Parallel PageRank Computation using GPUs

Nhat Tan Duong
High Performance Computing Center
Hanoi University of Science and Technology
Hanoi, Vietnam
dn.nhattan@gmail.com

Anh Tu Nguyen
High Performance Computing Center
Hanoi University of Science and Technology
Hanoi, Vietnam
ng.tuna@gmail.com

Quang Anh Pham Nguyen
Naiscorp Information Technology Service Company
Hanoi, Vietnam
anhpnq@socbay.com

Huu-Duc Nguyen
High Performance Computing Center
Hanoi University of Science and Technology
Hanoi, Vietnam
ducnh-fit@mail.hut.edu.vn

ABSTRACT
Fast & efficient computing of web rank scores is a necessary issue of search engines today. Because of the enormous size of data and the dynamic nature of World Wide Web, this computation is generally executed on large web graphs (to billions webpages) and requires refreshing quite often, so it becomes a challenging task. In this paper, we propose an efficient method for computing PageRank score – a Google ranking method based on analyzing the link structure of the Web on graphics processing units (GPUs). We have employed a slightly modification of a storage data format called binary 'link structure file' which inspirited from [2] for storing the web graph data. We then divided the PageRank calculating phases into parallel operations for exploiting the computing power of the graphics cards. Our program was written in CUDA language to experiment on a system equipped two double NVIDIA GeForce GTX 295 graphics cards, using two real datasets which were crawled from Vietnamese sites containing 7 million pages, 132 million links and 15 million pages, 200 million links, respectively. The experimental results showed that the computation speed increase from 10 to 20 times when compared to a CPU Intel Q8400 at 2.67 GHz based version, on both datasets. Our method can also scale up well for larger web graphs.

Categories and Subject Descriptors
D.1.3 [**Concurrent Programming**]: Distributed computing, parallel computing

General Terms
Performance, Algorithms

Keywords
PageRank, CUDA, GPU, parallel algorithm.

1. INTRODUCTION
Search Engine is an indispensable tool in our age because of its ability of fast retrieving information from a huge amount of data.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
SoICT 2012, August 23–24, 2012, Ha Long, Vietnam.
Copyright 2012 ACM 978-1-4503-1232-5/12/08…$10.00.

The usefulness of a search engine depends on the relevance of the results returned to a given input query. While there may be millions of pages containing particular words or phrase in the query, some pages may be more relevant, popular and therefore more expected by users than others. Most search engines rank the results so that better results are on top of the list. Although the methods of ranking may vary among search engines, the comparison is often based on two factors: the relevance between page content and the query, and the popularity of the page. The popularity of a webpage is often calculated by analyzing the Web's hyperlinks. In 1998, Sergey Brin and Larry Page found a link analysis system to determine the popularity score of web pages called PageRank [1], which is then used by Google in its famous search engine. The idea behind PageRank is that a page is important if it is pointed to by other important pages. The PageRank score of a web page is calculated from those of pages pointing to it. With the success of Google, PageRank has proved its superiority over other link analyzing methods.

PageRank calculation is a non-trivial task. There are many challenges we encounter when calculating PageRank value of web pages. The first difficulty is that the input data is extremely huge; therefore, it requires a lot of computing effort. It is estimated that the number of web pages on World Wide Web is over 40 billion. Even when the size of data we have is just a fraction of that number (few billion or hundreds of million), it is still not easy to compute efficiently. The second problem comes from a characteristic of the Web: it is dynamic. This characteristic is reflected in two aspects. First, the content of web pages may be changed along the time. This leads to the change of hyperlinks in the pages and therefore the change of the Web's structure. Moreover, the size of the World Wide Web, which is determined by the number of web page, increased rapidly with billions of web pages being created every year. To make PageRank values always up-to-date despite these changes, PageRank calculation should to be carried out in as short a time period as possible. This comes to the need of deploying PageRank to run on high performance computing infrastructures such as specialized hardware or clusters including computing nodes.

There have been a few implementations of PageRank on CPU-based infrastructures, including PC cluster [2,3,4] or P2P architectures [5]. These methods need many processors connected together via network; therefore, a common problem issued is that the communication overhead over the computation. For example, A. Rungsawang and B. Manaskasemsak [2] computed pagerank

scores on an CPU cluster of 8 nodes, each node equipped one AMD Operaton 240. Due to the large RAM, their version can executed for the web graph containing 174 million pages and over 1.5 billion links but they took 14.6 seconds for each PageRank iteration.

In recent years, the trend of using specialized hardware gives more benefits of many sides: cost, electrical energy, space area and easy-to-use as well as maintain. A growing tendency is being developed for the next generation computers is multi, many cores architecture, which helps solving a lot of problems requiring large computing power. Graphics cards (GPU) are such example. With specific designs for visualization, the GPU ability outperforms that of multi-core CPUs in both Floating-Point Operations per Second and Memory Bandwidth. In addition to the appearance of software architectures such as CUDA, CAL, developers can easily exploits the power of graphics cards in non-graphics computing tasks using the high-level programming languages. In this paper, we study on the approach of calculating PageRank on the GPU environment.

There have been many researches of computing PageRank score on GPUs recently, mostly focused on implementing the SPMV problem efficiently [7, 8, 10]. In [9], the authors used NVIDIA's SPMV libraries as CUBLAS; CUSPARSE for their PageRank computation on a Fermi-based GPU GTX480. They ranked a quite small dataset – about 280,000 web pages and gained speed up 3-4 times in term of computation time. Another research which employed SPMV in computing PageRank scores is [10], Tianji WU and his colleagues used CRS format for storing sparse matrix of web graph and executed SPMV on single GPU AMD Radeon 870. Their best results achieved 15 times faster than CPU Phenom II with the web graph size from 2 – 3 million pages. Besides SPMV approaches, state-of-the-art hypergraph-based partitioning schemes also bring researchers a feasible solution for speeding up PageRank computation. The results of [11, 12] have proved this method. In [11] Ali Cevahir, Cevdet Aykanat and Ata Turk reported scalability of PageRank on the large-scale supercomputer TSUBAME which has 96 Tesla GPUs, the best speed up on 64 GPUs achieved about 8 times faster than the CPU cluster implementation with the same number of CPU cores for 39.5 million web pages.

In this paper, we aim to accelerate Google's PageRank algorithm on multi-GPUs. We first use a slight modification of link structure file [2] which specially designed to represent the connectivity of a web graph. This organization seems natural to the Web graph because from each web page we can find a list of other pages pointed by it through hyperlinks in html page. This is also the least memory consuming presentation of a graph, which makes this method suitable in this case when the size the Web graph is huge. Based on this webgraph representation and by analyzing carefully operations of PageRank formula, we then design a parallel PageRank algorithm on multi-GPUs, in which PageRank scores computation is distributed to GPUs and further divided to GPU threads. We ran some experiments to compare our approach's performance with that of sequential algorithm on CPU, using a set of real data set which crawled within the Vietnamese sites; the results are quite promising as we obtained an about 20 times speed up on four GPUs GeForce GTX 295. Our method is general and does scale up well with large web graph as long as it can fit to the limited size of device memory.

This paper is organized in the following way. The second section of this paper introduces basic knowledge of PageRank, the binary "link structure file" for storing large web graph for convenient computation later, we also gives a brief description about CUDA and GPGPU era. In section 3, we describe details of our solution for PageRank problem on the multi-GPUs system. Section 4 shows some experimental results of PageRank computation on both CPU and GPU and gives some discussions. Finally, section 5 concludes the paper.

2. BACKGROUND

2.1 Review of PageRank

The PageRank system assigns a score to every document in the system which indicates the importance of that web page. The basic idea of PageRank is: a page is important if it is pointed to by other important pages. To understand this thesis, we assume that each hyperlink is a recommendation; thus, a page which has many recommendations, which correspond to inlinks, must be an important page, and vice versa. However, the weights of recommenders are not the same. The recommendations from important pages are considered more valuable than those of less important ones. Therefore, an important page is the page which (1) has many inlinks, (2) has inlinks from other important pages or (3) both. From that assumption, the original formula of PageRank is defined as:

$$r(P_i) = \sum_{P_j \in B_{P_i}} \frac{r(P_j)}{|P_j|} \quad (1)$$

in which, $r(P_i)$ is the PageRank of page P_i, B_{Pi} is the set of pages pointing to P_i and $|P_j|$ is the number of outlinks (recommendations) from page P_j. Since the PageRank of source pages P_j are still unknown, the above formula is converted to an iterative formula where the web page's PageRank is calculated from other PageRank of the previous iteration. Let $r_{k+1}(P_i)$ denote the PageRank of page P_i at iteration k + 1. Then,

$$r_{k+1}(P_i) = \sum_{P_j \in B_{P_i}} \frac{r_k(P_j)}{|P_j|} \quad (2)$$

The initial PageRank values of all pages are assigned to 1/n where n is the number of pages. The iterative process is repeated until PageRank scores converge to the final stable values. However, there is a problem with this definition of PageRank. In practice, with the above initial values, some pages will accumulate more and more PageRank after each iteration and refuse to share to other pages. This problem is possible because the web graph is not strongly connected and there are many pages without any outlink in the graph. This results in that there will be PageRank value sinking to zero at the end of iterative process; hence, it becomes hard to rank web pages using PageRank when these values are mostly 0. To overcome this challenge, Brin and Page introduce some adjustments: firstly, they replaced nodes with out-degree zero (called dangling nodes) by nodes linking to all other nodes; and secondly, they added a damping factor which influencing the random walks of the random surfer process of the graph. Equation (2) can be written as:

$$r_{k+1}(P_i) = \frac{1-\alpha}{N} + \alpha\left(\frac{\pi^{k(T)}a}{N} + \sum_{P_j \in B_{P_i}} \frac{r_k(P_j)}{|P_j|}\right) \quad (3)$$

Where $\alpha \in (0, 1)$ is the damping factor that descripts the probability of a user follows hyperlinks in the web graph. In the above equation, $\pi^{(k)T}$ is the transpose of $\pi^{(k)}$ which is row vector containing PageRank scores of N web pages at the k^{th} iteration, i.e. $\pi^{(k)} = \{r_k(P_1), r_k(P_2), \ldots, r_k(P_n)\}$, a is a binary vector so that $a_i = 1$ if page i is a dangling node, $a_i = 0$ otherwise.

In this paper, we will use Equation (3) to computing PageRank.

2.2 Web Graph Representation

The input data for PageRank computation is a Web graph, in which web pages are vertex and hyperlinks are edges. Search engines collect data from WWW by a program called Web crawler (or Web spider) which works like an automatic browser: it arrives at a web page, analyses hyperlinks in that page and follows those links to jump to other pages. Web crawlers create a copy of all visited page for later processing by a search engine such as indexing, displaying search results, etc. The Web graph is also constructed from these data.

Building the Web graph is challenging due to its size: it may contain millions to billions of nodes (web pages) and the number of edges (hyperlinks) is about ten times more. Thus, it is important to choose an efficient data structure for the representation of web graph. We found that adjacency list is an appropriate option because Web graph is very sparse; it is more saving to represent only pointed nodes than to store unnecessary information of other nodes. In this work, we are inspired by the storage technique called "link structure file" that are presented in [2], which is very similar to adjacency list. More specifically, we have created a web graph file containing records which include 3 fields: the first, a 4-byte integer refers to a source node, the second, also a 4-byte integer, denotes its out-degree (the number of edge starting from it); and the last, a list of integers represents the ids of destination nodes. Here, we store the number of out links for each node instead of the number of inlinks as in [2]

source-id	out-degree	dest-id
id1	2	id45; id189
id2	1	id832
id3	0	
id4	3	id236; id2501; id12042
…	…	….

Figure 1 The binary "link structure file"

Figure 1 shows an example of a binary "link structure file" that represent for the web graph. Each row in the figure corresponds to a record in the file. The first record contains information of the Web node having the id 1. This node has the out-degree of 2, which means that this Web page points to other 2 pages. The ids of these 2 pages can be fetched in the following list of integers, which are id45 and id189 in this example. The same pattern is applied to the other remaining records in the Web graph file. By using this storage structure, the web graph occupies 4*(2*V + E) bytes in memory, where V is the number of pages, E is the number of hyperlinks.

Note that the source-ids listed in the first column are disordered: Due process of data collection (web crawling) is implemented in distributed environments, so assigning id continuously for web pages is very difficult. Also for the deduplication purpose, web page's id is generated directly by hashing the URL to create a 4-byte number; ensuring two different sites will have different ids.

Our PageRank computational algorithm does not directly employ the input web graph files as illustrated in Figure 1 because we found that it is less efficient in accessing graph's elements as well as splitting data for parallel execution later, so we rather convert the above web graph in to a new web graph in which source – ids are ordered, i.e. from 1 to N, causing dest-ids are also reordered. We performed this preprocessing process on CPU, which results in a new web graph with the same structure of the old, but node-ids were new indexes that are sorted.

2.3 CUDA and GP-GPU

Modern Graphic Processing Units (GPUs) provide high computational power at low costs, for example NVIDIA's GPU GT200s generation can deliver close to 1 TFLOPs of computing power, more than 10 times over dual-core processor the Intel Xeon 3.2 GHz [15]. The main difference between GPU and CPU computing power is at design purpose: while CPU technology speeds up a single task, GPU technology tries to increase the number of tasks that can be performed in parallel, so latest GPUs can consist of hundreds cores while the number of cores in common CPUs has not reached 8 cores yet. GPUs dedicate more transistors for ALU instead of data cache and control flow so the power of a single thread on GPU is much weaker than that of a CPU thread; however, it is the massive parallel processing ability that makes GPU powerful. Currently, all processing cores on one single GPU can only execute a single piece of code at a time, so GPU is suitable for data parallel problems, in which the same program code will be executed in parallel for several different data sets.

GP-GPU which stands for General-purpose computing on graphics processing units (GPGPU) is the technique of using a GPU on solving computing tasks which are traditionally handled by CPU. GP-GPU is not new concept when used from 2000s. Before 2006, developers had to convert their problems into graphics field and resolved them using graphics APIs (OpenGL, DirectX) or vertex shading language / code (GLSL, Cg, HLSL) which are complex. To overcome this drawback, GPU manufactures are interested in providing better application development environments for common developers to create applications on GPU. NVIDIA CUDA is good example for such effort, which is a parallel computing architecture including the device driver, the application development tool (compiler, debugger, profiler, high-level programming languages on GPUs). With CUDA, programmers can exploit many well-known programming languages such as C/C++ to write their application on GPUs while they no longer need graphics-specific knowledge.

A CUDA program which written in an small extension of C language (called CUDA C language) includes one or a few special piece of code, called parallel kernels. This kernel can be executed in parallel on the large number of threads on GPUs. Since GPU threads execute the same code of the kernel, only be identified by the Ids of threads; so to increase the application performance, programmers need to apply data parallel techniques (data is divided into small parts and they are assigned to threads for implementing). Programmers can determine the number of threads when activating the parallel kernel. To ensure that the program do not depend the hardware (the number of Streaming Processor - SM on GPUs), threads are divided into small groups which are executed on the same SM, called thread blocks, these blocks are also designed to a grid. When a grid is executed, the CUDA scheduler which based on the hardware identifies the number of

thread blocks concurrently executed. If this number is less than the number of SM of GPUs, the whole grid will be executed concurrently; on the contrary, the thread blocks will be automatically divided into different parallel session. This technology is one of important factors for the opening of the recent GPGPU (General-Purpose computation on Graphics Processing Units) era. Although it has been interested in the recent few years but the number of applications as well as the scientific work involved to CUDA is staggering [14]. CUDA has been applied successfully in applications from numerous domains such as bioinformatics, digital signal processing, cryptanalysis, simulation, etc.

With the ability to perform data parallelism on such a lot of threads, GPU is an appropriate choice to the problem of PageRank Computation, where each thread can take one node in the web graph to compute its PageRank score. The next section of the paper explains details of our GPU-based PageRank computation using CUDA.

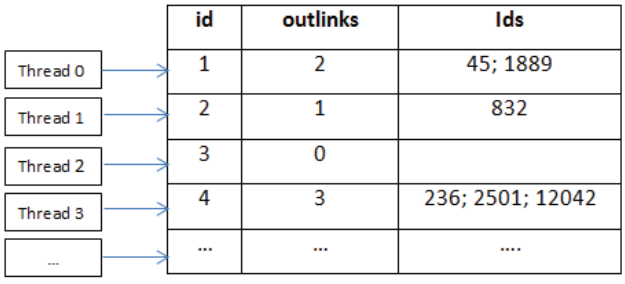
3. ACCELERATING PAGERANK COMPUTATION ON GPUs

3.1 Strategy

To parallelize the PageRank computation, we first analyze the fundamental formula (3): This formula will be applied for each node of the web graph to update PageRank value in the same manner. The computing PR process is repeated until PageRank scores converge to the final stable values, in which the initial PageRank values of all pages are 1/n where n is the number of pages. The right hand side of (3) can be divided into three operands that are separated with each other by plus signs:

$$(1-\alpha)/n \qquad \alpha\pi^{(k)T}a/n \qquad \alpha\sum_{P_j\in B_{P_i}}\frac{r_k(P_j)}{|P_j|}$$

Except the first operand, the computation of the two remaining ones required to use the PageRank values of the previous iteration. Therefore, PageRank value of a node in current iteration is completely independent of those of the remaining nodes. So we can exploit this data parallelism characteristic for paralleling on GPU, we let each CUDA thread takes one page-id and perform pagerank computation for outlinks pages pointed by that source page, as illustrated in Figure 2.



id	outlinks	lds
1	2	45; 1889
2	1	832
3	0	
4	3	236; 2501; 12042
...	...	....

Figure 2 Each CUDA thread takes one source-id and update for *oulinks* dest-ids

Each thread in the kernel analyzes one node-id of the web graph based on thread identity, for example, thread 0 takes page-id 1, thread 1 take page-id 2, …, etc., and checks the number of outlinks of this source, each outlink corresponds to another node called destination as mentioned in section 2.2. The PageRank value of these dest-ids at the iteration k + 1 is accumulated by the PageRank of the source at the previous iteration (k):

$$r_{k+1}(P_i) = r_{k+1}(P_i) + \frac{r_k(P_j)}{|P_j|}$$

In the case that one page-id has no out link, that mean this node is a dangling node, then the corresponding thread will accumulate pagerank at the previous iteration of this node into a *dangling value* which represents for the second operand $\alpha\pi^{(k)T}a/n$ in the right hand side of formula (3).

Figure 3 illustrates a step in the PageRank calculating process of all threads. This step calculates the PageRank value of the $(k+1)^{th}$ iteration from that of the previous iteration (k^{th}). Two vector $\pi^{(k)}$ and $\pi^{(k+1)}$ are used for storing the PageRank values of these 2 consecutive iterations. Each array contains n elements which is the PageRank values of web pages in the corresponding iteration. Before each iteration calculation, values in the $\pi^{(k+1)}$ array should be set to 0 in preparation for the calculation in this step. At a moment, a thread visits an element, which is also a Web node, of the π^{k}; from this source node, that thread can fetch the list of pointed nodes by looking up in the Web graph file. The PageRank values of these pointed nodes are updated to the $\pi^{(k+1)}$.

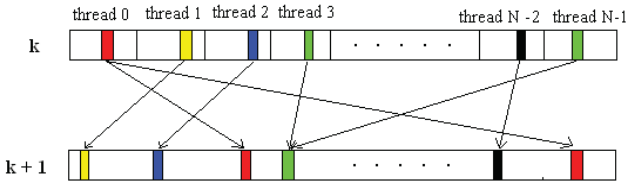


Figure 3: Calculating PageRank values in parallel at the $(k+1)^{th}$ iteration

3.2 Parallel Algorithm Execution

It is possible in theory to compute PageRank of n nodes using p threads, where p = n, executing at the same time; hence, we need only one pass on each iteration. However, because of the limitation of hardware, the number of threads p is much smaller than the number node in the graph (n). If this is the case, we further divide the input data D or the web graph to many parts, each of which is calculated by GPU kernel. In the other words, if n is number of webpages and p is the optimize number of threads can be executed in parallel on GPU, we will call sequentially (n/p) times the GPU kernel for processing (n/p) parts, in which each part contains p records of source pages or source-ids in the web graph structure. The basic idea of paralleling PageRank computation on GPU can be illustrated as below; to simplify the presentation, we assume that the number of threads is enough for the number of nodes:

```
1  // convergence condition
2  while ( |π^(k+1) − π^(k)| > ε )
3  {
4     // Reset PageRank value to 0
5     for i = 0 to n - 1 in parallel
6     {
7           π_i^(k+1) = 0
8     }
```

```
9
10    dangling_value = 0
11    for i = 0 to n - 1 in parallel
12    {
13        if Vi is not a dangling node
14        {
15            for each e_{i,j} ∈ E
16            {
17                // atomicAdd
18                π_j^(k+1) = π_j^(k+1) + α × π_i^(k) / |P_i|
19            }
20        }
21        else // Vi is a dangling node
22        {
23            // atomicAdd
24            dangling_value =
                dangling_value + α × π_i^(k)
25        }
26    }
27
28    for i = 0 to n - 1 in parallel
29    {
30        π_j^(k+1) = π_j^(k+1) + (dangling_value + 1 - α) / n
31    }
32 }
```

Figure 4 Pseudo code of parallel algorithm on GPU

Before iteration, we check the condition for convergence (line 2). If this condition is not satisfied, the loop continues. The first step is reset all PageRank value of the previous step to 0 to prepare for the computation in this iteration. This can be accomplished in parallel by a trivial CUDA kernel (lines from 5 to 8). Then we go over each node, which is done in parallel by many threads (lines from 11 to 27). If the current node is not a dangling node, we update PageRank value of its destinations. This action is actually an **atomicAdd** operation preventing us from race condition (line 18). In the case when this node is a dangling node, according to the theory, it is assumed that this node links to every other nodes. We maintain a variable *dangling_value*, the value of which is calculated for each dangling node (line 24). Again this action is an atomic operation because more than one thread may access different dangling node at a same time. In the final step (lines from 28 to 31), all threads update in parallel the PageRank value of every node with the value of *dangling_value* (the second operand) and the constant $\frac{(1-\alpha)}{n}$ (the first operand in (3)).

3.2.1 Thread Collision

We can soon realize that calculating PageRank by this approach leads to the possibility of data collision. This happens when two different sources points to the same destination, then there is a chance two threads will access this destination at the same time. As can be seen from the example in the Figure 2, there is a collision between Thread 3 and Thread (N – 1) when these threads are executing their own source nodes, which happen to link to a same destination node. In the parallel environment, this may cause unexpected results without any protecting mechanism. Fortunately, GPU devices which support the CUDA have compute capability 1.1 and above provide *atomic operations*. An atomic function performs a read-modify-write atomic operation on one 32-bit or 64-bit word residing in global or shared memory. The operation is atomic in the sense that it is guaranteed to be performed without interference from other threads. In other words, no other thread can access this address until the operation is complete. In our situation, we use atomic function **atomicAdd()**, this operation reads a word at some place in global or shared memory, adds an integer to it, an writes the result back to the same address. This atomic function is applied in calculation the PageRank of destination; the access to the same destination from different threads is serialized and therefore the result is reliable. Because the probability that two random pages point to a same page is low, the collision during the atomic increment operation is the exception and the computation is still highly parallel.

3.2.2 The Condition for Convergence

In theory, the calculating loop finishes when the PageRank comes to stable values. Because it is very hard to achieve the exact PageRank, we can assume that the convergence condition is when the PageRank values of two consecutive iterations are not changed so much. The condition for convergence of PageRank scores can be expressed as:

$$\left|\pi^{(k+1)} - \pi^{(k)}\right| < \varepsilon$$

in which, $\left|\pi^{(k+1)} - \pi^{(k)}\right|$ is the distance of two PageRank vectors of k^{th} and $(k + 1)^{th}$ iteration, ε is the threshold for the convergence condition. Following the Chebyshev formula [12], the distance between two vectors is the greatest of their differences along any coordinate dimension, that means:

$$\left|\pi^{(k+1)} - \pi^{(k)}\right| = \max_i\left(\left|\pi_i^{(k+1)} - \pi_i^{(k)}\right|\right)$$

Thus, in essence, the Chebyshev distance calculation is finding the maximum value among $\left(\pi_i^{(k+1)} - \pi_i^{(k)}\right)$ values. This work has complexity O(n) if using a sequential algorithm that loop all elements of n-elements array, where n is the number of nodes. Here, we exploit GPU to find quickly the maximum value of an array by applying scan algorithm that presented in [13]. The idea of the algorithm is to build in parallel a balanced binary tree using bottom-up approach. The two nodes at the same level are compared with each other, the greater value will be used to make new root of the tree. The process repeated until only a single node, this node contains the maximum value of the element in the array.

The followings describe steps of finding max value using *scan*:

- At first, building an array containing the difference between corresponding elements of two arrays $\pi^{(k)}$ and $\pi^{(k+1)}$. This process is done in parallel on GPU.
- Dividing the array to multiple blocks; each block consists of 1024 elements (except the last block can be less). So we have $\left\lceil \frac{n}{1024} \right\rceil$ blocks.

- Applying algorithm *scan* for each block, we obtain an array of $\left\lceil \frac{n}{1024} \right\rceil$ elements containing the largest values in each block.
- We continue to divide array into 1024-elements blocks and apply *scan* algorithm on these blocks. This process is repeated until only one block and we find the largest element of that block, this is the largest element of the original array.

3.3 Implementing PageRank On Multi-GPUs

Based on the strategy of paralleling PageRank computation on graphics card as mentioned above, we need to divide data (the web graph) into GPUs to execute the algorithm on multi-GPUs system. In other words, we aim to distribute computing PageRank values on multiple GPUs for take advantage of GPUs computing power of the system as well as gain more speed up. Assuming that the experimental system equipped *m* GPUs, so we can divide the web graph to *m* parts: D_0, D_1, …, D_{m-1}, such that each D_i contains records of source-ids from $\left\lfloor \frac{i \times n}{m} \right\rfloor^{th}$ to the $\left\lfloor \frac{(i+1) \times n}{m} \right\rfloor^{th}$ in the web graph, and allot each of them to a GPU. Each GPU then compute in parallel local PageRank scores in part of Web Graph corresponding it, these PR values will be sent to CPU/host for synchronization in every iteration. Pseudo code describes tasks executed on multi-GPUs in shown in figure 5.

```
1   /*******************ASSUMPTION**************
    np -> the number of CPU processes, is equal
    to the number computing devices - m GPUs.
    π_i^k , π_i^{k+1} -> arrays containing PR values of n
    pages at the k^th and (k+1)^th iteration.
    D_i -> the webgraph chunk corresponding to
    GPU_i
    host_PR -> n-elements host array consist of
    PageRank value after mixing m arrays π_i^{k+1}
    ****************************************/
2   // CPU read the graph file and divide data
    Split the web graph into m parts: D_0,.. D_{m-1}
3   Initialize n-elements array host_PR
4   // np threads for controlling m GPUs
    for i = 0 to np - 1 in parallel
    {
5      copy D_i to GPU_i;
6      allocate π_i^k and π_i^{k+1} on GPU_i;
7      set π_i^k to 1/n;
8      while (max(threshold_i) > ε)
       {
9         set π_i^{k+1} to 0;
10        //Call PageRank computation kernel
          π_i^{k+1} = PRKernel(π_i^k, D_i);
11        copy π_i^{k+1} from GPU_i to host;
12        //merging results
          host_PR = ∑_i π_i^{k+1};
13        copy host_PR to π_i^{k+1};
14        //apply scan algorithm
          threshold_i = scan(π_i^{k+1} − π_i^k);
15        //update π_i^k for next iteration
          copy π_i^{k+1} to π_i^k
16     }
16  }
```

Figure 5 Computing PageRank on multiple GPUs

The whole program contains host code and device code. At first, host determines the number of GPUs and split data for GPUs appropriately. Each GPU then compute PageRank corresponding to its web graph chunk (line 10). In addition to the line 10 – performing a kernel call as described in the previous section, lines - 7, 9, 14 also executed in parallel on GPU by other kernels for gaining better performance. Because GPUs compute its PageRank locally, we need to synchronize computation results after each iteration. This is illustrated at line 12, in which the vector host_PR is sum of vectors π_i^{k+1} (i=0, ... , m) returned from GPUs. threshold values are then calculated based on new π_i^{k+1} (line 13-14) for determining the stop condition. In fact, each GPU only need scan 1/m of distance vector, for performance benefits again, which lead to threshold values on GPUs are different but the maximum of these values (line 8) ensure the correctness of convergence. Finally, at the line 15, each GPU make a copy from device to device to update values in π_i^k.

The above algorithm allocates on each GPU the following components: π^k, π^{k+1} and a part of the webgraph D_i which has size about 1/m size of original web graph. In addition, the scan kernel in our implementation also requires one temp n-elements array for finding the maximum value on GPU. Therefore, each GPU's device memory has to fetch about: $(n+n+\frac{1}{m}(Webgraph)+n)$ 4-bytes integer elements, where n is the number of pages, and m is the number of GPUs. As mentioned in section 2.2, the web graph has size of 2*V + E, here V is equal to n. So, the memory load on a GPU is:

$$L = 3n + \frac{1}{m}(2n+V) \quad (4)$$

(4) represents the constraints between web graph size and device memory, thus, as long as the GPU device memory can fit (4), *m* such GPUs can perform pagerank computation for the webgraph of *n* nodes and *V* edges.

4. EXPERIMENTAL RESULTS

4.1.1 Experimental environment and input data

The algorithms described in the previous two sections have been implemented and tested on a system consisting of:

- Processor: Intel Core 2 Quad Q8400 2.66 GHz
- Host Memory: 8 GB
- 2 dual graphics cards NVIDIA GeForce GTX 295 (4 GPUs)
- GPU DRAM: 1 GB
- OS: Linux Centos 5.3

Input Data Set: Our data are two web graphs derived from a crawl during January 2011 within Vietnamese web pages. The first data contains about 7 million web pages and 132 million hyperlinks. The second one is larger with 15 million pages and 200 million hyperlinks. We summarize statistic of our data sets in Table 1.

Table 1 Statistic of two experimental datasets

Data sets	#nodes	#links	#dangling nodes	Web Graph Size (MB)
Data 1	7,000,000	132,000,000	192,806	558
Data 2	15,000,000	200,000,000	885,045	~1000

We ran experiments on CPU and GPU with two data sets above and also tested with different values of parameter α to analyze the impact of this parameter on the PageRank calculation.

CPU version: There are two experimental programs: a sequential program on a single process and a multithreaded program running on 4 threads in order to exploit all the cores of the processor. The multithreaded algorithm on CPU is similar to the parallel algorithm devised for GPU. Instead of using **atomicAdd** as the atomic operation, on CPU we utilize the similar function **__sync_fetch_and_add**.

GPU version: The versions computing on GPU was also built in two programs for running on single-GPU and multiGPU.

4.1.2 Experimental Results

In the first experiments, we measure the running time (in seconds) of 4 programs (single CPU, multithreaded CPU and GPU) on the first data set (7 M pages, 132 M links) with different values α .

The parallel algorithms implemented on the above hardware environment shown that the speed of computing PageRank scores increase markedly when comparing to CPU-based versions. Table 2 compares the running time in seconds of PageRank computation on GPUs with that on CPU.

Table 2 Running time in seconds of experiments on the first data set (7 million pages, 132 million links) corresponding to different values of factor α

Damping factor α	CPU 1 thread	CPU 4 threads	1 GPU	4 GPUs
0.5	29.19165	15.76771	3.01419	1.60587
0.6	34.97308	18.90085	3.60987	2.05013
0.7	46.57511	25.19583	4.82156	2.62897
0.75	58.37653	31.62574	6.01332	3.45451
0.8	75.73101	40.95702	7.81736	4.60283
0.85	99.06957	53.49526	10.24953	5.59358
0.9	157.2996	85.28265	16.24351	8.29304
0.95	309.1649	167.2162	31.89439	16.17999

As we can see from the Table 2, the speed of computing PageRank scores on GPU increases from 10 to 20 times higher than that of on sequential CPU-based program (approximately 10 times on single GPU, and 20 times on four GPUs). Figure 6 also demonstrates this result, but more intuitive. This is true with most of alpha values.

By doing experiments with different values of the parameter α , we also can estimate the sensitivity of PageRank with respect to α . Although, the recommended value of α , which is also used in production by Google, is 0.85 [6], we tested with α ranged from 0.5 to 0.95 (Figure 7).

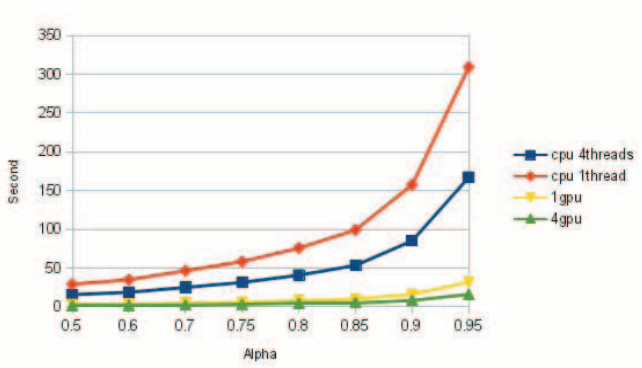


Figure 6 Running-time (lines) concluded from the experiments on the first data set (7M pages, 132 M links)

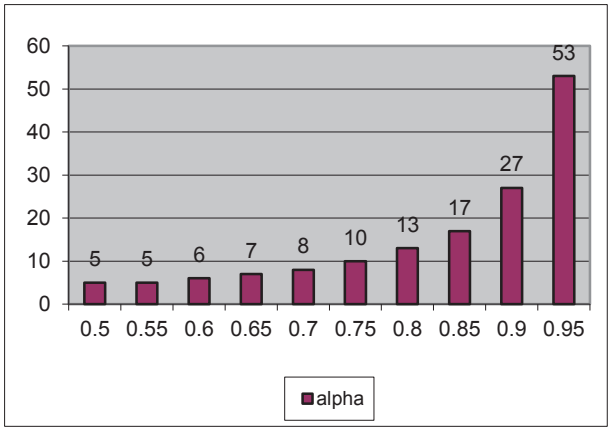


Figure 7 Number of iterations corresponding to different alpha values on the first data set (ε = 1e-5)

It can be seen from these experiments that when the parameter α grows, the running time on both CPU and GPU increases. It is proved in theory that this parameter controlled the asymptotic rate of convergence; as α comes near 1, the expected number of iterations required raises dramatically.

In the second experiment, we experiment on data set of 15 million pages and 200 million links. This data set is too large to fit the memory of 1 GPU GeForce GTX 295 which has about 900MB DRAM, but it is possible on multiGPU: applying formula (4) with m = 4GPUs, n = 15 M, V = 200M, we have:

$$L = 3* 15 + \frac{1}{4}(2*15 + 200) = 102.5$$

that means 1 GPU need 4*L = 410 MB memory for this computation. Indeed, this number is much smaller than global memory of GeForce GTX 295 (about 900MB). We provide experimental results of three programs (1CPU, 4CPU, 4 GPUs) for this data set in Table 3. These results are quite similar to the previous measurement.

Table 3 Running-time in seconds of experiments on the second data set (15 million pages, 200 million links)

Damping factor α	CPU 1 thread	CPU 4 threads	4 GPUs	Loop
0.5	89.49824	48.34198	4.75066	4
0.6	116.49927	62.96086	6.35976	5
0.7	156.96133	84.91168	8.62999	7
0.75	174.69417	94.64132	10.41547	8
0.8	200.79332	108.5935	12.09123	9
0.85	292.72221	158.06317	16.4891	13
0.9	424.21375	229.99457	22.31618	19
0.95	784.20441	424.14843	41.01353	35

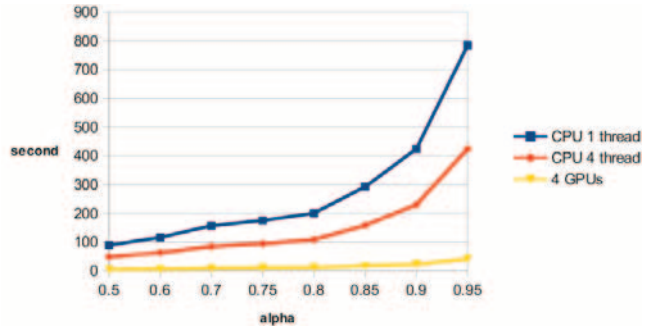


Figure 8 Running-time (lines) concluded from the experiments on the second data set (15 M pages, 200M links)

In the second experiment, the growth rate is not much compared to the first test. The performance comparison chart for this experiment is depicted in Figure 8, in which, the best speedup reaches about 19 times with the value α of 0.95 when compared between the 4 GPU and 1CPU version.

5. CONCLUSION

In this paper, we have proposed a parallel algorithm to accelerate the PageRank computation using multi-GPUs. We ran experiments on CPU and GPU with different data to test the speed up ability. The experimental results are quite promising when they show that the speed of computing PageRank scores by our approach gives a significantly greater performance than the CPU-based algorithm

Our experiments were implemented on not-too-large data sets. In practice, the input data of the PageRank problem is often the graph which contains hundreds of millions or billions of web pages. With such huge web graphs, we can improve the computing power of system for PageRank computation in two directions: firstly, we continue use multi-GPUs system equipped powerful GPUs with bigger memory such as Tesla GPUs which can provide us 4GB RAM memory, which helps resolve larger webgraphs as long as it satisfy the constrains condition from formula (4). Secondly, the applications of GPU cluster for implementing the proposed algorithms are an interesting direction should also be considered.

The quality of computational improvement on GPU opens other researches: applying GPU on query-dependent ranking methods, such as TrustRank or HITS, which are executed at query time and are required to give the results within a very short time period, from which demonstrate the ability to solve problems on web mining using GPGPU technology.

6. ACKNOWLEDGMENTS

We would like to thanks the High Performance Computing Center, Hanoi University of Science and Technology, Vietnam to let us perform many experiments on their BKLUSTER system.

7. REFERENCES

[1] S. Brin and L.Page. 1998. The anatomy of a large-scale hypertextual web search engine. In *Proceedings of the 7th WWW Conference*.

[2] A. Rungsawang and B. Manaskasemsak. 2004. Parallel PageRank Computation on a Gigabit PC Cluster. In *Proceedings of the 18th International Conference on Advance Information Networking and Application*.

[3] A. Rungsawang and B. Manaskasemsak. 2003. PageRank computation using PC cluster. In *Proceedings of the 10th European PVM/MPI User's Group Meeting*.

[4] A. Rungsawang and B. Manaskasemsak. 2004. An Efficient Partition-Based Parallel PageRank Algorithm. In *Proceedings of the 11th International Conference Parallel and Distributed Computing.*

[5] K. Sankaralingam, S. Sethumadhavan and J.C. Browne. 2003. Distributed PageRank for P2P system. In *Proceedings of the 11th IEEE HPD'03 Conference.*

[6] Amy N. Langville and Carl D. Meyer. 2006. *Google's PageRank and Beyond: The Science of Search Engine Rankings.* Princeton University Press, 41 William Street, Princeton, New Jersey, 2006, p. 31-46.

[7] Nathan Bell and Michael Garland. 2008. *Ecient Sparse Matrix-Vector Multiplication on CUDA*. NVIDIA Technical Report.

[8] Xintian Yang, Srinivasan Parthasarathy, P. Sadayappan. 2011. *Fast Sparse Matrix-Vector Multiplication on GPUs: Implications for Graph Mining*. Proceedings of the VLDB Endowment,Vol. 4, No. 4. Seattle, Washington.

[9] Praveen K., Vamshi Krishna K., Anil Sri Harsha B., S. Balasubramanian, P.K. Baruah. 2011. *Cost Efficient PageRank Computation using GPU*. IEEE International Conference on High Performance Computing (HiPC), Student Research Symposium

[10] Tianji WU, Bo WANG, Yi SHAN, Feng YAN, Yu WANG and Ningyi XU. 2010. *Efficient PageRank and SpMV Computation on AMD GPUs*. 39th International Conference on Parallel Processing, DOI 10.1109, p.81-89

[11] Ali Cevahir, Cevdet Aykanat, Ata Turk, B. Barla Cambazoglu,Akira Nukada and Satoshi Matsuoka. 2010. *Efficient PageRank on GPU Clusters*. IPSJ SIG Technical Report, Vol.2010-HPC-128.

[12] Chebyshev distance. http://en.wikipedia.org/wiki/ Chebyshev_distance

[13] M. Harris. 2007. *Parallel Prefix Sum (Scan) with CUDA*. NVIDIA Corporation.

[14] CUDA zone, http://www.NVIDIA.com/object/cuda_home_new.html

[15] NVIDIA, 2009 "NVIDIA CUDA Programming Guide 3.0".