the four GPU kernels. It has the following form.

$$t_{GPU} = \beta_1 N_f N_{doc} + b_1 N_f$$

The first term represents the total execution time of kernel Hist and Reduce. The second term represents the total execution time of kernel Expand and Scan.

The CPU time consists of the time for obtaining the maximum r, weight update and π calculation. It has the form

$$t_{CPU} = \alpha N_{pair} + b_2 N_f$$

where the first term corresponds to the time of weight update and π calculation, and the second term represents the time for obtaining the maximum r.

In each round π values are downloaded onto the graphic memory, and the integral histogram results are uploaded to the system memory. The time taken for the former transfer can be represented as aN_{doc}, the later can be represented as $b_3 N_f$. To discuss the case that the dataset is too large that only part of it can be stored in the local (graphic) memory, we define γ as the portion of the data that can be fitted into the graphic memory. In each round, the time taken to transfer the data in the system memory to the graphic memory can be represented as $\beta_2(1-\gamma)N_f N_{doc}$.

Then the performance model for the total execution time is as follow.

$$\begin{aligned} t = \alpha N_{pair} &+ (\beta_1 + \beta_2(1-\gamma))N_f N_{doc} \\ &+ aN_{doc} + (b_1 + b_2 + b_3)N_f \end{aligned}$$

In real cases, the N_{doc} term is so small that can be ignored. Fig. 4 shows the estimated speed up of ATI Stream.

VIII. Conclusion

In this paper, we accelerated the RankBoost algorithm by implementing the histogram procedure on both NVIDIA CUDA and ATI Stream platforms. The program on these two platforms outperforms the CPU counterpart by 22.9X and 9.2X respectively. The implementation is guided by the idea to maximize the parallelism and utilize memory more efficiently. The major factor leading to the differences is the memory hierarchy.

For the future work, the other parts of the RankBoost algorithm can also be implemented on GPU to obtain higher speedup. Streaming method can be used to remove the constraints of the limited global memory size on the GPU. Additionally, multi-GPU techniques can be adopted to further exploit the parallelism of the algorithm.

Acknowledgement

This work is supported by Microsoft Research Asia and AMD China University Program. This work is also partially supported by National Natural Science Foundation of China (No.60870001), 863 project (No. 2009AA01Z130). The authors would like to thank Bingsheng He for his support and discussion in this work.

References

[1] Advanced Micro Devices, Inc. *ATI Stream Computing User Guide*, 2008.

[2] W. Fan and et al. Ranking function optimization for effective web search by genetic programming: An empirical study. *HICSS*, pages 8–16, January 2004.

[3] R. S. Y. S. Y. Freund and R. Iyer. An efficient boosting algorithm for combining preferences. *Journal of Machine Learning Research*, 4:933–969, 2003.

[4] N. Fuhr. Optimum polynomial retrieval functions based on the probability ranking principle. *ACM TOIS*, 7(3):183–204, July 1989.

[5] T. Joachims. Optimizing search engines using click through data. *SIGKDD*, pages 133–142, 2002.

[6] Z. Li, N. Xu, F. Hsu, X. Cai, R. Gao, and Z. Xia. Distributed rankboost acceleration using fpga and mpi for web relevance ranking. In *ICPADS*, 2008.

[7] R. Nallapati. Discriminative models for information retrieval. *SIGIR*, pages 64–71, 2004.

[8] NVIDIA CORPORATION. *CUDA Compute Unified Device Architecture Programming Guide*, 2008.

[9] N. Xu, X. Cai, R. Gao, L. Zhang, and F. H. Hsu. Fpga-based accelerator design for rankboost in web search engines. In *International Conference on Field-Programmable Technology*, pages 33–40, 2007.